EE 569 Homework 5 Write Up

Armin Bazarjani

April 14th, 2021

1 CNN Training on LeNet-5

1.1 Motivation

With the rise of computational power, we have seen a resurgence in a once forgotten technique of machine learning, neural networks. Now, thanks to the parallel processing power of GPU's, we can train deeper and larger neural networks, this gave rise to the field now known as deep learning. With deep learning came a huge boost in accuracy metrics across many different domains for problems that were once incredibly difficult. The current domain that is undergoing this huge paradigm shift is the field of computer vision.

An architecture that has played a vital role in computer vision adopting deep learning has been the convolutional neural network (CNN). At a very high level a CNN is based on a shared-weight architecture that uses different convolution kernels across an image. These kernels are very similar to the ones we have seen in traditional image processing. However, the largest difference is that the CNN decides what the weights of the kernels are. This is done through optimizing a loss function through gradient descent (more on this later).

In today's day and age, you can't be in the field computer vision without being somewhat familiar with machine learning and especially deep learning. This is a powerful technique that has completely shifted many fields and as of now, is the undisputed champion.

1.2 Approach and Procedures

1.2.1 CNN

As I stated earlier a CNN is an extension to the traditional structure of a neural network as they are made up of layers of neurons with different learnable weights and biases. Each neuron will receive some input, do a dot product multiplication, and often times will follow up with a non-linearity function. The CNN itself is still differentiable, meaning we can apply backpropagation to iteratively tune the weights and biases according to an appropriate loss function.

The biggest change is that a CNN was initially designed under the assumption that it would be used on images. Because of this assumption, we can code in some useful techniques to improve the efficiency of the network.

1.2.2 PyTorch

PyTorch is an open source machine learning library. It is built on top of tensors which operate the same way as arrays except for the fact that we can also use them with a GPU to speed up the operations. There are also many different types of tensors that we can use depending on the task at hand.

Without getting bogged down by too many of the implementation details, it will be useful to mention one other thing about PyTorch, that being its autograd module. The autograd module performs automatic differentiation within the computation graph of a neural network that you construct using PyTorch. This saves a lot of time during the training process because it computes the differentiation of the networks parameters during its forward pass.

1.2.3 LeNet-5

LeNet-5 is a convolutional neural network that was created by Yann Lecun et al. in 1989. It was one of the earliest proposed CNN architectures and it helped promote the use of deep learning for image classification problems.

The structure of an LeNet-5 is quite simple given the extravagant and deep architectures of today. It is composed of two convolutional layers which are immediately followed by a max-pooling operation. Then, there are 3 fully-connected layers following into the output.

1.2.4 My Approach

My approach was to use PyTorch to construct a LeNet-5 convolutional neural network. Luckily, PyTorch also has different benchmark datasets saved that we can use at our disposal to quickly and easily test our models.

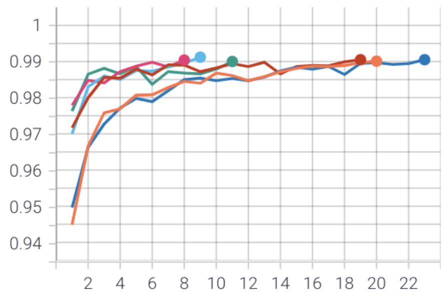
My approach was to create a few different python scripts to keep everything separated and readable. I have two different scripts for parts (b) and (c). I also have two different scripts for reading in the data. The first I designed for part (b) that asks which dataset you want to read in (mnist, fashion-mnist, cifar10) and also if you want to normalize the data. The second dataset reading script was designed for part (c) and it has three different reading functions. One to read in the positive data, one to read in the negative data, and a final to to concatenate the two datasets. My final separate script contains the LeNet-5 neural network class. I thought it would be easier to define it separately so I can just import it into my scripts for the different problems rather than constructing it twice.

As for training, I essentially follow the normal training procedure for any PyTorch neural network. I also took advantage of Tensorboard when training so I can monitor the training and testing accuracy at each epoch. When testing out the different hyperparameter configurations, I employed early-stopping as soon as the test accuracy was above the specified threshold.

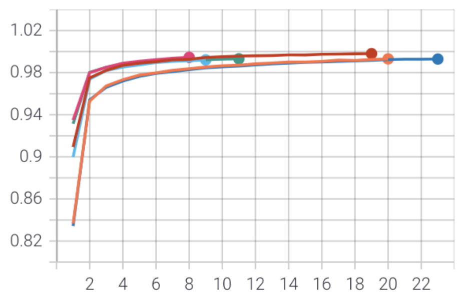
1.3 Experimental Results

1.3.1 Classification on Different Datasets

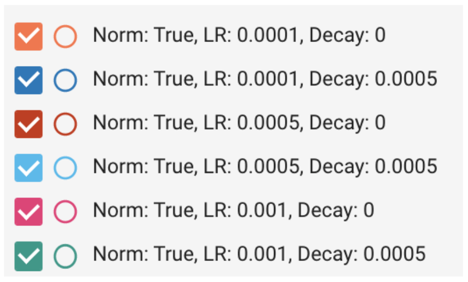
The 6 different training and testing runs for MNIST as well as the key:



(a) training



(b) testing

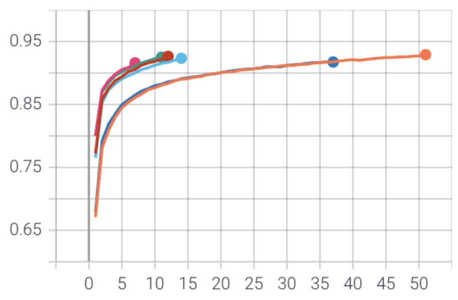


✓ ○ Norm: True, LR: 0.0001, Decay: 0
✓ ○ Norm: True, LR: 0.0001, Decay: 0.0005
✓ ○ Norm: True, LR: 0.0005, Decay: 0
✓ ○ Norm: True, LR: 0.0005, Decay: 0.0005
✓ ○ Norm: True, LR: 0.001, Decay: 0
✓ ○ Norm: True, LR: 0.001, Decay: 0.0005

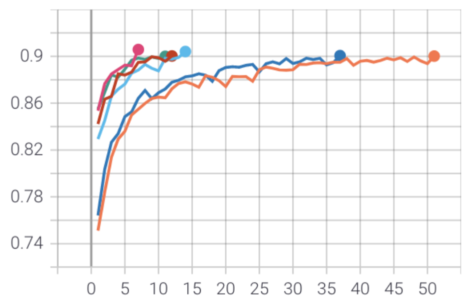
(c) key

Figure 1: Different hyperparameters on MNIST

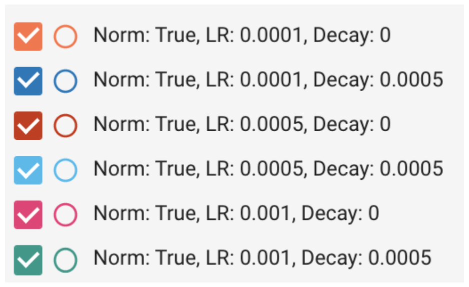
The 6 different training and testing runs for Fashion-MNIST as well as the key:



(a) training



(b) testing

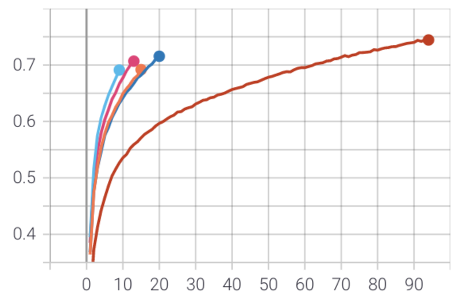


✓ ○ Norm: True, LR: 0.0001, Decay: 0
✓ ○ Norm: True, LR: 0.0001, Decay: 0.0005
✓ ○ Norm: True, LR: 0.0005, Decay: 0
✓ ○ Norm: True, LR: 0.0005, Decay: 0.0005
✓ ○ Norm: True, LR: 0.001, Decay: 0
✓ ○ Norm: True, LR: 0.001, Decay: 0.0005

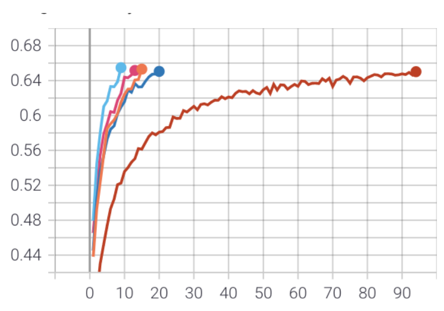
(c) key

Figure 2: Different hyperparameters on Fashion-MNIST

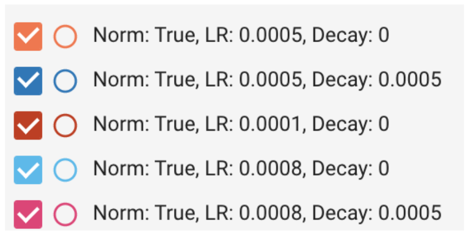
The 5 different training and testing runs for Cifar-10 as well as the key:



(a) training



(b) testing



✓ ○ Norm: True, LR: 0.0005, Decay: 0

✓ ○ Norm: True, LR: 0.0005, Decay: 0.0005

✓ ○ Norm: True, LR: 0.0001, Decay: 0

✓ ○ Norm: True, LR: 0.0008, Decay: 0

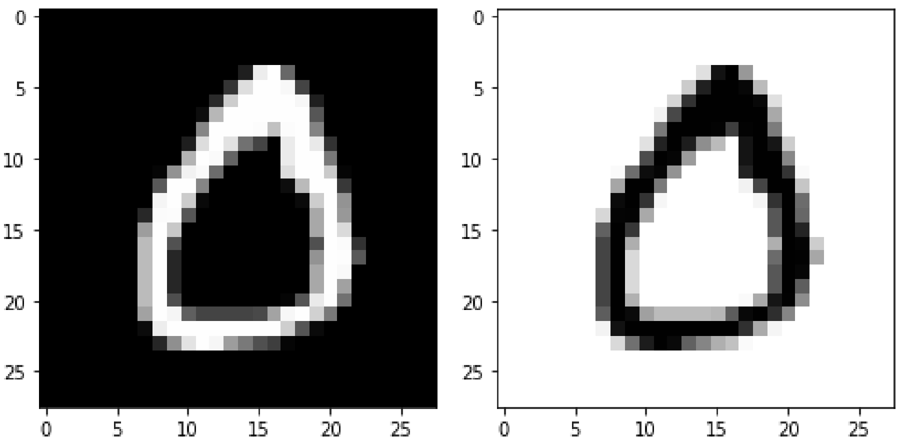
✓ ○ Norm: True, LR: 0.0008, Decay: 0.0005

(c) key

Figure 3: Different hyperparameters on Cifar-10

1.3.2 Training on Negative Images

A sample from the positive MNIST dataset and the same sample from the negative MNIST datast are as follows:



(a) positive (b) negative

Figure 4: Sample from positive and negative MNIST datasets

1.4 Discussion

1.4.1 (a) CNN Architecture

Question 1 - A fully connected layer is a layer of a neural network where the neurons are connected to the activations of all of the neurons of a previous layer. In the context of a CNN, fully connected layers are often used at the end of the network to sequentially downsample the network until we get it to a desired size. For classification problems this desired size will be the number of classes.

The convolutional layer of a CNN is the core building block. The parameters of a convolutional layer consist of a set of learnable filters (or kernels). They are usually pretty small and have equal height and width. However, they extend through the full depth of the volume of whatever its input is. Intuitively, we can think that these filters learn different visual features. The first few convolutional layers will learn high level visual features, while the latter convolutional layers will learn very low level features.

The max pooling layer works by taking the max of values within a certain neighborhood (a 3x3 region for example). We typically iteratively apply this max-pooling operation over the output of a convolutional layer before it goes to another one. The main reason for its use is to reduce the spatial size of the object going through the model to alleviate computational pressure and to lessen the number of parameters to learn.

Activation functions are used in a neural network as a way to add non-linearity to the model. This will not only allow the model to learn more complex data, but also greatly improve the models ability to generalize.

The softmax function is normally used in the last layer of a neural network to normalize the output to a probability distribution. This is very useful in classification problems where you would like to output a probability of a certain class.

Question 2 - Overfitting occurs when a model has low bias and high variance. This occurs when the model fits the noise that is present in the training data instead of the underlying relationship. There are many different ways to handle the overfitting problem. The first method is to add more data as this will give us more training samples to hopefully learn the relationship. An extension of this is to augment the existing training data to create more samples and to also give a new perspective on the current samples. Some data augmentation techniques include random cropping and rotating the image. The next most common technique is to use some sort of regularization. This includes dropout (Deleting a random subset of activations during training) or L_1/L_2 regularization which add penalty terms to the cost function this encourages the values of our weights to decrease.

Question 3 - ReLU computes the function

$$f(x) = max(0, x)$$

Leaky ReLU computes

$$f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x)$$

where α is some small constant.

ELU is

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

The ReLU activation function has been a widespread favorite and outperforms any previous activation function, the main con of it is that sometimes it can "die". For example a large gradient flow could cause a weight update such that the neuron will never activate again, thus the gradient will always be zero.

Leaky ReLU attempts to mitigate this problem by instead of setting the output to 0 if $x < 0$, instead it will be have a very small negative slope. However, it is worthwhile to note that the results of using this activation function are not always consistent.

ELU was another response the dying ReLU problem. It conatins all of the benefits of ReLU and it is slightly more robust to noise when compared to Leaky ReLU. The main pitfall of ELU activation is that it requires the use of a very costly exponential function.

Question 4 - L1Loss and MSELoss are both different types of regression losses. Thus they will find use if we were interested in a regression problem (for example pricing a house, predicting the temperature, etc.). Now, comparing the two I because the L1Loss is equivalent to using mean absolute error loss it would probably provide more robustness to outliers compared to the using the mean squared error loss. Generally, this will not be the case and MSELoss would work quite well.

BCELoss or Binary Cross Entropy Loss is, as the name most commonly suggests, a loss that is used for binary classification problems, ie. there are only two classes. It makes more sense to use this loss for a two class problem as it is more computationally efficient.

1.4.2 (b) Compare classification performance on different datasets

Questions 1-4 - Please look above in section 1.3.1 "Classification on Different Datasets" to see my training and testing accuracy results for the 5 different hyperparameter configurations on the different datasets, as well as the best performing one (in terms of speed).

MNIST - The fastest hyperparameter selection was with a learning rate of 0.001 and a decay of 0, taking 8 epochs to reach the threshold of 99% accuracy. The slowest hyperparameter combination was seen with a learning rate of 0.0001 and a decay of 0.0005, taking 23 epochs. Among all of the tests the higher learning reached the threshold quicker and the decay only slowed the process.

Fashion-MNIST The fastest hyperparameter combination was with a learning rate of 0.001 and a decay of 0, taking 6 epochs. The slowest combination was with a learning rate of 0.0001 and a decay of 0, taking 51 epochs. Interestingly the decay of 0.0005 quickened the learning process when using learning rates of 0.0001 and 0.0005, however it was slower to reach the threshold with a learning rate of 0.001.

Cifar-10 For cifar-10 I had to make the learning rates lower than what I used for the MNIST datasets. I noticed that using a learning rate of 0.001 made the model converge too quickly to an accuracy of 63%. The best hyperparameter combination was with a learning rate of 0.0008 and a decay of 0, taking 9 epochs. The worst combination was with a learning rate of 0.0001 and a decay of 0, taking 94 epochs.

Question 5 - Because we were asked to train to different accuracy levels, I will only comment on the number of epochs it took on average, as well as the quickest training one. So, we can see that the order of learning from fastest to slowest goes MNIST, Fashion-MNIST, and Cifar-10. To me, this makes quite a lot of sense. MNIST and Fashion-MNIST are both far easier datasets than Cifar-10. This is because the images are more clearly defined and the images themselves are more easily distinguishable from one another, ie. the differences between the classes in the two MNIST datasets are much greater than in Cifar-10. On top of that, the variability within each class in Cifar-10 is far greater than in MNIST or Fashion-MNIST.

1.4.3 (c) Apply trained network to negative images

Question 1 - Please look above in section 1.3.2 "Training on Negative Images" to see the same sample from the both the positive MNIST and negative MNIST datasets that I constructed. The way I did this was by taking advantage of the PyTorch transform functionality. Because there is no function to implement an inversion transformation, I constructed a class that takes as input an image and subtracts each element from 1. The reason I subtract it from 1 instead of 255 is because I first use the .ToTensor() transformation and that resamples the pixel values to be between 0-1 instead of 0-255. I then compose this class I constructed with the other transformations I performed.

The mean and standard deviation of each is as follows. For the positive dataset I have a mean of 0.1285 and a standard deviation of 0.3057. For the negative dataset I have a mean of 0.8654 and a standard deviation of 0.3117. This makes sense as the standard deviations are pretty much the same and the mean for the negative dataset is one minus the mean of the positive dataset. I also normalized the negative dataset with the new mean and standard deviation.

Question 2 - The accuracy of testing negative images on the LeNet-5 that was trained on positive images was around 20%. This is better than random guessing which would be around 10%, so you can make the argument that the model did learn some underlying information about classifying the numbers. However, it is still very bad compared to the 99% test accuracy it gets on positive test images. Therefore, it is safe to say that for the most part the model does not do a very good job generalizing past even the most minute change in the dataset.

Question 3 - For this, my test was very simple. I made no structural modifications to the LeNet-5 network. Instead I combined both the positive and negative MNIST images into a single dataset and shuffled them together so they were in no particular order. I used this combined and shuffled dataset to then train a LeNet-5 model using the same optimizer (Adam, 0.001 learning rate, 0 weight decay) for 15 epochs. I was pleasantly surprised to see that this worked pretty well. After training for 10 epochs I was greeted with a 98.95% accuracy on the negative test set and the combined test set.