EE 559- Deep Learning : Mini Project Report

Ankita Humne
ankita.humne@epfl.ch

Elliott Pinel
elliott.pinel@epfl.ch

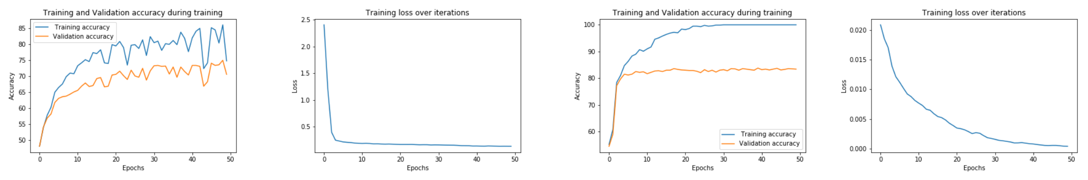
Harshvardhan
harshvardhan.harshvardhan@epfl.ch

May 22, 2020

1 Project 1

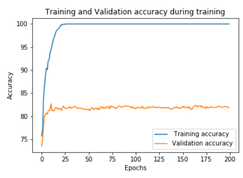
In project 1, we first extract features using the ”base layer”, which is followed by a comparator layer and then a classifier to incorporate auxiliary loss. The best chosen solution contains convolution layers with max pool to detect the numbers, followed by a layer to compare the digits. This solution has weight sharing, batch normalization but no auxiliary loss. However, we get similar results using auxiliary loss and no weight sharing as shown in figure 4. But if we include both auxiliary loss and weight sharing, the results are not as good as shown in figures 4i and 4j. Weight sharing reduces the number of parameters, helping to make the training easier. In our case, the variance between images is less hence, weight sharing fastens the process by incorporating useful inductive priors about the problem. In our case, we share weights by using a base model to generate feature vectors for each image. If the actual label of the image can be predicted accurately, we can also predict the label for each pair of images. Thus, using the true label information and the loss associated with it, we should, in principle improve the model performance. To include this loss, we add an additional layer to predict the label of the image from the generated features. As expected, inclusion of weight sharing and auxiliary loss improve the model's performance, however, when both these methods are used together, we do not obtain the best possible results as they counteract the effects of each other.

Batch normalization is known to reduce the epochs used to train and hence, was added to the network. This is verified in Figure 2. The activation function used is ReLU. We did not find much difference between results obtained with ReLU and LeakyReLU for the optimal case and decided to go with LeakyReLU because sometimes we encountered results as shown in Figure 3. This is because Leaky ReLU eliminates situation when the gradient becomes zero and we have vanishing gradients. With ReLU, we end up with a neural network that never learns if the neurons are not activated at the start, while Leaky ReLU makes us independent of initialization.

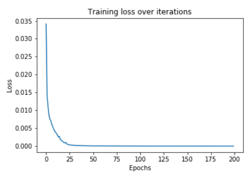


(a) Accuracy - 1 layer (b) Loss - 1 layer (c) Accuracy - 2 layers (d) Loss - 2 layers

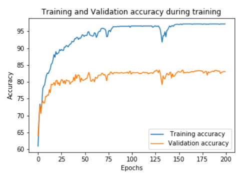
Figure 1: Comparison of layers



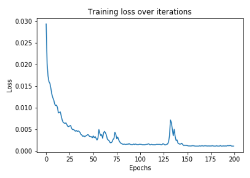
(a) Accuracy - with batch norm



(b) Loss - with batch norm

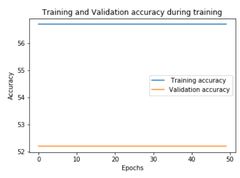


(c) Accuracy - without batch norm

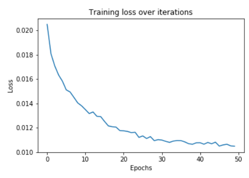


(d) Loss - without batch norm

Figure 2: Effect of batch normalization

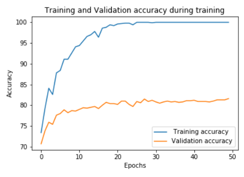


(a) Accuracy

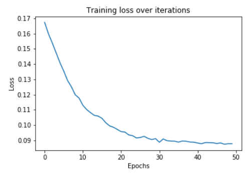


(b) Loss

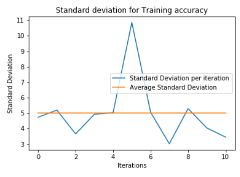
Figure 3: Faulty result with ReLU



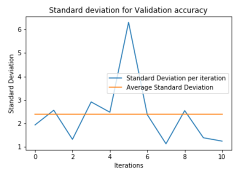
(a) Accuracy - auxiliary loss



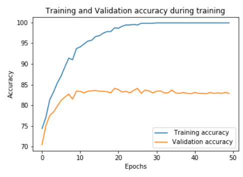
(b) Loss - auxiliary loss



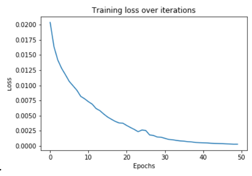
(c) Std dev training - auxiliary loss



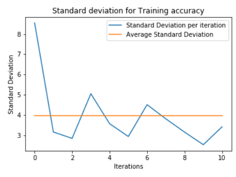
(d) std dev validation - auxiliary loss



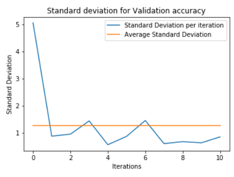
(e) Accuracy - weight sharing



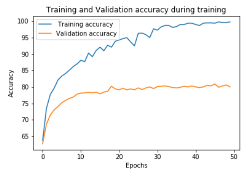
(f) Loss - weight sharing



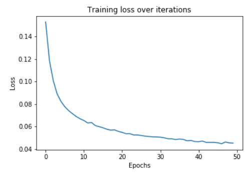
(g) std dev training - weight sharing



(h) std dev validation - weight sharing



(i) Accuracy - with both



(j) Loss - with both

Figure 4: Best obtained results

2 Mini Project 2

2.1 Introduction

The objective of this second mini-project is to design a mini "deep learning framework" using only pytorch's tensor operations and the standard math library, training it and testing its performance, hence in particular without using autograd or the neural-network modules.

2.2 Structure

In order to construct a framework that would have given the ability to the user to generate Neural Networks with linear layers, Relu and Tanh activations, MSE Loss and SGD optimizers, we implement the following modules.

- **Module parent class** : All of our subsequent modules inherit from the same class "Module" that gives the general architecture of a module. It contains 3 methods - forward(), backward and param(), for the forward pass, backward pass and to return the parameters of the module. The param method is fixed for each module and it returns the parameter for the given module. We derive all our layers from this class.
- **Parameter class** : We define a Parameter class to store the parameters of each layer. The parameter class has only an initialization method, and it contains the parameter, its gradient and the input to the parameter.
- **Sequential container** The sequential container is a module that is used to instantiate a model. It allows to define a new network by simply listing its components which in this case are : Linear layers and activation functions. The forward function sequentially applies forward to each layer and uses it as the output of the next layer and the backward applies backward to the reversed list of layers.
- **Linear Layer** : The basic structure is to create a Module, initializing all parameters of the layer into a single parameter class. For the linear layer, this is only a matrix, which has kaiming initialization, with an option for bias. The forward function stores the input for the layer in the parameter class, and applies a matrix multiplication. The backward function takes the previous gradient and uses the input for the layer to store the gradient for the parameter in the parameter class and returns the error for the previous layer. For the linear layer, we additionally implemented dropout. The structure of Linear Layer can be used to create other layers like convolutions, LSTMs or RNNs.
- **Activation functions** : The activation functions are also layers in our structure, so they are inherited from the Module class. Additionally, we do not need to learn the parameters for a function, but we do require its input and gradients for the forward and backward pass. Thus, we create a Parameter with $None$ parameter value, to store only the input and gradients. As before, the forward pass for the layer includes storing the input and returning the output after applying the function and the backward pass includes computing the gradient of the function and applying it to previous gradients. Both the tanh and the ReLU were implemented. For the derivative of the forward function tanh we consider the backward function $1 - tanh^2$ and and the derivative of the forward function ReLU is the backward Heaviside function.
- **MSE loss function** : The loss functions are separate objects with only forward and backward methods. The forward method takes the actual and predicted labels and computes the loss while the backward method returns the gradient of the loss from the predicted and the true labels.
- **Optimizer** : The optimizer class is initialized with the parameters of the complete model and the parameters for the optimizer. Along with each parameter, we also create additional parameters which are used for their update steps, like gradient moments, for momentum or Adam. Each optimizer has 2 additional methods – $step()$ and $zero_grad()$. The $step()$ method

updates all parameters which don't have $None$ data (i.e., excluding activation functions), using the gradient and additional update variables. The $zero_grad()$ function sets the input and gradient for each parameter supplied to the optimizer to $None$, so that we forget previous gradients. We implemented mini-batch SGD with momentum and Adam as our optimizers.

- **Utils** : We created additional functions for computing accuracy and creating dataset.

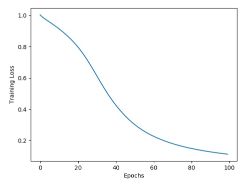
  All loss functions and modules are present in $nn.py$, all optimizers in $optimizers.py$ and the utils in $utils.py$.

2.3 Experiments

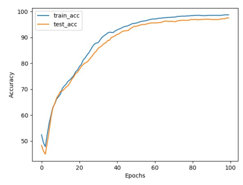
The input is sampled uniformly from $[0,1]^2$ and the label is 1 if the point lies in a circle of radius $\frac{1}{\sqrt{2\pi}}$ centered at (0.5,0.5) and -1 if outside. As this circle covers half of the area of the unit square, the labels are equally distributed. Our architecture contains 3 linear layers with 25 hidden nodes each with a single output node. The activation functions which we use are (ReLU, ReLU, Tanh, Tanh).

Training Procedure : We create a model as a Sequential container with all the layers, the optimizer with all the parameters of the model and create an instance of the loss function. Then, for each minibatch, we compute the output and the loss using the output. In each forward pass, we store the input for each layer in its parameter. Then, we use the output to compute the gradient wrt the loss and this gradient is used for the backward pass for the Sequential model. After the backward pass for the sequential model, the gradients for all parameters are stored in the model parameters and we can use $optimizer.step()$ to update the parameters of the model using the gradient information.

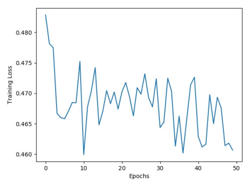
2.3.1 Results



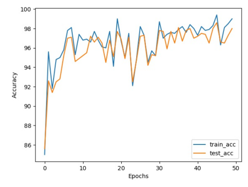
(a) SGD Loss - batch size =1 , lr = 10^{-4}



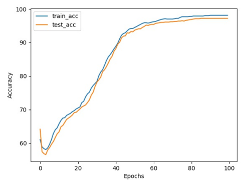
(b) SGD Accuracy batch size =1 , lr = 10^{-4}



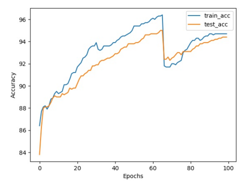
(c) SGD Loss - batch size =2 , lr = 5×10^{-2}



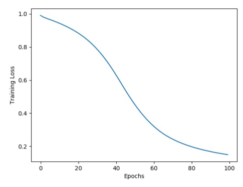
(d) SGD Accuracy batch size =2 , lr = 5×10^{-2}



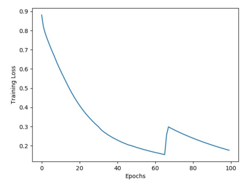
(a) SGD with momentum - accuracy



(b) Adam - accuracy



(c) SGD with momentum -loss



(d) Adam - loss

We used different optimizers –SGD,Adam and SGD with momentum, with different batch sizes(here 1,2), to obtain the best possible configuration. We found that for approximately the same number of

Table 1

Optimizer	$Train\_acc$	$Test\_acc$
SGD batch size = 1	98.7	97.5
SGD batch size = 2	97.6	96.5
SGD with momentum batch size = 1	98.7	97.2
Adam	94.7	94.7

gradient evaluations per data point,(100), all these methods had similar training and test accuracy except for Adam, which performed slightly worse. $test.py$ file runs the model with SGD, without momentum and batch size 1.