**Using Autoencoders to Compress Medical Literature**

**Background:**

Machine learning, which is a subset of artificial intelligence, is one of the most published and researched fields in computer science. Machine learning first came about in 1950 when Alan Turing proposed the Turing Test, a test to determine whether a computer was truly “intelligent” or not. If a computer can fool a human into thinking that the computer is a human, then the computer passed the Turing Test.

Since Alan Turing proposed the Turing Test, machine learning has thrived and advanced in ways that he never imaged. The first computer program to be considered a machine-learning algorithm was introduced in 1952 by Arthur Samuel.This simple program took note of the most crucial moves during games of checkers it played and in subsequent games tried to use those moves. Next large leap in machine learning was the invention of the perceptron by Frank Rosenblatt in 1957. The perceptron was a simplified mathematical model of a neuron in a brain. Perceptrons evolved into the nodes of neural networks. However, even though the nodes of neural networks resemble the neurons found inside of brains, neural networks are not trying to mechanistically imitate the processes of the brain.

Between 1960 and 1990 the field of machine learning developed many important algorithms. However, the most important change that happened in this time was the shift from knowledge-based approaches to data-driven approaches. Computer scientists discovered that computers perform best when they try to mathematically optimize a model instead of picking out the most influential behavior as had been done with Samuel’s algorithm in the 50s. This meant that now most machine-learning algorithms converted a problem into numerical form, usually a vector or matrix-based representation, and tried to minimize an equation given this input vector and the “answer” to the vector. This minimization is done by “training” the algorithm on many example problems and their solutions. For example, a machine learning checkers algorithm may be given 500 million examples of checkers games, all of which are represented as vectors. It would then try to minimize some function by adjusting parameters. Each time it gets a move wrong in the checkers games it is simulating, one of the parameters is pushed slightly in one direction to attempt to reduce the amount of errors.

Ever since this realization the field of machine learning has blossomed. There have been many astounding feats of machine learning that have occurred just within the past year. The machine learning algorithm AlphaGo beat a human master of Go, which is considered the hardest game for a computer to play due to the seemingly human intuition needed to make good moves. Moreover, image classification algorithms have achieved error rates of less than 7% (for perspective, humans who have “trained” themselves for a week on the same task had an error rate of about 5%). As well, machine-learning algorithms have far surpassed us in games such as chess and even Jeopardy.

Up until the 21st century no individual or company had the computational power to effectively train a neural network. However, a breakthrough in 2011 changed this. Google Brain developed an image classification program with accuracy unheard of using the previous machine learning techniques. This breakthrough opened the floodgates for the development of neural networks. Since then neural networks have only become more and more researched. Now there are many competitions each year held for computers to do tasks like image classification and speech synthesis. Companies and institutions have released code to allow an individual to easily develop and research neural networks. The most recent of these releases is called TensorFlow. Developed by Google and released in the fall of 2015, TensorFlow is a complex and efficient framework for developing neural networks. TensorFlow allows a neural network to be trained and run on nearly anything, from a phone to a supercomputer.

There are many similarities between a neural network and the brain. Nonetheless, the two are quite different.

First, what does a biological neuron do? A biological neuron takes inputs from other neurons in the form of an electrical charge. The charges from each of the inputs are combined in the nucleus of the neuron and then, only if the charge in the nucleus reaches a certain threshold, the neuron will release some electricity itself. That neuron will act as an input to many other neurons. This large network of neurons is what forms the brain.

Artificial neural networks are much similar. They take inputs in the form of numbers, usually between 0 and 1. They sum up all of their inputs and evaluate them on an activation function. The biological neural network’s activation function called a stepper function. It can be easily defined as:

a(x) = 1 if x > t

0 otherwise

Where t is the threshold needed to activate the function. There are a number of problems with this activation function for artificial neural networks, including their simplicity and the function’s inability to be fully differentiated. This is why a different class of activation functions is often used. These are called sigmoid functions. They all resemble a stretched out S shape. The most common sigmoid function used in artificial neurons is the hyperbolic tangent function.

Now each of the neurons is connected to make an artificial neural network (ANN). First there is an input layer of nodes. These nodes are fed the values that represent the problem. For example, if the neural network were classifying 28 pixel by 28 pixel gray scale handwritten digits then the input layer would be composed of 784 numbers from 0-1 representing the 784 pixels of the image and the normalized value of that pixel. These nodes do not perform any activation functions, they simply output whatever their input is. Next each input node leads into each artificial neuron in the hidden layer. In the most basic ANN there is only 1 layer of neurons. The number of neurons in this hidden layer is often determined by a lot of guess and check work. The amount of neurons varies greatly between tasks. The tuning of this parameter along with a number of other parameters involved in training and testing a neural network is often the most time consuming step in building, training, and testing a neural network. As well, each of the connects between the input layer and the hidden layer have a weight associated with it. These weights are what are adjusted and changed. The weight acts as a multiplier for the “importance” of the input. For example, if a connection’s weight is 0.5 and its input is 0.9 then the neuron would get an input of 0.9 x 0.5 = 0.45.

Finally each neuron in the hidden layer is connected to every neuron in the output layer. Again these connections have weights associated with them. Once the output layer has evaluated the activation function on their input, they output the answer in numerical form.

The final step in constructing a neural network is the “training.” Through linear algebra the weights connecting neurons are gradually changed to try to find the minimum value of a function. This process is the heart of neural networks, but is also by far the most in-depth. First, it can take as many as 500 million examples for a neural network to be fully trained. Second, there are many ways to measure how far off from the answer the neural network currently is and many ways to choose which and by how much to adjust weights. As well, a neural network could be “trapped” in a local minimum, not the global minimum. Although a process called dropout can often combat this problem, there are still many other things that can go wrong in training neural networks.

I hope to use machine learning in order to aid doctors in making diagnosis by combining traditional classification algorithms, which involve looking only at a patient’s data, with information gathered from medical literature. However, in order to do this I first have to gather data from medical literature in a format which a neural network downstream will have a good chance at being able to use to improve its predictions. Thus, I have to create an autoencoder neural network in order to properly compress and encode the given medical literature.

There has been extensive research in the field of machine learning and neural networks over the past approximately 20 years. There have been neural networks that can recognize and categorize pictures with greater accuracy and speed than a human and a question answering system that has outperformed humans in Jeopardy. This question answering system, dubbed Watson, is a project created by IBM. Watson has since evolved to not only read through dictionaries and textbooks to answer Jeopardy questions, but also to read through medical literature to help doctors in recommending treatment. Watson is currently being used by multiple hospitals to help doctors in diagnosing and treating patients. Although IBM has published relatively little on Watson, they have published their work on unsupervised entity relation, which is an effective way to keep track of relationships between entities when inserting them into a database (Kalyanpur and Murdock 2015). As well, they have published on one of their strategies of searching through a patient’s record in order to retrieve the most relevant information (Prager et al. 2017).

More specifically, extensive research has been done on information retrieval and question answering neural networks. The task that I face will likely require some elements of both information retrieval and question answering architectures. There have been many proposed models, however, Lew et al. have shown that strong similarity measures and indexing tools have proven to be some of the most effective general strategies (2006). Furthermore, Zhang et al. have shown that the recurrent neural network currently is the best model for information retrieval, but that the convolutional neural network shows a lot of promise in this area (2016). Recurrent neural networks have also been proven effective in the area of question answering; Iyyer et al. demonstrated an effective question answering recurrent neural network that uses an accompanying parsing algorithm (2014). Andreas et al. describe a recurrent neural network that first puts data into a database then, given a question, parses it and generates a query to get the relevant information from the database. This significantly more structured approach allowed more freedom to explore the information that the model “knew” (2016).

However, given the highly open-ended nature of the data, I believe that such methods would perform poorly in the final task of diagnosing patients. Thus, I have settled on an architecture which is composed of two parts, an autoencoder, which has been used previously to encode text for further use by machine learning algorithms (Yang et al. 2017), and a classification algorithm which will use the encoded literature along with patient data. The classification network is not the focus of this project.

**Research Question:**

Can medical literature improve traditional machine learning classification neural networks for predicting diagnoses.

**Hypothesis:**

I believe that I will be able to successfully encode medical literature for appropriate use in further neural networks.

**Abstract:**

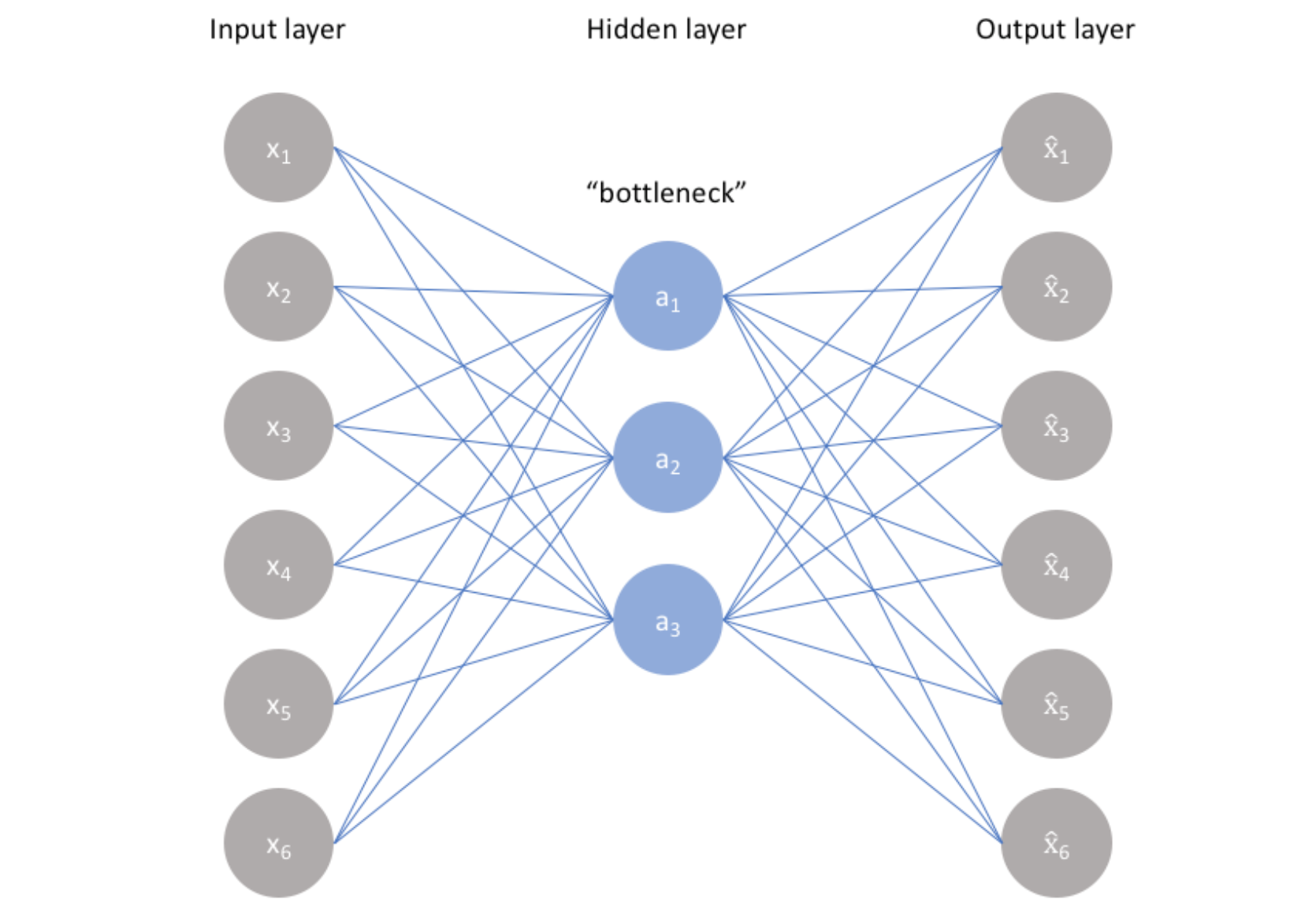
Computers are the solution to the massive amount of data being published every day. Machine learning, which is a subset of artificial intelligence, learns much like a human; by looking at large amounts of data, the algorithm is able to make generalizations and rules about this data. This means that when it is fed data it has never seen before, the algorithm is able to return the conclusion it has drawn from that data given the generalizations and rules it figured out during training. For example, approximately 10% of all papers on PubMed are on intestinal cancers. This means that if the algorithm were to be trained on these millions of papers (and it could continue to be trained using the approximately 500 papers a day on this subject), the algorithm might be able to look at a patient’s records and determine something such as whether the patient needs surgery, and if so, what type of surgery they should get.

I hope to use machine learning in order to aid doctors in making diagnosis by combining traditional classification algorithms, which involve looking only at a patient’s data, with information gathered from medical literature. However, in order to do this I first have to gather data from medical literature in a format which a neural network downstream will have a good chance at being able to use to improve its predictions. Thus, I have to create an autoencoder neural network in order to properly compress and encode the given medical literature.

**Methods:**

In my project, I plan to use machine learning, more specifically neural networks, to have a computer “read” through medical literature on a particular topic and then the computer will be able to recommend a treatment given finite options.

I have split this task into two parts, “reading” through the literature and seeing if the data gleaned from the papers helps in diagnosing patients. The focus of this project is on the former half. This reading is done through a method called autoencoding. Autoencoding is a form of machine-learning based compression in which a neural network is trained to compress data by minimizing the compression loss. In practice, this means that an hourglass shaped neural network is used.



**Figure 2.** A diagram of an autoencoder with an 6 input nodes, one hidden layer, labeled the bottleneck layer, with 3 nodes, and an output layer of the same size as the input (Jordan).

When text is fed into the input layer, the neural network is forced to find a way to represent this text in a more compact way so that it can pass through the bottleneck. The compressed text is then uncompressed by the neural network and the output is compared to the input. By minimizing the difference between the output and the input, the neural network naturally becomes very good at compressing the given input data. This differs from traditional compression algorithms as this method creates a compression algorithm hand tailored to the given inputs it was trained on, so in essence instead of simply performing some mathematical tricks on the data, the neural network extracts the most important words from the original text.

However, autoencoders can only take numbers, not text, as an input. Thus, an intermediary model, called word2vec, is used to convert the text to numbers. This model, written by Google, fits a vector corresponding to each word’s meaning. For example, each word might be represented by a 512-dimensional vector. After training the word2vec, the vectors associated with each word will have semantic meaning. The names of cities or people would be close in vector space and other semantic relationships would be represented by these vectors.

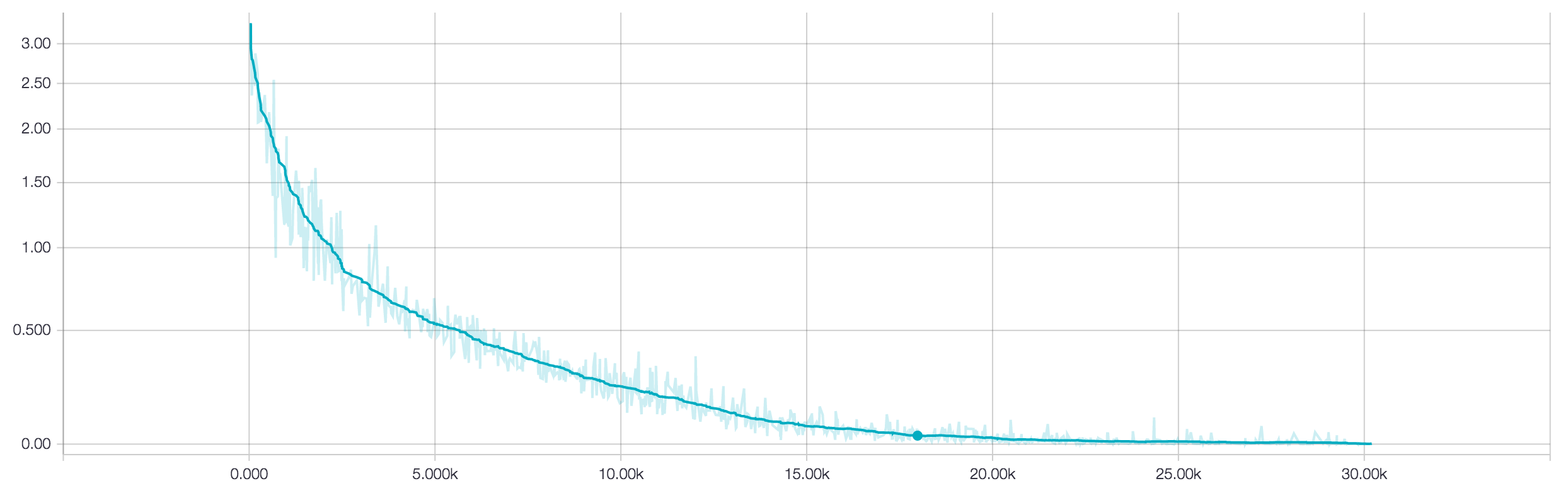
This compressed text can then be used downstream. By adding this compressed text to a traditional classification neural network, I can determine if this helps or hurts classification accuracy.

In summary:

* Scrape medical literature on colon cancer from arXiv and extract the text from the PDF
* Write a basic autoencoder using TensorFlow and Python 3
* The autoencoder has 2 layers, the first layer of size 256 and the second of size 128 with a batch size of 256 and input size of 512
* Train TensorFlow’s word2vec model on the extracted text with encoding size of 768
* Train the autoencoder using batches from the word2vec
* Record loss using TensorFlow’s logging program, TensorBoard
* Record the cosine similarity between a few semantically similar words to TensorBoard as a sanity check to the training process

**Results:**

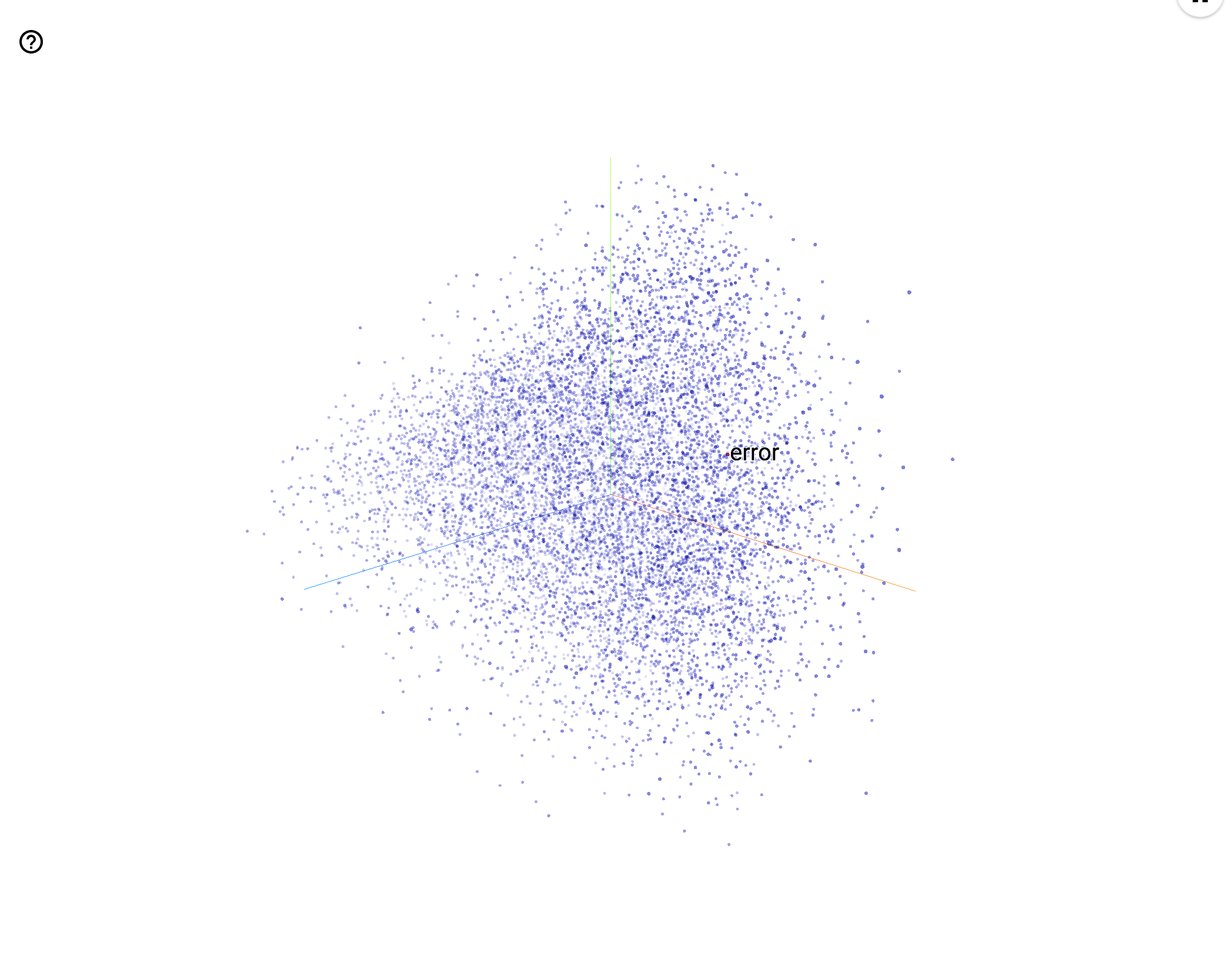
Loss Curve



**Figure 3.** This chart shows the loss over time during the training of the autoencoder. The x axis shows time in training steps while the y axis shows loss computed with the Noise Contrastive Estimation loss formula. This run was done with a learning rate of 0.01.

Since there is no objective measurement of accuracy for autoencoders, I instead measured the cosine similarity between some semantically similar words, and this followed the same pattern as the loss.

Embedding visualization



**Figure 4.** Using Tensorboard’s embedding visualizer feature, you can take the results of the word2vec and visualize them in space. Since word2vec accociates a (high-dimensional vector with each word, this visualizer tool runs PCA to reduce these vectors down to 3 dimensions. Thus, each dot represents a word and it’s position in space has a vague correlation with its semantic meaning (the correlation is only vague because PCA was used to take these vectors down from 768 to 3 dimensions, losing a lot of information in the process).

**Discussion**

With little to compare this data to, I am unable to draw a certain conclusion. However, the loss curve did follow an expected, predictable, and non-erratic pattern, which suggests that the training was at least stable. The fact that the cosine similarities of sampled words decreased over time as well only confirmed this. Of course, this says nothing about the effectiveness of the final product, only that the training process has been tuned and dialed in.

**Conclusion**

Currently, it is unclear whether my hypothesis was supported. I did make an autoencoder, but I currently have neither the means nor time to examine its usefulness in full. In the future I hope to use this autoencoder in conjunction with a classifier in order to test if the addition of the medical literature will help in classification. With data gathered from this, I may be able to more accurately assess whether the autoencoder was able to encode the text data to a sufficient standard.

**Literature Cited:**

Anguita, D., A. Boni, and S. Ridella. "A Digital Architecture for Support Vector Machines: Theory, Algorithm, and Fpga Implementation." *IEEE Transactions on Neural Networks* 14, no. 5 (2003): 993-1009. Accessed October 30, 2016. doi:10.1109/tnn.2003.816033.

Andreas J, Rohrbach M, Darrell T, Klein D. 2016. Learning to Compose Neural Networks for Question Answering. [accessed 2017 Apr 9]; https://arxiv.org/pdf/1601.01705.pdf

Arpathy, K. "CS231n Convolutional Neural Networks for Visual Recognition." CS231n Convolutional Neural Networks for Visual Recognition. Accessed October 30, 2016. http://cs231n.github.io/convolutional-networks/.

Haykin, Simon S. *Neural Networks: A Comprehensive Foundation*. New York: Macmillan, 1994.

Iyyer M, Boyd-Graber J, Claudino L, Socher R, Daumé H. 2014. A Neural Network for Factoid Question Answering over Paragraphs.Empirical Methods in Natural Language Processing [accessed 2017 Apr 9]; https://pdfs.semanticscholar.org/2872/52a5fb2f2a9e311eebf06e5ac49eb52eaadc.pdf

Jaeger, Herbert. "A Tutorial on Training Recurrent Neural Networks, Covering BPPT, RTRL, EKF and the "echo State Network" Approach." *Fraunhofer Institute for Autonomous Intelligent Systems*, December 2013. Accessed October 30, 2016. http://minds.jacobs-university.de/sites/default/files/uploads/papers/ESNTutorialRev.pdf.

Jordan, Jeremy. *Introduction to Autoencoders*. 2018. Web. 31 Dec 2018.

Kalchbrenner, Nal, Edward Grefenstette, and Phil Blunsom. "A Convolutional Neural Network for Modelling Sentences." *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2014. doi:10.3115/v1/p14-1062.

Kalyanpur Aditya, Murdock J. 2015. Unsupervised Entity-Relation Analysis in IBM Watson. Proceedings of the Third Annual Conference on Advances in Cognitive Systems [accessed 2017 Apr 9]; http://www.cogsys.org/papers/ACS2015/article12.pdf?cm\_mc\_uid=32656084035014917521018&cm\_mc\_sid\_50200000=1491755807

Lew MS, Djeraba C, Jain R. 2006. Content-Based Multimedia Information Retrieval: State of the Art and Challenges. ACM Transactions on Multimedia Computing, Communications and Applications [accessed 2017 Apr 9]; 2:1–19. https://pdfs.semanticscholar.org/d0c8/354b7710f50a2e94d28937bca26ab8de456c.pdf

Miller, Steven. *Small Neural Network*. 2015. Web. 9 Apr. 2017.

Nielsen M. 2017 Jan. Why are deep neural networks hard to train? erastz, G Squared Capital, TinEye, VisionSmarts; [accessed 2017 Apr 9]. http://neuralnetworksanddeeplearning.com/chap5.html

Powers DM. 2011. EVALUATION: FROM PRECISION, RECALL AND F-MEASURE TO ROC, INFORMEDNESS, MARKEDNESS & CORRELATION.Empirical Methods in Natural Language Processing [accessed 2017 Apr 9]; 2:37–63. http://dspace2.flinders.edu.au/xmlui/bitstream/handle/2328/27165/Powers%20Evaluation.pdf?sequence=1

Prager JM, Liang JJ, Devarakonda MV. 2016. SemanticFind: Locating What You Want in a Patient Record, Not Just What You Ask For. IBM Research Division [accessed 2017 Apr 9]; http://domino.research.ibm.com/library/cyberdig.nsf/papers/93F3B60E55AE2D42852580530067E897/$File/rc25627.pdf

Rong X. 2014. word2vec Parameter Learning Explained. [accessed 2017 Apr 9]; https://arxiv.org/pdf/1411.2738.pdf

Rosenblatt, F. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review* 65, no. 6 (1958): 386-408. doi:10.1037/h0042519.

Samuel, A. L. "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development* 3, no. 3 (1959): 210-29. doi:10.1147/rd.33.0210.

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529, no. 7587 (2016): 484-89. doi:10.1038/nature16961.

"TensorFlow Is an Open Source Software Library for Machine Intelligence." TensorFlow. Accessed October 30, 2016. https://www.tensorflow.org/.

"What I Learned from Competing against a ConvNet on ImageNet." What I Learned from Competing against a ConvNet on ImageNet. September 2, 2014. Accessed October 30, 2016. http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/.

Yang Z, Hu Z, Salakhutdinov R, Berg-Kirkpatrick T. 2017. Improved Variational Autoencoders for Text Modeling using Dilated Convolutions. [accessed 2018 Dec 31]; https://arxiv.org/pdf/1702.08139

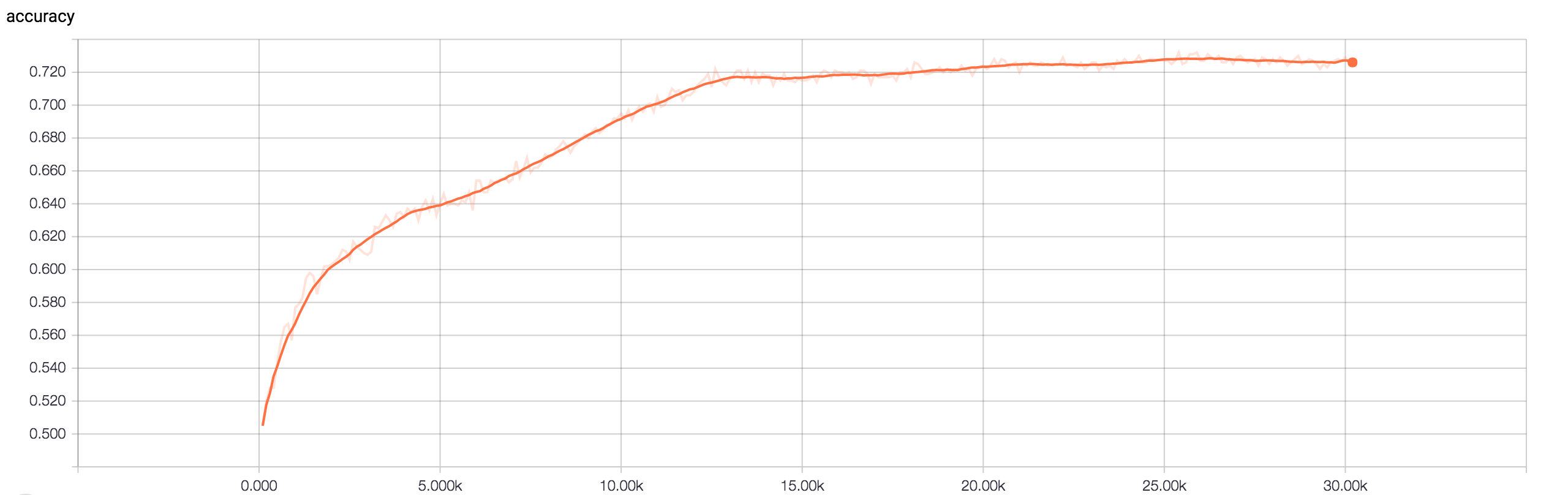
Zhang Y, Rahman M, Brayla A, Dang B, Chang H-L, Kim H, McNamara Q, Angert A, Banner E, Khetan V, et al. 2016. Neural Information Retrieval: A Literature Review. Clinical Orthopaedics and Related Research [accessed 2017 Apr 9]; 1611.06792. https://arxiv.org/pdf/1611.06792.pdf

All of the following data was collected using TensorFlow in Python 3 and the graphs were generated using TensorBoard. These graphs show the results of multiple experiments regarded text analysis and encoding using neural networks. Most of this research was conducted to formulate an architecture for the full project and establish the effectiveness of certain specific algorithms.

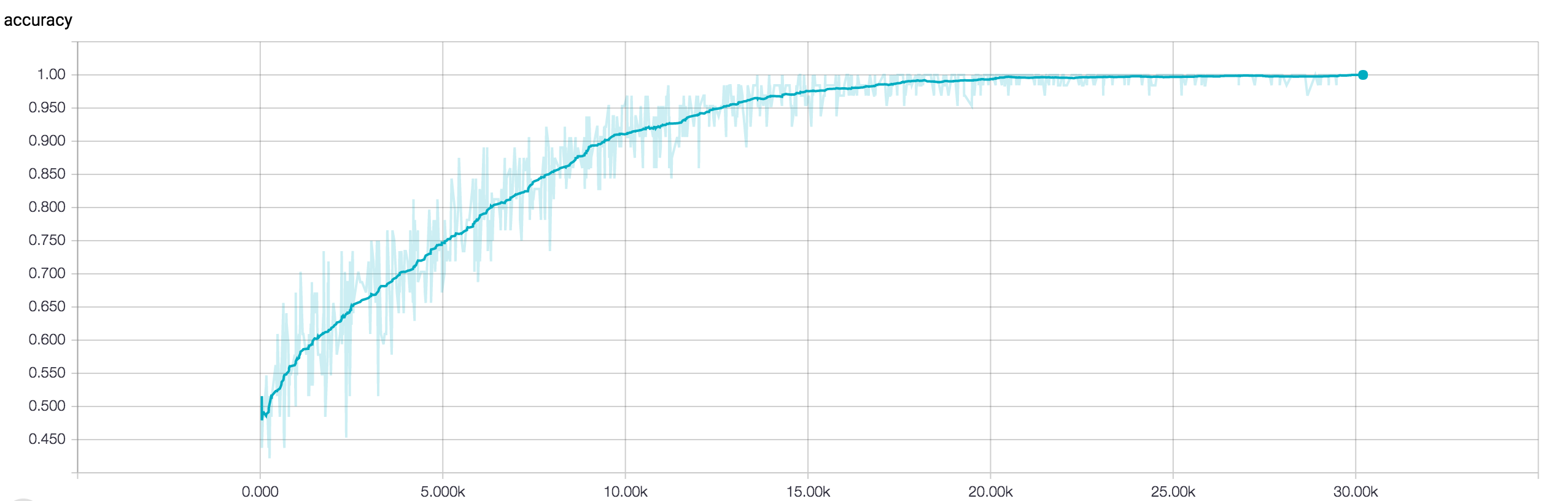
**Accuracy graphs**

On all of the graphs, the x-axis represents percent accuracy while the y-axis represents number of training steps. On each of the training graphs, the accuracy was calculated via self-validation (accuracy was tested using the training data), which is why they all approach 100%. On all of the development graphs, accuracy was calculated on a separate validation set. Unfortunately I was only able to get training accuracy graphs for the DCNN. Each of these graphs are smoothed using 60% exponential smoothing. The hyperparameters (except for the number of epochs) are as follows:

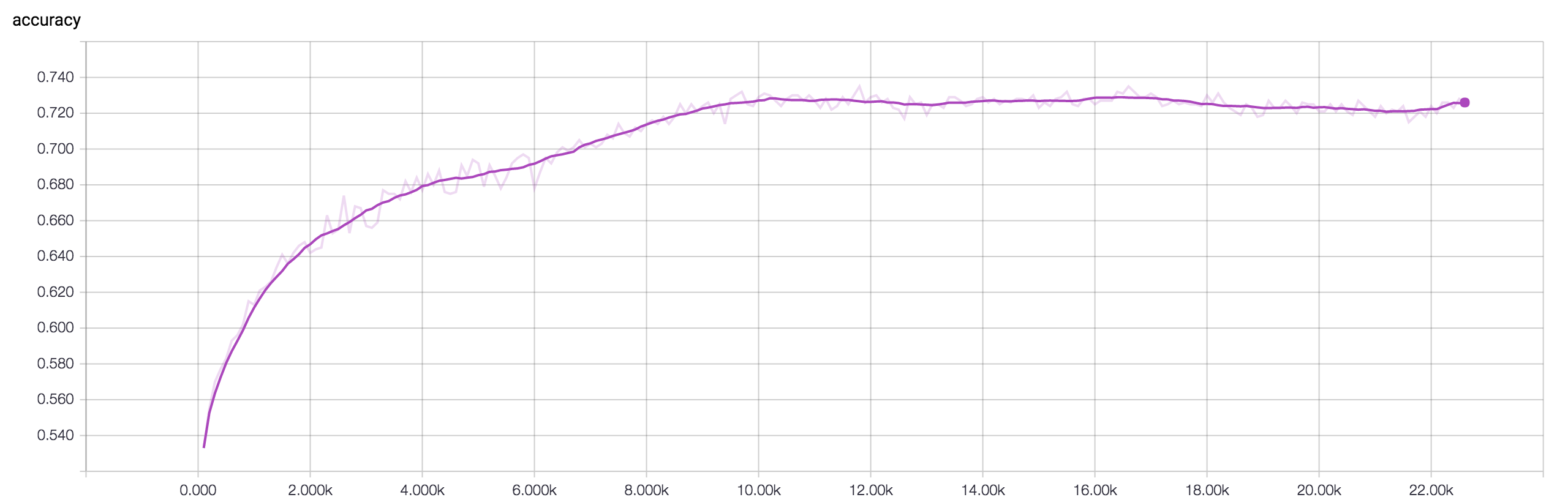
* 128 dimensional character embedding
* 3 filters of sizes 3, 4, and 5
* 128 filters (per size)
* 0.5 dropout-keep probability
* Batch size of 64



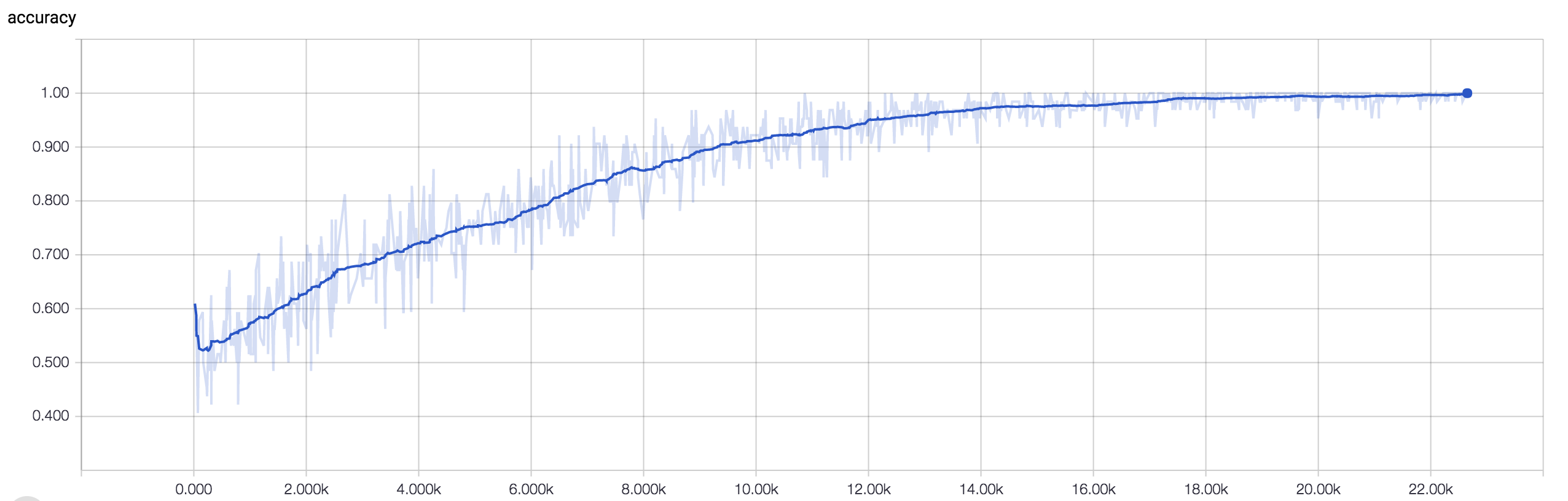
**Figure 1.** The development accuracy graph of a CNN run that was 200 epochs long.



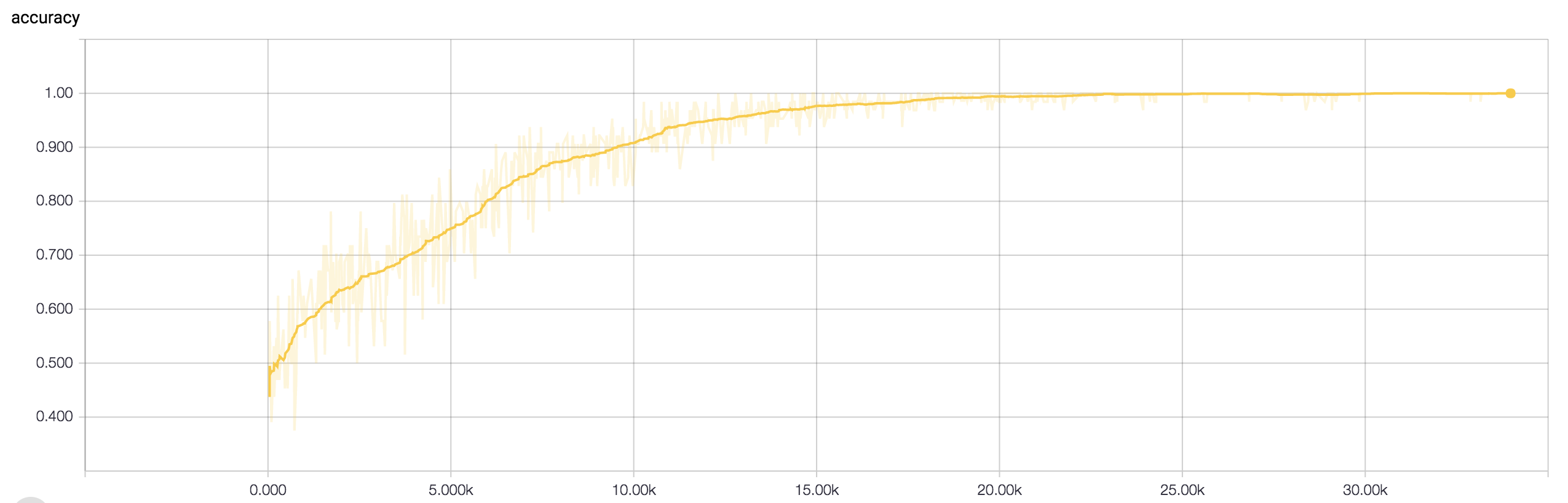
**Figure 2.** The training accuracy graph of a CNN run that was 200 epochs long.



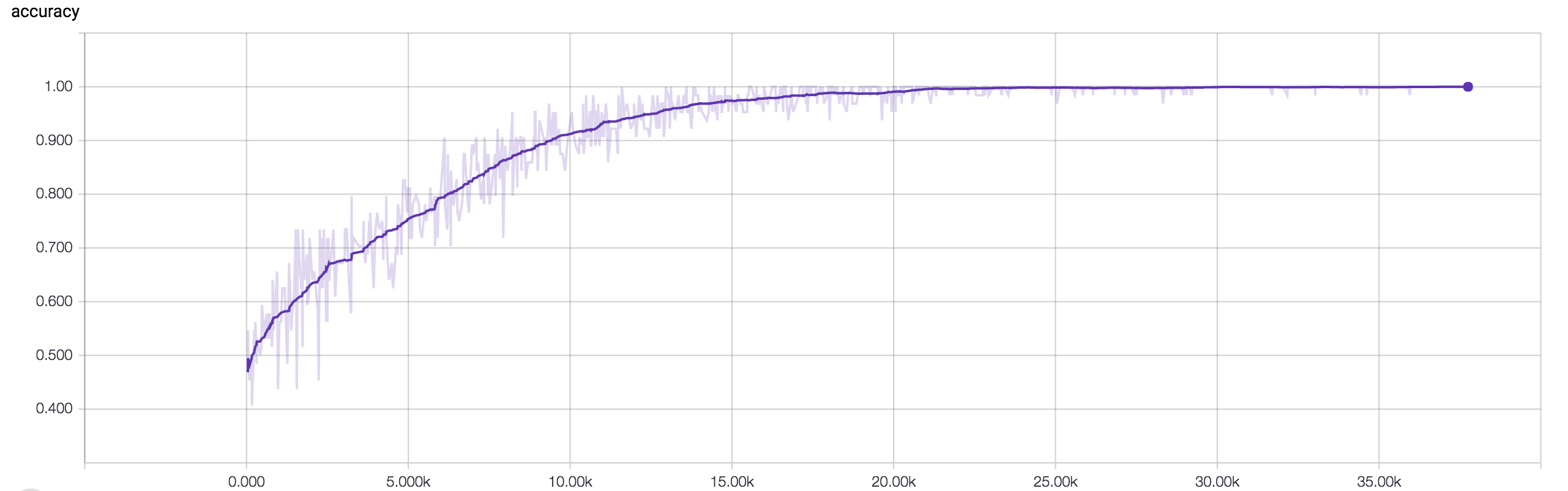
**Figure 3.** The development accuracy graph of a CNN run that was 150 epochs long.



**Figure 4.** The training accuracy graph of a CNN run that was 150 epochs long.



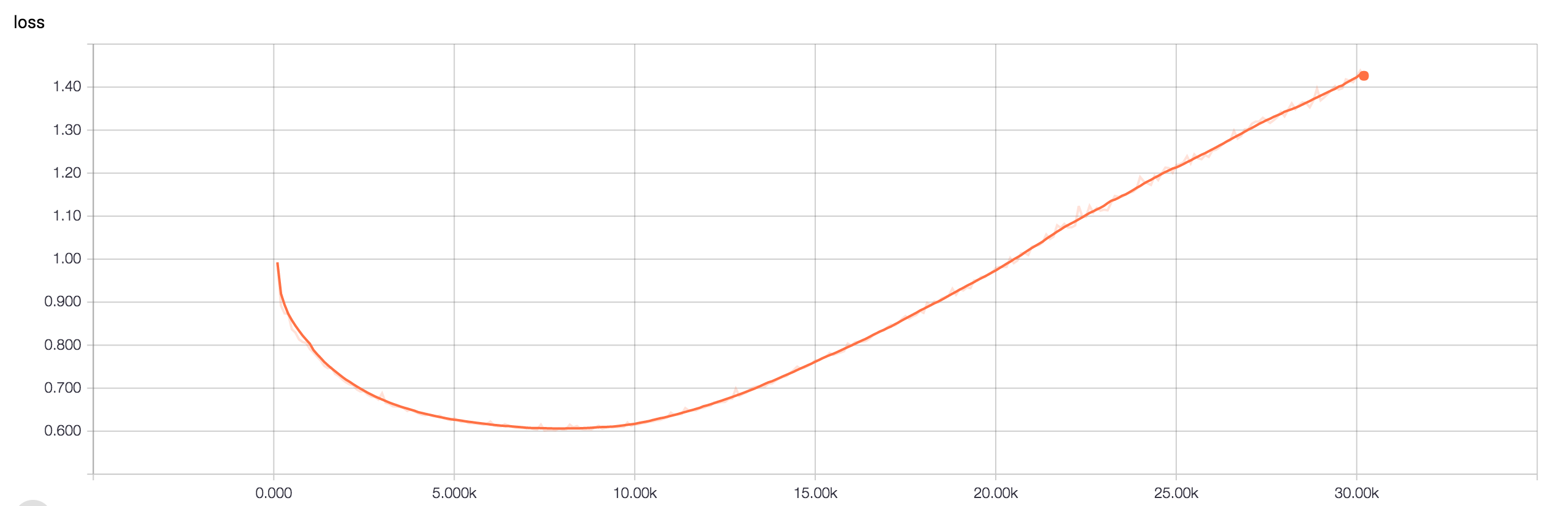
**Figure 5.** The training accuracy graph of a DCNN run that was 225 epochs long.



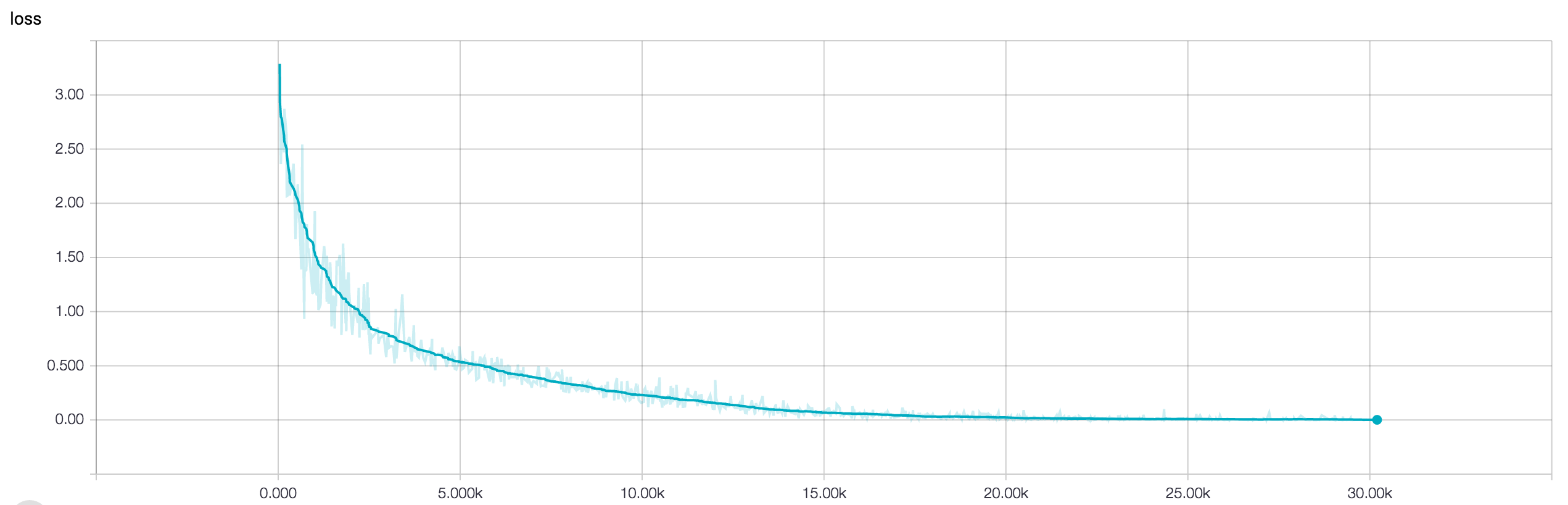
**Figure 6.** The training accuracy graph of a DCNN run that was 250 epochs long.

**Loss graphs**

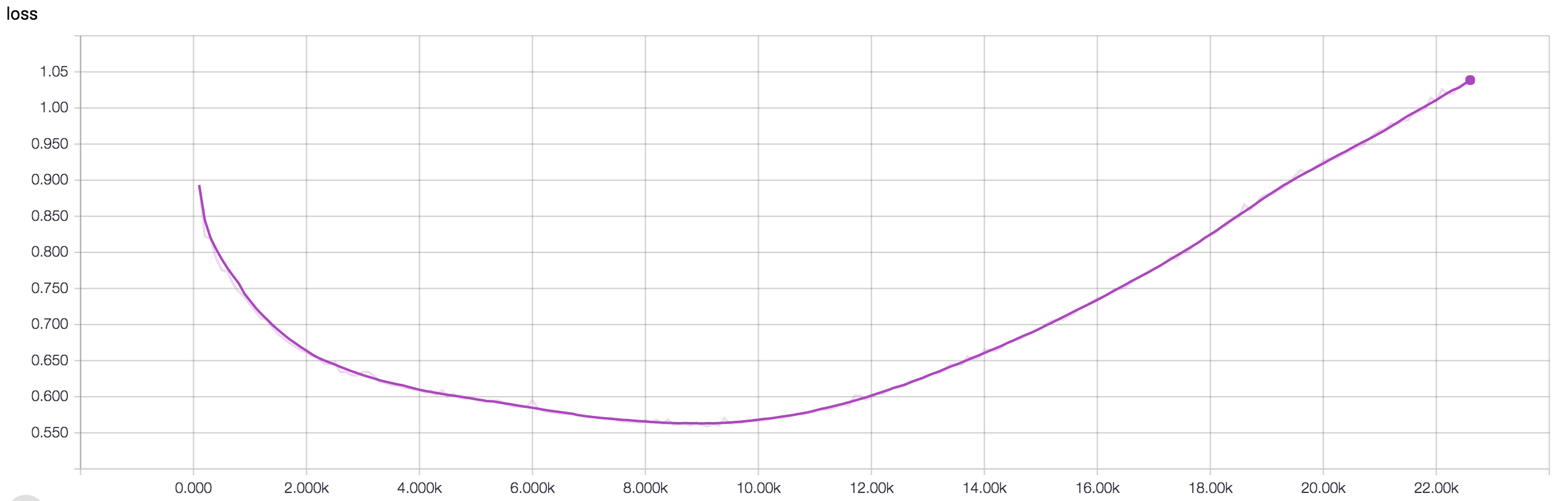
All of the loss graphs were generated using TensorFlow and TensorBoard during the same runs as the accuracy graphs. The loss was calculated using softmax with logits. Again training graphs were self-validated while development graphs used an independent validation set. These are not the same as the loss graphs shown in the results.



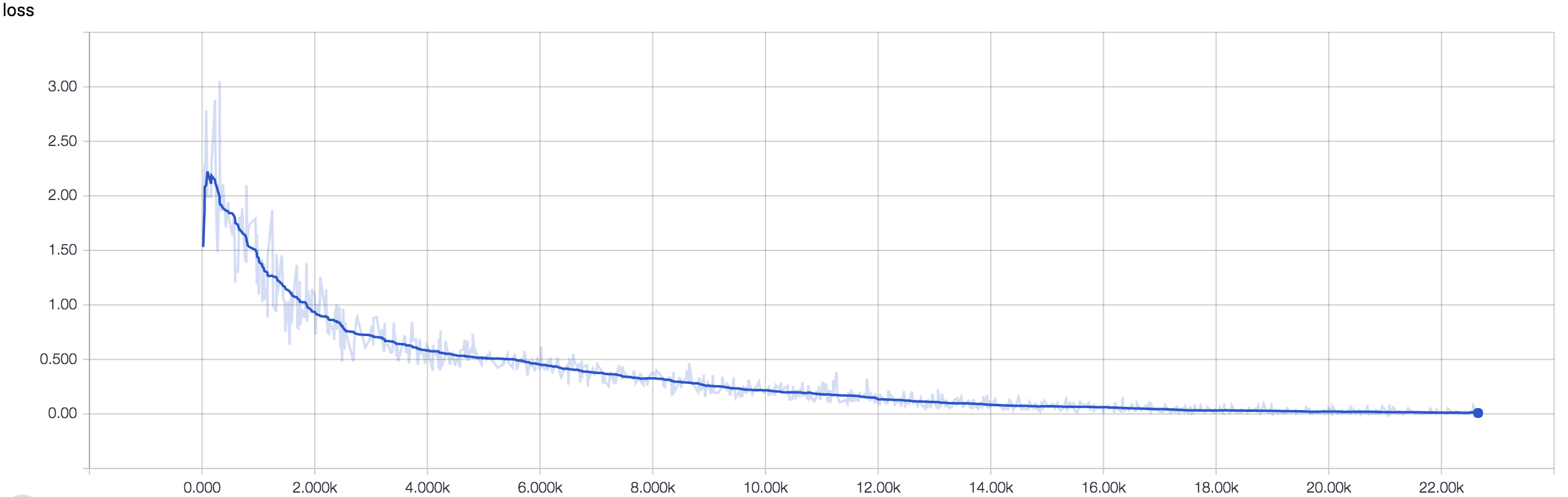
**Figure 7.** The development loss graph of a CNN run that was 200 epochs long.



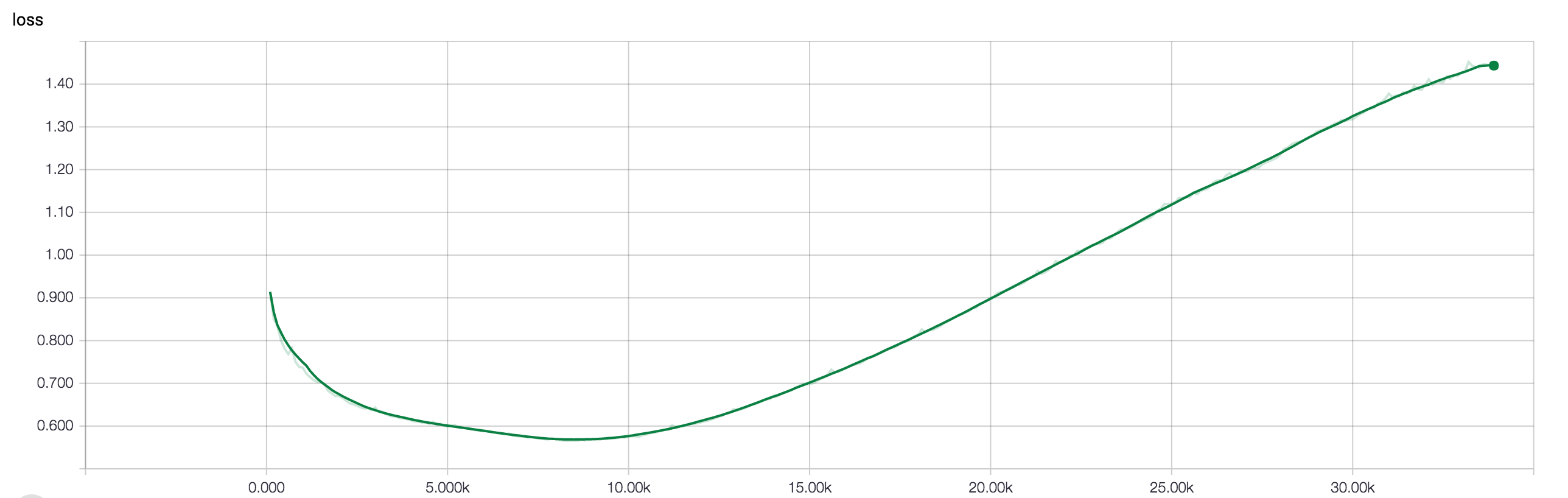
**Figure 8.** The training loss graph of a CNN run that was 200 epochs long.



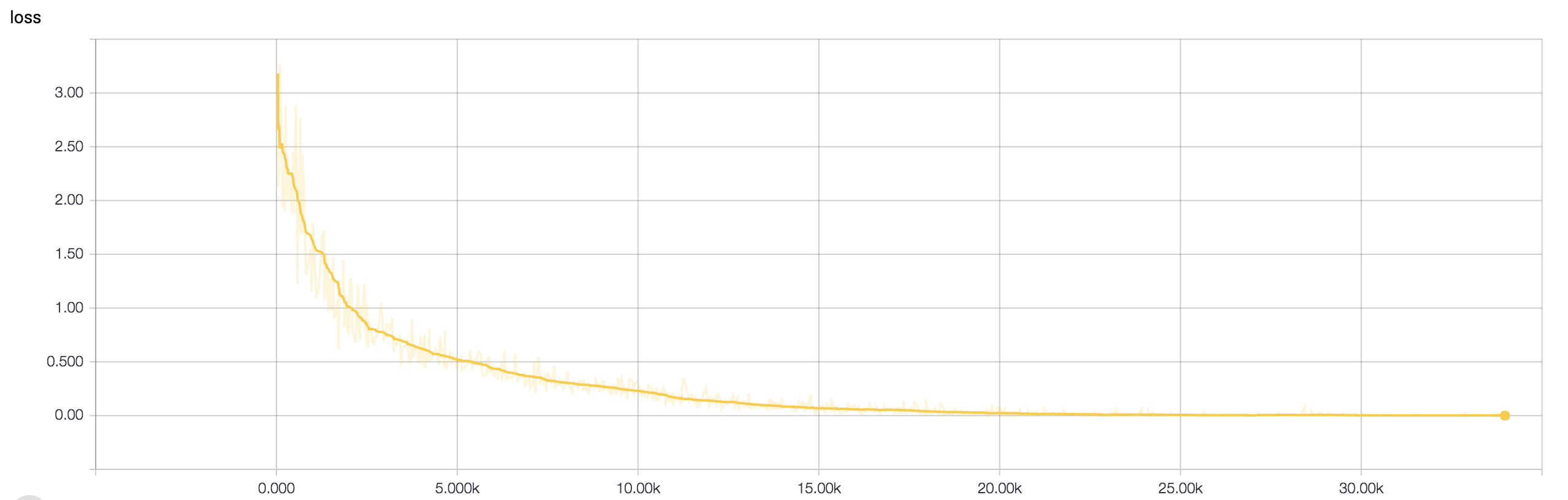
**Figure 9.** The development loss graph of a CNN run that was 150 epochs long.



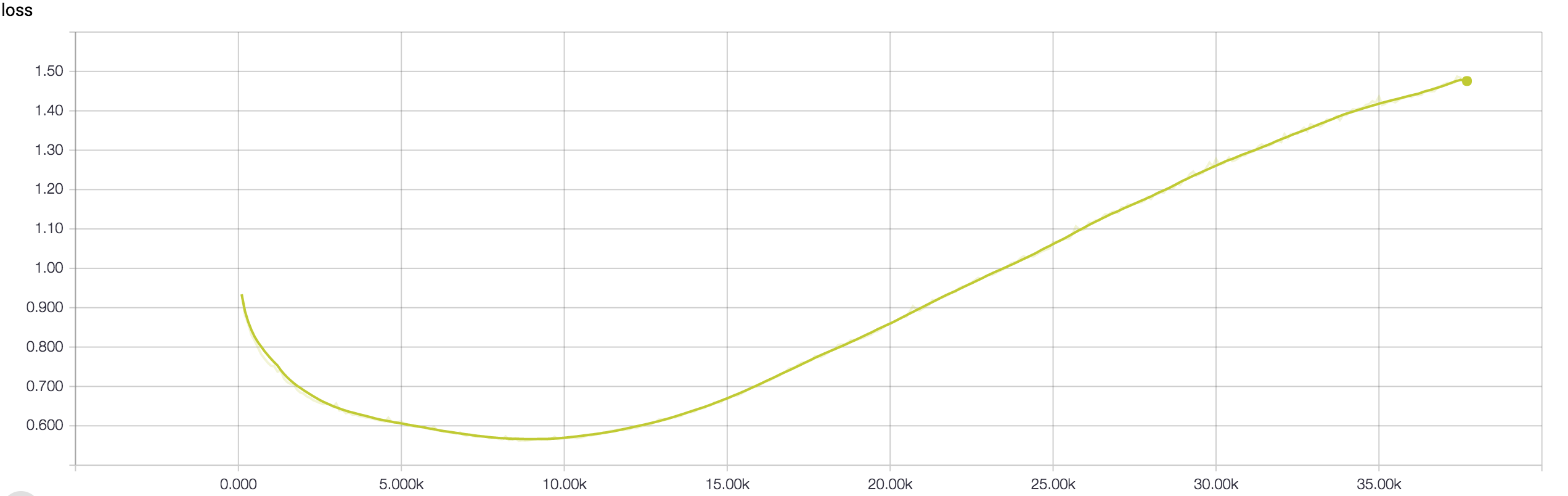
**Figure 10.** The training loss graph of a CNN run that was 150 epochs long.



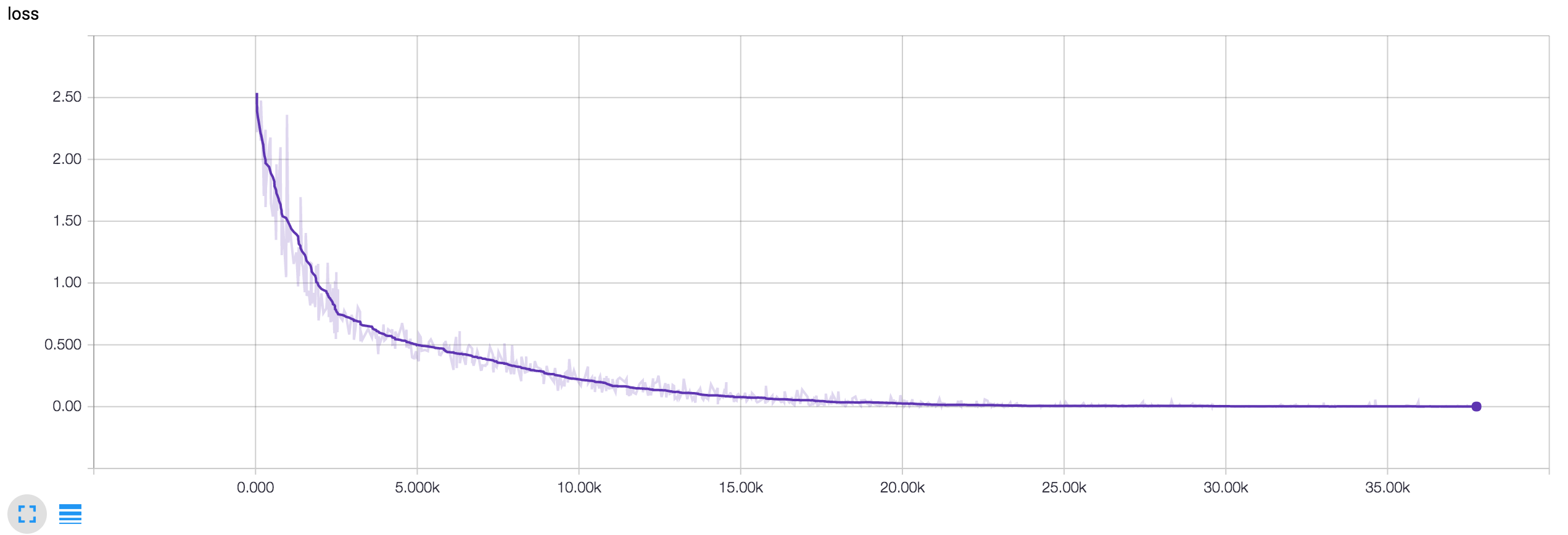
**Figure 11.** The development loss graph of a DCNN run that was 225 epochs long.



**Figure 11.** The training loss graph of a DCNN run that was 225 epochs long.



**Figure 12.** The development loss graph of a DCNN run that was 250 epochs long.



**Figure 13.** The training loss graph of a DCNN run that was 250 epochs long.

**SVM Data**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 98% | 20% | 33% |
| Positive | 55% | 97% | 70% |
| Average/Total | 72% | 59% | 52% |

**Table 1.** The precision, recall, and F1-Score of the SVM. This SVM was trained using scikit learn’s SVC class.

**Other Data**

After deciding to use the SVM as my control, I decided to explore some of the other classifiers built into scikit learn. Because these classifiers were easy to train, I trained each one using basic parameters. For some of the models it seemed that the accuracy was 99-100%, but I believe that that was a pipelining issue with scikit-learn’s cross validation method that may have effected all of this data; in the future I would like to investigate further as some of the architectures (particularly the multi-layer perceptron) were lightweight but still quite effective.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 100% | 99% | 100% |
| Positive | 99% | 100% | 100% |
| Average/Total | 100% | 100% | 100% |

**Table 2.** This is the precision, recall, and F1-Score scikit learn’s KNeighborsClassifier with k set to 3. The validation was clearly messed up on this one.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 82% | 47% | 60% |
| Positive | 63% | 89% | 74% |
| Average/Total | 72% | 68% | 67% |

**Table 3.** This is the precision, recall, and F1-Score scikit learn’s SVC with a linear kernel and C set to 0.025.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 100% | 100% | 100% |
| Positive | 100% | 100% | 100% |
| Average/Total | 100% | 100% | 100% |

**Table 4.** This is the precision, recall, and F1-Score scikit learn’s SVC with a RBF kernel and C set to 1. The validation was clearly messed up on this one.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 58% | 68% | 62% |
| Positive | 61% | 50% | 55% |
| Average/Total | 59% | 59% | 59% |

**Table 5.** This is the precision, recall, and F1-Score scikit learn’s DecisionTree with max depth set to 5.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 98% | 1% | 2% |
| Positive | 50% | 100% | 67% |
| Average/Total | 74% | 50% | 34% |

**Table 6.** This is the precision, recall, and F1-Score scikit learn’s RandomForest with max depth set to 5, estimators set to 10, and max features set to 1.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 84% | 83% | 83% |
| Positive | 83% | 84% | 83% |
| Average/Total | 83% | 83% | 83% |

**Table 6.** This is the precision, recall, and F1-Score scikit learn’s MLP with alpha at 1. I have suspicions that there were pipelining issues that caused this data to look more promising than it actually is.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Precision | Recall | F1-Score |
| Negative | 65% | 67% | 66% |
| Positive | 66% | 64% | 65% |
| Average/Total | 66% | 66% | 66% |

**Table 6.** This is the precision, recall, and F1-Score scikit learn’s AdaBoost.

**Final Data**

|  |  |
| --- | --- |
|  | Accuracy |
| DCNN | 85.00% |
| CNN | 71.92% |
| SVM | 72.00% |

**Table 2.** The final accuracies of the SVM, DCNN, and CNN when validated using an IMDB validation set that none of the architectures had been trained on.

I learned machine learning and TensorFlow primarily by reading and watching videos online. Throughout this process I made many smaller programs along the way. All of these programs can be found at <https://github.com/dsiegler19/learning_machine_learning>.

As well, along the way in my code I commented about some of the things I learned or found.

**Notes runs of classifiers for sentiment analysis:**

Good parameters for the classifiers  
All data trained using parameter\_tuning.py.  
  
NuSVC classifier data (with movie\_reviews) [3/7/16]:  
  
(n = 30)  
+------+----------+  
| Nu | Accuracy |  
+------+----------+  
| 0.4 | 71.0 |  
| 0.5 | 73.8 |  
| 0.6 | 72.3 |  
| 0.7 | 72.9 |  
| 0.75 | 72.8 |  
| 0.8 | 73.0 | <==  
+------+----------+  
  
Best nu value is 0.8  
  
Degree is inconsequential, so degree will be 3. As well, changing decision\_function\_shape to "ovr" is recommended so  
it will be done.  
  
RandomForestClassifier (with movie\_reviews) [4/7/16]:  
  
(n = 30)  
+---------------------------+----------+  
| n\_estimators (# of trees) | Accuracy |  
+---------------------------+----------+  
| 6 | 64.0 |  
| 8 | 64.1 |  
| 10 | 65.9 |  
| 12 | 66.8 |  
| 14 | 67.8 |  
| 16 | 69.1 |  
| 20 | 71.6 |  
| 25 | 75.1 | <==  
+---------------------------+----------+  
  
n\_estimators = 25 is the point at which the growth in accuracy slows down significantly but the running time increases  
significantly, so n\_estimator = 25 is the best value.  
  
(n = 30)  
+------------------+----------+  
| min\_samples\_leaf | Accuracy |  
+------------------+----------+  
| 1 | 74.4 |  
| 2 | 75.0 |  
| 3 | 75.1 |  
| 4 | 74.9 |  
| 5 | 74.5 |  
| 6 | 76.6 | <==  
| 7 | 75.9 |  
| 8 | 74.4 |  
| 9 | 75.7 |  
| 10 | 73.9 |  
| 11 | 75.2 |  
| 12 | 74.5 |  
| 13 | 74.0 |  
| 14 | 74.3 |  
| 15 | 73.5 |  
| 16 | 73.7 |  
| 17 | 72.8 |  
| 18 | 74.3 |  
| 19 | 74.1 |  
| 20 | 71.1 |  
| 25 | 69.9 |  
| 30 | 67.7 |  
| 35 | 69.2 |  
| 40 | 67.7 |  
| 45 | 68.0 |  
| 50 | 65.7 |  
| 55 | 63.1 |  
| 60 | 64.6 |  
| 65 | 62.5 |  
| 70 | 62.8 |  
| 75 | 62.2 |  
| 80 | 64.3 |  
| 85 | 60.5 |  
| 90 | 61.8 |  
| 95 | 59.4 |  
| 100 | 57.9 |  
+------------------+----------+  
  
The best min\_samples\_leaf is 6.  
A graph of this data can be found at min\_samples\_leaf\_to\_accuracy.png.  
  
The average accuracy of the OpinionLexiconClassifier in opinion\_lexicon\_classifier.py (n = 100) is 62.2.  
  
The average accuracy of the LinearSVC algorithm (n = 60) is 78.3.  
  
(n = 30)  
+-----+----------+  
| C | Accuracy |  
+-----+----------+  
| 0.2 | 76.9 |  
| 0.4 | 77.8 |  
| 0.6 | 78.9 |  
| 0.8 | 78.4 |  
| 1.0 | 78.3 | <==  
| 1.2 | 76.8 |  
| 1.4 | 78.6 |  
| 1.6 | 77.1 |  
| 1.8 | 77.1 |  
| 2.0 | 78.0 |  
| 2.2 | 76.9 |  
| 2.4 | 77.6 |  
| 2.6 | 77.7 |  
| 2.8 | 77.6 |  
| 3.0 | 78.1 |  
| 3.2 | 78.6 |  
| 3.4 | 77.8 |  
| 3.6 | 78.5 |  
| 3.8 | 75.7 |  
| 4.0 | 76.3 |  
| 4.2 | 76.8 |  
| 4.4 | 77.0 |  
| 4.6 | 77.3 |  
| 4.8 | 77.7 |  
| 5.0 | 78.4 |  
| 5.2 | 78.3 |  
| 5.4 | 77.4 |  
| 5.6 | 77.6 |  
| 5.8 | 78.0 |  
| 6.0 | 77.9 |  
| 6.2 | 78.6 |  
| 6.4 | 78.9 |  
| 6.6 | 77.8 |  
| 6.8 | 76.9 |  
| 7.0 | 77.7 |  
| 7.2 | 79.2 |  
| 7.4 | 77.8 |  
| 7.6 | 78.0 |  
| 7.8 | 78.0 |  
| 8.0 | 76.9 |  
| 8.2 | 77.5 |  
| 8.4 | 77.6 |  
| 8.6 | 77.2 |  
| 8.8 | 78.2 |  
| 9.0 | 79.4 |  
| 9.2 | 77.2 |  
| 9.4 | 77.3 |  
| 9.6 | 77.7 |  
| 9.8 | 77.9 |  
+-----+----------+  
  
Although C values at 9.0 and 7.2 do provide accuracy above 79.0%, this is hardly statistically significant. It seems  
that by C = 0.6 the C value has hit a critical mass and after this point raising the C value provides no increase in  
accuracy. For this reason, C = 1.0.  
  
The default accuracy of the scikit learn Support Vector Classifier (SVC) is ~50%. However, this can be greatly increased  
by upping the C value.  
  
(n = 30)  
+-----+----------+-----------------------------+  
| C | Accuracy | Average Training Time (sec) |  
+-----+----------+-----------------------------+  
| 5 | 75.3 | 23.2 |  
| 10 | 78.3 | 22.6 |  
| 15 | 80.4 | 22.3 | <==  
| 20 | 79.1 | 22.1 |  
| 25 | 80.0 | 22.0 |  
| 30 | 80.6 | 21.9 |  
| 35 | 79.7 | 21.8 |  
| 40 | 80.2 | 21.7 |  
| 45 | 80.7 | 21.8 |  
| 50 | 79.9 | 21.7 |  
| 55 | 80.7 | 21.5 |  
| 60 | 80.7 | 21.6 |  
| 65 | 79.5 | 21.5 |  
| 70 | 80.2 | 21.4 |  
| 75 | 80.1 | 21.4 |  
| 80 | 81.2 | 21.4 |  
| 85 | 81.0 | 21.4 |  
| 90 | 80.6 | 21.4 |  
| 95 | 80.0 | 21.3 |  
| 100 | 81.2 | 21.3 |  
+-----+----------+-----------------------------+  
  
C value of 15 seems to provide sufficient accuracy. As well, changing decision\_function\_shape to "ovr" is recommended so  
it will be done.  
  
(n = 30)  
+--------+----------+-----------------------------+  
| Degree | Accuracy | Average Training Time (sec) |  
+--------+----------+-----------------------------+  
| 1 | 47.8 | 22.9 |  
| 2 | 50.3 | 22.9 |  
| 3 | 47.9 | 22.9 |  
| 4 | 48.3 | 22.8 |  
| 5 | 46.4 | 22.9 |  
| 6 | 51.8 | 22.9 |  
| 7 | 50.2 | 22.9 |  
| 8 | 46.3 | 22.8 |  
| 9 | 46.3 | 22.9 |  
| 10 | 46.5 | 22.9 |  
+--------+----------+-----------------------------+  
  
The degree remains inconsequential, so it will be kept at the default of 3.  
  
The scikit learn Stochastic Gradient Descent Classifier (SGDC) provides a default accuracy of 77.4%.  
  
(n = 30)  
+---------------------------+----------+---------------------+  
| Loss | Accuracy | Training Time (sec) |  
+---------------------------+----------+---------------------+  
| hinge | 78.8 | 20.2 | <==  
| log | 77.1 | 18.1 |  
| modifier\_huber | 78.3 | 18.0 |  
| squared\_hinge | 77.0 | 18.0 |  
| perceptron | 76.8 | 18.0 |  
| squared\_loss | 48.3 | 18.0 |  
| huber | 53.7 | 18.0 |  
| epsilon\_insensitive | 50.4 | 18.0 |  
| squared\_epsilon\_intensive | 49.7 | 18.1 |  
+---------------------------+----------+---------------------+  
  
The default of hinge provides the best results, so it will remain.  
  
The scikit learn Logistic Regression algorithm provides a default accuracy of 77.9%.  
  
(n = 30)  
+----+----------+---------------------+  
| C | Accuracy | Training Time (sec) |  
+----+----------+---------------------+  
| 2 | 79.4 | 18.9 | <==  
| 4 | 79.6 | 22.5 |  
| 6 | 80.2 | 21.9 |  
| 8 | 79.5 | 20.3 |  
| 10 | 80.0 | 18.2 |  
| 12 | 80.0 | 18.6 |  
| 14 | 79.4 | 20.3 |  
| 16 | 80.1 | 17.6 |  
| 18 | 79.2 | 17.6 |  
| 20 | 80.5 | 17.5 |  
| 22 | 78.5 | 17.5 |  
| 24 | 80.0 | 17.6 |  
| 26 | 79.6 | 17.5 |  
| 28 | 79.9 | 17.6 |  
| 30 | 78.7 | 17.5 |  
| 32 | 79.2 | 17.5 |  
| 34 | 79.4 | 17.5 |  
| 36 | 80.1 | 17.4 |  
| 38 | 78.4 | 17.6 |  
| 40 | 80.0 | 17.5 |  
| 42 | 79.7 | 17.6 |  
| 44 | 78.7 | 17.6 |  
| 46 | 79.6 | 17.4 |  
| 48 | 79.4 | 17.4 |  
+----+----------+---------------------+  
  
Since C seems inconsequential, it will be set to 2.  
  
The Bernoulli Naïve Bayes provides a default accuracy of 77.5% with training time of 22.1 seconds (n = 60). There are no  
parameters to tune.  
  
The Multinomial Naïve Bayes provides a default accuracy of 79.0% with training time of 21.0 seconds (n = 60). There are no  
parameters to tune.  
  
The AdaBoostClassifier provides a default accuracy of 73.6% with a training time of 18.8 seconds (n = 60).  
  
(n = 30)  
+--------------+----------+---------------------+  
| n\_estimators | Accuracy | Training Time (sec) |  
+--------------+----------+---------------------+  
| 5 | 62.6 | 18.2 |  
| 10 | 67.5 | 18.3 |  
| 15 | 67.5 | 18.4 |  
| 20 | 70.2 | 18.3 |  
| 25 | 69.2 | 18.3 |  
| 30 | 71.5 | 18.4 |  
| 35 | 73.2 | 18.6 |  
| 40 | 72.9 | 18.5 |  
| 45 | 73.4 | 18.6 |  
| 50 | 74.9 | 18.8 |  
| 55 | 74.9 | 19.0 |  
| 60 | 75.7 | 19.1 | <==  
| 65 | 75.6 | 19.3 |  
| 70 | 74.1 | 19.1 |  
| 75 | 74.3 | 19.2 |  
| 80 | 76.1 | 19.3 |  
| 85 | 75.9 | 19.6 |  
| 90 | 75.9 | 19.4 |  
| 95 | 75.6 | 19.8 |  
| 100 | 76.3 | 19.6 |  
+--------------+----------+---------------------+  
  
The default of 50 provides adequate accuracy, so it shall remain.  
  
DON'T USE:  
NLTK MAXENTCLASSIFIER  
NLTK DECISIONTREECLASSIFIER  
NLTK RANDOMFORESTCLASSIFIER  
SCIKIT LEARN GAUSSIANNB  
  
------------------------------------------------------------------------------------------------------------------------  
Feature Extraction  
All data gathered from parameter\_tuning.py  
  
Using stop word filtering yielded the following results for the NuSVC classifier with nu = 0.75 (n = 10):  
66.97% Accuracy (with an old data set)  
  
Stemming yielded a loss of ~5-10% accuracy (on average)  
  
(n = 30)  
+----------------------------+----------+  
| # of Top Features Included | Accuracy |  
+----------------------------+----------+  
| 2000 | 67.9 |  
| 3000 | 71.5 |  
| 4000 | 70.6 |  
| 5000 | 75.2 |  
| 6000 | 75.4 |  
| 7000 | 76.9 |  
| 8000 | 77.2 |  
| 9000 | 79.3 | <==  
| 10000 | 80.1 |  
| 11000 | 81.1 |  
| 12000 | 81.1 |  
| 13000 | 80.1 |  
| 14000 | 81.7 |  
| 20000 | 82.3 |  
+----------------------------+----------+  
  
9000 seems to be the point at which the increase in accuracy slows down the most. For now, 9000 will be used.  
  
Using only adverbs (as apposed to adverbs and verbs) in featureset provides an increase in accuracy.  
  
Surprisingly, bigrams provided no advantage in accuracy and in general only made training times slower.  
  
+---------------+-----------------+----------+  
| Top all\_words | Top all\_bigrams | Accuracy |  
+---------------+-----------------+----------+  
| 0 | 9000 | 50.1 |  
| 1000 | 8000 | 64.9 |  
| 2000 | 7000 | 68.4 |  
| 3000 | 6000 | 70.6 |  
| 4000 | 5000 | 71.0 |  
| 5000 | 4000 | 71.9 |  
| 6000 | 3000 | 73.3 |  
| 7000 | 2000 | 74.5 |  
| 8000 | 1000 | 76.1 |  
| 9000 | 0 | 76.1 | <==  
| 9000 | 500 | 74.8 |  
| 9000 | 1000 | 75.4 |  
| 9000 | 1500 | 74.9 |  
| 9000 | 2000 | 75.8 |  
| 9000 | 2500 | 74.3 |  
| 9000 | 3000 | 75.9 |  
| 9000 | 3500 | 74.8 |  
| 9000 | 4000 | 75.2 |  
| 9000 | 4500 | 74.9 |  
| 9000 | 5000 | 74.9 |  
| 9000 | 5500 | 75.4 |  
| 9000 | 6000 | 75.5 |  
| 9000 | 6500 | 75.1 |  
| 9000 | 7000 | 75.8 |  
| 9000 | 7500 | 75.6 |  
| 9000 | 8000 | 74.3 |  
| 9000 | 8500 | 74.1 |  
| 9000 | 9000 | 75.4 |  
+---------------+-----------------+----------+  
  
The original configuration of the top 9000 of all\_words and no bigrams yields the best results. Although trigrams and  
higher ngrams (with n as high as 5) are recommended, it seems that even bigrams simply make way to much noise, so other  
ngrams won't be considered or tested.  
  
------------------------------------------------------------------------------------------------------------------------  
  
Old data:  
  
NuSVC classifier data (with short\_reviews) [2/7/16]:  
  
(n = 10):  
+------+----------+  
| Nu | Accuracy |  
+------+----------+  
| 0.4 | 60.97 |  
| 0.45 | 63.55 |  
| 0.5 | 63.65 |  
| 0.55 | 64.76 |  
| 0.6 | 66.00 |  
| 0.65 | 65.36 |  
| 0.7 | 67.12 |  
| 0.75 | 67.23 | <==  
| 0.8 | 66.74 |  
| 0.85 | 67.17 |  
+------+----------+  
  
Best nu value is 0.75.  
  
(n = 10)  
+--------+----------+  
| Degree | Accuracy |  
+--------+----------+  
| 2 | 67.02 |  
| 3 | 66.40 |  
| 4 | 66.94 |  
+--------+----------+  
  
Degree is inconsequential, so degree is 3.

**Notes on general topics when I was learning about NLP using NLTK (some of the formatting is messed up):**

Tokenizing -  
 Word tokenizer - separates by word  
 Sentence tokenizer - separates by sentence  
There are many ways to tokenize using nltk. There are 3 included in Tutorials/tokenizing.py (nltk's naitive tokenizer,  
trained PunktSentenceTokenizer, and pretrained PunktSentenceTokenizer):  
  
==========================================================  
  
Lexicon - The words and their meanings (dictionary, financial speak). On this computer, lexicon (and corpora) are found  
in /Users/dsiegler19/nltk\_data/corpora.  
Corpora - A body of text (i.e. medical journals, presidential speeches, anything in the english language). On this  
computer, corpora (and lexicon) are found in /Users/dsiegler19/nltk\_data/corpora.  
  
Some examples are found in Tutorials/lexicon\_and\_corpora.py  
  
===========================================================  
  
Stop words - Commonly used words like and, by, then, at, etc. that provide no real meaning to the computer, so  
they are often ignored completely. Code to do this is in Tutorials/stop\_words.py.  
  
===========================================================  
  
Stemming - A process of "normalizing" words. It removes things like tense, person, number, etc.  
  
 I was taking a ride in the car  
 I was riding in the car  
  
These sentences mean the same thing, so stemming would turn both of these verbs into simply ride.  
  
 Eat  
 Eating  
 Was eating  
 About to eat  
 Will eat  
 Eater  
 Eaten  
  
These would all simply become eat. Code to do this is in Tutorials/stemming.py.  
  
===========================================================  
  
Part of speech tagging - Labeling the part of speech for every word.  
  
Part of speech tag list:  
|-------------------------------------------------------------------------------|  
|Abbreviation | Meaning/Explanation | Example |  
|-------------|----------------------------------------|------------------------|  
| CC | coordinating conjunction | and |  
| CD | cardinal digit | 7 |  
| DT | determiner | an |  
| EX | existential | there is/there exists |  
| FW | foreign word | bonjour |  
| IN | preposition/subordinating conjunction | under/because |  
| JJ | adjective | big |  
| JJR | adjective, comparative | bigger |  
| JJS | adjective, superlative | biggest |  
| LS | list marker | 1) |  
| MD | modal | will, could |  
| NN | noun, singular | desk |  
| NNS | noun plural | desks |  
| NNP | proper noun, singular | Smith |  
| NNPS | proper noun, plural | Americans |  
| PDT | predeterminer | all |  
| POS | possessive ending | parent's |  
| PRP | personal pronoun | I, he, she |  
| PRP$ | possessive pronoun | my, his, hers |  
| RB | adverb | silently |  
| RBR | adverb, comparative | silenter |  
| RBS | adverb, superlative | silentist |  
| RP | particle (doesn't inflect) | give up (only up) |  
| TO | the word to in any use | to |  
| UH | interjection | errrrrrrrm |  
| VB | verb, base form | take |  
| VBD | verb, past tense | took |  
| VBG | verb, gerund/present participle | taking |  
| VBN | verb, past participle | taken |  
| VBP | verb, sing. present, non-3d | take |  
| VBZ | verb, 3rd person sing. present | takes |  
| WDT | wh-determiner | which |  
| WP | wh-pronoun | who, what |  
| WP | possessive wh-pronoun | whose |  
| WRB | wh-abverb | where, when |  
|-------------------------------------------------------------------------------|  
  
Code to do this is in Tutorials/POS\_tagging.py.  
  
===========================================================

Chunking - Given that some text has been tokenized by word and sentence and has been tagged by part of speech, chunking  
finds the named entities (nouns), words that modify each of these named entities, and what each one is referring to.  
Chunking splits each sentence into noun phrases. Chunked words must be next to each other.  
  
Chunking is mainly done using regular expressions. Here is a tutorial on regular expressions:  
https://pythonprogramming.net/regular-expressions-regex-tutorial-python-3/  
  
By convention the chunk is included in a raw triple quote string (see example code):  
  
 regex expression  
r"""ChunkName: {<POSIdentifier>}"""  
  
Code to do this (with a regular expression that can be improved upon) is in Tutorials/chunking.py.  
  
===========================================================  
  
Chinking - Removing something from a chunk. One can say to chunk everything and then chink (remove) a few things from  
the chunks. Chinking is also done via regular expressions and is in the same string as the chunk (see example code).  
Code to do this (with regular expressions that can be improved upon) is in Tutorials/chinking.py.  
  
===========================================================  
  
Name Entity Recognition - A way of chunking to find most proper nouns. The nltk.ne\_chunk() method finds things such as  
names, places, organizations, dates, money, and other named entities. When binary argument of nltk.ne\_chunk() is true,  
named entities will all be categorized as NE. This means a phrase like "White House" will be categorized as one chunk  
since they are both simply NEs. When binary is false (default), it will provide more specific categories of named  
entities such as PERSON or GPE. As well, when binary is false it will separate a phrase such as "White House" into  
White (FACILITY) and House (ORGANIZATION).  
  
Named entity types and examples:  
|----------------------------------------------------------|  
| Name | Example |  
|----------------------------------------------------------|  
| ORGANIZATION | Georgia-Pacific Corp., WHO |  
| PERSON | Eddy Bonte, President Obama |  
| LOCATION | Murray River, Mount Everest |  
| DATE | June, 2008-06-29 |  
| TIME | two fifty a m, 1:30 p.m. |  
| MONEY | 175 million Canadian Dollars, GBP 10.40 |  
| PERCENT | twenty pct, 18.75 % |  
| FACILITY | Washington Monument, Stonehenge |  
| GPE | South East Asia, Midlothian |  
|----------------------------------------------------------|  
  
Code to do this can be found in Tutorials/NE\_recognition.py.  
  
===========================================================  
  
Lemmatizing - A similar operation to stemming, but the end result is an actual word. That word may be the root of the  
original word or it may be a synonym. Lemmatizing is often better and more effective then stemming. The lemmatizer  
assumes that all words it is given are nouns, so one must specify if the word is not a noun for the lemmatizer to  
lemmatize it correctly.  
  
Code to do this can be found in Tutorials/lemmatizing.py  
  
===========================================================  
  
WordNet - WordNet is a tool for looking up synonyms, antonyms, definitions, context of words, and many other useful  
tools. This tool also includes a lexicon of the English language, a synonym and antonym dictionary, and a word context  
dictionary. Some ways to use WordNet are:  
  
- Find all of the synonyms and antonyms of words  
- Find the definitions of a word  
- Find example uses of a word  
- Test to see how similar 2 words are  
  
Code to do all of this can be found in Tutorials/wordnet.py.  
  
===========================================================  
  
Text Classification using Sentimental Analysis - A way of classifying text as having a positive, negative, or neutral  
opinion/connotation of what they are talking about.  
  
Code to do this can be found in TextClassifier/text\_classifier.py  
  
===========================================================  
  
Pickle - A module that can be used to save Python objects. In this case it is used to save the training data in  
TextClassifier/text\_classifier.py.  
  
===========================================================  
  
Sicit-Learn - A machine learning module used in conjunction with nltk.

Many of my notes can be found in the code at <https://github.com/dsiegler19/learning_machine_learning>.