

# OCANNL

Mysteries of NN training unveiled

# OCANNL

My work on OCANNL is sponsored by



# OCaml Compiles Algorithms for Neural Network Learning

- OCANNL is distributed as two opam packages:
  - `arrayjit`: a backend for compilers for numerical array programming,
  - `neural_nets_lib`: a neural networks (Deep Learning) framework.
- You express numerical computations at runtime, OCANNL will optimize them for the runtime-dependent shape of your data, compile and dynamically load them using one of its backends.

# OCaml Compiles Algorithms for Neural Network Learning

- OCANNL is distributed as two opam packages:
  - `arrayjit`: a backend for compilers for numerical array programming languages,
  - `neural_nets_lib`: a neural networks (Deep Learning) framework.
- You express numerical computations at runtime, OCANNL will compile them.
- There are startups doing it in other languages, so it must be worth it!
  - In Python: [tinygrad](#) – democratizing DL – at home or on premise training of large models.
  - In Rust: [Luminal](#) – simplifies deployment of large models with on-device inference.
- There are many optimization / NN frameworks in Rust, but few in OCaml!

[Luminal](#)

[Candle](#)

[Cubecl](#)

[r-nn](#)

[Burn](#)

[Autograph](#) (with [krnl](#))

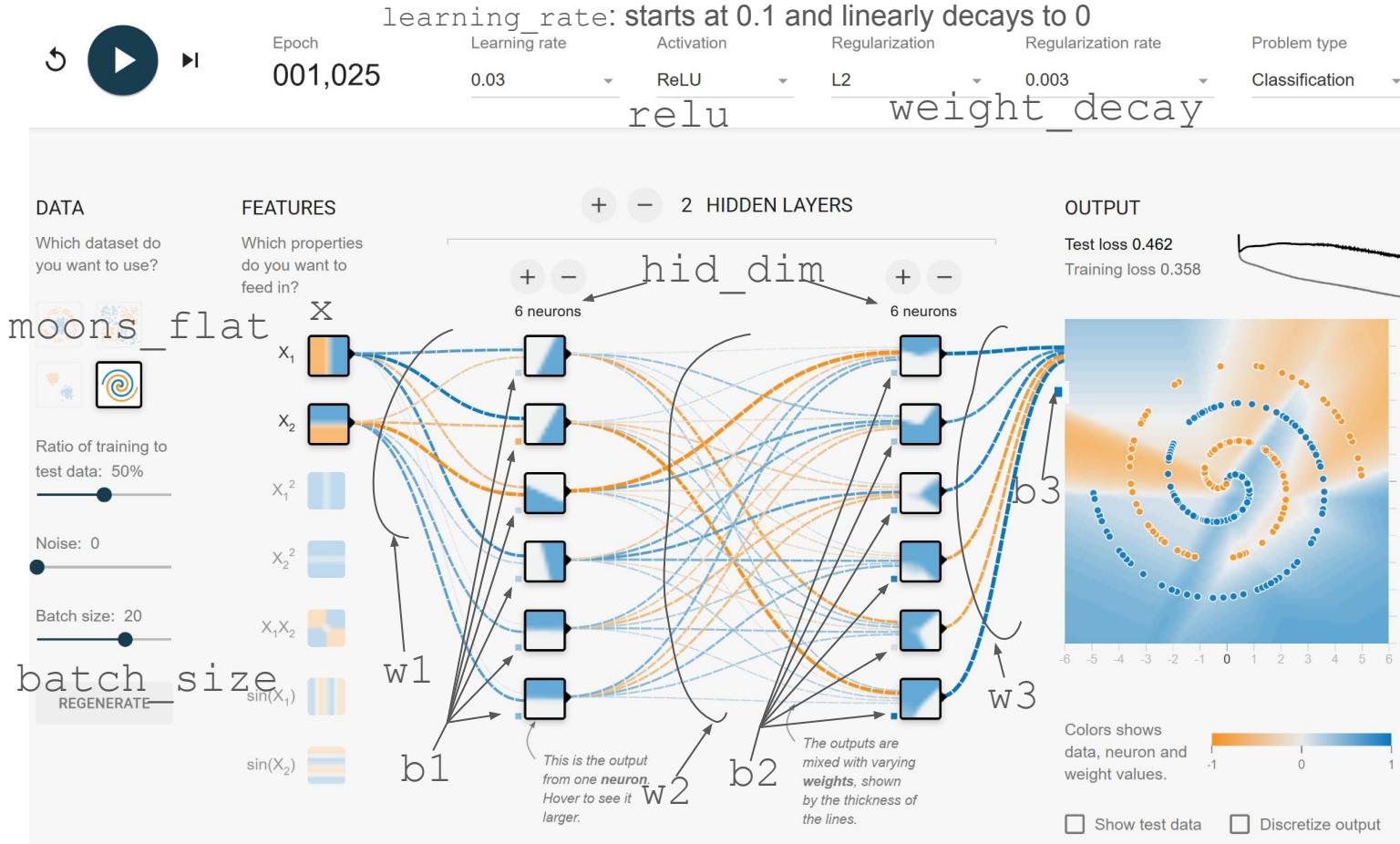
# OCaml Compiles Algorithms for Neural Network Learning

- OCANNL is distributed as two opam packages:
  - `arrayjit`: a backend for compilers for numerical array programming languages,
  - `neural_nets_lib`: a neural networks (Deep Learning) framework.
- You express numerical computations at runtime, OCANNL will compile them.
- There are startups doing it in other languages, so it must be worth it!
  - In Python: [tinygrad](#) – democratizing DL – at home or on premise training of large models.
  - In Rust: [Luminal](#) – simplifies deployment of large models with on-device inference.
- Value added:
  - OCaml is a good fit for writing optimizing compilers.
  - OCANNL has concise notation thanks to better shape inference (i.e. type inference for the data and transformation matrices).
  - OCANNL might be a good fit for heterogeneous computing (e.g. combining GPUs from different companies), it is explicit about the backends (and devices) used.

**OCANNL** is still at a  
proof-of-concept stage,  
but it will grow and evolve.

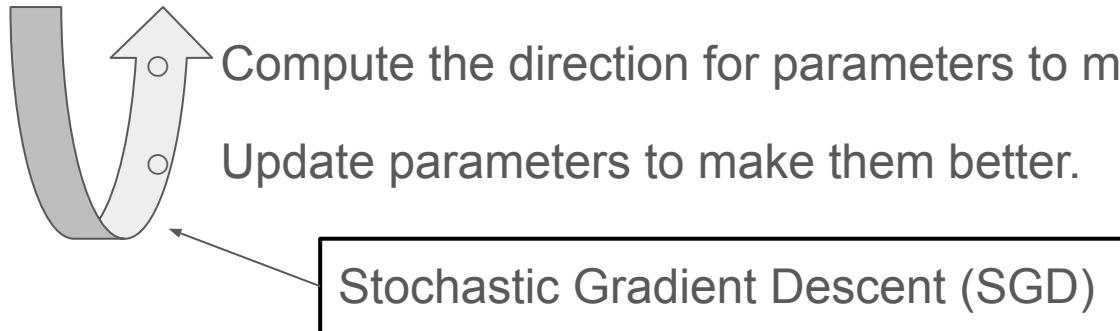
Let's train a feed-forward neural network with 2 hidden layers (aka. a 3-layer MLP) to classify points on a plane.

Try Tensorflow Playground for yourself. Compare with bin/moons\_demo.ml:



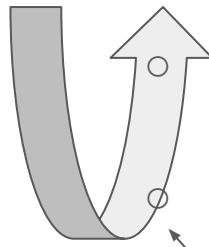
# Training NNs is first-order (i.e. gradient based) optimization

- Collect examples to learn from. ← **a dataset**
- Express a solution as a parameterized differentiable computation. ← **a model**
- Figure out a formula for how bad the solution is on a datapoint. ← **a loss function**
- For each datapoint in the dataset:



# Training NNs is first-order (i.e. gradient based) optimization

- Collect examples to learn from. ← **a dataset**
- Express a solution as a parameterized differentiable computation. ← **a model**
- Figure out a formula for how bad the solution is on a datapoint. ← **a loss function**
- For each (mini-)batch of data points in the dataset:



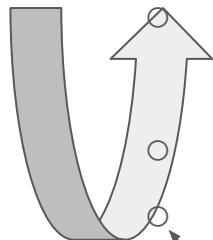
For each datapoint in the batch, compute the direction for parameters to make them better/worse → datapoint gradient.  
Add up the batch datapoint gradients.  
Update parameters to make them better.

**Stochastic Gradient Descent (SGD)**



# Training NNs is first-order (i.e. gradient based) optimization

- For each (mini-)batch of data points in the dataset:



For each datapoint in the batch, compute the direction for parameters to make them better/worse → datapoint gradient.

Add up the batch datapoint gradients.

Update parameters to make them better.

Stochastic Gradient Descent (SGD)

(Mini-)Batches are equal-sized subsets of the dataset. Main options for defining a batch:

- An element of a (fixed but random) partition of the dataset. We'll use this.
- A random subset of the dataset.

# Half-moons toy dataset

```
let noise () = Rand.float_range (-0.1) 0.1 in
let moons_flat =
  Array.concat_map (Array.create ~len ())
    ~f:
      Float.(
        fun () ->
          let i = Rand.int len in
          let v = of_int i * pi / of_int len in
          let c = cos v and s = sin v in
          [| c + noise (); s + noise (); 1.0 - c + noise (); 0.5 - s + noise () |])
in
let moons_flat = TDSL.init_const ~l:"moons_flat" ~o:[ 2 ] moons_flat in
let moons_classes = Array.init (len * 2) ~f:(fun i -> if i % 2 = 0 then 1. else -1.) in
let moons_classes = TDSL.init_const ~l:"moons_classes" ~o:[ 1 ] moons_classes in
```



output axis dimension  
(batch axes are inferred)



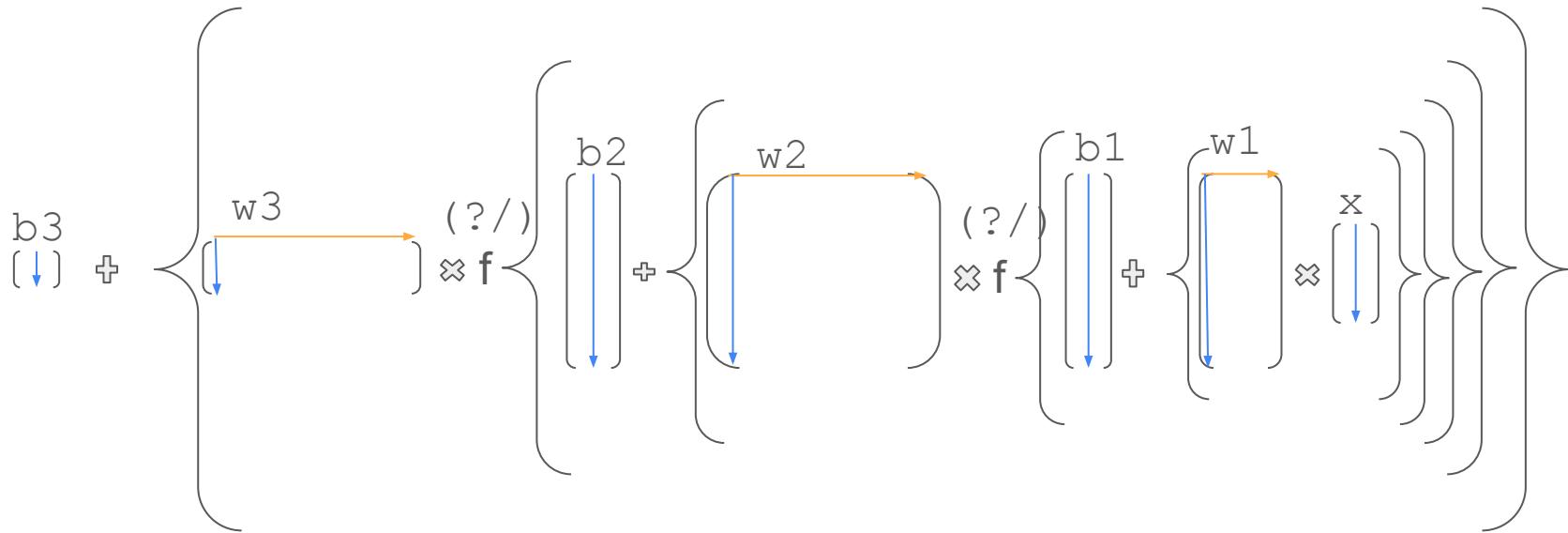
# Half-moons toy dataset

```
let points = Tensor.value_2d_points ~xdim:0 ~ydim:1 moons_flat in
let classes = Tensor.value_1d_points ~xdim:0 moons_classes in
let points1, points2 = Array.partitioni_tf points ~f:Float.(fun i _ -> classes.(i) > 0.) in
let plot_moons =
  let open PrintBox_utils in
  plot ~size:(120, 40) ~x_label:"ixes" ~y_label:"ygreks"
  [
    Scatterplot { points = points1; pixel = "#" }; Scatterplot { points = points2; pixel = "+" };
  ]
in
Stdio.printf "\nHalf-moons scatterplot:\n%!";
PrintBox_text.output Stdio.stdout plot_moons;
```

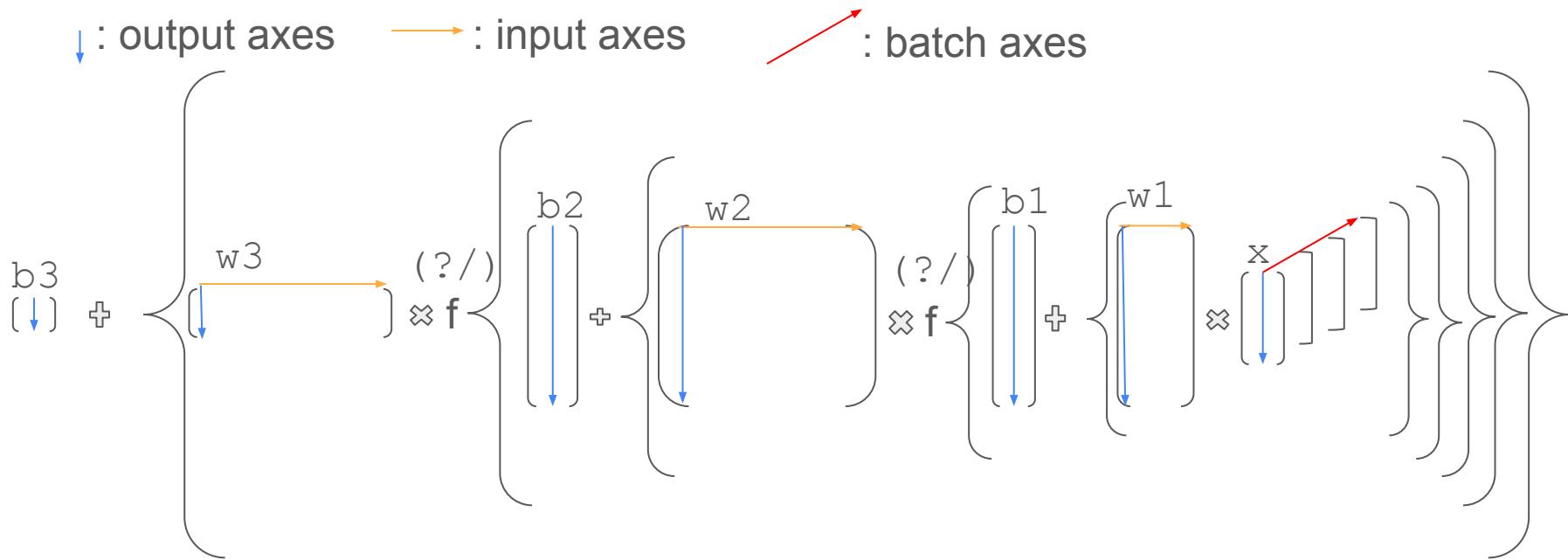


# Tensors as “differentiable” multidimensional matrices

↓ : output axes      → : input axes

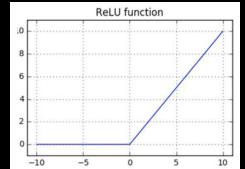


# Tensors as “differentiable” multidimensional matrices



# Multi Layer Perceptron in one line of code

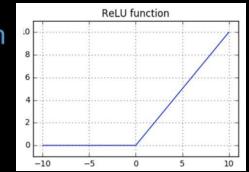
```
relu =
```



```
let%op mlp x =
  "b3" + ("w3" * relu ("b2" hid_dim + ("w2" * relu ("b1" hid_dim + ("w1" * x))))) in
```

# Multi Layer Perceptron in one line of code

```
let%op mlp x = "b3" + ("w3" * relu ("b2" hid_dim + ("w2" * relu ("b1" hid_dim + ("w1" * x)))))) in
```



Tensor function, expands to:

```
let w1 = ... in let b1 = ... in ... let mlp x = ... in ...
```

Declarative expressions for differentiable tensor operations

Introduces identifier `w1` for a parameter tensor

Sets the output dimension of `b1` to `hid_dim`

“Rectified Linear Unit” unary operation, see graph on top-right

Tensor (e.g. matrix) multiplication

`relu =`

# Hinge loss function: maximum margin classification

$$l(y, f(x)) = \max(0, 1 - y * f(x))$$

pointwise multiplication

```
let%op margin_loss = relu (1 - (moons_class *. mlp moons_input)) in  
let%op scalar_loss = (margin_loss ++ "...|... => 0") /. !..batch_size in
```

sum out all batch and output axes  
to dimension 0 of the result

embed an int

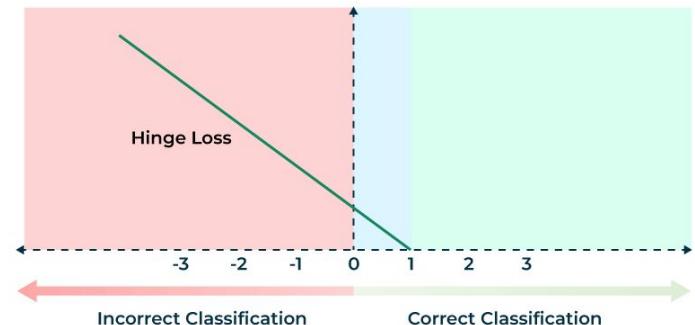
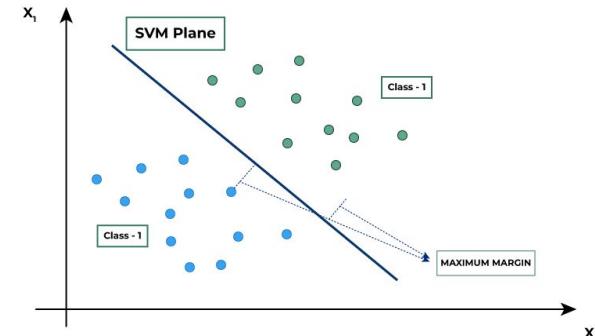


Image source:

[Hinge-loss & relationship with Support Vector Machines - GeeksforGeeks](#)



# Regularization – weight decay

- **Regularization:** keep models simple to be less accidentally wrong and to stabilize training.
- Can add a **regularizer** to the loss function.
- Or can modify the SGD update step – we'll do this.
- **Weight decay** is L2 regularization because the gradient of  $\rho^2$  is  $2\rho$ .

Image source:

[Stay away from overfitting:](#)  
[L2-norm Regularization,](#)  
[Weight Decay and L1-norm](#)  
[Regularization techniques | by](#)  
[Inara Koppert-Anisimova |](#)  
[unpack | Medium](#)

L1 Regularization	L2 Regularization
1. L1 penalizes sum of absolute values of weights.	1. L2 penalizes sum of square values of weights.
2. L1 generates model that is simple and interpretable.	2. L2 regularization is able to learn complex data patterns.
3. L1 is robust to outliers.	3. L2 is not robust to outliers.

# Backprop: compositionally deriving gradient computations

- **Backprop** is a special case of **reverse mode automatic differentiation**, that's limited to first-order (cannot compute e.g. Hessians).
- Generalizes the chain rule:  $df/dx = df/dy * dy/dx$ .
- **Forward pass**: computing the value of a tensor from the values of tensors in its definition.
- **Backward pass**: computing the gradient of the value of interest  $f$  (e.g. loss) with respect to a tensor  $x$ :  $df/dx = x.grad$ , by adding up contribution from each place in which  $x$  appears.

# Backprop: compositionally deriving gradient computations

- **Forward pass:** computing the `value` of a tensor from the values of tensors in its definition.
- **Backward pass:** computing the gradient of the value of interest  $f$  (e.g. loss) with respect to a tensor  $x$ :  $df/dx = x.grad$ , by adding up contribution from each place in which  $x$  appears.
  - Once we have  $df/dy$ , we use  $dy/dx$  and proceed backward to compute  $df/dx$ .
  - The order of computation is reversed:  $x \rightarrow y(x) \rightarrow f(y(x))$ , but  $df \rightarrow df/dy \rightarrow df/dx$ .
  - But the order of composition is the same – bottom-up: we just need to **prepend** the  $df/dy$ -specific code to the backward code of  $y$  to build the backward code of  $f$ .



# Backprop: compositionally deriving gradient computations

- **Forward pass:** computing the value of a tensor from the values of tensors in its definition.
- **Backward pass:** computing the gradient of the value of interest  $f$  (e.g. loss) with respect to a tensor  $x$ :  $df/dx = x.grad$ , by adding up contribution from each place in which  $x$  appears.
  - Example:  $f(t(t_1, t_2))$ ,  $t=t_1 * t_2$ . Let  $g=df/dt$  (incoming gradient).
  - $dt/dt_1 = t_2$ , therefore  $df/dt_1 = df/dt * dt/dt_1 = g * t_2$ .
  - $dy/dt_2 = t_1$ , therefore  $df/dt_2 = df/dt * dt/dt_2 = g * t_1$ .
  - At node  $t=t_1 * t_2$ , we back-propagate  $g * t_2$  toward  $t_1$ , and  $g * t_1$  toward  $t_2$ .
  - See [Step-by-step example of reverse-mode automatic differentiation - Cross Validated](#)

# Interlude: what is code / computation?

- A tensor in OCANNL is associated with two tensor nodes: its **value** and (optionally) its **gradient**.
- The numeric code OCANNL works with, at a high level, is mostly: sequences of **accumulating assignments**, with a **unary** or **binary** operation on the right-hand-side.
- The most frequent accumulation operators are
  - “don’t accumulate” (aka. `fun x y -> y`),
  - addition.
- The accumulating assignment can optionally reset the left-hand-side tensor node to the neutral element of the accumulation operator.
- The syntax extension `%cd` is about all that!

# Backprop by example

Set the Left-Hand-Side tensor to...

```
let add ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 + v2 in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 += g;
    g2 += g
  in
  Tensor.binop ~label:( "+" :: label) ~compose_op:Pointwise_bin ~op_asn ~grad_asn
```

Without resetting the LHS,  
add up...



# Backprop by example

Forward pass

Backprop pass

Set the Left-Hand-Side tensor to...

```
let add ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 + v2 in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 += g;
    g2 += g
  in
  Tensor.binop ~label:( "+" :: label) ~compose_op:Pointwise_bin ~op_asn ~grad_asn
```

Without resetting the LHS,  
add up...

Shorthand for `t1.value`

Shorthand for `t2.grad`

# Backprop by example

Forward pass

Backprop pass

Set the Left-Hand-Side tensor to...

```
let add ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 + v2 in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 += g;
    g2 += g
  in
  Tensor.binop ~label:( "+" :: label) ~compose_op:Pointwise_bin ~op_asn ~grad_asn
```

v =: v1 + v2 is a shorthand  
for t.value =: t1.value + t2.value.

Without resetting the LHS,  
add up...

Shorthand for v1.grad

Shorthand for t2.grad



# Backprop by example

Forward pass

Backprop pass

Set the Left-Hand-Side tensor to...

```
let add ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 + v2 in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 += g;
    g2 += g
  in
  Tensor.binop ~label:( "+" :: label) ~compose_op:Pointwise_bin ~op_asn ~grad_asn
```

Without resetting the LHS,  
add up...

Shorthand for  $v1.grad$

Shorthand for  $t2.grad$



# Backprop by example

Forward pass

Backprop pass

Set the Left-Hand-Side tensor to...

```
let sub ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 - v2 in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 += g;
    g2 =- g
  in
  Tensor.binop ~label:(("-" :: label)) ~compose_op:Pointwise_bin ~op_asn ~grad_asn
```

Shorthand for  $t_2.\text{grad}$

Without resetting the LHS,  
deduct up...

Gradient  $t_1.\text{grad}$  increases  
and  $t_2.\text{grad}$  decreases by  
 $t.\text{grad}$  because  $d(t_1 - t_2)/dt_2 = -1$ .

Shorthand for  $v1.\text{grad}$

# Backprop by example

```
let mul compose_op ~op_asn =
  let module NTDSL = Initial_NTDSL in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 =+ g * v2;
    g2 =+ v1 * g
  in
  Tensor.binop ~compose_op ~op_asn ~grad_asn
```

```
let pointmul ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 * v2 in
  mul Pointwise_bin ~op_asn ~label:(".*" :: label)
```

```
let matmul ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =:+ v1 * v2 in
  mul Compose ~op_asn ~label:( "*" :: label)
```

For tensor (e.g. matrix) multiplication  $t = t_1 * t_2$ , the value is  $v = :+ v_1 * v_2$ , which means: first set all cells of  $v$  to zero, then add up  $v_1[\dots] * v_2[\dots]$  for all combinations of those indices used by  $v_1$  and  $v_2$  that are not used by  $v$ .

These specify which indices to use when accessing a tensor (node / array) cell.

First reset  $v$ , then add up the results of...

# Backprop by example

```
let mul compose_op ~op_asn =
  let module NTDSL = Initial_NTDSL in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 =+ g * v2;
    g2 =+ v1 * g
  in
  Tensor.binop ~compose_op ~op_asn ~grad_asn
```

```
let pointmul ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 * v2 in
  mul Pointwise_bin ~op_asn ~label:(".*" :: label)
```

```
let matmul ?(label = []) =
  let module NTDSL = Initial_NTDSL in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =:+ v1 * v2 in
  mul Compose ~op_asn ~label:( "*" :: label)
```

For both pointwise (`let ( *. ) = pointmul ~grad_spec:If_needed`) and tensor multiplication (`let ( * ) = matmul ~grad_spec:If_needed`), gradient propagation follows from the explanation on p. 21: multiply the incoming gradient by the other term.

These specify which indices to use when accessing a tensor (node / array) cell.

First reset  $v$ , then add up the results of...



# Backprop by example

```
?label:string list -> grad_spec:Tensor.grad_spec -> float -> Tensor.t -> Tensor.t
let rec pointpow ?(label : string list = []) ~grad_spec p t1 : Tensor.t =
  let module NTDSL = struct
    include Initial_NTDSL
    lukstafi, 12 months ago | 1 author (lukstafi)
    module O = struct
      include NDO_without_pow
      let ( **. ) ?label base exp = pointpow ?label ~grad_spec:Tensor.Prohibit_grad exp base
      end
    end in
  let p_t = NTDSL.number p in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 ** v2 ~projections in
  let%cd grad_asn =
    if Tensor.is_prohibit_grad grad_spec then fun ~t:_ ~g:_ ~t1:_ ~t2:_ ~projections:_ -> Asgns.Noop
    else if Float.equal p 2.0 then fun ~t:_ ~g ~t1 ~t2:_ ~projections -> g1 += p_t *. t1 * g
    else if Float.equal p 1.0 then fun ~t:_ ~g ~t1 ~t2:_ ~projections -> g1 += g
    else fun ~t:_ ~g ~t1 ~t2:_ ~projections -> g1 += p_t *. (t1 **. (p -. 1.)) * g
  in
  Tensor.binop ~label:(**. :: label) ~compose_op:Pointwise_bin ~op_asn ~grad_asn ~grad_spec t1 p_t
```

This defines pointwise  $t_1$  to-the-power-of  $p$ :

```
let ( **. ) ?label base exp =
  pointpow ?label ~grad_spec:Tensor.Prohibit_grad exp base
```

# Backprop by example

```
?label:string list -> grad_spec:Tensor.grad_spec -> Tensor.t -> Tensor.t
let rec pointdiv ?(label : string list = []) ~grad_spec t1 t2 =
  let module NTDSL = struct
    include Initial_NTDSL
    lukstafi, 13 months ago | 1 author (lukstafi)
    module O = struct
      include NDO_without_div
      let ( /. ) = pointdiv ~grad_spec:Tensor.Prohibit_grad
    end
  end in
  let%cd op_asn ~v ~t1 ~t2 ~projections = v =: v1 / v2 in
  let%cd grad_asn ~t:_ ~g ~t1 ~t2 ~projections =
    g1 += g / v2;
    g2 += g * (-1 *. t1 /. (t2 **. 2))
  in
  Tensor.binop ~label:("/." :: label) ~compose_op:Pointwise_bin ~op_asn ~grad_asn ~grad_spec t1 t2
```

This defines pointwise division:  
 $t1 /. t2$ .

$$\nabla\left(\frac{t_1}{t_2}\right) = \frac{t_2 \nabla t_1 - t_1 \nabla t_2}{t_2^2}$$

# Putting the forward and backward passes together

Tensor.t -> update

```
let grad_update_nochecks loss =
  let params = get_params loss in
  let diff = diff_or_error loss "Train.grad_update_nochecks" in
  let fwd_bprop =
    [%cd
      ~~(loss "gradient update";
          ~~(loss "fwd";
              loss.forward);
          ~~(loss "zero grads";
              diff.zero_grads));
      loss.grad := 1;
      ~~(loss "bprop";
          diff.backprop))]
    in
    { loss; params; fwd_bprop }
```

Actual Train.grad\_update is safer.

Block comment syntax, used for debugging and generating file names.

Zero out gradients before starting to accumulate in the current backward pass.

# Stochastic Gradient Descent with Momentum

- Vanilla SGD subtracts scaled gradients from parameters:

```
p =- learning_rate *. p.grad
```

- This is slow in regions of small loss differences. SGD with momentum speeds training up by accumulating gradients with a form of exponential smoothing.

```
"sgd_momentum" =: (!.momentum *. sgd_momentum) + p.grad;
```

```
p =- learning_rate *. sgd_momentum
```

- It adds up the gradients, with weight for  $n$  steps back =  $momentum^n$ .



# Stochastic Gradient Descent with apx. Nesterov Momentum

- Vanilla SGD and SGD with momentum can lead to oscillations like here:
- To avoid jumping back and forth over the canyon, Nesterov Accelerated Gradient computes the gradient at the “next point”, using just momentum to compute “next point”.
- We approximate that without recomputing the loss gradient by adding the gradient twice but scaled down:

```
"sgd_delta" =: p.grad;  
  
"sgd_momentum" =: (!.momentum *. sgd_momentum) + sgd_delta;  
  
sgd_delta += !.momentum *. sgd_momentum;  
  
p -= learning_rate *. sgd_delta
```

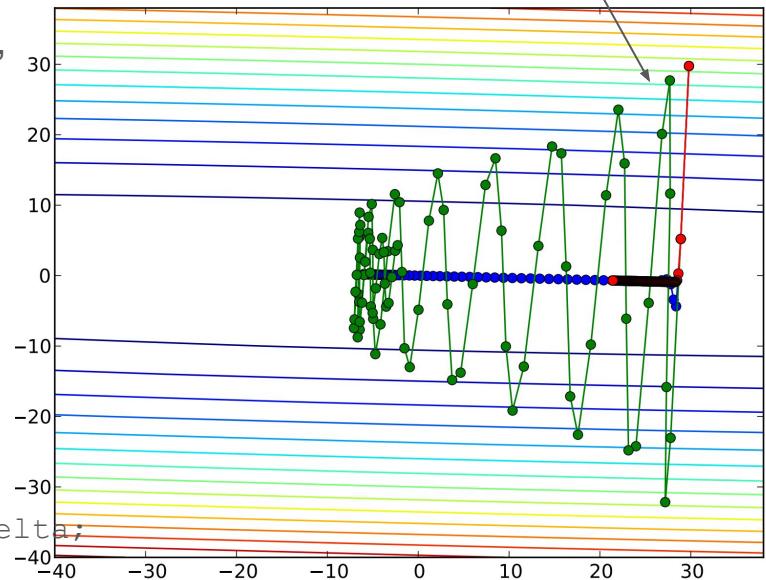


Image from [On the importance of initialization and momentum in deep learning \(mlr.press\)](#)

# Stochastic Gradient Descent with apx. Nesterov Momentum

```
learning_rate:Tensor.t -> ?momentum:float -> ?weight_decay:float -> ?nesterov:bool -> Tensor.t -> Asgns.t
let sgd_one ~learning_rate ?(momentum = 0.0) ?(weight_decay = 0.0) ?(nesterov = false) p =
  if not @@ is_param p then raise @@ Tensor.Session_error ("Train.sgd_one: not a parameter", Some p);
  [%cd
    ~~(p "param sgd step";
      "sgd_delta" =: p.grad + (!.weight_decay *. p);
      if Float.(momentum > 0.0) then (
        "sgd_momentum" =: (!.momentum *. sgd_momentum) + sgd_delta;
        if nesterov then sgd_delta += !.momentum *. sgd_momentum else sgd_delta =: sgd_momentum);
      p =- learning_rate * sgd_delta ~logic:".")]
```

Instead of adding the regularizer to the loss tensor, regularize here.

Specifies that computations should be pointwise.

Inline declarations of (non-differentiable) tensors.

After optimizations, this is the same as:

```
p =- learning_rate *. sgd_delta
```

# Stochastic Gradient Descent with apx. Nesterov Momentum

```
learning_rate:Tensor.t -> ?momentum:float -> ?weight_decay:float -> ?nesterov:bool -> Tensor.t -> Asgns.t
let sgd_one ~learning_rate ?(momentum = 0.0) ?(weight_decay = 0.0) ?(nesterov = false) p =
  if not @@ is_param p then raise @@ Tensor.Session_error ("Train.sgd_one: not a parameter", Some p);
  [%cd
    ~~(p "param sgd step";
      "sgd_delta" =: p.grad + (!.weight_decay *. p);
      if Float.(momentum > 0.0) then (
        "sgd_momentum" =: (!.momentum *. sgd_momentum) + sgd_delta;
        if nesterov then sgd_delta += !.momentum *. sgd_momentum else sgd_delta =: sgd_momentum);
      p =- learning_rate * sgd_delta ~logic:".")]
  
```

Instead of adding the regularizer to the loss tensor, regularize here.

Specifies that computations should be pointwise.

Side note: above is a binary accumulating assignment, below it's unary.

p =- learning\_rate \*. sgd\_delta

# Compilation illustrated by running bin/moons\_demo.ml

We don't use momentum; we use weight decay.

```
54 let update = Train.grad_update scalar_loss in
55 let%op learning_rate = 0.1 *. (!..steps - !@step_n) /. !..steps in
56 Train.set_hosted learning_rate.value;
57 let sgd = Train.sgd_update ~learning_rate ~weight_decay update in
58
59 let module Backend = (val Arrayjit.Backends.fresh_backend ~backend_name:"cc" ()) in
60 let device = Backend.(new_virtual_device @@ get_device ~ordinal:0) in
61 let ctx = Backend.init device in
62 let routine = Backend.(link ctx @@ compile bindings (Seq (update.fwd_bprop, sgd))) in
```

C compiler CPU backend.

We combine backpropagation and SGD update in a single routine;  
in more complex examples they'll be separate routines.

This file is for debugging, doesn't include indexing information (projections).

## Compilation – assignments: scalar\_loss\_gradient\_then\_sgd\_update.cd

```
2   scalar_loss_gradient_then_sgd_update (i1 : [0..31], i2):           ...
3     # "scalar_loss gradient update";
4     # "scalar_loss fwd"; Slice the node at the index
5     n39 := 32;
6     moons_class := moons_classes @| i1;                                ↗
7     moons_input := moons_flat @| i1;
8     n16 =:+ w1 * moons_input ~logic:"@";
9     n18 =: b1 + n16;
10    n20 =: ?/ n18;
11    n22 =:+ w2 * n20 ~logic:"@";
12    n24 =: b2 + n22;
13    n26 =: ?/ n24;
14    n28 =:+ w3 * n26 ~logic:"@";
15    mlp_moons_input =: b3 + n28;
16    n32 =: moons_class * mlp_moons_input ~logic:".";
17    n34 := 1;
18    n35 =: n34 - n32;
19    margin_loss =: ?/ n35;
20    n40 =:+ margin_loss ~logic:"...|... => 0";
21    scalar_loss =: n40 / n39 ~logic:".";
```

```
86    # "b3 param sgd step";
87    n67 := 0.0002;
88    n68 =: n67 * b3 ~logic:".";
89    sgd_delta_b3 =: b3.grad + n68;
90    b3 =- learning_rate * sgd_delta_b3 ~logic:".";
91    # "w1 param sgd step";
92    n71 := 0.0002;
93    n72 =: n71 * w1 ~logic:".";
94    sgd_delta_w1 =: w1.grad + n72;
95    w1 =- learning_rate * sgd_delta_w1 ~logic:".";
96    # "w2 param sgd step";
97    n75 := 0.0002;
98    n76 =: n75 * w2 ~logic:".";
99    sgd_delta_w2 =: w2.grad + n76;
100   w2 =- learning_rate * sgd_delta_w2 ~logic:".";
101   # "w3 param sgd step";
102   n79 := 0.0002;
103   n80 =: n79 * w3 ~logic:".";
104   sgd_delta_w3 =: w3.grad + n80;
105   w3 =- learning_rate * sgd_delta_w3 ~logic:".";
```

# Compilation – low level: scalar\_loss\_gradient\_then\_sgd\_update-unoptimized.ll

```
2  scalar_loss_gradient_then_sgd_update (i1 : [0..31], i2):
3      /* scalar_loss gradient update */
4      /* scalar_loss fwd */
5      n39[0] := 32;
6      for i4 = 0 to 31 {
7          moons_class[i4, 0] := moons_classes[i1, i4, 0];
8      }
9      for i7 = 0 to 31 {
10         for i8 = 0 to 1 {
11             moons_input[i7, i8] := moons_flat[i1, i7, i8];
12         }
13     }
14     zero_out n16;
15     for i12 = 0 to 31 {
16         for i13 = 0 to 15 {
17             for i14 = 0 to 1 {
18                 n16[i12, i13] := (n16[i12, i13] + (w1[i13, i14] * moons_input[i12, i14]));
19             }
20         }
21     }
```

```
268     /* w3 param sgd step */
269     n79[0] := 0.0002;
270     for i120 = 0 to 15 {
271         n80[0, i120] := (n79[0] * w3[0, i120]);
272     }
273     for i122 = 0 to 15 {
274         sgd_delta_w3[0, i122] := (w3.grad[0, i122] + n80[0, i122]);
275     }
276     for i124 = 0 to 15 {
277         w3[0, i124] := (w3[0, i124] - (learning_rate[0] * sgd_delta_w3[0, i124]));
278     }
279     /* end */
280     /* end */
```

...

# Compilation – optimized: scalar\_loss\_gradient\_then\_sgd\_update.ll

```
2  scalar_loss_gradient_then_sgd_update (i1 : [0..31], i2):
3      /* scalar_loss gradient update */
4      /* scalar_loss fwd */
5      for i7 = 0 to 31 {
6          for i8 = 0 to 1 {
7              moons_input[i7, i8] := moons_flat[i1,
8          }
9      }
10     zero_out n16;
11     for i12 = 0 to 31 {
12         for i13 = 0 to 15 {
13             for i14 = 0 to 1 {
14                 n16[i12, i13] := (n16[i12, i13] + (w1[i13, i14] * moons_input[i12, i14]));
15             }
16         }
17     }
18     for i21 = 0 to 31 {
19         for i22 = 0 to 15 {
20             n20[i21, i22] := relu((b1[i22] + n16[i21, i22]));
21         }
22     }
```

```
168     /* w3 param sgd step */
169     for i124 = 0 to 15 {
170         w3[0, i124] :=
171             (w3[0, i124] - (learning_rate[0] * (w3.grad[0, i124] + (0.0002 * w3[0, i124]))));
172     }
173     /* end */
174     /* end */
```

...

↑

Inlining reduces 3 loops to 1.



I removed empty lines to get a better idea of line count.

## Compilation – C code: scalar\_loss\_gradient\_then\_sgd\_update.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 /* Global declarations. */
5 #define learning_rate ((float*)0x555ef
6 #define w1 ((float*)0x55eff81d62b0)
7 #define b3 ((float*)0x55eff81f6890)
8 #define moons_classes ((float*)0x555ef
9 #define b1 ((float*)0x55eff81d6340)
10 #define moons_flat ((float*)0x55eff82
11 #define w2 ((float*)0x55eff81d4400)
12 #define b2 ((float*)0x55eff81d6390)
13 #define w3 ((float*)0x55eff81d2840)
14 #define scalar_loss ((float*)0x55eff818f840)
15
16 void scalar_loss_gradient_then_sgd_update(int i1, int i2) {
17     /* Local declarations and initialization. */
18     float b3_grad[1] = {0};
19     float n16[512] = {0};
20     float n20[512];
21     float n18_grad[512] = {0};
22     float w2_grad[256] = {0};
23     float mlp_moons_input_grad[32] = {0};
24     float n28_grad[32] = {0};
25
26     ...
27
28     /* scalar_loss zero grads */
29     /* end */
30     /* scalar_loss bprop */
31     for (int i55 = 0; i55 <= 31; ++i55) {
32         mlp_moons_input_grad[i55 * 1 + 0] =
33             (mlp_moons_input_grad[i55 * 1 + 0] +
34              (moons_classes[(i1 * 32 + i55) * 1 + 0] *
35                 ((float)(-1) *
36                  (fmaxf(0.0, ((float)(1) - (moons_classes[(i1 * 32 + i55) * 1 + 0] * (b3[0] + n28[i55 * 1 + 0])))) *
37                  (float)(0.03125) : 0.0)));
38     }
39
40     ...
41
42     /* w3 param sgd step */
43     for (int i124 = 0; i124 <= 15; ++i124) {
44         w3[0 * 16 + i124] =
45             (w3[0 * 16 + i124] -
46                 (learning_rate[0] * (w3_grad[0 * 16 + i124] + ((float)(0.0002) * w3[0 * 16 + i124])));
47
48         ...
49
50         /* end */
51         /* end */
52     }
53
54     ...
55 }
```

Zeroing gradients got optimized away.

# Running bin/moons\_demo.ml

```
lukstafi@DESKTOP-6RRUNR4:~/ocannl$ dune exec bin/moons_demo.exe
```

Welcome to OCANNL! Reading configuration defaults from /home/lukstafi/ocannl/ocannl\_config.

Retrieving commandline, environment, or config file variable ocannl\_log\_level

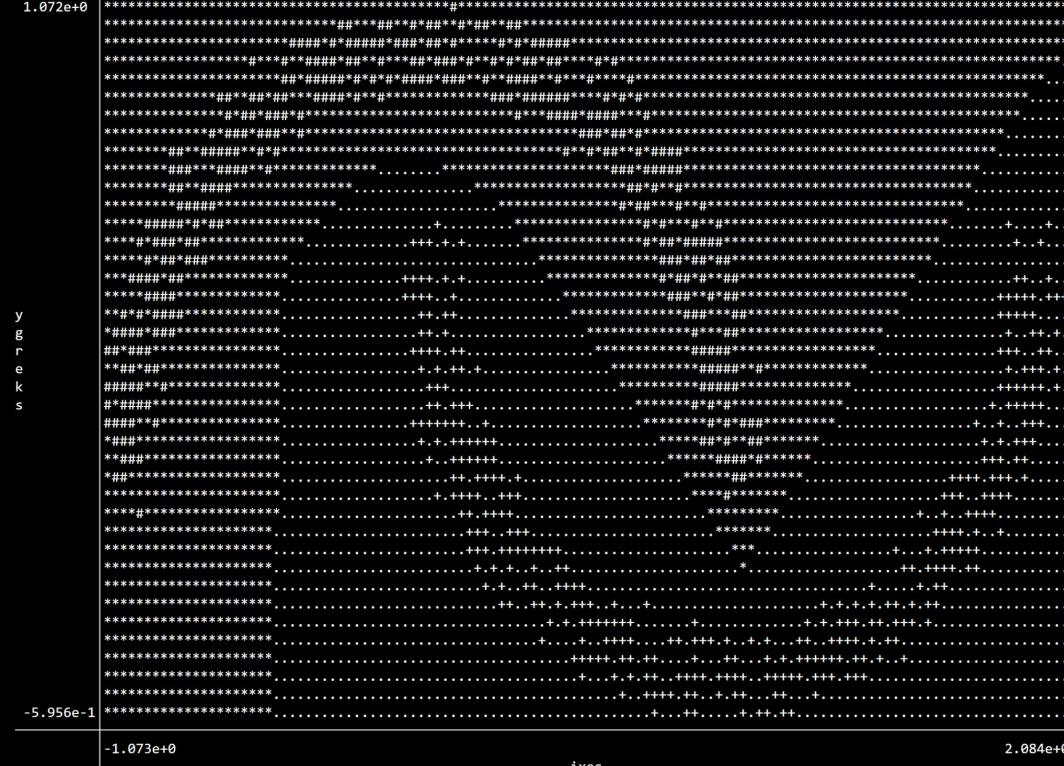
Found 0, in the config file

.. ..

Epoch 73, lr=0.001375, epoch loss=0.006816

Epoch 74, lr=0.000042, epoch loss=0.006639

Half-moons scatterplot and decision boundary:



# cuda-gdb session, including CUDA source position

```
lukstafi@DESKTOP-6RRUNR4:~/ocannl$ dune build bin/moons_demo.exe
lukstafi@DESKTOP-6RRUNR4:~/ocannl$ /usr/local/cuda-12.5/bin/cuda-gdb _build/default/bin/moons_demo.exe
NVIDIA (R) cuda-gdb 12.5
Portions Copyright (C) 2007-2024 NVIDIA Corporation
Based on GNU gdb 13.2
Copyright (C) 2023 Free Software Foundation, Inc.

Reading symbols from _build/default/bin/moons_demo.exe...
(cuda-gdb) break scalar_loss_gradient_then_sgd_update.cu:252
No source file named scalar_loss_gradient_then_sgd_update.cu.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (scalar_loss_gradient_then_sgd_update.cu:252) pending.
(cuda-gdb) run
Starting program: /home/lukstafi/ocannl/_build/default/bin/moons_demo.exe
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Welcome to OCANNL! Reading configuration defaults from /home/lukstafi/ocannl/ocannl_config.

CUDA thread hit Breakpoint 1, scalar_loss_gradient_then_sgd_update<<<(1,1,1),(1,1,1)>>> (
    i1=0, i2=0, scalar_loss=0x705c03000, w3=0x705c02e00, b2=0x705c02c00, w2=0x705c02800,
    moons_flat=0x705c01400, b1=0x705c01200, moons_classes=0x705c00800, b3=0x705c00600,
    w1=0x705c00400, learning_rate=0x705c00200) at scalar_loss_gradient_then_sgd_update.cu:252
252      b3[0] = (b3[0] - (learning_rate[0] * (b3_grad[0] + ((0.000200) * b3[0]))));
(cuda-gdb) print b3_grad[0]
$1 = 0.50000006
(cuda-gdb) print learning_rate[0]
$2 = 0.0989999995
```

```
BEGIN DEBUG SE
("Set log_level t
  • {orphaned fro
▶ set_log_level
▶ optimize_proc
▶ cuda_to_ptx
▶ link
▶ create_array
▶ create_array
▶ from_host
▶ _train_loop
▶ optimize_proc
▶ cuda_to_ptx
▶ link
▶ _plotting
```

# ppx\_minidebug logs, including cuda backend comps!

```
▼ _train_loop
  • "bin/moons_demo.ml":86:17 at elapsed 752ms / 751603304ns
    • ▼ for:moons_demo:87
      ○ "bin/moons_demo.ml":87:4 at elapsed 752ms / 751611474ns
        ○ epoch = 0
        ○ ▼ <for epoch>
          ▪ "bin/moons_demo.ml":87:8 at elapsed 752ms / 751614543ns
          ▪ ▼ for:moons_demo:88
            ▪ "bin/moons_demo.ml":88:6 at elapsed 752ms / 751615151
              ▪ batch = 0
              ▪ ▼ <for batch>
                ▪ "bin/moons_demo.ml":88:10 at elapsed 752ms /
                ▪ ▼ fun:moons_demo:90
                  ▪ "bin/moons_demo.ml":90:37 at elapsed 752ms /
                  ▪ ▶ (run "launches scalar_loss_gradient"
                  ▪ ▶ to_host
                  ▪ ▶ to_host
                  ▪ ▼ on dev 0_0
                    ▪ ▼ log_trace_tree
                      ▪ "arrayjit/lib/utils.ml":441:32 at elap
                      ▪ ▼ trace tree
                        ▪ float *b1 = 0x506600a00
                        ...
```

```
...  
  ▪ int i2 = 0  
  ▪ ▼ scalar_loss gradient update
    ▪ ▶ scalar_loss fwd
    ▪ ▶ scalar_loss bprop
  ▪ ▼ scalar_loss sgd update
    ▪ ▼ b1 param sgd step
      ▪ ▶ learning_rate[0]{=0} = 0.1
    ▪ index i88 = 0
    ▪ ▼ b1[0]{=0.85595} = 0.817884
      ▪ b1[i88] := (b1[i88] - (learning_rate[0] * (b1.grad[i88] + (0.0002 * b1[i88]))));
      ▪ = (b1[0]{=0.85595} - (learning_rate[0]{=0.1} * (b1.grad[0]{=0.380482} +
        index i88 = 1
      ▪ ▶ b1[1]{=0.0379179} = 0.0359901
```

# Data parallel training



- Subdivide a batch into mini-batches for each device.
- Schedule copying the data to each device.
- Schedule updating gradients (running `fwd_bprop`) on each device.
- Pairwise merge the gradients, in each round adding a gradient from device  $\text{half}+i$  to device  $i$ , (for  $i$  from 0 to  $\text{half}-1$ ) until all gradients are added to device 0. (The destination needs to synchronize on the source.)
- Run SGD update on device 0.
- Schedule copying parameters from device 0 to all other devices.

# Multi-stream computations in OCANNL

- OCANNL has a loose notion of a **stream**: e.g. CPU cores and CUDA streams.
- Code might be **compiled** for a backend independent of any device, but is **linked with a context** for execution, forming a new context.
  - Context → stream → device.
- **Tensor nodes** needed for the computation are represented (if needed) on the device as **arrays** (via the context or its ancestors).

# Multi-device computations in OCANNL (design might likely change)

- OCANNL has a loose notion of a **stream**: e.g. CPU cores and CUDA streams.
- Code might be **compiled** for a backend independent of any device, but is **linked** with a **context** for execution, forming a new context.
- **Tensor nodes** needed for the computation are represented (if needed) on the device as **arrays** (via the context or its ancestors).
- Each stream has a **merge buffer**, which represents an array coming from another device.
  - It can come via: copying, directly pointing to (for CPU or streams on the same GPU), someday streaming...
- Regular device-to-device data transfer for a tensor node writes into its destination array, but one `into_merge_buffer` does not.

# Data parallel training: merging gradients in OCANNL

```
let grad_merges : Asgns.t array =
  Array.map all_params ~f:(fun p -> [%cd p.grad += p.grad.merge])
in
let grad_merges_to : Backend.routine option array array =
  Array.mapi ctxs ~f:(fun dst_n ctx -
    if occupancy_dst ~dst_n then
      snd @@ Backend.link_batch ctx
      @@ Backend.compile_batch ~shared:true ~occupancy:Idx.Empty grad_merges
    else [])
in

let merge_grads ~(from : int) ~(to_ : int) : unit =
  Array.iteri all_params ~f:(fun i p -
    let grad_merge = Option.value_exn grad_merges_to.(to_).(i) in
    assert (
      Backend.device_to_device (Option.value_exn p.diff).grad ~into_merge_buffer:BT.Copy
      ~dst:grad_merge.context ~src:ctxs.(from));
    (Task.run grad_merge.schedule : unit))
in
```

p.grad must be sent into\_merge\_buffer before this code runs.

Only compile the code if it's needed.

Synchronization now happens automatically.

Fill the merge buffer before running merging.

# Thank you! Questions?

For example:

- “Is it functional?”
  - tl;dr: no, not like JAX, less than OWL.
- “What are the future directions?”

