Homework 3

Evan Sidrow

October 14, 2017

1 Back Propagation

1.1 Programing questions

See nn.py

1.2 Analysis

1.2.1 What is the structure of your neural network?

tinyMNIST Our neural network has an input dimension of 196, since this is the number of pixels in the training pictures. From there, we have a hidden layer of dimension 30, which is a parameter put in to the original network. Therefore, our first W matrix should be size (30,196), and our first bias vector should be dimension (30,1). From there, we undergo a sigmoid activation function and move to an output layer of dimension 10, since we have 10 different digits. Therefore, our second W matrix should be size (10,30), and our second bias vector should be size (10,1). Finally, we have the last sigmoid activation function to classify our instance.

tinyTOY Our neural network has an input dimension of 2, since the toy data set has two dimensions for each training instance. Like tiny MNIST, we have 30 nodes in the hidden layer, so W^1 has shape (30,2) and b^1 has shape (30,1) (the data then goes through sigmoid activation). Finally, we only have two different classifications, so the output layer is of dimension 2, so W^2 has shape (2,30) and b^2 has shape (2,1) (the data then goes through sigmoid activation before it is classified).

1.2.2 What is the role of the size of the hidden layer on the train and test accuracy?

Figure 3 shows the test and training accuracy as a function of the hidden layer size after 195 epochs. We see that, at first, increasing the number of hidden nodes greatly increases accuracy. However, as we start to add more and more nodes, the model actually loses accuracy, likely because the model is too complex to converge (at least in a reasonable amount of time). We know that the data is not necessarily over-fit at a high number of hidden nodes because the training accuracy is also low when the number of hidden nodes is high.

1.2.3 How does number of epochs affect train and test accuracy?

Figure 4 shows the test and training accuracy as a function of the number of epochs. We can see that, for every number of internal nodes, the training and test accuracy increases with more epochs. If we have very few internal nodes (5, magenta lines), then the training accuracy quickly jumps up and then meanders slightly up for a while. If we have a "good" number of nodes, then the accuracy converges after ≈ 20 epochs and stays high. If we have too many hidden nodes (200, blue lines), then it takes too long for the accuracy to converge, likely because the model is too complex.

2 Keras CNN

2.1 Programing Questions

See CNN.py for details, but I got an accuracy of 98.58%

2.2 Analysis

2.2.1 Point out at least 3 layer types you used in your model. Explain what they are used for.

Conv2D I used a 2D convolutional layer first in my model. This layer takes 32 separate filters, and runs a 3-by-3 kernel with a stride size of 3 over the image and outputs a linear combination of the pixels. This outputs data to the next layer that has information about how things are "clustered" in the image, since the kernel acts only locally.

MaxPool2D I used a MaxPool2D layer after the 2D convolutional layer. This will move a 2-by-2 window across the image and return the largest value inside of that window. This drops computation time and regularizes the data a bit.

Dense I finished my model with a dense layer. This layer takes the output of the Maxpool and connects every node with 256 different nodes in the dense layer. This is a very robust layer, but it can overfit, so I used a dropout parameter of 0.5 to prevent overfitting.

2.2.2 How did you improve your model?

At first, my test accuracy was too low ($\approx 96\%$), so I increased the number of nodes in my dense layer from 128 to 256, to increase the model complexity. At that point, I was using two dense layers with no dropout. When I did this, my training accuracy was very high ($\approx 99\%$), but the test accuracy was terrible ($< 20\%$). I figured that this was due to overfitting, so I got rid of one of my dense layers and included a dropout parameter of 0.5. This caused the model to converge more slowly, but the test accuracy became much better and was over the 98.5% benchmark.

2.2.3 Try different activation functions and batch sizes. Show the resulting accuracy.

For this exercise, I used a limit of 1000 instances and 15 epochs for computational efficiency. Table 1 shows the the resulting accuracy when using batch sizes of 64 and 256, as well as different activation functions (sigmoid and tanh, as opposed to relu). Note that the sigmoid function is terrible and failed to have an accuracy any higher than random chance ($\approx 10\%$). Also note that a lower batch size resulted in higher accuracy, but it took the CNN longer to run when the batch size was lower.

3 Keras RNN

3.1 Programing questions

See LSTM.py for specifics, but I got an accuracy of 87.5% with my model.

3.2 Analysis

3.2.1 What is the purpose of the embedding layer?

The input of the model is in the form of a list of indices of words in the review. Each of these lists can be of different sizes, and are not interpretable by the model. Therefore, we use the embedding layer to turn the lists of indices into dense vectors of fixed length. Then, the model can interpret the data and output a label.

3.2.2 What is the effect of the hidden dimension size in LSTM?

I ran LSTM with the output size of LSTM switching between (16,32,64,128) to see what the effect on accuracy is. Note that I combine the output of the LSTM into a single dense node as the output layer. For the sake of time, I limited the training set to 1000 instances and 10 epochs. The results are shown in Table 2. I find that the training accuracy always increases as the number of hidden dimensions increase, but the test accuracy starts to level off a bit as hidden dimension increases. This is consistent with the results from problem number 1.

3.2.3 Replace LSTM with GRU and compare the performance

Note that I set the embedding output dimension to 64, the output dimension of LSTM/GRU to 64, the epochs to 15, and the number of training instances to 5000 for the following analysis. LSTM has an accuracy of 0.7734 and loss function ≈ 0.6. GRU has an accuracy of 0.7558 and loss function ≈ 0.7. GRU is slightly less accurate, but the results are comparable. The only change needed to run the code with GRU is to import GRU and replace "LSTM" with "GRU" in the code.

4 Figures

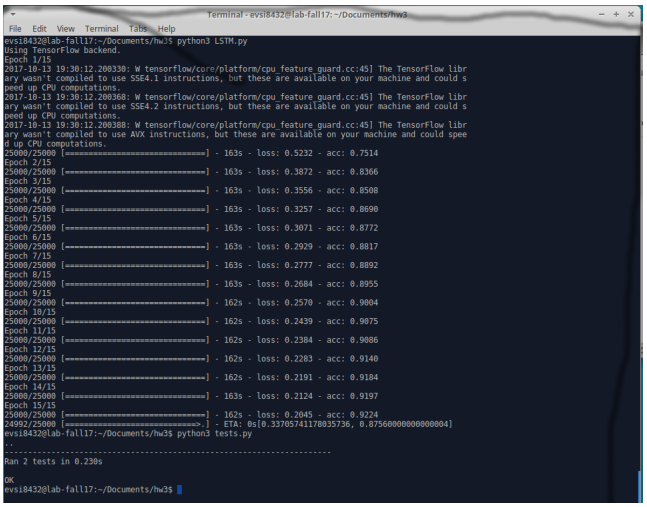

Figure 1: Proof that the test cases were passed and that LSTM had an accuracy of more than 87%

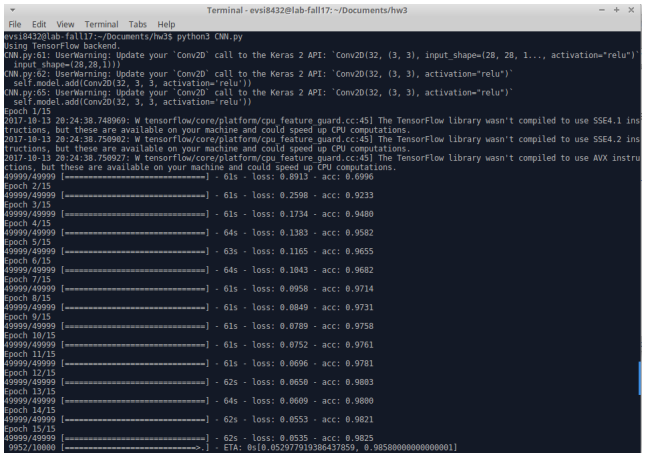

Figure 2: Proof that the CNN achieved an accuracy of 98.5%

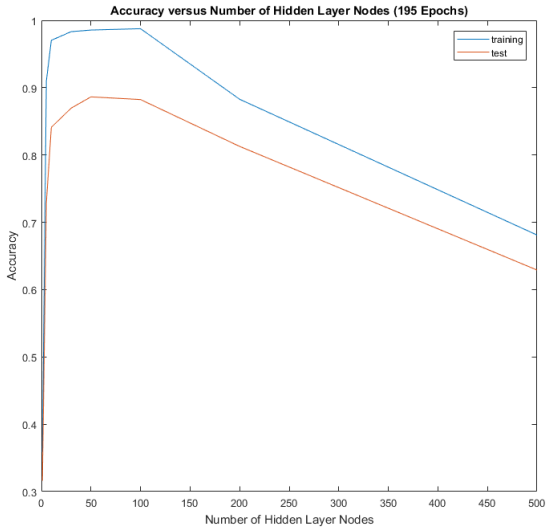

Figure 3: Accuracy after 195 epochs versus hidden layer dimension when 1000 training instances were used.

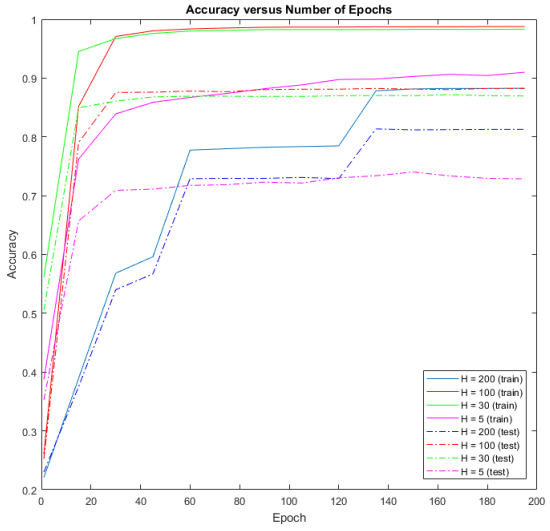

Figure 4: Accuracy versus number of epochs for various hidden layer dimensions when 1000 training instances were used.

Table 1: CNN accuracy for different batch sizes and activation functions (batch size on the left)

	tanh	relu	sigmoid
64		0.9749	
128	0.9565	0.9647	0.1064
256		0.9615	

Table 2: Test Accuracy for various dimensions of the embedding output

Embedding output dim	16	32	64	128
Test Accuracy	0.74356	0.74772	0.746	0.7616