Expressiveness of Deep Learning

Alexander Bentkamp

September 13, 2023

Abstract

Deep learning has had a profound impact on computer science in recent years, with applications to search engines, image recognition and language processing, bioinformatics, and more. Recently, Cohen et al. [2] provided theoretical evidence for the superiority of deep learning over shallow learning. For my master’s thesis [1], I formalized their mathematical proof using Isabelle/HOL. This formalization simplifies and generalizes the original proof, while working around the limitations of the Isabelle type system. To support the formalization, I developed reusable libraries of formalized mathematics, including results about the matrix rank, the Lebesgue measure, and multivariate polynomials, as well as a library for tensor analysis.

Contents

1 Tensor 2
2 Subtensors 8
3 Tensor Addition 11
4 Tensor Scalar Multiplication 17
5 Tensor Product 20
6 Unit Vectors as Tensors 27
7 Tensor CP-Rank 29
8 Tensor Matricization 33
9 CP-Rank and Matrix Rank 39
10 Matrix to Vector Conversion 42
11 Deep Learning Networks 43
12 Concrete Matrices

13 Missing Lemmas of Finite_Set

14 Deep Network Model

15 Polynomials representing the Deep Network Model

16 Alternative Lebesgue Measure Definition

17 Lebesgue Measure of Polynomial Zero Sets

18 Shallow Network Model

19 Fundamental Theorem of Network Capacity

1 Tensor

theory Tensor

imports Main

begin

typedef 'a tensor = {t::nat list × 'a list. length (snd t) = prod-list (fst t)}
by (simp add: Ex-list-of-length)

definition dims:'a tensor ⇒ nat list where
dims A = fst (Rep-tensor A)

definition vec:'a tensor ⇒ 'a list where
vec A = snd (Rep-tensor A)

definition tensor-from-vec::nat list ⇒ 'a list ⇒ 'a tensor where
  tensor-from-vec d v = Abs-tensor (d,v)

lemma assumes length v = prod-list d
shows dims-tensor[simp]: dims (tensor-from-vec d v) = d
and vec-tensor[simp]: vec (tensor-from-vec d v) = v
by (simp add: Abs-tensor-inverse assms dims-def tensor-from-vec-def vec-def)+

lemma tensor-from-vec-simp[simp]: tensor-from-vec (dims A) (vec A) = A
by (simp add: Rep-tensor-inverse Tensor.vec-def dims-def tensor-from-vec-def)

lemma length-vec: length (vec A) = prod-list (dims A)
by (metis (mono-tags, lifting) Rep-tensor Tensor.vec-def dims-def mem-Collect-eq)

lemma tensor-eql[intro]:
assumes \( \text{dims } A = \text{dims } B \) and \( \text{vec } A = \text{vec } B \)
shows \( A=B \)
by (metis assms tensor-from-vec-simp)

abbreviation \( \text{order} :: \langle a \text{ tensor} \Rightarrow \mathbb{N} \rangle \) where
\( \text{order } t = \text{length } (\text{dims } t) \)

inductive valid-index :: nat list \( \Rightarrow \) nat list \( \Rightarrow \) bool (infix \( < \) 50) where
\( \text{Nil} : [] < [] \mid \text{Cons} : is < ds \Rightarrow i<d \Rightarrow i#is < d#ds \)

inductive-cases valid-indexE [elim]: is < ds
inductive-cases valid-index-dimsE [elim]: is < dims A

lemma valid-index-length: is < ds \( \Rightarrow \) length is = length ds
by (induction rule: valid-index.induct; auto)

lemma valid-index-lt: is < ds \( \Rightarrow \) m<length ds \( \Rightarrow \) is!m < ds!m
proof (induction arbitrary:m rule: valid-index.induct)
case Nil
then show ?case by auto
next
case Cons
then show ?case by (metis gr0_conv_Suc length_Cons linorder_neqE_nat not_less_eq nth.Cons nth.Cons_Suc)
qed

lemma valid-index1:
assumes length is = length ds and \( \land m. m<length ds \Rightarrow is!m < ds!m \)
shows is < ds
using assms proof (induction is arbitrary:ds)
case Nil
then show ?case by (metis length_0_conv valid-index.simps)
next
case (Cons a is ds)
then obtain d ds' where ds = d # ds' by (metis length_Suc_conv)
then have is < ds' using Cons by (metis length_cons less_irrefl linorder_neqE_nat not_less_eq nth.Cons_Suc)
then show ?case using Cons.prems(2) \( \langle ds = d \# ds' \rangle \) valid-index.Cons by fastforce
qed

lemma valid-index-append:
assumes is1-valid:is1 < ds1 and is2-valid:is2 < ds2
shows is1 \& is2 < ds1 \& ds2
apply (rule valid-index1[of is1 \& is2 ds1 \& ds2])
unfolding nth_append
using valid-index-lt[OF is2-valid] valid-index-lt[OF is1-valid] valid-index-length[OF
lemma valid-index-list-all2-iff: is < ds ⟷ list-all2 (<) is ds
by (metis list-all2-cone-all-nth list-all2-nthD valid-indexI valid-index-length valid-index-lt)

definition fixed-length-sublist::'a list ⇒ nat ⇒ nat ⇒ 'a list where
fixed-length-sublist xs l i = (take l (drop (l*i) xs))

fun lookup-base::nat list ⇒ 'a list ⇒ nat list ⇒ 'a
where
lookup-base-Nil: lookup-base [] v [] = hd v |
lookup-base-Cons: lookup-base (d # ds) v (i # is) =
lookup-base ds (fixed-length-sublist v (prod-list ds) i) is

definition lookup::'a tensor ⇒ nat list ⇒ 'a
where
lookup A = lookup-base (dims A) (vec A)

fun tensor-vec-from-lookup::nat list ⇒ (nat list ⇒ 'a) ⇒ 'a list where
tensor-vec-from-lookup-Nil: tensor-vec-from-lookup [] e [] = [] |
tensor-vec-from-lookup-Cons: tensor-vec-from-lookup (d # ds) e (i # is) =
concat (map (λ i. tensor-vec-from-lookup ds (λ is. e (i # is))) [0..<d])

definition tensor-from-lookup::nat list ⇒ (nat list ⇒ 'a) ⇒ 'a tensor where
tensor-from-lookup ds e = tensor-from-vec ds (tensor-vec-from-lookup ds e)

lemma concat-parts-leq:
assumes a * d ≤ length v
shows concat (map (fixed-length-sublist v d) [0..<a]) = take (a*d) v
using assms proof (induction a)
  case 0
  then show ?case by simp
next
  case (Suc a)
  then have concat (map (fixed-length-sublist v d) [0..<a]) = take (a*d) v
  by auto
  then have concat (map (fixed-length-sublist v d) [0..<Suc a]) =
    take (a*d) v @ fixed-length-sublist v d a using fixed-length-sublist-def
  by auto
  then show ?case using Suc by (metis add.commute mult.commute mult-Suc take-add fixed-length-sublist-def)
qed

lemma concat-parts-eq:
assumes a * d = length v
shows concat (map (fixed-length-sublist v d) [0..<a]) = v
by (simp add: concat-parts-leq assms)

lemma tensor-lookup-base:
assumes length v = prod-list list ds
and $\forall is. \ is < ds \implies \text{lookup-base ds v is} = e \ is$

shows $\text{tensor-vec-from-lookup ds e} = v$

using assms proof (induction ds arbitrary:v e)
  case Nil
  then show ?case unfolding $\text{tensor-vec-from-lookup.simps}$
    by (metis One-nat-def Tensor.lookup-base-Nil length-0-conv length-Suc-conv list.sel(1) prod-list.Nil valid-index.Nil)

next
  case (Cons a ds)
  then have $a \ast \text{prod-list ds} = \text{length v}$ by auto
  \{ fix i assume $i < a$
      then have $\text{prod-list ds} \ast (i+1) \leq \text{length v}$ using $a \ast \text{prod-list ds} = \text{length v}$
      using discrete mult.commute mult-le-monot by metis
      have $\forall is'. \ is' < ds \implies e (i \# is') = \text{lookup-base ds} (\text{fixed-length-sublist v} (\text{prod-list ds}) i) is'$
      using $i < a$ by (metis Cons.prems(2) Tensor.lookup-base-Cons valid-index.simps)
      then have $\text{tensor-vec-from-lookup ds} (\lambda is'. \ e (i \# is')) = \text{fixed-length-sublist v} (\text{prod-list ds}) i$
        using Cons using $\text{prod-list ds} \ast (i + 1) \leq \text{length v}$ by (simp add: Cons.IH fixed-length-sublist-def)
  \}
  then show ?case unfolding $\text{tensor-vec-from-lookup-Cons.lookup-base-Cons}$
    using concat-parts-eq[OF $a \ast \text{prod-list ds} = \text{length v}$]
    atLeastLessThan-iff map-eq-conv set-upt Cons by (metis (no-types, lifting))

qed

lemma tensor-lookup:
assumes $\forall is. \ is < \text{dims A} \implies \text{lookup A is} = e \ is$
shows $\text{tensor-from-lookup} (\text{dims A}) e = A$

using tensor-lookup-base lookup-def length-vec tensor-from-lookup-def by (metis tensor-assms tensor-from-vec-simp)

lemma concat-equal-length:
assumes $\forall xs. \ xs \in \text{set xss} \implies \text{length xs} = l$
shows $\text{length} (\text{concat xss}) = \text{length xss} * l$

using assms by (induction xss;auto)

lemma concat-equal-length-map:
assumes $\forall i. \ i < a \implies \text{length} (f i) = d$
shows $\text{length} (\text{concat} (\text{map} (\lambda i. \ f i) [0..<a])) = a * d$

using assms by (induction a;auto)

lemma concat-parts:
assumes $\forall xs. \ xs \in \text{set xss} \implies \text{length xs} = d$ and $i < \text{length xss}$
shows $\text{fixed-length-sublist} (\text{concat xss}) d i = xss ! i$

using assms proof (induction xss arbitrary:i)
  case Nil
  then show ?case by simp
next
case (Cons xs xss)
then have length (concat xss) = length xss * d by (simp add: Cons.prems(1)
concat-equal-length)
show ?case
proof (cases i)
case 0
then have fixed-length-sublist (concat (xs # xss)) d i = xs
  unfolding fixed-length-sublist-def by (simp add: Cons.prems(1))
then show ?thesis using 0 by auto
next
case (Suc i')
then have fixed-length-sublist (concat xss) d i' = xss ! i' using Cons by auto
then show ?thesis unfolding fixed-length-sublist-def using Suc Cons.prems(1)
by auto
qed
qed

lemma concat-parts':
assumes \( \forall i. i<a \Rightarrow \text{length } (f \ i) = d \)
and \( i<a \)
shows fixed-length-sublist (concat (map (\lambda i. f \ i) [0..<a])) d i = f i
using assms proof (induction a)
case 0
then show ?case by simp
next
case (Suc a)
then have \( \forall i. i < a \Rightarrow \text{length } (f \ i) = d \) by auto
then have length (concat (map f [0..<a])) = a*d using concat-equal-length-map
by auto
show ?case
proof (cases i=a)
assume i=a
then have fixed-length-sublist (concat (map f [0..<Suc a])) d i = f a
  by (simp add: Suc.prems(1) \( \text{length } (concat (map f [0..<a])) = a * d \)
fixed-length-sublist-def)
then show ?case using \( i=a \) by auto
next
assume i\#a
then have fixed-length-sublist (concat (map f [0..<a])) d i = f i
  concat (map f [0..<Suc a]) = concat (map f [0..<a]) @ f a using Suc by auto
show ?case unfolding \( \text{concat } (map f [0..<Suc a]) = \text{concat } (map f [0..<a]) \)
@ f a
  unfolding fixed-length-sublist-def drop-append
  using length (concat (map f [0..<a])) = a * d  \( \text{fixed-length-sublist } (concat (map f [0..<a])) d i = f i \)
  using append-assoc append-eq-conv-conj append-take-drop-id assms(1) assms(2)
fixed-length-sublist-def
by metis
qed

lemma length-tensor-vec-from-lookup:
length (tensor-vec-from-lookup ds e) = prod-list ds
by (induction ds arbitrary:e; auto simp add: concat-equal-length-map)

lemma lookup-tensor-vec:
assumes is ds
shows lookup-base ds (tensor-vec-from-lookup ds e) is = e is
using assms proof (induction arbitrary:e rule:valid-index.induct)
  case Nil
  then show ?case by simp
next
  case (Cons is ds i d e)
  then show ?case unfolding tensor-vec-from-lookup-Cons lookup-base-Cons
  by (simp add: length-tensor-vec-from-lookup concat-parts[of d λ i. tensor-vec-from-lookup ds (λ is. e (i # is)) prod-list ds i] i < d)
qed

lemma lookup-tensor-from-lookup:
assume is ds
shows lookup (tensor-from-lookup ds e) is = e is
  unfolding lookup-def tensor-from-lookup-def
by (simp add: length-tensor-vec-from-lookup assms)

lemma dims-tensor-from-lookup:
dims (tensor-from-lookup ds e) = ds
unfolding tensor-from-lookup-def
by (simp add: length-tensor-vec-from-lookup)

lemma tensor-lookup-cong:
assumes tensor-from-lookup ds e1 = tensor-from-lookup ds e2
and is ds
shows e1 is = e2 is using assms lookup-tensor-from-lookup by metis

lemma tensor-from-lookup-eqI:
assumes /\ is. is<ds \= e1 is = e2 is
shows tensor-from-lookup ds e1 = tensor-from-lookup ds e2
by (metis assms lookup-tensor-from-lookup tensor-vec-from-lookup tensor-lookup-base)

lemma tensor-lookup-eqI:
assumes dims A = dims B and /\ is. is<(dims A) \= lookup A is = lookup B is
shows A = B by (metis assms(1) assms(2) tensor-lookup)
end
2 Subtensors

theory Tensor-Subtensor
imports Tensor
begin

definition subtensor:: ‘a tensor ⇒ nat ⇒ ‘a tensor where
subtensor A i = tensor-from-vec (tl (dims A)) (fixed-length-sublist (vec A) (prod-list (tl (dims A)))) i

definition subtensor-combine::nat list ⇒ ‘a tensor list ⇒ ‘a tensor where
subtensor-combine ds As = tensor-from-vec (length As # ds) (concat (map vec As))

lemma length-fixed-length-sublist[simp]:
  assumes (Suc i)∗l ≤ length xs
  shows length (fixed-length-sublist xs l i) = l
unfolding fixed-length-sublist-def
by (metis assms diff-add-inverse2 length-drop length-take min.absorb2 mult.commute mult-Suc take-drop)

lemma vec-subtensor[simp]:
  assumes dims A ≠ [] and i < hd (dims A)
  shows vec (subtensor A i) = fixed-length-sublist (vec A) (prod-list (tl (dims A))) i
by (metis (no-types, lifting) Suc-leI assms(1) assms(2) hd-Cons-tl length-fixed-length-sublist length-vec prod-list.Cons mult-le-mono1 subtensor-def vec-tensor)

lemma dims-subtensor[simp]:
  assumes dims A ≠ [] and i < hd (dims A)
  shows dims (subtensor A i) = tl (dims A)
using Suc-leI assms(1) assms(2) dims-tensor length-fixed-length-sublist length-vec list.collapse prod-list.Cons mult-le-mono1 subtensor-def
by metis

lemma subtensor-combine-subtensor[simp]:
  assumes dims A ≠ []
  shows subtensor-combine (tl (dims A)) (map (subtensor A) [0..<hd (dims A)]) = A
proof –
  have length-vec-A: hd (dims A) * prod-list (tl (dims A)) = length (Tensor.vec A)
      by (metis assms length-vec list.collapse prod-list.Cons)
  let ?subtensor-vec = fixed-length-sublist (vec A) (prod-list (tl (dims A)))
  { 
      fix i assume i < hd (dims A)
      then have (Suc i)∗(prod-list (tl (dims A))) ≤ length (vec A)
          by (metis Suc-leI length-vec-A mult-le-mono1)
      then have (vec o (λi. tensor-from-vec (tl (dims A)) (?subtensor-vec i))) i = ?subtensor-vec i
          by simp
  }
then have 1: map (Tensor.vec o (λi. tensor-from-vec (tl (dims A))) (?subtensor-vec i))) [0..<hd (dims A)] = map ?subtensor-vec [0..<hd (dims A)] by auto

then have subtensor-combine (tl (dims A)) (map (λi. subtensor A i) [0..<hd (dims A)]) = A unfolding subtensor-combine-def subtensor-def using concat-parts-eq[OF length-vec-A] by (auto simp add: 1 assms)

then show ?thesis by auto qed

lemma assumes ⌈A. A ∈ set As ⇀ dims A = ds ⌋
shows subtensor-combine-dims[simp]: dims (subtensor-combine ds As) = length As # ds (is ?D)
and subtensor-combine-vec[simp]: vec (subtensor-combine ds As) = concat (map vec As) (is ?V)
proof –
  have ⌈v. v ∈ set (map Tensor.vec As) ⇀ length v = prod-list ds using assms length-vec length-vec by fastforce
  then have length As * prod-list ds = length (concat (map Tensor.vec As)) using concat-equal-length
    by (metis length-map)
  then show ?D ?V unfolding subtensor-combine-def by simp+
qed

lemma subtensor-subtensor-combine:
assumes ⌈A. A ∈ set As ⇀ dims A = ds and i < length As ⌋
shows subtensor (subtensor-combine ds As) i = As ! i
proof –
  have fixed-length-sublist (concat (map vec As)) (prod-list ds) i = vec (As ! i)
    using concat-parts[of map vec As prod-list ds i] assms imageE length-map length-vec
    nth-map set-map in-set-conv-nth by fastforce
  then show ?thesis unfolding subtensor-def using subtensor-combine-dims subtensor-combine-vec
    by (metis assms list.sel(3) nth-mem tensor-from-vec-simp)
qed

lemma subtensor-induct[case-names order-0 order-step]:
assumes order-0: ⌈A. dims A = [] ⇀ P A ⌋
and order-step: ⌈A. dims A ≠ [] ⌋ ∀i. i < hd (dims A) ⇒ (P (subtensor A i)) ⇒ P A
shows P B
using assms proof (induction dims B arbitrary:B)
  case Nil
  then show ?case by auto
next
  case Cons
then show ?case by (metis dims-subtensor list.sel(3))
qed

lemma subtensor-combine-induct[case-names order-0 order-step]:
assumes order-0:∀ A. dims A = [] ⇒ P A
and order-step:∀ As ds. (∀ A. A∈ set As ⇒ P A) ⇒ (∀ A. A∈ set As ⇒ dims A = ds) ⇒ P (subtensor-combine ds As)
shows P A
proof (induction A rule: subtensor-induct)
case (order-0 A)
then show ?case by (simp add: assms(1))
next
case (order-step A)
have P (subtensor-combine (tl (dims A)) (map (subtensor A) [0..<hd (dims A)]))
  apply (rule assms(2))
  using atLeastLessThan-iff dims-subtensor imageE set-map set-upt order-step
by auto
then show ?case using subtensor-combine-subtensor[OF order-step.hyps] by metis
qed

lemma lookup-subtensor1 [simp]:
assumes i # is < dims A
shows lookup (subtensor A i) is = lookup A (i # is)
using assms
proof (induction A rule: subtensor-combine-induct)
case order-0
then show ?case by auto
next
case (order-step As ds)
have 0:subtensor (subtensor-combine ds As) i = As ! i
  by (metis list.discl list.sel(1) order-step.hyps order-step.prems subtensor-combine-dims subtensor-subtensor-combine valid-index-dimsE)
  have 1:dims (subtensor-combine ds As) = length As # ds
    using order-step subtensor-combine-def subtensor-combine-dims by force
  show ?case unfolding 0 lookup-def 1 unfolding lookup-base-Cons using order-step.prems
    using Tensor.lookup-base-Cons dims-subtensor lookup-def list.discl list.sel(1) list.sel(3) valid-index-dimsE vec-subtensor by (metis 0 1)
qed

lemma lookup-subtensor:
assumes is < dims A
shows lookup A is = hd (vec (λ i A. subtensor A i) is A))
using assms proof (induction is arbitrary: A)
case Nil
then show ?case by (metis Tensor.lookup-base-Nil lookup-def fold-simps(1) length-0-conv valid-index-length)
next
case (Cons a is A)
  then show ?case
    using dims-subtensor lookup-subtensor1 fold-simps(2) list.discI list.sel(1) list.sel(3)
    valid-indexE by (metis (no-types, lifting))
qed

lemma subtensor-eqI:
  assumes dims A ≠ []
  and dims-eq: dims A = dims B
  and ∃i. i < hd (dims A) ⊢ subtensor A i = subtensor B i
  shows A = B
proof –
  { fix is assume is < dims A
  2. then obtain i is' where is-Cons: is = i # is' using assms(1) by blast
  3. then have lookup A is = lookup B is
  4. using lookup-subtensor1 assms by (metis ‹is < dims A› is-Cons list.sel(1)
    valid-index-dimsE)
  }
  then show ?thesis using tensor-lookup-eqI[OF dims-eq] by auto
qed

end

3 Tensor Addition

theory Tensor-Plus
imports Tensor-Subtensor
begin

definition vec-plus a b = map (λ(x,y). plus x y) (zip a b)
definition plus-base::'a::semigroup-add tensor ⇒ 'a tensor ⇒ 'a tensor
  where plus-base A B = (tensor-from-vec (dims A) (vec-plus (vec A) (vec B)))

instantiation tensor:: (semigroup-add) plus
begin
  definition plus-def: A + B = (if (dims A = dims B)
  2. then plus-base A B
  3. else undefined)
  instance ..
end

lemma plus-dim1[simp]: dims A = dims B ⇒ dims (A + B) = dims A unfolding
  plus-def plus-base-def
  using dims-tensor length-vec length-map map-fst-zip vec-plus-def by (metis (full-types))
lemma plus-dim2[simp]: dims A = dims B ⇒ dims (A + B) = dims B using
lemma plus-base: \(\text{dims } A = \text{dims } B \implies A + B = \text{plus-base } A B\) unfolding plus-def by metis

lemma fixed-length-sublist-plus:
assumes length \(xs1 = c * l\) length \(xs2 = c * l < c\)
shows fixed-length-sublist (vec-plus \(xs1 \cdot xs2\)) \(l i\)
= vec-plus (fixed-length-sublist \(xs1 \cdot l i\)) (fixed-length-sublist \(xs2 \cdot l i\))
unfolding vec-plus-def fixed-length-sublist-def using drop-map drop-zip take-map take-zip by metis

lemma vec-plus[simp]:
assumes \(\text{dims } A = \text{dims } B\)
shows \(\text{vec } (A+B) = \text{vec-plus } (\text{vec } A) (\text{vec } B)\)
unfolding plus-def plus-base-def vec-plus-def by (auto; metis (no-types, lifting) length-map length-tensor-vec-from-lookup map-fst-zip take-zip)

lemma subtensor-plus:
fixes \(A::'a::semigroup-add\) tensor and \(B::'a::semigroup-add\) tensor
assumes \(i < \text{hd } (\text{dims } A)\)
and \(\text{dims } A = \text{dims } B\)
and \(\text{dims } A \neq []\)
shows subtensor \((A + B) \cdot i = \text{subtensor } A \cdot i + \text{subtensor } B \cdot i\)
proof –
have length \((\text{vec } A) = \text{hd } (\text{dims } A) * \text{prod-list } (\text{tl } (\text{dims } A))\)
length \((\text{Tensor.vec } B) = \text{hd } (\text{dims } A) * \text{prod-list } (\text{tl } (\text{dims } A))\)
using length-vec prod-list.Cons assms by (metis (no-types) list.exhaust-sel)+
then show \(?thesis\)
using Tensor-Plus.vec-plus assms fixed-length-sublist-plus vec-subtensor tensor-eqI
dims-subtensor plus-dim1 by fastforce
qed

lemma lookup-plus[simp]:
assumes \(\text{dims } A = \text{dims } B\)
and \(\text{is} < \text{dims } A\)
shows lookup \((A+B) \cdot is = \text{lookup } A \cdot is + \text{lookup } B \cdot is\)
using assms proof (induction \(A+B\) arbitrary; \(A B\) is rule: subtensor-induct)
case (order-0 \(A B\) is)
then have \(is = []\) by auto
have 1:\(\text{is} < \text{dims } A\) using order-0 ⟨\(\text{is} = []\)⟩ by auto
have 2:\(\text{is} < \text{dims } B\) using order-0 ⟨\(\text{is} = []\)⟩ by auto
have 3:\(\text{is} < \text{dims } (A + B)\) using order-0 ⟨\(\text{is} = []\)⟩ by auto
have length \((\text{vec } A) = 1\) length \((\text{vec } B) = 1\)
by (metis length-vec prod-list.Nil order-0.hyps order-0.prems(1) plus-dim1)+
then show \(?case\) unfolding lookup-subtensor[OF 1] lookup-subtensor[OF 2]
lookup-subtensor[OF 3] ⟨\(\text{is} = []\)⟩
fold-simps(1) vec-plus[OF order-0.prems(1)] unfolding vec-plus-def using or-
der-0.prems length-map
list.mapSel(1) list.size(3) map-fst-zip map-snd-zip order-0.hyps
zero-neq-one case-prod-unfold length-vec by metis

next
case (order-step A B is)
then obtain i is’ where is = i # is’ by auto
have 1:is < dims A using order-step by auto
have 2:is < dims B using order-step by auto
have 3:is < dims (A + B) using order-step by auto
have lookup (subtensor A i + subtensor B i) is’ = lookup (subtensor A i) is’ +
lookup (subtensor B i) is’
apply (rule order-step.hyps(2)[of i])
using ⟨is = i # is’, 3 hd-cone-nth length-greater-0-conv nth-Cons-0 or-
der-step.hyps(1) valid-index-lt
apply auto[1]
apply (metis 2 ⟨is = i # is’, list.inject list.sel(1) list.simps(3) order-step.prems(1)
subtensor-plus valid-index-cases)
using 1 ⟨is = i # is’, order-step.prems(1) plus-dim1 apply auto[1]
using 1 ⟨is = i # is’, plus-dim1 by auto
using order-step ⟨is = i # is’, plus-dim1 lookup-subtensor1 list.sel(1) subten-
sor-plus valid-index-dimsE by metis
qed

lemma plus-assoc:
assumes dimsA: dims A = ds and dimsB: dims B = ds and dimsC: dims C = ds
shows (A + B) + C = A + (B + C)
by (rule tensor-lookup-eqI; simp add: dimsA dimsB dimsC add.assoc)+

lemma tensor-comm[simp]:
fixes A::'a::ab-semigroup-add tensor
shows A + B = B + A
proof (cases dims A = dims B)
case True
then show ?thesis unfolding plus-def plus-base-def
by (metis lookup-plus plus-dim1 tensor-lookup-eqI vec-plus)
next
case False
then show ?thesis unfolding plus-def plus-base-def by simp
qed

definition vec0 n = replicate n 0

definition tensor0::nat list ⇒ 'a::zero tensor where
tensor0 d = tensor-from-vec d (vec0 (prod-list d))
lemma dims-tensor0[simp]: dims (tensor0 d) = d
and vec-tensor0[simp]: vec (tensor0 d) = vec0 (prod-list d)
  unfolding tensor0-def vec0-def by simp-all

lemma lookup-is-in-vec: is ⊨ (dims A) ⇒ lookup A is ∈ set (vec A)
proof (induction arbitrary: is rule:subtensor-induct)
case order-0
then show ?case unfolding lookup-def using lookup-base-Nil
by (metis length-0-conv length-vec list.sel(1) prod-list.Nil valid-index-length zero-neq-one)
next
case (order-step A is)
then obtain i is' where is = i # is' using valid-index-dimsE by blast
then have 1:is' ⊨ dims (subtensor A i) using 〈is = i # is'〉 dims-subtensor or-step.prems by auto
have 2:is' ⊨ dims (subtensor A i) using 〈is = i # is'〉 dims-subtensor order-step.prems by auto
have lookup A is ∈ set (Tensor.vec (subtensor A i))
  using order-step.IH [OF 1 2] lookup-subtensor1 〈is = i # is'〉 order-step.prems
by auto
then show ?case using vec-subtensor fixed-length-sublist-def by (metis 1 in-set-dropD in-set-takeD order-step.hyps)
qed

lemma lookup-tensor0:
assumes is ⊨ ds
shows lookup (tensor0 ds) is = 0
proof –
have lookup (tensor0 ds) is ∈ set (vec (tensor0 ds)) using lookup-is-in-vec assms
by (metis dims-tensor0)
moreover have set (vec (tensor0 ds)) ⊆ {0} unfolding vec-tensor0 vec0-def
by (metis in-set-replicate singleton-iff subsetI)
ultimately show 〈thesis by auto
qed

lemma
fixes A::′a::monoid-add tensor
shows tensor-add-0-right[simp]: A + tensor0 (dims A) = A
  unfolding plus-def plus-base-def dims-tensor0
apply (simp-all)
apply (rule tensor-lookup-eqI)
apply (metis (no-types, lifting) dims-tensor dims-tensor0 length-vec plus-dim2 vec-plus vec-tensor0)
by (metis add.right-neutral dims-tensor0 lookup-plus lookup-tensor0 plus-dim2 tensor-from-vec-simp vec-plus vec-tensor0)

lemma
fixes A::′a::monoid-add tensor
shows tensor-add-0-left[simp]: tensor0 (dims A) + A = A
  unfolding plus-def plus-base-def dims-tensor0
apply (simp-all)
apply (rule tensor-lookup-eqI)
apply (metis (no-types, lifting) dims-tensor dims-tensor0 length-vec plus-dim2 vec-plus vec-tensor0)
by (metis add.left-neutral dims-tensor0 lookup-plus lookup-tensor0 plus-dim2 tensor-from-vec-simp vec-plus vec-tensor0)

by (metis add.left-neutral dims-tensor0 lookup-plus lookup-tensor0 plus-dim2 tensor-from-vec-simp vec-plus vec-tensor0)

definition listsum::nat list ⇒ 'a::monoid-add tensor list ⇒ 'a tensor where
listsum ds As = foldr (+) As (tensor0 ds)

definition listsum'::'a::monoid-add tensor list ⇒ 'a tensor where
listsum' As = listsum (dims (hd As)) As

lemma listsum-Nil: listsum ds [] = tensor0 ds by (simp add: Tensor-Plus.listsum-def)

lemma listsum-one: listsum (dims A) [A] = A unfolding listsum-def by simp

lemma listsum-Cons: listsum ds (A # As) = A + listsum ds As unfolding listsum-def by auto

lemma listsum-dims:
assumes ∃A. A∈set As ⇒ dims A = ds
shows dims (listsum ds As) = ds
using assms proof (induction As)
case Nil
then show ?case by (metis dims-tensor0 listsum-Nil)
next
case (Cons A As)
then show ?case using listsum-Cons
by (metis list.set-intros(1) list.set-intros(2) plus-dim2)
qed

lemma subtensor0:
assumes ds ≠ [] and i<hd ds
shows subtensor (tensor0 ds) i = tensor0 (tl ds)
proof (rule tensor-lookup-eqI)
show 1: dims (subtensor (tensor0 ds) i) = dims (tensor0 (tl ds)) by (simp add: assms(1) assms(2))
fix is assume is ⊊ dims (subtensor (tensor0 ds) i)
then have i ≠ is ⊊ dims (tensor0 ds) using assms(1) assms(2) valid-index.Cons by fastforce
then show lookup (subtensor (tensor0 ds) i) is = lookup (tensor0 (tl ds)) is
using lookup-subtensor1 1 ⟨is ⊊ dims (subtensor (tensor0 ds) i)⟩ dims-tensor0 lookup-tensor0
bymetis
qed

lemma subtensor-listsum:
assumes $\bigwedge A. \ A \in \text{set } As \implies \text{dims } A = ds$
and $ds \neq []$ and $i < \text{hd } ds$
shows $\text{subtensor} \ (\text{listsum } ds \ As) \ i = \text{listsum} \ (\text{tl } ds) \ (\text{map } (\lambda A. \ \text{subtensor } A \ i) \ As)$
using assms proof (induction As)
case Nil
then show $?\text{case using } \text{lookup-tensor0} \ \text{assms}(2) \ \text{assms}(3) \ \text{subtensor0} \ \text{by } (\text{auto simp add: listsum-Nil})$
next
case $(\text{Cons } A \ As)$
then show $?\text{case by } (\text{simp add: listsum-Cons}; \ \text{metis subtensor-plus listsum-dims})$
qed

lemma listsum0:
assumes $\bigwedge A. \ A \in \text{set } As \implies A = \text{tensor0 } ds$
shows $\text{listsum } ds \ As = \text{tensor0 } ds$
using assms proof (induction As)
case Nil
show $?\text{case by } (\text{simp add: listsum-Nil})$
next
case Cons
then show $?\text{case using } \text{listsum-Cons}$
by $(\text{metis dims-tensor0 } \text{list.set-intros}(1) \ \text{set-subset-Cons} \ \text{subsetCE} \ \text{tensor-add-0-right})$
qed

lemma listsum-all-0-but-one:
assumes $\bigwedge i. \ i \neq j \implies i < \text{length } As \implies As!i = \text{tensor0 } ds$
and $\text{dims } (As!j) = ds$
and $j < \text{length } As$
shows $\text{listsum } ds \ As = As!j$
using assms proof (induction As arbitrary;j)
case Nil
then show $?\text{case by } \text{auto}$
next
case $(\text{Cons } A \ As \ j)$
then show $?\text{case}$
proof (cases j)
case 0
then have $\bigwedge i. \ i < \text{length } As \implies As ! i = \text{tensor0 } ds$ using Cons using Suc-less-eq length-Cons list.sel(3) nat.simps(3) nth-tl by fastforce
then have $\text{listsum } ds \ As = \text{tensor0 } ds$ using listsum0 by $(\text{metis in-set-conv-nth})$
then show $?\text{thesis by } (\text{metis 0 } \text{Cons.prems}(2) \ \text{listsum-Cons} \ \text{nth-Cons-0 tensor-add-0-right})$
next
case $(\text{Suc } j')$
then have $\text{listsum } ds \ As = As!j' \text{ by } (\text{metis (no-types, lifting) Cons.IH } \text{Cons.prems}(1) \ \text{Cons.prems}(2) \ \text{Cons.prems}(3) \ \text{Suc-less-eq length-Cons less-Suc-eq list.sel}(3) \ \text{not-less-eq nth-tl})$
then show $?\text{thesis by } (\text{metis } \text{Cons.prems}(1) \ \text{Cons.prems}(2) \ \text{Suc length-greater-0-conv}$
lemma lookup-listsum:
  assumes is ⊂ ds
  and ∩A. A ∈ set As ⇒ dims A = ds
  shows lookup (listsum ds As) is = (∑ A← As. lookup A is)
  using assms proof (induction As)
  case Nil
  then show ?case by (simp add: assms(1) listsum-Nil lookup-tensor0)
next
  case (Cons A As)
  then show ?case by (simp add: listsum-Cons listsum-dims)
qed

4 Tensor Scalar Multiplication

theory Tensor-Scalar-Mult
imports Tensor-Plus Tensor-Subtensor
begin

definition vec-smult::_:ring ⇒ 'a list ⇒ 'a list where
vec-smult α β = map ((*) α) β

lemma vec-smult0: vec-smult 0 as = vec0 (length as)
  by (induction as; auto simp add:vec0-def vec-smult-def)

lemma vec-smult-distr-right:
  shows vec-smult (α + β) as = vec-plus (vec-smult α as) (vec-smult β as)
  unfolding vec-smult-def vec-plus-def
  by (induction as; simp add: distrib-right)

lemma vec-smult-Cons:
  shows vec-smult α (a # as) = (α * a) # vec-smult α as by (simp add: vec-smult-def)

lemma vec-plus-Cons:
  shows vec-plus (a # as) (b # bs) = (a+b) # vec-plus as bs by (simp add: vec-plus-def)

lemma vec-smult-distr-left:
  assumes length as = length bs
  shows vec-smult α (vec-plus as bs) = vec-plus (vec-smult α as) (vec-smult α bs)
  using assms proof (induction as arbitrary:bs)
  case Nil
  then show ?case unfolding vec-smult-def vec-plus-def by simp
next
case (Cons a as)
then obtain b bs' where bs = b # bs' by (metis Suc-length-conv)
then have \(0 \cdot vec-smult \alpha (vec-plus (a # as') bs) = (\alpha \cdot (a + b)) \# vec-smult \alpha (vec-plus as' bs')\)
   unfolding vec-smult-def vec-plus-def using Cons.IH[of bs'] by simp
have length bs' = length as' using Cons.prems \(\langle bs = b \# bs' \rangle\) by auto
then show \(\text{thesis}\) unfolding \(0\) unfolding \(\langle bs = b \# bs' \rangle\) vec-smult-Cons vec-plus-Cons
   by (simp add: Cons.IH distrib-left)
qed

lemma length-vec-smult: length (vec-smult \alpha v) = length v unfolding vec-smult-def by simp

definition smult::\(\alpha::\text{ring} \Rightarrow \text{ a tensor} \Rightarrow \text{ a tensor} \) (infixl \(\cdot\) 70)
where smult \(\alpha\) A = (tensor-from-vec (dims A) (vec-smult \alpha (vec A)))

lemma tensor-smult0: fixes A::\(\alpha::\text{ring}\) tensor
shows \(0 \cdot A = \text{tensor0} (\text{dims A})\)
unfolding smult-def tensor0-def vec-smult-def using vec-smult-def vec-plus-def by (metis (no-types) vec-smult-def)

lemma dims-smult[simp]: dims (\alpha \cdot A) = dims A
and vec-smult[simp]: vec (\alpha \cdot A) = map ((\ast) \alpha) (vec A)
unfolding smult-def vec-smult-def by (simp add: length-vec vec-smult-def)

lemma tensor-smult-distr-right: (\alpha + \beta) \cdot A = \alpha \cdot A + \beta \cdot A
unfolding plus-def plus-base-def
by (auto; metis smult-def vec-smult-def vec-smult-distr-right)

lemma tensor-smult-distr-left: dims A = dims B \Rightarrow \alpha \cdot (A + B) = \alpha \cdot A + \alpha \cdot B
proof –
assume a1: dims A = dims B
then have f2: length (vec-plus (vec A) (vec B)) = length (vec A)
by (simp add: length-vec vec-plus-def)
have f3: dims (tensor-from-vec (dims B) (vec-smult \alpha (vec A))) = dims B
   using a1 by (simp add: length-vec vec-smult-def)
have f4: vec (\alpha \cdot A) = vec-smult \alpha (vec A)
   by (simp add: vec-smult-def)
have length (vec-smult \alpha (vec B)) = length (vec B)
   by (simp add: vec-smult-def)
then show \(\text{thesis}\)
   unfolding plus-def plus-base-def using f4 f3 f2 a1
by (simp add: length-vec smult-def vec-smult-distr-left)
qed
lemma smult-fixed-length-sublist:
assumes length xs = l * c i < c
shows fixed-length-sublist (vec-smult α xs) l i = vec-smult α (fixed-length-sublist xs l i)
unfolding fixed-length-sublist-def vec-smult-def by (simp add: drop-map take-map)

lemma smult-subtensor:
assumes dims A ≠ [] i < hd (dims A)
shows α · subtensor A i = subtensor (α · A) i
proof (rule tensor-eqI)
  show dims (α · subtensor A i) = dims (subtensor (α · A) i)
    using dims-smult dims-subtensor assms(1) assms(2) by simp
  show vec (α · subtensor A i) = vec (subtensor (α · A) i)
    unfolding vec-smult
    unfolding vec-subtensor[OF ‹dims A ≠ []› ‹i < hd (dims A)›]
    using vec-subtensor[of α · A i]
    by (simp add: assms(1) assms(2) drop-map fixed-length-sublist-def take-map)
qed

lemma lookup-smult:
assumes is ⊢ dims A
shows lookup (α · A) is = α * lookup A is
using assms proof (induction A arbitrary:is rule:subtensor-induct)
  case (order-0 A is)
    then have length (vec A) = 1 by (simp add: length-vec)
    then have hd (vec-smult α (vec A)) = α * hd (vec A) unfolding vec-smult-def
    by (metis list.map.sel(1) list.size(3) zero-neq-one)
    moreover have is = [] using order-0 by auto
    ultimately show ?case unfolding smult-def by (auto simp add: ‹length (Tensor.vec A) = 1› lookup-def length-vec-smult order-0.hyps)
next
  case (order-step A is)
    then obtain i is' where is = i ≠ is' by blast
    then have lookup (α · subtensor A i) is' = α * lookup (subtensor A i) is'
      by (metis (no-types, lifting) dims-subtensor list.sel(1) list.sel(3) order-step.IH order-step.hyps order-step.prems valid-index-dimsE)
    then show ?case using smult-subtensor ‹is = i ≠ is'› dims-smult lookup-subtensor1 list.sel(1) order-step.hyps order-step.prems valid-index-dimsE
      by metis
qed

lemma tensor-smult-assoc:
fixes A::'a::ring tensor
shows α · (β · A) = (α * β) · A
by (rule tensor-lookup-eqI, simp, metis lookup-smult dims-smult mult.assoc)
end
5 Tensor Product

theory Tensor-Product
imports Tensor-Scalar-Mult Tensor-Subtensor
begin

instantiation tensor :: (ring) semigroup-mult
begin

definition tensor-prod-def: A * B = tensor-from-vec (dims A @ dims B) (concat (map (λa. vec-smult a (vec B)) (vec A)))

abbreviation tensor-prod-otimes :: 'a tensor ⇒ 'a tensor ⇒ 'a tensor (infixl ⊗)
where A ⊗ B ≡ A * B

lemma vec-tensor-prod[simp]: vec (A ⊗ B) = concat (map (λa. vec-smult a (vec B)) (vec A)) (is ?V)
and dims-tensor-prod[simp]: dims (A ⊗ B) = dims A @ dims B (is ?D)
proof –
  have length (concat (map (λa. vec-smult a (vec B)) (vec A))) = prod-list (dims A @ dims B)
  proof –
    have ∀xs. xs ∈ set (map (λa. vec-smult a (vec B)) (vec A)) ⇒ length xs = length (vec B)
    using length-vec-smult by force
    then show ?thesis using concat-equal-length by (metis length-map length-vec prod-list.append)
  qed
  then show ?V ?D by (simp add: tensor-prod-def)+
  qed

lemma tensorprod-subtensor-base:
shows concat (map f (concat xss)) = concat (map (λxs. concat (map f xs)) xss)
by (induction xss; auto)

lemma subtensor-combine-tensor-prod:
assumes ∀A. A ∈ set As ⇒ dims A = ds
shows subtensor-combine ds As ⊗ B = subtensor-combine (ds @ dims B) (map (λA. A ⊗ B) As)
proof –
  let ?f = λa. vec-smult a (Tensor.vec B)
  let ?xss = map Tensor.vec As
  have 1:prod-list (length As ≠ ds) = length (concat ?xss) by (metis assms length-vec subtensor-combine-dims subtensor-combine-vec)
  have 2:∀A. A ∈ set As ⇒ prod-list (dims A @ dims B) = length (concat (map ?f (Tensor.vec A)))
    by (metis dims-tensor-prod length-vec vec-tensor-prod)
  have 3: length As ≠ ds @ dims B = (length (map (λA. tensor-from-vec (dims
A @ dims B
(concat (map (λa. vec-smult a (vec B)) (vec A))) As) # ds @ dims B) by simp

have 4:(concat (map (λxs. concat (map (λa. vec-smult a (vec B)) xs)) (map vec As)))
= (concat (map vec (map (λA. tensor-from-vec (dims A @ dims B) (concat
(map (λa. vec-smult a (vec B)) (vec A)))) As)))

unfolding map-map[unfolded comp-def] using vec-tensor by (metis (no-types,
lifting) 2 map-eq-cone)

have subtensor-combine ds As ⊗ B = tensor-from-vec (length As # ds @ dims
B) (concat (map (λf (concat (?fss)))))

unfolding subtensor-combine-def tensor-prod-def using 1 by auto

also have ... = tensor-from-vec (length As # ds @ dims B) (concat (map (λxs.
concat (map (λf xs)) ?xss))


also have ... = subtensor-combine (ds @ dims B) (map (λA. A ⊗ B) As)

unfolding subtensor-combine-def tensor-prod-def using 3 4 by metis

finally show ?thesis by metis

qed

lemma subtensor-tensor-prod:

assumes dims A ≠ [] and i < hd (dims A)

shows subtensor (A ⊗ B) i = subtensor A i ⊗ B

using assms proof (induction A rule:subtensor-combine-induct)

case order-0
then show ?case by auto

next

case (order-step As ds)

have 1:i < length (map (λA. A ⊗ B) As) using order-step by (simp add:
order-step.hyps order-step.prems(I))

have 2:(\A. A ∈ set (map (λA. A ⊗ B) As) → dims A = ds @ dims B)

using order-step by auto

have subtensor (subtensor-combine ds As ⊗ B) i = subtensor (subtensor-combine
(ds @ dims B) (map (λA. A ⊗ B) As)) i

using subtensor-combine-tensor-prod order-step by metis

also have ... = As ! i ⊗ B

using order-step subtensor-subtensor-def[of (map (λA. A ⊗ B) As) ds
@ dims B i] 1 2 by auto

also have ... = subtensor (subtensor-combine ds As) i ⊗ B

by (metis 1 length-map order-step.hyps subtensor-subtensor-combine)

finally show ?case by auto

qed

lemma lookup-tensor-prod[simp]:

assumes is1-valid:is1 < dims A and is2-valid:is2 < dims B

shows lookup (A ⊗ B) (is1 @ is2) = lookup A is1 * lookup B is2

using assms proof (induction A arbitrary:is1 rule:subtensor-induct)

case (order-0 A is1)
then obtain $a$ where vec $A = [a]
\text{using Suc-length-cone Tensor.tensor-vec-from-lookup Nil length-0-conv length-tensor-vec-from-lookup length-vec by metis}
$
\text{then have } A \otimes B = a \cdot B \text{ unfolding tensor-prod-def smult-def using order-0 by simp}
$
\text{moreover have lookup } A [[]] = a \text{ by (simp add: Tensor.vec A = [a] lookup-def order-0.hyps)}
\text{ultimately have lookup } (A \otimes B) ([is2]) = a \ast lookup B is2 \text{ by (simp add: lookup-smult is2-valid)}
\text{then show ?case using lookup A [[]] = a null-rec (1) order-0.hyps order-0.prems(1) by auto}
$
\text{next}
\text{case (order-step A is1)}
\text{then obtain } i is1' \text{ where } i \# is1' = is1 \text{ by blast}
\text{have lookup } (\text{subtensor } A i \otimes B) (is1' @ is2) = \text{lookup } (\text{subtensor } A i) \text{ is1' } \text{lookup } B is2 \text{ using order-step by (metis \langle i \# is1' = is1 \rangle \text{ dims-subtensor list.sel(1) list.sel(3) valid-index-dimsE)}}
\text{then show lookup } (A \otimes B) (is1 @ is2) = \text{lookup } A is1 \ast \text{lookup } B is2 \text{ using lookup-subtensor1[of i is1' A] lookup-subtensor1[of i is1' @ is2 A \otimes B] subtensor-tensor-prod[of A i B] Cons-eq-appendI \langle i \# is1' = is1 \rangle \text{ dims-tensor-prod is2-valid list.sel(1) order-step.hyps order-step.prems(1) valid-index-append valid-index-dimsE by metis qed}
\text{lemma valid-index-split:}
\text{assumes is } \triangleleft ds1 \triangleleft right ds2
\text{obtains is1 is2 where is1 @ is2 = is is1 } \triangleleft \text{ ds1 is2 } \triangleleft \text{ ds2}
\text{proof}
\text{assume a: } \bigwedge \text{is1 is2. is1 @ is2 = is is1 } \triangleleft \text{ ds1 is2 } \triangleleft \text{ ds2 } \Longrightarrow \text{ thesis}
\text{have length-is:length is } = \text{ length ds1 } + \text{ length ds2 using valid-index-length by auto}
\text{using assms by auto}
\text{show take } (\text{length ds1}) \text{ is } \triangleleft \text{ ds1}
\text{apply (rule valid-indexI)}
\text{using valid-index-length using assms apply auto[1]}
\text{by (metis add-leD1 assms length-append not-less nth-append nth-take valid-index-lt)}
\text{show drop } (\text{length ds1}) \text{ is } \triangleleft \text{ ds2}
\text{apply (rule valid-indexI)}
\text{using valid-index-length using assms apply auto[1]}
\text{using nth-drop[of length ds1 is] valid-index-lt[of assms(1)] nth-append[of ds1 ds2] length-is by (metis length-append nat-add-left-cancel-less nat-le-iff-add nth-append-length-plus)}
\text{show take } (\text{length ds1}) \text{ is } \triangleright \text{ drop } (\text{length ds1}) \text{ is } = \text{ is using length-is by auto qed}
\text{instance proof}
\text{fix } A B C :: 'a::ring tensor
\text{show } (A \otimes B) \otimes C = A \otimes (B \otimes C)
\text{proof (rule tensor-lookup-eqI, simp)
fix is assume is < dims \((A \otimes B) \otimes C\) 

obtain is1 is23 where is1 < dims A is23 < dims \((B \otimes C)\) is1 @ is23 = is

by (metis (mono-tags, lifting) is < dims \((A \otimes B) \otimes C\)) Tensor-Product.dims-tensor-prod append-assoc valid-index-split)

obtain is2 is3 where is2 < dims B is3 < dims C is2 @ is3 = is23

by (metis is23 < dims (local.tensor-prod-otimes B C), dims-tensor-prod valid-index-split)

define is12 where is12 = is1 @ is2

have is12 < dims \((A \otimes B)\) by (simp add: is1 < dims A is2 < dims B)

have is12 @ is3 = is by (simp add: is1 @ is23 = is) is2 @ is3 = is23 is12-def)

show lookup \(((A \otimes B) \otimes C)\) is = lookup \((A \otimes (B \otimes C))\)

unfolding lookup-tensor-prod[OF is1 < dims A is23 < dims \((B \otimes C)\)], unfolding is1 @ is23 = is]

lookup-tensor-prod[OF is12 < dims \((A \otimes B)\) is3 < dims C], unfolded is12 @ is3 = is]

using (is1 < dims A) (is2 @ is3 = is23) is2 < dims B) is3 < dims C is12-def mult.assoc by fastforce

qed

end

lemma tensor-prod-distr-left:

assumes dims A = dims B

shows \((A + B) \otimes C = (A \otimes C) + (B \otimes C)\)

proof –

have \(\forall is. is < dims A @ dims C \Longrightarrow lookup \(((A + B) \otimes C)\) is = lookup \((A \otimes C) + (B \otimes C)\) is

proof –

fix is assume is < dims A @ dims C

obtain is1 is2 where is = is1 @ is2 is1 < dims A is2 < dims C using valid-index-split using is < dims A @ dims C, by blast

then show lookup \(((A + B) \otimes C)\) is = lookup \(((A \otimes C) + (B \otimes C))\) is

using lookup-plus

(is1 < dims A) (is2 < dims C) assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distrib(2) valid-index-append

by fastforce

qed

moreover have tensor-from-lookup \((\text{dims } A @ \text{dims } C)\) (lookup \(((A + B) \otimes C)\)) = \((A + B) \otimes C\)

by (metis (no-types, lifting) assms plus-dim1 dims-tensor-prod tensor-lookup) + ultimately show \(\forall is. is < \text{dims } A @ \text{dims } C \Longrightarrow lookup \(((A + B) \otimes C)\) is = lookup \((A \otimes C + (B \otimes C))\) is)

qed

23
lemma tensor-prod-distr-right:
assumes \( \text{dems } A = \text{dems } B \)
shows \( C \otimes (A + B) = (C \otimes A) + (C \otimes B) \)
proof
  have \( \forall \text{is}. \text{is } \lessdot \text{dems } C \otimes \text{dems } A \implies \text{lookup } (C \otimes (A + B)) \text{is } \text{lookup } (C \otimes A + C \otimes B) \) is
  proof
    fix \text{is} assume \text{is } \lessdot \text{dems } C \otimes \text{dems } A
    obtain \text{is1} \text{is2} where \text{is } = \text{is1 } \lessdot \text{is2 } \lessdot \text{dems } C \otimes \text{is2 } \lessdot \text{dems } A \text{ using valid-index-split using } \langle \text{is} \lessdot \text{dems } C \otimes \text{dems } A \rangle \text{ by blast}
    then show \text{lookup } (C \otimes (A + B)) \text{is } = \text{lookup } ((C \otimes A) + (C \otimes B)) \text{is}
      using lookup-plus using \( \langle \text{is2 } \lessdot \text{dems } A \rangle \langle \text{is1 } \lessdot \text{dems } C \rangle \text{ asms plus-dim1 dems-tensor-prod lookup-tensor-prod ring-class ring-distrib(1) valid-index-append } 
      \text{ by fastforce}
    qed
  moreover have \text{tensor-from-lookup } (\text{dems } C \otimes \text{dems } A) (\text{lookup } (C \otimes (A + B)))
    = C \otimes (A + B)
    \text{tensor-from-lookup } (\text{dems } C \otimes \text{dems } A) (\text{lookup } ((C \otimes A) + (C \otimes B)))
    = (C \otimes A + (C \otimes B))
    \text{by (metis (no-types, lifting) asms plus-dim1 dems-tensor-prod tensor-lookup)+}
    ultimately show \( \exists \text{thesis using } \text{tensor-from-lookup-eqI} \)
      by (metis \( \langle \forall \text{is}. \text{is } \lessdot \text{dems } C \otimes \text{dems } A \implies \text{lookup } (C \otimes (A + B)) \text{is } = \text{lookup } (C \otimes A + C \otimes B) \text{is} \rangle \))
  qed

instantiation tensor :: (ring-1) monoid_mult
begin

definition tensor-one-def\( : = \text{tensor-from-vec } [] [1] \)

lemma tensor-one-from-lookup\( : = \text{tensor-from-lookup } [] (\lambda - 1) \)
unfolding tensor-one-def by (rule tensor-eqI; simp-all add: tensor-from-lookup-def )

instance proof
  fix A\( :\)':a:ring-1 tensor
  show \( A \cdot 1 = A \text{ unfolding tensor-one-from-lookup} \)
    by (rule tensor-lookup-eqI; metis lookup-tensor-prod[of - A [] tensor-from-lookup \( \langle \lambda - 1 \rangle \)])
next
  fix A\( :\)':a:ring-1 tensor
  show \( 1 \cdot A = A \text{ unfolding tensor-one-from-lookup} \)
    by (rule tensor-lookup-eqI; metis lookup-tensor-prod[of [] tensor-from-lookup \( \langle \lambda - 1 \rangle - A \)])
length-tensor-vec-from-lookup mult.left-neutral tensor-from-lookup-def)

qed
end

lemma order-tensor-one: order 1 = 0 unfolding tensor-one-def by simp

lemma smult-prod-extract1:
fixes a :: ′a::comm-ring-1
shows a · (A ⊗ B) = (a · A) ⊗ B
proof (rule tensor-lookup-eqI)
  show dims (a · (A ⊗ B)) = dims ((a · A) ⊗ B) by simp
  fix is assume is < dims (a · (A ⊗ B))
  then have is < dims (A ⊗ B) by auto
  then obtain is1 is2 where is1 < dims A is2 < dims B is = is1 @ is2 by (metis dims-tensor-prod valid-index-split)
  then have is1 < dims (a · A) by auto
  show lookup (a · (A ⊗ B)) is = lookup (a · A ⊗ B) is
    using lookup-tensor-prod[OF is1 < dims A; is2 < dims B] lookup-tensor-prod[OF is1 < dims (a · A) ; is2 < dims B]
    lookup-smult[OF is < dims (A ⊗ B);] lookup-smult[OF is1 < dims A;] is = is1 @ is2 by simp
qed

lemma smult-prod-extract2:
fixes a :: ′a::comm-ring-1
shows a · (A ⊗ B) = A ⊗ (a · B)
proof (rule tensor-lookup-eqI)
  show dims (a · (A ⊗ B)) = dims (A ⊗ (a · B)) by simp
  fix is assume is < dims (a · (A ⊗ B))
  then have is < dims (A ⊗ B) by auto
  then obtain is1 is2 where is1 < dims A is2 < dims B is = is1 @ is2 by (metis dims-tensor-prod valid-index-split)
  then have is2 < dims (a · B) by auto
  show lookup (a · (A ⊗ B)) is = lookup (A ⊗ (a · B)) is
    using lookup-tensor-prod[OF is1 < dims A; is2 < dims B] lookup-tensor-prod[OF is1 < dims (a · A) ; is2 < dims B]
    lookup-smult[OF is < dims (A ⊗ B);] lookup-smult[OF is2 < dims B] is = is1 @ is2 by simp
qed

lemma order-0-multiple-of-one:
assumes order A = 0
obtains a where A = a · 1
proof
  assume (\A a. A = a · 1 \implies thesis)
  have length (vec A) = 1 using assms by (simp add:length-vec)
  then obtain a where vec A = [a] by (metis One-nat-def Suc-length-conv length-0-conv)
moreover have vec \((a \cdot 1) = [a]\) unfolding smult-def tensor-one-def by (simp add: vec-smult-def)

ultimately have \(A = a \cdot 1\) using tensor-eqI by (metis assms dims-smult length-\(\emptyset\)-cone order-tensor-one)

then show \(A = \text{hd} (\text{vec} A) \cdot 1\) using \(\text{vec} A = [a]\) by auto
qed

lemma smult-1:
fixes \(\cdot\) :: 'a::{ring_1, tensor}
shows \(A = 1 \cdot A\) unfolding smult-def tensor-one-def
apply (rule tensor-eqI)
apply (simp add: length-vec length-vec-smult)
by (metis dims-tensor length-vec length-vec-smult lookup-smult mult-zero-neutral)
lmna tensor0-prod-right[simp]: \(A \otimes \text{tensor0} ds = \text{tensor0} (\text{dims} A @ ds)\)
proof (rule tensor-lookup-eqI, simp)
fix \(is\) assume \(is < \text{dims} (A \otimes \text{tensor0} ds)\)
then obtain \(is1 is2\) where \(is1 < \text{dims} A\) \(is2 < \text{dims} (\text{tensor0} ds)\) \(is = is1 @ is2\)
by (metis dims-tensor0 dims-tensor-prod valid-index-split)
then show \(\text{lookup} (A \otimes \text{tensor0} ds) is = \text{lookup} (\text{tensor0} (\text{dims} A @ ds)) is\)
by (metis (no-types, lifting) \(is < \text{dims} (A \otimes \text{tensor0} ds)\) dims-tensor0 dims-tensor-prod
lookup-tensor0 lookup-tensor-prod mult-zero-right)
qed

lemma tensor0-prod-left[simp]: \(\text{tensor0} ds \otimes A = \text{tensor0} (ds @ \text{dims} A)\)
proof (rule tensor-lookup-eqI,simp)
fix \(is\) assume \(is < \text{dims} (\text{tensor0} ds \otimes A)\)
then obtain \(is1 is2\) where \(is1 < \text{dims} (\text{tensor0} ds)\) \(is2 < \text{dims} A\) \(is = is1 @ is2\)
by (metis dims-tensor0 dims-tensor-prod valid-index-split)
then show \(\text{lookup} (\text{tensor0} ds \otimes A) is = \text{lookup} (\text{tensor0} (ds @ \text{dims} A)) is\)
by (metis (no-types, lifting) \(is < \text{dims} (\text{tensor0} ds \otimes A)\) dims-tensor0 dims-tensor-prod
lookup-tensor0 lookup-tensor-prod mult-zero-left)
qed

lemma subtensor-prod-with-vec:
assumes \(\text{order} A = 1\) \(i < \text{hd} (\text{dims} A)\)
shows \(\text{subtensor} (A \otimes B) i = \text{lookup} A [i] \cdot B\)
proof (rule tensor-lookup-eqI)
  have \(\text{dims} (A \otimes B) \neq []\) using assms(1) by auto
  have \(\text{hd} (\text{dims} A) = \text{hd} (\text{dims} (A \otimes B))\)
    by (metis One_nat_def Suc-length-\(\emptyset\)-cone append-Cons assms(1) dims-tensor-prod
list.sel(1))
  show \(\text{dims} (\text{subtensor} (A \otimes B) i) = \text{dims} (\text{lookup} A [i] \cdot B)\)
    unfolding \(\text{dims-smult} \text{dims-subtensor}(\text{OF} \cdot \text{dims} (A \otimes B) \neq []\), \(i < \text{hd} (\text{dims} A))\)

\[\text{unfolded} \langle \text{hd} (\text{dims} A) = \text{hd} (\text{dims} (A \otimes B))\rangle i\]
by (metis One-nat-def Suc-length-conv append.simps(2) append-self-conv2 assms(1)
dims-tensor-prod length-0-conv list.sel(3))

next
fix is assume is < dims (subtensor (A ⊗ B) i)
have dims (A ⊗ B) ≠ [] using assms(1) by auto
have hd (dims A) = hd (dims (A ⊗ B))
  by (metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod list.sel(3))

have is < dims B
  using ⟨is < dims (subtensor (A ⊗ B) i)⟩[unfolded dims-subtensor[OF ⟨dims (A ⊗ B) ≠ []⟩; i < hd ⟨dims A⟩⟩[unfolded ⟨hd (dims A) = hd (dims (A ⊗ B))⟩]]]
  by (metis One-nat-def Suc-length-conv append-self-conv2 assms(1) dims-tensor-prod list.sel(3) il-append2)

have [i] < dims A using assms by (metis One-nat-def Suc-length-conv length-0-conv list.sel(1) valid-index Nil valid-index.simps)

then have i ≠ is < dims (A ⊗ B) using ⟨is < dims (subtensor (A ⊗ B) i)⟩
dims-subtensor valid-index.Cons by auto

then show lookup (subtensor (A ⊗ B) i) is = lookup (lookup A [i] · B) is
unfolding lookup-subtensor1[OF ⟨i ≠ is < dims (A ⊗ B)⟩]
  using lookup-tensor-prod[OF ⟨[i] < dims A; is < dims B⟩] lookup-smult
  ⟨is < dims B⟩ using append-Cons by fastforce

qed

end

6 Unit Vectors as Tensors

theory Tensor-Unit-Vec
imports Tensor-Product
begin

definition unit-vec :: nat ⇒ nat ⇒ 'a::ring-1 tensor
where unit-vec n i = tensor-from-lookup [n] (λx. if x=int[i] then 1 else 0)

lemma dims-unit-vec: dims (unit-vec n i) = [n] unfolding unit-vec-def by (simp add: tensor-from-lookup-def)

lemma lookup-unit-vec:
assumes j<n
shows lookup (unit-vec n i) [j] = (if i=j then 1 else 0)
proof –
  have [j] < [n] by (simp add: assms valid-index.Cons valid-index.Nil)
  then have lookup (unit-vec n i) [j] = (λx. if x=int[i] then 1 else 0) [j]
    by (simp add: lookup-tensor-prod-with-unit-vec unit-vec-def)
  then show ?thesis by auto
qed

lemma subtensor-prod-with-unit-vec:
fixes A::'a::ring-1 tensor
assumes $j < n$
shows $\text{subtensor} \ (\text{unit-vec} \ n \ i \otimes A) \ j = (\text{if } i = j \ \text{then } A \ \text{else} \ (\text{tensor0} \ (\text{dims} \ A)))$
proof
  have $0 \cdot \text{lookup} \ (\text{unit-vec} \ n \ i) \ [j] = (\text{if } i = j \ \text{then } 1 \ \text{else } 0)$ unfolding \text{unit-vec-def}
  by (simp add: assms lookup-tensor-from-lookup valid-index.\text{Cons valid-index.\Nil})
  have $1 \cdot \text{order} \ (\text{unit-vec} \ n \ i) = 1$ unfolding \text{unit-vec-def}
  by (simp add: tensor-from-lookup-def)
  from assms have $2 \cdot j < \text{hd} \ (\text{dims-tensor-from-lookup} \ [n] \ (\lambda x. \text{if } x = [i] \ \text{then } 1 \ \text{else } 0))$
    by (simp add: \text{dims-tensor-from-lookup})
  show $\forall j. \ j < \text{hd} \ (\text{dims} \ A) \implies \text{subtensor} \ ?LS \ j = \text{subtensor} \ A \ j$
  proof
    fix $j$
    assume $j < \text{hd} \ (\text{dims} \ A)$
    have $1 \cdot \text{subtensor} \ ?LS \ j = \text{listsun} \ (\text{tl} \ (\text{dims} \ A)) \ (\text{map} \ (\lambda A. \ \text{subtensor} \ A \ j) \ (\text{map} \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)]))$
      using \text{subtensor-listsun}[\text{of} \ (\text{map} \ (\lambda i. \ ?f \ i) \ [0..<\text{hd} \ (\text{dims} \ A)]) \ \text{dims} \ A, \ \text{OF correct-dims asms} \ j < \text{hd} \ (\text{dims} \ A)]$
      by linarith
    also have $\ldots = \text{listsun} \ (\text{tl} \ (\text{dims} \ A)) \ (\text{map} \ (\lambda i. \ \text{subtensor} \ (\text{?f} \ i) \ [0..<\text{hd} \ (\text{dims} \ A)])$)
      proof
        have $\text{map} \ (\lambda A. \ \text{subtensor} \ A \ j) \ (\text{map} \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)]) = \text{map} \ (\lambda i. \ \text{subtensor} \ (\text{?f} \ i) \ [0..<\text{hd} \ (\text{dims} \ A)])$
          unfolding \text{map-map}[\text{of} \ (\lambda A. \ \text{subtensor} \ A \ j) \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)]]$ by simp
        with $1$ show $\forallthesis \ by \ \text{metis}$
      qed
    qed
qed

lemma \text{subtensor-decomposition:}
assumes $\text{dims} \ A \neq []$
shows $\text{listsun} \ (\text{dims} \ A) \ (\text{map} \ (\lambda i. \ \text{unit-vec} \ (\text{hd} \ (\text{dims} \ A))) \ i \otimes \text{subtensor} \ A \ i)$
$[0..<\text{hd} \ (\text{dims} \ A)]) = A \ \text{(is } ?LS = A)$
proof
  let $?f = \lambda i. \ \text{unit-vec} \ (\text{hd} \ (\text{dims} \ A))) \ i \otimes \text{subtensor} \ A \ i$
  have $\text{correct-dims}$: $\forall B. \ B \in \text{set} \ (\text{map} \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)]) \implies \text{dims} \ B = \text{dims} \ A$
  proof
    fix $B$
    assume $B \in \text{set} \ (\text{map} \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)])$
    then obtain $i \ where B \cdot B = ?f \ i$ \text{and} $i < \text{hd} \ (\text{dims} \ A)$ by auto
    then have $\text{dims} \ (\text{subtensor} \ A \ i) = \text{tl} \ (\text{dims} \ A)$ using \text{dims-subtensor using asms} by blast
    then show $\text{dims} \ B = \text{dims} \ A$ unfolding $B$
      by (metis append-\text{Cons asms} \text{dims-tensor-prod dims-unit-vec list.exhaust-sel self-append-conv2})
  qed
  have $\forall j. \ j < \text{hd} \ (\text{dims} \ A) \implies \text{subtensor} \ ?LS \ j = \text{subtensor} \ A \ j$
  proof
    fix $j$
    assume $j < \text{hd} \ (\text{dims} \ A)$
    have $1 \cdot \text{subtensor} \ ?LS \ j = \text{listsun} \ (\text{tl} \ (\text{dims} \ A)) \ (\text{map} \ (\lambda A. \ \text{subtensor} \ A \ j) \ (\text{map} \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)])])$
      using \text{subtensor-listsun}[\text{of} \ (\text{map} \ (\lambda i. \ ?f \ i) \ [0..<\text{hd} \ (\text{dims} \ A)]) \ \text{dims} \ A, \ \text{OF correct-dims asms} \ j < \text{hd} \ (\text{dims} \ A)]$
      by linarith
    also have $\ldots = \text{listsun} \ (\text{tl} \ (\text{dims} \ A)) \ (\text{map} \ (\lambda i. \ \text{subtensor} \ (\text{?f} \ i) \ [0..<\text{hd} \ (\text{dims} \ A)])$)
      proof
        have $\text{map} \ (\lambda A. \ \text{subtensor} \ A \ j) \ (\text{map} \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)]) = \text{map} \ (\lambda i. \ \text{subtensor} \ (\text{?f} \ i) \ [0..<\text{hd} \ (\text{dims} \ A)])$
          unfolding \text{map-map}[\text{of} \ (\lambda A. \ \text{subtensor} \ A \ j) \ ?f \ [0..<\text{hd} \ (\text{dims} \ A)]]$ by simp
        with $1$ show $\forallthesis \ by \ \text{metis}$
      qed
    qed
  qed

28
also have ... = map (λi. if i = j then subtensor A i else tensor0 (dims (subtensor A i))) [0..<hd (dims A)] ! j

unfolding subtensor-prod-with-unit-vec[OF i < hd (dims A)]
using listsum-all-0-but-one[of j (map (λi. if i = j then subtensor A i else tensor0 (dims (subtensor A i))) [0..<hd (dims A)]) tl (dims A)]
by (simp add: i < hd (dims A); assms)
also have ... = subtensor A j by (simp add: i < hd (dims A))
finally show subtensor ?LS j = subtensor A j by auto
qed

moreover have dims ?LS = dims A using correct-dims listsum-dims by blast
ultimately show ?thesis using subtensor-eqI by (metis (no-types, lifting) assms)
qed

end

7 Tensor CP-Rank

theory Tensor-Rank
imports Tensor-Unit-Vec
begin

inductive cprank-max1::'a::ring-1 tensor ⇒ bool where
order1: order A ≤ 1 ⇒ cprank-max1 A
higher-order: order A = 1 ⇒ cprank-max1 B ⇒ cprank-max1 (A ⊗ B)

lemma cprank-max1-order0: cprank-max1 B ⇒ order A = 0 ⇒ cprank-max1 (A ⊗ B)
proof (induction B rule:cprank-max1.induct)
case order1
then show ?case by (simp add: cprank-max1.order1)
next
case (higher-order A' B)
then have order (A ⊗ A') = 1 by simp
then show ?case using higher-order cprank-max1.higher-order by (metis mult.assoc)
qed

lemma cprank-max1-order-le1: order A ≤ 0 ⇒ cprank-max1 B ⇒ cprank-max1 (A ⊗ B)
by (simp add: cprank-max1-order0)

lemma cprank-max1-prod: cprank-max1 A ⇒ cprank-max1 B ⇒ cprank-max1 (A ⊗ B)
apply(induction A rule: cprank-max1.induct)
apply (meson higher-order le-neq-trans less-one cprank-max1-order0)
by (simp add: higher-order mult.assoc)

lemma cprank-max1-prod-list:
assumes A∈set Bs ⇒ cprank-max1 B
shows cprank-max1 (prod-list Bs)
using assms by (induction Bs, metis dims-smult dims-tensor0 list.size(3) prod-list.Nil order1 order-0-multiple-of-one zero-le-one, simp add: cprank-max1-prod)

lemma cprank-max1-prod-listE:
  fixes A::'a::comm-ring-1 tensor
  assumes cprank-max1 A
  obtains Bs a where \( \forall B. B \in \text{set} \, Bs \implies \text{order} B = 1 \cdot a \cdot \text{prod-list} \, Bs = A \)
  using assms proof (induction A arbitrary:thesis rule:cprank-max1.induct)
  case (order1 A)
  then show ?case proof (cases order A = 0)
    case True
    then obtain a where A = a \cdot \text{prod-list} [\]
    using order-0-multiple-of-one
    using prod-list.Nil by auto
    then show ?thesis using length-pos-if-in-set order1.prems by fastforce
  next
    case False
    then have order A = 1 using order1 by linarith
    then have A = 1 \cdot \text{prod-list} [A]
    by (simp add: smult-1)
    then show ?thesis by (metis \( \\langle \text{order} A = 1 \rangle \cdot \text{length-greater-0-conv} \cdot \text{length-pos-if-in-set} \cdot \text{order1.prems} \cdot \text{set-ConsD} \))
  qed

next
  case (higher-order A B)
  then obtain Bs b where \( \forall B'. B' \in \text{set} \, Bs \implies \text{order} B' = 1 \)\cdot b \cdot \text{prod-list} \, Bs = B\) by metis
  then have \( \forall B. B \in \text{set} \, (A \# Bs) \implies \text{order} B = 1 \)\)\cdot b \cdot \text{prod-list} \, Bs = B\) by metis
  then show ?case by (metis \( \langle \text{order} A = 1 \rangle \cdot \text{length-greater-0-conv} \cdot \text{length-pos-if-in-set} \cdot \text{order1.prems} \cdot \text{set-ConsD} \))

qed

inductive cprank-max :: nat \Rightarrow 'a::ring-1 tensor \Rightarrow bool where
  cprank-max0: cprank-max 0 (tensor0 ds) |
  cprank-max-Suc: dims A = dims B \implies cprank-max1 A \implies cprank-max \, (Suc j) \, (A+B)

lemma cprank-max1: cprank-max1 A \implies cprank-max 1 A
  by (metis One-nat-def dims-tensor0 cprank-max.simps cprank-max0 tensor-add-0-right)

lemma cprank-max-plus: cprank-max i A \implies cprank-max j B \implies dims A = dims B \implies cprank-max (i+j) \, (A+B)
  apply (induction A arbitrary:thesis rule:cprank-max.induct)
  apply auto[1]
  by (metis add-Suc plus-assoc plus-dim1 cprank-max.intros(2))

lemma cprank-max-listsum:
\(\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds\)

\(\bigwedge A. A \in \text{set } As \implies \text{cprank-max } n A\)

\text{shows} \(\text{cprank-max } (n * \text{length } As) (\text{listsum } ds As)\)

\text{using} \text{assms} \text{proof} (\text{induction } As)

\text{case} Nil

\text{then show} \ ?\text{case using} \text{listsum-Nil cprank-max}\text{.simps by fastforce}

\text{next}

\text{case} (\text{Cons } A As)

\text{then show} \ ?\text{case using} \text{cprank-max-plus} [\text{of } n A n * \text{length } As \text{ listsum } ds As]

\text{by} (\text{simp add: length-Cons list.set-intros(1) listsum-Cons listsum-dims set-subset-Cons subsetCE})

\text{qed}

\text{lemma} \text{cprank-maxE}:

\text{assumes} \text{cprank-max } n A

\text{obtains} BS \text{ where} (\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B) \text{ and} (\bigwedge B. B \in \text{set } BS \implies \text{dims } A = \text{dims } B) \text{ and} \text{ listsum } (\text{dims } A) BS = A \text{ and} \text{ length } BS = n

\text{using} \text{assms} \text{proof} (\text{induction } \text{arbitrary:thesis rule: cprank-max.induct})

\text{case} (\text{cprank-max0 } ds)

\text{have} \text{Tensor-Plus. listsum} (\text{dims } (\text{tensor0 } ds)) [] = \text{tensor0 } ds \text{ by} (\text{simp add: listsum-Nil})

\text{then show} \ ?\text{case using} \text{cprank-max0}.prems by fastforce

\text{next}

\text{case} (\text{cprank-max-Suc } A B j)

\text{then obtain} BS \text{ where} BS-def: (\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B) (\bigwedge B'. B' \in \text{set } BS \implies \text{dims } B' = \text{ dims } B)

\text{ listsum} (\text{dims } B) BS = B \text{ length } BS = j \text{ by metis}

\text{then have} \text{ listsum} (\text{ dims } (A + B)) (A \neq BS) = A + B

\text{by} (\text{simp add: listsum-Cons cprank-max-Suc.hyps(1)})

\text{then show} \ ?\text{case using} BS-def length-Cons cprank-max-Suc.hyps(2) cprank-max-Suc.prems set-ConsD

\text{by} (\text{metis plus-dim1 cprank-max-Suc.hyps(1)})

\text{qed}

\text{lemma} \text{cprank-maxI}:

\text{assumes} \bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B

\text{and} \bigwedge B. B \in \text{set } BS \implies \text{ dims } B = ds

\text{shows} \text{cprank-max } (\text{length } BS) (\text{listsum } ds BS)

\text{using} \text{assms} \text{proof} (\text{induction } BS)

\text{case} Nil

\text{then show} \ ?\text{case by} (\text{simp add: listsum-Nil cprank-max0})

\text{next}

\text{case} (\text{Cons } B BS)

\text{then show} \ ?\text{case}

\text{by} (\text{simp add: length-Cons list.set-intros(1) list.set-intros(2) listsum-Cons listsum-dims cprank-max-Suc})

\text{qed}

\text{lemma} \text{cprank-max-0E:} \text{ cprank-max } 0 A \implies A = \text{tensor0 } (\text{dims } A) \text{ by} (\text{metis}
 dims-tensor0 length-0-conv cprank-max0 cprank-maxE)

lemma listsum-prod-distr-right:
assumes \((\forall C. C \in \text{set } CS \implies \text{dims } C = ds)\)
shows \(A \otimes \text{listsum } ds \ CS = \text{listsum } (\text{dims } A @ ds) \ (\text{map } (\lambda C. A \otimes C) \ CS)\)
using assms proof (induction CS)
  case Nil
  then show \(?case\) by (simp add: listsum-Nil)
next
  case (Cons C CS)
  then have \(\text{dims } C = \text{dims } (\text{listsum } ds \ CS)\)
  by (simp add: list.set-intros(1) list.set-intros(2) listsum-dims)
  then show \(?case unfolding listsum-Cons list.map(2)\)
  using tensor-prod-distr-right Cons.IH Cons.prems list.set-intros(2) by fastforce
qed

lemma cprank-max-prod-order1:
assumes \(\text{order } A = 1\) and \(\text{cprank-max } n \ B\)
shows \(\text{cprank-max } n \ (A \otimes B)\)
proof
  obtain \(CS\) where \((\forall C. C \in \text{set } CS \implies \text{cprank-max1 } C)\)
  and \((\forall C. C \in \text{set } CS \implies \text{dims } C = \text{dims } B)\)
  and \(\text{listsum } (\text{dims } B) \ CS = B\)
  and \(\text{length } CS = n\)
  using assms(2) cprank-maxE by metis
  define \(CS'\) where \(CS' = \text{map } (\lambda C. A \otimes C) \ CS\)
  then have \((\forall C'. C' \in \text{set } CS' \implies \text{cprank-max1 } C')\)
  using assms(1) higher-order \((\forall C. C \in \text{set } CS \implies \text{cprank-max1 } C) \ \text{imageE set-map by auto}\)
  have \(\text{listsum } (\text{dims } A @ \text{dims } B) \ CS' = A \otimes B\) using \(CS'\)-def (Tensor-Plus.listsum (dims B) CS = B)
  using \((\forall Ca. Ca \in \text{set } CS \implies \text{dims } Ca = \text{dims } B)\ \text{listsum-prod-distr-right by fastforce}\)
  then show \(?thesis\) by (metis (mono-tags, lifting) CS'-def \((\forall C'. C' \in \text{set } CS' \implies \text{cprank-max1 } C') \ \text{imageE length-map cprank-max1 set-map})\)
qed

lemma cprank-max-upper-bound:
shows \(\text{cprank-max } (\text{prod-list } (\text{dims } A)) \ A\)
proof (induction A rule:subtensor-induct)
  case (order-0 A)
  then have \(\text{cprank-max} 1 \ A\) using order1 cprank-max1 by force
  then show \(?case using order-0 by auto\)
next
  case (order-step A)
  define \(B$s\) where \(B$s = \text{map } (\lambda i. \text{unit-vec } (\text{hd } (\text{dims } A)) \ i \otimes \text{subtensor } A \ i) [0..<\text{hd } (\text{dims } A)]\)
have \( \forall B. B \in \text{set} \, Bs \implies \text{ dims} \, A = \text{ dims} \, B \)
proof -
  fix \( B \) assume \( B \in \text{set} \, Bs \)
  obtain \( i \) where \( i < \text{hd} \, \text{ dims} \, A \) \( \text{Bs}!i = B \) using \( \text{Bs-def} \, \langle B \in \text{set} \, Bs \rangle \) by auto
  then have \( \text{ dims} \, (\text{unit-vec} \, (\text{hd} \, \text{ dims} \, A)) \, i \otimes \text{subtensor} \, A \, i = \text{ dims} \, A \)
  using \( \text{dims-unit-vec} \, \text{order-step}\,\text{hyps} \)
  by (metis \append-Cons \text{dims-subtensor} \text{dims-tensor-prod} \text{exhaust-sel} \text{self-append-conv2})
  then show \( \text{ dims} \, A = \text{ dims} \, B \) using \( \text{Bs-def} \langle \text{Bs}!i = B, \, i < \text{hd} \, \text{ dims} \, A \rangle \)
by auto
qed

have \( \forall B. B \in \text{set} \, Bs \implies \text{ cprank-max} \, (\text{prod-list} \, (\text{tl} \, (\text{dims} \, A))) \, \text{Bs}!i = B \)
proof -
  fix \( B \) assume \( B \in \text{set} \, Bs \)
  obtain \( i \) where \( i < \text{hd} \, \text{ dims} \, A \) \( \text{Bs}!i = B \) using \( \text{Bs-def} \, \langle B \in \text{set} \, Bs \rangle \) by auto
  then have \( \text{ cprank-max} \, (\text{prod-list} \, (\text{tl} \, (\text{dims} \, A))) \, (\text{unit-vec} \, (\text{hd} \, \text{ dims} \, A)) \, i \otimes \text{subtensor} \, A \, i \)
  by (metis \text{One-nat-def} \text{dims-subtensor} \text{dims-unit-vec} \text{length-Cons} \text{list.size(3)} \text{order-step.IH} \text{order-step.hyps} \text{cprank-max-prod-order1})
  then show \( \text{ cprank-max} \, (\text{prod-list} \, (\text{tl} \, (\text{dims} \, A))) \, B \) using \( \text{Bs-def} \langle \text{Bs}!i = B, \, i < \text{hd} \, \text{ dims} \, A \rangle \) by auto
  qed
then show \( \text{ case using} \, \text{subtensor-decomposition}[\text{OF} \, \text{order-step.hyps}] \, \text{cprank-max-listsum} \)
  by (metis \text{no-types} \, \text{lifting} \, \text{Bs-def} \, \langle B \in \text{set} \, Bs \implies \text{ dims} \, A = \text{ dims} \, B, \, i < \text{hd} \, (\text{ dims} \, A) \rangle \, \text{by auto}\)
  qed

definition \( \text{cprank} : \, 'a::\text{ring-1} \, \text{tensor} \Rightarrow \text{nat} \) \text{where}
\( \text{cprank} \, A = (\text{LEAST} \, n. \, \text{cprank-max} \, n \, A) \)

lemma \( \text{cprank-upper-bound} : \, \text{cprank} \, A \leq \text{prod-list} \, (\text{dims} \, A) \)
unfolding \( \text{cprank-def} \) \text{using} \( \text{cprank-max-upper-bound} \) \text{Least-le} \text{by fastforce}

lemma \( \text{cprank-max-cprank} : \, \text{cprank-max} \, (\text{cprank} \, A) \, A \)
  unfolding \( \text{cprank-def} \) \text{using} \( \text{cprank-max-upper-bound} \) \text{by} \, \text{(metis} \, \text{LeastI))}
end

8 Tensor Matricization

theory Tensor-Matricization
imports Tensor-Plus
Jordan-Normal-Form.Matrix Jordan-Normal-Form.DL-Missing-Sublist
begin

fun \( \text{digit-decode} : \, \text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{nat} \) \text{where}
\( \text{digit-decode} \, \langle \rangle \, \langle \rangle = 0 \ |
\text{digit-decode} \, (d \, \# \, ds) \, (i \, \# \, is) = i + d \times \text{digit-decode} \, ds \, is \)


fun digit-encode :: nat list ⇒ nat ⇒ nat list where
digit-encode [] a = [] |
digit-encode (d # ds) a = a mod d # digit-encode ds (a div d)

lemma digit-encode-decode[simp]:
assumes is < ds
shows digit-encode ds (digit-decode ds is) = is
  using assms apply (induction rule:valid-index.induct)
  unfolding digit-decode.simps digit-encode.simps
  by simp-all

lemma digit-decode-encode[simp]:
shows digit-decode ds (digit-encode ds a) = a mod (prod-list ds)
by (induction ds arbitrary:a; simp add: Divides.mod2-eq add.commute)

lemma digit-decode-encode-lt[simp]:
assumes a < prod-list ds
shows digit-decode ds (digit-encode ds a) = a
by (simp add: assms)

lemma digit-decode-lt:
assumes is < ds
shows digit-decode ds is < prod-list ds
using assms proof (induction rule:valid-index.induct)
  case Nil
    then show ?case by simp
next
  case (Cons d ds is i d)
    have (i + d * digit-decode ds is) div (d * prod-list ds) = 0
      using Cons.IH Cons.hyps(2) div-mult2-eq by force
    then show ?case unfolding digit-decode.simps prod-list.Cons
      by (metis (no-types) Cons.IH Cons.hyps(2) div-eq-0-iff mult-0-iff not-less0)
  qed

lemma digit-encode-valid-index:
assumes a < prod-list ds
shows digit-encode ds a < ds
using assms proof (induction ds arbitrary:a)
  case Nil
    show ?case by (simp add: valid-index.Nil)
next
  case (Cons d ds a)
    then have a < d * prod-list ds
      by simp
    then have a div d < prod-list ds
      by (metis div-0-iff mult2-eq mult-0-right not-less0)
    then have digit-encode ds (a div d) < ds
      by (rule Cons)
    moreover have d > 0

34
using \( a < d \ast \prod\text{-list } ds \) by (cases \( d = 0 \)) simp-all
then have \( a \mod d < d \)
by simp
ultimately show \(?case\)
by (simp add; valid-index.Cons)
qed

lemma length-digit-encode:
shows \( \text{length } (\text{digit-encode } ds \ a) = \text{length } ds \)
by (induction \( ds \) arbitrary; \( a \); simp-all)

lemma digit-encode-0:
\( \prod\text{-list } ds \ \text{dvd } a \Rightarrow \text{digit-encode } ds \ a = \text{replicate } (\text{length } ds) 0 \)
proof (induction \( ds \) arbitrary)
next
case (\( \text{Cons } d \ \text{ds} \ a \))
then have \( \prod\text{-list } ds \ \text{dvd } (a \ \text{div } d) \)
unfolding \( \text{prod-list.Cons} \)
by (metis dvd-0-right dvd-div-iff-mult dvd-mult-left mult.commute split-div)
then show \(?case\)
unfolding \( \text{digit-encode.simps } length\text{-Cons } \text{replicate-Suc } \text{prod-list.Cons} \)
using \( \text{Cons} \)
using \( \text{dvd-imp-mod-0 } \text{dvd-mult-left} \text{ prod-list.Cons} \)
by force
qed

lemma valid-index-weave:
assumes \( \text{is1} < (\text{nths } ds \ A) \)
and \( \text{is2} < (\text{nths } ds \ (-A)) \)
shows \( \text{weave } A \ \text{is1} \ \text{is2} \ \text{ds} \)
and \( \text{nths } (\text{weave } A \ \text{is1} \ \text{is2}) \ A = \text{is1} \)
and \( \text{nths } (\text{weave } A \ \text{is1} \ \text{is2}) \ (-A) = \text{is2} \)
proof
  have \( \text{length-ds} \text{: length is1 + length is2 = length } ds \)
  using valid-index-length[of \( \text{assms}(1) \)] valid-index-length[of \( \text{assms}(2) \)]
  length-weave weave-complementary-nthss by metis
  have 1: \( \text{length is1} = \text{card } \{ i \in A. \ i < \text{length is1} + \text{length is2} \} \)
  unfolding length-ds
  using \( \text{length-nths'} \text{ assms}(1) \) valid-index-length by auto
  have 2: \( \text{length is2} = \text{card } \{ i \in -A. \ i < \text{length is1} + \text{length is2} \} \)
  unfolding length-ds
  using \( \text{length-nths'} \text{of } ds \ (-A) \) assms(2) valid-index-length by auto
  show \( \text{nths } (\text{weave } A \ \text{is1} \ \text{is2}) \ A = \text{is1 } \text{nths } (\text{weave } A \ \text{is1} \ \text{is2}) \ (-A) = \text{is2} \)
  using nths-weave[OF 1 2] by blast+
  then have \( \text{nths } (\text{weave } A \ \text{is1} \ \text{is2}) \ A \subset (\text{nths } ds \ A) \)
  \( \text{nths } (\text{weave } A \ \text{is1} \ \text{is2}) \ (-A) \subset (\text{nths } ds \ (-A)) \) assms by auto
  then show \( \text{weave } A \ \text{is1} \ \text{is2} \subset ds \)
  using list-all2-nths valid-index-list-all2-iff by blast
qed

definition matricize :: nat set \( \Rightarrow 'a \text{ tensor } \Rightarrow 'a \text{ mat } \) where
matricize $r$modes $T = mat$

$(\text{prod-list} (\text{nths} (\text{Tensor}.\text{dims} T) \text{rmodes}))$

$(\text{prod-list} (\text{nths} (\text{Tensor}.\text{dims} T) (\text{−rmodes})))$

$(\lambda (r, c). \text{Tensor}.\text{lookup} T (\text{weave} \text{rmodes} (\text{digit-encode} (\text{nths} (\text{Tensor}.\text{dims} T) \text{rmodes}) r))$

$(\text{digit-encode} (\text{nths} (\text{Tensor}.\text{dims} T) (\text{−rmodes})) c)\)$

\[ \text{definition dematricize::nat set ⇒ 'a mat ⇒ nat list ⇒ 'a tensor where} \]
\[ \text{dematricize rmodes A ds} = \text{tensor-from-lookup ds} \]

\[ (\lambda s. A \# (\text{digit-decode} (\text{nths} ds \text{rmodes}) (\text{nths is rmodes}), \text{digit-decode} (\text{nths} ds (\text{−rmodes}) (\text{nths is} (\text{−rmodes})))) \]
have decode-r: digit-decode ?rds (nths is rmodes) < prod-list ?rds
  by (simp add: iis < Tensor.dims T \ valid-index-nths digit-decode-lt)
have decode-c: digit-decode ?cds (nths is (¬ rmodes)) < prod-list ?cds
  by (simp add: iis < Tensor.dims T \ valid-index-nths digit-decode-lt)
have (matricize rmodes T) $\$$
  (digit-decode ?rds (nths is rmodes),
   digit-decode ?cds (nths is (¬ rmodes))) =
  Tensor.lookup T is
  unfolding matricize-def
  by (simp add: decode-r decode-c iis < Tensor.dims T \ valid-index-nths)
then show Tensor.lookup (dematricize rmodes (matricize rmodes T) (Tensor.dims T)) is = Tensor.lookup T is
  by (simp add: dematricize-def dims-tensor-from-lookup lookup-tensor-from-lookup[of iis < Tensor.dims T])
qed

lemma matricize-dematricize:
assumes dim-row A = prod-list (nths ds rmodes)
and dim-col A = prod-list (nths ds (¬ rmodes))
shows matricize rmodes (dematricize rmodes A ds) = A
proof (rule eq-matI)
  show dim-row (matricize rmodes (dematricize rmodes A ds)) = dim-row A
    unfolding assms(1) dematricize-def dims-tensor-from-lookup matricize-def dim-row-mat
  by metis
  show dim-col (matricize rmodes (dematricize rmodes A ds)) = dim-col A
    unfolding assms(2) dematricize-def dims-tensor-from-lookup matricize-def dim-col-mat
  by metis
  fix r c assume r < dim-row A c < dim-col A
  have valid1: digit-encode (nths ds rmodes) r < nths ds rmodes and
    valid2: digit-encode (nths ds (¬ rmodes)) c < nths ds (¬ rmodes)
  using r < dim-row A \ assms(1) iis < dim-col A \ assms(2) digit-encode-valid-index
  by auto
  have 0: Tensor.lookup (dematricize rmodes A ds)
    (weave rmodes
      (digit-encode (nths (Tensor.dims (dematricize rmodes A ds))) rmodes) r)
    (digit-encode (nths (Tensor.dims (dematricize rmodes A ds))) (¬ rmodes)) c)
    = A $\$$ (r, c)
  unfolding dematricize-def unfolding dims-tensor-from-lookup
  unfolding lookup-tensor-from-lookup[of valid-index-weave(1) [OF validI valid2]]
  using digit-decode-encode-lt[of iis < dim-col A \ unfolded assms(2)]
  digit-decode-encode-lt[of r < dim-row A \ unfolded assms(1)]
    valid-index-weave(2)[OF validI valid2] valid-index-weave(3)[OF validI valid2]
    by presburger
  from r < dim-row A \ have r-le: r < prod-list (nths (Tensor.dims (dematricize rmodes A ds))) rmodes)
    by (metis dim-row (matricize rmodes (dematricize rmodes A ds)) = dim-row A \ matricize-def dim-row-mat(1))
  from c < dim-col A \ have c-le: c < prod-list (nths (Tensor.dims (dematricize rmodes A ds))) (¬ rmodes))

37
by (metis \<dim-col\> (matricize rmodes (dematricize rmodes A ds)) = \<dim-col\> A \> matricize-def \<dim-col\>-mat(1))

then show (matricize rmodes (dematricize rmodes A ds)) \$$ (r, c) = A \$$ (r, c)

unfolding matricize-def using r-le c-le 0 by simp

qed

lemma matricize-add:
assumes \(\text{dims A} = \text{dims B}\)
shows matricize I A + matricize I B = matricize I (A+B)
proof (rule eq-matI)
  show \(\text{dim-row} (\text{matricize I A + matricize I B}) = \text{dim-row} (\text{matricize I (A + B)})\) by (simp add: assms dims-matricize(1))
  show \(\text{dim-col} (\text{matricize I A + matricize I B}) = \text{dim-col} (\text{matricize I (A + B)})\)
  by (simp add: assms dims-matricize(2))
  fix i j assume \(ij-le1<i < \text{dim-row} (\text{matricize I (A+B)})\) \(j < \text{dim-col} (\text{matricize I (A + B)})\)
  then have \(ij-le2<i < \text{prod-list} (\text{nths (Tensor.dims A)} I) \) \(<\text{dim-col} (\text{matricize I (A+B)})) (-1)\)
  and \(ij-le3<i < \text{prod-list} (\text{nths (Tensor.dims B)} I) \) \(<\text{dim-col} (\text{matricize I (A+B)})) (-1)\)
  and \(ij-le4<i < \text{prod-list} (\text{nths (Tensor.dims (A + B))} I) \) \(<\text{dim-col} (\text{matricize I (A+B)})) (-1)\)
  by (simp add: assms dims-matricize)
  then have \(ij-le5<i < \text{dim-row} (\text{matricize I (A+B)} j < \text{dim-col} (\text{matricize I (A+B)}\)
  by (simp add: assms dims-matricize)
  show (matricize I A + matricize I B) \$$ (i, j) = \text{matricize I (A + B)} \$$ (i, j)
  using assms digit-encode-valid-index ij-le2(1) ij-le2(2) valid-index-weave(1)
  by auto
qed

lemma matricize-0:
shows matricize I (\(\text{tensor0 ds}\) = \(0_m \) (\(\text{dim-row} (\text{matricize I (tensor0 ds)})\) (\(\text{dim-col} (\text{matricize I (tensor0 ds)})\))
proof (rule eq-matI)
  show \(\text{dim-row} (\text{matricize I (tensor0 ds)}) = \text{dim-row} (0_m (\text{dim-row} (\text{matricize I (tensor0 ds)})))\)
  (\text{dim-col} (\text{matricize I (tensor0 ds)})))
  unfolding zero-mat-def dim-row-mat by (simp add: assms dims-matricize(1))
  show \(\text{dim-col} (\text{matricize I (tensor0 ds)}) = \text{dim-col} (0_m (\text{dim-row} (\text{matricize I (tensor0 ds)})))\)
  (\text{dim-col} (\text{matricize I (tensor0 ds)})))
  unfolding zero-mat-def dim-row-mat by (simp add: assms dims-matricize(2))
  fix i j assume \(ij-le1<i < \text{dim-row} (\(0_m \) (\(\text{dim-row} (\text{matricize I (tensor0 ds)})\))\) (\(\text{dim-col} (\text{matricize I (tensor0 ds)})\)) j < \text{dim-col} (\(0_m \) (\(\text{dim-row} (\text{matricize I (tensor0 ds)})\)) (\(\text{dim-col} (\text{matricize I (tensor0 ds)})\)))
  then have \(ij-le2<i < \text{dim-row} (\text{matricize I (tensor0 ds)}) j < \text{dim-col} (\text{matricize I (tensor0 ds)})\)

38
unfolding zero-mat-def dim-row-mat by (simp-all add: dims-matricize)
show matricize I (tensor0 ds) $\langle i, j \rangle = 0$
show (dim-col (matricize I (tensor0 ds))) $\langle i, j \rangle$
unfolding zero-mat-def index-mat[OF ij-le2]
unfolding matricize-def index-mat[OF ij-le2[unfolded dims-matricize]]
by (simp, metis lookup-tensor0 digit-encode-valid-index dims-matricize(1) dims-matricize(2)
dims-tensor0
ij-le2(1) ij-le2(2) valid-index-weave(1))
qed

9 CP-Rank and Matrix Rank

theory DL-Rank-CP-Rank
imports Tensor-Rank Jordan-Normal-Form.DL-Rank Tensor-Matricization
Jordan-Normal-Form.DL-Submatrix Jordan-Normal-Form.Missing-VectorSpace
begin

abbreviation mrank A == vec-space.rank (dim-row A) A

no-notation normal-rel (infixl $\prec$ 60)

lemma lookup-order1-prod:
assumes $\forall B. B \in set Bs \Rightarrow Tensor.order B = 1$
assumes is $\prec$ dims (prod-list Bs)
shows lookup (prod-list Bs) is = prod-list (map ($\lambda(i,B). lookup B [i]$) (zip is Bs))
using assms proof (induction Bs arbitrary: is)
case Nil then show ?case unfolding prod-list.Nil unfolding zip.simps tensor-one-def
by (metis (no-types, lifting) dims-tensor-prod length-greater-0-conv length-map
prod-list.Nil
lookup-tensor-from-lookup tensor-one-def tensor-one-from-lookup)
next
case (Cons B Bs is')
then obtain i is where is' = i $\neq$ is
by (metis append-is-Nil-conv dims-tensor-prod length-0-conv list-set-intros(1)
prod-list.Cons valid-index.simps zero-neq-one)
have Tensor.order B = 1 using Cons by auto
then have valid1:[i] $\prec$ dims B
using i' $\prec$ dims (prod-list (B $\neq$ Bs))[unfolded prod-list.Cons dims-tensor-prod
i' = i $\neq$ i]
by (metis One-dat-def Suc-length-conv hd-append2 length-0-conv list.sel(1)
list.simps(3) valid-index.Nil valid-index.simps)
have valid2:i $\prec$ dims (prod-list Bs)
using i' $\prec$ dims (prod-list (B $\neq$ Bs))[unfolded prod-list.Cons dims-tensor-prod
i' = i $\neq$ i] (Tensor.order B = 1)
by (metis One-dat-def Suc-length-conv append-eq-Cons-conv length-0-conv list.sel(3)
list.simps(3) self-append-conv2 valid-indexE

show ?case unfolding is' = i # i. List.zip-Cons-Cons List.list.map(2) prod-list.Cons
lookup-tensor-prod[OF valid1 valid2, simplified] by (simp add: Cons.IH Cons.prems(1)
valid2)

qed

lemma matricize-cprank-max1:
fixes A::'a::field tensor
assumes cprank-max1 A
shows mrank (matricize I A) ≤ 1
proof –
  obtain Bs a where AB. B ∈ set Bs ⇒ Tensor.order B = 1 a · prod-list Bs = A
    using cprank-max1-prod-listE assms by metis
  define row-factor
  where row-factor ris = a * prod-list (map (λ(i,B). lookup B [i]) (zip ris (nths Bs Bs I)))
    for ris
  define col-factor
  where col-factor cis = prod-list (map (λ(i,B). lookup B [i]) (zip cis (nths Bs Bs
(−I))))
    for cis
  have AB. i < dims A ⇒ lookup A is = row-factor (nths is I) * col-factor
(nths is (−I))
    proof –
      fix is assume is < dims A
      then have lookup A is = a * (prod-list (map (λ(i,B). lookup B [i]) (zip is Bs)))
        using lookup-order1-prod[OF AB. B ∈ set Bs ⇒ Tensor.order B = 1] lookup-smult
        using a · prod-list Bs = A · dims-smult by fastforce
      also have ... = a * (prod-list (map (λ(i,B). lookup B [i]) (nths (zip is Bs Bs
(−I))))
        * (prod-list (map (λ(i,B). lookup B [i]) (nths (zip is Bs Bs (−I)))))
        using prod-list-complementary-nthss by auto
      also have ... = row-factor (nths is I) * col-factor (nths is (−I))
        using nths-zip row-factor-def col-factor-def by metis
      finally show lookup A is = row-factor (nths is I) * col-factor (nths is (−I)).
    qed
  define row-factor'
  where row-factor' r = row-factor (digit-encode (nths (Tensor.dims A) I) r)
  for r
  define col-factor'
  where col-factor' c = col-factor (digit-encode (nths (Tensor.dims A) (−I)) c)
  for c
  have AB. c < dim-row (matricize I A) ⇒ c < dim-col (matricize I A) ⇒
  matricize I A $$(r,c) = row-factor' r * col-factor' c
    proof –
      fix r c assume r < dim-row (matricize I A) c < dim-col (matricize I A)
      then have matricize I A $$(r,c) = Tensor.lookup A (weave I

40
(digit-encode (nths (Tensor.dims A) I) r)
(digit-encode (nths (Tensor.dims A) (−I)) c)
)
unfolding dims-matricize unfolding matricize-def by simp
also have ... = row-factor′ r * col-factor′ c
using \{is. is < dims A \implies lookup A is = row-factor (nths is I) * col-factor
(nths is (− I))\}
valid-index-weave[OF
digit-encode-valid-index[OF r < dim-row (matricize I A)][unfolded dims-matricize]]
digit-encode-valid-index[OF c < dim-col (matricize I A)][unfolded dims-matricize]]
valid-index-weave(2) valid-index-weave(3) row-factor′-def col-factor′-def by metis
finally show matricize I A $$ (r,c) = row-factor′ r * col-factor′ c.$$ .
qed
then show \?thesis using vec-space.rank-le-1-product-entries[of matricize I A] by blast
qed

lemma matrix-rank-le-cp-rank-max:
fixes A :: ('a::field) tensor
assumes cp-rank-max r A
shows mrank (matricize I A) ≤ r
using assms
proof (induction rule:cp-rank-max.induct)
fix ds :: nat list
have matricize I (tensor0 ds) = 0_m (dim-row (matricize I (tensor0 ds))) (dim-col
(matricize I (tensor0 ds)))
using matricize-0 by auto
then show mrank (matricize I (tensor0 ds)) ≤ 0
using eq-imp-le vec-space.rank-0I by metis
next
fix A B::'a tensor and j::nat
assume dims A = dims B
assume cp-rank-max1 A
assume mrank (matricize I B) ≤ j
have mrank (matricize I A) ≤ I using \{cp-rank-max1 A\} matricize-cp-rank-max1
by auto
have mrank (matricize I (A + B)) ≤ mrank (matricize I A) + mrank (matricize I B)
using matricize-add vec-space.rank-subadditive dims-matricize
carrier-mat! index-add-mat(2) of dims A = dims B\ by metis
then show mrank (matricize I (A + B)) ≤ Suc j
using \{mrank (matricize I A) ≤ I\} \{mrank (matricize I B) ≤ j\} by linarith
qed

lemma matrix-rank-le-cp-rank:
fixes A :: ('a::field) tensor
shows mrank (matricize I A) ≤ cp-rank A
using matrix-rank-le-cp-rank-max using cp-rank-max-cp-rank by auto

41
theory DL-Flatten-Matrix
imports Jordan-Normal-Form.Matrix
begin

definition extract-matrix :: (nat ⇒ 'a) ⇒ nat ⇒ nat ⇒ 'a mat where
extract-matrix a m n = mat m n (∑(i, j). a (i * n + j))

definition flatten-matrix :: 'a mat ⇒ (nat ⇒ 'a) where
flatten-matrix A k = A $$ (k div dim-col A, k mod dim-col A)

lemma two-digit-le:
i * n + j < m * n if i < m j < n for i j :: nat
using that by (auto dest!: less-imp-Suc-add simp add: algebra-simps)

lemma extract-matrix-cong:
assumes ⋀i. i < m * n ⇒ a i = b i
shows extract-matrix a m n = extract-matrix b m n
proof –
have ⋀i j. i < m ⇒ j < n ⇒ a (i * n + j) = b (i * n + j)
using two-digit-le
assms by blast
then show ?thesis unfolding extract-matrix-def by auto
qed

lemma extract-matrix-flatten-matrix:
extract-matrix (flatten-matrix A) (dim-row A) (dim-col A) = A
unfolding extract-matrix-def flatten-matrix-def by auto

lemma extract-matrix-flatten-matrix-cong:
assumes ⋀x. x < dim-row A * dim-col A ⇒ f x = flatten-matrix A x
shows extract-matrix f (dim-row A) (dim-col A) = A
unfolding extract-matrix-def
by (metis assms extract-matrix-cong extract-matrix-def extract-matrix-flatten-matrix)

lemma flatten-matrix-extract-matrix:
flatten-matrix (extract-matrix a m n) k = a k if k < m * n
proof –
from that have m * n > 0
by (cases m * n = 0) simp-all
then have m > 0 and n > 0
by simp-all
with that have k div n < m
by (metis div-eq-0-iff div-mul2-eq mult.commute neq0_conv)
moreover have k mod n < n
using ⟨n > 0⟩ by simp
ultimately show ?thesis
end
by (auto simp add: extract-matrix-def flatten-matrix-def)
qed

lemma index-extract-matrix:
assumes i:<m j:<n
shows extract-matrix a m n $$ (i,j) = a (i\cdot n + j)$$
unfolding extract-matrix-def using assms by simp

lemma dim-extract-matrix:
shows dim-row (extract-matrix as m n) = m
and dim-col (extract-matrix as m n) = n
unfolding extract-matrix-def by simp-all
end

11 Deep Learning Networks

theory DL-Network
imports Tensor-Product
  Jordan-Normal-Form.Matrix Tensor-Unit-Vec DL-Flatten-Matrix
  Jordan-Normal-Form.DL-Missing-List
begin
  This symbol is used for the Tensor product:
  no-notation Group.monoid.mult (infixl ⊗s 70)
  notation Matrix.unit-vec (unitv)
  hide-const (open) Matrix.unit-vec

datatype 'a convnet = Input nat | Conv 'a 'a convnet | Pool 'a convnet 'a convnet

fun input-sizes :: 'a convnet ⇒ nat list where
input-sizes (Input M) = [M] |
input-sizes (Conv A m) = input-sizes m |
input-sizes (Pool m1 m2) = input-sizes m1 @ input-sizes m2

fun count-weights :: bool ⇒ (nat × nat) convnet ⇒ nat where
count-weights shared (Input M) = 0 |
count-weights shared (Conv (r0, r1) m) = r0 * r1 + count-weights shared m |
count-weights shared (Pool m1 m2) =
  (if shared
    then max (count-weights shared m1) (count-weights shared m2)
    else count-weights shared m1 + count-weights shared m2)

fun output-size :: (nat × nat) convnet ⇒ nat where
output-size (Input M) = M |
output-size (Conv (r0,r1) m) = r0 |
output-size (Pool m1 m2) = output-size m1
inductive valid-net :: (nat × nat) convnet ⇒ bool where
valid-net (Input M) |
output-size m = r1 ⇒ valid-net m ⇒ valid-net (Conv (r0, r1) m) |
output-size m1 = output-size m2 ⇒ valid-net m1 ⇒ valid-net m2 ⇒ valid-net
(Pool m1 m2)

fun insert-weights :: bool ⇒ (nat × nat) convnet ⇒ (nat ⇒ real) ⇒ real mat
convnet where
insert-weights shared (Input M) w = Input M |
insert-weights shared (Conv (r0, r1) m) w = Conv
(extract-matrix w r0 r1) |
(insert-weights shared m (λi. w (i + r0 ∗ r1))) |
insert-weights shared (Pool m1 m2) w = Pool
(insert-weights shared m1 w) |
(insert-weights shared m2 (if shared then w else (λi. w (i + (count-weights shared
m1)))))

fun remove-weights :: real mat convnet ⇒ (nat × nat) convnet where
remove-weights (Input M) = Input M |
remove-weights (Conv A m) = Conv (dim-row A, dim-col A) (remove-weights m) |
remove-weights (Pool m1 m2) = Pool (remove-weights m1) (remove-weights m2)

abbreviation output-size' == (λm. output-size (remove-weights m))
abbreviation valid-net' == (λm. valid-net (remove-weights m))

fun evaluate-net :: real mat convnet ⇒ real vec list ⇒ real vec where
evaluate-net (Input M) inputs = hd inputs |
evaluate-net (Conv A m) inputs = A ∗ v evaluate-net m inputs |
evaluate-net (Pool m1 m2) inputs = component-mult
(evaluate-net m1 (take (length (input-sizes m1)) inputs)) |
(evaluate-net m2 (drop (length (input-sizes m1)) inputs))

definition mat-tensorlist-mult :: real mat ⇒ real tensor vec ⇒ nat list ⇒ real

where mat-tensorlist-mult A Ts ds = Matrix.vec (dim-row A) (λj. tensor-from-lookup
ds (λis. (A ∗ v (map-vec (λT. Tensor.lookup T is) Ts)) $j))

lemma insert-weights-cong:
assumes (\(\forall i. i < \text{count-weights } s \Rightarrow w1 i = w2 i\))
shows insert-weights s m w1 = insert-weights s m w2
using assms proof (induction m arbitrary: w1 w2)
case Input
then show ?case by simp
next
case (Conv r01 m)
then obtain r0 r1 where r01 = (r0, r1) by (meson surj-pair)

44
have 2: insert-weights s m (λ. w1 (i + r0 * r1)) = insert-weights s m (λ. w2 (i + r0 * r1)) using Conv
using (r01 = (r0, r1)) add.commute add-less-cancel-right count-weights.simps(2)
by fastforce
then show ?case unfolding (r01 = (r0, r1)) insert-weights.simps
by (metis Conv.prems (r01 = (r0, r1)) count-weights.simps(2) extract-matrix-cong
trans-less-add1)
next

have 1: insert-weights s m1 w1 = insert-weights s m2 w2 using Pool (1)[of w1 w2]
Pool (3)[unfolded count-weights.simps] by (cases s; auto)
have shared: s= True ⇒ insert-weights s m2 w1 = insert-weights s m2 w2
using Pool (2)[of w1 w2] Pool (3)[unfolded count-weights.simps] by auto
have unshared: s= False ⇒ insert-weights s m2 (λ. w1 (i + count-weights s m1)) = insert-weights s m2 (λ. w2 (i + count-weights s m1))
using Pool (2) Pool (3) count-weights.simps by fastforce
show ?case unfolding insert-weights.simps 1 using unshared shared by simp

qed

lemma dims-mat-tensorlist-mult:
assumes T∈ setv (mat-tensorlist-mult A Ts ds)
shows Tensor.dims T = ds
proof
  obtain j where T = tensor-from-lookup ds (λis. (A *v (map-vec (λT. Tensor.lookup T is) Ts)) $j)
  using vec-setE[OF assms, unfolded mat-tensorlist-mult-def] by (metis dim-vec index-vec)
  then show ?thesis by (simp add: length-tensor-vec-from-lookup tensor-from-lookup-def)
qed

fun tensors-from-net :: real mat convnet ⇒ real tensor vec
where
tensors-from-net (Input M) = Matrix.vec M (λi. unit-vec M i) |
tensors-from-net (Conv A m) = mat-tensorlist-mult A (tensors-from-net m) (input-sizes m) |
tensors-from-net (Pool m1 m2) = component-mult (tensors-from-net m1) (tensors-from-net m2)

lemma output-size-correct-tensors:
assumes valid-net' m
shows output-size' m = dim-vec (tensors-from-net m)
using assms proof (induction m)
  case Input
  then show ?case by simp
next
  case (Conv A m)
  then show ?case
    unfolding remove-weights.simps output-size.simps tensors-from-net.simps
    using mat-tensorlist-mult-def by auto

qed
next
  case (Pool m1 m2)
  then show ?case by (metis convnet.distinct(3) convnet.distinct(5) convnet.inject(3)
   dim-component-mult
   min_idem output-size.simps(3) remove-weights.simps(3) tensors-from-net.simps(3)
   valid-net.simps)
qed

lemma output-size-correct:
assumes valid-net' m
and map dim-vec inputs = input-sizes m
shows output-size' m = dim-vec (evaluate-net m inputs)
using assms proof (induction m arbitrary:inputs)
  case Input
  then show ?case using length-Cons list.map-sel(1) list.sel(1) list.simps(8)
   list.size(3) nat.simps(3) by auto
next
  case (Conv A m)
  then show ?case unfolding evaluate-net.simps remove-weights.simps output-size.simps
dim-mult-mat-vec
   by auto
next
  case (Pool m1 m2)
  then have valid-net' m1 valid-net' m2
   using convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-weights.simps(3)
   valid-net.cases by fastforce+
   moreover have map dim-vec (take (length (input-sizes m1)) inputs) = in-
   put-sizes m1
   map dim-vec (drop (length (input-sizes m1)) inputs) = input-sizes m2
   using Pool.prems(2) by (metis append-eq-conv-conj drop-map input-sizes.simps(3)
    take-map)+
   ultimately have
    output-size' m1 = dim-vec (evaluate-net m1 (take (length (input-sizes m1))
     inputs))
    output-size' m2 = dim-vec (evaluate-net m2 (drop (length (input-sizes m1))
     inputs))
   using Pool.IH by blast+
  then show ?case unfolding output-size.simps remove-weights.simps output-size.simps
   by (metis Pool.prems(1) valid-net' m1 valid-net' m2 dim-component-mult
    output-size.simps(3) output-size-correct-tensors remove-weights.simps(3) ten-
    sors-from-net.simps(3))
qed

lemma input-sizes-remove-weights: input-sizes m = input-sizes (remove-weights m)
  by (induction m; simp)

lemma dims-tensors-from-net:
assumes \( T \in \text{set}_v \) (tensors-from-net \( m \))

shows \( \text{Tensor.dims } T = \text{input-sizes } m \)

using assms proof (induction \( m \) arbitrary: \( T \))

- case (Input \( M \))
  then obtain \( j \) where \( T = \text{unit-vec } M j \)
    using vec-setE tensors-from-net.simps(1) by (metis dim-vec index-vec)
  then show \( \text{?case} \) by (simp add: dims-unit-vec)

- next
  case (Conv \( A \) \( m \))
  then show \( \text{?case} \) unfolding remove-weights.simps input-sizes.simps
    using dims-mat-tensorlist-mult by (simp add: input-sizes-remove-weights)

- next
  case (Pool \( m1 \) \( m2 \) \( T \))
  then obtain \( i \) where \( \text{component-mult} \) (tensors-from-net \( m1 \)) (tensors-from-net \( m2 \)) \( i \)
    \( \text{< dim-vec} \) (tensors-from-net \( m1 \)) \( i \)
    \( \text{< dim-vec} \) (tensors-from-net \( m2 \))
  using vec-setE dim-component-mult by (metis min.strict-boundedE)
  then obtain \( T1 \) \( T2 \) where \( T = T1 \otimes T2 \)
    \( T1 \in \text{set}_v \) (tensors-from-net \( m1 \))
  \( T2 \in \text{set}_v \) (tensors-from-net \( m2 \))
  using vec-setI by (metis index-component-mult)
  then show \( \text{?case} \) unfolding remove-weights.simps input-sizes.simps
    using dimensions by (simp add: Pool.IH(1) Pool.IH(2))
qed

definition base-input :: \( \text{real mat convnet } \Rightarrow \text{nat list } \Rightarrow \text{real vec list} \)
where base-input \( m \) is \( = \text{(map } \lambda (n, i). \text{unit}_v n i) (\text{zip } \text{input-sizes } m \text{ )} \)

lemma base-input-length:
assumes \( \text{is } \triangleq \text{input-sizes } m \)
shows \( \text{input-sizes } m = \text{map dim-vec} \) (base-input \( m \) is)
proof (rule nth-equalityI)
  have length (input-sizes \( m \)) = length is using assms valid-index-length by auto
  then show length (input-sizes \( m \)) = length (map dim-vec (base-input \( m \) is))
unfolding base-input-def by auto
{
  fix \( i \)
  assume \( i < \text{length} \) (input-sizes \( m \))
  then have map \( \lambda (n, i). \text{unit}_v n i) (\text{zip } \text{input-sizes } m \text{ ) is} \) ! \( i \) = \text{unit}_v
  (input-sizes \( m \) ! \( i \) ) (is ! \( i \))
  using length (input-sizes \( m \)) = length is by auto
  then have input-sizes \( m \) ! \( i \) = map dim-vec (base-input \( m \) is) ! \( i \)
unfolding base-input-def using index-unit-vec(3)
  using \( i < \text{length} \) (input-sizes \( m \)) \( \triangleq \text{length} \) (input-sizes \( m \)) = length (map dim-vec (base-input \( m \) is))
  base-input-def assms length-map nth-map valid-index-lt by (simp add: input-sizes-remove-weights)
}
  then show \( \forall i. \ i < \text{length} \) (input-sizes \( m \)) \( \Rightarrow \) input-sizes \( m \) ! \( i \) = map dim-vec
  (base-input \( m \) is) ! \( i \) by auto

47
lemma nth-mat-tensorlist-mult:
assumes ∏ A. A ∈ set T $ Ts ⟹ dims A = ds
assumes i < dim-row A
assumes dim-vec Ts = dim-col A
shows mat-tensorlist-mult A Ts ds $ i = listsum ds (map (λ j. (A $s (i, j)) · Ts $ j) [0..<dim-vec Ts])
(is - = listsum ds ?Ts')
proof (rule tensor-lookup-eqI)
  have dims-Ts': \[ T ∈ set ?Ts' \implies \text{dims } T = ds \]
  proof
  - fix T assume T ∈ set ?Ts'
    then obtain k where T = ?Ts' ! k and k < length ?Ts' k < dim-vec Ts using
      in-set-conv-nth by force
    show dims T = ds unfolding \( T = ?T's' ! k \) nth-map[OF \( k < \text{length } ?T's' \)\[\text{unfolded } \text{length-map}]\]
      using assms(1) (\< k < dim-vec Ts\)
      by (simp add: \< k < \text{length } (map (λ j. A $s (i, j)) · Ts $ j) [0..<dim-vec Ts]) vec-setI)
  qed
then show dims-eq:dims (mat-tensorlist-mult A Ts ds $ i) = dims (Tensor-Plus.listsum ds (map (λ j. A $s (i, j)) · Ts $ j) [0..<dim-vec Ts]) using
  dims-mat-tensorlist-mult assms mat-tensorlist-mult-def listsum-dims
  by (metis (no-types, lifting) dim-vec vec-setI)

fix is assume is-valid:is ∈ dims (mat-tensorlist-mult A Ts ds $ i)
then have is ∈ ds using dims-eq dims-Ts' listsum-dims by (metis (no-types, lifting))

have summand-eq: ∏ j. j ∈ {0..<dim-vec Ts} ⟹ row A i $ j * (map-vec (λ T. Tensor.lookup T is) T $ j) $ j = lookup (A $s (i, j)) · Ts $ j)
  using index-vec \< i < \text{dim-row } A \< \text{row-def } \< \text{dim-vec } T s = \text{dim-col } A\>
  \< \< i < \text{ds} \>\> assms(1) lookup-smult atLeastLessThan-iff index-map-vec(1) vec-setI
  by metis

have lookup (mat-tensorlist-mult A Ts ds $ i) is = (A $v (map-vec (λ T. Tensor.lookup T is)) T $ i)
  unfolding mat-tensorlist-mult-def using lookup-tensor-from-lookup[OF \< is ∈ \text{ds} \> using \< i < \text{dim-row } A \> by auto
  also have ... = row A i · map-vec (λ T. Tensor.lookup T is) T $ j
    using \< i < \text{dim-row } A \> by simp
  also have ... = (∑ j ∈ {0..<dim-vec Ts}. row A i $ j * (map-vec (λ T. Tensor.lookup T is) T $ j))
    unfolding scalar-prod-def nth-rows[OF \< i < \text{dim-row } A \> by simp
  also have ... = (∑ j∈{0..<dim-vec Ts}. lookup (A $s (i, j)) · Ts $ j) is using
    summand-eq by force
  also have ... = (∑ A←?Ts'. lookup A is) unfolding map-map
    Groups-List.sum-set-upt-conv-conv-list-nat[symmetric] atLeastLessThan-upt[symmetric]
by auto
also have \( ... = \text{lookup} \ (\text{listsum} \ ds \ ?Ts') \) is using \( \text{lookup-listsum} [\text{OF} \ (is < \ ds)] \)
dims-Ts' by fastforce
finally show \( \text{lookup} \ (\text{mat-tensorlist-mult} \ A \ Ts \ ds \ \$ \ i) \) is = \( \text{lookup} \ (\text{listsum} \ ds \ ?Ts') \) is by metis
qed

lemma lookup-tensors-from-net:
assumes valid-net' m
and \( is < \text{input-sizes} \ m \)
and \( j < \text{output-size'} m \)
shows \( \text{Tensor.lookup} (\text{tensors-from-net} \ m \ \$ \ j) \) is = \( \text{evaluate-net} \ m \) (base-input m is) \$ j
using assms proof (induction m arbitrary; j is)
case (Input M)
then have \( j < M \) using output-size.simps(1) using Input by auto
then have 1:_tensors-from-net (Input M) \$ j = unit-vec M \$ j by simp
obtain i where \( is = [i] \ i< M \) using Input Suc-length-conv input-sizes.simps(1)
length-0-one list.size(3) valid-index-length by auto
then have 2:Tensor.lookup (tensors-from-net (Input M) \$ j) is = (if i=j then 1 else 0) using lookup-unit-vec j by metis
have evaluate-net (Input M) (map (\( \lambda \ (n, i) \) unit_v n i) (zip (input-sizes (Input M)) is)) = unit_v M i using (is = [i]); by auto
then show \( \text{case using 2} \ (j < M) \) base-input-def by (simp add: (i < M))
next
case (Conv A m j is)
have is-valid: is < input-sizes m using Conv.prems by simp
have valid-net: valid-net' m using Conv.prems(1) unfolding remove-weights.simps
using valid-net.simps convnet.distinct(1) convnet.distinct(5) convnet.inject(2)
by blast
then have length-em: dim-vec (evaluate-net m (base-input m is)) = output-size' m

using output-size-correct base-input-length is-valid by metis
have IH':map-vec (\( \lambda T. \) Tensor.lookup T is) (tensors-from-net m) =
evaluate-net m (base-input m is)
proof (rule rule eq-vecI)
show equal-lengths: dim-vec (map-vec (\( \lambda T. \) lookup T is) (tensors-from-net m))
= dim-vec (evaluate-net m (base-input m is)) using length-em 
by (simp add: output-size-correct-tensors valid-net)
show \( \forall i. \ i < \text{dim-vec} \ (\text{evaluate-net} \ m \ (\text{base-input} \ m \ is)) \) \( \Rightarrow \)
map-vec (\( \lambda T. \) lookup T is) (tensors-from-net m) \$ i = evaluate-net m (base-input m is) \$ i
proof
fix i
assume i < dim-vec (evaluate-net m (base-input m is))
then have i < output-size' m using equal-lengths length-em by auto
then show map-vec (\( \lambda T. \) lookup T is) (tensors-from-net m) \$ i
= evaluate-net m (base-input m is) \$ i

49
using Conv.IH is-valid equal-lengths valid-net base-input-def length-em
nth-map-upt
length-map nth-map by auto
qed
qed

have Tensor.lookup ((tensors-from-net (Conv A m)) $ j) is =
(A *v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))) $ j
proof
have dim-vec (tensors-from-net (Conv A m)) = output-size' (Conv A m)
using Conv by (simp add: mat-tensorlist-mult-def)
then have j < dim-vec (tensors-from-net (Conv A m)) using Conv.prems by auto
then have (tensors-from-net (Conv A m)) $ j = tensor-from-lookup (input-sizes m)
(λis. (A *v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m)))) $ j
proof
    unfolding tensors-from-net.simps mat-tensorlist-mult-def by fastforce
    then show ?thesis using lookup-tensor-from-lookup[OF is-valid] by auto
qed

also have (A *v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))) $ j
= (A *v (evaluate-net m (base-input m is))) $ j using IH' by auto
also have ... = evaluate-net (Conv A m) (base-input (Conv A m) is) $ j
    unfolding base-input-def using evaluate-net.simps by auto
finally show ?thesis by auto
next
case (Pool m1 m2 j is)
    We split "is" into two parts for each subnet:
    obtain is1 is2 where is12-def:is = is1 @ is2 < input-sizes m1 is2 < input-sizes m2
    by (metis Pool.prems(2) input-sizes.simps(3) valid-index-split)
    Apply the induction hypothesis to the subnets:
    have IH:Tensor.lookup (tensors-from-net m1 $ j) is1
      = evaluate-net m1 (map (λ(x, y). unitv x y) (zip (input-sizes m1) is1)) $ j
    Tensor.lookup (tensors-from-net m2 $ j) is2
      = evaluate-net m2 (map (λ(x, y). unitv x y) (zip (input-sizes m2) is2)) $ j
    using Pool convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-weights.simps(3)
    valid-net.simps (is1 < input-sizes m1) (is2 < input-sizes m2) output-size.simps(3)
    by (metis base-input-def)+
    In the Pool layer tensor entries get multiplied:
    have lookup-prod: Tensor.lookup (tensors-from-net (Pool m1 m2) $ j) is
      = Tensor.lookup (tensors-from-net m1 $ j) is1 * Tensor.lookup (tensors-from-net m2 $ j) is2
    proof
      have j-small: j < dim-vec (tensors-from-net m1) j < dim-vec (tensors-from-net m2)
by \text{metis Pool.prems(1) Pool.prems(3) convnet.distinct(3) convnet.inject(3)}
convnet.simps(9)
output-size.simps(3) output-size-correct-tensors remove-weights.simps(3) valid-net.cases)+
then have 0:tensors-from-net \((Pool m1 m2) \otimes j) = tensors-from-net m1 \otimes j \otimes tensors-from-net m2 \otimes j

unfolding tensors-from-net.simps using j-small index-component-mult by
blast
have Tensor.dims \((tensors-from-net m1) \otimes j) = input-sizes m1
Tensor.dims \((tensors-from-net m2) \otimes j) = input-sizes m2
using dims-tensors-from-net j-small nth-mem by \text{(simp-all add: vec-setI)}
then have is12-valid:
  is1 < Tensor.dims \((tensors-from-net m1) \otimes j)
  is2 < Tensor.dims \((tensors-from-net m2) \otimes j)
using is12-def by presburger+
then show ?thesis
unfolding 0 using lookup-tensor-prod[OF is12-valid] is12-def by auto
qed

Output values get multiplied in the Pool layer as well:

have evaluate-net \((Pool m1 m2) \otimes base-input \((Pool m1 m2) \otimes is)) \otimes j
  = evaluate-net m1 \otimes base-input m1 is1 \otimes j \otimes evaluate-net m2 \otimes base-input m2 is2 \otimes j

proof -
  have valid-net’ m1 valid-net’ m2
    using remove-weights.simps valid-net.simps Pool.prems
    by \text{metis convnet.distinct(3) convnet.distinct(5) convnet.inject(3)}+
  have input-sizes m1 = map dim-vec \((base-input m1 is1))
    input-sizes m2 = map dim-vec \((base-input m2 is2))
  using base-input-def base-input-length base-input-def is12-def by auto
  have j < dim-vec \((evaluate-net m1 \otimes base-input m1 is1)) j < dim-vec \((evaluate-net m2 \otimes base-input m2 is2))
    using Pool.prems \text{\langle input-sizes m1 = map dim-vec \((base-input m1 is1)\rangle)}
  \langle valid-net’ m1\rangle
  output-size-correct by \text{(auto,metis Pool.prems(1) Pool.prems(3) \langle input-sizes m2 = map dim-vec \((base-input m2 is2)\rangle)}
    convnet.distinct(3) convnet.distinct(5) convnet.inject(3) output-size.simps(3)
  output-size-correct
  remove-weights.simps(3) valid-net.cases)
  then show ?thesis unfolding evaluate-net.simps unfolding base-input-def
    using is12-def(1) is12-def(2) valid-index-length by \text{(simp add: append-eq-conv-conj drop-map
    drop-zip index-component-mult input-sizes-remove-weights take-map take-zip)}
  qed

then show ?case using lookup-prod IH base-input-def by auto
qed

primrec extract-weights:
\text{\langle bool ⇒ real mat convnet ⇒ nat ⇒ real\rangle where}
extract-weights-Input: \text{extract-weights shared \((Input M) = (\lambda x. 0)\)

51
extract-weights-Conv: extract-weights shared (Conv A m) =
(λx. if x < dim-row A * dim-col A then flatten-matrix A x
else extract-weights shared m (x − dim-row A * dim-col A))

extract-weights-Pool: extract-weights shared (Pool m1 m2) =
(λx. if x < count-weights shared (remove-weights m1)
then extract-weights shared m1 x
else extract-weights shared m2 (x − count-weights shared (remove-weights m1)))

inductive balanced-net::(nat × nat) convnet ⇒ bool where
balanced-net-Input: balanced-net (Input M)
| balanced-net-Conv: balanced-net m ⇒ balanced-net (Conv A m)
| balanced-net-Pool: balanced-net m1 ⇒ balanced-net m2 ⇒
count-weights True m1 = count-weights True m2 ⇒ balanced-net (Pool m1 m2)

inductive shared-weight-net::real mat convnet ⇒ bool where
shared-weight-net-Input: shared-weight-net (Input M)
| shared-weight-net-Conv: shared-weight-net m ⇒ shared-weight-net (Conv A m)
| shared-weight-net-Pool: shared-weight-net m1 ⇒ shared-weight-net m2 ⇒
count-weights True (remove-weights m1) = count-weights True (remove-weights m2) ⇒
(∀x. x < count-weights True (remove-weights m1) ⇒ extract-weights True m1 x = extract-weights True m2 x)
⇒ shared-weight-net (Pool m1 m2)

lemma insert-extract-weights-cong-shared:
assumes shared-weight-net m
assumes ∀x. x < count-weights True (remove-weights m) ⇒ f x = extract-weights True m x
shows m = insert-weights True (remove-weights m) f
using assms proof (induction m arbitrary; f)
case (shared-weight-net-Input M)
  then show ?case
    by simp
next
case (shared-weight-net-Conv m A)
have extract-matrix f (dim-row A) (dim-col A) = A
  by (simp add: extract-matrix-cong extract-matrix-flatten-matrix shared-weight-net-Conv.prems)
  then show ?case
    using shared-weight-net-Conv.IH[of (λi. f (i + dim-row A * dim-col A))]
    using shared-weight-net-Conv.prems by auto
next
case (shared-weight-net-Pool m1 m2)
have m1 = insert-weights True (remove-weights m1) f
  using shared-weight-net-Pool.IH(1) shared-weight-net-Pool.prems by auto
have m2 = insert-weights True (remove-weights m2) f
  using local.shared-weight-net-Pool(3) shared-weight-net-Pool.IH(2)
  shared-weight-net-Pool.hyps(4) shared-weight-net-Pool.prems by fastforce

52
then show ?case
  using \( m1 = \text{insert-weights} \ True (\text{remove-weights} \ m1) \ f \) by auto
qed

lemma insert-extract-weights-cong-unshared:
assumes \( \forall x. \ x < \text{count-weights} \ False (\text{remove-weights} \ m) \implies f \ x = \text{extract-weights} \ False \ m \ x \)
shows \( m = \text{insert-weights} \ False (\text{remove-weights} \ m) \ f \)
using assms proof (induction \( m \) arbitrary; \( f \))
  case (Input \( M \))
    then show ?case
      by simp
  next
    case (Conv \( A \) \( m \))
    then have \( \text{extract-matrix} \ f \ (\text{dim-row} \ A) (\text{dim-col} \ A) = A \)
      by (metis \( \text{count-weights} \simps(2) \text{extract-matrix-flatten-matrix-cong} \text{extract-weights-Conv} \text{remove-weights} \simps(2) \text{trans-less-add1} \))
    then show ?case
      using Conv.IH Conv.prems by auto
  next
    case (Pool \( m1 \) \( m2 \))
    then show ?case
      using Pool.IH(1) Pool.IH(2) Pool.prems by auto
qed

lemma remove-insert-weights:
shows \( \text{remove-weights} (\text{insert-weights} \ s \ m \ w) = m \)
proof (induction \( m \) arbitrary; \( w \))
  case Input
    then show ?case by simp
next
  case (Conv \( r12 \) \( m \))
  then obtain \( r1 \) \( r2 \) where \( r12 = (r1, r2) \) by fastforce
  then have \( \text{remove-weights} (\text{insert-weights} \ s \ m \ w) = m \) using Conv.IH by blast
  then have \( \text{remove-weights} (\text{insert-weights} \ s (\text{Conv} \ (r1, r2) \ m) \ w) = \text{Conv} \ (r1, r2) \ m \)
    unfolding \( \text{insert-weights} \simps \text{remove-weights} \simps \text{extract-weights} \simps \)
    using extract-matrix-def Conv.IH dim-extract-matrix(1) by (metis \( \text{dim-col-mat}(1) \))
  then show ?case using \( r12 = (r1, r2) \) by blast
next
  case (Pool \( m1 \) \( m2 \) \( w \))
  then show ?case unfolding \( \text{insert-weights} \simps \text{remove-weights} \simps \text{extract-weights} \simps \) using Pool.IH by blast
qed

lemma extract-insert-weights-shared:
assumes \( x < \text{count-weights} \ True \ m \)
and \( \text{balanced-net} \ m \)
shows extract-weights True (insert-weights True m w) x = w x
using assms
proof (induction m arbitrary: w x)
case (Input x)
  then show ?case by simp
next
case (Conv r01 m)
  obtain r0 r1 where r01 = (r0, r1) by force
  then show ?case unfolding \( r01 = (r0, r1) \).
next
case (Pool m1 m2)
  then obtain r0 r1 where r01 = (r0, r1) by force
  then show ?case unfolding insert-weights.simps extract-weights.simps
applying cases x < dim-row (extract-matrix w r0 r1) * dim-col (extract-matrix w r0 r1)
applying (auto simp add: dim-extract-matrix(1) dim-extract-matrix(2) flatten-matrix-extract-matrix)
using Conv.IH[of - \( \lambda i. w(i + r0 * r1) \)] Conv.prems(1) Conv.prems(2) r01 = (r0, r1) balanced-net.cases by force
next
case (Pool m1 m2)
  then have balanced-net m1 balanced-net m2 using Pool.prems balanced-net.simps by blast+
  have \( \forall x. x < \text{count-weights True m1} \implies \) extract-weights True (insert-weights True m1 w) x = extract-weights True (insert-weights True m2 w) x
applying (metis Pool.prems balanced-net.simps convnet.distinct(3) convnet.distinct(5) convnet.inject(3))

lemma shared-weight-net-insert-weights: balanced-net m \implies \text{shared-weight-net (insert-weights True m w)}
proof (induction m arbitrary: w)
case (Input x)
  then show ?case using insert-weights.simps balanced-net.simps shared-weight-net.simps by metis
next
case (Conv r01 m)
  then obtain r0 r1 where r01 = (r0, r1) by force
  then show ?case unfolding \( r01 = (r0, r1) \).
next
case (Pool m1 m2)
  have balanced-net m1 balanced-net m2 using Pool.prems balanced-net.simps by blast+
  have \( \forall x. x < \text{count-weights True m1} \implies \) extract-weights True (insert-weights True m1 w) x = extract-weights True (insert-weights True m2 w) x
applying (metis Pool.prems balanced-net.simps convnet.distinct(3) convnet.distinct(5) convnet.inject(3))

54
then show \(\text{?case unfolding insert-weights.simps using Pool}(1)[\text{of } w] \text{ Pool}(2)[\text{of } w]\)
\[\text{by (metis Pool.prems balanced-net.simps convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-insert-weights shared-weight-net-Pool)}\]
\text{qed}

\text{lemma finite-valid-index: finite \{is. is < ds\}}
\text{proof (induction ds)}
\text{ case Nil then show ?case by (metis List.finite-set finite-subset length-0-conv list.set-intros(1) mem-Collect-eq subsetI valid-index-length)}
\text{ next case (Cons d ds)}
\text{ have \{is. is < d \# ds\} \subseteq (\bigcup i<.d. \{i \# is | is. is < ds\})}
\text{ proof (rule subsetI)}
\text{ fix is assume is \in \{is. is < d \# ds\}}
\text{ then have is < d \# ds by auto}
\text{ then obtain i is’ where is = i \# is’ by blast}
\text{ then have i<.d using \langle is < d \# ds\rangle by blast}
\text{ have is’ < ds using \langle is = i \# is’ \# is’ < ds\rangle by blast}
\text{ have is \in \{i \# is | is. is < ds\} by (simp add: \langle is = i \# is’ \# is’ < ds\rangle)}
\text{ then show is \in (\bigcup i<.d. \{i \# is | is. is < ds\}) using \langle i < d\rangle by blast}
\text{ qed}
\text{ moreover have \(\exists i. \) finite \{i \# is | is. is < ds\} by (simp add: Cons.IH)}
\text{ ultimately show finite \{is. is < d \# ds\} by (simp add: finite-subset)}
\text{ qed}

\text{lemma setsum-valid-index-split:}
\(\sum is \mid is < ds1 @ ds2. f \langle is \rangle = (\sum is1 \mid is1 < ds1. (\sum is2 \mid is2 < ds2. f \langle is1 @ is2\rangle))\)
\text{ proof –}
\text{ have 1:\(\langle \lambda(is1, is2). is1 @ is2 \\rangle \cdot (\{is1. is1 < ds1\} \times \{is2. is2 < ds2\}) = \{is. is < ds1 @ ds2\}\langle is ?A = ?B\rangle)}
\text{ proof (rule subset-antisym; rule subsetI)}
\text{ fix x assume x \in ?A}
\text{ then show x \in ?B using valid-index-append by auto}
\text{ next fix x assume x \in ?B}
\text{ then have x < ds1 @ ds2 by auto}
\text{ then obtain x1 x2 where x = x1 @ x2 x1 < ds1 x2 < ds2 by (metis valid-index-split)}
\text{ then have \langle x1, x2\rangle \in (\{is1. is1 < ds1\} \times \{is2. is2 < ds2\}) by auto}
\text{ then show x \in ?A using imageI \langle x = x1 @ x2\rangle by blast}
\text{ qed}
\text{ have 2:\langle inj-on \langle \lambda(is1, is2). is1 @ is2 \\rangle \cdot (\{is1. is1 < ds1\} \times \{is2. is2 < ds2\}) by (simp add: inj-on-def valid-index-length)
\text{ show \?thesis unfolding Groups-Big.comm-monoid-add-class.sum.cartesian-product[of \lambda is1 is2. f \langle is1 @ is2\rangle]}\]
using Groups-Big.comm-monoid-add-class.sum.reindex[OF 2, of f] 1
2 SigmaE prod.simps(2) sum.reindex-cong by (simp add: split-def)

qed

lemma prod-lessThan-split:
fixes g :: nat ⇒ real shows prod g {..<n+m} = prod g {..<n} * prod (λx. g
(x+n)) {..<m}
using Groups-Big.comm-monoid-mult-class.prod.union-inter-neutralf{..<n} {n..<n+m}
g, unfolded iivl-disj-an-one(2)[OF le-add1], OF finite-lessThan finite-atLeastLessThan
by (metis (no-types) add.commute add.left-neutral atLeast0LessThan empty-iff iivl-disj-int-one(2)
prod.shift-bounds-nat-ivl)

lemma evaluate-net-from-tensors:
assumes valid-net' m
and map dim-vec inputs = input-sizes m
and j < output-size' m
shows evaluate-net m inputs $ j
= (∑ is∈{is. is < input-sizes m}. (∏ k<length inputs. inputs ! k $(is)k) *
Tensor.lookup (tensors-from-net m $ j) is)
using assms proof (induction m arbitrary: j is inputs)
case (Input M)
then have length inputs = 1 input-sizes (Input M) = [M] by auto
{
  fix is assume is < input-sizes (Input M)
  then have length is = 1 by (simp add: valid-index-length)
  then have is = [hd is] by (metis One-nat-def length-0-conv length-Suc-conv
list.sel(1))
  then have Tensor.lookup (tensors-from-net (Input M) $ j) is = (if hd is=j
then 1 else 0)
  by (metis Input.prems(3) input-sizes (Input M) = [M] ‹is < input-sizes
(Input M)› list.distinct(1)
lookup-unit-vec nth-Cons-0 output-size.simps(1) remove-weights.simps(1) ten-
sors-from-net.simps(1) valid-indexE index-vec)
  then have (∏ k<length inputs. inputs ! k $(is)k) * lookup (tensors-from-net
(Input M) $ j) is =
  (if is=[j] then (∏ k<length inputs. inputs ! k $(is)k) else 0) using
  (is = [hd is]) by auto
}
then have (∑ is | is < input-sizes (Input M). (∏ k<length inputs. inputs ! k $(
is)k) *
lookup (tensors-from-net (Input M) $ j) is)
= (∑ is | is < input-sizes (Input M). (if is=[j] then (∏ k<length inputs. inputs
! k $(is)k) else 0)) by auto
also have (∑ is | is < input-sizes (Input M). (if is=[j] then (∏ k<length inputs. inputs
! k $(is)k) else 0))
= (∏ k<length inputs. inputs ! k $(jj)k) unfolding sum.delta[OF finite-valid-index]
using Input.prems(3) valid-index.Cons valid-index.Nil by auto
also have ... = inputs ! 0 $ j using ‹length inputs = 1› by (simp add: prod.lessThan-Suc)
also have ... = evaluate-net (Input M) inputs $ j unfolding evaluate-net.simps
by (metis \(\text{length inputs} = 1\) \& \text{hd-conv-nth list.size(3)} \& \text{zero-neq-one})

finally show \(?\text{case by auto}\)

next

case (Conv A \(m\) \(j\))

have \(j < \text{dim-row } A\) using Conv.prems(\(3\)) \&\& auto

have \(0: \{\text{is } < \text{input-sizes } (\text{Conv A } m) \rightarrow (\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } (\text{Conv A } m))\} \& j\) \text{ is } =

\((\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A \ j \ \ i \ast ((\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } m \ \ i \ (\text{is } ! k))))\)

proof

fix is assume is \(<\text{input-sizes } (\text{Conv A } m)\)

then have is \(<\text{input-sizes } m\) by simp

have \(0: \text{lookup } (\text{tensors-from-net } (\text{Conv A } m) \ \ j)\) \text{ is } =

\((\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A \ j \ \ i \ast \text{lookup } (\text{tensors-from-net } m \ \ i \ (\text{is } ! k))\)

unfolding tensors-from-net.simps mat-tensorlist-mult-def index-vec[of \(j<\text{dim-row } A\)]

lookup-tensor-lookup[of \(\text{is } < \text{input-sizes } m\)] index-mat-mult-vector[of \(\text{is } < \text{dim-row } A\)] scalar-prod-def

using index-map-vec by auto

show \((\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } (\text{Conv A } m) \ \ j)\) \text{ is } =

\((\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A \ j \ \ i \ast ((\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } m \ \ i \ (\text{is } ! k))))\)

unfolding 0 sum-distrib-left by (simp add: semiring-normalization-rules(\(19\)))

qed

have valid-net' \(m\) by (metis Conv.prems(\(1\)) connet.distinct(\(1\)) connet.distinct(\(5\)) connet.inject(\(2\)) remove-weights.simps(\(2\)) valid-net.simps)

have map \text{dim-vec inputs } = \text{input-sizes } m \text{ by } (simp add: Conv.prems(\(2\)))

have \text{output-size' } m = \text{dim-vec } (\text{tensors-from-net } m) \text{ by } (simp add: \text{valid-net' } m \text{ \&\& \text{output-size-correct-tensors})}

have \(1: \{\text{is } < \text{dim-vec } (\text{tensors-from-net } m) \Rightarrow (\sum \text{is } < \text{input-sizes } (\text{Conv A } m). ((\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } m \ \ i \ (\text{is } ! k)))) \text{ is } = \text{evaluate-net } m \text{ inputs } \ \ i \ \ \text{unfolding } \text{input-sizes.simps})\}

using Conv.INH [valid-net' \(m\)] map \text{dim-vec inputs } = \text{input-sizes } m \text{ \&\& \text{output-size' } m = \text{dim-vec } (\text{tensors-from-net } m)\} \text{ by simp}

have \((\sum \text{is } < \text{input-sizes } (\text{Conv A } m). ((\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } m \ \ i) \ (\text{is } ! k)))) \text{ is } = \text{evaluate-net } m \text{ inputs } \ \ i \ \ \text{unfolding } \text{input-sizes.simps})\)

using Groups-Big.comm-monoid-add-class.sum.swap 0 by auto

also have \(\ldots = (\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A \ j \ \ i \ast ((\sum \text{is } < \text{input-sizes } (\text{Conv A } m). ((\prod k < \text{length inputs. inputs } ! k \ (\text{is } ! k)) \ast \text{lookup } (\text{tensors-from-net } m \ \ i) \ (\text{is } ! k))))\)

by (simp add: sum-distrib-left)

also have \(\ldots = (\sum i = 0..<\text{dim-vec } (\text{tensors-from-net } m). \text{row } A \ j \ \ i \ast \text{evalu-} \)
ate-net \( m \) inputs $i$ using 1 by auto
also have ... = row A j • evaluate-net \( m \) inputs
  by (metis (full-types) (map dim-vec inputs = input-sizes \( m \)) • output-size' \( m \) =
  dim-vec (tensors-from-net \( m \))
  (valid-net' \( m \)) output-size-correct scalar-prod-def)
also have ... = (A *@v evaluate-net \( m \) inputs) $j$ by (simp add: \( \bullet j < \) dim-row \( A \))
also have ... = evaluate-net (Conv A \( m \)) inputs $j$ by simp
finally show ?case by auto

next
case (Pool \( m1 m2 j \))
have valid-net' \( m1 \) valid-net' \( m2 \)
  by (metis Pool.prems(1) convnet.distinct(3) convnet.inject(3) convnet.simps(9)
remove-weights.simps(3) valid-net.simps)
  have \( j < \) output-size' \( m2 j < \) output-size' \( m1 \)
  apply (metis Pool.prems(1) Pool.prems(3) convnet.distinct(3) convnet.inject(3)
convnet.simps(9)
output-size.simps(3) remove-weights.simps(3) valid-net.simps) using Pool.prems
by auto
then have \( j < \) dim-vec (tensors-from-net \( \text{m1} \)) \( j < \) dim-vec (tensors-from-net \( \text{m2} \))
  by (simp-all add: (valid-net' \( \text{m1} \)) (valid-net' \( \text{m2} \)) output-size-correct-tensors)

define inputs1 where inputs1 = take (length (input-sizes \( \text{m1} \))) inputs
define inputs2 where inputs2 = drop (length (input-sizes \( \text{m1} \))) inputs
have map dim-vec inputs1 = input-sizes \( \text{m1} \) map dim-vec inputs2 = input-sizes \( \text{m2} \)
  apply (metis Pool.prems(2) append-eq-conv-conj input-sizes.simps(3) inputs1-def take-map)
  by (metis Pool.prems(2) append-eq-conv-conj drop-map input-sizes.simps(3)
inputs2-def)
have inputs = inputs1 @ inputs2 by (simp add: inputs1-def inputs2-def)
{ 
  fix \( \text{is1} \) \( \text{is2} \) assume \( \text{is1} \) $\triangleleft$ input-sizes \( \text{m1} \) \( \text{is2} \) $\triangleleft$ input-sizes \( \text{m2} \)
  have length \( \text{is1} \) = length inputs1
    using (\( \text{is1} \) $\triangleleft$ input-sizes \( \text{m1} \)) (map dim-vec inputs1 = input-sizes \( \text{m1} \))
valid-index-length by fastforce
  have length \( \text{is2} \) = length inputs2
    using (\( \text{is2} \) $\triangleleft$ input-sizes \( \text{m2} \)) (map dim-vec inputs2 = input-sizes \( \text{m2} \))
valid-index-length by fastforce
  have 1:$(\prod k < \text{length inputs1. (inputs1 @ inputs2)} ! k \bullet ((\text{is1} @ \text{is2}) ! k)) =
(\prod k < \text{length inputs1. inputs1} ! k \bullet (\text{is1} ! k))$
    using (length \( \text{is1} \) = length inputs1) (length \( \text{is2} \) = length inputs2)
nth-append by (metis (no-types, lifting) lessThan-iff prod.cong)
  have 2:$(\prod x < \text{length inputs2. (inputs1 @ inputs2)} ! (x + \text{length inputs1}) \bullet ((\text{is1} @ \text{is2}) ! (x + \text{length inputs1})))) =
(\prod k < \text{length inputs2. inputs2} ! k \bullet (\text{is2} ! k))$
    using (length \( \text{is1} \) = length inputs1) (length \( \text{is2} \) = length inputs2)
by (metis (no-types, lifting) add.commute nth-append-length-plus)
\[
\text{have } \prod k < \text{length inputs}, \text{inputs}! k \in ((\text{is1} \oplus \text{is2})! k)) = (\prod k < \text{length inputs1}, \text{inputs1}! k \in (\text{is1} \oplus \text{is2})! k) * (\prod k < \text{length inputs2}, \text{inputs2}! k \in (\text{is2} \oplus \text{is2})! k))
\]

\[
\text{unfolding } (\text{inputs} = \text{inputs1} \oplus \text{inputs2}) \text{ length-append prod-lessThan-split using 1 2 by metis}
\]

\[
\text{note 1 = this}
\]

\[
\text{fix is1 is2 assume is1 < \text{input-sizes m1} is2 < \text{input-sizes m2}}
\]

\[
\text{then have is1 < \text{dims (tensors-from-net m1} $ j $) is2 < \text{dims (tensors-from-net m2} $ j $)
\]

\[
\text{using } j < \text{dim-vec (tensors-from-net m1)} \text{ j < dim-vec (tensors-from-net m2) by force+}
\]

\[
\text{have lookup (tensors-from-net (Pool m1 m2) $ j $) (is1 $ @ $ is2) = lookup (tensors-from-net m1 $ j $) is1 * lookup (tensors-from-net m2 $ j $) is2}
\]

\[
\text{unfolding tensors-from-net.simps index-component-mult [OF j < dim-vec (tensors-from-net m1) j < dim-vec (tensors-from-net m2)]}
\]

\[
\text{lookup-tensor-prod[OF is1 < dims (tensors-from-net m1 $ j $) is2 < dims (tensors-from-net m2 $ j $)] by metis}
\]

\[
\text{note 2 = this}
\]

\[
\text{have j-le-eval: j < dim-vec (evaluate-net m1 (take (length (input-sizes m1)) inputs))}
\]

\[
\text{using j < dim-vec (evaluate-net m2 (drop (length (input-sizes m1)) inputs))}
\]

\[
\text{using j < output-size'} m1 \langle \text{map dim-vec inputs1 = input-sizes m1} \langle \text{valid-net'} m1 \langle \text{inputs1-def output-size-correct}
\]

\[
\text{using j < output-size'} m2 \langle \text{map dim-vec inputs2 = input-sizes m2} \langle \text{valid-net'} m2 \langle \text{inputs2-def by auto}
\]

\[
\text{have } (\sum \text{i is } \langle \text{is < input-sizes (Pool m1 m2)} (\prod k < \text{length inputs, inputs}! k \in (\text{is1} \oplus \text{is2})! k)) * \text{lookup (tensors-from-net (Pool m1 m2) $ j $) is1} = (\sum \text{i is1 < input-sizes m1. \sum i is2 < input-sizes m2})
\]

\[
(\prod k < \text{length inputs1, inputs1}! k \in (\text{is1} \oplus \text{is2})! k) * (\prod k < \text{length inputs2, inputs2}! k \in (\text{is2} \oplus \text{is2})! k)) * \text{lookup (tensors-from-net m2 $ j $) is2}
\]

\[
\text{unfolding input-sizes.simps sets-sum-valid-index-split using 1 2}
\]

\[
\text{using mem-Collect-eq sum.cong by (simp add: mult.assoc)}
\]

\[
\text{also have ...} = (\sum \text{i is1 < input-sizes m1. (\prod k < \text{length inputs1, inputs1}! k \in (\text{is1} \oplus \text{is2})! k)) * \text{lookup (tensors-from-net m1} $ j $ \text{is1}) * (\sum \text{i is2 < input-sizes m2. (\prod k < \text{length inputs2, inputs2}! k \in (\text{is2} \oplus \text{is2})! k)) * \text{lookup (tensors-from-net m2} $ j $ \text{is2})}
\]

\[
\text{unfolding sum-product by (rule sum.cong, metis, rule sum.cong, metis, simp)}
\]

\[
\text{also have ... = evaluate-net (Pool m1 m2) inputs} $ j $ \text{unfolding evaluate-net.simps index-component-mult [OF j-le-eval]}
\]

\[
\text{using Pool.IH(1)[OF \langle \text{valid-net'} m1 \langle - j < \text{output-size'} m1]} \text{ Pool.IH(2)[OF \langle \text{valid-net'} m2 \langle - j < \text{output-size'} m2]}
\]

\[
\text{using \langle \text{map dim-vec inputs1 = input-sizes m1} \langle \text{map dim-vec inputs2 = input-sizes m2} \langle \text{inputs1-def inputs2-def by auto}
\]

\[
\text{finally show ?case by metis}
\]

59
lemma tensors-from-net-eqI:
assumes valid-net' m1 valid-net' m2 input-sizes m1 = input-sizes m2
assumes \( \forall \) inputs. input-sizes m1 = map dim-vec inputs \( \implies \) evaluate-net m1 inputs = evaluate-net m2 inputs
shows tensors-from-net m1 = tensors-from-net m2
proof –
  have map dim-vec (map 0 v (input-sizes m2)) = input-sizes m2 
  map dim-vec (map 0 v (input-sizes m1)) = input-sizes m1 by (auto intro: nth-equalityI)
  then have output-size' m1 = output-size' m2 using
    output-size-correct[OF \( \langle \text{valid-net'} m1 \rangle \langle \text{map dim-vec (map 0 v (input-sizes m1))} \) = input-sizes m1;]
    output-size-correct[OF \( \langle \text{valid-net'} m2 \rangle \langle \text{map dim-vec (map 0 v (input-sizes m2))} \) = input-sizes m2;]
    assms(3) assms(4)
    by (metis (no-types))
  have \( \forall is. \) base-input m1 is = base-input m2 is
    unfolding base-input-def \( \langle \text{input-sizes m1} = \text{input-sizes m2} \rangle \) by metis
  show ?thesis by (rule eq-vecI, rule tensor-lookup-eqI; metis
    lookup-tensors-from-net[OF \( \langle \text{valid-net'} m1 \rangle \langle \text{output-size'} m1 = output-size' m2 \rangle \) \( \forall is. \) base-input m1 is = base-input m2 is \( \langle \text{output-size'} m1 = output-size' m2 \rangle \)
    lookup-tensors-from-net[OF \( \langle \text{valid-net'} m2 \rangle \langle \text{output-size'} m1 = output-size' m2 \rangle \) assms(3) base-input-length
    assms(1) assms(2) dims-tensors-from-net output-size-correct-tensors vec-setI
    \( \langle \text{output-size'} m1 = output-size' m2 \rangle \) assms(4))
qed

end

12 Concrete Matrices

theory DL-Concrete-Matrices
imports Jordan-Normal-Form.Matrix
begin

The following definition allows non-square-matrices, mat_one (mat_one n) only allows square matrices.
definition id-matrix::nat \( \Rightarrow \) nat \( \Rightarrow \) real mat
where id-matrix nr nc = mat nr nc (\( \lambda \) (r, c). if r=c then 1 else 0)

lemma id-matrix-dim: dim-row (id-matrix nr nc) = nr dim-col (id-matrix nr nc) = nc by (simp-all add: id-matrix-def)

lemma row-id-matrix:
assumes i < nr
shows row (id-matrix nr nc) i = unit-vec nc i
  by (rule eq-vecI, simp add: assms id-matrix-def unit-vec-def, simp add: id-matrix-dim(2))
lemma unit-eq-0[simp]:
assumes i: i ≥ n
shows unit-vec n i = 0
by (rule eq-vecI, insert i, auto simp: unit-vec-def)

lemma mult-id-matrix:
assumes i < nr
shows (id-matrix nr (dim-vec v) ∗ v) $ i = (if i < dim-vec v then v $ i else 0) (is
?a $ i = ?b)
proof –
  have ?a $ i = row (id-matrix nr (dim-vec v)) i · v using index-mult-mat-vec
assms id-matrix-dim by auto
  also have ... = unit-vec (dim-vec v) i · v using row-id-matrix assms by auto
  also have ... = ?b using scalar-prod-left-unit carrier-vecI unit-eq-0 scalar-prod-left-zero
by fastforce
  finally show ?thesis by auto
qed

definition all1-vec::nat ⇒ real vec
where all1-vec n = vec n (λi. 1)
definition all1-matrix::nat ⇒ nat ⇒ real mat
where all1-matrix nr nc = mat nr nc (λ(r, c). 1)

lemma all1-matrix-dim: dim-row (all1-matrix nr nc) = nr dim-col (all1-matrix nr nc) = nc
  by (simp-all add: all1-matrix-def)

lemma row-all1-matrix:
assumes i < nr
shows row (all1-matrix nr nc) i = all1-vec nc
apply (rule eq-vecI)
apply (simp add: all1-matrix-def all1-vec-def assms)
by (simp add: all1-matrix-def all1-vec-def)

lemma all1-vec-scalar-prod:
shows all1-vec (length xs) · (vec-of-list xs) = sum-list xs
proof –
  have all1-vec (length xs) · (vec-of-list xs) = (∑ i = 0..<length xs. vec-of-list xs).vec-of-list xs § i)
unfolding scalar-prod-def by (metis (no-types, lifting) all1-vec-def mult-cancel-right1
sum.iel-cong
  vec.abs-eq dim-vec index-vec vec-of-list.abs-eq)
also have ... = (∑ i = 0..<length xs. xs ! i) using vec.abs-eq dim-vec vec-of-list.abs-eq
  by (metis sum.iel-cong index-vec)
also have ... = sum-list xs by (simp add: sum-list-sum-nth)
finally show ?thesis by auto
qed
lemma mult-all1-matrix:
assumes $i < nr$
shows $((all1-matrix nr (dim-vec v)) \ast_v v) \Sigma i = \text{sum-list (list-of-vec v)}$ (is $\forall i \in \text{list-of-vec v}$)

proof –
  have $\forall i \in \text{list-of-vec v}$ unfolding all1-matrix[OF assms]
  using all1-vec-scalar-prod[of list-of-vec v]
  by (metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq)
finally show $\forall i \in \text{list-of-vec v}$ unfolding all1-matrix[OF assms]
using all1-vec-scalar-prod[of list-of-vec v]
  by (metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq)
qed

definition copy-first-matrix :: nat \Rightarrow nat \Rightarrow real mat
where $\forall nr \in \text{nat}, nc \in \text{real mat}$ (\lambda (r, c). if $c = 0$ then 1 else 0)

lemma copy-first-matrix-dim: dim-row (copy-first-matrix nr nc) = nr
  dim-col (copy-first-matrix nr nc) = nc
by (simp-all add: copy-first-matrix-def)

lemma row-copy-first-matrix:
assumes $i < nr$
shows $\forall i \in \text{list-of-vec v}$ unfolding all1-matrix[OF assms]
using all1-vec-scalar-prod[of list-of-vec v]
  by (metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq)
qed

lemma mult-copy-first-matrix:
assumes $i < nr$ and dim-vec v > 0
shows $(copy-first-matrix nr (dim-vec v)) \ast_v v \Sigma i = v \Sigma 0$ (is $\forall i \in \text{list-of-vec v}$)

proof –
  have $\forall i \in \text{list-of-vec v}$ unfolding all1-matrix[OF assms]
  using all1-vec-scalar-prod[of list-of-vec v]
  by (metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq)
finally show $\forall i \in \text{list-of-vec v}$ unfolding all1-matrix[OF assms]
using all1-vec-scalar-prod[of list-of-vec v]
  by (metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq)
qed

13 Missing Lemmas of Finite_Set

theory DL-Missing-Finite-Set
imports Main

62
begin

lemma card-even\{simp\}: card \{a ∈ Collect even. a < 2 * n\} = n
proof (induction n)
case 0
then show ?case by auto
next
case (Suc n)
have \{a ∈ Collect even. a < 2 * Suc n\} = insert (2*n) \{a ∈ Collect even. a < 2 * n\}
  using le-eq-less-or-eq less-Suc-eq-le subset-antisym by force
show ?case
  unfolding \{a ∈ Collect even. a < 2 * Suc n\} = insert (2*n) \{a ∈ Collect even. a < 2 * n\}
  using Suc card-insert-disjoint[of \{a ∈ Collect even. a < 2 * n\} 2*n]
  by (simp add: finite-M-bounded-by-nat less-not-refl2)
qed

lemma card-odd\{simp\}: card \{a ∈ Collect odd. a < 2 * n\} = n
proof (induction n)
case 0
then show ?case by auto
next
case (Suc n)
have \{a ∈ Collect odd. a < 2 * Suc n\} = insert (2*n+1) \{a ∈ Collect odd. a < 2 * n\}
  using le-eq-less-or-eq less-Suc-eq-le subset-antisym by force
show ?case
  unfolding \{a ∈ Collect odd. a < 2 * Suc n\} = insert (2*n+1) \{a ∈ Collect odd. a < 2 * n\}
  using Suc card-insert-disjoint[of \{a ∈ Collect even. a < 2 * n\} 2*n]
  by (simp add: finite-M-bounded-by-nat less-not-refl2)
qed

end

14 Deep Network Model

theory DL-Deep-Model
imports DL-Network Tensor-Matricization Jordan-Normal-Form.DL-Submatrix DL-Concrete-Matrices
begin
hide-const(open) Polynomial.order
hide-const(open) Matrix.unit-vec

fun deep-model and deep-model’ where
depth model’ Y [] = Input Y |
deep-model' Y (r ≠ rs) = Pool (deep-model Y r rs) (deep-model Y r rs) | 
deep-model Y r rs = Conv (Y,r) (deep-model' r rs)

abbreviation deep-model'-l rs == deep-model' (rs!0) (tl rs)
abbreviation deep-model-l rs == deep-model (rs!0) (rs!1) (tl (tl rs))

lemma valid-deep-model: valid-net (deep-model Y r rs)
apply (induction rs arbitrary: Y r)
apply (simp add: valid-net.intros(1) valid-net.intros(2))
using valid-net.intros(2) valid-net.intros(3) by auto

lemma valid-deep-model': valid-net (deep-model' r rs)
apply (induction rs arbitrary: r)
apply (simp add: valid-net.intros(1))
by (metis deep-model'.elims deep-model'.simps(2) deep-model.elims output-size.simps valid-net.simps)

lemma input-sizes-deep-model':
assumes length rs ≥ 1
shows input-sizes (deep-model'-l rs) = replicate (2^\((\text{length } rs - 1)\)) (last rs)
using assms proof (induction butlast rs arbitrary:rs)
case Nil
  then have rs = [rs!0]
    by (metis One-nat-def diff-diff-cancel diff-zero length-0-conv length-Suc-conv
        length-butlast nth-Cons-0)
  then have input-sizes (deep-model'-l rs) = [last rs]
    by (metis deep-model'.simps(1) input-sizes.simps(1) last.simps list.sel(3))
  then show input-sizes (deep-model'-l rs) = replicate (2 ^ (\text{length } rs - 1)) (last rs)
    by (metis One-nat-def (\text{[]} = \text{butlast } rs) empty-replicate length-butlast list.size(3)
        power-0 replicate.simps(2))
next
case (Cons r rs')
  then have IH: input-sizes (deep-model'-l (tl rs)) = replicate (2 ^ (\text{length } (tl rs) - 1)) (last rs)
    by (metis (no-types, lifting) One-nat-def butlast-tl diff-is-0-eq' last-tl length-Cons
        length-butlast length-tl list.sel(3) list.size(3) nat-le-linear not-one-le-zero)
  have rs = r ≠ (tl rs) by (metis Cons.hyps(2) Cons.prems append-Cons append-butlast-last-id length-greater-0-conv less-le-trans list.sel(3) zero-less-Suc)
  then have deep-model'-l rs = Pool (deep-model-l rs) (deep-model-l rs)
    by (metis Cons.hyps(2) One-nat-def butlast.simps(2) deep-model'.elims list.sel(3)
        list.simps(3) nth-Cons-0 nth-Cons-Suc)
  then have input-sizes (deep-model'-l rs) = input-sizes (deep-model-l rs) @ input-sizes (deep-model-l rs)
    using input-sizes.simps(3) by metis
  also have ... = input-sizes (deep-model'-l (tl rs)) @ input-sizes (deep-model'-l (tl rs))
    by (metis (no-types, lifting) Cons.hyps(2) One-nat-def deep-model.elims input-sizes.simps(2))
lemma input-sizes-deep-model:
assumes length rs ≥ 2
shows input-sizes (deep-model-l rs) = replicate (2^(length rs - 2)) (last rs)
proof
have input-sizes (deep-model-l rs) = input-sizes (deep-model'-l (tl rs))
by (metis One-nat-def Suc-1 assms hd-Cons-tl deep-model.elims input-sizes.simps(2)
length-Cons
length-greater-0-conv lessI linorder-not-le list.size(3) not-numeral-le-zero nth-tl)
also have ... = replicate (2^(length rs - 2)) (last rs) using input-sizes-deep-model'
by (metis (no-types, lifting) One-nat-def Suc-1 Suc-eq-plus1 assms diff-diff-left
hd-Cons-tl
last-tl length-Cons length-tl linorder-not-le list.size(3) not-less-eq not-numeral-le-zero
numeral-le-one-iff semiring-norm(69))
finally show ?thesis by auto
qed

lemma evaluate-net-Conv-id:
assumes valid-net' m
and input-sizes m = map dim-vec input
and j < nr
shows evaluate-net (Conv (id-matrix nr (output-size' m)) m) input $ j = (if j < output-size' m then evaluate-net m input $ j else 0)
unfolding evaluate-net.simps output-size-correct[OF assms(1) assms(2)[symmetric]]
using mult-id-matrix[OF j<nr], of evaluate-net m input, unfolded dim-vec-of-list]
by metis

lemma tensors-from-net-Conv-id:
assumes valid-net' m
and i < nr
shows tensors-from-net (Conv (id-matrix nr (output-size' m)) m) $ i = (if i < output-size' m then tensors-from-net m $ i else tensor0 (input-sizes m))
(is ?a $ i = ?b)
proof (rule tensor-lookup-eqI)
have Tensor.dims (?a $ i) = input-sizes m by (metis assms(1) assms(2) dims-tensors-from-net
id-matrix-dim(1) id-matrix-dim(2) input-sizes.simps(2) output-size.simps(2)

\[\text{output-size-correct-tensors remove-weights} \text{simps(2)} \text{valid-net.intro(2)} \text{vec-setI)}\]

\textbf{moreover have} \(\text{Tensor.dims (\$b) = input-sizes m}\) \textbf{using} \(\text{dims-tensors-from-net output-size-correct-tensors\[OF assms(1)\]} \text{dims-tensor0 by (simp add: vec-setI)}\)

\textbf{ultimately show} \(\text{Tensor.dims (\$a \& i) = Tensor.dims (\$b)}\) \textbf{by auto}

\textbf{define} \(\text{Convm where} \text{Convm = Conv (id-matrix nr (output-size' m)) m}\)
\textbf{fix} is
\textbf{assume} is \(<\text{Tensor.dims (\$a\&i)}\)
\textbf{then have} is \(<\text{input-sizes m using} \text{Tensor.dims (\$a\&i) = input-sizes m}\) \textbf{by auto}
\textbf{have} \(\text{base-input' Conv m by (simp add: assms id-matrix-dim valid-net.intro(2)} \text{Convm-def)}\)
\textbf{have} \(\text{base-input m is = base-input Conv m by (simp add: Convm-def base-input-def)}\)
\textbf{have} \(i < \text{output-size' Conv unfolding Conv-def remove-weights}.\text{simps output-size.simps}\)
\textbf{id-matrix-dim using} \(\text{assms by metis}\)
\textbf{have} is \(<\text{input-sizes (Conv (id-matrix nr (output-size' m)) m)}\) \textbf{by (metis is \(<\text{input-sizes m} \text{ input-sizes}.\text{simps(2)}))}\)
\textbf{then have} \(f1: \text{lookup (tensors-from-net (Conv (id-matrix nr (output-size' m))) m)}\) \textbf{is = evaluate-net (Conv (id-matrix nr (output-size' m)) m)} \textbf{base-input (Conv (id-matrix nr (output-size' m)) m) is} \textbf{\$ i}\)
\textbf{using} \(\text{Convm-def (i < output-size' Conv) (valid-net Conv) lookup-tensors-from-net}\)
\textbf{by blast}\n\textbf{have} \(\text{lookup (tensor0 (input-sizes m)) is = (0::real)}\)
\textbf{by (meson is \(<\text{input-sizes m} \text{ lookup-tensor0})}\)
\textbf{then show} \(\text{Tensor.lookup (\$a \& i) is = Tensor.lookup \$b is}\)
\textbf{using} \(\text{Convm-def (base-input m is = base-input Conv m) (i < input-sizes m} \text{assms(1) assms(2)}\)
\textbf{base-input-length evaluate-net-Conv-id f1 lookup-tensors-from-net by auto}\n
\textbf{qed}\n
\textbf{lemma} \(\text{evaluate-net-Conve-copy-first:}\)
\textbf{assumes} valid-net' m\n\textbf{and} input-sizes m = map dim-vec input\n\textbf{and} \(\text{j < nr}\)
\textbf{and} output-size' m > 0\n\textbf{shows} evaluate-net (Conv (copy-first-matrix nr (output-size' m)) m) input \$ j = evaluate-net m input \$ 0\n\textbf{unfolding} \(\text{evaluate-net.simps output-size-correct\[OF assms(1) assms(2)|symmetric]}\)
\textbf{using} \(\text{mult-copy-first-matrix\[OF \text{'j < nr'}}, \text{ of evaluate-net m input, unfolded dim-vec-of-list]}\)
\textbf{assms(3) copy-first-matrix-dim(1) by (metis output-size' m = dim-vec (evaluate-net m input)) assms(4))}\n
\textbf{lemma} \(\text{tensors-from-net-Conve-copy-first:}\)
\textbf{assumes} valid-net' m\n\textbf{and} \(\text{i < nr}\)
\textbf{and} output-size' m > 0\n\textbf{shows} tensors-from-net (Conv (copy-first-matrix nr (output-size' m)) m) \$ i = tensors-from-net m \$ 0

66
\begin{verbatim}
(is \( ?a \$ i = ?b \))

proof (rule tensor-lookup-eqI)
  have \( \text{Tensor.dims} \ (?a\$i) = \text{input-sizes} \ m \)
    by (metis assms(1) assms(2) copy-first-matrix-dim(1) copy-first-matrix-dim(2)
      dims-tensors-from-net
        input-sizes.simps(2) output-size.simps(2) output-size-correct-tensors remove-weights.simps(2)
        valid-net.intros(2) vec-setI)
  moreover have \( \text{Tensor.dims} \ (?b) = \text{input-sizes} \ m \) using dims-tensors-from-net
    output-size-correct-tensors[OF assms(1)] using assms(3) by (simp add: vec-setI)
  ultimately show \( \text{Tensor.dims} \ (?a\$i) = \text{Tensor.dims} \ (?b) \) by auto

define \text{Conv}m where \text{Conv}m = \text{Conv} \ (\text{copy-first-matrix} \ nr \ (\text{output-size'} \ m)) \ m
fix is
assume is \( < \) \( \text{Tensor.dims} \ (?a\$i) \)
then have is \( < \) \( \text{input-sizes} \ m \) using \( \langle \text{Tensor.dims} \ (?a\$i) = \text{input-sizes} \ m \rangle \) by auto
have valid-net' \text{Conv}m by (simp add: assms copy-first-matrix-dim valid-net.intros(2)
  Convm-def)
have base-input \( m \) is \( = \) base-input \text{Conv}m is by (simp add: Convm-def base-input-def)
have \( i < \) output-size' \text{Conv}m unfolding Convm remove-weights.simps output-size.simps
  copy-first-matrix-dim using assms by metis
  show \text{Tensor.lookup} \ (\?a \$ i) \ is = \text{Tensor.lookup} \ ?b \ is
    by (metis Convm-def base-input m is = base-input Convm is \langle i < output-size' \text{Conv}m \rangle)
    \langle is \( < \) \text{input-sizes} \ m \rangle \ valid-net' \text{Conv}m \ assms(1) assms(2) assms(3) base-input-length
    evaluate-net-Conv-copy-first input-sizes.simps(2) lookup-tensors-from-net)
qed

lemma evaluate-net-Conv-all1:
assumes valid-net' \( m \)
and input-sizes \( m = \) map dim-vec input
and \( i < nr \)
shows evaluate-net \ (\text{Conv} \ (all1-matrix \ nr \ (\text{output-size'} \ m)) \ m) \ input \$ i
  = Groups-List.sum-list \ (list-of-vec \ (evaluate-net \ m \ input))
unfolding evaluate-net.simps output-size-correct[OF assms(1) assms(2)] symmetric]
using mult-all1-matrix[OF \langle i < nr \rangle, of evaluate-net \ m \ input, unfolded dim-vec-of-list]
  assms(3) all1-matrix-dim(1) by metis

lemma tensors-from-net-Conv-all1:
assumes valid-net' \( m \)
and \( i < nr \)
shows tensors-from-net \ (\text{Conv} \ (all1-matrix \ nr \ (\text{output-size'} \ m)) \ m) \$ i
  = listsum \ (input-sizes \ m) \ (list-of-vec \ (tensors-from-net \ m))
  \langle is \ ?a \$ i = \?b \rangle
proof (rule tensor-lookup-eqI)
  have \( i < \) dim-vec \ ?a \ by (metis assms all1-matrix-dim output-size.simps(2)
    output-size-correct-tensors remove-weights.simps(2) valid-net.intros(2))
  then show \( \text{Tensor.dims} \ (?a \$ i) = \text{Tensor.dims} \ (?b) \)
end
\end{verbatim}
using dims-tensors-from-net input-sizes.simps(2) listsum-dims
by (metis index-vec-of-list in-set-conv-nth length-list-of-vec vec-list vec-setI)

define Convm where Convm = Conv (all1-matrix nr (output-size' m)) m
fix is assume is < Tensor.dims (?a $ i)
then have is < input-sizes m
using (i < dim-vec ?a) dims-tensors-from-net input-sizes.simps(2) by (metis vec-setI)
then have is < input-sizes Convm by (simp add: Convm-def)
have valid-net' Convm by (simp add: Convm-def assms all1-matrix-dim valid-net.intros(2))
have i< output-size' Convm using Convm-def (i < dim-vec ?a) (valid-net' Convm)

have valid-net' Convm by (simp add: Convm-def assms all1-matrix-dim valid-net.intros(2))

have output-size-correct-tensors by presburger
have base-input Convm is = base-input m is unfolding base-input-def Convm-def input-sizes.simps by metis
have Tensor.lookup (?a $ i) is = evaluate-net Convm (base-input Convm is) $ i
using lookup-tensors-from-net[OF valid-net' Convm] (i < input-sizes Convm)
by (metis Convm-def)
also have ... = monoid-add-class.sum-list (list-of-vec (evaluate-net m (base-input Convm is)))
using evaluate-net-Conv-all1 Convm-def (i < input-sizes Convm) assms base-input-length
(i < nr)
by simp
also have ... = monoid-add-class.sum-list (list-of-vec (map-vec (λA. lookup A is)(tensors-from-net m)))
by force
also have ... = monoid-add-class.sum-list (map (λA. lookup A is) (list-of-vec (tensors-from-net m)))
using eq-vect[of-vec-of-list (list-of-vec (map-vec (λA. lookup A is) (tensors-from-net m))))
vec-of-list (map (λA. lookup A is) (list-of-vec (tensors-from-net m)))) vec-of-list (map (λA. lookup A is) (list-of-vec (tensors-from-net m)))) vec-of-list (map (λA. lookup A is) (list-of-vec (tensors-from-net m))))
by (metis (no-types, lifting))
also have ... = Tensor.lookup ?b is using dims-tensors-from-net set-list-of-vec
using lookup-listsum[OF (i < input-sizes m), of list-of-vec (tensors-from-net m)]
by metis
finally show Tensor.lookup (?a $ i) is = Tensor.lookup ?b is by blast
qed

fun witness and witness' where
witness' Y r = Input Y |
witness' Y (r # rs) = Pool (witness Y r rs) (witness Y r rs) |
witness Y r rs = Conv ((if length rs = 0 then id-matrix else (if length rs = 1 then all1-matrix else copy-first-matrix)) Y r) (witness' Y rs)

abbreviation witness-l rs == witness (rs!0) (rs!1) (tl (tl rs))
abbreviation witness'-l rs == witness' (rs!0) (tl rs)

lemma witness-is-deep-model: remove-weights (witness Y r rs) = deep-model Y r rs
proof (induction rs arbitrary; Y r)
  case Nil then show ?case unfolding witness.simps witness'.simps deep-model.simps deep-model'.simps
    by (simp add: id-matrix-dim)
next
  case (Cons r' rs Y r)
    have dim-row (((if length (r' # rs) = 0 then id-matrix else (if length (r' # rs) = 1 then all1-matrix else copy-first-matrix)) Y r) = Y
        dim-col (((if length (r' # rs) = 0 then id-matrix else (if length (r' # rs) = 1 then all1-matrix else copy-first-matrix)) Y r) = r
      by (simp-all add: all1-matrix-dim copy-first-matrix-dim)
    then show ?case unfolding witness.simps unfolding witness'.simps unfolding remove-weights.simps
      using Cons by simp
  qed

lemma witness'-is-deep-model: remove-weights (witness' Y rs) = deep-model' Y rs
proof (induction rs arbitrary; Y)
  case Nil then show ?case unfolding witness.simps witness'.simps deep-model.simps deep-model'.simps
    by (simp add: id-matrix-dim)
next
  case (Cons r rs Y)
    have dim-row (((if length rs = 0 then id-matrix else (if length rs = 1 then all1-matrix else copy-first-matrix)) Y r) = Y
        dim-col (((if length rs = 0 then id-matrix else (if length rs = 1 then all1-matrix else copy-first-matrix)) Y r) = r
      by (simp-all add: all1-matrix-dim copy-first-matrix-dim id-matrix-dim)
    then show ?case unfolding witness'.simps unfolding witness.simps unfolding remove-weights.simps
      using Cons by simp
  qed

lemma witness-valid: valid-net' (witness Y r rs)
  using valid-deep-model witness-is-deep-model by auto

lemma witness'-valid: valid-net' (witness' Y rs)
  using valid-deep-model' witness'-is-deep-model by auto
lemma \textit{shared-weight-net-witness}: \textit{shared-weight-net} (witness \(Y\) \(r\) \(rs\))

proof (induction \(rs\) arbitrary; \(Y\) \(r\))

\begin{itemize}
  \item case \texttt{Nil}
    \begin{itemize}
      \item then show \texttt{?case unfolding witness.simps witness’.simps by (simp add: shared-weight-net-Conv shared-weight-net-Input)}
    \end{itemize}

  \item case \(\texttt{(Cons a rs)}\)
    \begin{itemize}
      \item then show \texttt{?case unfolding witness.simps witness’.simps}
        \begin{itemize}
          \item by (simp add: shared-weight-net-Conv shared-weight-net-Input shared-weight-net-Pool)
        \end{itemize}
    \end{itemize}
\end{itemize}

qed

lemma \textit{witness-l0’}: \(\text{witness’} Y [M] =\)
\(\text{(Pool)}\)
\(\text{(Conv (id-matrix Y M) (Input M))}\)
\(\text{(Conv (id-matrix Y M) (Input M))}\)

\textit{unfolding witness’.simps witness.simps by simp}

lemma \textit{witness-l1}: \(\text{witness Y r0 [M]} =\)
\(\text{Conv (all1-matrix Y r0) (witness’ r0 [M])}\)

\textit{unfolding witness’.simps by simp}

lemma \textit{tensors-ht-l0}:
assumes \(j < r0\)
shows \(\text{tensors-from-net (Conv (id-matrix r0 M) (Input M))} \emptyset j = (\text{if } j < M \text{ then unit-vec M } j \text{ else } \text{tensor0 [M]})\)
by (metis assms input-sizes.simps(1) output-size.simps(1) remove-weights.simps(1)
tensors-from-net.simps(1)
tensors-from-net-Conv-id valid-net.intros(1) index-vec)

lemma \textit{tensor-prod-unit-vec}:
unit-vec M \(j \otimes \) unit-vec M \(j = \text{tensor-from-lookup [M,M]} \lambda is. \text{if } is = [j,j] \text{ then } 1 \text{ else } 0\) (is \(\text{?A=?B}\))

proof (rule tensor-lookup-eqI)
show \(\text{Tensor.dims ?A = Tensor.dims ?B}\)
by (metis append-Cons self-append-conv2 dims-unit-vec dims-tensor-prod dims-tensor-from-lookup)
fix \(\text{is = valid:is \text{< } Tensor.dims (unit-vec M } j \otimes \text{ unit-vec M } j\)\)
then have \(\text{is = [M,M] by (metis append-Cons self-append-conv2 dims-unit-vec dims-tensor-prod)}\)
then obtain \(i1 \ i2\) where \(\text{is-split: is = [i1, i2]} i1 < M i2 < M\) using list.different(1)
by blast
then have \([i1] < \text{Tensor.dims (unit-vec M } j\) \([i2] < \text{Tensor.dims (unit-vec M } j\)
by (simp-all add: valid-index.Cons valid-index.Nil dims-unit-vec)
have \(\text{is = [i1] @ [i2]}\) by (simp add: is-split(1))
show \(\text{Tensor.lookup ?A is = Tensor.lookup ?B is}\)
unfolding \(\langle is = [i1] @ [i2]\rangle\)
lookup-tensor-prod[OF \(\langle [i1] < \text{Tensor.dims (unit-vec M } j\rangle \langle [i2] < \text{Tensor.dims (unit-vec M } j\rangle\)]\)
lookup-tensor-from-lookup[OF \(\langle is < [M, M]\rangle, \text{unfolded } \langle is = [i1] @ [i2]\rangle\)]
lemma tensors-HT-l0':
assumes j < r
shows \(\text{Tensors} \cdot \text{lookup} \ (\text{Tensors} \cdot \text{from} \cdot \text{net} \ (\text{witness} ' r0 [M]) \ [j]) \ = \ (\text{if } j < M \text{ then } \text{unit-vec } M \ j \ \otimes \ \text{unit-vec } M \ j \ \text{else } \text{tensor0 } [M,M]) \) (is = ?b)
proof
  have valid-net' (Conv (id-matrix r0 M) (Input M))
    by (metis convnet.inject(3) list.discl witness'.elims witness-l0' witness-valid)
  have j-le: j < dim-vec (Tensors-from-net (Conv (id-matrix r0 M) (Input M)))
    using output-size-correct-tensors[OF valid-net' (Conv (id-matrix r0 M) (Input M))],
    unfolded remove-weights.simps output-size.simps id-matrix-dim]
    assms by simp
  show ?thesis unfolding tensors-HT-l0'[OF assms]
  by auto
qed

lemma lookup-tensors-HT-l0':
assumes j < r0
and is \(\prec [M,M]\)
shows \((\text{Tensor} \cdot \text{lookup} \ (\text{Tensors} \cdot \text{from} \cdot \text{net} \ (\text{witness} ' r0 [M]) \ [j]) \) is = (if is = [j,j] then 1 else 0)
proof (cases j < M)
  assume j < M
  show ?thesis unfolding tensors-HT-l0'[OF assms(1)] tensor-prod-unit-vec
    apply (cases is = [j,j]) using j < M assms(2)
    by (simp-all add:lookup-tensor-from-lookup)
next
  assume \(\neg j < M\)
  then have is \(\neq [j,j]\) using assms(2) using list.distinct(1) nths-Cons-0 valid-index.simps
  by blast
  show ?thesis unfolding tensors-HT-l0'[OF assms(1)] tensor-prod-unit-vec
    using \(\neg j < M\), by (simp add:lookup-tensor0[OF assms(2)] is \(\neq [j,j]\))
qed

lemma lookup-tensors-HT-l1:
assumes j < r1
and is \(\prec [M,M]\)
shows Tensor.lookup (Tensors-from-net (witness r1 r0 [M]) \ [j]) is = (if is!0 = is!1 \ \land \ is!0 < r0 then 1 else 0)
proof
  have witness-l0'-valid: valid-net' (witness' r0 [M]) unfolding witness-l0'
    by (simp add: id-matrix-dim valid-net.intros)
  have input-sizes (witness' r0 [M]) = [M,M] unfolding witness-l0' by simp
qed
have output-size' (witness' r0 [M]) = r0 unfolding witness-l0' using witness-l0'-valid
   by (simp add: id-matrix-dim)
have dim-vec (tensors-from-net (witness' r0 [M])) = r0
  using output-size' (witness' r0 [M]) = r0 witness-l0'-valid output-size-correct-tensors
by fastforce
have all0-but1:∀ i. i≠0 → i<r0 → Tensor.lookup (tensors-from-net (witness' r0 [M])) $ i = 0
  using lookup-tensors-ht-l0' (is < [M, M]) by auto

have tensors-from-net (witness r1 r0 [M]) $ j =
  Tensor-Plus.listsum [M,M] (list-of-vec (tensors-from-net (witness' r0 [M])))
unfolding witness-l1 using tensors-from-net-Conv-allI[OF witness-l0'-valid
assms(1)]
  witness-l0' (output-size' (witness' r0 [M]) = r0) by simp
then have Tensor.lookup (tensors-from-net (witness r1 r0 [M]) $ j) is
  = monoid-add-class.sum-list (map (λA. Tensor.lookup A is) (list-of-vec (tensors-from-net
  (witness' r0 [M]))))
  using lookup-listsum[OF is < [M, M];] input-sizes (witness' r0 [M]) = [M, M]
  using dim-tensors-from-net by (metis set-list-of-vec)
also have ... = monoid-add-class.sum-list (map (λi. lookup (tensors-from-net
  (witness' r0 [M]) $ i) is) [0..<r0])
  using map-map (λi. Tensor.lookup A is) (λi. (tensors-from-net (witness' r0
  [M]) $ i) [0..<r0])
  using list-of-vec-map (dim-vec (tensors-from-net (witness' r0 [M])) = r0) by
  (metis (mono-tags, lifting) comp-apply map-eq-conv)
also have ... = (∑ i<r0. Tensor.lookup ((tensors-from-net (witness' r0 [M]) $ i) is)
  using sum-set-apt-conv-sum-list-atLeast0LessThan by (metis atLeast-apt)
also have ... = (if is0 = is1 ∧ is0<r0 then 1 else 0)
proof (cases is0<r0)
  case True
    have finite {0..<r0} by auto
    have is0 ∈ {0..<r0} using True by auto
  case False
    have (∑ i<r0. Tensor.lookup ((tensors-from-net (witness' r0 [M]) $ i) is)
      = Tensor.lookup (tensors-from-net (witness' r0 [M]) $ (is0)) is
  using dim-vec (tensors-from-net (witness' r0 [M])) = r0
  using sum.remove[OF finite {0..<r0}; is0 ∈ {0..<r0}],
    of λi. (Tensor.lookup (tensors-from-net (witness' r0 [M])$i) is)
  using all0-but1 atLeast0LessThan by force
then show ?thesis using lookup-tensors-ht-l0' (is ! 0 < r0) (is < [M, M]) by
  fastforce
next
  case False
  then show ?thesis using all0-but1 atLeast0LessThan sum.neutral by force
qed
finally show \( \textit{thesis} \) by auto

qed

lemma length-output-deep-model:
assumes remove-weights \( m = \text{deep-model-l \( rs \)} \)
shows \( \text{dim-vec (tensors-from-net \( m \)) = \( rs \rightarrow 0 \)} \)
using output-size-correct-tensors valid-deep-model
deeper-model.elims output-size.simps(2) by (metis assms)

lemma length-output-deep-model':
assumes remove-weights \( m = \text{deep-model'\( l \) \( rs \)} \)
shows \( \text{dim-vec (tensors-from-net \( m \)) = \( rs \rightarrow 0 \)} \)
using output-size-correct-tensors valid-deep-model'
deeper-model'.elims output-size.simps by (metis assms)

lemma length-output-witness:
\( \text{dim-vec (tensors-from-net (witness-l \( rs \))) = \( rs \rightarrow 0 \)} \)
using length-output-deep-model witness-is-deep-model by blast

lemma length-output-witness':
\( \text{dim-vec (tensors-from-net (witness'\( l \) \( rs \))) = \( rs \rightarrow 0 \)} \)
using length-output-deep-model' witness'-is-deep-model by blast

lemma dims-output-deep-model:
assumes length \( rs \geq 2 \)
and \( \forall r. \ r \in \text{set \( rs \) \( \implies r > 0 \)} \)
and \( j < rs!0 \)
and remove-weights \( m = \text{deep-model-l \( rs \)} \)
shows \( \text{Tensor.dims (tensors-from-net \( m \$ j \) = replicate (} \( 2^{(\text{length \( rs \) - 2)}) \) \( (last \( rs \)) \)
using dims-tensors-from-net input-sizes-deep-model[of assms(1)] output-size-correct-tensors valid-deep-model
assms(3) assms(4) input-sizes-remove-weights length-output-witness witness-is-deep-model
by (metis vec-setI)

lemma dims-output-witness:
assumes length \( rs \geq 2 \)
and \( \forall r. \ r \in \text{set \( rs \) \( \implies r > 0 \)} \)
and \( j < rs!0 \)
shows \( \text{Tensor.dims (tensors-from-net (witness-l \( rs \) \$ j) = replicate (} \( 2^{(\text{length \( rs \) - 2)}) \) \( (last \( rs \)) \)
using dims-output-deep-model witness-is-deep-model assms by blast

lemma dims-output-deep-model':
assumes length \( rs \geq 1 \)
and \( \forall r. \ r \in \text{set \( rs \) \( \implies r > 0 \)} \)
and \( j < rs!0 \)
and remove-weights \( m = \text{deep-model'\( l \) \( rs \)} \)
shows \( \text{Tensor.dims (tensors-from-net \( m \$ j \) = replicate (} \( 2^{(\text{length \( rs \) - 1)}) \) \( (last \( rs \)) \)

73
rs)

proof –
  have dim-vec (tensors-from-net m) > j
    using length-output-deep-model' \(\langle remove-weights m = deep-model'-l rs, j < rs!0, rs!0 \rangle\) by auto
  then have Tensor.dims (tensors-from-net m \$ j) = input-sizes m
    using dims-tensors-from-net[of - m] output-size-correct-tensors
    vec-setI by metis
  then show \(?thesis\)
    using assms \(1\) input-sizes-deep-model'
    input-sizes-remove-weights[of m, unfolded \(\langle remove-weights m = deep-model'-l rs, rs!0, rs!0 \rangle\)] by auto
qed

lemma dims-output-witness':
  assumes \(length rs \geq 1\)
  and \(\mathop{\bigwedge} r. r \in \text{set rs} \Rightarrow r > 0\)
  and \(j < rs!0\)
  shows Tensor.dims (tensors-from-net (witness'-l rs) \$ j) = replicate \(2^{\mathop{\bigwedge}(length rs - 1)} (last rs)\)
  using dims-output-deep-model' assms witness'-is-deep-model by blast

abbreviation ten2mat == matricize \(\{n. \text{even } n\}\)
abbreviation mat2ten == dematricize \(\{n. \text{even } n\}\)

locale deep-model-correct-params =
  fixes shared-weights::bool
  fixes rs::nat list
  assumes deep:length rs \geq 3
  and no-zeros:\(\mathop{\bigwedge} r. r \in \text{set rs} \Rightarrow 0 < r\)
begin

definition \(r = \text{min} (last rs) (last (butlast rs))\)
definition \(N\text{-half} = 2^{\mathop{\bigwedge}(length rs - 3)}\)
definition weight-space-dim = count-weights shared-weights (deep-model-l rs)
end

locale deep-model-correct-params-y = deep-model-correct-params +
  fixes y::nat
  assumes y-valid:y < rs!0
begin

definition \(A :: (nat \Rightarrow real) \Rightarrow real \text{ tensor}\)
  where \(A ws = \text{tensors-from-net (insert-weights shared-weights (deep-model-l rs) ws)} \$ y\)
definition \(A' :: (nat \Rightarrow real) \Rightarrow real \text{ mat}\)
  where \(A' ws = \text{ten2mat} (A ws)\)
lemma dims-tensor-deep-model:
assumes remove-weights m = deep-model-l rs
shows dims (tensors-from-net m $ y) = replicate (2 * N-half) (last rs)
proof
  have dims (tensors-from-net m $ y) = replicate (2 ^ (length rs - 2)) (last rs)
    using dims-output-deep-model[OF - no-zeros y-valid assms] using less-imp-le-nat
    Suc-le-lessD deep numeral-3-eq-3
      by auto
  then show thesis using N-half-def by (metis One-nat-def Suc-1 Suc-eq-plus1
    Suc-le-lessD deep diff-diff-left less-numeral-extra(3) numeral-3-eq-3 power-eq-if zero-less-diff)
qed

lemma order-tensor-deep-model:
assumes remove-weights m = deep-model-l rs
shows order (tensors-from-net m $ y) = 2 * N-half
  using dims-tensor-deep-model by (simp add: assms)

lemma dims-A:
shows Tensor.dims (A ws) = replicate (2 * N-half) (last rs)
  unfolding A-def
  using dims-tensor-deep-model remove-insert-weights by blast

lemma order-A:
shows order (A ws) = 2 * N-half using dims-A length-replicate by auto

lemma dims-A'?
shows dim-row (A' ws) = prod-list (nths (Tensor.dims (A ws)) (n. even n))
and dim-col (A' ws) = prod-list (nths (Tensor.dims (A ws)) (n. odd n))
  unfolding A'-def matricize-def by (simp-all add: A-def Collect-neg-eq)

lemma dims-A'-pow:
shows dim-row (A' ws) = (last rs) ^ N-half dim-col (A' ws) = (last rs) ^ N-half
  unfolding dims-A' dims-A nths-replicate set-le-in card-even card-odd prod-list-replicate
    by simp-all

definition Aw = tensors-from-net (witness-l rs) $ y

definition Aw' = ten2mat Aw

definition witness-weights = extract-weights shared-weights (witness-l rs)

lemma witness-weights:witness-l rs = insert-weights shared-weights (deep-model-l rs) witness-weights
  by (metis full-types insert-extract-weights-cong-shared insert-extract-weights-cong-unshared
    shared-weight-net-witness witness-is-deep-model witness-weights-def)
lemma Aw-def': $A w = A$ witness-weights unfolding $A w$ def A-def using witness-weights by auto

lemma Aw'-def': $A w' = A'$ witness-weights unfolding $A w$' def A'-def Aw-def' by auto

lemma dims-Aw: Tensor.dims Aw = replicate $(2 * \text{N-half})$ (last rs) unfolding Aw-def' using dims-A by auto

lemma order-Aw; order $A w = 2 * \text{N-half}$ unfolding Aw-def' using order-A by auto

lemma dims-Aw':
\begin{align*}
\text{dim-row } A w' & = \text{prod-list} (\text{nths} (\text{Tensor.dims } A w) \{ n. \text{ even } n \}) \\
\text{dim-col } A w' & = \text{prod-list} (\text{nths} (\text{Tensor.dims } A w) \{ n. \text{ odd } n \})
\end{align*}

unfolding $A w$'-def' $A w$'-def using dims-A' by auto

lemma dims-Aw'-pow: dim-row $A w' = (\text{last } rs) \wedge N$-half dim-col $A w' = (\text{last } rs) \wedge N$-half unfolding $A w$'-def' $A w$'-def using dims-A'-pow by auto

lemma witness-tensor:
assumes is $\subset Tensor$.
dims Aw
shows $\text{Tensor.lookup } A w$ is
\begin{align*}
& = (\text{if } \text{nths is } \{ n. \text{ even } n \} = \text{nths is } \{ n. \text{ odd } n \} \land (\forall \, i \in \text{set is. } i < \text{last (butlast } rs)) \text{ then } 1 \text{ else } 0) \\
\text{using assms deep no-zeros y-valid unfolding } A w\text{-def proof (induction butlast (butlast (butlast } rs)) arbitrary: } rs \text{ is } y
\end{align*}

case Nil
have length $rs = 3$
by (rule antisym, metis Nil.hyps One-nat-def Suc-1 Suc-eq-plus1 add-2-eq-Suc diff-diff-left
  length-butlast less-numeral-extra(3) list.size(3) not-le numeral-3-eq-3 zero-less-diff, metis $3 \leq \text{length } rs$)
then have $rs = [rs!0, \text{rs}!1, \text{rs}!2]$ by (metis (no-types, lifting) Cons-nth-drop-Suc One-nat-def Suc-eq-plus1
append-nil id-take-nth-drop length-0-conv length-tl lessI list.sel(3) list.size(4)
not-le numeral-3-eq-3
numeral-le-one-iff one-add-one semiring-norm(70) take-0 zero-less-Suc)

have input-sizes (witness-l [rs ! 0, rs ! 1, rs ! 2]) = [rs!2, rs!2]
using witness.simps witness'..simps input-sizes.simps by auto
then have Tensor.dims (tensors-from-net (witness-l rs) $y) = [rs!2, rs!2]
using dims-tensors-from-net[of tensors-from-net (witness-l rs) $y$ y witness-l rs]
  Nil.prems(4) length-output-witness $\langle rs = [rs ! 0, rs ! 1, rs ! 2], vec-set I \rangle$ by metis
then have is $\subset [rs!2, rs!2]$ using Nil.prems by metis
then have Tensor.lookup (((tensors-from-net (witness-l rs))$y)$y) is
\begin{align*}
& = (\text{if } is ! 0 = \text{is ! 1} \land \text{is ! 0 } < \text{rs ! 1 then } 1 \text{ else } 0)
\end{align*}

76
using Nil.prems(4) :rs = [rs ! 0, rs ! 1, rs ! 2], by (metis list.sel(3) lookup-tensors-ht-II)

have is ! 0 = is ! 1 ∧ is ! 0 < rs ! 1

⇒ nths is \{n. even n\} = nths is \{n. odd n\} ∧ (∀ i∈ set is. i < last (butlast rs))

proof

have length is = 2 by (metis One-nat-def Suc-eq-plus1 \is < [rs ! 2, rs ! 2]\)

list.size(3) list.size(4) numeral-2-eq-2 valid-index-length

have nths is \{n. even n\} = [is!0]

apply (rule nths-only-one)

using subset-antisym less-2-cases \length is = 2\, by fastforce

have nths is \{n. odd n\} = [is!1]

apply (rule nths-only-one)

using subset-antisym less-2-cases \length is = 2\, by fastforce

have last (butlast rs) = rs!1 by (metis One-nat-def Suc-eq-plus1 \rs = [rs ! 0, rs ! 1, rs ! 2]\)

append-butlast-last-id last-conv-nth length-butlast length-tl lessI list.sel(3)

list.simps(3)

list.size(3) list.size(4) nat.simps(3) nth-append

show ?thesis unfolding \last (butlast rs) = rs!1\)

apply (rule iffI; rule conil)

apply (simp add: nths is (Collect even) = [is ! 0]; nths is \{n. odd n\} = [is ! 1];)

apply (metis \length is = 2\, One-nat-def in-set-conv-nth less-2-cases)

apply (simp add: nths is (Collect even) = [is ! 0]; nths is \{n. odd n\} = [is ! 1];)

apply (simp add: \length is = 2\)

done

qed

then show \case unfolding \Tensor.lookup (tensors-from-net (witness-l rs) $ y) \is = (if is ! 0 = is ! 1 ∧ is ! 0 < rs ! 1 then 1 else 0)\)

using witness-is-deep-model witness-valid \rs = [rs ! 0, rs ! 1, rs ! 2]\, by auto

next

case (Cons r rs' rs is j)

We prove the Induction Hypothesis for "tl rs" and j=0:

have rs = r # tl rs by (metis Cons.hyps(2) append-butlast-last-id butlast.simps(1)
hd-append2 list.collapse list.discI list.sel(1))

have 1:rs' = butlast (butlast (tl rs)) by (metis Cons.hyps(2) butlast-tl
list.sel(3))

have 2:3 ≤ \length (tl rs)\ by (metis (no-types, lifting) Cons.hyps(2) Cons.prems(2)
Nilpick.size-list-simp(2) One-nat-def Suc-eq-plus1 \rs = r # tl rs\ \rs' = butlast
(butlast (tl rs)))

diff-diff-left diff-self-eq-0 gr0-cone-Suc le-Suc-eq-length-butlast length-tl less-numeral-extra(3)
list.simps(3) numeral-3-eq-3)

have 3:\∀ r. r ∈ set (tl rs) ⇒ 0 < r by (metis Cons.prems(3) list.sel(2)
list.set.sel(2))

have 4:0 < (tl rs) ! 0 using 2 3 by auto

have IH: \ exists'. is' < \Tensor.dims (tensors-from-net (witness-l (tl rs)) $ 0)\)

⇒ Tensor.lookup (tensors-from-net (witness-l (tl rs)) $ 0) is' =
(if nths is′ (Collect even) = nths is′ {n. odd n} ∧ (∀ i ∈ set is′. i < last (butlast (tl rs))) then 1 else 0)

using 1 2 3 4 Cons.hyps(1) by blast

The list "is" can be split in two parts:

have is < replicate (2^\((length rs − 2)\)) (last rs)
using Cons.prems(3) dims-output-witness 2 by (metis (no-types, lifting) Cons.prems(1)) Cons.prems(3)
Cons.prems(4) Nitpick.size-list-simp(2) One-nat-def diff-diff-left diff-is-0-eq length-tl
nat-le-linear not-numeral-le-zero numeral-le-one-iff one-add-one semiring-norm(70))
then have is < replicate (2^\((length (tl rs) − 2)\)) (last rs) @ replicate (2^\((length (tl rs) − 2)\)) (last rs)
using Cons.prems dims-output-witness by (metis 2 Nitpick.size-list-simp(2) One-nat-def
diff-diff-left length-tl mult-2 not-numeral-le-zero numeral-le-one-iff one-add-one semiring-norm(70))
then obtain is1 is2 where is = is1 @ is2 and
   is1-replicate: is1 < replicate (2^\((length (tl rs) − 2)\)) (last rs) and
   is2-replicate: is2 < replicate (2^\((length (tl rs) − 2)\)) (last rs) by (metis valid-index-split)
then have
   is1-valid: is1 < Tensor.dims (tensors-from-net (witness-l (tl rs)) $ 0) (is ?is1)
   and
   is2-valid: is2 < Tensor.dims (tensors-from-net (witness-l (tl rs)) $ 0) (is ?is2)
   proof –
   have last (tl rs) = last rs by (metis 2 \(rs = r \# tl rs\) last-ConsR list.size(3)
not-numeral-le-zero)
   then show ?is1 ?is2 using dims-output-witness[of tl rs]
   using dims-output-witness[of tl rs] 2 3 is1-replicate is2-replicate \(\langle last \ (tl rs) = last rs\ \rangle\) by auto
qed

A shorthand for the condition to find a "1" in the tensor:

let ?cond = λis rs. nths is {n. even n} = nths is {n. odd n} ∧ (∀ i ∈ set is. i < last (butlast rs))

We can use the IH on our newly created is1 and is2:

have IH-is12:
   Tensor.lookup (tensors-from-net (witness-l (tl rs)) $ 0) is1 =
   (if (?cond is1 (tl rs)) then 1 else 0)
   Tensor.lookup (tensors-from-net (witness-l (tl rs)) $ 0) is2 =
   (if (?cond is2 (tl rs)) then 1 else 0)
   using IH is1-valid is2-valid by fast+

In the induction step we have to add two layers: first the Pool layer, then the Conv layer.

The Pool layer connects the two subtrees. Therefore the two conditions on is1 and is2 become one, and we have to prove that they are equivalent:
have \( \exists \text{cond is1} (tl \, rs) \wedge \exists \text{cond is2} (tl \, rs) \iff \text{cond is rs} \)

proof
- have length is1 = 2 ^ (length (tl \, rs) - 2)
  length is2 = 2 ^ (length (tl \, rs) - 2)
  using is1-replicate is2-replicate by (simp-all add: valid-index-length)
then have even (length is1) even (length is2)
  by (metis Cons.hyps(2) One-nat-def add-gr-0 diff-diff-left even-numeral even-power
length-butlast length-tl list.size(\{ \} one-add-one zero-less-Suc)+
then have \{ j, j + length is1 \in \{ n. even n \}\} = \{ n. even n \}
  \{ j, j + length is1 \in \{ n. odd n \}\} = \{ n. odd n \} by simp-all
have length (nths is2 (Collect even)) = length (nths is2 (Collect odd))
  using length-nths-even \langle even (length is2) \rangle by blast
have cond1-iff: \langle nths is1 (Collect even) = nths is1 \{ n. odd n \} \wedge nths is2 (Collect even) = nths is2 \{ n. odd n \} \rangle
  = \langle nths is (Collect even) = nths is \{ n. odd n \} \rangle
  unfolding \langle is = is1 \otimes is2 \rangle nths-append
  \langle \{ j, j + length is1 \in \{ n. odd n \}\} = \{ n. odd n \} \rangle \langle \{ j, j + length is1 \in \{ n. even n \}\} = \{ n. even n \}\rangle
  by (simp add: \langle length (nths is2 (Collect even)) = length (nths is2 (Collect odd)) \rangle)
  have last (butlast (tl \, rs)) = last (butlast rs) using Nitpick.size-list-simp(2)
  \langle even (length is1) \rangle
  \langle length is1 = 2 ^ (length (tl \, rs) - 2) \rangle butlast-tl last-tl length-butlast length-tl
  not-less-eq zero-less-diff
  by (metis (full-types) Cons.hyps(2) length-Cons less-nat-zero-code)
  have cond2-iff: \langle \forall i \in \text{set is1}. i < \text{last (butlast (tl \, rs))} \rangle \wedge \langle \forall i \in \text{set is2}. i < \text{last (butlast (tl \, rs))} \rangle
    \longleftrightarrow \langle \forall i \in \text{set is}. i < \text{last (butlast rs)} \rangle
  unfolding \langle \text{last (butlast (tl \, rs))} = \text{last (butlast rs)} \rangle \langle \text{is} = \text{is1} \otimes \text{is2} \rangle \text{set-append}
  by blast
then show \?thesis using cond1-iff cond2-iff by blast
qed

Now we can make the Pool layer step:

have lookup-witness': Tensor.lookup ((tensors-from-net (witness' (rs ! 1) (tl (tl \, rs) ))) $ 0) is =
  (if \?cond is rs then 1 else 0)
proof
- have lookup-prod: Tensor.lookup ((tensors-from-net (witness-l (tl \, rs)) $ 0) \otimes
  (tensors-from-net (witness-l (tl \, rs))) $ 0) is =
  (if \?cond is rs then 1 else 0)
  using \?cond is1 (tl \, rs) \wedge \?cond is2 (tl \, rs) \iff \?cond is rs
  unfolding \langle is = is1 \otimes is2 \rangle lookup-tensor-prod[OF is1-valid is2-valid] IH-is12
  by auto
have witness-l-tl: witness-l (tl \, rs) = witness (rs ! 1) (rs ! 2) (tl (tl \, rs))
  by (metis One-nat-def Suc-1 \langle rs = r \# tl \, rs \rangle nths-Cons-Suc)
have tl-tl:tl (tl (tl \, rs)) = ((rs ! 2) \# tl (tl \, rs))
proof
- have length (tl (tl \, rs)) \neq 0

79
by (metis One-nat-def Suc-eq-plus1 diff-diff-left diff-is-0-eq length-tl not-less-eq-eq
Cons.prems(2) numeral-3-eq-3)
then have tl (tl rs) ≠ []
  by fastforce
then show "thesis"
  by (metis list.exhaust-sel nth-Cons-0 nth-Cons-Suc numeral-2-eq-2 tl-Nil)
qed
have length-gt0:dim-vec (tensors-from-net (witness (rs ! 1) (rs ! 2) (tl (tl (tl rs)))))) > 0
  using output-size-correct-tensors[of witness (rs ! 1) (rs ! 2) (tl (tl (tl rs)))]
  witness-is-deep-model[of rs ! 1 rs ! 2 tl (tl (tl rs))]
  valid-deep-model[of rs ! 1 rs ! 2 tl (tl (tl rs))] output-size.simps witness.simps
  by (metis 2 3 One-nat-def rs = r # tl rs) deep-model.elims length-greater-0-conv
list.size(3)
  not-numeral-le-zero nth-Cons-Suc nth-mem)
then have tensors-from-net (witness' (rs ! 1) ((rs ! 2) # tl (tl (tl rs)))) $ 0
  = (tensors-from-net (witness-l (tl rs)) $ 0) ⊗ (tensors-from-net (witness-l (tl rs)) $ 0)
  unfolding witness'..simps tensors-from-net.simps witness-l-tl using index-component-mult
by blast
then show "thesis using" lookup-prod tl-tl by simp
qed

Then we can make the Conv layer step:

show "case"
proof ~
  have valid-net' (witness' (rs ! 1) (tl (tl rs))) by (simp add: witness'.valid)
  have output-size' (witness' (rs ! 1) (tl (tl rs))) = rs ! 1
    by (metis 2 Nitpick.size-list-simp(2) diff-diff-left diff-is-0-eq hd-Cons-tl deep-model'..simps(2)
depth-model.elims length-tl not-less-eq-eq numeral-2-eq-2 numeral-3-eq-3 one-add-one
output-size.simps(2) output-size.simps(3) tl-Nil witness'.is-deep-model)
  have if-resolve:(if length (tl (tl rs)) = 0 then id-matrix else if length (tl (tl rs))
    = 1 then all1-matrix else copy-first-matrix) = copy-first-matrix
    by (metis 2 Cons.prems(2) Nitpick.size-list-simp(2) One-nat-def Suc-n-not-le-n
not-numeral-le-zero numeral-3-eq-3)
  have tensors-from-net (Conv (copy-first-matrix (rs ! 0) (rs ! 1)) (witness' (rs ! 1) (tl (tl rs)))) $ j =
    tensors-from-net (witness' (rs ! 1) (tl (tl rs))) $ 0
    using tensors-from-net-Conv-copy-first[OF valid-net' (witness' (rs ! 1) (tl (tl rs))) j < rs ! 0], unfolded output-size' (witness' (rs ! 1) (tl (tl rs))) = rs ! 1]
    using 4 One-nat-def rs = r # tl rs nth-Cons-Suc by metis
  then show "thesis unfolding" witness.simps if-resolve output-size' (witness' (rs ! 1) (tl (tl rs))) = rs ! 1
    using lookup-witness' valid-net' (witness' (rs ! 1) (tl (tl rs))) hd-conv-nth output-size-correct-tensors
    by fastforce
qed

qed
lemma witness-matricization:
assumes $i < \text{dim-row } Aw'$ and $j < \text{dim-col } Aw'$
shows $Aw' \equiv (i, j) \in \{ \text{digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{n. even } n \} \} \text{ i}. \text{ i0 < last (butlast rs) then 1 else 0)}$
proof

define is where is = weave \{ \text{n. even } n \}
\begin{align*}
& \text{(digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{n. even } n \} \) i) \\
& \text{(digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{n. odd } n \} \) j)
\end{align*}

have lookup-eq: $Aw' \equiv (i, j) = \text{Tensor.lookup } Aw$ is
using $Aw'$-def matricize-def $\text{dims-Aw'}(1) \equiv [\text{symmetric, unfolded A-def } \text{dims-Aw'}(2) \equiv [\text{symmetric, unfolded A-def } \text{Collect-neg-eq}]
index-mat(1) \equiv [\text{OF } i < \text{dim-row } Aw', j < \text{dim-col } Aw') \equiv \text{def } \text{Collect-neg-eq}
\text{case-prod-conv}
\text{by } \text{metis (no-types) } Aw'$-def \text{Collect-neg-eq } \text{case-prod-conv is-def matricize-def}
have is $\% Tensor.dims Aw
using \text{is-def valid-index-weave } A\text{-def } \text{Collect-neg-eq } \text{assms } \text{digit-encode-valid-index}
\text{dims-Aw'} \equiv \text{by } \text{metis}

have even (order $Aw$)
unfolding $Aw$-def using assms $\text{dims-output-witness even-numeral le-eq-less-or-eq}$
umeral-2-eq-2\numeral-3-eq-2\numeral-3-eq-3\text{deep } no-zeros y-valid \text{ by fastforce}

have $\text{nths-dimsAw: } nths \{ \text{Tensor.dims } Aw \} \equiv nths \{ \text{Tensor.dims } Aw \} \{ \text{n. odd } n \}$
proof
have $0: \% Tensor.dims \{ \text{tensors-from-net } (\text{witness-l } rs) \equiv \text{replicate } (2 ^ \text{length rs - 2}) \} (\text{last rs})$
using $\text{dims-output-witness[OF - no-zeros y-valid]}$ using deep by linarith
show $?thesis$ unfolding $A$-def
using $nths$-replicate
by $\text{metis (no-types, lifting) 0 } Aw$-def $\langle \text{even (order } Aw) \rangle$ $\text{length}$-replicate
\text{length-nths-even)}
qed

have $i = j \mapsto nths \{ \text{Collect even } \equiv nths \{ \text{n. odd } n \}$
proof
have $\text{eq-lengths: } \text{length } \text{digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{n. even } n \} \equiv \text{length } \text{digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{n. odd } n \}$
unfolding $\text{length}$-$\text{digit}$-$\text{encode}$ by $\text{metis (even (order } Aw) \equiv \text{length}$-$\text{nths-even)}$

then show $i = j \mapsto nths \{ \text{Collect even } \equiv nths \{ \text{n. odd } n \}$ unfolding $\text{is-def}$
using $nths$-$\text{weave}$ of $\text{digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{Collect even } \}$ $i$
\text{Collect even } $\text{digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{n. odd } n \}$ $j$, unfolded $\text{eq-lengths}$, unfolded $\text{Collect-neg-eq[symmetric] card-even mult-2[ symmetric] card-odd}$
nths-dimsAw by simp
show $nths \{ \text{Collect even } \equiv nths \{ \text{n. odd } n \}$ $\mapsto i = j$ unfolding $\text{is-def}$
using $nths$-$\text{weave}$ of $\text{digit-encode } nths \{ \text{Tensor.dims } Aw \} \{ \text{Collect even } \}$ $i$

81
Collect even digit-encode (nths (Tensor.dims Aw) \{n. odd n\}) j, unfolded eq-lengths, unfolded Collect-neg-eq[symmetric] card-even mult-2[symmetric] card-odd]
using nths (Tensor.dims Aw) (Collect even) = nths (Tensor.dims Aw) \{n. odd n\},
depth no-zeros y-valid assms digit-decode-encode dims-Aw'
by auto (metis digit-decode-encode-lt)
qed

have i=j \implies set (digit-encode (nths (Tensor.dims Aw) \{n. even n\}) i) = set is unfolding is-def nths-dimsAw
using set-weave[of (digit-encode (nths (Tensor.dims Aw) \{n. odd n\}) j) Collect even
(digit-encode (nths (Tensor.dims Aw) \{n. odd n\}) j), unfolded mult-2[symmetric] card-even Collect-neg-eq[symmetric] card-odd]
Un-absorb card-even card-odd mult-2 by blast
then show ?thesis unfolding lookup-eq
using witness-tensor[OF \{is < Tensor.dims Aw\}]
by (simp add: A-def ((i = j) = (nths is (Collect even) = nths is \{n. odd n\})))
qed
definition rows-with-1 = \{i. (\forall i0\in \text{set (digit-encode (nths (Tensor.dims Aw) \{n. even n\}) i). i0 < last (butlast rs))}\}
lemma card-low-digits:
assumes \(m \geq 0 \land d. d\in \text{set ds} \implies m \leq d\)
shows card \(\{i. i < \text{prod-list ds } \wedge (\forall i0\in \text{set (digit-encode ds i). i0 < m})\} = m \sim (\text{length ds})\)
using assms proof (induction ds)
case Nil
then show \(?case\ using \text{prod-list.Nil}\ by \text{simp}\)
next
case (Cons d ds)
define low-digits:
where \(\text{low-digits \ dm i } \leftrightarrow i < \text{prod-list ds } \wedge (\forall i0\in \text{set (digit-encode ds i). i0 < m})\) for ds i
have card \(\{i. \text{low-digits ds i}\} = m \sim (\text{length ds})\) unfolding low-digits-def
by (simp add: Cons.IH Cons.prems(1) Cons.prems(2))
have card \(\{i. \text{low-digits \ (d \# ds) i}\} = \text{card} ((\ldots < m) \times \{i. \text{low-digits ds i}\})\)
proof -
define f where \(f p = \text{fst p + ds * snd p for p}\)
have inj-on f \((\ldots < m) \times \{i. \text{low-digits ds i}\}\)
proof (rule inj-onI)
fix \(x y\) assume \(x \in (\ldots < m) \times \{i. \text{low-digits ds i}\}\) \(y \in (\ldots < m) \times \{i. \text{low-digits ds i}\}\)
have \(f x = f y\)
proof
fix \(x y\) assume \(x < m \text{ snd} y < m\) by auto
then have \(\text{fst x < d \text{ snd y < d}}\) using Cons(3) by (meson list.set-intros(1) not-le order-trans)+
then have \( f \ x \mod d = \text{fst} \ x \ y \mod d = \text{fst} \ y \) unfolding f-def by simp-all
have \( f \ x \div d = \text{snd} \ x \ y \div d = \text{snd} \ y \) using \( \langle f \ x = f \ y \rangle \ (f \ x \mod d = \text{fst} \ x) \own f \text{def} \ by \ auto \)
show \( x = y \) using \( \langle f \ x = f \ y \rangle \ (f \ x \div d = \text{snd} \ x) \ (f \ x \mod d = \text{fst} \ x) \) f-def by auto
qed
have \( f \ ('\{..<m\}\times\{i. \ low-digits \ ds \ i\}) = \{i. \ low-digits \ (d \ # \ ds) \ i\} \)
proof (rule subset-antisym; rule subsetI)
fix \( x \) assume \( x \in f \ ('\{..<m\}\times\{i. \ low-digits \ ds \ i\}) \)
then obtain \( i0 \ i1 \) where \( x = i0 + d \ast i1 \) i0 < m low-digits ds i1 using f-def by force
then have \( i0<i \) unfolding low-digits-def by (meson list.set-intros(1) not-le order-trans)
show \( x \in \{i. \ low-digits \ (d \ # \ ds) \ i\} \) unfolding low-digits-def
proof (rule; rule conjI)
have \( i1 < \text{prod-list} \ ds \ \forall i0\in\text{set} \ (\text{digit-encode} \ ds \ i1) \). i0 < m \ using \( \langle \text{low-digits} \ ds \ i1 \rangle \) low-digits-def by auto
show \( x < \text{prod-list} \ (d \ # \ ds) \) unfolding prod-list.Cons \( \langle x = i0 + d \ast i1 \rangle \)
using \( \langle i0<d \rangle \langle i1 < \text{prod-list} \ ds \rangle \)
proof -
have \( d \neq 0 \)
by (metis \( i0 < d \) gr-implies-not0)
then have \( i0 + d \ast i1 \) div \( d \ast \text{prod-list} \ ds \) = 0
by (simp add: Divides.div-mul2-eq i0 < d; i1 < prod-list ds)
then show \( i0 + d \ast i1 < d \ast \text{prod-list} \ ds \)
by (metis (no-types) \( i0 < d \) i1 < prod-list ds) div-eq-0-iff gr-implies-not0 no-zero-divisors)
qed
show \( \forall i0\in\text{set} \ (\text{digit-encode} \ (d \ # \ ds) \ x) \). i0 < m \ using \( \forall i0\in\text{set} \ (\text{digit-encode} \ ds \ i1) \). i0 < m \langle i0 < d \rangle i0 < m \langle x = i0 + d \ast i1 \rangle \) by auto
qed
next
fix \( x \) assume \( x \in \{i. \ low-digits \ (d \ # \ ds) \ i\} \)
then have \( x < \text{prod-list} \ (d \ # \ ds) \ \forall i0\in\text{set} \ (\text{digit-encode} \ (d \ # \ ds) \ x) \). i0 < m \ using low-digits-def by auto
have \( x \mod d < m \) using \( \forall i0\in\text{set} \ (\text{digit-encode} \ (d \ # \ ds) \ x) \). i0 < m unfolding digit-encode.simps by simp
have \( x \div d < \text{prod-list} \ ds \) using \( x < \text{prod-list} \ (d \ # \ ds) \) unfolding prod-list.Cons
by (metis div-eq-0-iff div-mul2-eq mult-0-right not-le0)
have \( \forall i0\in\text{set} \ (\text{digit-encode} \ ds \ (x \div d)) \). i0 < m \ by (simp add: \( \forall i0\in\text{set} \ (\text{digit-encode} \ (d \ # \ ds) \ x) \). i0 < m)
have \( f \ ('\{x \mod d\}\ (x \div d) \) = x \) by (simp add: f-def)
show \( x \in f \ ('\{..<m\}\times\{i. \ low-digits \ ds \ i\}) \) by (metis SigmaI \( \forall i0\in\text{set} \ (\text{digit-encode} \ ds \ (x \div d)) \). i0 < m \langle x = \text{mod} \ d, x \div d \rangle = x \) x \div d < prod-list ds \langle x \mod d < m \) image-eqI lessThan-iff low-digits-def mem-Collect-eq)
qed
then have bij-betw \( f \ ('\{..<m\}\times\{i. \ low-digits \ ds \ i\}) \) \( \{i. \ low-digits \ (d \ # \ ds) \ i\} \) by (simp add: inj-on f ('\{..<m\}\times\{i. \ low-digits \ ds \ i\}) bij-betw-def)
then show \( ? \text{thesis by (simp add: bij-betw-same-card}) \)
qed
then show \( ? \text{case unfolding (card [i. low-digits ds i] = m ^ (length ds)}} \)
card-cartesian-product using low-digits-def by simp
qed

lemma card-rows-with-1: card \( \{i \in \text{rows-with-1. i} < \dim-row \text{Aw}' \} = r \wedge N\text{-half} \)
proof
have \( 1:\{i \in \text{rows-with-1. i} < \dim-row \text{Aw}' \} = \{i. i < \text{prod-list (nths (\text{Tensor.dims Aw}) (Collect even))} \} \) \( \langle \forall i0 \in \text{set (digit-encode (nths (\text{Tensor.dims Aw}) (Collect even)) i). i0 < r} \rangle \) \( (\text{is} ?A = ?B) \)
proof (rule subset-antisym; rule subsetI)
  fix \( i \) assume \( i \in ?A \)
  then have \( i < \dim-row \text{Aw}' \langle \forall i0 \in \text{set (digit-encode (nths (\text{Tensor.dims Aw}) \{n. even n\}) i). i0 < last (\text{butlast rs})} \)
  using rows-with-1-def by auto
  then have \( i < \text{prod-list (nths (\text{dims Aw}) (Collect even))} \) \( \text{using dims-Aw'} \) \( \text{by linarith} \)
  then have \( \text{digit-encode (nths (\text{dims Aw}) (Collect even)) i} \in \text{nths (\text{dims Aw})} \)
  (Collect even)
  using digit-encode-valid-index by auto
  have \( \langle \forall i0 \in \text{set (digit-encode (nths (\text{Tensor.dims Aw}) \{n. even n\}) i). i0 < r} \rangle \)
  proof
    fix \( i0 \) assume \( i0 \in \text{set (digit-encode (nths (\text{dims Aw}) (Collect even)) i)} \)
    then obtain \( k \) where \( k < \text{length (digit-encode (nths (\text{dims Aw}) (Collect even)) i)} \)
    digit-encode (nths (\text{dims Aw}) (Collect even)) i ! k = i0 by (meson in-set-conv-nth)
    have \( i0 < \text{last (butlast rs)} \)
    using \( \langle \forall i0 \in \text{set (digit-encode (nths (\text{dims Aw}) (Collect even)) i). i0 < last (\text{butlast rs})} \rangle \) \( \text{by blast} \)
    have set (nths (\text{dims Aw}) (Collect even)) \( \subseteq \{\text{last rs}\} \) unfolding dims-Aw
    using subset-eq by fastforce
  then have \( \text{nths (\text{dims Aw}) (Collect even)} \) \( \langle k = \text{last rs} \) \( \text{using (\text{digit-encode (nths (\text{dims Aw}) (Collect even)) i} < \text{nths (\text{dims Aw}) (Collect even)})} \)
  \( \langle k < \text{length (digit-encode (nths (\text{dims Aw}) (Collect even)) i)} \rangle \)
  nth-mem valid-index-length by auto
  then have \( i0 < \text{last rs} \)
  using valid-index-lt (digit-encode (nths (\text{dims Aw}) (Collect even)) i ! k = i0)
  (digit-encode (nths (\text{dims Aw}) (Collect even)) i) < nths (\text{dims Aw}) (Collect even))
  \( \langle k < \text{length (digit-encode (nths (\text{dims Aw}) (Collect even)) i)} \rangle \) \( \text{valid-index-length by fastforce} \)
  then show \( i0 < r \) unfolding r-def by (simp add: \( i0 < \text{last (butlast rs)}) \)
qed
then show \( i \in ?B \) using \( \langle i < \text{prod-list (nths (\text{dims Aw}) (Collect even)}) \rangle \) \( \text{by} \)

84
blast

next
  fix i assume i∈?B
then show i∈?A by (simp add: dims-Aw' r-def rows-with-1-def)
qed
have 2:∀d. d ∈ set (nths (Tensor.p_dims Aw) (Collect even)) ⟹ r ≤ d
proof –
  fix d assume d ∈ set (nths (Tensor.p_dims Aw) (Collect even))
then have d ∈ set (Tensor.p_dims Aw) using in-set-nthsD by fast
then have d = last rs using dims-Aw by simp
then show r ≤ d by (simp add: r-def)
qed

have 3:0 < r unfolding r-def by (metis deep diff-diff-cancel diff-zero dual-order.trans
in-set-butlastD last-in-set length-butlast list.size(3) min-def nat-le-linear no-zeros
not-numeral-le-zero numeral-le-one-iff rel-simps(3))
proof –
  fix i assume dvd-i: listpr dvd i
{k
  fix i0::nat
  assume i0∈set (digit-encode (nths (Tensor.p_dims Aw) {n. even n}) i)
then have i0=0 using digit-encode-0 dvd-i listpr-def by auto
then have i0 < last (butlast rs) using deep no-zeros
by (metis Nitpick.size-list-simp(2) One-nat-def Suc-le-lessD in-set-butlastD
last-in-set length-butlast length-tl not-numeral-less-zero numeral-2-eq-2 numeral-3-eq-3
numeral-le-one-iff semiring-norm(70))
}
then show i∈rows-with-1 by (simp add: rows-with-1-def)
qed

lemma infinite-rows-with-1: infinite rows-with-1
proof –
  define listpr where listpr = prod-list (nths (Tensor.p_dims Aw) {n. even n})
  have ∃i. listpr dvd i ⟷ i ∈ rows-with-1
proof –
  fix i assume dvd-i: listpr dvd i
{
  fix i0::nat
  assume i0∈set (digit-encode (nths (Tensor.p_dims Aw) {n. even n}) i)
  then have i0=0 using digit-encode-0 dvd-i listpr-def by auto
  then have i0 < last (butlast rs) using deep no-zeros
  by (metis Nitpick.size-list-simp(2) One-nat-def Suc-le-lessD in-set-butlastD
  last-in-set length-butlast length-tl not-numeral-less-zero numeral-2-eq-2 numeral-3-eq-3
  numeral-le-one-iff semiring-norm(70))
}
  then show i∈rows-with-1 by (simp add: rows-with-1-def)
qed

have 0:Tensor.p_dims Aw = replicate (2 ^ (length rs - 2)) (last rs) unfolding
Aw-def
  using dims-output-witness[OF - no-zeros y-valid] using deep by linarith
then have listpr > 0 unfolding listpr-def 0
by (metis 0 deep last-in-set length-greater-0-cone less-le-trans no-zeros dims-Aw' pow(1)
dims-Aw'(1)
  zero-less-numeral zero-less-power)
then have inj (∗) listpr by (metis inj1 mult-left-cancel neq0_conj)
then show ?thesis using (∃i. listpr dvd i ⟷ i ∈ rows-with-1)
by (meson dvd-triv-left image-subset-iff infinite-iff-countable-subset)

qed

lemma witness-submatrix: submatrix Aw' rows-with-1 rows-with-1 = 1_m (r^N-half)
proof
show dim-row (submatrix Aw' rows-with-1 rows-with-1) = dim-row (1_m (r^N-half))
  unfolding index-one-mat(2) dim-submatrix(1)
  by (metis (full-types set-le-in card-rows-with-1)
  show dim-col (submatrix Aw' rows-with-1 rows-with-1) = dim-col (1_m (r^N-half))
    by (metis dim-row (submatrix Aw' rows-with-1 rows-with-1) = dim-row (1_m (r^N-half))

show \(\forall i. j. i < \text{dim-row} (1_m (r^N-half)) \implies j < \text{dim-col} (1_m (r^N-half)) \implies \text{submatrix} Aw' \text{rows-with-1} \text{rows-with-1}\)

\[ (i, j) = 1_m (r^N-half) \] \(\text{then have} \ (i < r^N-half) \implies \text{card} \ \{i \in \text{rows-with-1}. i < \text{dim-row} Aw'\} = \text{card} \ \{i < \text{dim-col} Aw'\}\)

using card-rows-with-1 dims-Aw'-pow by auto
then have \(\text{pick rows-with-1} i < \text{dim-row} Aw' \text{pick rows-with-1} j < \text{dim-col} Aw'\)
  using card-le-pick-inf[of infinite-rows-with-1, of dim-row Aw' i]
  using card-le-pick-inf[of infinite-rows-with-1, of dim-col Aw' j] by force+

have \(\forall i0 \in \text{set} (\text{digit-encode (nths (dims Aw)} (\text{Collect even})) (\text{pick rows-with-1} i). i0 < \text{last (butlast rs)}\)
  using infinite-rows-with-1 pick-in-set-inf rows-with-1-def by auto
then have \(\text{Aw'} (\text{pick rows-with-1} i, \text{pick rows-with-1} j) = (\text{if pick rows-with-1} i = \text{pick rows-with-1} j \text{then 1 else 0})\)
  using witness-matricization[of pick rows-with-1 i < dim-row Aw' j < dim-col Aw'] by simp
then have \(\text{submatrix} Aw' \text{rows-with-1} \text{rows-with-1} (i, j) = (\text{if pick rows-with-1} i = \text{pick rows-with-1} j \text{then 1 else 0})\)
  using submatrix-index by (metis (no-types, lifting)
    \(\text{dim-col (submatrix} Aw' \text{rows-with-1} \text{rows-with-1}) = \text{dim-col} (1_m (r^N-half))\)
    \(\text{dim-row (submatrix} Aw' \text{rows-with-1} \text{rows-with-1}) = \text{dim-row} (1_m (r^N-half))\)

\(\text{then have} \ \text{submatrix} Aw' \text{rows-with-1} \text{rows-with-1} (i, j) = (\text{if i = j then 1 else 0})\)
  using pick-eq-iff-inf[of infinite-rows-with-1] by auto
then show \(\text{submatrix} Aw' \text{rows-with-1} \text{rows-with-1} (i, j) = 1_m (r^N-half)\)
  \(\text{by (simp add: i < r^N-half, j < r^N-half)}\)
qed

qed

86
lemma witness-det: \( \text{det} (\text{submatrix} \ A_{w}' \ \text{rows-with-1} \ \text{rows-with-1}) \neq 0 \) unfolding witness-submatrix by simp

end

interpretation example : deep-model-correct-params False \([10,10,10]\) unfolding deep-model-correct-params-def by simp


15 Polynomials representing the Deep Network Model


lemma polyfun-det:
assumes \( \forall x. (A \ x) \in \text{carrier-mat} \ n \ n \)
assumes \( \forall i \ j. i<n \implies j<n \implies \text{polyfun} N (\lambda x. (A \ x) \ (i,j)) \)
shows \( \text{polyfun} N (\lambda x. \text{det} (A \ x)) \)
proof –
\[
\begin{align*}
\text{fix } p & \text{ assume } p \in \{p. \ p \text{ permutes } \{0..<n\}\} \\
\text{then have } & p \text{ permutes } \{0..<n\} \text{ by auto} \\
\text{then have } & \forall x. x < n \implies p x < n \text{ using } \text{permutes-in-image} \text{ by auto} \\
\text{then have } & \text{polyfun} N (\lambda x. \prod i = 0..<n. A x \ (i, p i)) \\
\text{using } & \text{polyfun-Prod[\{0..<n\} N \lambda i x. A x \ (i, p i)] asms by simp} \\
\text{then have } & \text{polyfun} N (\lambda x. \text{signof} p * (\prod i = 0..<n. A x \ (i, p i))) \text{ using polyfun-const polyfun-mult by blast} \\
\end{align*}
\]
moreover have \( \text{finite } \{i. i \text{ permutes } \{0..<n\}\} \) by (simp add: finite-permutations)
ultimately show \( \text{thesis} \) unfolding det-def[OF asms(1)]
using polyfun-Sum[OF \{finite \( \{i. i \text{ permutes } \{0..<n\}\}\), of N \( \lambda x \text{ signof} p * (\prod i = 0..<n. A x \ (i, p i))\)]
by blast
qed

lemma polyfun-extract-matrix:
assumes \( i<m \ j<n \)
shows \( \text{polyfun} \ \{..<a + (m * n + c)\} (\lambda f. \text{extract-matrix} (\lambda i. \ f (i + a)) \ m \ n \ (i,j)) \)
unfolding \textit{index-extract-matrix}[OF \textit{assms}] \textbf{apply} (rule \textit{polyfun-single}) \textbf{using} \textit{two-digit-le}[OF \textit{assms}] \textbf{by} \textit{simp}

\textbf{lemma} \textit{polyfun-mult-mat-vec}:
\begin{enumerate}
\item \textit{assumes} \(\forall x. v \in \text{carrier-vec} n\)
\item \textit{assumes} \(\forall j. j < n \Rightarrow \text{polyfun} (\lambda x. v \cdot j)\)
\item \textit{assumes} \(\forall x. A = \text{carrier-mat} m n\)
\item \textit{assumes} \(\forall i. i < m \Rightarrow j < n \Rightarrow \text{polyfun} (\lambda x. A \cdot (i, j))\)
\end{enumerate}
\textit{shows} \(\text{polyfun} (\lambda x. ((A \cdot v) \cdot (v \cdot x)) \cdot j)\)

\textbf{proof} –
\begin{enumerate}
\item \textit{have} \(\forall x. j < \text{dim-row} (A \cdot x) \text{ using} (j < m) \text{ carrier-MatD(1) by force}\)
\item \textit{have} \(\forall x. n = \text{dim-vec} (v \cdot x) \text{ using} \text{assms(1) carrier-vecD by fastforce}\)
\begin{enumerate}
\item \textit{fix} \(i\) \textit{assume} \(i \in \{0..<n\}\)
\item \textit{then have} \(i < n\) \textit{by auto}\)
\begin{enumerate}
\item \textit{fix} \(x\)
\item \textit{have} \(i < \text{dim-vec} (v \cdot x) \text{ using} \text{assms(1) carrier-vecD \text{ \textit{i<n} by fastforce}}\)
\item \textit{have} \(j < \text{dim-row} (A \cdot x) \text{ using} (j < m) \text{ carrier-MatD(1) by force}\)
\item \textit{have} \(\text{dim-col} (A \cdot x) = \text{dim-vec} (v \cdot x) \text{ by} (\text{metis} \text{ assms(1) assms(3) carrier-MatD(2) carrier-vecD})\)
\item \textit{then have} \(\text{row} (A \cdot x) \cdot j \cdot i = A \cdot x \cdot (j, i) \cdot i \cdot n\) \textit{using} \(j < \text{dim-row} (A \cdot x)\)
\item \textit{\langle i < n\rangle by (simp-all add: \text{\langle i < dim-vec (v \cdot x)\rangle})}\)
\end{enumerate}
\item \textit{then have} \(\text{polyfun} (\lambda x. \text{row} (A \cdot x) \cdot j \cdot i \cdot v \cdot x \cdot i)\)
\item \textit{\textit{using} \textit{polyfun-Mult assms(4)[OF \text{\langle j < m\rangle assms(2) by fastforce}}\)
\end{enumerate}
\item \textit{then show} \textit{\langle \text{thesis unfolding} \textit{index-Mult-mat-vec}[OF \\forall x. j < \text{dim-row} (A \cdot x)\rangle]\textit{scalar-prod-def}\)
\begin{enumerate}
\item \textit{using} \textit{polyfun-Sum}[OF \\{0..<n\} \textit{N \lambda i. row (A \cdot x) \cdot j \cdot i \cdot v \cdot x \cdot i}] \textit{finite-atLeastLessThan[of 0 n \\langle \forall x. n = \text{dim-vec} (v \cdot x)\rangle}\}
\item \textit{by simp}\)
\end{enumerate}
\end{enumerate}
\textbf{qed}

\textbf{lemma} \textit{polyfun-evaluate-net-plus-a}:
\begin{enumerate}
\item \textit{assumes} \(\text{map dim-vec inputs = input-sizes m}\)
\item \textit{assumes} \(\text{valid-net m}\)
\item \textit{assumes} \(j < \text{output-size m}\)
\item \textit{shows} \(\text{polyfun} \{..<a + \text{count-weights s m} \text{ (\lambda f. evaluate-net (insert-weights s m (\lambda i. f (i + a))) inputs \$ j)\)
\item \textit{using} \textit{assms proof} \(\text{(induction m arbitrary:inputs j a)\)
\item \textit{case} \textit{(Input)\)
\item \textit{then show} \textit{\langle case unfolding insert-weights.simps evaluate-net.simps using polyfun-const by metis\)
\item \textit{next\)
\item \textit{case} \textit{(Conv x m)\)
\item \textit{then obtain} \(x1 x2\) \textit{where} \(x = (x1,x2)\) \textit{by fastforce\)
\end{enumerate}
show ?case unfolding \( \langle x = (x_1, x_2) \rangle \) insert-weights.simps evaluate-net.simps drop-map
unfolding list-of-vec-index
proof (rule polyfun-mult-mat-vec)
  
  fix \( f \)
  have 1:valid-net' (insert-weights s m (\( \lambda i. f (i + x_1 * x_2) \)))
  using \( \langle \vdots \rangle \) valid-net (Conv x m) valid-net.simps by (metis convnet.distinct(1) connect.distinct(5) convnet.inject(2) remove-insert-weights)
  have 2:map dim-vec inputs = input-sizes (insert-weights s m (\( \lambda i. f (i + x_1 * x_2) \)))
  using input-sizes-remove-weights remove-insert-weights
  by (simp add: Conv.prems(1))
  have dim-vec (evaluate-net (insert-weights s m (\( \lambda i. f (i + x_1 * x_2) \))) inputs)
  = output-size m
  using output-size-correct[\( \langle \vdots \rangle \)] unfolding vec-of-list-index count-weights.simps
  then show evaluate-net (insert-weights s m (\( \lambda i. f (i + x_1 * x_2) \))) inputs \( \in \)
  carrier-vec (output-size m)
  using carrier-vec-def by (metis (full-types) mem-Collect-eq)
  
  have map dim-vec inputs = input-sizes m by (simp add: Conv.prems(1))
  have valid-net m using Conv.prems(2) valid-net.cases by fastforce
  show \( \bigwedge j. j < \text{output-size } m \Rightarrow \text{polyfun } \{ \langle \vdots \rangle < a + \text{count-weights } s \text{ (Conv } x_1, x_2) m \} \)
  (\( \lambda f. \text{evaluate-net (insert-weights } s m \text{ (\( \lambda i. f (i + x_1 * x_2 + a) \))) inputs} \) \$ j)
  unfolding vec-of-list-index count-weights.simps
  using Conv.prems(1) unfolding vec-of-list-index count-weights.simps
  then show evaluate-net (insert-weights s m (\( \lambda i. f (i + x_1 * x_2) \))) inputs \( \in \)
  \( \langle \vdots \rangle \) valid-net m, of - x1 * x2 + a
  unfolding semigroup-add-class.add.assoc ab-segment-segment-class.add.commute[of x1 * x2 a]
  by blast
  
  have output-size m = x2 using Conv.prems(2) \( \langle x = (x_1, x_2) \rangle \) valid-net.cases by fastforce
  show \( \bigwedge j. \text{extract-matrix } (\langle \vdots \rangle \text{ (\( \lambda i. f (i + a) \))) x_1 x_2} \text{ \( \in \) carrier-mat } x_1 \text{ (output-size m) \( \rangle \) unfolding \( \langle \vdots \rangle \) output-size m = x2 \rangle \) using dim-extract-matrix
  using carrier-mat by (metis (no-types, lifting))
  
  show \( \bigwedge i. j < x_1 \Rightarrow j < \text{output-size } m \Rightarrow \text{polyfun } \{ \langle \vdots \rangle < a + \text{count-weights } s \text{ (Conv } x_1, x_2) m \} \) (\( \lambda f. \text{extract-matrix } (\langle \vdots \rangle \text{ (\( \lambda i. f (i + a) \))) x_1 x_2} \text{ \( \in \) } \langle \vdots \rangle \text{ valid-net m, of - } x_1 \text{ - } x_2 \text{ a count-weights } s m \} \) by blast
  
  show j < x1 using Conv.prems(3) \( \langle x = (x_1, x_2) \rangle \) by auto
  qed
next
  case (Pool m1 m2 inputs j a)
  have A2: \( \langle \vdots \rangle \) map dim-vec (take (length (input-sizes (insert-weights s m1 \( \langle \vdots \rangle \) (\( \lambda i. f (i + a) \)))) inputs) = input-sizes m1

89
by (metis Pool.prems(1) append-eq-conv-conj input-sizes.simps(3) input-sizes-remove-weights remove-insert-weights take-map)
have B2:∀ f. dim-vec (drop (length (input-sizes (insert-weights s m1 (λ i. f (i + a)))))) inputs = input-sizes m2
  using Pool.prems(1) append-eq-conv-conj input-sizes.simps(3) input-sizes-remove-weights remove-insert-weights by (metis drop-map)
have A3:valid-net m1 and B3:valid-net m2 using <valid-net (Pool m1 m2)>
valid-net.simps by blast+
have output-size (Pool m1 m2) = output-size m2 unfolding output-size.simps
  using <valid-net (Pool m1 m2)> valid-net.cases by fastforce
then have A4;j < output-size m1 and B4;j < output-size m2 using <j < output-size (Pool m1 m2)> by simp-all

let ?net1 = λf. evaluate-net (insert-weights s m1 (λi. f (i + a)))
(take (length (input-sizes (insert-weights s m1 (λi. f (i + a)))))) inputs
let ?net2 = λf. evaluate-net (insert-weights s m2 (if s then λi. f (i + a) else
(λi. f (i + count-weights s m1 + a))))
(drop (length (input-sizes (insert-weights s m1 (λi. f (i + a))))) inputs)
have length1:∀ f. output-size m1 = dim-vec (?net1 f)
by (metis A2 A3 input-sizes-remove-weights output-size-correct remove-insert-weights)
then have jlength1:∀ f. j < dim-vec (?net1 f) using A4 by metis
have length2:∀ f. output-size m2 = dim-vec (?net2 f)
by (metis B2 B3 input-sizes-remove-weights output-size-correct remove-insert-weights)
then have jlength2:∀ f. j < dim-vec (?net2 f) using B4 by metis
have cong1:∀xf. (λf. evaluate-net (insert-weights s m1 (λi. f (i + a))))
(take (length (input-sizes (insert-weights s m1 (λi. xf (i + a)))))) inputs $ j
= (λf. ?net1 f $ j)
  using input-sizes-remove-weights remove-insert-weights by auto
have cong2:∀xf. (λf. evaluate-net (insert-weights s m2 (λi. f (i + a + (if s then 0 else count-weights s m1)))))
(drop (length (input-sizes (insert-weights s m1 (λi. xf (i + a)))))) inputs $ j
= (λf. ?net2 f $ j)
unfolding semigroup-add-class.add.assoc[symmetric] ab-semigroup-add-class.add.commute[of a if s then 0 else count-weights s m1]
using input-sizes-remove-weights remove-insert-weights by auto

show ?case unfolding insert-weights.simps evaluate-net.simps count-weights.simps
  unfolding index-component-mult[OF jlength1 jlength2]
apply (rule polyfun-mult)
using Pool.IH(1)[OF A2 A3 A4, of a, unfolded cong1]
apply (simp add:polyfun-subset[of ..<a + count-weights s m1] ..<a + (if s then max (count-weights s m1) (count-weights s m2) else count-weights s m2))
using Pool.IH(2)[OF B2 B3 B4, of a + (if s then 0 else count-weights s m1), unfolded cong2 semigroup-add-class.add.assoc[of a]]
by (simp add:polyfun-subset[of ..<a + ((if s then 0 else count-weights s m1) + count-weights s m2)] ..<a + (if s then max (count-weights s m1) (count-weights s m2))
s m2) else count-weights s m1 + count-weights s m2])])
qed

lemma polyfun-evaluate-net:
assumes map dim-vec inputs = input-sizes m
assumes valid-net m
assumes j < output-size m
shows polyfun \{{..<count-weights s m}} (\lambda f. evaluate-net (insert-weights s m f) inputs $ j)
using polyfun-evaluate-net-plus-a[where a=0, OF assms] by simp

lemma polyfun-tensors-from-net:
assumes valid-net m
assumes is = input-sizes m
assumes j < output-size m
shows polyfun \{{..<count-weights s m}} (\lambda f. Tensor.lookup (tensors-from-net (insert-weights s m f) $ j) is)
proof −
have 1:∀f. valid-net' (insert-weights s m f) by (simp add: assms(1) remove-insert-weights)
have input-sizes:∀f. input-sizes (insert-weights s m f) = input-sizes m
  unfolding input-sizes-remove-weights by (simp add: remove-insert-weights)
have 2:∀f. is < input-sizes (insert-weights s m f)
  unfolding input-sizes using assms(2) by blast
have 3:∀f. j < output-size' (insert-weights s m f)
  by (simp add: assms(3) remove-insert-weights)
have 1|f1 f2. base-input (insert-weights s m f1) is = base-input (insert-weights s m f2) is
  unfolding base-input-def by (simp add: input-sizes)
then have 1|zf. (λf. evaluate-net (insert-weights s m f) (base-input (insert-weights s m zf) is) $ j)
  = (λf. evaluate-net (insert-weights s m f) (base-input (insert-weights s m f) is) $ j)
  by metis
then show ?thesis unfolding lookup-tensors-from-net[OF 1 2 3]
  using polyfun-evaluate-net[OF base-input-length[OF 2, unfolded input-sizes, symmetric] assms(1) assms(3), of s]
  by simp
qed

lemma polyfun-matricize:
assumes \forall x. dims (T x) = ds
assumes \forall is. is < ds ⇒ polyfun N (λx. Tensor.lookup (T x) is)
assumes \forall x. dim-row (matricize I (T x)) = nr
assumes \forall x. dim-col (matricize I (T x)) = nc
assumes i < nr
assumes j < nc
shows polyfun N (λx. matricize I (T x) $$ (i,j))
proof −
let ?weave = λ x. (weave I

91
(digit-encode (nths ds I) i)
(digit-encode (nths ds (−I)) j)

have 1: (∀ x. matricize I (T x) $$(i,j) = Tensor.lookup (T x) (?weave x)$$ unfolding matricize-def
by (metis (no-types, lifting) assms(1) assms(3) assms(4) assms(5) assms(6)
case-prod-conv
dim-col-mat(1) dim-row-mat(1) index-mat(1) matricize-def)
have …

lemma (~ (a::nat) < b) = (a ≥ b)
by (metis not-le)

lemma polyfun-submatrix:
assumes (∀ x. (A x) ∈ carrier-mat m n
assumes (∀ i j. i<m ⇒ j<n ⇒ polyfun N (λx. (A x) $$(i,j))
assumes i < card {i. i < m ∧ i ∈ I}
assumes j < card {j. j < n ∧ j ∈ J}
assumes infinite I infinite J
shows polyfun N (λx. (submatrix (A x) I J) $$ (i,j))
proof –
have 1: (∀ x. (submatrix (A x) I J) $$ (i,j) = (A x) $$ (pick I i, pick J j)

using submatrix-index
by (metis (no-types, lifting) Collect-cong assms(1)
assms(3) assms(4) carrier-matD(1) carrier-matD(2))
have pick I i < m pick J j < n using card-le-pick-inf[OF ‹infinite I›] card-le-pick-inf[OF ‹infinite J›]
\langle i < card {i. i < m ∧ i ∈ I}\rangle[unfolded set-le-in] \langle j < card {j. j < n ∧ j ∈ J}\rangle[unfolded set-le-in] not-less by metis+
then show ‹thesis unfolding 1› by (simp add: assms(2))
qed

context deep-model-correct-params-y
begin

definition witness-submatrix where
witness-submatrix f = submatrix (A' f) rows-with-1 rows-with-1

lemma polyfun-tensor-deep-model:
assumes is < input-sizes (deep-model-l rs)
shows polyfun {..<weight-space-dim}
(λf. Tensor.lookup (tensors-from-net (insert-weights shared-weights (deep-model-l rs) f) $ y) is)
proof

have 1: \( \forall f. \\text{remove-weights} (\text{insert-weights shared-weights} (\text{deep-model-l} \ rs) f) = \text{deep-model-l} \ rs \)
using remove-insert-weights by metis
then have \( y < \text{output-size} (\text{deep-model-l} \ rs) \) using valid-deep-model y-valid
length-output-deep-model by force
have 0: \( \{..<\text{weight-space-dim}\} = \{0..<\text{weight-space-dim}\} \) by auto
then show ?thesis unfolding weight-space-dim-def using polyfun-tensors-from-net
assms(1) valid-deep-model
\( y < \text{output-size} (\text{deep-model-l} \ rs) \) by metis
qed

lemma input-sizes-deep-model:
assumes \( i < (\text{last} \ rs)^\text{N-half} \)
assumes \( j < (\text{last} \ rs)^\text{N-half} \)
shows \( \text{polyfun} \ \{..<\text{weight-space-dim}\} \ (\lambda f. A' f \$(i,j)) \)
proof
have 1: \( \forall f. \\text{remove-weights} (\text{insert-weights shared-weights} (\text{deep-model-l} \ rs) f) = \text{deep-model-l} \ rs \)
using remove-insert-weights by metis
have 2: \( (\forall \text{is. is } \text{replicate} (2 * \text{N-half}) (\text{last} \ rs) \implies \text{polyfun} \ \{..<\text{weight-space-dim}\} \ (\lambda x. \text{Tensor.lookup} (A x) \text{is}) \)
unfolding A-def using polyfun-tensor-deep-model[unfolded input-sizes-deep-model]
\( 0 \) by blast
show ?thesis unfolding A'-def A-def apply (rule polyfun-matricize)
using dims-tensor-deep-model[OF 1] 2[unfolded A-def]
using dims-A'-pow[unfolded A'-def A-def] \( i < (\text{last} \ rs)^\text{N-half} \), \( j < (\text{last} \ rs)^\text{N-half} \)
by auto
qed

lemma polyfun-submatrix-deep-model:
assumes \( i < r^\text{N-half} \)
assumes \( j < r^\text{N-half} \)
shows \( \text{polyfun} \ \{..<\text{weight-space-dim}\} (\lambda f. \text{witness-submatrix} f \$(i,j)) \)
unfolding witness-submatrix-def
proof (rule polyfun-submatrix)
have 1: \( \forall f. \\text{remove-weights} (\text{insert-weights shared-weights} (\text{deep-model-l} \ rs) f) = \text{deep-model-l} \ rs \)
using remove-insert-weights by metis
show $\forall f. A' f \in \text{carrier-mat} ((\text{last rs}) ^ {N\text{-half}}) ((\text{last rs}) ^ {N\text{-half}})$
  using 1 $\text{dims-A'}^\text{pow}$ using $\text{weight-space-dim-def}$ by auto
show $\forall f i j. i < \text{last rs} ^ {N\text{-half}} \implies j < \text{last rs} ^ {N\text{-half}} \implies$
  polyfun $\{..<\text{weight-space-dim}\} (\lambda f. A' f \$(i, j))$
  using polyfun-matrix-deep-model weight-space-dim-def by force
show $i < \text{card} \{i. i < \text{last rs} ^ {N\text{-half}} \land i \in \text{rows-with-1}\}$
  using assms(1) card-rows-with-1 $\text{dims-A}'^\text{pow}$ set-le-in by metis
show $j < \text{card} \{i. i < \text{last rs} ^ {N\text{-half}} \land i \in \text{rows-with-1}\}$
  using assms(2) card-rows-with-1 $\text{dims-A}'^\text{pow}$ set-le-in by metis
show infinite rows-with-1 infinite rows-with-1 by (simp-all add: infinite-rows-with-1)
qed

lemma polyfun-det-deep-model:
shows polyfun $\{..<\text{weight-space-dim}\} (\lambda f. \text{det} (\text{witness-submatrix} f))$
proof (rule polyfun-det)
  fix f
  have remove-weights $\text{(insert-weights shared-weights (deep-model-l rs) f)} = \text{deep-model-l rs}$
    using remove-insert-weights by metis

    show witness-submatrix $f \in \text{carrier-mat} (r ^ {N\text{-half}}) (r ^ {N\text{-half}})$
      unfolding witness-submatrix-def apply (rule carrier-matI) unfolding dim-submatrix[unfolded set-le-in]
    unfolding $\text{dims-A'}^\text{pow}[\text{unfolded weight-space-dim-def}]$ using card-rows-with-1 $\text{dims-A'}^\text{pow}$ by simp-all
    show $\forall i j. i < r ^ {N\text{-half}} \implies j < r ^ {N\text{-half}} \implies$
      polyfun $\{..<\text{weight-space-dim}\} (\lambda f. \text{witness-submatrix} f \$(i, j))$
    using polyfun-submatrix-deep-model by blast
qed

end

16 Alternative Lebesgue Measure Definition

theory Lebesgue-Functional
imports HOL-Analysis.Lebesgue-Measure
begin

  Lebesgue_Measure.lborel is defined on the type class euclidean_space, which does not allow the space dimension to be dependent on a variable. As the Lebesgue measure of higher dimensions is the product measure of the one dimensional Lebesgue measure, we can easily define a more flexible version of the Lebesgue measure as follows. This version of the Lebesgue measure measures sets of functions from nat to real whose values are undefined for arguments higher than n. These 'Extensional Function Spaces' are defined in HOL/FuncSet.
definition \( \text{lborel-f} :: \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{real}) \text{ measure} \) 
where 
\( \text{lborel-f} \ n = (\prod_{b \in \{..<n\}} \text{lborel}) \)

lemma \( \text{product-sigma-finite-interval} \): \( \text{product-sigma-finite (} \lambda b. \text{ interval-measure (} \lambda x. x) \) 
unfolding \( \text{product-sigma-finite-def} \) using \( \text{sigma-finite-interval-measure} \) by auto

lemma \( \text{l-borel-f-1} \): \( \text{distr (} \text{lborel-f 1} \) \( \text{lborel (} \lambda x. x 0) = \text{lborel} \) 
unfolding \( \text{lborel-f-def} \) unfolding \( \text{lborel-eq-real lessThan-Suc} \) by auto

lemma \( \text{space-lborel-f} \): \( \text{space (} \text{lborel-f n} = \Pi_{E \{..<n\}} (\lambda - \text{UNIV}) \) 
unfolding \( \text{lborel-f-def} \) unfolding \( \text{space-PiM space-lborel space-borel} \) by metis

lemma \( \text{space-lborel-f-subset} \): \( \text{space (} \text{lborel-f n} \subseteq \text{space (} \text{lborel-f (} \text{Suc n} \) )} 
unfolding \( \text{space-lborel-f} \) by (rule subsetI, rule PiE-I, blast, metis PiE-E Suc-n-not-le-n le-cases lessThan-subset-iff subsetCE)

lemma \( \text{space-lborel-add-dim} \):  
assumes \( f \in \text{space (} \text{lborel-f n} \) 
shows \( f(n:=x) \in \text{space (} \text{lborel-f (} \text{Suc n} \) )} 
unfolding \( \text{space-lborel-f} \) using \( \text{assms lessThan-Suc space-lborel-f} \) by auto

lemma \( \text{integral-lborel-f} \):  
assumes \( f \in \text{borel-measurable (} \text{lborel-f (} \text{Suc n} \) )} 
shows \( \text{integral} \ N (\text{lborel-f (} \text{Suc n} \) \ f = \int_{+} x \ f (x(n := y)) \text{ lborel-f n} \)) 
\( \text{lborel} \) 

lemma \( \text{emeasure-lborel-f-Suc} \):  
assumes \( A \in \text{sets (} \text{lborel-f (} \text{Suc n} \) )} 
assumes \( \lambda y. (x \in \text{space (} \text{lborel-f n} \) \ x(n := y) \in A) \in \text{sets (} \text{lborel-f n} \) ) 
shows \( \text{emeasure (} \text{lborel-f (} \text{Suc n} \) \ A = \int_{+} x \ \text{emeasure (} \text{lborel-f n} \) \ x(n := y) \in A) \text{ lborel} \) 
proof –
\{ 
  fix \( x \ \text{y assume} \ x \in \text{space (} \text{lborel-f n} \) 
  then have \( (\text{indicator} A) (x(n := y)) = (\text{indicator} \ \{x \in \text{space (} \text{lborel-f n} \) \ x(n := y) \in A\}) x \) 
  by (simp add: indicator-def)
\}
then show \( \text{thesis} \) 
unfolding \( \text{nn-integral-indicator[of assms(1), symmetric]} \text{ nn-integral-indicator[of} 

95
assms(2), symmetric]

\[ \text{integral-lborel-f}\{\text{OF borel-measurable-indicator, OF assms(1)}\] using \text{nn-integral-cong by (metis (no-types, lifting))} \]

qed

**Lemma lborel-f-measurable-add-dim:**
\[ (\lambda f(n := x)) \in \text{measurable (lborel-f n) (lborel-f (Suc n))} \]

**Proof**
- have \( x \in \text{space lborel by simp} \)
- have \( \theta:(\lambda f, y). f(n := y)) \circ (\lambda x a. (xa, x)) = (\lambda f(n := x)) \) unfolding \text{comp-def}
- using \text{case-prod-conv by fast}
- show \( \exists \)thesis unfolding \text{lborel-f-def}
- using \text{measurable-comp[OF measurable-Pair2[of x lborel Pi M \{..<n\} (lborel), OF \{x \in space lborels\] measurable-add-dim[of n \{..<n\} lborel], unfolded 0] lessThan-Suc by auto} \]

qed

**Lemma sets-lborel-f-sub-dim:**
- assumes \( A \in \text{sets (lborel-f (Suc n))} \)
- shows \( \{x. x(n := y) \in A\} \cap \text{space (lborel-f n) } \in \text{sets (lborel-f n)} \)
- proof
  - have \( (\lambda f(n := y)) - A \cap \text{space (lborel-f n) } \in \text{sets (lborel-f n)} \) using \text{measurable-sets[OF lborel-f-measurable-add-dim assms] by auto}
  - moreover have \( (\lambda f(n := y)) - A = \{x. x(n := y) \in A\} \) by auto
  - finally show \( \exists \)thesis by \text{metis} \]

qed

**Lemma lborel-f-measurable-restrict:**
- assumes \( m \leq n \)
- shows \( (\lambda f. \text{restrict f } \{..<m\}) \in \text{measurable (lborel-f n) (lborel-f m)} \)
- using \text{measurable-restrict-subset lborel-f-def assms by auto}

**Lemma lborel-measurable-sub-dim:**
\[ (\lambda f. \text{restrict f } \{..<n\}) \in \text{measurable (lborel-f (Suc n)) (lborel-f n)} \]

using \text{lborel-f-measurable-restrict[of n Suc n] by linarith}

**Lemma measurable-lborel-component [measurable]:**
- assumes \( k < n \)
- shows \( (\lambda x. x k) \in \text{borel-measurable (lborel-f n)} \)
  unfolding \text{lborel-f-def using assms measurable-PiM-component-rev by simp-all}

end

17 Lebesgue Measure of Polynomial Zero Sets
Polynomials.MPoly-Type-Univariate
begin

lemma measurable-insertion [measurable]:
assumes vars p ⊆ {..<n}
shows (λf. insertion f p) ∈ borel-measurable (lborel-f n)
using assms proof (induction p rule:mpoly-induct)
case (monom m a)
then show ?case
proof (cases a = 0)
case True
show ?thesis unfolding insertion-single ⟨a = 0⟩ MPoly-Type.monom.abs-eq single-zero
  zero-mpoly.abs-eq[symmetric] insertion-zero by measurable
next
case False
  have Poly-Mapping.keys m ⊆ {..<n} using monom by (simp add: False vars-monom-keys)
than show ?thesis using ⟨a≠0⟩
proof (induction m arbitrary:a rule:poly-mapping-induct)
case (single x i a)
then show ?case
proof (cases i = 0)
case True
  show ?thesis unfolding insertion-single ⟨i = 0⟩ by simp
next
case False
then show ?thesis unfolding insertion-single apply measurable
  using vars-monom-single-cases single False insert-subset lessThan-iff ⟨a≠0⟩
by fastforce
qed
next
case (sum m1 m2 x i)
then have Poly-Mapping.keys m1 ∩ Poly-Mapping.keys m2 = {} by simp
then have Poly-Mapping.keys m1 ∪ Poly-Mapping.keys m2 = Poly-Mapping.keys
  (m1 + m2) using keys-add by metis
then have 1:Poly-Mapping.keys m1 ⊆ {..<n} and 2:Poly-Mapping.keys m2 ⊆ {..<n}
  using sum.prems by auto
  show ?case unfolding MPoly-Type.malt-monom[af m1 a m2 1,simplified,symmetric]
    insertion-malt using sum.IH(1)[OF 1 ⟨a≠0⟩] sum.IH(2)[OF 2, of 1,
    simplified] by measurable
qed
qed
next
case (sum p1 p2 m a)
then have (λf. insertion f p1) ∈ borel-measurable (lborel-f n)
  (λf. insertion f p2) ∈ borel-measurable (lborel-f n)
using var-add-monom[OF sum.prems] le-sap-iff by blast+
then show ?case unfolding insertion-add by measurable

97
This proof follows Richard Caron and Tim Traynor, "The zero set of a polynomial" http://www1.uwindsor.ca/math/sites/uwindsor.ca.math/files/05-03.pdf

lemma lebesgue-mpoly-zero-set:
fixes p::real mpoly
assumes \( p \neq 0 \) vars \( p \subseteq \{..<n\} \)
shows \( \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f \ p = 0\} \in \text{null-sets } (\text{lborel-f } n) \)
using assms proof (induction \( n \) arbitrary:p)
case 0
then have vars \( p \) = {} by simp
then have \( \forall f. \text{insertion } f \ p = \text{MPoly-Type.coeff } p \ 0 \) unfolding insertion-trivial[symmetric] using insertion-irrelevant-vars by blast
have \( \forall m. m \neq 0 \Rightarrow \text{MPoly-Type.coeff } p \ m = 0 \) proof (rule ccontr)
fix \( m :: \text{nat} \Rightarrow \text{nat} \) assume \( m \neq 0 \text{MPoly-Type.coeff } p \ m \neq 0 \)
then obtain \( v \) where \( \text{Poly-Mapping.lookup } m \ v \neq 0 \) using aux by auto
then have \( v \in \text{vars } p \)
unfolding More-MPoly-Type.vars-def using \( \text{coeff } p \ m \neq 0 \) by (meson UN-I coeff-keys lookup-not-eq-zero-eq-in-keys)
then show False using \( \text{vars } p = {} \) by auto
qed
then have \( \text{MPoly-Type.coeff } p \ 0 \neq 0 \)
by (metis coeff-all-0)
then have \( \{f. \text{insertion } f \ p = 0\} = {} \) using \( \forall f. \text{insertion } f \ p = \text{MPoly-Type.coeff } p \ 0 \)
by auto
then show \( \forall \) case by auto
next
case (Suc \( n \))
Show that \( N \) is finite:
then have \( \text{extract-var } p \ n \neq 0 \)
using reduce-nested-mpoly-0
by (metis reduce-nested-mpoly-extract-var)
let \( \text{q } j = \lambda . \text{MPoly-Type.coeff } (\text{extract-var } p \ n) \ j \)
obtain \( j \) where \( \text{q } j \neq 0 \)
using \( \forall \) extract-var \( p \ n \neq 0 \)
by (metis coeff-all-0)
then have finite \( \{x. \text{insertion } (\lambda . x) \ (\text{q } j) = 0\} \)
using univariate-mpoly-roots-finite[OF vars-coeff-extract-var] by metis
then have finite \( \bigcap \{x. \forall j. \text{insertion } (\lambda . x) \ (\text{q } j) = 0\} = (\bigcap \{x. \text{insertion } (\lambda v. x) \ (\text{q } j) = 0\}) \) by blast
moreover have \( \{x. \forall j. \text{insertion } (\lambda . x) \ (\text{q } j) = 0\} = (\bigcap \{x. \text{insertion } (\lambda v. x) \ (\text{q } j) = 0\}) \) by blast
ultimately have finite \( \{x. \forall j. \text{insertion } (\lambda . x) \ (\text{q } j) = 0\} \)
by metis
define \( p\text{-fix1 } x1 = \text{replace-coeff } (\text{insertion } (\lambda . x1)) \ (\text{extract-var } p \ n) \) for \( x1 \)
define \( N \) where \( N = \{x1. \text{p-fix1 } x1 = 0\} \)
have \( N \subseteq \{x. \forall j. \text{insertion } (\lambda . x) \ (\text{q } j) = 0\} \)
proof
  fix x assume x∈N
  then have p-fix1 x = 0 using N-def by auto
  then have \( \forall m. \text{MPoly-Type}.coeff (p-fix1 x) m = 0 \) by (metis More-MPoly-Type.coeff-monom monom-zero when-def)
    have \( \forall j. \text{insertion} (\lambda \cdot x) (?q j) = 0 \)
      using \( \forall m. \text{MPoly-Type}.coeff (p-fix1 x) m = 0 \) unfolded p-fix1-def coeff-replace-coeff[of insertion (\lambda \cdot x), OF insertion-zero]
      by metis
    then show \( x \in \{ x. \forall j. \text{insertion} (\lambda \cdot x) (\text{MPoly-Type}.coeff (\text{extract-var} p n) j) = 0 \} \) by blast
  qed
  then have finite N by (simp add: finite \( \{ x. \forall j. \text{insertion} (\lambda \cdot x) (\text{MPoly-Type}.coeff (\text{extract-var} p n) j) = 0 \} \) finite-subset)
  Use the IH:
  define A where \( A = \{ f \in \text{space} (\text{lborel-f} (\text{Suc} n))\text{. insertion} f p = 0 \} \)
  have \( \forall x1. \text{vars} (p-fix1 x1) \subseteq \{1..<n\} \)
    proof
      fix x1
      have \( \text{vars} (\text{extract-var} p n) \subseteq \{1..<n\} \)
        using \( \forall p \subseteq \{1..<\text{Suc} n\} \) lessThan-Suc v-not-in-vars-extract-var vars-extract-var-subset
        by fastforce
      then show \( \forall x1. \text{vars} (p-fix1 x1) \subseteq \{1..<n\} \) unfolding p-fix1-def
        using vars-replace-coeff[of insertion (\lambda \cdot x1), OF insertion-zero] by blast
    qed
  have set-eq:\( \forall x1. \{ x \in \text{space} (\text{lborel-f} n). x(n := x1) \in A \} = \{ f \in \text{space} (\text{lborel-f} n). \text{insertion} f (p-fix1 x1) = 0 \} \)
    proof
      fix x1
      show \( \forall x1. \{ x \in \text{space} (\text{lborel-f} n). x(n := x1) \in A \} = \{ f \in \text{space} (\text{lborel-f} n). \text{insertion} f (p-fix1 x1) = 0 \} \)
        (proof (rule subset-antisym; rule subsetI)
          fix x assume x \in \( \{ x \in \text{space} (\text{lborel-f} n). x(n := x1) \in A \} \)
          then have insertion (x(n := x1)) p = 0 \( x \in \text{space} (\text{lborel-f} n) \)
            using A-def by auto
          then have insertion x (p-fix1 x1) = 0 unfolding p-fix1-def
            unfolding replace-coeff-extract-var-cong[of \( \lambda \cdot x1 n x(n := x1) \) p, OF fun-upd-same[ symmetric]]
              using insertion-replace-coeff[of x(n := x1)]
              using insertion-irrelevant-vars[of replace-coeff (insertion (x(n := x1))]
              \( \text{extract-var} p n x x(n := x1) \)
            vars-replace-coeff fun-upd-other insertion-zero reduce-nested-mpoly-extract-var subset-eq
              v-not-in-vars-extract-var by metis
          then show \( x \in \{ f \in \text{space} (\text{lborel-f} n). \text{insertion} f (p-fix1 x1) = 0 \} \) using \( x \in \text{space} (\text{lborel-f} n) \) by blast
        next
    qed
  qed

 qed
fix \( f \) assume \( f \in \{ f \in \text{space (lborel-f n)}. \text{insertion} f (p-fix1 x1) = 0 \} \)
then have \( f \in \text{space (lborel-f n)} \) insertion \( f (p-fix1 x1) = 0 \) by auto
have insertion \( (f(n := x1)) p = 0 \) using insertion \( f (p-fix1 x1) = 0 \) \[\text{[unfolded p-fix1-def]}\]
insertion-replace-coeff insertion-irrelevant-vars replace-coeff-extract-var-cong
by (metis (no-types, lifting) insertion \( f (p-fix1 x1) = 0 \) \( \text{vars} (p-fix1 x1) \)
\[\subset \{..<n}\}, \text{fun-upd-other fun-upd-same lessThan-iff order-less-irrefl p-fix1-def reduce-nested-mpoly-extract-var subsetCE}\]
then have \( f(n := x1) \in A \) unfolding A-def using space-lborel-add-dim
using \( f \in \text{space (lborel-f n)} \) lborel-f-def mem-Collect-eq by blast
then show \( f \in \{ f \in \text{space (lborel-f n)}. f(n := x1) \in A \} \) using \( f \in \text{space (lborel-f n)} \)
\[\text{by auto}\]
Qed

have \( \bigwedge x1. x1 \in N \implies \{ x \in \text{space (lborel-f n)}. x(n := x1) \in A \} \in \text{sets (lborel-f n)} \)
and emeasure-in-N: \( \bigwedge x1. x1 \in N \implies \text{emeasure (lborel-f n)} \{ x \in \text{space (lborel-f n)} x(n := x1) \in A \} = \text{emeasure (lborel-f n)} \{ \text{space (lborel-f n)} \} \)
\[\text{by simp}\]
show \( \{ x \in \text{space (lborel-f n)}. x(n := x1) \in A \} \in \text{sets (lborel-f n)} \) \[\text{unfolding set-eq}\]
by (simp add: \( \{ f \in \text{space (lborel-f n)}. \text{insertion} f (p-fix1 x1) = 0 \} = \text{space (lborel-f n)} \)))
show emeasure (lborel-f n) \( \{ x \in \text{space (lborel-f n)} x(n := x1) \in A \} = \text{emeasure (lborel-f n)} \{ \text{space (lborel-f n)} \} \)
unfolding set-eq
by (simp add: \( \{ f \in \text{space (lborel-f n)}. \text{insertion} f (p-fix1 x1) = 0 \} = \text{space (lborel-f n)} \)))
Qed

have emeasure-not-in-N: \( \bigwedge x1. x1 \notin N \implies \text{emeasure (lborel-f n)} \{ x \in \text{space (lborel-f n)}. x(n := x1) \in A \} = 0 \)
and \( \bigwedge x1. x1 \notin N \implies \{ x \in \text{space (lborel-f n)}. x(n := x1) \in A \} \in \text{sets (lborel-f n)} \)
\[\text{by simp}\]
fix \( x1 \) assume \( x1 \notin N \)
then have \( p-fix1 x1 \neq 0 \) using p-fix1-def N-def by auto
then have emeasure (lborel-f n) \( \{ f \in \text{space (lborel-f n)}. \text{insertion} f (p-fix1 x1) = 0 \} = 0 \) \[\text{using Suc.IH[OF p-fix1 x1 \( \neq 0 \)] \( \bigwedge x1. \text{vars} (p-fix1 x1) \subseteq \{..<n\} \) by auto}\]
then show emeasure (lborel-f n) \( \{ x \in \text{space (lborel-f n)}. x(n := x1) \in A \} = 0 \)
\( \{ x \in \text{space (lborel-f n)}. x(n := x1) \in A \} \in \text{sets (lborel-f n)} \)
have \( N \in \text{null-sets borel} \) using \( \langle \text{finite } N \rangle \) finite-imp-null-set-lborel by blast
have ae-zero: \( AE \ x1 \) in borel. emeasure (lborel-f n) \( \{ x \in \text{space (lborel-f n)} . \ x(n := x1) \} \in A \) = 0, using set-eq by auto qed

have measurable: \( (\lambda x1. \ \text{emeasure (lborel-f n)} \ \{ x \in \text{space (lborel-f n)} . \ x(n := x1) \}) \in \text{borel-measurable borel} \)
proof (rule borel-measurableI)
let \( \forall f = (\lambda x1. \ \text{emeasure (lborel-f n)} \ \{ x \in \text{space (lborel-f n)} . \ x(n := x1) \in A \}) \)
fix \( S : \text{ennreal set} \) assume open S
have \( \{ x \in S . \ x \neq S \implies - S \subseteq \exists f \neq ' S \)
using emeasure-not-in-N by auto
have \( 1 : \text{emeasure (lborel-f n)} (\text{space (lborel-f n)}) \in S \implies N \subseteq \exists f \neq ' S \)
using emeasure-in-N by auto
have \( 2 : 0 \notin S \implies \exists f \neq ' S \subseteq N \) using emeasure-not-in-N by fastforce
have \( 3 : \text{emeasure (lborel-f n)} (\text{space (lborel-f n)}) \notin S \implies \exists f \neq ' S \subseteq - N \) using emeasure-in-N by auto
have \( \{ x \in S . \ x \neq S \implies - N \}

\( \vee \exists f \neq ' S = UNIV \vee \exists f \neq ' S = - N \)
apply (cases 0 \in S; cases \text{emeasure (lborel-f n)} (\text{space (lborel-f n)}) \notin S)
using 0 1 2 3 by auto
then show \( \exists f \neq ' S \cap \text{space borel} \in \text{sets borel} \)
using \( \langle \text{finite } N \rangle \) finite-imp-null-set-lborel comp null-setsD2 sets-lborel by fastforce

have \( A \in \text{sets (lborel-f (Suc n))} \) unfolding A-def
using pred-eq-const1[OF measurable-insertion OF \( \langle \text{vars } p \subseteq \{.<\text{Suc n}\} \rangle \)]

pred-def by force
then have in-sets: \( \{ f \in \text{space (lborel-f (Suc n))}. \ \text{insertion } f p = 0 \} \in \text{sets (lborel-f (Suc n))} \) using A-def by metis
have \( \{ x \in \text{space (lborel-f n)} . \ x(n := x1) \in A \} \in \text{sets (lborel-f n)} \)
using \( \{ x \in \text{space (lborel-f n)} . \ x(n := x1) \in A \} \in \text{sets (lborel-f n)} \)
by auto
have emeasure (lborel-f (Suc n)) \( A = \int y. \ \text{emeasure (lborel-f n)} \ \{ x \in \text{space (lborel-f n)} . \ x(n := y) \in A \} \) borel
using emeasure-lborel-f-Suc[OF \( A \in \text{sets (lborel-f (Suc n))} \)]
\( \{ x \in \text{space (lborel-f n)} . \ x(n := x1) \in A \} \in \text{sets (lborel-f n)} \) by blast
also have \( ... = 0 \)

\[ \text{using nn-integral-0-iff-AE[0F measurable] ac-zero by blast} \]

finally have \( \text{emeasure (lborel-f (Suc n)) A = 0 by auto} \)

then show \( \text{?case unfolding null-sets-def using in-sets A-def by blast} \)

qed

end

18 Shallow Network Model

theory DL-Shallow-Model
imports DL-Network Tensor-Rank
begin

fun shallow-model' where
shallow-model' \( Z \ M \ 0 \) = Conv \((Z, M) (\text{Input} M)\) |
shallow-model' \( Z \ M \ (\text{Suc} \ N) \) = Pool \((\text{shallow-model'} \ Z \ M \ 0) (\text{shallow-model'} \ Z \ M \ N)\)

definition shallow-model where
shallow-model \( Y \ Z \ M \ N \) = Conv \((Y, Z) \ (\text{shallow-model'} \ Z \ M \ N)\)

lemma valid-shallow-model': valid-net \((\text{shallow-model'} \ Z \ M \ N)\)
  apply \((\text{induction} \ N)\) unfolding shallow-model'..simps
  by \((\text{simp add: valid-net.intros,metis shallow-model'.elims shallow-model'..simps(1)} \) valid-net.intros output-size.simps)  

lemma output-size-shallow-model': output-size \((\text{shallow-model'} \ Z \ M \ N)\) = \( Z \)
  apply \((\text{induction} \ N)\) unfolding shallow-model'..simps using output-size.simps
  by simp-all

lemma valid-shallow-model: valid-net \((\text{shallow-model} \ Y \ Z \ M \ N)\)
  unfolding shallow-model-def using valid-net.intros output-size.simps output-size-shallow-model' by metis

lemma output-size-shallow-model: output-size \((\text{shallow-model} \ Y \ Z \ M \ N)\) = \( Y \)
  unfolding shallow-model-def using output-size-shallow-model' output-size.simps
  by simp

lemma input-sizes-shallow-model: input-sizes \((\text{shallow-model} \ Y \ Z \ M \ N)\) = replicate \((\text{Suc} \ N) \ M\)
  apply \((\text{induction} \ N)\) unfolding shallow-model-def input-sizes.simps by simp-all

lemma balanced-net-shallow-model': balanced-net \((\text{shallow-model'} \ Z \ M \ N)\)
proof \((\text{induction} \ N)\)
  case 0
  then show \( \text{?case} \)
    by \((\text{metis balanced-net.simps shallow-model'..simps(1)}\))

next
case (Suc N)
  have count-weights True (Conv (Z, M) (Input M)) = count-weights True (shallow-model' Z M N)
    by (induction N; simp)
  then show ?case unfolding shallow-model'.simps
    by (simp add: Suc.IH balanced-net-Conv balanced-net-Input balanced-net-Pool)
qed

lemma balanced-net-shallow-model:
  unfolding shallow-model-def
  by (simp add: balanced-net-Conv balanced-net-shallow-model' simps insert-weights simps input-sizes simps by metis)

lemma cprank-max1-shallow-model':
  assumes y < output-size (shallow-model' Z M N)
  shows cprank-max1 (tensors-from-net (insert-weights s (shallow-model' Z M N) w) $ y)
    using assms proof (induction N arbitrary: w)
      case 0
        then have input-sizes (insert-weights s (shallow-model' Z M 0) w) = [M]
          unfolding shallow-model-def shallow-model'.simps insert-weights.simps input-sizes.simps by metis
        then have dims (tensors-from-net (insert-weights s (shallow-model' Z M 0) w) $ y) = [M]
          using dims-tensors-from-net[OF vec-setI 0.prems(1) output-size-correct-tensors remove-insert-weights valid-shallow-model' by metis]
        then show ?case
          using order1 by (metis One-nat-def eq-imp-le length-Cons list.size(3))
    next
      case (Suc N)
        have g-le-IH: y < dim-vec (tensors-from-net (insert-weights s (shallow-model' Z M N) (λi. w (i + (count-weights s (shallow-model' Z M 0))))))
          using output-size-correct-tensors[of insert-weights s (shallow-model' Z M N)]
          (λi. w (i + (count-weights s (shallow-model' Z M 0)))))
          unfolded remove-insert-weights, OF valid-shallow-model'
        using Suc.prems(1) output-size-shallow-model' by auto
        have cprank-max1-IH:cprank-max1 (tensors-from-net (insert-weights s (shallow-model' Z M N) (λi. w (i + (count-weights s (shallow-model' Z M 0)))))) $ y)
          using Suc.IH Suc.prems(1) output-size-shallow-model' by auto
        have y-le-0:y < dim-vec (tensors-from-net (insert-weights s (shallow-model' Z M 0) w) $ y)
          by (metis assms output-size-correct-tensors output-size-shallow-model' remove-insert-weights valid-shallow-model')
        have cprank-max1-0:cprank-max1 (tensors-from-net (insert-weights s (shallow-model' Z M 0) w) $ y)
          proof
            have input-sizes (insert-weights s (shallow-model' Z M 0) w) = [M]
              unfolding shallow-model-def shallow-model'.simps insert-weights.simps input-sizes.simps by metis
            then show ?thesis using order1 dims-tensors-from-net[OF vec-setI] One-nat-def

103
then show \( \text{case unfolding shallow-model'.simp{2}} \) insert-weights.simps tensors-from-net.simps
  using cprank-max1-IH cprank-max1-0 cprank-max1-prod index-component-mult
  y-le-0 y-le-IH
  by (metis Suc.IH output-size-correct-tensors remove-insert-weights valid-shallow-model')
qed

lemma cprank-shallow-model:
assumes \( m = \text{insert-weights } s \text{ (shallow-model } Y Z M N) \ w \)
assumes \( y < Y \)
shows \( \text{cprank (tensors-from-net } m \ y) \leq Z \)
proof –
  have \( s \Longrightarrow \text{shared-weight-net } m \)
    by (simp add: assms(1) balanced-net-shallow-model shared-weight-net-insert-weights)
  have \( \text{cprank-max } Z \text{ (tensors-from-net } m \ y) \)
    proof –
    have \( \text{dim-extract: dim-row (extract-matrix } w Y Z) = Y \)
      using dim-extract-matrix(1) by force
    have \( \text{dimc-extract-matrix: dim-col (extract-matrix } w Y Z) = Z \)
      using dim-extract-matrix(1) by force
    have \( \text{input-sizes: (input-sizes } \text{insert-weights } s \text{ (shallow-model'} Z M N) \text{ (} \lambda i. w (i + Y * Z))))) = \text{input-sizes (shallow-model'} Z M N) \)
      using input-sizes-remove-weights remove-insert-weights by auto
    have \( 0 : \text{tensors-from-net } m \ y = \text{Tensor-Plus} \cdot \text{listsum (input-sizes (shallow-model'} Z M N))} \)
    (map (\( \lambda j. \text{extract-matrix } w Y Z) $$ (y, j) \cdot \text{tensors-from-net (insert-weights } s \text{ (shallow-model'} Z M N) \text{ (} \lambda i. w (i + Y * Z))))) $$ (j) [0..<Z]
    unfolding \( \text{map = insert-weights } s \text{ (shallow-model } Y Z M N) \text{ w} \text{ shallow-model-def insert-weights.simps tensors-from-net.simps} \)
      using nth-nat-tensorlist-mult dims-tensors-from-net assms(2) dim-extract
    output-size-correct-tensors[of insert-weights s (shallow-model' Z M N) \text{ (} \lambda i. w (i + Y * Z), unfolded remove-insert-weights, OF valid-shallow-model']
    dimc-extract-matrix output-size-shallow-model' input-sizes by auto
    define Bs where \( Bs = \text{map (} \lambda j. \text{extract-matrix } w Y Z $$ (y, j) \cdot \text{tensors-from-net (insert-weights } s \text{ (shallow-model'} Z M N) \text{ (} \lambda i. w (i + Y * Z))))) $$ (j) [0..<Z] \)
    have \( \forall B. B \in \text{set } Bs \Longrightarrow \text{cprank-max}1 B \forall B. B \in \text{set } Bs \Longrightarrow \text{dims } B = \text{input-sizes (shallow-model'} Z M N) \)
    proof –
    fix B assume \( B \in \text{set } Bs \)
    then obtain j where \( B = Bs ! j \ j < \text{length } Bs \) by (metis in-set-conv-nth)
    then have \( j < Z \) using length-map Bs-def by simp
    have \( 1:\text{cprank-max}1 \text{ (tensors-from-net (insert-weights } s \text{ (shallow-model'} Z M N) \text{ (} \lambda i. w (i + Y * Z))))) $$ j)
using \( j < Z \) output-size-shallow-model' cprank-max1-shallow-model' by auto
then have cprank-max1 (extract-matrix w Y Z \( \langle y, j \rangle \cdot \) tensors-from-net
(insert-weights s (shallow-model' Z M N) (\( \lambda \), w (\( i + Y \cdot Z \))) \( \langle y, j \rangle \) )
using smult-prod-extract1 cprank-max1-order0 \( \langle OF \cdot 1, \) of extract-matrix w Y
Z \( \langle y, j \rangle \cdot 1 \) ] by (metis dims-smult mul.left-neutral order-tensor-one)
then show cprank-max1 B by (simp add: Bs-def \( \langle B = Bs \rangle \cdot j < Z \) )
show dims B = input-sizes (shallow-model' Z M N) unfolding \( \langle B = Bs \rangle \cdot j \)
Bs-def
nth-map \( \langle j \cdot 0..<Z \rangle \), unfolded length-apt Nat.diff-0, OF \( \langle j < Z \rangle \) )
dims-smult
input-sizes\( \langle symmetric \rangle \)
by (rule dims-tensors-from-net; rule vec-setI \[ \langle \) where \( i=j \) \]\), simp add: \( \langle j < Z \rangle \),
mantis (no-types) \( \langle j < Z \rangle \) output-size-correct-tensors output-size-shallow-model'
remove-insert-weights valid-shallow-model')
qed
then show \(?thesis unfolding \( \langle 0 \) using cprank-max1 length-map Bs-def by
(metis (no-types, lifting) diff-zero length-apt)
qed
then show \(?thesis unfolding cprank-def by (simp add: Least-le)
qed

end

19 Fundamental Theorem of Network Capacity

theory DL-Fundamental-Theorem-Network-Capacity
imports DL-Rank-CP-Rank DL-Deep-Model-Poly Lebesgue-Zero-Set
Jordan-Normal-Form.DL-Rank-Submatrix HOL-Analysis.Complete-Measure DL-Shallow-Model
begin

context deep-model-correct-params-y
begin

definition polynomial-f w = det (submatrix (matricize \{ n. even n \} (A w)) rows-with-1
rows-with-1)

lemma polyfun-polynomial:
shows polyfun \( \langle ..< \) weight-space-dim \( \rangle \) polynomial-f
unfolding polynomial-f-def using polyfun-det-deep-model unfolding witness-submatrix-def
A'-def .

definition polynomial-p = (SOME p. vars p \( \subseteq \langle ..< \) weight-space-dim \( \rangle \) \& (\( \forall x. \) insertion x p = polynomial-f x))

lemma polynomial-p-not-0: polynomial-p \( \not = 0 \) and

105
vars-polynomial-p; vars polynomial-p ⊆ \{..<weight-space-dim\} and
polynomial-pf: \( \forall w. \text{insertion } w \text{ polynomial-p } = \text{polynomial-f } w \)
proof
- have vars polynomial-p ⊆ \{..<weight-space-dim\} \( \land \forall x. \text{insertion } x \text{ polynomial-p } = \text{polynomial-f } x \)
  using someI-ex[OF polyfun-polynomial[unfolded polyfun-def]] .
then show vars polynomial-p ⊆ \{..<weight-space-dim\} \( \land w. \text{insertion } w \text{ polynomial-p } = \text{polynomial-f } w \)
  by auto
show polynomial-p ≠ 0 using A′-def Aw′-def ′ \( \land w. \text{insertion } w \text{ polynomial-p } = \text{polynomial-f } w \)
  polynomial-f-def witness-det by auto
qed

lemma if-polynomial-0-rank:
assumes polynomial-f w ≠ 0
shows \( r \wedge N\text{-half} \leq \text{cprank } (A w) \)
proof
- have \( r \wedge N\text{-half} = \text{dim-row } (\text{submatrix } (\text{matricize } \{n. \text{ even } n\} (A w)) \text{ rows-with-1 rows-with-1}) \)
  by (metis (full-types) Aw′-def card-rows-with-1 dim-submatrix(1) dims-A dims-Aw dims-matricize(1) set-le-in)
also have \( ... \leq \text{rank } (\text{matricize } \{n. \text{ even } n\} (A w)) \)
  using assms vec-space.rank-gt-minor[OF carrier-matI[OF dims-A′-pow, unfolded weight-space-dim-def]]
  by (metis (full-types) A′-def dim-submatrix(1) dims-A′-pow(1) polynomial-f-def)
also have \( ... \leq \text{cprank } (A w) \) using matrix-rank-le-cp-rank by blast
finally show \( \text{thesis} \).
qed

lemma if-polynomial-0-evaluate:
assumes polynomial-f wd ≠ 0
assumes \( \forall \text{ inputs. } \text{input-sizes } (\text{deep-model-l } rs) = \text{map dim-vec } \text{inputs} \rightarrow \text{evaluate-net } (\text{insert-weights shared-weights } (\text{deep-model-l } rs) \text{ wd}) \text{ inputs} = \text{evaluate-net } (\text{insert-weights shared-weights } (\text{shallow-model } (rs ! 0) Z (last rs) (2*N-half − 1)) \text{ ws}) \text{ inputs} \)
shows \( Z \geq r \wedge N\text{-half} \)
proof
- have valid1:valid-net′ (insert-weights shared-weights (deep-model-l rs) wd)
  using remove-insert-weights valid-deep-model by presburger
have valid2:valid-net′ (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half − 1)) us)
  by (simp add: remove-insert-weights valid-shallow-model)
have input-sizes: input-sizes (insert-weights shared-weights (deep-model-l rs) wd)
  = input-sizes (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2 * N-half − 1)) us)
  using input-sizes-remove-weights input-sizes-deep-model remove-insert-weights
    by (simp add: N-half-def input-sizes-shallow-model)
have 0:tensors-from-net (insert-weights shared-weights (deep-model-l rs) wd)
  = tensors-from-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half − 1)) us)
using [tensors-from-net-eqI [OF valid1 valid2 input-sizes, unfolded input-sizes-remove-weights remove-insert-weights]
using assms by blast
have cprank (tensors-from-net (insert-weights shared-weights (deep-model-l rs)) wd) $ y \leq Z
unfolding 0 using y-valid cprank-shallow-model by blast
then show \?thesis
using if-polynomial-0-rank assms
using A-def assms(1) less-le-trans not-le remove-insert-weights
by fastforce
qed

lemma if-polynomial-0-evaluate-notex:
assumes polynomial-f wd \neq 0
shows \neg(\exists \text{weights-shallow } Z. \ Z < r ^ N-half \land (\forall \text{ inputs. input-sizes (deep-model-l rs)) = map dim-vec inputs \longrightarrow evaluate-net (insert-weights shared-weights (deep-model-l rs) wd) inputs
= evaluate-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half-1)) ws) inputs))
using assms if-polynomial-0-evaluate not-le by blast

theorem fundamental-theorem-network-capacity:
AE x in lebesgue-f weight-space-dim. \ r ^ N-half \leq cprank (A x)
using AE-I [OF lebesgue-mpoly-zero-set [OF polynomial-p-not-0 vars-polynomial-p]]
by (metis (mono-tags, lifting) Collect-mono if-polynomial-0-rank polynomial-pf)

theorem fundamental-theorem-network-capacity-v2:
shows AE wd in lebesgue-f weight-space-dim.
\neg(\exists ws Z. \ Z < r ^ N-half \land (\forall \text{ inputs. input-sizes (deep-model-l rs)) = map dim-vec inputs \longrightarrow evaluate-net (insert-weights shared-weights (deep-model-l rs) wd) inputs
= evaluate-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half-1)) ws) inputs))
apply (rule AE-I [OF lebesgue-mpoly-zero-set [OF polynomial-p-not-0 vars-polynomial-p], unfolded polynomial-pf])
apply (rule subsetI) unfolding mem-Collect-eq
using if-polynomial-0-evaluate-notex by metis

abbreviation lebesgue-f where lebesgue-f n \equiv completion (lborel-f n)

lemma space-lebesgue-f: space (lebesgue-f n) = PiE {..<n} (\lambda. UNIV)
by (simp add: space-lborel-f)

theorem fundamental-theorem-network-capacity-v3:
assumes S = {wd \in space (lebesgue-f weight-space-dim).
\exists ws Z. \ Z < r ^ N-half \land (\forall \text{ inputs. input-sizes (deep-model-l rs)) = map dim-vec inputs \longrightarrow evaluate-net (insert-weights shared-weights (deep-model-l rs) wd) inputs

107
evaluate-net (insert-weights shared-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half-1)) ws) inputs)
  shows S ∈ null-sets (completion (borel-f weight-space-dim))
  unfolding assms
  using fundamental-theorem-network-capacity-v2 [unfolded completion. AE-iff-null-sets [unfolded AE-completion-iff], unfolded not-not]
  by blast

end
end

References
