DELHI TECHNOLOGICAL UNIVERSITY

Software Engineering Project File

Topic: **Overfitting in Neural Networks.**

Submitted To: **Jamkhongam Touthang Sir**

Submitted By: **Sarthak Goel 2K18/MC/102**

Sarthi Nigam 2K18/MC/104

INDEX

1. Abstract

Deep Neural Networks generally contain a lot of features. The more the features the better the classification but at the same time we must also remember that it makes the system very complex and also takes a lot of time to process. Our goal in this paper is hence going to be a way to find out how we can reduce the number of features which is called “dropping out features of over fitted networks” ,While doing this we also want to make sure that there is no information loss pertaining to it That is why we are implementing this project to make sure that we can remove unnecessary features without significant or in some cases no loss in information so we will be working on the MNIST image dataset for this to happen.

2. Introduction:

Deep neural networks contain many non-linear hidden layers, and this makes them capable of learning complicated relationships between the inputs provided and outputs they got, because of this the noise in the training dataset is very high there are many samples which exist in training data but they do not exist in the testing data This is leading to overfitting and there are many ways to prevent it some ways are-stop the training as soon as performance on a validation set starts to get worse, introducing weight penalties of various kinds such as L1 and L2 regularization and soft weight sharing.

If we are allowed to perform unlimited computation then we the method of average predictions using posterior probability will be the best method but we would not use this method due to the fact that the number of computation are very high. Model combination improves the performance of machine learning methods. With large neural networks, however, the obvious idea of averaging the outputs of many separately trained nets is highly time taking and complicated.
Combining several models is most helpful when the individual models are different from each other and in order to make neural net models different, they should either have different architectures or be trained on different data sets.

Training different networks can get very difficult if we consider the fact about finding the correct parameter for each network on top of that we have to consider to do it then finding high amount of training data can get very straining, if we manage to get the training dataset also we get tuck because test large amount will again take time but in most situations quick response is desired.

Dropout is a technique that can solve these issues. It prevents overfitting and provides a way of approximately combining many different neural network architectures efficiently. The term “dropout” refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, even its incoming and outgoing connections, as shown in Figure 1. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

A neural net with n units, be thought of as a collection of 2^n possible thinned neural networks. These networks all share weights so that the total number of parameters is still O(n^2) or less. For each presentation of each training case, a new thinned network is sampled and trained. So, training a neural network with dropout be training a collection of 2^n thinned networks where it is very rare for a thinned network to even be trained if at all it is trained.

At test time, it is not feasible to explicitly average the predictions from exponentially so many thinned models. However, a very simple approximate averaging method works well in use. The idea is to use a single neural net at test time without applying dropout. The weights of this network are scaled-down versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time. This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output received at test time. By doing this scaling, 2n networks with shared weights can be combined into a single neural network to be used at test time. We found that training a network with dropout and using this approximate averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods.

3. MOTIVATION:

A motivation for dropout comes from a theory of the role of sex in evolution.

Generally Asexual Reproduction(cloning) which is supposed to be better because it produces exact topics but we are still using advancement and the reasons could be.

One possible explanation for the superiority of sexual reproduction is that, over the long term, the criterion for natural selection may not be individual fitness but rather mix-ability of genes. The ability of a set of genes to be able to work well with another random set of genes makes them more robust.

According to this theory, the role of sexual reproduction is not just to allow useful new genes to spread throughout the population, but also to facilitate this process by reducing complex co- adaptations that would reduce the chance of a new gene improving the fitness of an individual. Similarly, each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes.

But slightly different motivation for dropout comes from thinking about successful conspiracies. Ten conspiracies each involving five people is probably a better way to create havoc than one big conspiracy that requires fifty people to all play their parts correctly. If conditions do not change and there is plenty of time for rehearsal, a big conspiracy can work well, but with non-stationary conditions, the smaller the conspiracy the greater its chance of still working. Different co- adaptations can be trained to work well on a training set, but on novel test data they are far more likely to fail than multiple simpler co-adaptations that achieve the same thing.

4. Methodology:

This research is concerned with identifying the role of improvement of overfitting in neural network management system. Data has been taken from many sources and analyzed. As a result of understanding the aims and objective of this study and exploring previous research similar to this topic. We feel using a dropout on neural networks and thinning the network has helped us working on a dataset like MNIST. Dropout refers to research doing numbers and is the collection of facts and the study of the relationship between the attributes of the MNIST dataset .
Qualitative Method has also been used in the case study. Finally, We consider how the dataset reacts.

5. THE MODEL:

In this section we see the dropout neural network model. Consider a neural network with L hidden layers. Let $l \in \{1, \ldots, L\}$ index the hidden layers of the network. Let $z^{(l)}$ denote the vector of inputs into layer l, $y^{(l)}$ denotes the vector of outputs from layer l ($y^{(0)} = x$ is the input). $W^{(l)}$ and $b^{(l)}$ are the weights and biases at layer l. The feed-forward operation of a standard neural network (Figure 3a) can be described as (for $l \in \{0, \ldots, L-1\}$ and any hidden unit.

$z^{(l+1)} i = w^{(l+1)} i\, y^{l} + b^{(l+1)}$

$y^{(l+1)} i = f(z^{(l+1)})$

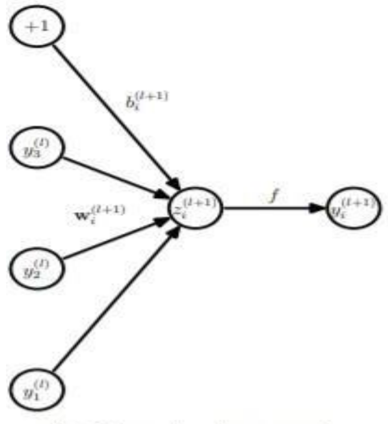
after applying dropout, the equations change as

follows $r^{(l)} j \sim \text{Bernoulli}^{(p)}$

$e^{(l)} = r^{(l)} * y^{(l)}$

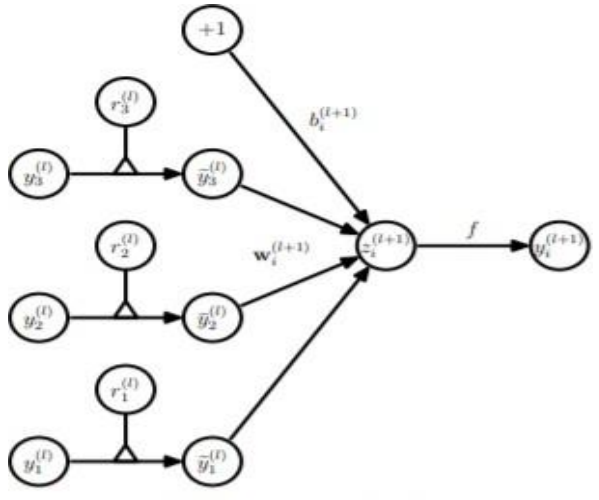
$z^{(l+1)} i = w^{(l+1)} i\, y^{(l)} + b^{(l+1)}$

Next we see the image of standard network and the network in which dropout has been applied and compare the two.



(a) Standard network

This is the network before applying dropout.



(b) Dropout network

This is the network after applying dropout.

Now we need to focus on learning the dropout nets.

There are 2 main methods to learn about drop out nets one is backpropagation and the other is unsupervised pretraining lets look at this in detail.

Backpropagation:

Dropout neural networks can be trained using stochastic gradient descent in a way similar to standard neural nets. The only difference is that for each training case in a mini batch, we will have to sample a thinned network by dropping out some units. Forward and backpropagation for that training case are done only on this thinned network that we got after dropout. The gradients for each parameter are averaged over the training cases in small batches. Any training case which does not use a parameter contributes a gradient of zero for that parameter. Many methods have been used to improve stochastic gradient descent such as momentum, annealed learning rates and L2 weight decay. Those were found to be useful for dropout neural networks as well.

One form of regularization was found to be especially useful for dropout— constraining the norm of the incoming weight vector at each hidden unit to be upper bounded by a fixed constant
c. In other words, if w represents the vector of weights incident on any hidden unit, the neural network was optimized under the constraint $||w||^2$ should be less than c. This constraint was imposed during optimization by projecting w onto the surface of a ball of radius c, whenever w went out of it. This is also called max-norm regularization since it implies that the maximum value that the norm of any weight can take is c. The constant c is a tunable hyperparameter, which is determined using a validation set. Max-norm regularization has been previously used in the context of collaborative filtering. It typically improves the performance of stochastic gradient descent training of deep neural nets, even when no dropout is used.

Unsupervised Pretraining:

Neural networks can be trained using stacks of RBMs,autoencoders or Boltzmann Machines . Pretraining is an effective way of making use of data which has not been labelled. Pretraining followed by finetuning with backpropagation has been shown to give significant performance boosts over finetuning from random initializations in certain cases.

Dropout can be applied to finetune nets that have been pretrained using these techniques. The pretraining procedure stays the same. The weights obtained from pretraining are scaled up by a factor of 1/p. This makes sure that for each unit, the expected output from it under random dropout will be the same as the output during pretraining.

We were earlier concerned with the stochastic nature of dropout might wipe out the information in the pretrained weights. This did happen when the learning rates used duringfinetuning were comparable to the best learning rates for randomly initialized nets. However, when the learning rates were chosen to be smaller, the information in the pretrained weights seemed to be retained and we were able to get improvements in terms of the final generalization error compared to not using dropout when finetuning.

6. MNIST DATASET:

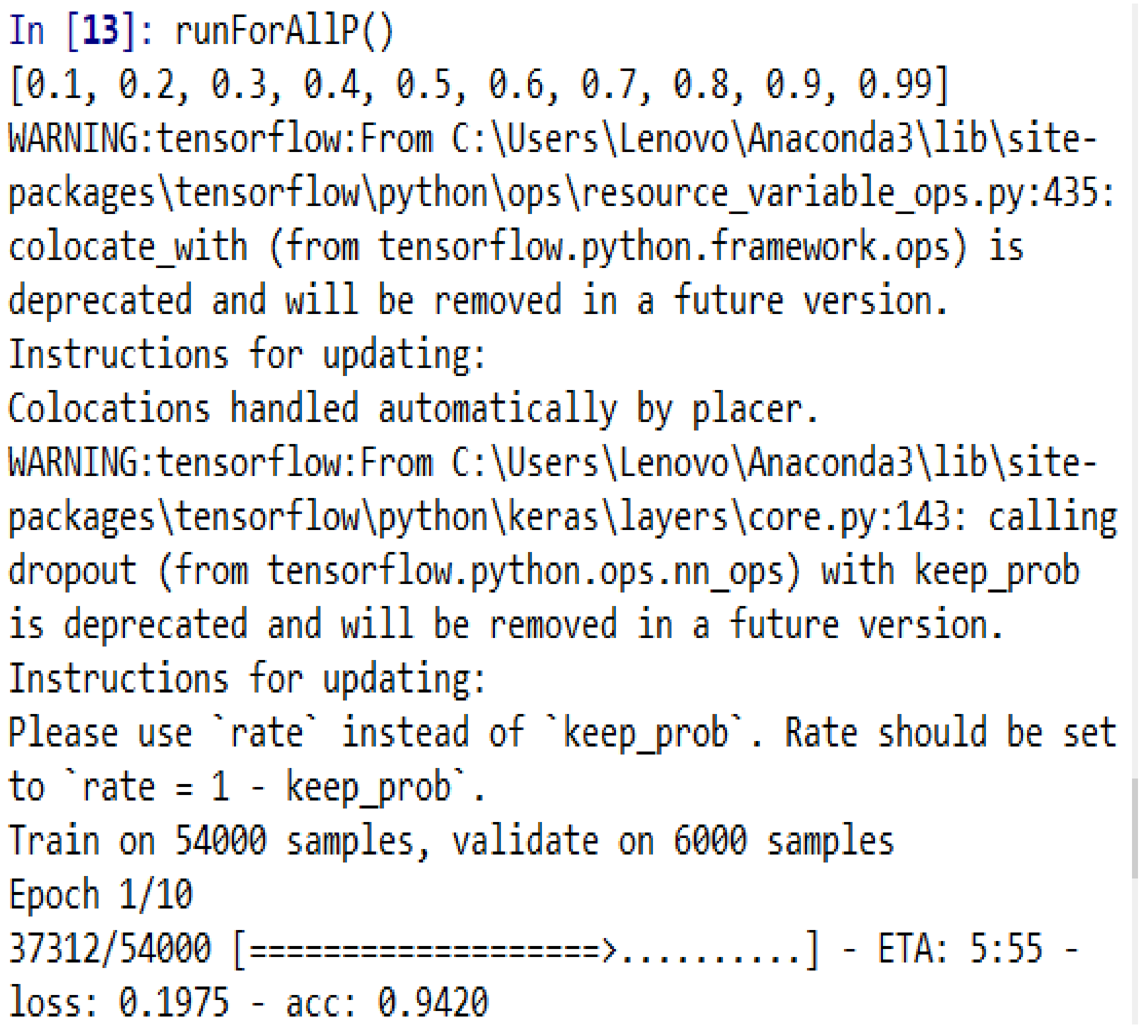
The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It contains small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9.The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.

It is a widely used and deeply understood dataset and, for the most part, is “solved.” Top-performing models are deep learning convolutional neural networks that achieve a classification accuracy of above 99%, with an error rate between 0.4 %and 0.2% on the hold out test dataset.

7. EXPERIMENTAL RESULTS ON MNIST DATASET:

The task is to classify the images into 10 digit classes

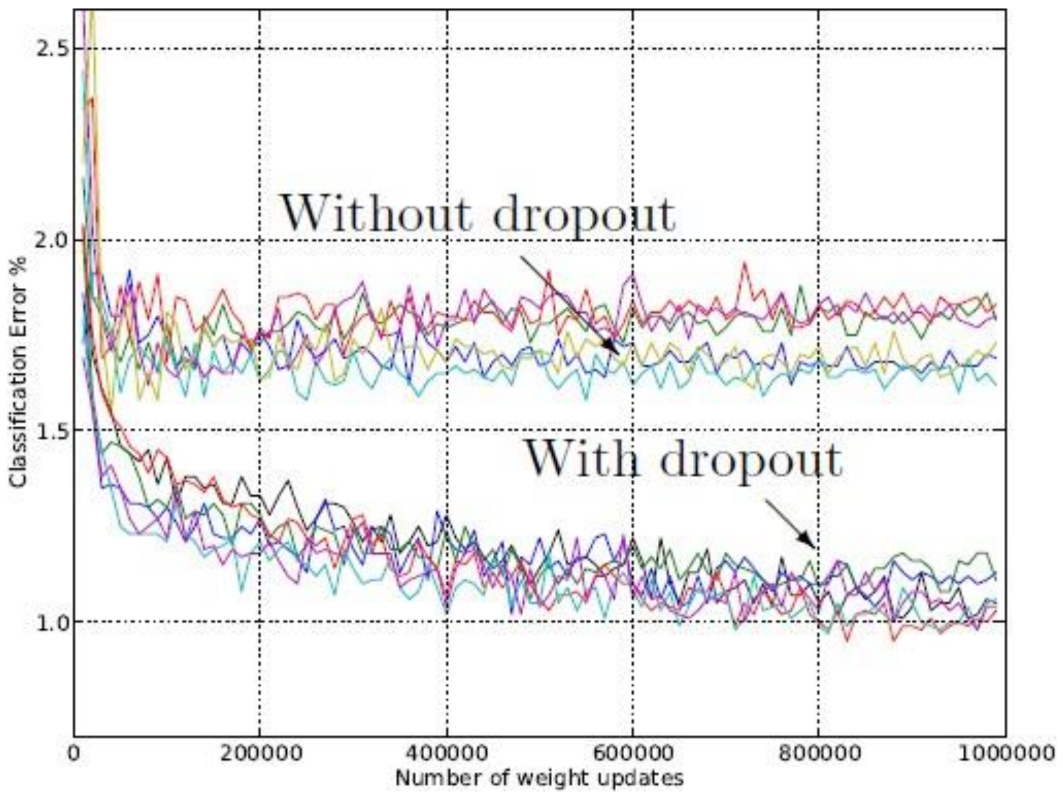
Outputs of the code after run:



```
In [13]: runForAllP()
[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99]
WARNING:tensorflow:From C:\Users\Lenovo\Anaconda3\lib\site-
packages\tensorflow\python\ops\resource_variable_ops.py:435:
colocate_with (from tensorflow.python.framework.ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From C:\Users\Lenovo\Anaconda3\lib\site-
packages\tensorflow\python\keras\layers\core.py:143: calling
dropout (from tensorflow.python.ops.nn_ops) with keep_prob
is deprecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set
to `rate = 1 - keep_prob`.
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
37312/54000 [===================>.........] - ETA: 5:55 -
loss: 0.1975 - acc: 0.9420
```

```
Epoch 2/10
54000/54000 [==============================] - 1275s 24ms/
sample - loss: 0.0628 - acc: 0.9836 - val_loss: 0.0668 -
val_acc: 0.9832
Epoch 3/10
54000/54000 [==============================] - 1279s 24ms/
sample - loss: 0.0383 - acc: 0.9896 - val_loss: 0.0873 -
val_acc: 0.9800
Epoch 4/10
54000/54000 [==============================] - 1202s 22ms/
sample - loss: 0.0277 - acc: 0.9923 - val_loss: 0.0730 -
val_acc: 0.9858
Epoch 5/10
54000/54000 [==============================] - 1180s 22ms/
sample - loss: 0.0249 - acc: 0.9942 - val_loss: 0.1762 -
val_acc: 0.9697
Epoch 6/10
54000/54000 [==============================] - 1141s 21ms/
sample - loss: 0.0207 - acc: 0.9951 - val_loss: 0.0985 -
val_acc: 0.9832
Epoch 7/10
54000/54000 [==============================] - 1136s 21ms/
sample - loss: 0.0221 - acc: 0.9952 - val_loss: 0.0806 -
val_acc: 0.9868
Epoch 8/10
54000/54000 [==============================] - 1132s 21ms/
sample - loss: 0.0145 - acc: 0.9972 - val_loss: 0.1035 -
val_acc: 0.9858
```

```
Epoch 8/10
54000/54000 [==============================] - 1132s 21ms/
sample - loss: 0.0145 - acc: 0.9972 - val_loss: 0.1035 -
val_acc: 0.9858
Epoch 9/10
54000/54000 [==============================] - 1163s 22ms/
sample - loss: 0.0172 - acc: 0.9965 - val_loss: 0.1469 -
val_acc: 0.9788
Epoch 10/10
54000/54000 [==============================] - 1164s 22ms/
sample - loss: 0.0205 - acc: 0.9961 - val_loss: 0.1269 -
val_acc: 0.9825
Traceback (most recent call last):
```



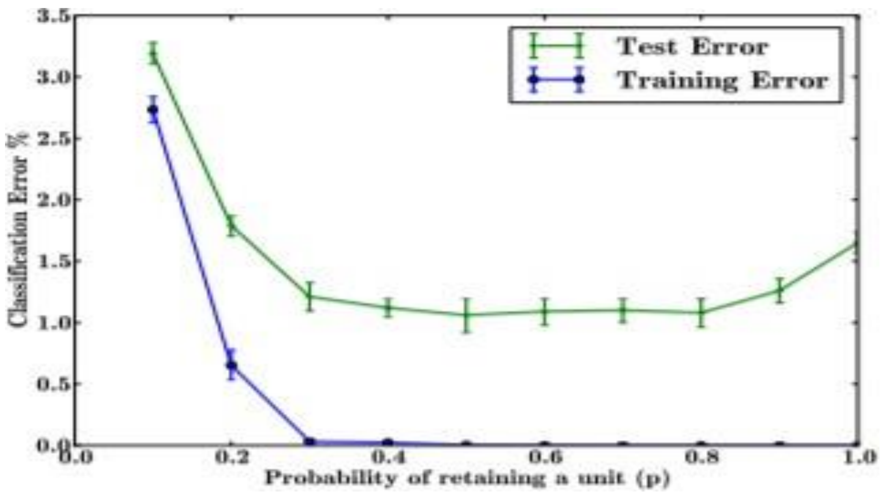
This is the number of weight updates proportional to the classification error percentage. Dropout has a tunable hyperparameter p (the probability of retaining a unit in the network). The comparison is done in two situations.

1. The number of hidden units is held constant.

2. The number of hidden units is changed so that the expected number of hidden units that will be retained after dropout is held constant.

In the first case, we train the same network architecture with different amounts of dropout. A 784-2048-2048-2048-10 architecture with no input dropout was used. Figure (a) shows the test error obtained as a function of p.

If the architecture is held constant, having a small p means very few units will turn on during training. It can be seen that this has led to underfitting since the training error is also high. We see that as p increases, the error goes down. It becomes at when 0:4 p 0:8 and then increases as p becomes close to 1.



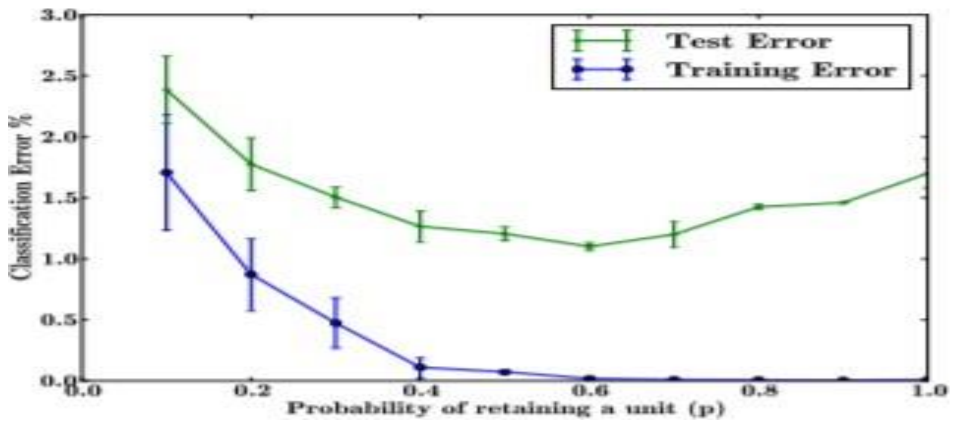
This is keeping n fixed.

In the second case in which the quantity pn is held constant where n is the number of hidden units in any particular layer. This means that networks that have small p will have a large number of hidden units.

Therefore, after applying dropout, the expected number of units that are present will be the same across different architectures. However, the test networks will be of different sizes.

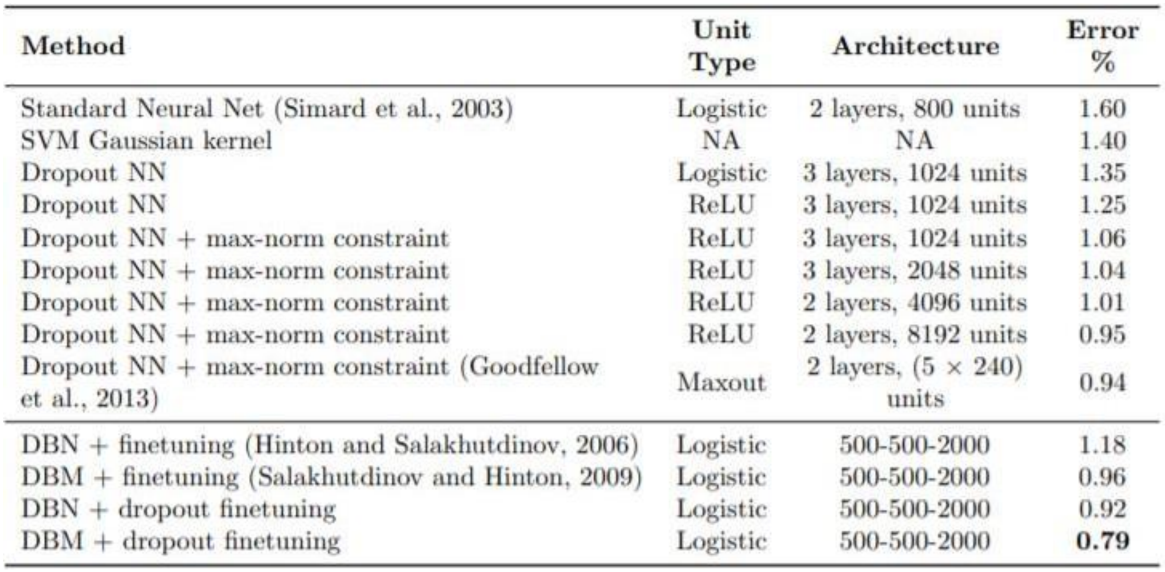
In this experiment, pn was set equal to 256 for the first two hidden layers and 512 for the last hidden layer.

Figure (b) shows the test error obtained as a function of p. It is noticeable that the magnitude of errors for small values of p has reduced by a lot compared to Figure (a) (for p = 0.1 it fell from 2.7% to 1.7%). Values of p that are close to 0.6 seem to perform best for this choice of pn but the usual default value of 0.5 is close to optimal.



This is keeping pn fixed

This table which is shown next has the final result along with error percentage calculation.



Method	Unit Type	Architecture	Error %
Standard Neural Net (Simard et al., 2003)	Logistic	2 layers, 800 units	1.60
SVM Gaussian kernel	NA	NA	1.40
Dropout NN	Logistic	3 layers, 1024 units	1.35
Dropout NN	ReLU	3 layers, 1024 units	1.25
Dropout NN + max-norm constraint	ReLU	3 layers, 1024 units	1.06
Dropout NN + max-norm constraint	ReLU	3 layers, 2048 units	1.04
Dropout NN + max-norm constraint	ReLU	2 layers, 4096 units	1.01
Dropout NN + max-norm constraint	ReLU	2 layers, 8192 units	0.95
Dropout NN + max-norm constraint (Goodfellow et al., 2013)	Maxout	2 layers, (5×240) units	0.94
DBN + finetuning (Hinton and Salakhutdinov, 2006)	Logistic	500-500-2000	1.18
DBM + finetuning (Salakhutdinov and Hinton, 2009)	Logistic	500-500-2000	0.96
DBN + dropout finetuning	Logistic	500-500-2000	0.92
DBM + dropout finetuning	Logistic	500-500-2000	**0.79**

8. CODE FOR REFERENCE

We have given the code for reference below:

```
import tensorflow as tf
import matplotlib.pyplot as
plt import numpy as np
import
decimal
import os
import csv

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten,
MaxPooling2D # gets rid of an error about cpu on Macbook Pro 2017
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

# Calculate hidden layer values, and add to list
```

```
def findValueN(prob):
   # pn = 256 for 2 first hidden
   layers # pn = 512 for last layer
      # n values are the number of hidden units at each
   layer n1 = 256.0/prob
   n2 =
   256.0/prob n3
   = 512.0/prob
   n =
      [n1,n2,n3,prob]
      # return
      number o
   float(n[3
   ]) return
   n

# Create the archetecture for constant
archetecture def addPValue(prob):
      # layers all have 2048 hidden units as described in the
   research paper n1 = 2048
   n2 = 2048
   n3 = 2048
   n = [n1,n2,n3,prob]
   float(n[3])
   return n

# Runs dropout in an neural network for specific layer
values def runDropout(*layer):
      # Sequential model and the layers that describe the
```

```
model model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape,
activation="relu")) # model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten()) # Flattening the 2D arrays for fully
connected layers model.add(Dense(layer[0],
activation=tf.nn.relu)) model.add(Dropout(layer[3]))
model.add(Dense(layer[1],
activation=tf.nn.relu))
model.add(Dropout(layer[3]))
model.add(Dense(layer[2],
activation=tf.nn.relu))
model.add(Dropout(layer[3]))
model.add(Dense(10,
activation=tf.nn.softmax))


 # Configure model before
training
model.compile(optimizer='adam'
,
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

 # Train the model for a fixed number of epochs
history_dropout = model.fit(x=x_train, y=y_train, validation_split=0.1, epochs=10,
batch_size=32)

 # Training accuracy
accuracy =
   history_dropout.history['accuracy']
   train_err = 100.0 - 100.0*(accuracy[-1])


      # Test Accuracy
         # Validation accuracy possibly not same as Test
   accuracy val_acc =
   history_dropout.history['val_accuracy']
   test_err = 100.0 -
   100.0*(val_acc[-1]) # * Calculate
   error bars HERE *
   finalError = [test_err, train_err, layer[3]]
   return finalError
   # END OF RUN DROPOUT #

# Function to calculate all p values in a float
range def floatRange(start, stop, step):
  while start <= stop:
   yield float(start)
   start += decimal.Decimal(step)

def runForAllP():
   # lists to store ploting values
      locally # figure a error
   a_test_error = []
   a_train_error = []
      # figure b error
   b_test_error = []
   b_train_error = []
      # returns the list of p values from 0.1-1.0,
into list pValues = list(floatRange(0, 0.9, '0.1'))
  # remove 0 from the pValue
list del pValues[0]
pValues.append(0.99)
print(pValues)
# run dropout on all values of p for both
figures for i in pValues:
  # check if pValue was already
    calculated # if so, import values
    and don't run
  varyLayer = findValueN(i)
  constLayer = addPValue(i)
    # Store error values for figure a
  errorFigA =
  runDropout(*constLayer)
  a_test_error.append(errorFigA[0]
  )
  a_train_error.append(errorFigA[1
  ]) print(a_train_error)
  print("\n")
for i in
pValues:
  varyLayer = findValueN(i)
  constLayer = addPValue(i)
  errorFigB =
  runDropout(*varyLayer) # Store
  error values for figure b
  b_test_error.append(errorFigB[0
  ])
```

```
        b_train_error.append(errorFigB[
        1])
    # Figure 9a
    plt.subplot(2, 2,
    1)
    plt.ylabel('Classification Error %')
    plt.xlabel('Probability of retaining a unit
    (p)') plt.yticks(np.arange(0, 4, 0.5))
    plt.plot(pValues, a_test_error, label = "Test Error")
    plt.plot(pValues, a_train_error, label = "Training
    Error") plt.legend(loc='upper right')
    # Figure 9b
    plt.subplot(2, 2,
    2)
    plt.ylabel('Classification Error %')
    plt.xlabel('Probability of retaining a unit
    (p)') plt.yticks(np.arange(0, 3.5, 0.5))
    plt.plot(pValues, b_test_error, label = "Test Error")
    plt.plot(pValues, b_train_error, label = "Training
    Error") plt.legend(loc='upper right')
    plt.tight_layout()
    # save plotted dropout
    figure plt.savefig('dropout-
    figure.png') plt.show()

# Load dataket to appropriate valiables
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()


# Reshaping the array to 4-dims so that it can work with
```

the API x_train = x_train.reshape(x_train.shape[0], 28,

28, 1)

x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)

input_shape = (28, 28, 1)

# Set values to float so that we can get decimal points after division

x_train = x_train.astype('float32')

x_test = x_test.astype('float32')

# Normalizing the RGB codes by dividing it to the max RGB

value x_train /= 255

x_test /= 255

# runs both archetectures for all values of p (from 0.1-1.0,

inclusive) runForAllP()

9. Conclusion:

Dropout is a technique for improving neural networks by reducing, sometimes completely removing overfitting. Standard backpropagation learning builds up rigid co-adaptations that work for the training data but do not generalize to unseen data. Random dropout breaks up these co-adaptations.

It takes very long to train neural networks and it takes even longer to train networks when dropout has been applied to it.

10. References:

https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

G. E. Dahl, M. Ranzato, A. Mohamed, and G. E. Hinton. Phone recognition with the meancovariance restricted Boltzmann machine. In Advances in Neural Information Processing Systems 23, pages 469– 477, 2010. O. Dekel, O. Shamir, and L. Xiao. Learning to classify with missing and corrupted features.

Machine Learning, 81(2):149–178, 2010. A. Globerson and S. Roweis. Nightmare at test time: robust learning by feature deletion. In Proceedings of the 23rd International Conference on Machine Learning, pages 353–360. ACM, 2006.

J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In Proceedings of the 30th International Conference on Machine Learning, pages 1319– 1327. ACM, 2013.

G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. Science, 313(5786):504 – 507, 2006. G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets.

Neural Computation, 18:1527–1554, 2006. K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In Proceedings of the International Conference on Computer Vision (ICCV'09). IEEE, 2009. A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. A. Krizhevsky, I. Sutskever, and G. E. Hinton.

Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, pages 1106–1114, 2012.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541–551, 1989.
Y. Lin, F. Lv, S. Zhu, M. Yang, T. Cour, K. Yu, L. Cao, Z. Li, M.-H. Tsai, X. Zhou, T. Huang, and T. Zhang.

Imagenet classification: fast descriptor coding and large-scale svm training. Large scale visual recognition challenge, 2010. A. Livnat, C. Papadimitriou, N. Pippenger, and M. W. Feldman. Sex, mixability, and modularity. Proceedings of the National Academy of Sciences, 107(4):1452–1457, 2010. V. Mnih.

CUDAMat: a CUDA-based matrix class for Python. Technical Report UTML TR 2009-004, Department of Computer Science, University of Toronto, November 2009.