OCANNL

Mysteries of NN training unveiled

OCaml **C**ompiles **A**lgorithms for **N**eural **N**etwork **L**earning

- 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 **C**ompiles **A**lgorithms for **N**eural **N**etwork **L**earning

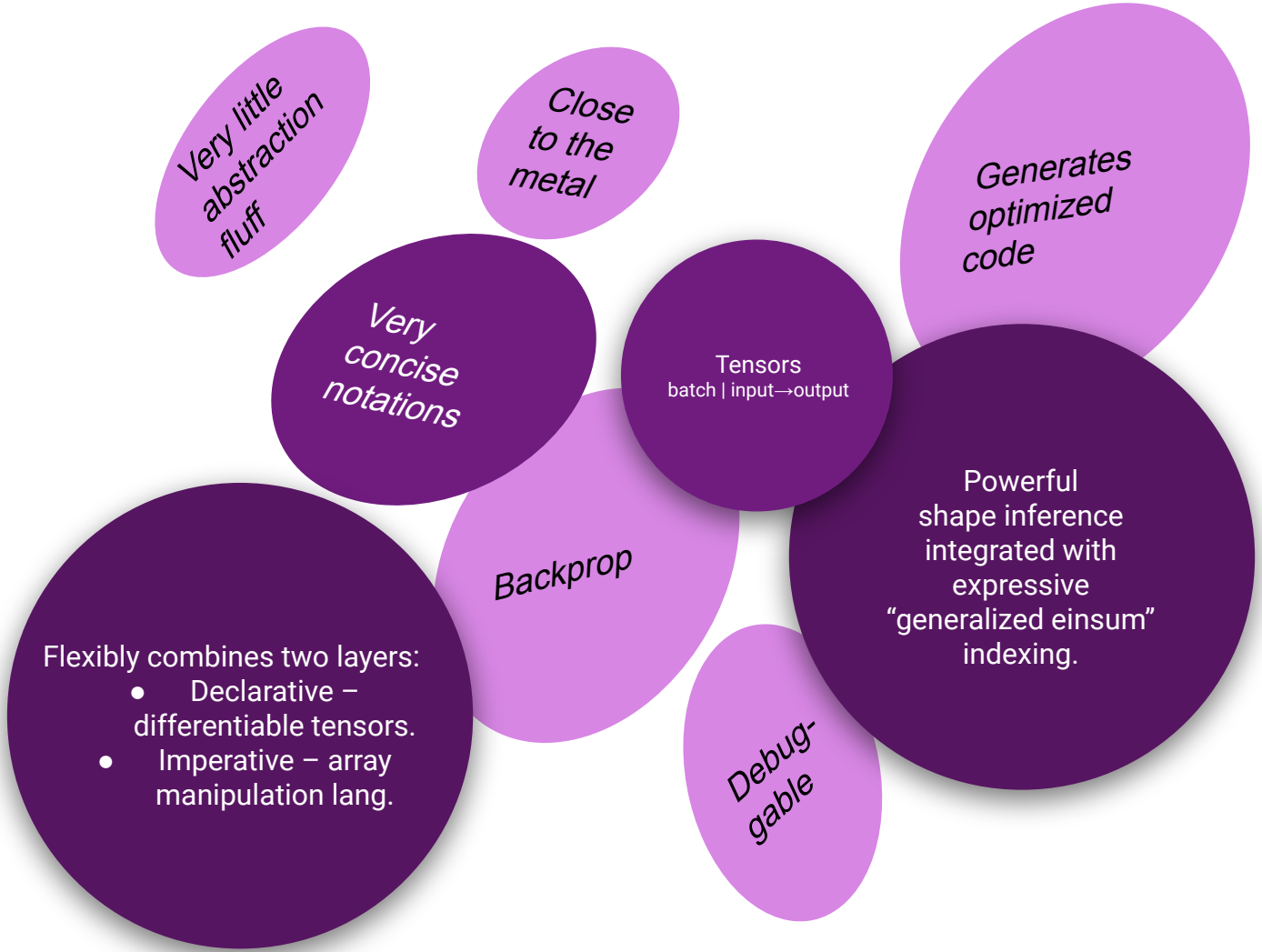
- OCANNL is distributed as two opam packages:
  - `arrayjit`: a backend for compilers for numerical array programming languages,
  - `neural_nets_lib`: a neural networks (Deep Learning) framework.
- You express numerical computations at runtime, OCANNL will compile them.
- There are startups doing it 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 **C**ompiles **A**lgorithms for **N**eural **N**etwork **L**earning

- 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.

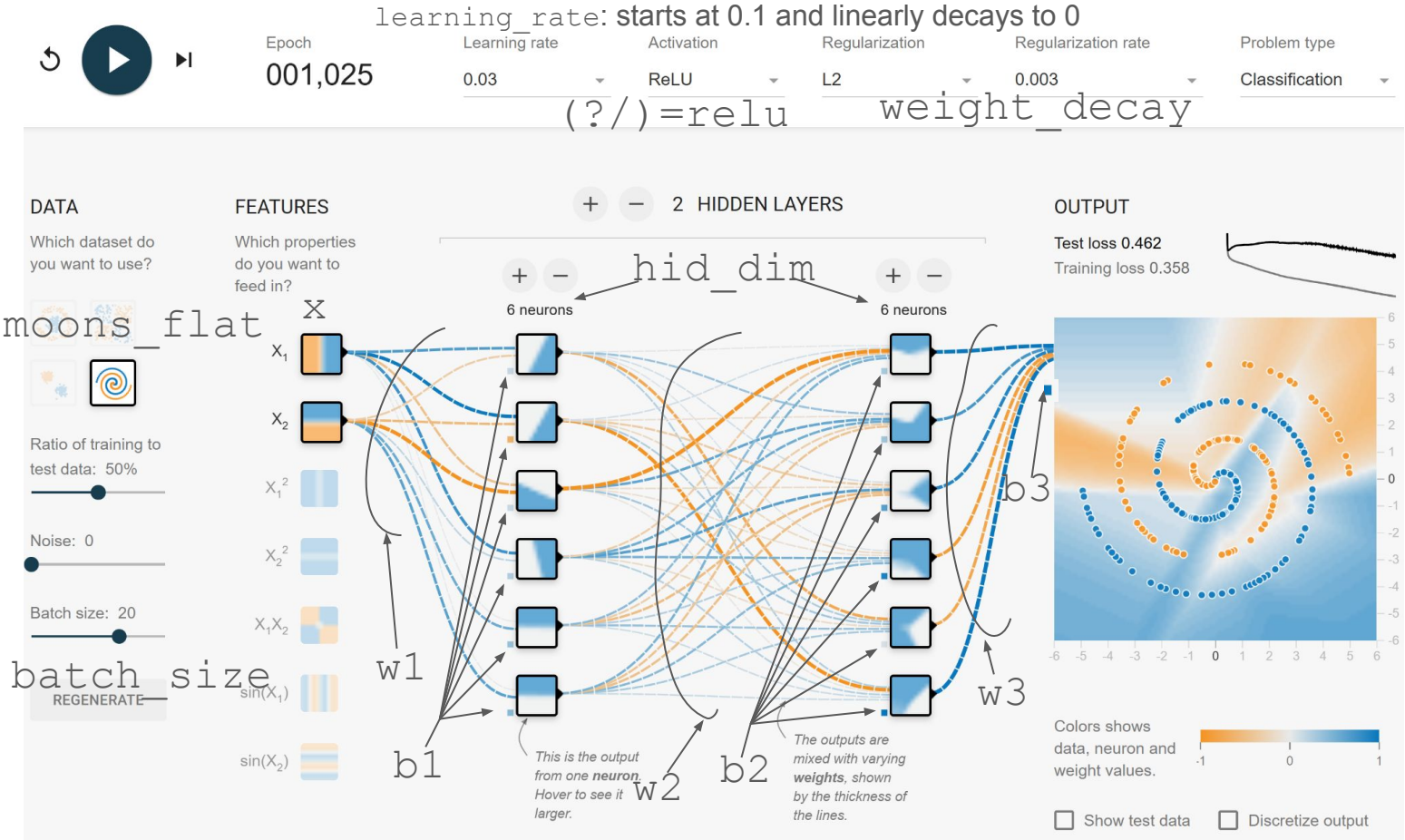
OCANNL is still at a proof-of-concept stage, but it will grow and evolve.


Very little abstraction fluff
Close to the metal
Generates optimized code
Very concise notations
Tensors
batch | input→output
Backprop
Powerful shape inference integrated with expressive “generalized einsum” indexing.
Flexibly combines two layers:
• Declarative – differentiable tensors.
• Imperative – array manipulation lang.
Debug-gable


SKIP
CLASS

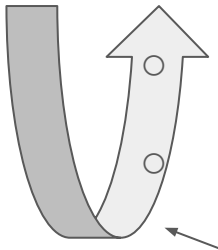
Let’s train a feed-forward neural network with 2 hidden layers (aka. a 3-layer MLP) to classify points on a plane.

[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.
- Express a solution as a parameterized differentiable computation.
- Figure out a formula for how bad the solution is on a datapoint.
- For each datapoint in the dataset:
  - Compute the direction for parameters to make them better/worse.
  - Update parameters to make them better.



a dataset

a model

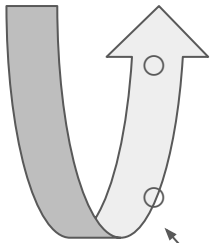
a loss function

a gradient

Stochastic Gradient Descent (SGD)

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

- Collect examples to learn from.
- Express a solution as a parameterized differentiable computation.
- Figure out a formula for how bad the solution is on a datapoint.
- 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.



a dataset

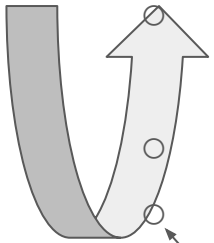
a model

a loss function

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

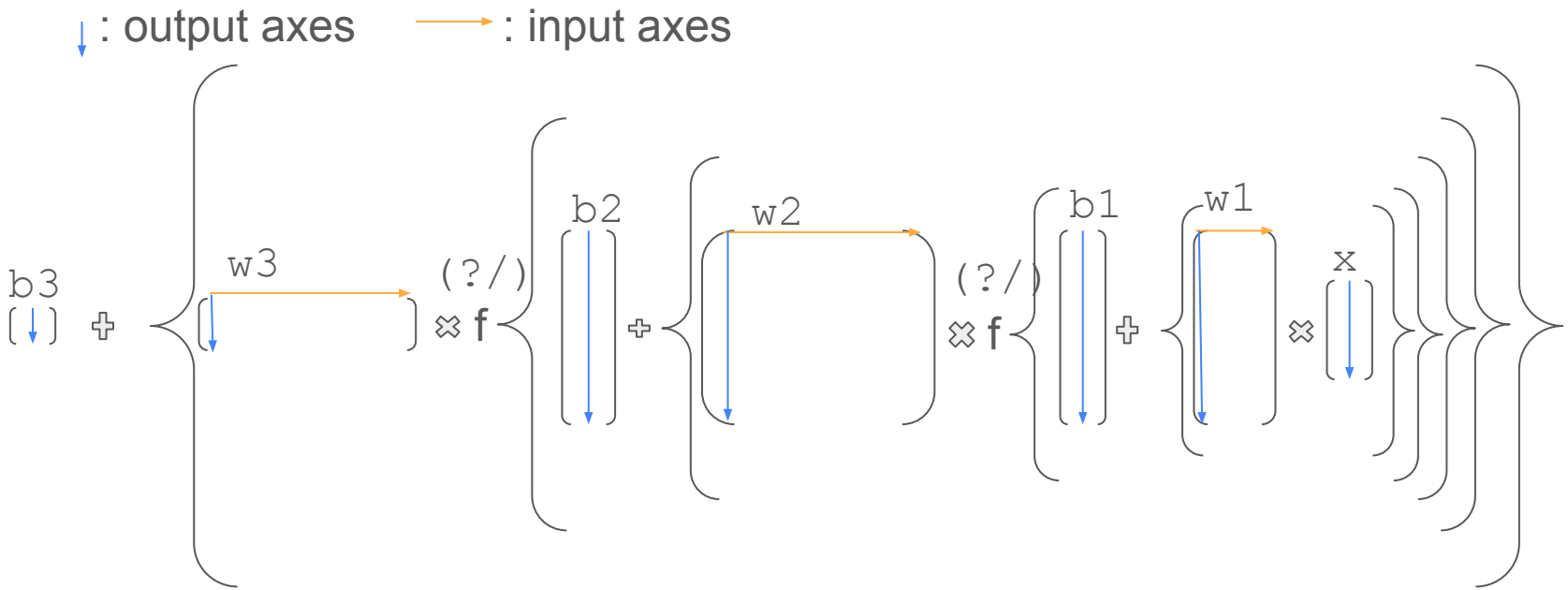

```
let noise () = Rand.float_range (-0.1) 0.1 in
let moons_flat =
  Array.concat_map (Array.create ~len ())
    ~f:
      Float.(
        fun () ->
          let i = Rand.int len in
          let v = of_int i * pi / of_int len in
          let c = cos v and s = sin v in
          [| c + noise (); s + 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)

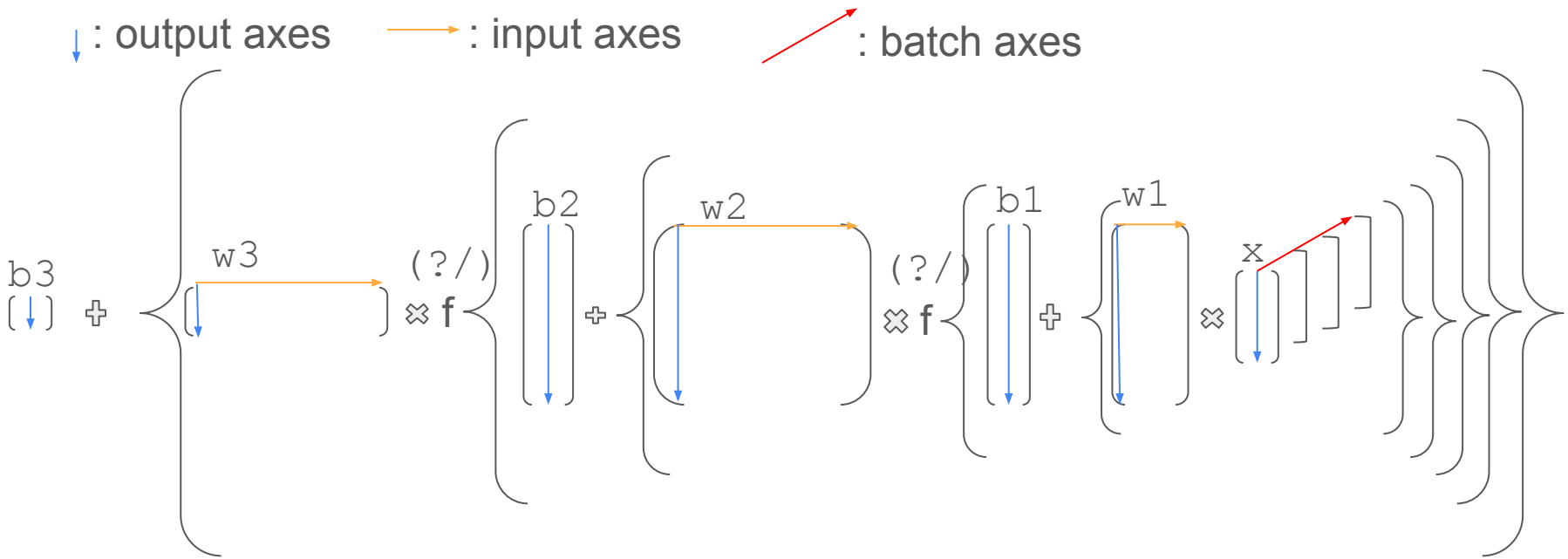
Half-moons toy dataset


```
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

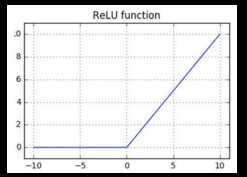


Tensors as differentiable multidimensional matrices



Multi Layer Perceptron in one line of code

?/ =

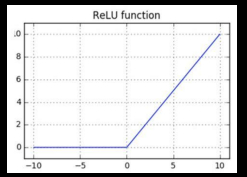



```
let%op mlp x =
  "b3" + ("w3" * ?/("b2" hid_dim + ("w2" * ?/("b1" hid_dim + ("w1" * x))))) in
```

Multi Layer Perceptron in one line of code

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

?/ =



Tensor (e.g. matrix) multiplication

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 function, expands to:
`let w1 = … in let b1 = … in … let mlp x = … in …`

Declarative expressions for differentiable tensor operations

Hinge loss function: maximum margin classification

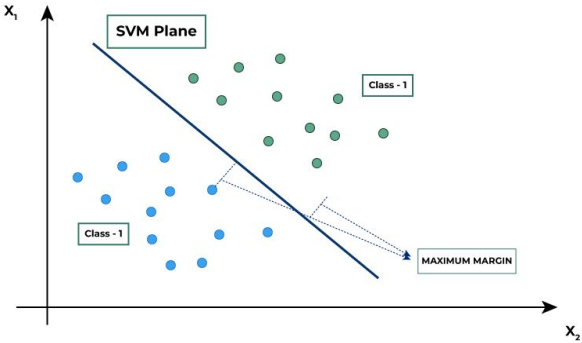
$$L(y, f(x)) = \max(0, 1 - y * f(x))$$

pointwise multiplication

```
let%op margin_loss = ?/(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`



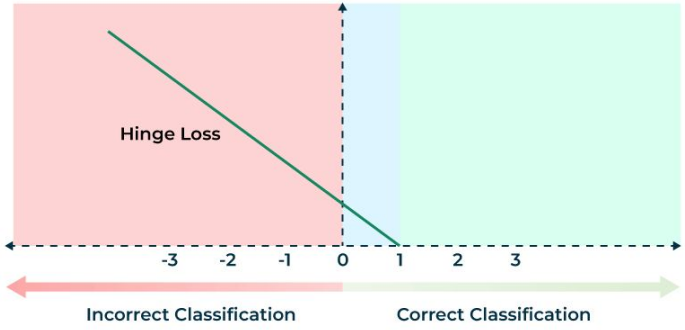


Image source:
[Hinge-loss & relationship with Support Vector Machines - GeeksforGeeks]

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 p^2 is $2p$.

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: $x \to y(x) \to f(y(x))$, but $df \to df/dy \to df/dx$.
    - 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]

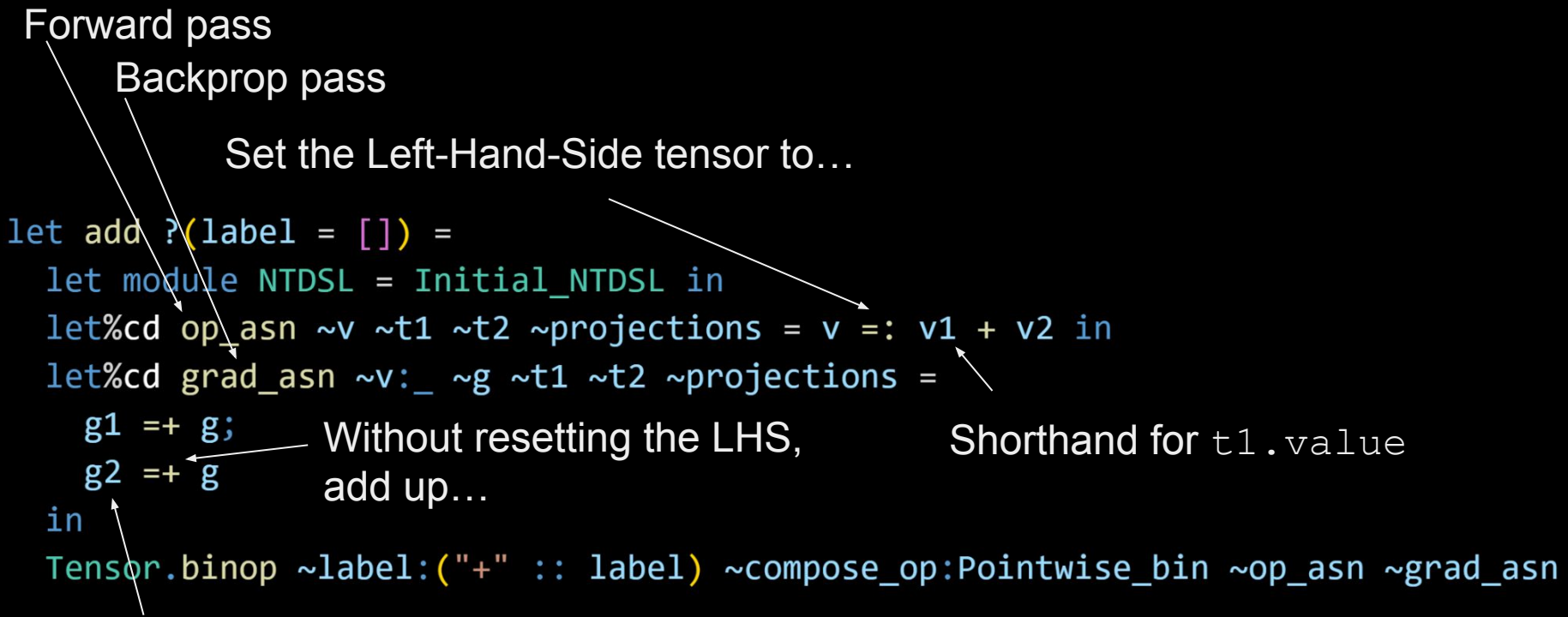
Backprop by example

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 ~v:_ ~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



Backprop by example

When tensor `t = t1 + t2`, the tensor value node is `t.value =: t1.value + t2.value`, shorthand `v =: v1 + v2`.

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 ~v:_ ~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

Both gradients `t1.grad` and `t2.grad` increase by `t.grad` because e.g. $d(t_1+t_2)/dt_1=1$.

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 ~v:_ ~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

Gradient `t1.grad` increases and `t2.grad` decreases by `t.grad` because $d(t_1-t_2)/dt_2=-1$.

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 ~v:_ ~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, deduct up…

Shorthand for `v1.grad`

Shorthand for `t2.grad`

Backprop by example

```
let mul compose_op ~op_asn =
  let module NTDSL = Initial_NTDSL in
  let%cd grad_asn ~v:_ ~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 = t1 * t2`, the value is `v =:+ v1 * v2`, which means: first set all cells of `v` to zero, then add up `v1[...] * v2[...]` for all combinations of those indices used by `v1` and `v2` 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 ~v:_ ~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. 22: 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 ~v:_ ~g:_ ~t1:_ ~t2:_ ~projections:_ -> Asgns.Noop
    else if Float.equal p 2.0 then fun ~v:_ ~g ~t1 ~t2:_ ~projections -> g1 =+ p_t *. t1 * g
    else if Float.equal p 1.0 then fun ~v:_ ~g ~t1 ~t2:_ ~projections -> g1 =+ g
    else fun ~v:_ ~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 `t1` 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 -> 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 ~v:_ ~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.`

$$\nabla(\frac{t1}{t2}) = \frac{t2\nabla t1 - t1\nabla t2}{t2^2}$$

Putting the forward and backward passes together

Tensor.t -> updaten

```
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 canion, 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:

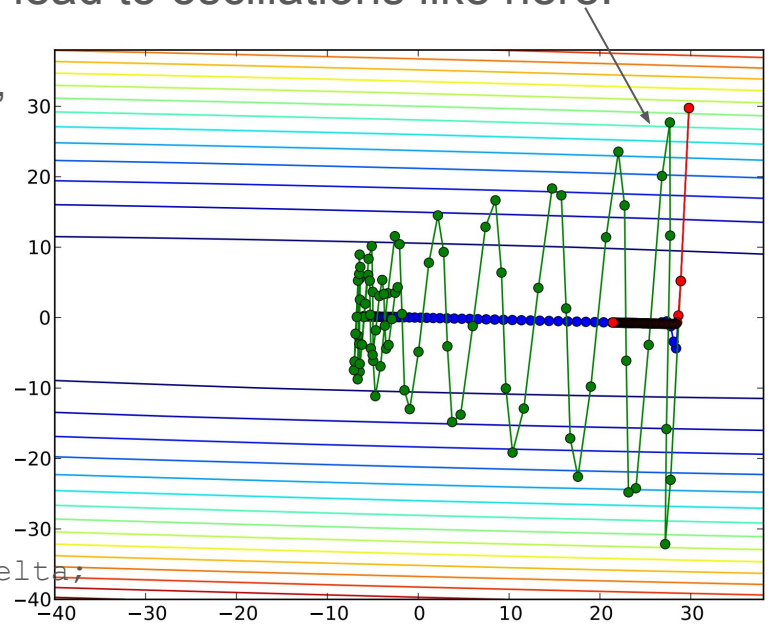


Image from [On the importance of initialization and momentum in deep learning (mlr.press)]

```
"sgd_delta" =: p.grad;

"sgd_momentum" =: (!.momentum *. sgd_momentum) + sgd_delta;

sgd_delta =+ !.momentum *. sgd_momentum;

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.

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.

Inline declarations of (non-differentiable) tensors.

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.

C compiler CPU backend.

```
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
```

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";
 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
277    for i124 = 0 to 15 {
278      w3[0, i124] := (w3[0, i124] - (learning_rate[0] * sgd_delta_w3[0, i124]));
279    }
280    /* end */
281    /* 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`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Global declarations. */
#define learning_rate ((float*)0x55ef
#define w1 ((float*)0x55eff81d62b0)
#define b3 ((float*)0x55eff81f6890)
#define moons_classes ((float*)0x55ef
#define b1 ((float*)0x55eff81d6340)
#define moons_flat ((float*)0x55eff82
#define w2 ((float*)0x55eff81d4400)
#define b2 ((float*)0x55eff81d6390)
#define w3 ((float*)0x55eff81d2840)
#define scalar_loss ((float*)0x55eff818f840)

void scalar_loss_gradient_then_sgd_update(int i1, int i2) {
  /* Local declarations and initialization. */
  float b3_grad[1] = {0};
  float n16[512] = {0};
  float n20[512];
  float n18_grad[512] = {0};
  float w2_grad[256] = {0};
  float mlp_moons_input_grad[32] = {0};
  float n28_grad[32] = {0};
```

...

```c
  /* scalar_loss zero grads */
  /* end */
  /* scalar_loss bprop */
  for (int i55 = 0; i55 <= 31; ++i55) {
    mlp_moons_input_grad[i55 * 1 + 0] =
      (mlp_moons_input_grad[i55 * 1 + 0] +
       (moons_classes[(i1 * 32 + i55) * 1 + 0] *
        ((float)(-1) *
         (fmaxf(0.0, ((float)(1) - (moons_classes[(i1 * 32 + i55) * 1 + 0] * (b3[0] + n28[i55 * 1 + 0])))) > 0.0 ? (float)(0.03125) : 0.0))));
  }
```

Zeroing gradients got optimized away.

...

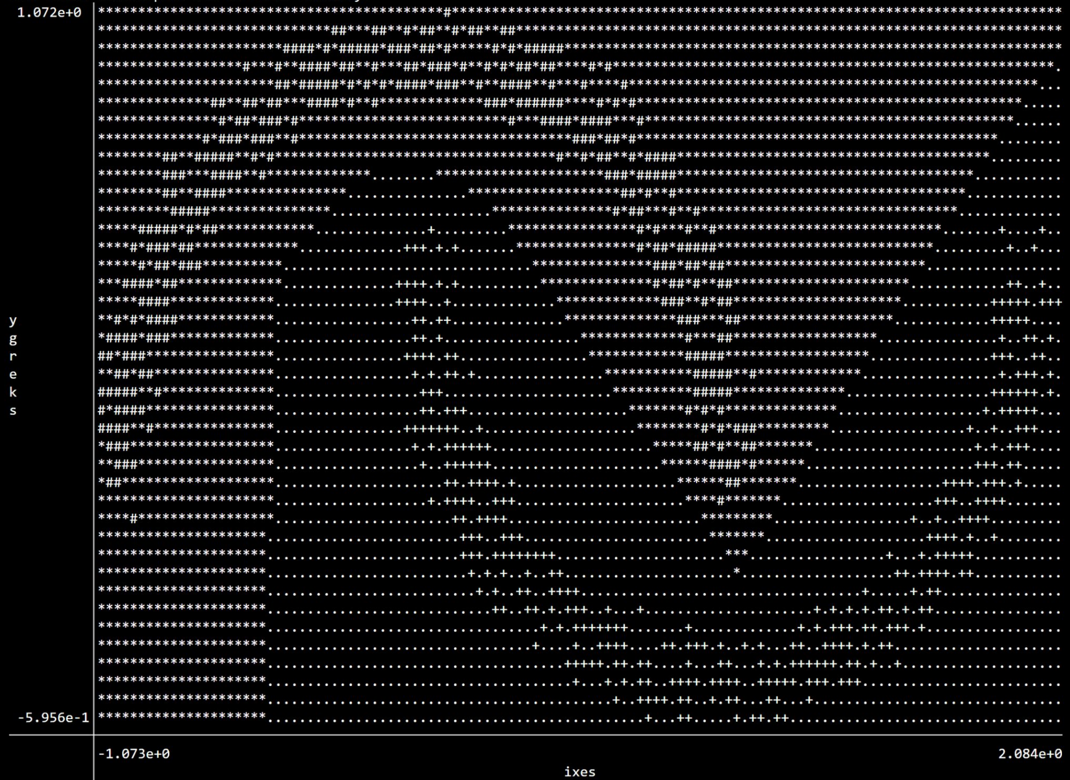
```c
  /* w3 param sgd step */
  for (int i124 = 0; i124 <= 15; ++i124) {
    w3[0 * 16 + i124] =
      (w3[0 * 16 + i124] -
       (learning_rate[0] * (w3_grad[0 * 16 + i124] + ((float)(0.0002) * w3[0 * 16 + i124]))));
  }
  /* end */
  /* end */
}
```

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

ppx_minidebug logs, including cuda backend comps!

- ▶ set_log_level
- ▶ optimize_proc
- ▶ cuda_to_ptx
- ▶ link
- ▶ create_array
- ▶ create_array
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ from_host
- ▶ _train_loop
- ▶ optimize_proc
- ▶ cuda_to_ptx
- ▶ link
- ▶ _plotting

▼ _train_loop

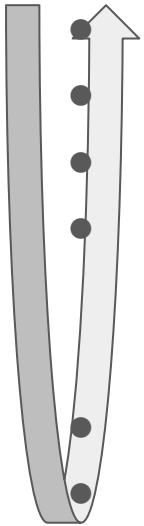
- "bin/moons_demo.ml":86:17 at elapsed 752ms / 751603304ns
- ▼ for:moons_demo:87
  - "bin/moons_demo.ml":87:4 at elapsed 752ms / 751611474ns
  - `epoch = 0`
  - ▼ <for epoch>
    - "bin/moons_demo.ml":87:8 at elapsed 752ms / 751614543ns
    - ▼ for:moons_demo:88
      - "bin/moons_demo.ml":88:6 at elapsed 752ms / 751615
      - `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 75
          - ▶ (run "launches scalar_loss_gradient_
          - ▶ to_host
          - ▶ to_host
        - ▼ on dev 0_0
          - ▼ log_trace_tree
            - "arrayjit/lib/utils.ml":441:32 at elaps
            - ▼ 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 `half+i` to device `i`, (for `i` from `0` to `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-device computations in OCANNL (design might likely change)

- OCANNL has a loose notion of a **device**: e.g. CPU cores and CUDA streams.
- Code might be **compiled** for a backend independent of any device, but is **linked** with a device **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).

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

- OCANNL has a loose notion of a **device**: e.g. CPU cores and CUDA streams.
- Code might be **compiled** for a backend independent of any device, but is **linked** with a device **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 device 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 into_merge_buffer = if copy_to_merge then BT.Copy else BT.Streaming in
let merge_grads ~(from : int) ~(to_ : int) : unit =
  (* Synchronize the source since we compute on the destionation. *)
  Backend.(await @@ get_ctx_device ctxs.(from));
  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
          ~dst:grad_merge.context ~src:ctxs.(from));
      (Tn.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.

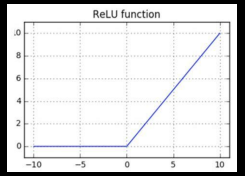
Fill the merge buffer before running merging.

Appendix: old slides below

Multi Layer Perceptron in one line of code

```
let%op mlp x = "b3" + ("w3" * ?/("b2" hid_dim + ("w2" * ?/("b1" hid_dim + ("w1" * x))))) 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
let batch_n, bindings = IDX.get_static_symbol ~static_range:n_batches IDX.empty in
let step_n, bindings = IDX.get_static_symbol bindings in
let%op moons_input = moons_flat @| batch_n in
let%op moons_class = moons_classes @| batch_n in
let%op margin_loss = ?/(1 - (moons_class *. mlp moons_input)) in
let%op scalar_loss = (margin_loss ++ "...|... => 0") /. !..batch_size in
let update = Train.grad_update scalar_loss in
let%op learning_rate = 0.1 *. (!..steps - !@step_n) /. !..steps in
Train.set_hosted learning_rate.value;
let sgd = Train.sgd_update ~learning_rate ~weight_decay update in
```

Specify just the output dimensions, batch dimensions will be inferred

`*.` is pointwise multiplication

Indices set in OCaml and passed to the backend code

Slice the tensors at their last batch axis

Pointwise division by the integer `steps`

Sum out all batch and output axes i.e. sum all values of `margin_loss`

Vanilla training loop

```
let module Backend = (val Train.fresh_backend ()) in
let device = Backend.(new_virtual_device @@ get_device ~ordinal:0) in
let ctx = Backend.init device in
let routine = Backend.(link ctx @@ compile bindings (Seq (update.fwd_bprop, sgd))) in
Train.all_host_to_device (module Backend) routine.context scalar_loss;
Train.all_host_to_device (module Backend) routine.context learning_rate;
let open Operation.At in
let step_ref = IDX.find_exn routine.bindings step_n in
let batch_ref = IDX.find_exn routine.bindings batch_n in
step_ref := 0;
let%track_this_sexp _train_loop : unit =
  for epoch = 0 to epochs - 1 do
    for batch = 0 to n_batches - 1 do
      batch_ref := batch;
      Train.run routine;
      assert (Backend.to_host routine.context learning_rate.value);
      assert (Backend.to_host routine.context scalar_loss.value);
      Backend.await device;
      learning_rates := learning_rate.@[0] :: !learning_rates;
      losses := scalar_loss.@[0] :: !losses;
      epoch_loss := !epoch_loss +. scalar_loss.@[0];
      Int.incr step_ref
    done;
    Stdio.printf "Epoch %d, lr=%f, epoch loss=%f\n%!" epoch learning_rate.@[0] !epoch_loss;
    epoch_loss := 0.
  done
in
```

This computes values and gradients for nodes of the tensor expression `scalar_loss`

This computes the update of the parameters inside the `scalar_loss` expression

That's for ppx_minidebug tracing

Tell the device to run the code when it's ready, and then to copy learning_rate and scalar_loss for accessing from OCaml.

Almost: `scalar_loss.value.array.{[|0|]}`

Vanilla inference loop

```
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%op mlp_result = mlp "point" in
Train.set_on_host Changed_on_devices mlp_result.value;
let result_routine =
  Backend.(
    link routine.context @@ compile IDX.empty @@ Block_comment ("moons infer", mlp_result.forward))
in
let callback (x, y) =
  Tensor.set_values point [| x; y |];
  assert (Backend.from_host result_routine.context point.value);
  Train.run result_routine;
  assert (Backend.to_host result_routine.context mlp_result.value);
  Backend.await device;
  Float.(mlp_result.@[0] >= 0.)
in
```

We want it in OCaml, don't optimize it away

Tell the device to fetch `point.value` from the OCaml side

```
let%track_this_sexp _plotting : unit =
  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 = "%" };
        Boundary_map { pixel_false = "."; pixel_true = "*"; callback };
      ]
  in
  Stdio.printf "\nHalf-moons scatterplot and decision boundary:\n%!";
  PrintBox_text.output Stdio.stdout plot_moons
in
Backend.unsafe_cleanup ~unsafe_shutdown:true ()
```