McKean_Brian_HW3

October 13, 2017

CSCI 5622 Fall 2017 HW#3

Brian McKean

1. Back Propagation (35pts)

In this homework, you'll implement a feed-forward neural network for classifying handwritten digits. Your tasks will be to implement back propagation to compute the parameter derivatives for SGD and also do L2 regularization for SGD. First, make sure your code works on a small dataset(tinyTOY.pkl.gz) before moving on to lower-resolution version of MNIST(tinyMNIST.pkl.gz).

1.1 Programming questions (20 pts)

Finish nn.py. — see code

1. Finish back prop function to compute the weights and biases.

2. Finish SGD train function to do L2 regularization.

3. Add code to test on the tinyMNIST dataset.

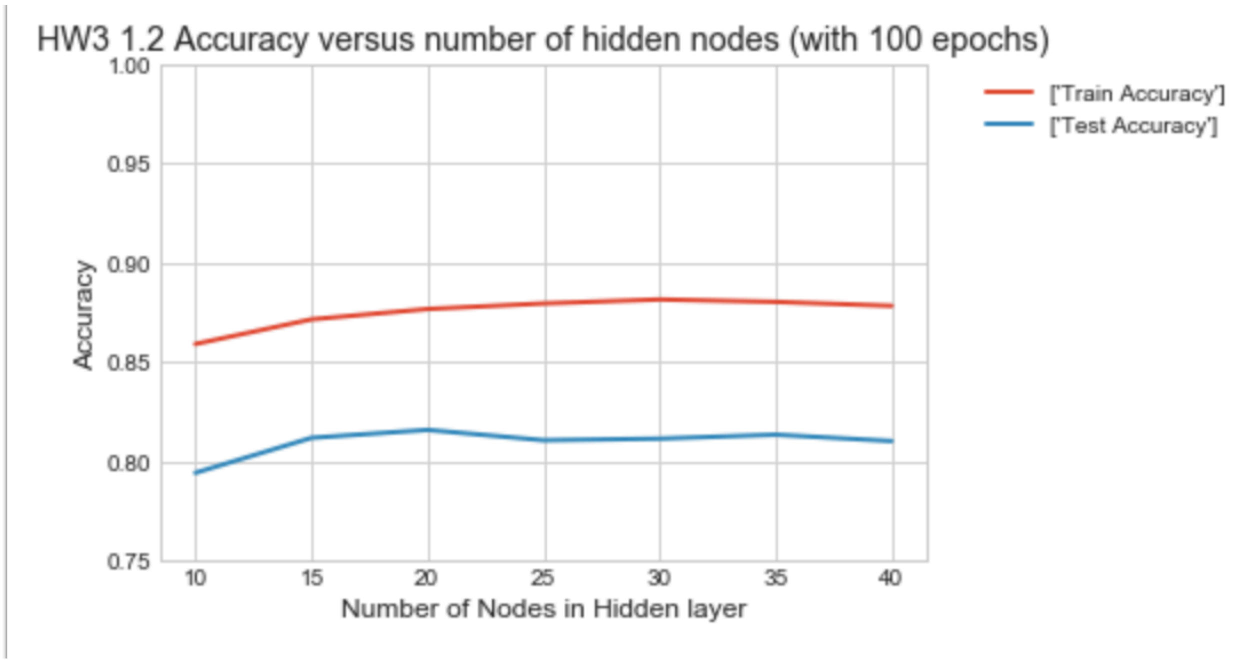
1.2 Analysis (15 points) 1. What is the structure of your neural network (for both tinyTOY and tinyMNIST dataset)?

1. Show the dimensions of the input layer, hidden layer and output layer. tinyToy dataset

• Input Layer 2 • Hidden layer 30 • Output Layer 2

tinyMIST - Input Layer 196 - Hidden layer 30 - Output Layer 10

2. What the role of the size of the hidden layer on train and test accuracy (plot accuracy vs. size of hidden layer using tinyMNIST dataset)?

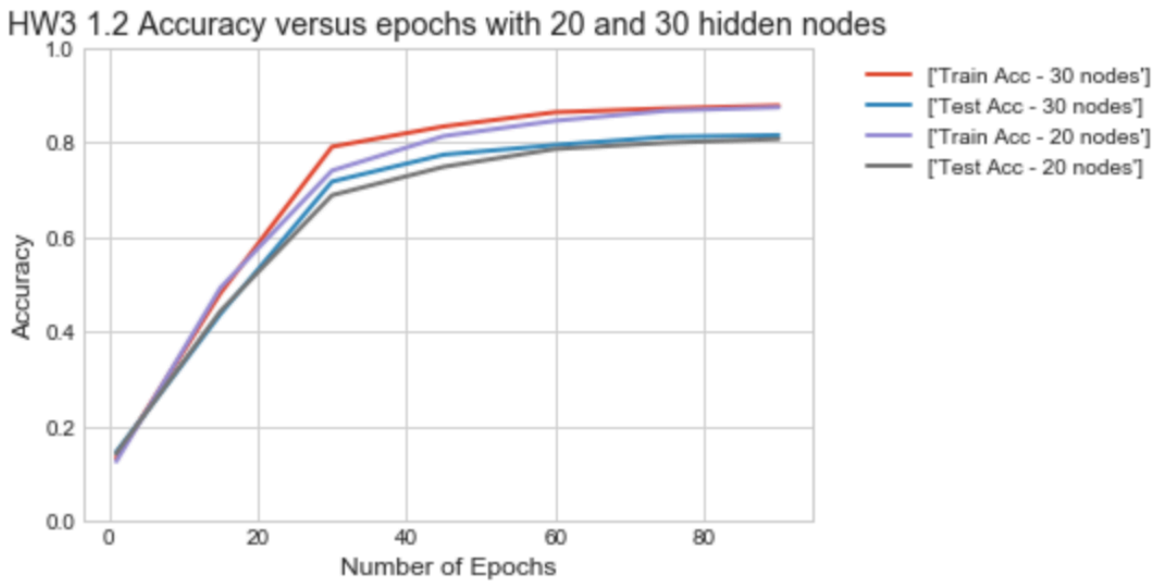


The number of nodes in the hidden layer helps with developing greater accuracy up to a point. In this data analysis, the best accuracy is achieved with a 30-node hidden layer and higher number of nodes do not significantly increase the test accuracy.

Increasing the number of nodes too far cause the test accuracy to vary higher and lower indicating overfitting

The number of nodes helps to reach the best accuracy faster as shown below.

3. How does the number of epochs affect train and test accuracy (plot accuracy vs. epochs using tinyMINST dataset)?



The number of epochs help to increase test accuracy. With more nodes, the higher test accuracies are achieved in fewer epochs.

1.3 2 Keras CNN (35pts)

Here, you will use the Conv2D layer in Keras to build a Convolutional Neural Network for the MNIST dataset. The input dataset is the same as the MNIST dataset in HW1, so you need to reshape the vector of each image into matrix for the use of Conv2D. And you need to build your model using the layers provided by Keras and achieve an accuracy higher than 98.5%

2.1 Programming questions (20pts)

Finish the CNN.py to build a CNN model, train and improve your model to achieve 98.5% accuracy on MNIST dataset. (Hint: use one hot encoding for label, input for the final Dense layer need to be flattened, try Dropout layer to improve your model and don't give up).

1. Reshape your MNIST data.
2. Finish init function to construct your model.
3. Finish train function and fit to your training data.

see code

2.1 Analysis (15pts)

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

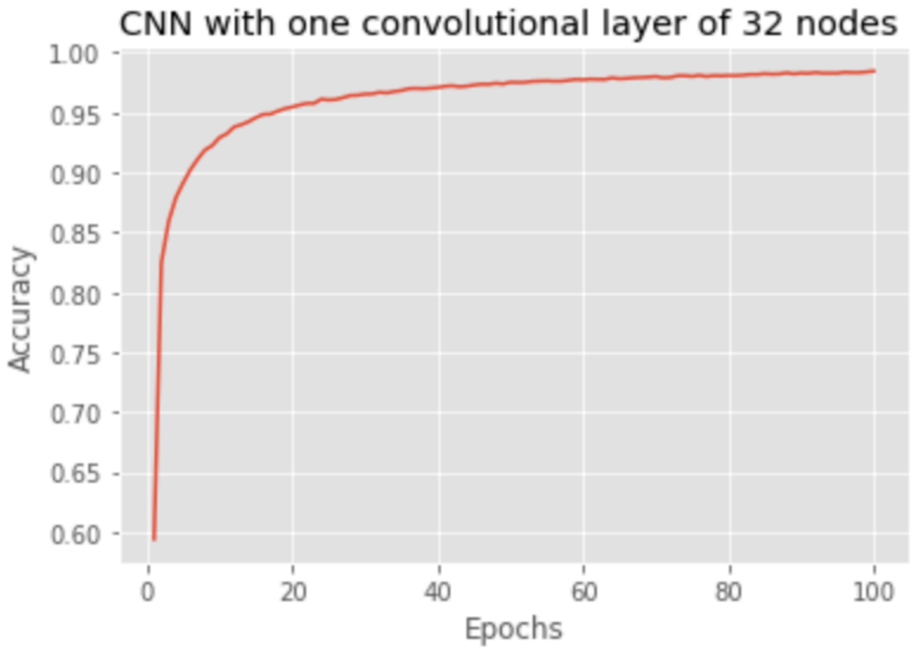
```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1000)              4609000
_________________________________________________________________
dropout_2 (Dropout)          (None, 1000)              0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                10010
 =================================================================
```

- Convolution Layer – slides filter over the 5x5 kernels with a stride of 1
- Max_Pooling Layer – make a single sample out if each 2x2 result
- Dropout – causes the previous results to be randomly dropped in each pass allowing more   nodes to have influence in the result
- Flatten layer – takes the result down to a 1D array
- Dense – layer of nodes that have full connections
- Final Dense Layer – prepares data in form desired for output with 10 classifiers

2. How did you improve your model for higher accuracy?

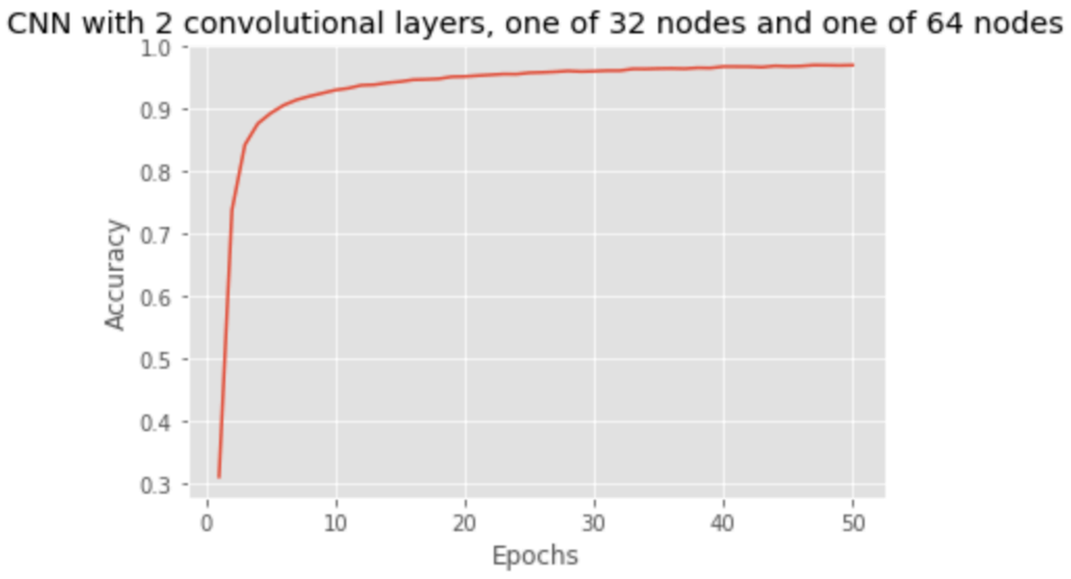
I added dropout to decrease the chances of overfitting.

Then I increase epochs from 50 to 100 with a single 32 node convolution layer and I also tried a second convolutional layer of 64 nodes and was able to get to the

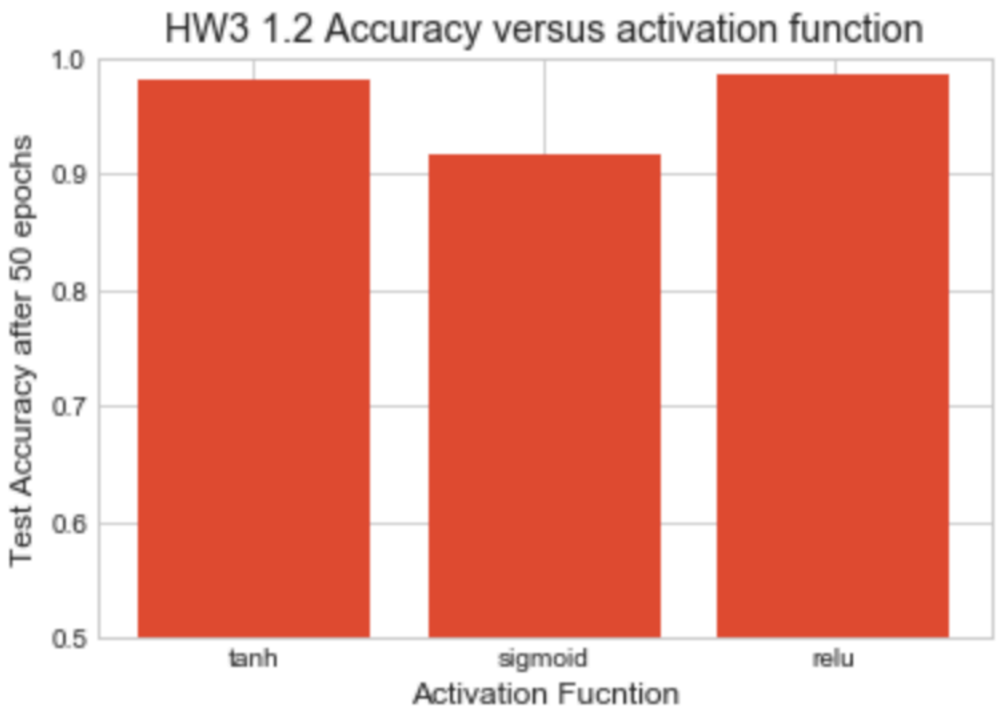


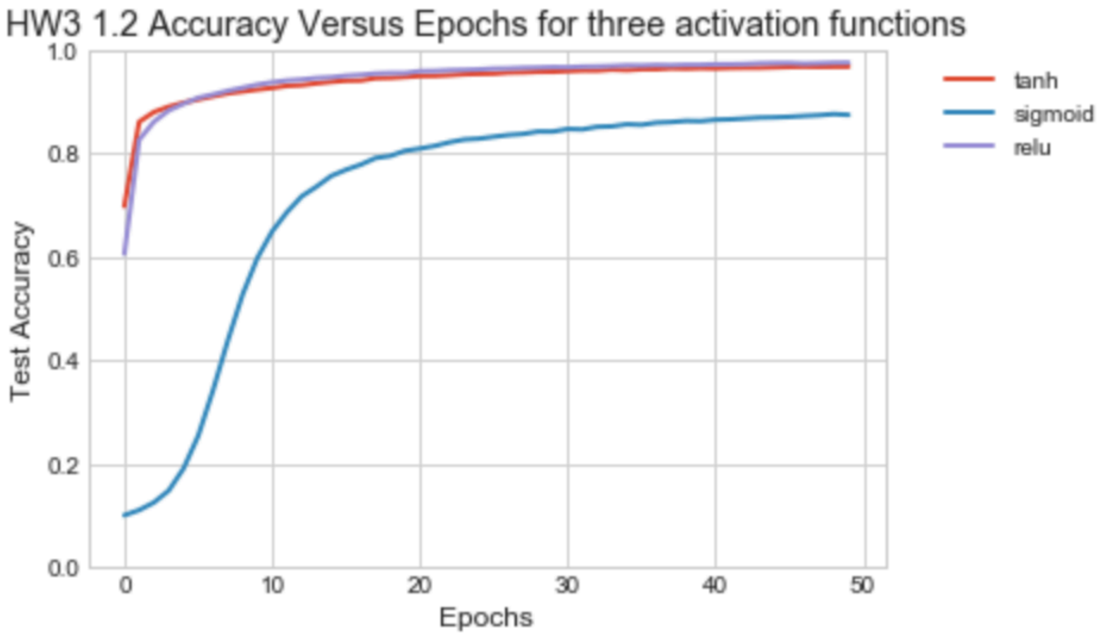
```
Layer (type)                 Output Shape              Param #
    =================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 64)          0
_________________________________________________________________
```

```
flatten_1 (Flatten)          (None, 1024)            0
_________________________________________________________________
dense_1 (Dense)              (None, 1000)            1025000
_________________________________________________________________
dropout_3 (Dropout)          (None, 1000)            0
_________________________________________________________________
dense_2 (Dense)              (None, 10)              10010
  =================================================================
```



3. Try different activation functions and batch sizes. Show the corresponding accuracy.

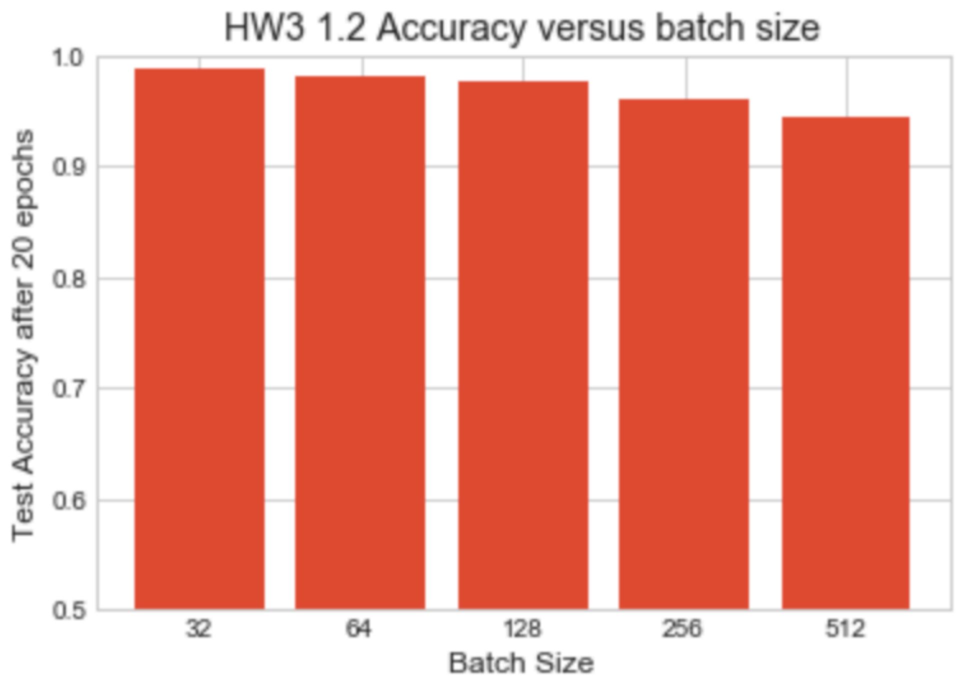


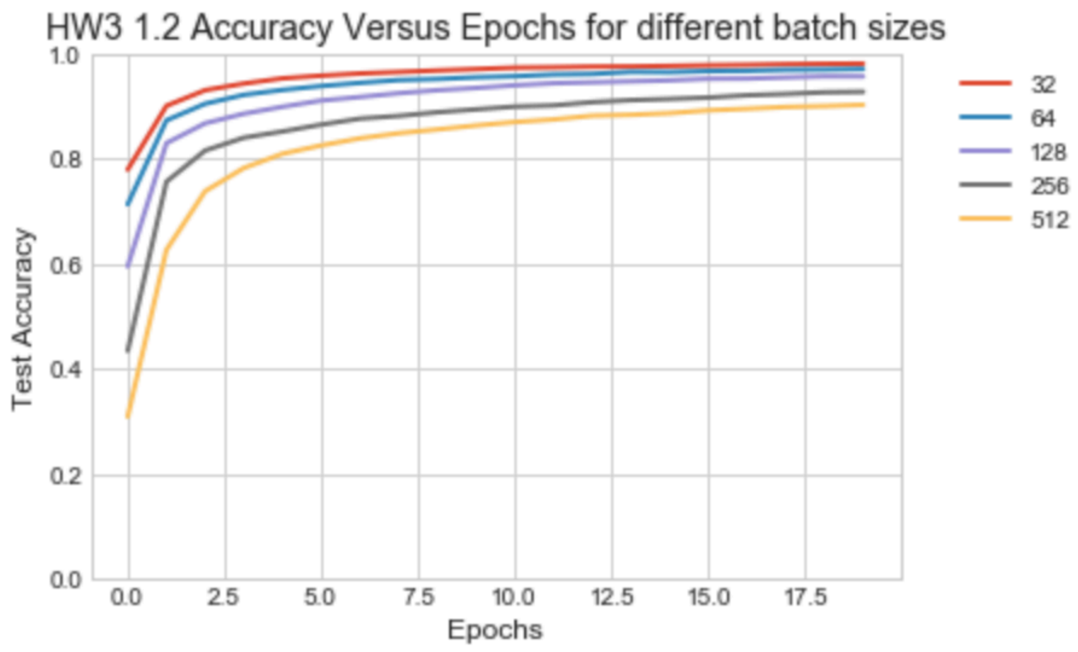


Both ‘tanh’ and ‘relu’ performed very well.

Sigmoid was OK but not as good.

The difference with just a few epochs was far more pronounced. This is likely due to the fact that sigmoid slope gets very small as you move out from the center – where the model is at the early part of training.





There is a definite correlation between accuracy and learning rate versus batch size.

With small batch sizes, higher accuracies ae achieved with many fewer epochs.

3. Keras RNN (30pts)

Here you will use Keras to build a RNN model for sentiment analysis. You should use word embeddings and LSTM to finish LSTM.py. You will test your model on the IMDB dataset. And you are expected to achieve an accuracy higher than 90%.

3.1 Programming questions (15pts)

Finish the LSTM.py to build an RNN model. Use word embeddings as the first layer and use LSTM for sequential prediction. 1. Preprocess data for LSTM (require data of the same length). 2. Finish init function to construct your model. 3. Finish train function and fit to your training data.

Here is one of my better results

```
Using TensorFlow backend.
dict_size=10000, example_length=512, embedding_length=128,  batch_size=4,
epochs=15
_________________________________________________________________
_
Layer (type)              Output Shape            Param #
=================================================================
========
embedding_1 (Embedding)     (None, 512, 128)        1280000
_________________________________________________________________
_
dropout_1 (Dropout)         (None, 512, 128)        0
_________________________________________________________________
_
lstm_1 (LSTM)              (None, 64)             49408
_________________________________________________________________
_
dense_1 (Dense)            (None, 1)              65
=================================================================
========
25000/25000 [==============================] - 5350s - loss: 0.4669
- acc: 0.7792 -
Epoch 2/15
25000/25000 [==============================] - 5300s - loss: 0.2429
- acc: 0.9028 -
Epoch 3/15
25000/25000 [==============================] - 5296s - loss: 0.1613
- acc: 0.9393 -
Epoch 4/15
25000/25000 [==============================] - 5309s - loss: 0.1064
- acc: 0.9622 -
```

3.2 Analysis (15pts)

1. What is the purpose of the embedding layer? (Hint: think about the input and the output).

The embedding layer build word vectors based on the data that you have. The input is a numerical representation of the word and the output vectors of the relationships between words

2. What is the effect of the hidden dimension size in LSTM?

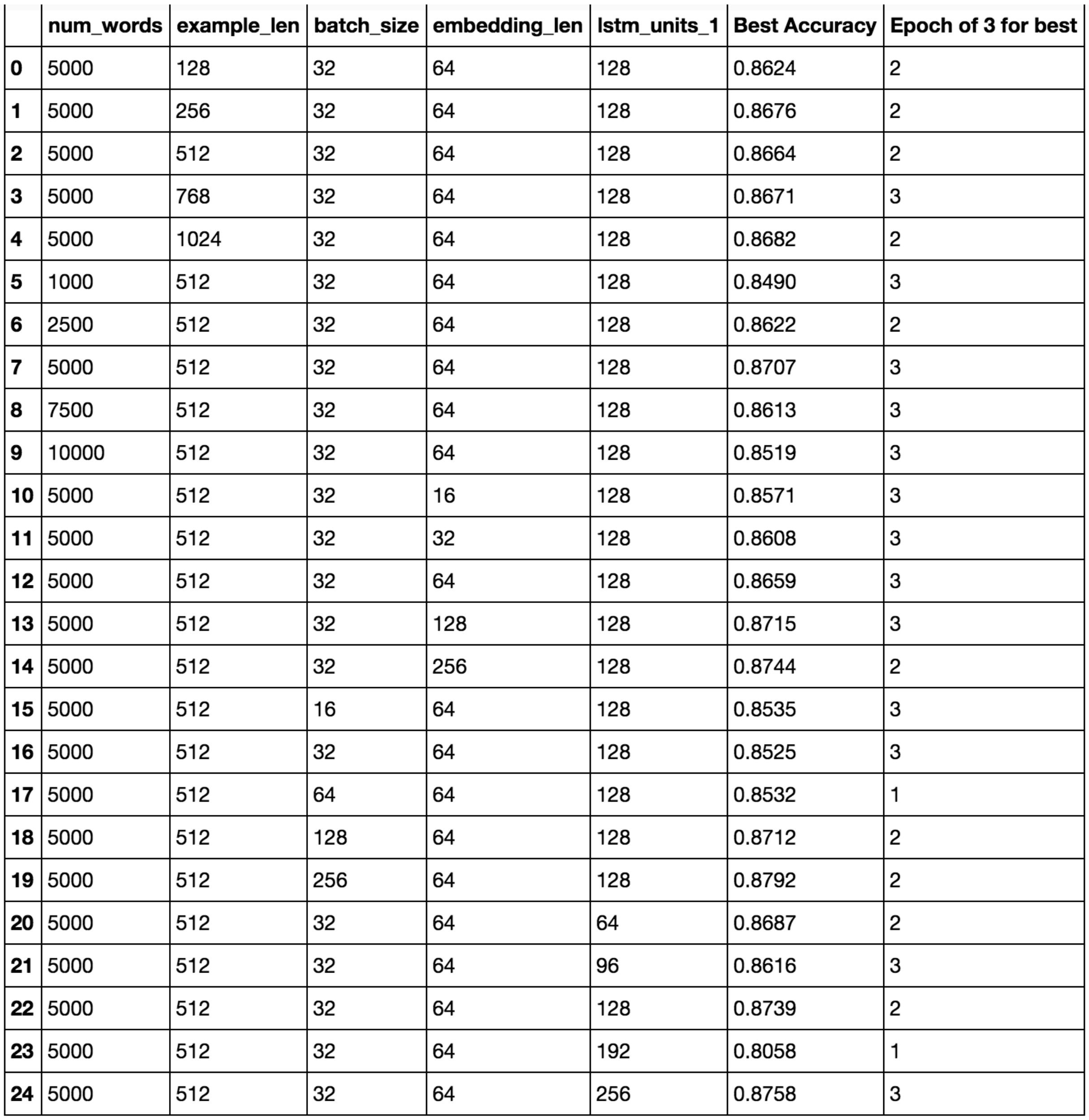
More nodes take longer to train and the accuracy rises over more epochs before plateauing. More nodes generally take longer to train and test.

In my chart below I ran many combinations at 3 epochs each to see if there was a pattern that could indicate a way to get to better results. I found no pattern. I may have needed to run more epochs to establish a pattern but running so many combinations would have taken too much more time.

In the chart indices 20-24 show changing the number of nodes. There was no pattern that came out in the test

The chart has variety of runs I did to check out how changing parameters affects the test accuracy.

The highest test accuracy I achieved was with a smaller number of nodes 64 and a larger embedding length – 128.



	num_words	example_len	batch_size	embedding_len	lstm_units_1	Best Accuracy	Epoch of 3 for best
0	5000	128	32	64	128	0.8624	2
1	5000	256	32	64	128	0.8676	2
2	5000	512	32	64	128	0.8664	2
3	5000	768	32	64	128	0.8671	3
4	5000	1024	32	64	128	0.8682	2
5	1000	512	32	64	128	0.8490	3
6	2500	512	32	64	128	0.8622	2
7	5000	512	32	64	128	0.8707	3
8	7500	512	32	64	128	0.8613	3
9	10000	512	32	64	128	0.8519	3
10	5000	512	32	16	128	0.8571	3
11	5000	512	32	32	128	0.8608	3
12	5000	512	32	64	128	0.8659	3
13	5000	512	32	128	128	0.8715	3
14	5000	512	32	256	128	0.8744	2
15	5000	512	16	64	128	0.8535	3
16	5000	512	32	64	128	0.8525	3
17	5000	512	64	64	128	0.8532	1
18	5000	512	128	64	128	0.8712	2
19	5000	512	256	64	128	0.8792	2
20	5000	512	32	64	64	0.8687	2
21	5000	512	32	64	96	0.8616	3
22	5000	512	32	64	128	0.8739	2
23	5000	512	32	64	192	0.8058	1
24	5000	512	32	64	256	0.8758	3

3. Replace LSTM with GRU and compare their performance.

GRU performance was about the same as LSTM with the same setup. It had slightly better results where I ran the same parameters. My best test accuracy from all runs was with GRU and 0.8944

Here is an example with 128 hidden units

```
dict_size=20000, example_length=512, embedding_length=128,  batch_size=32, epochs=15

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 512, 128)          2560000
_________________________________________________________________
gru_1 (GRU)                  (None, 128)               98688
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================

25000/25000 [==============================] - 935s - loss: 0.5010 - acc: 0.7541 -
val_loss: 0.3704 - val_acc: 0.8442
Epoch 2/15
25000/25000 [==============================] - 924s - loss: 0.3356 - acc: 0.8568 -
val_loss: 0.2737 - val_acc: 0.8864
Epoch 3/15
25000/25000 [==============================] - 904s - loss: 0.2209 - acc: 0.9134 -
val_loss: 0.2593 - val_acc: 0.8944
Epoch 4/15
25000/25000 [==============================] - 913s - loss: 0.1688 - acc: 0.9352 -
val_loss: 0.2717 - val_acc: 0.8909
Epoch 5/15
25000/25000 [==============================] - 920s - loss: 0.1272 - acc: 0.9538 -
val_loss: 0.3111 - val_acc: 0.8821
Epoch 6/15
25000/25000 [==============================] - 926s - loss: 0.1008 - acc: 0.9644 -
val_loss: 0.3314 - val_acc: 0.8846
Epoch 7/15
25000/25000 [==============================] - 916s - loss: 0.0765 - acc: 0.9741 -
val_loss: 0.3844 - val_acc: 0.8807
Epoch 8/15
25000/25000 [==============================] - 906s - loss: 0.0613 - acc: 0.9784 -
val_loss: 0.4389 - val_acc: 0.8696
Epoch 9/15
25000/25000 [==============================] - 908s - loss: 0.0463 - acc: 0.9836 -
val_loss: 0.4657 - val_acc: 0.8787
```

```
Epoch 10/15
25000/25000 [==============================] - 920s - loss: 0.0358 - acc: 0.9876 -
val_loss: 0.5233 - val_acc: 0.8756
Epoch 11/15
25000/25000 [==============================] - 843s - loss: 0.0295 - acc: 0.9893 -
val_loss: 0.5759 - val_acc: 0.8752
Epoch 12/15
25000/25000 [==============================] - 772s - loss: 0.0282 - acc: 0.9901 -
val_loss: 0.6120 - val_acc: 0.8713
Epoch 13/15
25000/25000 [==============================] - 767s - loss: 0.0198 - acc: 0.9935 -
val_loss: 0.6666 - val_acc: 0.8771
Epoch 14/15
25000/25000 [==============================] - 773s - loss: 0.0210 - acc: 0.9928 -
val_loss: 0.6514 - val_acc: 0.8734
Epoch 15/15
25000/25000 [==============================] - 775s - loss: 0.0193 - acc: 0.9934 -
val_loss: 0.6421 - val_acc: 0.8732
25000/25000 [==============================] - 200s
```

And another GRU example with 128 hidden units

```
dict_size=20000, example_length=512, embedding_length=128,  batch_size=32, epochs=15

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 512, 128)          2560000
_________________________________________________________________
dropout_1 (Dropout)          (None, 512, 128)          0
_________________________________________________________________
gru_1 (GRU)                  (None, 128)               98688
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
25000/25000 [==============================] - 1305s - loss: 0.5115 - acc: 0.7475 -
val_loss: 0.3811 - val_acc: 0.8444
Epoch 2/15
25000/25000 [==============================] - 1227s - loss: 0.3575 - acc: 0.8498 -
```

```
val_loss: 0.3368 - val_acc: 0.8539
Epoch 3/15
25000/25000 [==============================] - 1456s - loss: 0.2065 - acc: 0.9202 -
val_loss: 0.2895 - val_acc: 0.8784
Epoch 4/15
25000/25000 [==============================] - 1316s - loss: 0.1224 - acc: 0.9551 -
val_loss: 0.3285 - val_acc: 0.8743
Epoch 5/15
25000/25000 [==============================] - 1372s - loss: 0.0720 - acc: 0.9762 -
val_loss: 0.4011 - val_acc: 0.8704
Epoch 6/15
25000/25000 [==============================] - 1472s - loss: 0.0430 - acc: 0.9853 -
val_loss: 0.4521 - val_acc: 0.8607
Epoch 7/15
25000/25000 [==============================] - 1308s - loss: 0.0301 - acc: 0.9898 -
val_loss: 0.5105 - val_acc: 0.8640
Epoch 8/15
25000/25000 [==============================] - 1114s - loss: 0.0180 - acc: 0.9946 -
val_loss: 0.6100 - val_acc: 0.8609
Epoch 9/15
25000/25000 [==============================] - 1098s - loss: 0.0144 - acc: 0.9956 -
val_loss: 0.6792 - val_acc: 0.8538
Epoch 10/15
25000/25000 [==============================] - 1116s - loss: 0.0123 - acc: 0.9959 -
val_loss: 0.6622 - val_acc: 0.8522
Epoch 11/15
25000/25000 [==============================] - 1097s - loss: 0.0108 - acc: 0.9962 -
val_loss: 0.7350 - val_acc: 0.8570
Epoch 12/15
25000/25000 [==============================] - 1119s - loss: 0.0116 - acc: 0.9959 -
val_loss: 0.7120 - val_acc: 0.8472
Epoch 13/15
25000/25000 [==============================] - 1096s - loss: 0.0063 - acc: 0.9980 -
val_loss: 0.7980 - val_acc: 0.8534
Epoch 14/15
25000/25000 [==============================] - 1119s - loss: 0.0054 - acc: 0.9981 -
val_loss: 0.8704 - val_acc: 0.8454
Epoch 15/15
```

```
25000/25000 [==============================] - 1184s - loss: 0.0061 - acc: 0.9978 -
val_loss: 0.9434 - val_acc: 0.8522
25000/25000 [==============================] - 359s
```

Extra credits (5pts) Try to use pre-trained word embeddings to initialize the embedding layer and see how that changes the performance.

I did the glove glove.6B.100d.txt pre-trained word embeddings

https://nlp.stanford.edu/projects/glove/

http://nlp.stanford.edu/data/glove.6B.zip

Fitting to the training data is happening much slower. It looks like I'd need to run many more epochs before reaching best accuracy.

Initial test accuracy is much lower as is initial test accuracy. This is likely because the initial vectors are coming out of a different training set. The word relationships are different. It takes longer to find relationship mappings in the sample text that match the pre-trained word embeddings. The glove data set comes from Wikipedia and news wires. The word set and relationships may be much different than the imdb word set.

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 100)          8858500
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================

Epoch 1/15
25000/25000 [==============================] - 409s - loss: 0.6871 - acc: 0.5371 -
val_loss: 0.6752 - val_acc: 0.5704
Epoch 2/15
25000/25000 [==============================] - 405s - loss: 0.6733 - acc: 0.5712 -
val_loss: 0.6639 - val_acc: 0.5877
Epoch 3/15
```

```
25000/25000 [==============================] - 399s - loss: 0.6521 - acc: 0.6060 -
val_loss: 0.6188 - val_acc: 0.6650
Epoch 4/15
25000/25000 [==============================] - 370s - loss: 0.6106 - acc: 0.6618 -
val_loss: 0.5822 - val_acc: 0.6916
Epoch 5/15
25000/25000 [==============================] - 277s - loss: 0.5856 - acc: 0.6816 -
val_loss: 0.5574 - val_acc: 0.7156
Epoch 6/15
25000/25000 [==============================] - 276s - loss: 0.5585 - acc: 0.7092 -
val_loss: 0.5274 - val_acc: 0.7305
Epoch 7/15
25000/25000 [==============================] - 276s - loss: 0.5341 - acc: 0.7252 -
val_loss: 0.4976 - val_acc: 0.7546
Epoch 8/15
25000/25000 [==============================] - 276s - loss: 0.5107 - acc: 0.7454 -
val_loss: 0.5159 - val_acc: 0.7416
Epoch 9/15
25000/25000 [==============================] - 275s - loss: 0.4868 - acc: 0.7610 -
val_loss: 0.4473 - val_acc: 0.7873
Epoch 10/15
25000/25000 [==============================] - 274s - loss: 0.4582 - acc: 0.7802 -
val_loss: 0.4503 - val_acc: 0.7834
Epoch 11/15
25000/25000 [==============================] - 274s - loss: 0.4359 - acc: 0.7974 -
val_loss: 0.4201 - val_acc: 0.8038
Epoch 12/15
25000/25000 [==============================] - 275s - loss: 0.4183 - acc: 0.8021 -
val_loss: 0.4064 - val_acc: 0.8112
Epoch 13/15
25000/25000 [==============================] - 275s - loss: 0.4014 - acc: 0.8162 -
val_loss: 0.3973 - val_acc: 0.8188
Epoch 14/15
25000/25000 [==============================] - 274s - loss: 0.3883 - acc: 0.8246 -
val_loss: 0.3976 - val_acc: 0.8165
Epoch 15/15
25000/25000 [==============================] - 274s - loss: 0.3715 - acc: 0.8311 -
val_loss: 0.3824 - val_acc: 0.8264
25000/25000 [==============================] - 108s
```