67750 | Advanced Practical Course in Machine Learning | Exercise 1

Guy Lutsker 207029448

1 Understanding The Data

Firstly I started by looking our data, here we got a training sample of 5625 labeled pictures(32×32 pixels) of either cars, trucks, or cats.

The first thing I noticed is that most of the data were pictures of cars.

The second thing I noticed, is that there were a lot of mislabeled pictures:

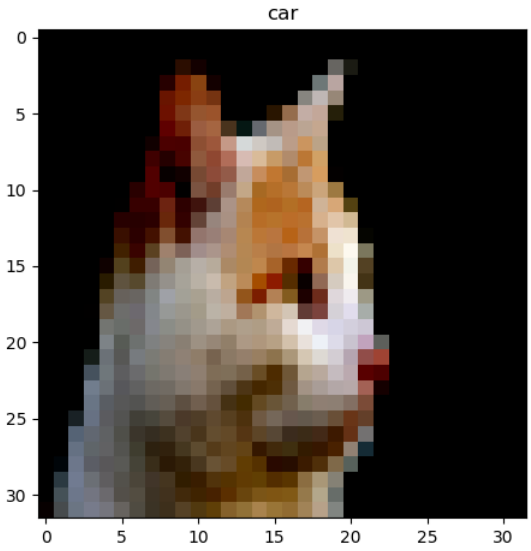

Figure 1: Cat Mislabeled As a Car.

And so the first challenge I was facing is thinking how to manage this corrupted data set.

2 Performance

2.1 Initial Performance

Next I tried running the naive network provided for us. The results(unsurprisingly) were not great: The network ran with 88% accuracy, which sounds not too bad, but when I dove into the labelings themselves I saw a big issue - the network simply labeled every single sample as a car - and since most of the samples were cars the accuracy seemed high. The training data results can be show in the next confusion matrix:

truth/predicted	cars	trucks	cats
cars	4949	0	0
trucks	449	0	0
cats	227	0	0

Figure 2: Initial Train Error Results of Naive Network

And the test data result:

truth/predicted	cars	trucks	cats
cars	551	0	0
trucks	51	0	0
cats	23	0	0

Figure 3: Initial Test Error Results of Naive Network

Next, since most of the mislabeling was classifying objects as cars, I thought of trying to train the network the cats and trucks. It seemed that most of the data that was really trucks/cats was a true labeling(actually, the trucks were labeled correctly, and cats were mostly fine..). my results were(82% accuracy):

truth/predicted	trucks	cats
trucks	43	8
cats	5	10

Figure 4: Initial Test Error Results of Naive Network on Trucks & Cats

2.2 Possible Improvements

When looking over the naive general code provided that generated the neural network, some immediate possible improvements came to mind. Firstly we needed to address the unbalances data - the fact that there were mostly cars in the data set. To overcome this issue I opted to use the WeightedRandomSampler tool in Pytorch - basically gave it a batch size of a certain size and let the network train over batches with balanced number of samples in each class. Next I wanted to address the default parameters of the network - I tried using the Adam optimizer, and tried using different learning rates. In addition I increased the epoch size. Over all, at this stage of development I found that the following hyper parameters gave the best results(90% accuracy in the next table): Using the Adam optimizer, default learning rate, batch size of 5, and an epoch magnitude of 20:

truth/predicted	cars	trucks	cats
cars	536	15	0
trucks	25	26	0
cats	20	2	1

Figure 5: Performance After Initial Optimization Attempt

First of all we see an increase in the accuracy(88⇒90), but more importantly we see a generalization in the way the network classifies the data - it no longer classified every single sample as a car! a big improvement in the right direction(I saw an even bigger generalization when running with epoch of magnitude 50, but it had 88% accuracy). This means the WeightedRandomSampler tool helped, but with limited successes, since it still struggles with cats. To try and solve this issue I had a idea to maybe try and relabel the cats using the cat/truck network. In more detail, maybe we can take images labeled as cars, and run them trough the truck/cat network, and if we label a car as a cat,

we can assume it was a cat mislabeled as a car, and relabel it correctly. The motivation is based of the idea that cars should(maybe) look more like trucks, and so if the trucks/cats network labeled a car as a truck it is a car, and if it labeled it as a cat, it is a mislabeled cat. With this idea at heart I applied the improvements from the general net to the truck/cat net and got 93-94% accuracy:

truth/predicted	trucks	cats
trucks	50	1
cats	4	19

Figure 6: Performance After Initial Optimization Attempt on Trucks & Cats Network

As it turns out this idea(as logical as it seemed in the beginning) did not prove to be as successful as I initially hoped. When I ran the cars from the test data on the trucks/cats network I saw the network still mislabeled cats as cars with a very high probability. Using softmax I tried to establish a threshold for a "confidence" score and see how confident was the labeling, unfortunately the results were as follows: The average confidence score of cars bring tagged as cats was 0.90319, whilst the same for cats(tagged initially as cars) was 0.948055 - also there were a lot of cars being tagged as cats with a score of 100%. That meant that even if I chose a really high threshold it wouldn't do much - from a quick calculation I saw that even if I set the threshold score for relabeling a car to a cat if the network(truck&cat) tagged it as a cat with 0.999 confidence I would have mislabeled 10(out of 57 in test set) cars as cats, and only relabeled correctly 5(out of 11 in test set) cats who were mislabeled initially as cars. And so, sadly it seems that this idea wont be used :(

2.3 Augmentation

With the disappointment of my previous ideas, I turned to the augmentation method. Augmentation is basically when we try to enlarge our training set by "simulating" more data from our training set. For example, by taking an image and cropping it, resulting in another image we can label with the same label as the original, and thusly we magically increased our training data. Its important to state that these methods should be used cautiously, since it could easily cause overfitting issues with the resulting model. I tried training the same network using torchvision.transforms with the following transforms: Horizontal/Vertical flips, Grayscale,Perspective, Normalization, and various rotations. Unfortunately this method did not prove to be useful as well, The resulting network seemed to have just about the same average accuracy - 90% accuracy for all classes, and 93% for trucks/cats. And so, this seems like a bust as well.

2.4 Adaptive Learning Rate

Another attempt I had at trying and increase the performance of my network was using various kinds of adaptive learning rates. The idea behind changing the learning rate though out the training process, is that during learning we are trying to find an optimum of a function. And so in the beginning we try to have "big steps" in the direction of the optimum, and as we approach, we want to decrease out steps in order to keep ourselves near the optimum(and not "fall" to a near by local minima/maxima). Here I tried using several initial learning rates(usually around 1×10^{-4}), and various ways to try and decrease it(usually multiplying by 0.1 every several epochs, or passes though the sample data), and all seemed to have very little effect on the network accuracy, it looked like almost nothing I did using traditional methods of trying to increase the networks performance worked.

3 Analyzing Learning Rate

In this section I seek to evaluate how the learning rate effects the learning process.

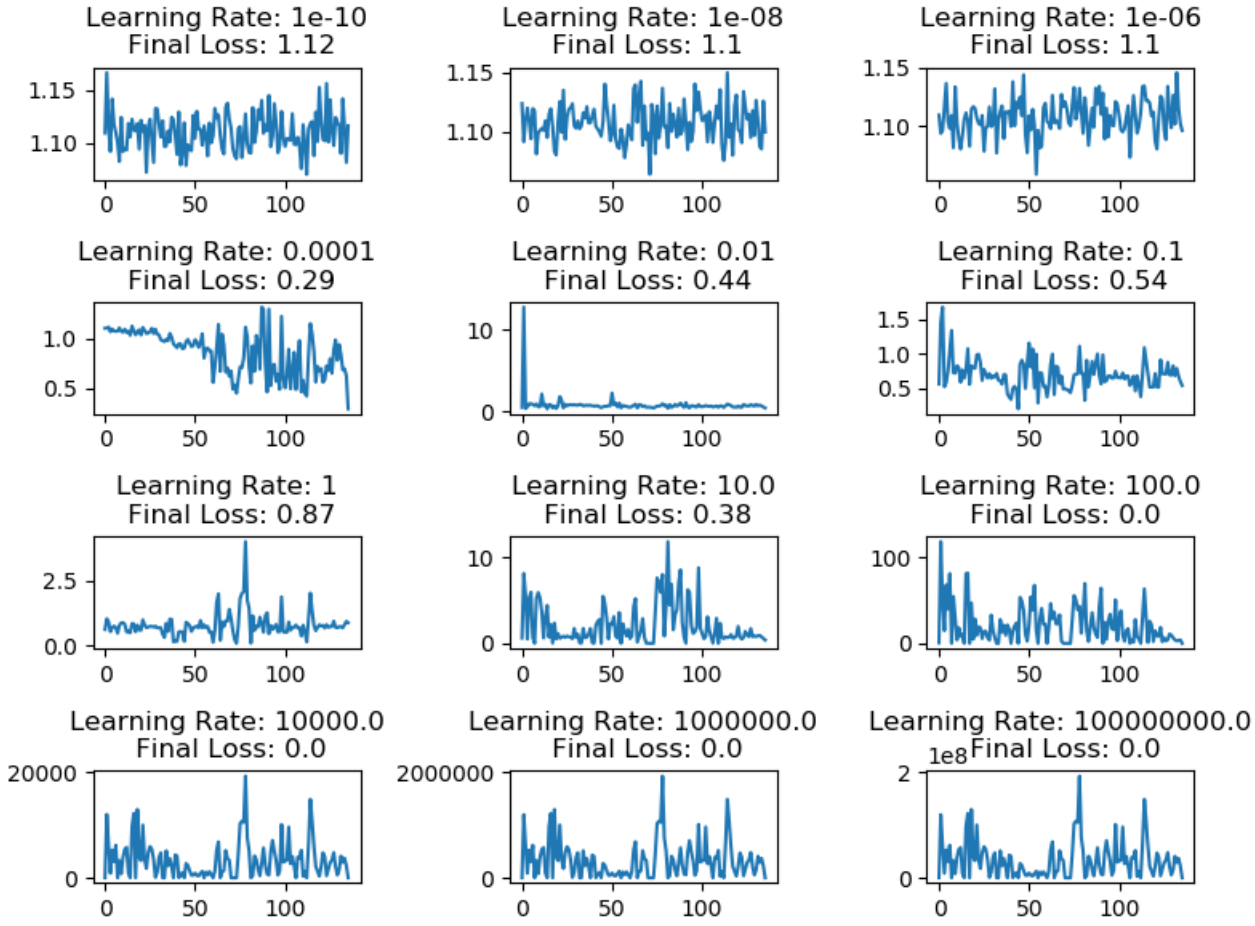

Figure 7: Different Learning Rates Effect on Loss

As we can see in Figure.7 different learning rates have a big effect on the loss generated by the network. We can see that when the learning rate is low the network has a hard time converging to an optimum, and the final training loss is staying consistently relatively high. On the other side of the spectrum, if the learning rate is too high we see that network cant converge to a "good" solution as well. Yet if the learning rate is just right(a good example is 1×10^{-4}) the loss is able to gradually decrease and converge to a minima. A note about really high learning rates(> 1): in extremely high learning rates I was expecting to see the loss function go crazy and not be able to converge at all. Yet here the final loss was 0, I am not quite sure why this happens(maybe such a high learning rate breaks something in Pytorch code? or maybe its part of my grandiose delusions to think I can break something in Pytorch code). The intuition behind how Learning rates effect the convergence quality is given in subsection 2.4.

4 Adversarial Analysis

In order to show that we are still able to fool our network, in this section I will try to create an adversarial example of a picture of a cat being labled by our network as a truck. In order to do this, I will take my model, and "take out" its parameters, so i can train it on a picture of a cat, whilsy telling the model its label is a truck. I will take the resulting weights, and apply them on an empty image to capture the noise:

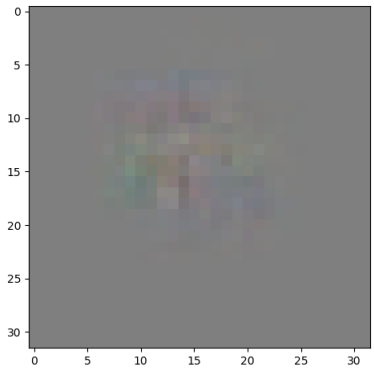

Figure 8: Noise Generated By The Adversarial Method

And so we can add it to an image of a cat, and to our eyes we wont see much of a difference, but our network will be fooled.

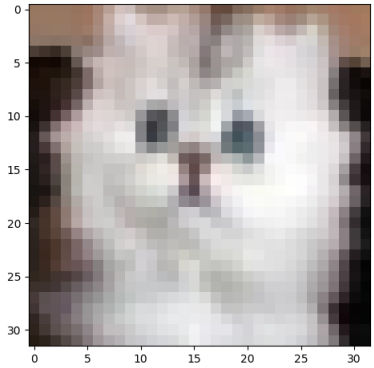

Figure 9: Pre Cat Treatment Being Labeled As A Cat

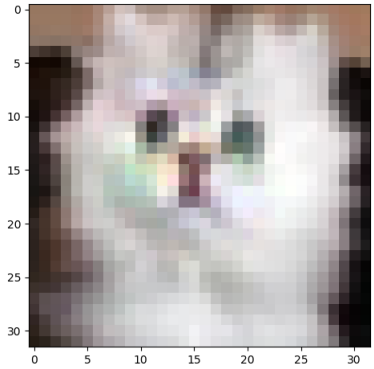

Figure 10: Post Cat Being Labeled As A Truck