Mini Deep-learning framework
Deep Learning

Karel Peeters, Lavinia Schlyter
Department of Computer Science, EPF Lausanne, Switzerland

***Abstract*—Deep learning frameworks offer building block for constructing deep neural networks to ease the burden on the coder. In this framework we implement a `Linear` layer, 4 activation functions `Sigmoid`, `(Leaky)Relu` and `Tanh`. Two losses, MSE or BCE that can be optimized by either `SGD` or `Adam`. By keeping track of which operation produced a given *HyperCube*, we implement an automatic differentiation feature.**

I. Introduction

In order to set up a deep neural networks, we use deep learning frameworks which offer the building blocks for designing, training and validating a network. The most widely used frameworks are such as MXNet, PyTorch, TensorFlow, Keras and others. In order to better understand the magic beneath the hood, we will design our own framework using Pytorch tensor operations without using the build-in autograd. It must correctly classify points that lay either inside or outside a disk of radius $\frac{1}{\sqrt{2 \cdot \pi}}$ centered at $(0.5, 0.5)$, using a network with two input units, one output unit and three hidden layers of 25 units.

II. Structure

A. *Autograd*

To compute the gradients Pytorch [1] uses *Automatic Differentiation*, which means that the end user can write simple python code using standard tensor operations, and the framework can automatically compute the correct gradient during backpropagation for all tensors used in the computation. This has a few big advantages over simpler solutions:

- Users can write any Python code they want, including loops or branching, as long as the tensors themselves are only manipulated using the supported operations.
- The user can't make any mistakes during gradient calculation, since that is handled by the framework.
- When the forward running code is changed the user doesn't need to change the corresponding backward code since there is none.

In this project we write our own implementation of this system and try to make it as user-friendly as possible.

B. *HyperCube*

We implement our own wrapper around the Pytorch `Tensor` class, which is called `HyperCube` to differentiate it. We only use the inner `Tensor` for storage and for operations, not for automatic differentiation, which we disable using `torch.set_grad_enabled(False)`.

```
class HyperCube:
    def __init__(self, value, grad_fn=None):
        assert isinstance(value, torch.Tensor)
        self.value = value
        self.grad = zeros_like(value)
        self.grad_fn = grad_fn
```

The `HyperCube` class has 3 fields:

- `value` the the PyTorch Tensor that contains the actual data stored in this `HyperCube`. With the first dimension being the batch size.
- `grad` is another Tensor that stores the gradient associated with this HyperCube. It is accumulated during backpropagation.
- `grad_fn` is an instance of a `GradFn` class that remembers which operation created this tensor if any.

The `backward` method is used to compute the gradients in the computational graph of the `HyperCube` and accumulate them.

```
def backward(self, output_grad=None):
    if output_grad is None:
        output_grad = ones_like(self.value)
    if self.grad_fn is not None:
        self.grad_fn.backward(output_grad)
    self.grad += output_grad
```

We first verify that `output_grad` is `None`; if True then we are at the first `backward()` call on the loss. Then we verify whether `self.grad_fn` is `None`; if False the `output_grad` (loss with respect to the output) is passed onto `self.grad_fn.backward()` in order to recurse back in the graph and calculate the loss with respect to the parameters; see eq. (1). Indeed, our goal is to understand how sensitive the cost function is to each parameter in order to make the "most efficient" adjustments for decreasing the loss. The gradient is then accumulated; this is necessary for weight sharing as the same parameter may appear multiple times in the operation graph .

$$\frac{\delta L}{\delta \kappa} = \frac{\delta L}{\delta z} \cdot \frac{\delta z}{\delta \kappa} \tag{1}$$

where L stands for the loss, κ for the parameters z for the output and where $\frac{\delta L}{\delta z} =$ `output_grad`.

C. *Module*

We introduced an abstract base class `Module` with methods `__call__` and `param` which the different layers classes like `Linear`, `Tanh`, `Sigmoid`, `Relu` extend. All layers have a corresponding `GradFn` class that is responsible for backpropagating the gradients.

1) `Sequential`: To combine the different layers in order to form an architecture, we implement a Sequential class that will apply different layers onto the input. The `param` method returns a concatenation of the parameters of the submodules inside the `Sequential`.

2) `Linear Layer`: The `__call__` (forward) method computes

$$\mathbf{z} = W * \mathbf{x} + \mathbf{b} \tag{2}$$

and remembers the information it will later need for backpropagation in an instance of the `LinearGradFn` class as part of the returned `HyperCube`. In this case that is the input, weight as bias. The weight and bias are initially randomly initialized using the same distributions used by PyTorch. The `param` method returns a list containing the weight and bias `HyperCubes`.

The following code snippet shows the `backward` method for `LinearGradFn` class.

```
def backward(self, output_grad):
    self.b.backward(output_grad.sum(dim=0))
    self.W.backward((self.input_.value[:,:,None]
    ...@output_grad[:,None,:]).sum(dim=0))
    input_grad = output_grad @ self.W.value.T
    self.input_.backward(input_grad)
```

The output of the layer is **z**; if we compute as defined in eq (2), the output with respect to the parameters (in this case `b` and `W`), we get eq (3):

$$\frac{\delta L}{\delta b} = \frac{\delta L}{\delta \Omega} \cdot 1 \qquad \frac{\delta L}{\delta W} = \frac{\delta L}{\delta \Omega} \cdot \mathbf{x} \tag{3}$$

$$\frac{\delta L}{\delta \mathbf{x}} = \frac{\delta L}{\delta \Omega} \cdot W^T \tag{4}$$

We continue propagate these gradients to b, W and x by calling `backward` on them. For b and W the gradient will be accumulated, the x gradient will continue recursing until the input tensor is reached.

Activation functions are used in neural networks in order to help the network learn complex patterns by adding non-linearity into the network. Activation functions are modules which do not contain parameters and thus `param` returns an empty list.

3) `(Leaky)Relu`: Relu stands for rectified linear unit and is defined as:

$$z = max(0, x) \tag{5}$$

as can be seen from the eq (5), it is only sparsely activated (zero for all negative inputs). Though sparsity can be good to actually process meaningful data [2], enable faster convergences and reduce overfitting, we may however encounter the "dying ReLU" problem. That is, when the layer will always output zero, in order to mitigate this possible issue, a variant named Leaky Relu was introduced and instead of being zero for negative values it has a small negative slope as can be seen from eq (6).

$$z = max(-\alpha x, x) \text{ with } 0 \leq \alpha < 1 \tag{6}$$

4) `Sigmoid`: The sigmoid function maps any value to a range between $[0, 1]$. It is used in classification where values > 0.5 or ≥ 0.5 will be mapped as 1 else 0. It is defined by eq. (7) and its derivative used in `.backward()` by eq. (8):

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{8}$$

5) `Tanh`: The hyperbolic tangent is similar to sigmoid but ranges from $[-1, 1]$ instead of $[0, 1]$.

$$tanh(x) = 2 \cdot \sigma(2 \cdot x) - 1 \tag{9}$$

$$tanh(x)' = 1 - tanh(x)^2 \tag{10}$$

D. *Loss/Cost functions*

Loss function are used in order to measure the performance of the model.

1) `Mean Squared Error` (MSE): In the forward pass, we compute:

$$L_{MSE} = \frac{1}{2 \cdot N} \sum_i (\mathbf{x} - \mathbf{y})^2 \tag{11}$$

where **y** is the target. The `input` and `target` are stored in a `LossMSEGradFn` instance. `backward` is then computed

```
def backward(self, output_grad):
    batch_size=len(self.input_.value)
    input_grad=(self.input_.value-self.target.value)
    ...* output_grad/batch_size
    self.input_.backward(input_grad)
```

where `input_grad` is the gradient of the loss w.r.t input. This corresponds to computing the derivative of eq (11) w.r.t **x**.

2) `Binary Cross Entropy` (BCE):

$$L_{BCE} = -\frac{1}{N} \sum_i y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \tag{12}$$

The loss is returned as a `HyperCube` with its `grad_fn=LossBCEGradFN(input_,target)`

Note that if x_n is either 0 or 1, one of the log terms would be mathematically undefined; using Pytorch's solution to deal with this problem, we clamped the log functions output to be greater than or equal to -100; the loss will thus have a finite loss value.

To compute the `GradFN`:

$$\frac{\delta L}{\delta \mathbf{x}} = \frac{\mathbf{x} - \mathbf{y}}{max((1 - \mathbf{x}) \cdot \mathbf{x}, \epsilon)} \tag{13}$$

where **x** is the input and $\epsilon = 1e - 12$ to avoid division by zero.

E. *Optimizer*

The optimizers goal is to find the parameters θ of a neural network that reduces the cost function J(θ).

All optimizers inherit from the base `Optimizer` class shown below.

```
class Optimizer:
    def __init__(self, params):
        self.params = params
    def zero_grad(self):
        for param in self.params:
            param.zero_grad()
```

The `zero_grad` method is used in order to set the gradient of all parameters to zero so the following `backward` call can start accumulating them again.

*1) **Stochastic Gradient Descent [SGD]:*** Central in many machine learning applications, SGD updates the parameters in the opposite direction of the gradient w.r.t to the parameters (θ) of the objective function J(θ). The learning rate determines the magnitude of the step in the given direction. We also introduced a weight decay parameter λ to implement the L2-norm penalty, used for the updating step:

```
def step(self):
    for param in self.params:
        param.value -= self.lr*(param.grad+
        ...self.lambda_ * param.value)
```

*2) **Adam:*** The learning rate being one of the most tedious hyperparameter to tune due to the significant impact it has on performance. Research has lead to the introduction of adaptive learning rates.

Adam, introduced in [3], stands for "adaptive moments", and computes adaptive learning rates for the parameters from estimates of first moment (the mean) and second moment (uncentered variance) of the gradient.

The updating step may be found in the code or in the paper [3]. The momentum is an estimate of the first-order moment with exponential weighting of the gradient. That is, it uses an exponentially decaying to discard information from long ago to more not diverge.

F. Concatenation and slicing

The (`[]`) operator was overloaded for slicing our `HyperCube`; this operation has a corresponding `SliceGradFn`, which propagates the output gradient only to the part of the tensor that was returned by the slicing operator.

For concatenation, we made a `cat` method in `HyperCube`, which combines two `HyperCubes` and sets the `grad_fn` to `CatGradFn` whose `backward` method is:

```
def backward(self, output_grad):
    in_grad_1=output_grad[:,:self.input1.shape[1]]
    in_grad_2=output_grad[:,self.input1.shape[1]:]
    self.input1.backward(in_grad_1)
    self.input2.backward(in_grad_2)
```

These two operations together are enough to implement simple weight sharing networks.

III. Testing the framework

In order to verify the implementations, we tested an architecture for 250 epochs which combines the different layers: Two input units, one output unit, three `Linear` hidden layers of 25 units, `Tanh` as the first activation function, `Relu` for the next two layers and a `Sigmoid` for the output (*Net_1*).

The results for various set-ups are given in Table I.

	Train Error $\pm$ std	Test Error $\pm$ std	lr
MSE_SGD	0.0317 ± 0.0137	0.0352 ± 0.0114	0.5
BCE_SGD	0.0642 ± 0.0177	0.0856 ± 0.0188	0.3
MSE_Adam	0.0072 ± 0.0021	0.0157 ± 0.0056	-
BCE_Adam	0.0095 ± 0.0100	0.0167 ± 0.0100	-
WS_MSE_SGD	0.1141 ± 0.0921	0.1172 ± 0.0812	0.5
WS_MSE_Adam	0.0062 ± 0.0036	0.0099 ± 0.0028	-

TABLE I: Experimental results

Using Adam enabled the best accuracy, for the given epochs as it figures out the optimal learning rate automatically. All results converge but weight sharing [`WS`] with `SGD` which is not performing optimally due to its instability.

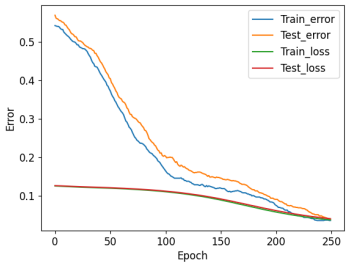

Fig. 1: Training

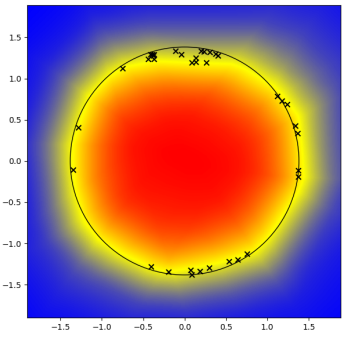

Fig. 2: Output of model

Figure 1 represents the training procedure with MSE loss optimized using SGD with 250 epochs on *Net_1* using `lr`$= 0.5$ and $\lambda = 1e-4$. We note the test error always greater that the training error with little overfitting, and the loss constantly decreasing. Figure 2 represents the output of the trained network after the final epoch. The color scale used represents 0 as blue, 0.5 as yellow and 1 as red. The crosses represent the missclassified points and the black circle corresponds to the boundary. The yellow area are points where the network is not confident about its prediction, and just like the misclassified points this is around the boundary. This matches our intuition of where the network is more likely to make mistakes.

IV. Future Improvements

Our mini framework is missing many layers that are important for modern deep networks like dropout, batch normalization, and convolutions.

V. Conclusion

In this project, we implemented a basic deep learning framework that allows the user to built architectures with a `Linear layer`, with several activation functions `(Leaky)Relu`, `Tanh`, `Sigmoid` and to optimize either the `Binary Cross Entropy` loss or `MSE` using `SGD` or `Adam`. It also allows for automatic differentiation, also known as `autograd`.

We conclude by providing a small code snippet in order to see how the presented framework may be used.

```
model = Sequential([Linear(2, 1), Sigmoid()])
loss_func = LossMSE()
optimizer = Adam(model.param())
pred = model(input_)
loss = loss_func(pred, target)
loss.backward()
optimizer.step()
```

VI. *

References

[1] Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[2] Yoshua Bengio. *Learning deep architectures for AI.* Now Publishers Inc, 2009.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.