CAMELAX: A LIGHTWEIGHT MACHINE LEARNING LIBRARY IN OCAML

Devan Shah & Daniel Yang
Department of Computer Science, Princeton
{devan.shah,daniel.yang}@princeton.edu
GitHub Repository

1 INTRODUCTION

Deep learning has had incredible success in recent years, due largely to increasingly large datasets and algorithms capable of scaling better with the amount of data (16). Frontier labs are training Large Language Models on trillions of tokens (7) across thousands of GPUs (19), and so deep learning libraries have naturally adapted to focus on parallelism and distributed training. However, maintaining global state can be challenging when operations are executed concurrently or across devices. In part to address these challenges, Google developed JAX, a high-performance machine learning library, built on functional principles and allowing for better parallelization and consistent execution for machine learning tasks (2). JAX has become very popular for reinforcement learning research (15) due to its execution speed and has wide-spread use at Google (5).

For our project, we intend to reproduce core features of PyTorch (13) and JAX in OCaml to produce a new library: CAMELAX. We hope this will empower OCaml machine learning workflows while providing an intuitive and OCaml-first library for performant single-threaded CPU training. We additionally strive for a clean, maintainable codebase that is easily modifiable and adaptable. We demonstrate Camelax on a variety of tasks, including the MNIST digit labeling task (8).

2 CAMELAX

In this project, we build a JAX-inspired machine-learning library in OCaml. Although JAX places an emphasis on efficient numerical computing, we intend to focus primarily on neural-network training, providing support for critical functions, training methods, and vector operations for realistic workloads. A GitHub repository for our project is linked at the start of our paper (and here).

In this project, we demonstrate:

- Training a neural network for scalar multiplication in Camelax.
- Training a neural network for binary classification of numbers under basic rules (i.e., in circle) in Camelax.
- Training a neural network on the MNIST dataset for digit classification.
- =We provide a basic GUI to test trained MNIST models.
- Training an MNIST Autoencoder to embed images and generate images.

This will require:

- Tensor/Array/Vector operations (MatMul, arithmetic, etc.)
- Neural Network operations (layers, ReLU, Sigmoid, Tanh, Softmax etc.)
- Initialization mechanisms (Kaiming-He (6), zero, from array, etc.)
- Loss Functions (cross-entropy loss, mean-squared error)
- Auto-differentiation across operations

- Optimizers (Gradient Descent)
- Training infrastructure (data loading, weight saving)

3 BACKGROUND

3.1 INTRODUCTION TO DEEP LEARNING

The crux of machine learning is to learn patterns from data. Given inputs x and outputs $y = f(x)$, typically in the form of dataset of n examples notated $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, machine learning aims to learn a function capable of predicting $f(x')$ for an unseen input x'. In this sense, machine learning allows us to generalize and predict relationships, offering powerful tools that have allowed for models capable of generating text (3), generating images (18), predicting medical diagnoses from X-Rays (14), playing Atari games (10), and so forth.

As a more concrete example, let $x_i \in \mathbb{R}^{28\times 28}$ represent 28 by 28 pixel images of different handwritten digits, and $y_i \in \{0, \dots 9\}$ represents the ground-truth label for the digit in the image. This is exactly the task of the MNIST dataset (Figure 1), a foundational machine learning challenge for visual tasks (8).

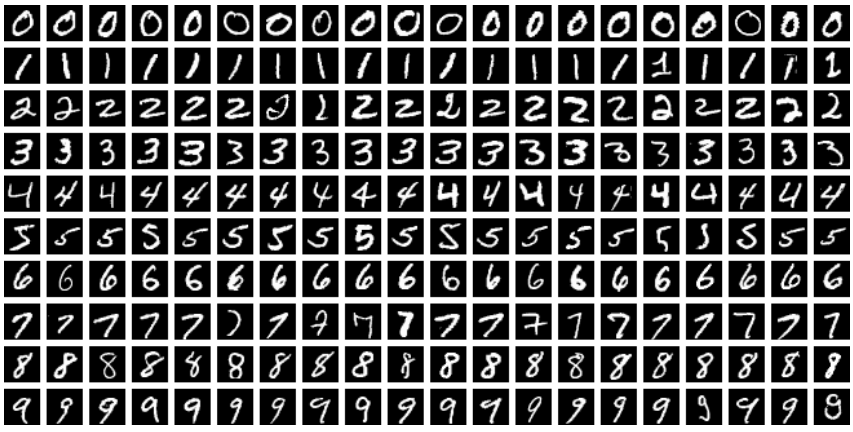

Figure 1: Sample images from MNIST test dataset. Image from (17).

One way we could learn this relationship is with a Multi-Layer Perceptron, the most common type of neural network that involves ℓ layers of matrix multiplication, addition, and activation functions.

For this challenge, we may parameterize our model with $\ell = 3$, compacting the input to $x_i \in \mathbb{R}^{784}$, and aiming to find the proper parameters $W_1 \in \mathbb{R}^{128\times 784}$, $W_2 \in \mathbb{R}^{64\times 128}$, and $W_3 \in \mathbb{R}^{10\times 64}$ and bias vectors $b_1 \in \mathbb{R}^{128}, b_2 \in \mathbb{R}^{64}, b_2 \in \mathbb{R}^{10}$ so that, for all i,

$$y_i \approx \tilde{f}(x_i) = \text{softmax} \circ (f : v \to W_3 v + b_3) \circ (f : v \to \sigma(W_2 v + b_2)) \circ \sigma(W_1 x_i + b_1))$$

In the above, $x_i \in \mathbb{R}^{784}$ represents the input vector, $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ is an activation function, which we will discuss in more detail shortly, and softmax normalizes the output 10-dimensional vector so that it is a probability distribution, for reasons we will further discuss. The character $\circ$ represents function composition, equivalently we could write the above as:

$$y_i \approx \tilde{f}(x_i) = \text{softmax}(W_3\sigma(W_2\sigma(W_1 x_i + b_1) + b_2) + b_3)$$

Activation functions σ introduce non-linearity into the above predictor $\tilde{f}$. Without a non-linear activation function, as the entire interior of the network consists of linear operations (matrix

multiplication and addition), the network would only be capable of modeling linear functions (with a softmax on top). With a properly chosen activation function, such a sigmoid$(x) = 1/(1 + e^{-x})$, $\tanh(x) = 2\text{sigmoid}(2x) - 1$, or $\text{relu}(x) = \max(x, 0)$, neural networks similar to $\tilde{f}$ can estimate a large class of functions and relationships. Each of these activation functions are applied to an input vector coordinate-wise.

With the softmax ending, the above network $\tilde{f}$ outputs a 10-dimensional probability distribution, which we hope encodes the probability of each digit. The next challenge, however, is finding the proper parameters $W_1, W_2, W_3, b_1, b_2, b_3$ so that $y_i \approx \tilde{f}(x_i)$.

To find the proper parameters, we define a loss function that quantifies how good the model is. For "classification" tasks, such as classifying which digit an image refers to, a common loss is Cross Entropy Loss. Letting $\tilde{f}(x_i)_{y_i}$ refer to the neural-network predicted probability that x_i refers to digit y_i, the dataset Cross Entropy Loss is defined as:

$$H(\mathcal{D}) = -\sum_{(x_i,y_i)\in\mathcal{D}} \log \tilde{f}(x_i)_{y_i}$$

Note that, if $H(\mathcal{D}) = 0$, then for all dataset elements $\log \tilde{f}(x_i)_{y_i} = 0$, as each $-\log \tilde{f}(x_i)_{y_i}$ is at least 0 (since we can predict a probability of at most 1 and $\log 1 = 0$). As $\log \tilde{f}(x_i)_{y_i} = 0 \implies \tilde{f}(x_i)_{y_i} = 1$, the dataset loss being 0 implies the model is perfect on each entry of the dataset. Similarly, lower loss corresponds to a better model, with the practically unachievable 0 loss corresponding to a perfect model. Loss functions are often defined to be non-negative.

As each operation in our network and loss function is differentiable, we can compute the gradients $\nabla_{W_i} H(\mathcal{D})$ and $\nabla_{b_i} H(\mathcal{D})$ for each parameter of our model. As the gradient corresponds to how a local change to each of the parameters affect the loss, we can attempt to lower the loss by updating each parameter with:

$$W_i := W_i - \eta \nabla_{W_i} H(\mathcal{D})$$
$$b_i := b_i - \eta \nabla_{b_i} H(\mathcal{D})$$

This operation is known as full gradient descent, leveraging the gradients of each parameter with respect to the loss function to update the parameters to minimize the loss function. Gradient descent is not guaranteed to find the loss-minimizing parameters, or in fact even decrease the loss at all, as the gradients serve only as linear-approximations in a potentially small local neighborhood for how parameter changes affect the loss. However, with well chosen learning rate η, gradient descent performs well in practice, usually finding a saddle point or local minima.

A popular alterative to full gradient descent is stochastic gradient descent, which computes the loss and performs a "gradient update" to the parameters using only subsets of the dataset. This introduces noise that aids the gradient descent procedure in avoiding saddle points (places that are not local minima but have gradients have 0) and allows for faster loss computation and thus more frequent updates in the same compute budget.

3.2 Introduction to Autodifferentiation and Machine Learning Libraries

Many challenges in deep learning look similar to the above. Although the activation functions may differ, and the parameterizations of each layer may change (i.e., not just $f : x \to \sigma(Wx + b)$), many

deep learning networks and problems look similar as above. We have a predetermined loss function to measure performance by, a dataset that illustrates the relationship, and a parameterized network that we wish to "train", or find the proper parameters for, often using an algorithm based in gradient descent.

A practical challenge that emerges is how to effectively compute the gradients for gradient descent. Each computation of $\nabla_{W_i} H(\mathcal{D})$ can be easily computed with computations on the order of the amount of parameters, but this is too slow for the large networks and many optimization steps we perform in practice. The efficient technique used in practice is known as backpropagation. Leveraging the chain rule, we can compute the gradients for each layer in terms of the gradients of the next layer, and thus compute all the gradients effectively by going in order of last layer to first. For example, in our above network, we can compute $\nabla_{W_3} H(\mathcal{D}), \nabla_{b_3} H(\mathcal{D})$, and $\nabla_{v_3} H(\mathcal{D})$ directly (where v_3 is the input to layer 3). Now note that:

$$\frac{dH(\mathcal{D})}{dW_2} = \frac{dH(\mathcal{D})}{dv_3} \cdot \frac{dv_3}{dW_2}$$

We have already computed the first term and $\frac{dv_3}{dW_2}$ is a one-step computation. We can similarly compute $\nabla_{b_2} H(\mathcal{D})$, and $\nabla_{v_2} H(\mathcal{D})$, and perform the same technique for earlier layers. This is the procedure of backpropagation, leveraging the chain-rule to effectively compute the gradient of the output with respect to each input.

Machine Learning libraries, such as JAX, Tensorflow, Keras, and PyTorch (2; 1; 4; 13), aim to simplify the process, allowing users to easily parameterize networks such as $\tilde{f}$, automatically handle gradient accumulation and computation from the loss function, and provide prebuilt optimizers to find the right parameters. These libraries additionally perform low-level compilation for acceleration and are available to users in high-level languages such as Python and JavaScript.

Camelax provides similar support, enabling convenient manipulation of matrices and tensor-functions to define neural networks, automatically computing gradients for each parameter used in computing a loss, and providing prebuilt optimizers to improve the parameters. Inspired by JAX and PyTorch, we maintain a graph of computations as they are performed, allowing us to determine how each parameter affected an end computation and effectively performing backpropagation. We additionally abstract the computation graph from the user, allowing them to perform computations as they would normally, with a computation graph created privately. To build the graph, each tensor maintains references to the tensors involved in its creation. For any tensor, we can recursively trace through its parents to perform gradient descent.

4 LIBRARY

The CAMELAX library has four core modules, each responsible for a different aspect of the machine learning training pipeline.

4.1 TENSOR MODULE

The TENSOR module provides the underlying data structures for the entire library, implementing tensor representations and basic operations for scalars, vectors, and matrices. At its core, a `Tensor.t` is a variant type

```
type t =
  | Scalar of float
  | Vector of float array
  | Matrix of float array array
type shape = int list
```

The underlying array representation is since we need to have fine-grained control in updating weights, which would result in costly overhead if we had to rebuild the entire structure.

Next, we have a variety of creation and accessor utility functions. We enabled creation of matrices and vectors from lists, and implemented initialization of zero and one matrices (just like in numpy). We also implement `map` and `map2` functions for `Tensor.t` types for convenience, and define a variety of basic operations:

```
let map f t =
  match t with
  | Scalar x -> Scalar (f x)
  | Vector v -> Vector (Array.map f v)
  | Matrix m -> Matrix (Array.map (Array.map f) m)

let map2 f t1 t2 =
  assert_same_shape t1 t2;
  match (t1, t2) with
  | Scalar x1, Scalar x2 -> Scalar (f x1 x2)
  | Vector v1, Vector v2 -> Vector (Array.map2 f v1 v2)
  | Matrix m1, Matrix m2 -> Matrix (Array.map2 (Array.map2 f) m1 m2)
  | _ -> raise (Invalid_argument "map2: incompatible tensor types")

let add t1 t2 = map2 ( +. ) t1 t2
let sub t1 t2 = map2 ( -. ) t1 t2
let mul t1 t2 = map2 ( *. ) t1 t2
let div t1 t2 = map2 ( /. ) t1 t2
(* Other operations in full code *)
```

We also implement linear algebra operations like `dot` and `matmul`, and functions for printing and debugging `Tensor.t` types.

4.2 AUTOGRAD MODULE

The AUTOGRAD module implements reverse-mode automatic differentiation through a computational graph with a limited set of primitive functions. This is the core mathematical component of the library.

```
type primitive =
  | Add | Sub | Mul | Div | Exp | Log
  | ReLU | MatMul | AddDistribute

type t = {
  mutable data : Tensor.t; (* tensor values *)
  mutable grad : Tensor.t option; (* gradient *)
  op : primitive option; (* node operation, none for leaf nodes *)
  parents : t list; (* parent nodes *)
  id : int; (* computational graph id *)
}
```

An `Autograd.t` represents a computational graph node, stored as a record:

- `data` represents the tensor value computed during the forward pass through this node.
- `grad` represents the gradient (initially `None`) computed through the backwards pass.
- `op` is the primitive operation that takes place at this node.
- `parents` are pointers to parent (input) nodes, forming the edges of the graph.

- `id` is a unique identifier for this node.

The `primitive` variant type lists all the differentiable operations the system supports—we can later compose these operations to make more complex functions.

Nodes in the computational graph are created with the following two functions, depending on their purpose in the network:

```
(* Leaf node: inputs and trainable parameters *)
let make data =
  { data; grad = None; op = None; parents = []; id = next_id () }

(* Operation node: result of a computation *)
let make_op op parents data =
  { data; grad = None; op = Some op; parents; id = next_id () }
```

The `next_id` function increments a mutable reference for global unique ID generation.

Before computing gradients, we have to determine the order of propagation through the graph (which will be a directed acyclic graph). We do this via a topological sort, implemented with DFS:

```
let topological_sort t =
  let visited = Hashtbl.create 16 in
  let result = ref [] in
  let rec visit node =
    if not (Hashtbl.mem visited node.id) then (
      Hashtbl.add visited node.id true;
      List.iter visit node.parents;
      result := node :: !result)
  in
  visit t;
  List.rev !result
```

Then, the `accumulate_grad` and `backward_primitive` functions implement chain rule and gradient rules for each primitive operation, with `backward` running a full pass through the network:

```
let backward_primitive node upstream =
  match (node.op, node.parents) with
  | None, _ -> ()
  | Some Add, [ x; y ] ->
      accumulate_grad x upstream;
      accumulate_grad y upstream
  | Some Sub, [ x; y ] ->
      accumulate_grad x upstream;
      accumulate_grad y (Tensor.neg upstream)
  | Some Mul, [ x; y ] ->
      accumulate_grad x (Tensor.mul upstream y.data);
      accumulate_grad y (Tensor.mul upstream x.data)
  (* ... *)

let backward expr =
  expr.grad <- Some (Tensor.ones_like expr.data);
  topological_sort expr |> List.rev
  |> List.iter (fun node ->
         match node.grad with
         | None -> ()
         | Some g -> backward_primitive node g)
```

We also include a variety of debugging and print operations, to visualize the graph. Furthermore, we use the `Marshal` module to save and load weights as raw binary files (12):

```
let save_weights filename (params : t array) =
  let data = Array.map (fun p -> p.data) params in
  Out_channel.with_open_bin filename (fun oc -> Marshal.to_channel oc data [])

let load_weights filename (params : t array) =
  In_channel.with_open_bin filename (fun ic ->
      let data : Tensor.t array = Marshal.from_channel ic in
      Array.iteri (fun i p -> p.data <- data.(i)) params)
```

To initialize `Autograd.t` types, the AUTOGRAD module includes mechanisms to initialize from an array, from a tensor, and includes Kaiming-He random initialization (6).

For example, consider the operation $z = (x+y)\cdot y$. We can construct and compute the computational graph as follows (Figure 2):

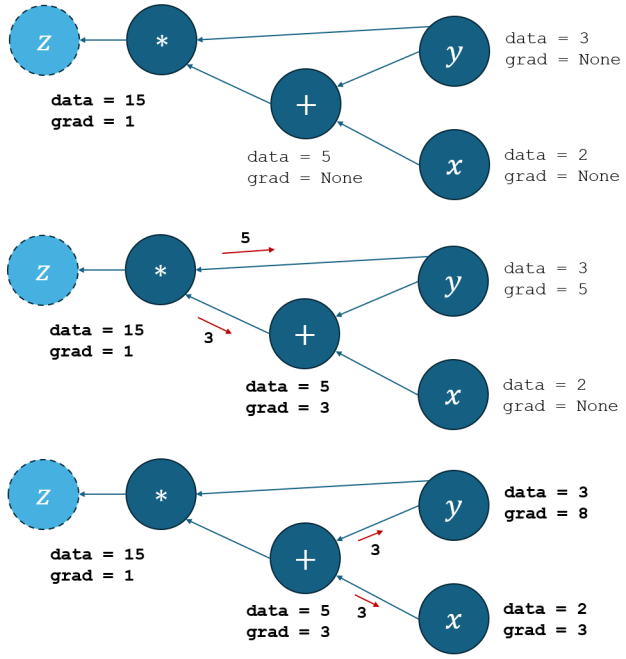

Figure 2: Computational graph for the operation $z = (x + y) \cdot y$, with backpropagation steps.

Testing our implementation, we get the same results:

```
let () =
  let open Camelax in
  print_endline "\nScalar gradient:";
  print_endline "z = (x + y) * y, where x=2, y=3";

  let x = Autograd.scalar 2.0 in
  let y = Autograd.scalar 3.0 in
  let z = Ops.(mul (add x y) y) in

  Printf.printf "z = %g (expected: 15)\n" (Tensor.get z.data []);

  Autograd.backward z;
```

```
  Printf.printf "dz/dx = %g (expected: 3)\n"
    (Tensor.get (Autograd.grad_exn x) []);
  Printf.printf "dz/dy = %g (expected: 8)\n"
    (Tensor.get (Autograd.grad_exn y) []);

  Autograd.print_graph z
```

Output:

```
Scalar gradient:
z = (x + y) * y, where x=2, y=3
z = 15 (expected: 15)
dz/dx = 3 (expected: 3)
dz/dy = 8 (expected: 8)
Computational Graph:
  [1] Leaf <- []: data=Scalar(2.), grad=Scalar(3.)
  [2] Leaf <- []: data=Scalar(3.), grad=Scalar(8.)
  [3] Add <- [1; 2]: data=Scalar(5.), grad=Scalar(3.)
  [4] Mul <- [3; 2]: data=Scalar(15.), grad=Scalar(1.)
```

4.3 OPS MODULE

The OPS module provides a simple user-facing interface for creating `Autograd.t` nodes (and composing the computational graph) automatically:

```
open Autograd

let add x y = make_op Add [ x; y ] (Tensor.add x.data y.data)
let sub x y = make_op Sub [ x; y ] (Tensor.sub x.data y.data)
let mul x y = make_op Mul [ x; y ] (Tensor.mul x.data y.data)
let div x y = make_op Div [ x; y ] (Tensor.div x.data y.data)
let exp x = make_op Exp [ x ] (Tensor.exp x.data)
let log x = make_op Log [ x ] (Tensor.log x.data)
let matmul x y = make_op MatMul [ x; y ] (Tensor.matmul x.data y.data)
let relu x = make_op ReLU [ x ] (Tensor.relu x.data)
let add_distribute x =
  make_op AddDistribute [ x ] (Tensor.add_distribute x.data)
(* ... *)
```

More complex functions can be written by composing basic functions, like `softmax` here:

```
let softmax x =
  let exp_x = exp x in
  let sum_exp_x = add_distribute exp_x in
  div exp_x sum_exp_x
```

4.4 TRAINING MODULE

The TRAINING module builds on AUTOGRAD and initiates parameters while defining the inference function which uses those parameters. The array and inference function both maintain references to the underlying parameters, and thus we can run the model with the inference function, and update the parameters using the parameter array. Below we include an example of a Categorical MLP similar the model we discuss in Section 3.1. The only distinction is that the amount of layers and the layer sizes are determined by the `hidden_sizes` input.

```
let categorical_mlp (hidden_sizes : int list) =
  let mul_layers = mul_layer_init hidden_sizes 0 in (*helper function*)
  let add_layers = add_layers_init hidden_sizes true in (*helper function*)
  let inference (v : Autograd.t) =
    let rec f_aux mul_layers_rem add_layers_rem vec =
      match (mul_layers_rem, add_layers_rem) with
      | mul_first :: [], add_first :: [] ->
          Ops.(softmax (add (matmul mul_first vec) add_first))
      | mul_first :: mul_layers_rem, add_first :: add_layers_rem ->
          f_aux mul_layers_rem add_layers_rem
            Ops.(tanh (add (matmul mul_first vec) add_first))
      | _ -> vec
    in
    f_aux mul_layers add_layers v
  in
  (inference, Array.of_list (mul_layers @ add_layers))
```

Other models we implement include a ReLU-based MLP for regression tasks and an autoencoder, which returns an inference function for training, encoder function, and decoder function, along with parameters.

We additionally implement a variety of loss functions and gradient updates. The below gradient descent function performs $p := p - \eta \nabla_p L$ as discussed above.

```
let gd_optimizer (params : Autograd.t array) (lr : float) =
  let n = Array.length params in
  let rec aux x =
    match x with
    | -1 -> ()
    | x ->
        (match params.(x).grad with
        | None -> failwith ("gradient is None for param " ^ string_of_int x)
        | Some g ->
            let update_nudge = Tensor.map (fun x -> lr *. x) g in
            params.(x).data <- Tensor.sub params.(x).data update_nudge;
            Autograd.zero_grad params.(x));
        aux (x - 1)
  in
  aux (n - 1)
```

We additionally implement mean squared error loss for scalars and vectors, and we implement cross entropy loss.

4.5 Data Loading

While not a part of the core library, we wrote utilities for loading data from the MNIST digit recognition dataset. The `mnist.py` script converts raw MNIST binary files into text versions that we can more easily parse in OCaml, like this:

```
type dataset = {
  images : Tensor.t list; (* 784 pixels each, 0-1 *)
  labels : int list;
  num_images : int;
}

let parse_floats sep s = String.split_on_char sep s |> List.map float_of_string
```

```
let normalize img = Tensor.map (fun p -> p /. 255.0) img

let load_images filename =
  let lines = In_channel.with_open_text filename In_channel.input_lines in
  let images =
    List.map
      (fun line -> parse_floats ' ' line |> Tensor.vector_of_list |> normalize)
      lines
  in
  (images, List.length images)

let load_labels filename =
  In_channel.with_open_text filename In_channel.input_lines
  |> List.map int_of_string

let load_dataset ~images_file ~labels_file =
  let images, num_images = load_images images_file in
  let labels = load_labels labels_file in
  { images; labels; num_images }
```

4.6 CAMELAX EXAMPLE

In this example, we'll consider the task from Section 3.1 on learning to recognize digits from 28×28 images. We leverage the MNIST (8) dataset as mentioned before, and train for 5 epochs on the $60,000$ examples. We then evaluate on a held-out set of $10,000$ samples. We use brackets to omit verbose sections of code. As before, full code is available in the library.

```
  [ ... ] (* loads data, defines arrays for images and labels*)
  let f, parameters = Training.categorical_mlp [ 784; 100; 100; 100; 10 ] in

  let step _ = Training.gd_optimizer parameters 0.0001 in

  let train_mnist samples =
    let rec train_mul_aux steps ewma =
      match steps = samples with
      | true -> ()
      | false ->
          let image = images.(steps) in
          let target = labels.(steps) in
          let pred = f (Autograd.make image) in
          let loss = Training.cross_entropy_loss pred target 10 in
          Autograd.backward loss;
          if steps mod 1 = 0 then step ();
          if steps mod 1000 = 0 then
            Printf.printf "steps: %d, loss ewma = %f\n%!" steps ewma;

          train_mul_aux (steps + 1)
            ((0.001 *. Tensor.get loss.data [ 0 ]) +. (0.999 *. ewma))
    in
    train_mul_aux 0 10.0
  in
  train_mnist (60000 * 5);
  [...] (* more data loading for evaluation set*)
  let eval n =[...] (* evaluates on training set *)
```

```
  in
  let res = List.fold_left ( + ) 0 (eval 10000) in
  Printf.printf "We predict %d correctly\n" res;
```

Out of $10,000$ held out samples, we predict $9,324$ correctly, giving our model a 93.24% accuracy. As will show in our experimental section, we additional train models for multiplication, basic classification tasks, the MNIST classification task, and an autoencoder model.

4.7 Graphics

As a fun extension, we wanted to test out some of our MNIST models interactively, which was possible with the functionality of saving / loading weights. To do this, we used the `Graphics` library in OCaml, which provides a set of graphics and drawing primitives (9; 11).

The program works by running a main event loop, which polls for any key or mouse actions, and handles them accordingly, like drawing pixels on a 28×28 grid. Every update, this grid is flattened and fed into the loaded model, and the predictions and their confidence values are displayed on the right. Below is the main event loop, and Figure 3 shows what the GUI looks like.

Figure 3: Pure OCaml MNIST GUI. In this case, the model thinks we are drawing a 2.

```
let () =
  (* Load model and set up canvas *)
  let model, params = Training.categorical_mlp [ 784; 100; 100; 100; 10 ] in
  Autograd.load_weights "mnist_catmlp.weights" params;

  let window_width = window_size + 200 in
  let window_height = window_size + 80 in
  open_graph (Printf.sprintf " %dx%d" window_width window_height);
  set_window_title "OCaml MNIST Model";
  auto_synchronize false;

  (* Main event loop *)
  let running = ref true in
```

```
let is_drawing = ref false in
let last_pred = ref (0, Array.make 10 0.0) in

while !running do
  (if key_pressed () then
     match read_key () with
     | 'q' -> running := false
     | 'c' ->
         clear_drawing ();
         last_pred := (0, Array.make 10 0.0)
     | _ -> ());

  let st = wait_next_event [ Poll; Button_down; Button_up; Mouse_motion ] in

  if st.button then is_drawing := true
  else if st.keypressed = false && not st.button then is_drawing := false;

  (* Update drawing and predictions *)
  if !is_drawing && st.mouse_x < window_size && st.mouse_y < window_size then (
    paint st.mouse_x st.mouse_y;
    last_pred := predict model);

  clear_graph ();
  draw_grid ();
  let label, probs = !last_pred in
  draw_predictions label probs;

  set_color black;
  write_at
    (10, window_size + 20)
    "Click and drag to draw | 'c' to clear | 'q' to quit";
  synchronize ()
done;

close_graph ()
```

5 Experiments

5.1 Multiplication

We train a model on the task of predicting $x \cdot y$ for $x, y \in \mathbb{R}$ with an MLP with ReLU activations. Note that, as the MLP parameterization only constitutes linear operations, and the ReLU operation is piecewise linear, there is no way for the ReLU-MLP to exactly express multiplication. Thus this constitutes a non-trivial challenge, and we contend it would be challenging for a person. We will consider two different models, a small model with a single hidden layer of dimension 5 and a large model with two hidden layers, each of dimension 100. Each layer will have a ReLU, with the exception of the output layer. We train with $200,000$ randomly chosen values of x, y with x, y being floats between 0.0 and 3.0. We report the exponentially weighted moving average of the loss, with $\alpha = 0.01$, in Figure 4. Note that the below plots are of the EWMA loss and begin at step 1000. We additionally include example predictions in the Appendix. The small model trained nearly instantly, while the large model was approximately 48 seconds.

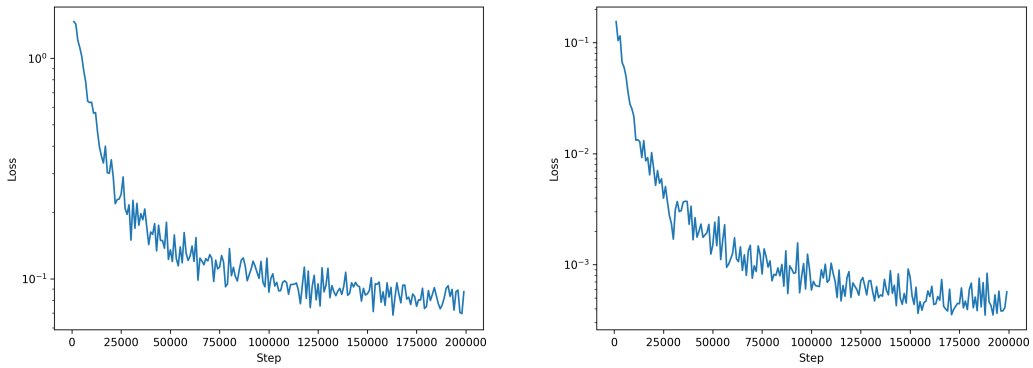

Figure 4: EWMA ($\alpha = 0.01$) Training Loss of the Small (Left) and Large (Right) multiplication models. Note that, while the loss curves are similar in appearance, the larger model reaches significantly lower training loss.

5.2 Circle Prediction

We train a model on the task of predicting, for $(x, y) \in [-2, 2] \times [-2, 2]$, whether the point is within the circle of radius 1.4. We train with a categorical MLP with two hidden layers, each of dimension 100, and we train with $200,000$ randomly chosen coordinate pairs and cross-entropy loss. We report the EWMA of the loss with $\alpha = 0.01$. The only difference between this setup and the setup with the large multiplication model is that the model now leverages `tanh` activation and a `softmax` final activation, which selects between 0 (in circle) and 1 (outside circle). In Figure 5, we plot the model's predictions for $1,000$ samples and the training loss.

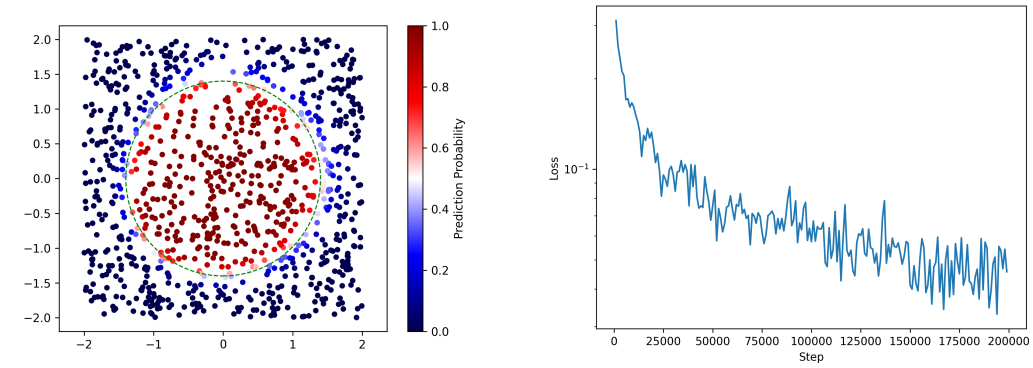

Figure 5: Example predictions of the trained model (left) and EWMA ($\alpha = 0.01$) Training Loss (right). Note that, near the radius, the model is less confident, and further from the radius, the model is more confident.

5.3 MNIST

We now train a categorical model on the MNIST task. MNIST contains $60,000$ example of image and label pairs in the training set, and $10,000$ examples for the test set. We train for 5 epochs (i.e., running through each example $5\times$) for a total of $300,000$ steps. The categorical MLP we leverage has 3 intermediate layers, each of size 100, and leverages `tanh` intermediate activations with a final `softmax` activation to produce a probability distribution over the 10 possible labels. We train with cross-entropy loss. Training may take between 0.5 and $3+$ hours depending on hardware. Below we plot the EWMA loss with $\alpha = 0.001$. We found from earlier runs on a smaller eval set that 1 epoch

enables $\approx 83\%$ accuracy and 3 epochs enables 91% accuracy. In total, with 5 training epochs and evaluated on the entire $10,000$ size evaluation set, we achieved $\approx 93.24\%$ accuracy.

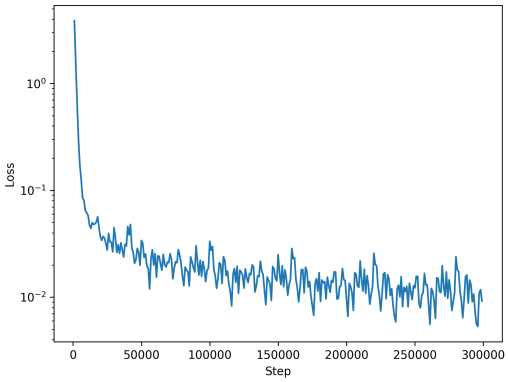

Figure 6: Training loss of Categorical MLP trained on MNIST for 5 epochs. We plot an EWMA of the Loss with $\alpha = 0.001$.

We additionally developed a graphics library to show the predictions of our trained MNIST model. We include a few examples in Figure 7.

Figure 7: Selected subset of interactive prediction examples for the categorical MLP.

5.4 MNIST AUTO-ENCODER

To demonstrate CAMELAX on a more challenging environment, we additionally train an MNIST Autoencoder. This model has 5 hidden layers of dimension 64, 32, 16, 32, and 64, before the final output of dimension 784. Intermediate layers have `tanh` activation and the final head has `sigmoid` activation, as we normalize pixels to between 0 and 1. The model is trained with two losses, one is the reconstruction loss that represents how different the input is the from the output. The second is a classification loss where, from the middle hidden layer with dimension 16, we add an additional separate head that consists of a matrix multiplication and bias before `softmax`. This head predicts the label using cross-entropy loss as before. Since the model has a bottleneck dimension of size 16, reconstructing the image is challenging, and we weight the reconstruction loss so that it has output magnitude 5 to $100\times$ larger than the easier classification loss. We train for 3 epochs on MNIST. The autoencoder can be decomposed into an embedding model from $\mathbb{R}^{784} \to \mathbb{R}^{16}$ and a decoder from $\mathbb{R}^{16} \to \mathbb{R}^{784}$. As a demonstration of the embedding performance, we embed several image pairs with the encoder and interpolate between the embeddings. We then decode the interpolated embeddings to show images at various stages of interpolation between two digit images. We show this result in Figure 8.

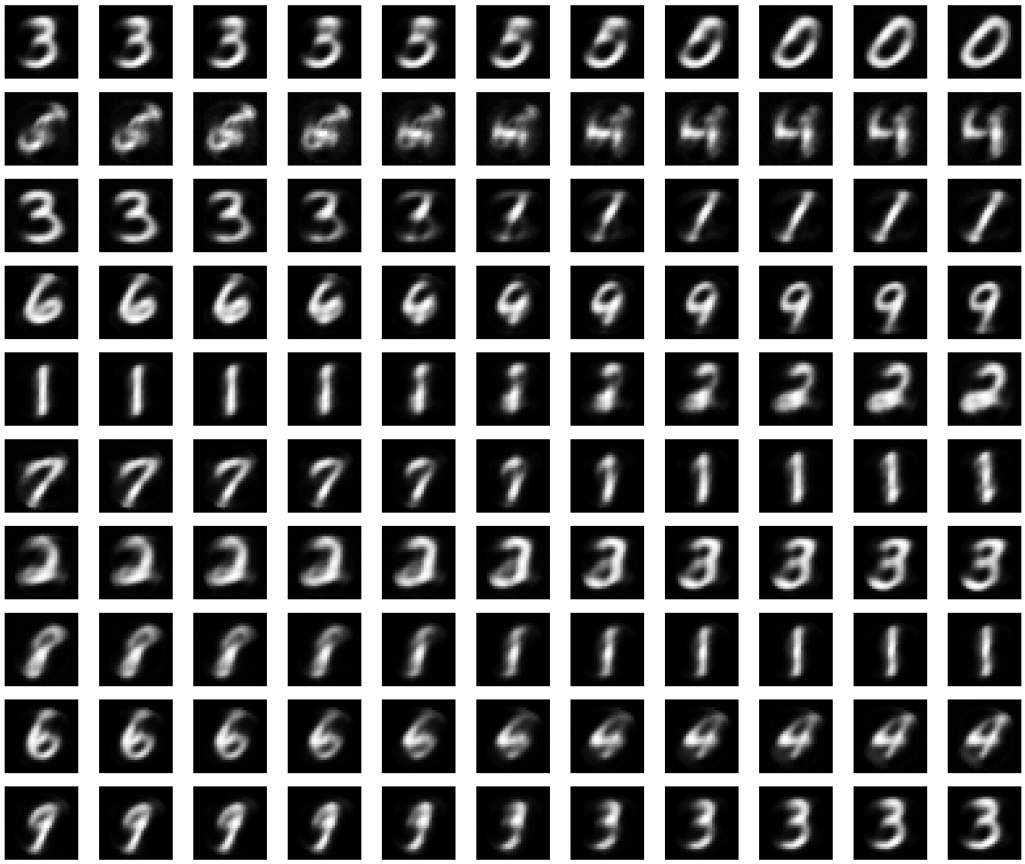

Figure 8: Interpolated images in the MNIST Autoencoder. We embed an image pair, interpolate between the embeddings, with interval ratio 0.1, and then decode with the decoder. For example, for images x_1 and x_2, and letting f be the encoder and g be the decoder so that $g(f(x_1)) \approx x_1$, we plot $g(\alpha f(x_1) + (1-\alpha) f(x_2))$, for $\alpha \in \{0, 0.1, 0.2, \ldots, 1.0\}$.

6 CONCLUSION

We develop CAMELAX, an OCaml based library for autodifferentiation and machine learning. CAMELAX is lightweight and easy-to-use, built on the TENSOR, AUTOGRAD, OPS, and TRAINING modules. TENSOR defines the scalar/vector/matrix type that we operate over, AUTOGRAD maintains the computational graph and stores gradients for tensors used in computation, and OPS provides a convenient function wrapper. TRAINING provides the high-level methods for easily instantiating deep learning layers, running inference, implementing loss functions, and performing gradient updates. The TRAINING library itself is sufficient for many use cases, and it is easy to declare one's own models, optimizers, and loss, if the predefined ones are not sufficient. We additionally develop a graphics companion for visualizing MNIST queries, and model helper functions, such as for saving and loading model weights.

As part of our demonstration we train machine learning models for multiplication, point-in-circle classification, MNIST digit classification, and we train an autoencoder model, allowing us to embed images and interpolate between images in the embedding space. For future work, we hope to parallelize elements of model training, allowing CAMELAX to take advantage of multi-threading, and eventually we would like to add GPU support.

REFERENCES

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (2016), pp. 265–283.

[2] BRADBURY, J., FROSTIG, R., HAWKINS, P., JOHNSON, M. J., LEARY, C., MACLAURIN, D., NECULA, G., PASZKE, A., VANDERPLAS, J., WANDERMAN-MILNE, S., AND ZHANG, Q. JAX: composable transformations of Python+NumPy programs, 2018.

[3] BROWN, T. B., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., AGARWAL, S., HERBERT-VOSS, A., KRUEGER, G., HENIGHAN, T., CHILD, R., RAMESH, A., ZIEGLER, D. M., WU, J., WINTER, C., HESSE, C., CHEN, M., SIGLER, E., LITWIN, M., GRAY, S., CHESS, B., CLARK, J., BERNER, C., MCCANDLISH, S., RADFORD, A., SUTSKEVER, I., AND AMODEI, D. Language models are few-shot learners, 2020.

[4] CHOLLET, F., ET AL. Keras. `https://keras.io`, 2015.

[5] DYRO, ROBERT, AND GUO, YUFENG. Jax in action. `https://io.google/2025/explore/technical-session-1`, 2025. Google I/O 2025 Technical Session. Accessed: 2025-11-18.

[6] HE, K., ZHANG, X., REN, S., AND SUN, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (2015), pp. 1026–1034.

[7] KANDPAL, N., LESTER, B., RAFFEL, C., MAJSTOROVIC, S., BIDERMAN, S., ABBASI, B., SOLDAINI, L., SHIPPOLE, E., COOPER, A. F., SKOWRON, A., KIRCHENBAUER, J., LONGPRE, S., SUTAWIKA, L., ALBALAK, A., XU, Z., PENEDO, G., ALLAL, L. B., BAKOUCH, E., PRESSMAN, J. D., FAN, H., STANDER, D., SONG, G., GOKASLAN, A., GOLDSTEIN, T., BARTOLDSON, B. R., KAILKHURA, B., AND MURRAY, T. The common pile v0.1: An 8tb dataset of public domain and openly licensed text, 2025.

[8] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (2002), 2278–2324.

[9] LEROY, X., FURUSE, J., GEFFROY, J.-M., NAVIA, J., AND WEIS, P. graphics – the OCaml graphics library. `https://opam.ocaml.org/packages/graphics/`, 2025. Version 5.2.0, opam package repository.

[10] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[11] OCAML DEVELOPMENT TEAM. Module graphics – OCaml standard library. `https://ocaml.org/manual/4.01/libref/Graphics.html`, 2014. Accessed: 2025.

[12] OCAML DEVELOPMENT TEAM. Module marshal – OCaml standard library. `https://ocaml.org/manual/5.4/api/Marshal.html`, 2025. Accessed: 2025.

[13] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., ET AL. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32* (2019).

[14] RAJPURKAR, P., IRVIN, J., ZHU, K., YANG, B., MEHTA, H., DUAN, T., DING, D., BAGUL, A., LANGLOTZ, C., SHPANSKAYA, K., LUNGREN, M. P., AND NG, A. Y. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning, 2017.

[15] RUTHERFORD, A., ELLIS, B., GALLICI, M., COOK, J., LUPU, A., INGVARSSON, G., WILLI, T., HAMMOND, R., KHAN, A., DE WITT, C. S., SOULY, A., BANDYOPADHYAY, S., SAMVELYAN, M., JIANG, M., LANGE, R. T., WHITESON, S., LACERDA, B., HAWES, N., ROCKTASCHEL, T., LU, C., AND FOERSTER, J. N. Jaxmarl: Multi-agent rl environments and algorithms in jax, 2024.

[16] SUTTON, R. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, Mar. 2019. Accessed: 2025-11-18.

[17] SUVANJANPRASAI. MNIST dataset example. Wikimedia Commons, 2020. Accessed: 2025.

[18] WU, C., LI, J., ZHOU, J., LIN, J., GAO, K., YAN, K., MING YIN, S., BAI, S., XU, X., CHEN, Y., CHEN, Y., TANG, Z., ZHANG, Z., WANG, Z., YANG, A., YU, B., CHENG, C., LIU, D., LI, D., ZHANG, H., MENG, H., WEI, H., NI, J., CHEN, K., CAO, K., PENG, L., QU, L., WU, M., WANG, P., YU, S., WEN, T., FENG, W., XU, X., WANG, Y., ZHANG, Y., ZHU, Y., WU, Y., CAI, Y., AND LIU, Z. Qwen-image technical report, 2025.

[19] XAI. Colossus. https://x.ai/colossus, 2025. Accessed: 2025-11-18.

7 APPENDIX

Here are example predictions for the small multiplication model:

```
2.283871 * 0.733408 = 1.675010, we predict 1.810796
1.860955 * 2.566095 = 4.775388, we predict 4.723838
0.294129 * 2.353929 = 0.692358, we predict 0.651905
0.080291 * 2.875398 = 0.230868, we predict 0.661163
2.807352 * 2.781336 = 7.808189, we predict 7.260179
0.320686 * 1.667847 = 0.534856, we predict 0.250611
2.686752 * 1.053322 = 2.830015, we predict 2.934259
0.679114 * 1.484547 = 1.008176, we predict 0.706524
1.220356 * 0.167603 = 0.204535, we predict -0.493337
1.719364 * 1.392808 = 2.394743, we predict 2.944006
```

Here are example predictions for the large multiplication model:

```
1.472774 * 0.523668 = 0.771245, we predict 0.779520
0.806431 * 2.311450 = 1.864026, we predict 1.847857
2.438863 * 1.129824 = 2.755486, we predict 2.759581
2.673253 * 0.229704 = 0.614056, we predict 0.605340
1.453912 * 1.024902 = 1.490118, we predict 1.513925
2.553687 * 1.163011 = 2.969967, we predict 2.964556
0.363610 * 1.740817 = 0.632978, we predict 0.655226
0.363670 * 0.210134 = 0.076420, we predict 0.089488
1.418888 * 2.030764 = 2.881428, we predict 2.858227
0.917791 * 0.913838 = 0.838712, we predict 0.816926
```