**Deep Learning**

**Assignment 1**

**Rajbir Bhattacharjee**

**R00195734**

Contents

[Implementation 3](#_Toc69561971)

[Common Functions 3](#_Toc69561972)

[Cross Entropy 3](#_Toc69561973)

[Calculation of Accuracy 3](#_Toc69561974)

[Forward Pass 3](#_Toc69561975)

[General Observations 6](#_Toc69561976)

[Results 7](#_Toc69561977)

[Question 1\_1\_1 7](#_Toc69561978)

[Question 1\_1\_2 10](#_Toc69561979)

[Question 1\_2\_1 13](#_Toc69561980)

[Question 1\_3\_1 18](#_Toc69561981)

[Autoencoders 23](#_Toc69561982)

[Variational Autoencoders 26](#_Toc69561983)

[Adam Optimization Algorithm 32](#_Toc69561984)

# Implementation

## Common Functions

### Cross Entropy

While implementing cross entropy, if the values are too small, calling tf.math.log() will result in NaN’s. To deal with this problem, the values are clipped to 1.0e-10.

# This function implements the cross-entropy loss

**def** cross\_entropy**(**y\_true**,** y\_pred**)->**tf**.**Tensor**:**

***# First clip the values to 1.0 x 10^(-8) because if there***

***# are any values which are 0, then this will cause an error***

y\_pred **=** tf**.**clip\_by\_value**(**y\_pred**,** \

clip\_value\_min**=**1.0e-10**,** \

clip\_value\_max**=**tf**.**reduce\_max**(**y\_pred**))**

y\_pred **=** tf**.**math**.**multiply**(**y\_true**,** tf**.**math**.**log**(**y\_pred**))**

y\_pred **=** tf**.**multiply**(**y\_pred**,** **-**1**)**

***# Find the average cross entropy, divide by number of samples***

divisor **=** y\_pred**.**shape**[**1**]** **if** len**(**y\_pred**.**shape**)** **>** 1 **else** 1

y\_pred **=** tf**.**math**.**divide**(**tf**.**reduce\_sum**(**y\_pred**),** divisor**)**

**return** y\_pred

### Calculation of Accuracy

The accuracy function just compares the vectors, it is the caller’s responsibility to pass it vectors that are not one-hot encoded if need be.

# This function gets the accuracy

**def** accuracy**(**y\_true**:**tf**.**Tensor**,** y\_pred**:**tf**.**Tensor**)->**float**:**

logging**.**debug**(**"y\_true = " **+** str**(**y\_true**.**shape**)** **+** \

" y\_pred = " **+** str**(**y\_pred**.**shape**))**

y\_true **=** tf**.**transpose**(**y\_true**)**

logging**.**debug**(**"y\_true = " **+** str**(**y\_true**.**shape**)** **+** \

" y\_pred = " **+** str**(**y\_pred**.**shape**))**

eq **=** tf**.**math**.**equal**(**y\_true**,** y\_pred**)**

**return** float**(**eq**[**eq **==** **True].**shape**[**0**])** **/** float**(**eq**.**shape**[**0**])**

### Forward Pass

The forward pass function is written to maximize code reuse. It can take a variety of networks, apply dropout if required, gather the data from each run for evaluation.

The forward\_pass() function itself is simple and just pushes the vector through the different matrices. The train\_network() function iterates through the epochs and trains the network. The network itself is specified as an array of tuples, each index of the array representing a layer. Each layer is a tuple containing the following:

1. A matrix of the weights
2. A matrix of the bias
3. The activation function
4. Boolean to indicate whether to apply dropout to the input to this layer or not

The train\_network() function has an outer loop that iterates through the epochs. For each of the epochs, it first creates a gradient tape and calls forward\_pass(). It then gathers a bunch of statistics for evaluation. Finally, from the gradients it got from the tape, it calls optimize\_network() which applies Adam to the matrices.

The forward\_pass() function just runs a loop and pushes the data through the layers of the network. It invokes get\_dropout\_input() if required to apply dropout to the input if required.

The optimize\_network() applies the gradients that are passed to it to all the matrices that make up the model. Since the matrices are in an array of tuples, these are not the easiest to work with. To help with this task it calls flatten\_matrix(), which extracts the relevant matrices from the array of tuples, and optimize\_network can easily work with them without the need for complicated handling.

**def** get\_dropout\_input**(**self**,** input**:**tf**.**Tensor**)->**tf**.**Tensor**:**

prob\_dropout **=** self**.**prob\_dropout

prob\_threshold **=** 1 **-** prob\_dropout

drop\_matrix **=** tf**.**random**.**uniform**(**shape**=**input**.**shape**,** minval**=**0**,** maxval**=**1**)**

drop\_matrix **=** tf**.**math**.**less**(**drop\_matrix**,** prob\_threshold**)**

drop\_matrix **=** tf**.**cast**(**drop\_matrix**,** dtype**=**tf**.**float32**)**

input **=** tf**.**math**.**multiply**(**input**,** drop\_matrix**)**

input **=** tf**.**math**.**divide**(**input**,** prob\_threshold**)**

**return** input

***# The forward pass function, pushes the input through all the layers***

***# the parameter layers is a list of tuples, each tuple is of the form***

***# (***

***# weight-matrix:list,***

***# bias-matrix:list,***

***# activation\_function,***

***# is\_dropout\_enabled:bool***

***# )***

**def** forward\_pass**(**self**,** tr\_x**,** tr\_y**,** layers**:**list**):**

input **=** tf**.**transpose**(**tr\_x**)**

**for** weight**,** bias**,** activation**,** dropout **in** layers**:**

***# If dropout is enabled on the layer, apply dropout on the input***

**if** dropout**:**

input **=** self**.**get\_dropout\_input**(**input**)**

input **=** activation**(**tf**.**matmul**(**weight**,** input**)** **+** bias**)**

**return** input

**def** predict**(**self**,** tr\_x**,** tr\_y**):**

input **=** tf**.**transpose**(**tr\_x**)**

**for** weight**,** bias**,** activation**,** dropout **in** self**.**network**:**

input **=** activation**(**tf**.**matmul**(**weight**,** input**)** **+** bias**)**

**return** input

"""

Take a network which is a list of tuples, and flatten it

to a list, so that we can call adam and gradient on it without

without any ugly code

"""

**def** flatten\_network**(**self**,** network**:**list**)->**list**:**

out **=** **[]**

**for** weight**,** bias**,** act**,** dropout **in** network**:**

out**.**append**(**weight**)**

out**.**append**(**bias**)**

**return** out

"""

Use the adam optimizer to optimize a network based on the

gradients passed to it

"""

**def** optimize\_network**(**self**,** network**:**list**,** gradients**:**list**)->None:**

adam\_optimizer **=** tf**.**keras**.**optimizers**.**Adam**()**

learnable\_array **=** **[]**

adam\_optimizer**.**apply\_gradients**(**zip**(**gradients**,** \

self**.**flatten\_network**(**network**)))**

**def** should\_log**(**self**,** i**:int,** iterations**:int):**

**return** **(**i **==** **(**iterations **-** 1**))** **or** all**([**i **==** '0' **for** i **in** list**(**str**(**i**)[**1**:])])**

"""

Train the network for a given number of iterations

"""

**def** train\_network**(**self**,** \

iterations**:int,** \

tr\_x**:**tf**.**Tensor**,** tr\_y**:**tf**.**Tensor**,** \

te\_x**:**tf**.**Tensor**,** te\_y**:**tf**.**Tensor**,** \

loss\_fn**,** acc\_fn**,** \

do\_logging**=True,** \

use\_argmax**=True)->None:**

**for** i **in** range**(**iterations**):**

***# Call forward pass on the network and use the gradient tape***

**with** tf**.**GradientTape**()** **as** tape**:**

y\_pred **=** self**.**forward\_pass**(**tr\_x**,** tr\_y**,** self**.**network**)**

loss **=** loss\_fn**(**tr\_y**,** y\_pred**)**

gradients **=** tape**.**gradient**(**loss**,** self**.**flatten\_network**(**self**.**network**))**

***# Record the training accuracy before updating the weights***

**if** **(**use\_argmax**):**

tr\_acc **=** acc **=** acc\_fn**(**tf**.**argmax**(**tr\_y**),** tf**.**argmax**(**y\_pred**))** \

**if** **None** **!=** acc\_fn **else** float**(**"NaN"**)**

**else:**

tr\_acc **=** acc **=** acc\_fn**(**tr\_y**,** y\_pred**)** **if** **None** **!=** acc\_fn **else** float**(**"NaN"**)**

tr\_loss **=** loss

***# Log the same if required***

**if** do\_logging **and** self**.**should\_log**(**i**,** iterations**):**

printstr **=** f"Iteration {i:5d} TRAIN LOSS - {loss:5.5f} "

printstr **=** printstr **+** f"TRAIN ACCURACY - {acc:5.5f}"

***# Now run the prediction on the test set, and record the accuracy***

***# We haven't updated the weights yet***

y\_pred **=** self**.**predict**(**te\_x**,** te\_y**)**

te\_loss **=** loss **=** loss\_fn**(**te\_y**,** y\_pred**)**

**if** **(**use\_argmax**):**

te\_acc **=** acc **=** acc\_fn**(**tf**.**argmax**(**te\_y**),** tf**.**argmax**(**y\_pred**))** \

**if** **None** **!=** acc\_fn **else** float**(**"NaN"**)**

**else:**

te\_acc **=** acc **=** acc\_fn**(**te\_y**,** y\_pred**)** **if** **None** **!=** acc\_fn **else** float**(**"NaN"**)**

***# Log if required***

**if** do\_logging **and** self**.**should\_log**(**i**,** iterations**):**

printstr **=** printstr **+** f" TEST\_LOSS = {loss:5.5f} "

printstr **=** printstr **+** f"TEST ACCURACY = {acc:5.5f}"

logging**.**info**(**printstr**)**

***# Record the performance of this run***

self**.**iterations**.**append**(**i**)**

self**.**train\_accuracies**.**append**(**tr\_acc**)**

self**.**train\_losses**.**append**(**tr\_loss**)**

self**.**test\_accuracies**.**append**(**te\_acc**)**

self**.**test\_losses**.**append**(**te\_loss**)**

***# Now use ADAM and update the weights***

self**.**optimize\_network**(**self**.**network**,** gradients**)**

self**.**iterations **=** np**.**array**(**self**.**iterations**)**

self**.**train\_accuracies **=** np**.**array**(**self**.**train\_accuracies**)**

self**.**train\_losses **=** np**.**array**(**self**.**train\_losses**)**

self**.**test\_accuracies **=** np**.**array**(**self**.**test\_accuracies**)**

self**.**test\_losses **=** np**.**array**(**self**.**test\_losses**)**

As an example of the use of the above functions:

**def** q\_1\_1\_1**():**

tr\_x**,** tr\_y**,** te\_x**,** te\_y **=** get\_fashion\_mnist\_data**()**

***# Layer 1 has 200 neurons, and 784 features,***

***# hence the array should be (200, 784)***

network **=** **[**

***# Layer 1, 200 nodes, ReLu***

**(**make\_layer**([**200**,** 784**],** "L1 Weights"**),**

make\_layer**([**200**,** 1**],** "L1 Bias"**),**

tf**.**nn**.**relu**,**

**False),**

***# Layer 2, 10 nodes, Softmax***

**(**make\_layer**([**10**,** 200**],** "L2 Weights"**),**

make\_layer**([**10**,** 1**],** "L2 Bias"**),**

softmax**,**

**False)**

**]**

nn **=** NeuralNetwork**(**network**)**

nn**.**train\_network**(**1000**,** tr\_x**,** tr\_y**,** te\_x**,** te\_y**,** cross\_entropy**,** accuracy**)**

**return** nn

make\_layer() creates a matrix with the specified shape and initializes it randomly.

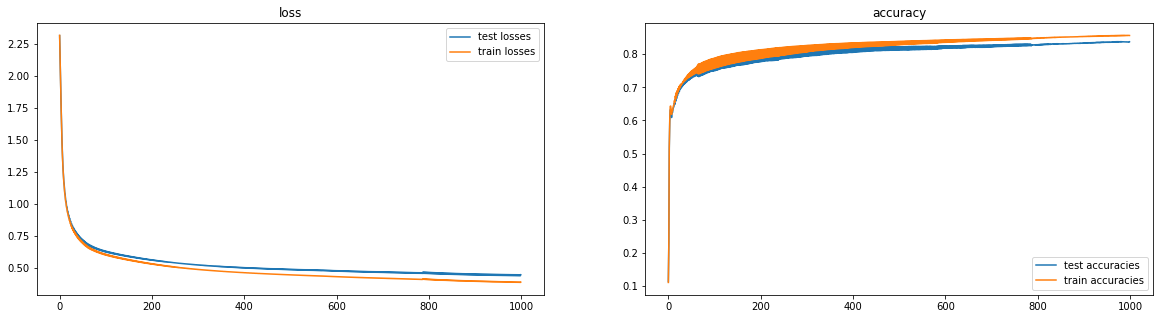
## General Observations

One general observation was that the model was very sensitive to the initial choice of the random weights. If the mean and standard deviation are not specified, then the model fails to converge even after a high number of iterations.

## Results

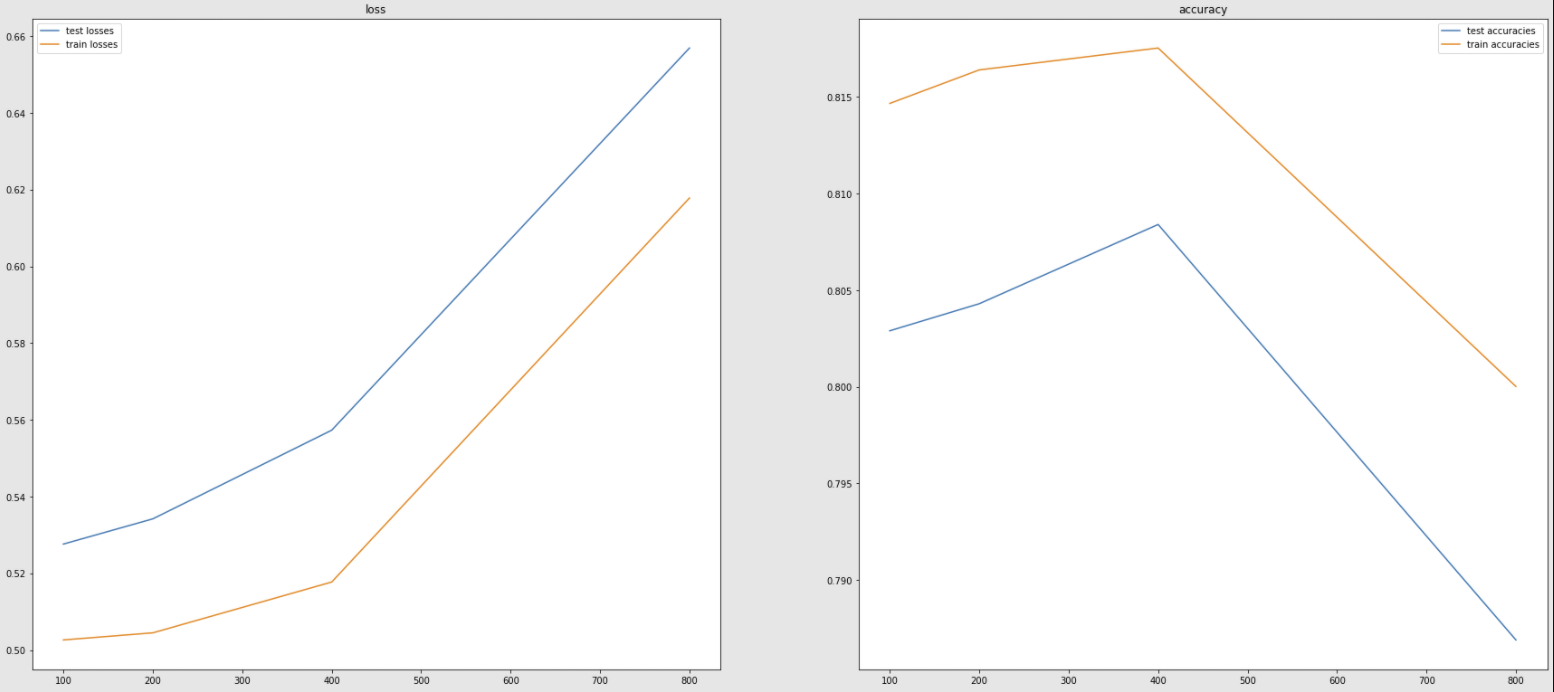
### Question 1\_1\_1

The plotted loss and accuracy functions take the following form



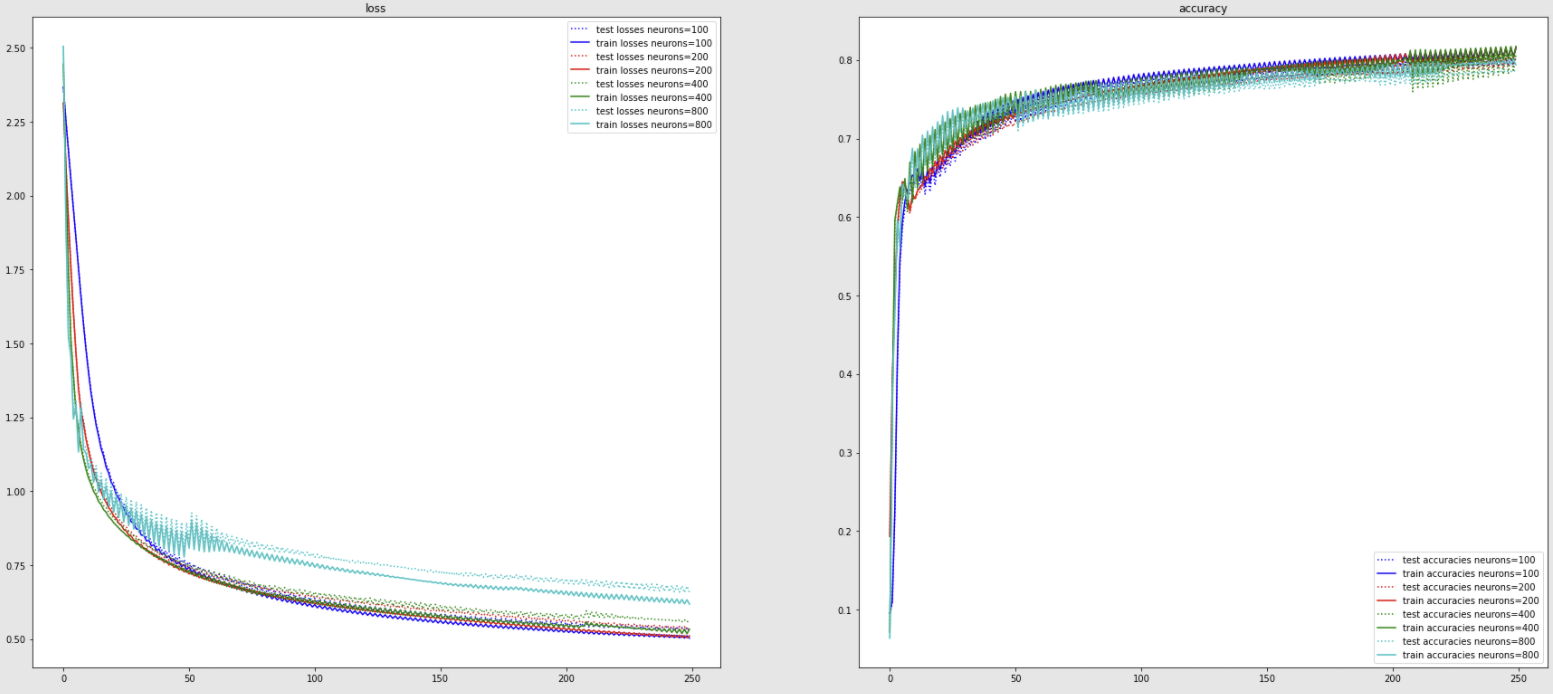
As we can see that beyond 400 iterations, the test and train losses and accuracies start diverging somewhat. At this stage, there is some indication of overfitting, although all through 0 to 1000 iterations, both the training and test loss do decrease, and the accuracies increase, and the overfitting is not severe.

The number of neurons in the hidden layer was varied, and the same test repeated multiple times. When the best run was chosen, this were the results when plotted across the number of neurons.



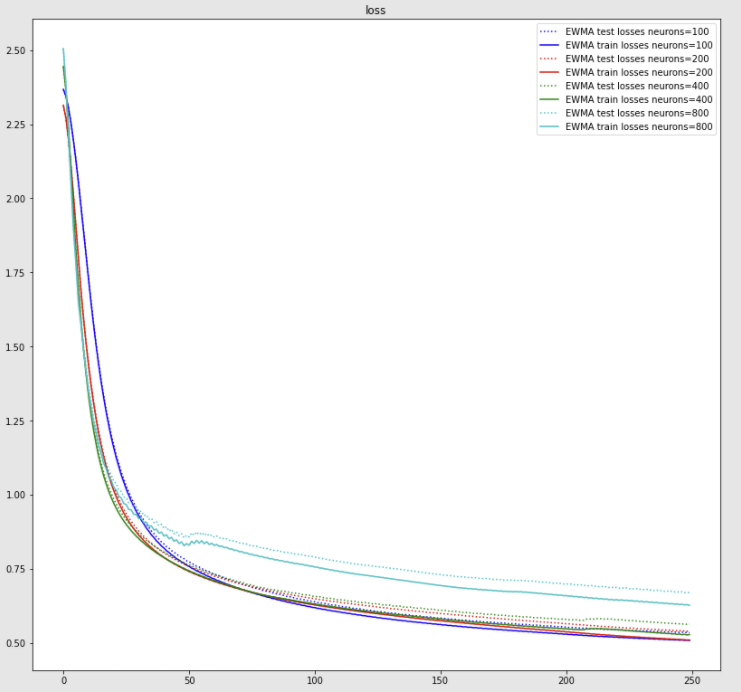
From what we see here is that as the number of neurons is increased, the loss actually increases instead of decreasing. However, strangely, the accuracies actually improve till 400 neurons, and then drop significantly. The losses also increase steeply beyond 400 neurons.

These were plotting the best epoch from all epochs, but if each epoch is plotted individually the graphs look like this.



The take-away from this is that in a shallow-network, increasing the number of neurons in each layer is not always beneficial.

Since this graph is too noisy, the same was plotted using a running average to give more clarity. Looking at the loss gives a better understanding, and for the benefit of readability, the accuracy graph has been left out.



What we see here is that with 100 neurons, the model starts out with worse performance, but soon the performance with 100 neurons gets better than other models with more than 100 neurons. When 800 neurons are used the performance tanks very early.

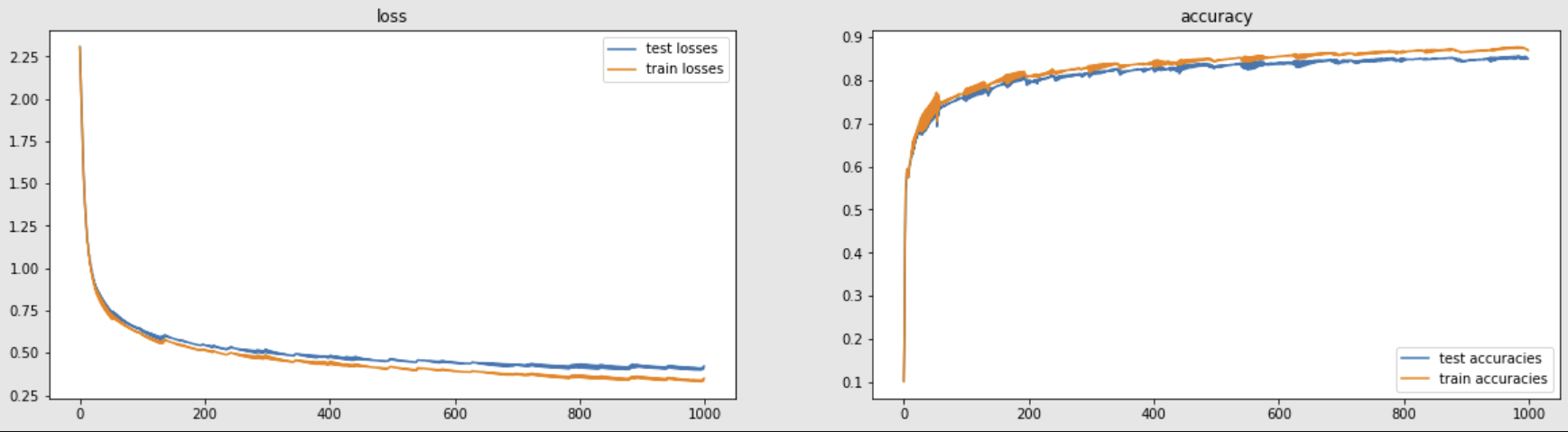
The other observations is that the higher the number of neurons the greater the overfitting as the iterations progress. With 100 neurons, we see low levels of overfitting, but much higher levels of overfitting with 800 neurons.

#### Take aways

The take-away here is that increasing the number of neurons in a shallow network may result in overfitting and may not be beneficial. It may always be a good idea to vary the number of neurons and see the results. Auto-ML may also help in discovering an optimal network architecture. Having a smaller number of neurons in each layer may actually produce better results with less overfitting even with a high number of epochs.

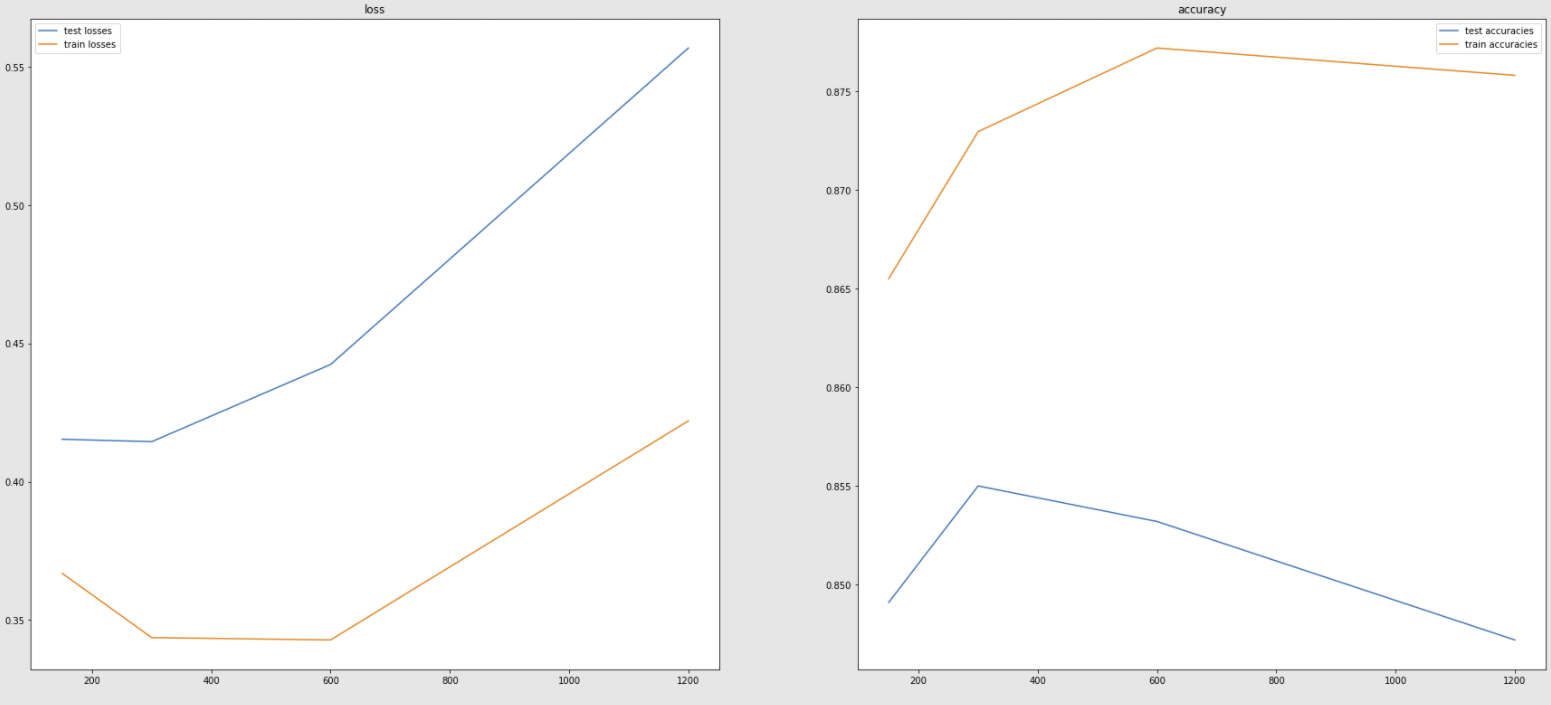
### Question 1\_1\_2

The plotted losses and accuracies take the following form



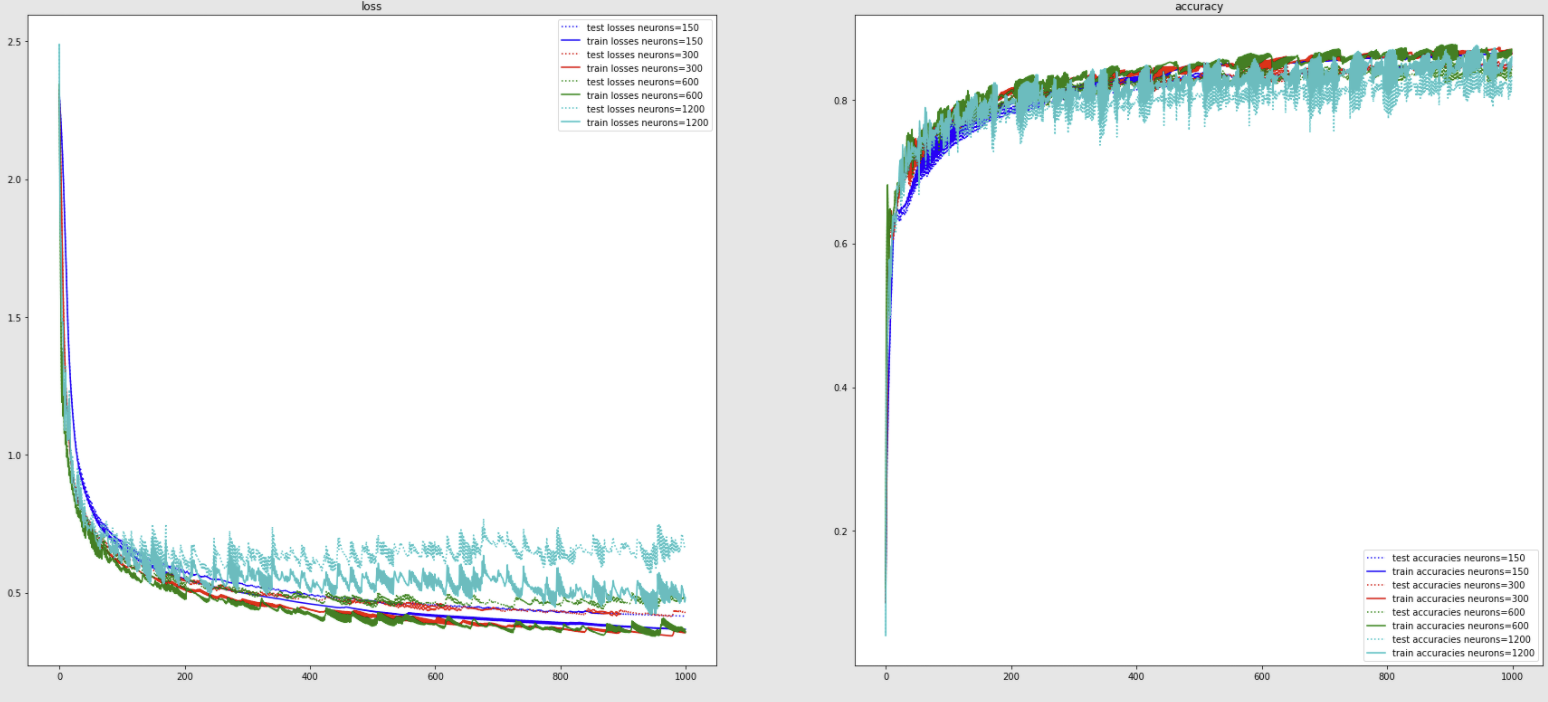
In the above, we see some overfitting starting from 200 iterations onwards, and things become significantly bad after 600 iterations.

The number of neurons in each layer was varied, and the best accuracy and loss was taken from the iterations and plotted. The x-axis shows the number of neurons in layer 1, but the number of neurons in other layers have also been scaled by a similar number.

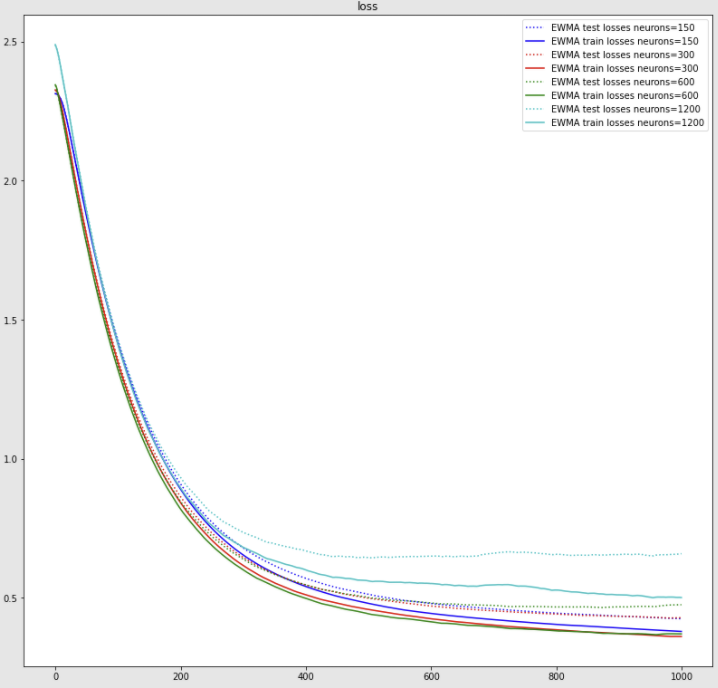


The differences here become clearer than the previous graph. Beyond 300 neurons, there seems to be little benefit.

When all the epochs were plotted, the graph looked like this:



Again, since the graph was very noisy, a rolling average was taken. We also see that with a higher number of neurons, the graph is more noisy. This is expected as the function becomes too complex and even a small change in parameters changes the form of this function drastically and the decision boundary changes rapidly thereby affecting the loss.



The trends are very clear here, and the overfitting is even more as the number of neurons increase. Interestingly, as the number of neurons increase beyond 600, both the training and test losses start to become worse.

#### Take Aways

Deeper neural networks tend to overfit more. As the number of neurons increase, both test and train accuracy may suffer beyond a certain point. The results are consistent with the case of the shallower network above, but the same effects are even more pronounced.

### Question 1\_2\_1

The forward pass function is implemented as below:

**def** q\_1\_2\_1**():**

tr\_x**,** tr\_y**,** te\_x**,** te\_y **=** get\_fashion\_mnist\_data**()**

network **=** **[**

***# Layer 1, 300 Nodes, ReLu***

**(**make\_layer**([**300**,** 784**],** "L2 Weghts"**,** **),**

make\_layer**([**300**,** 1**],** "L1 Bias"**,** **),**

tf**.**nn**.**relu**,**

**False),**

***# Layer 2, 100 nodes, ReLu, dropout (dropout applied on input)***

**(**make\_layer**([**100**,** 300**],** "L2 Weights"**,** **),**

make\_layer**([**100**,** 1**],** "L2 Bias"**,** **),**

tf**.**nn**.**relu**,**

**True),**

***# Layer 3, 10 nodes, Softmax***

**(**make\_layer**([**10**,** 100**],** "L3 Weghts"**,** **),**

make\_layer**([**10**,** 1**],** "L3 Bias"**,** **),**

softmax**,**

**False),]**

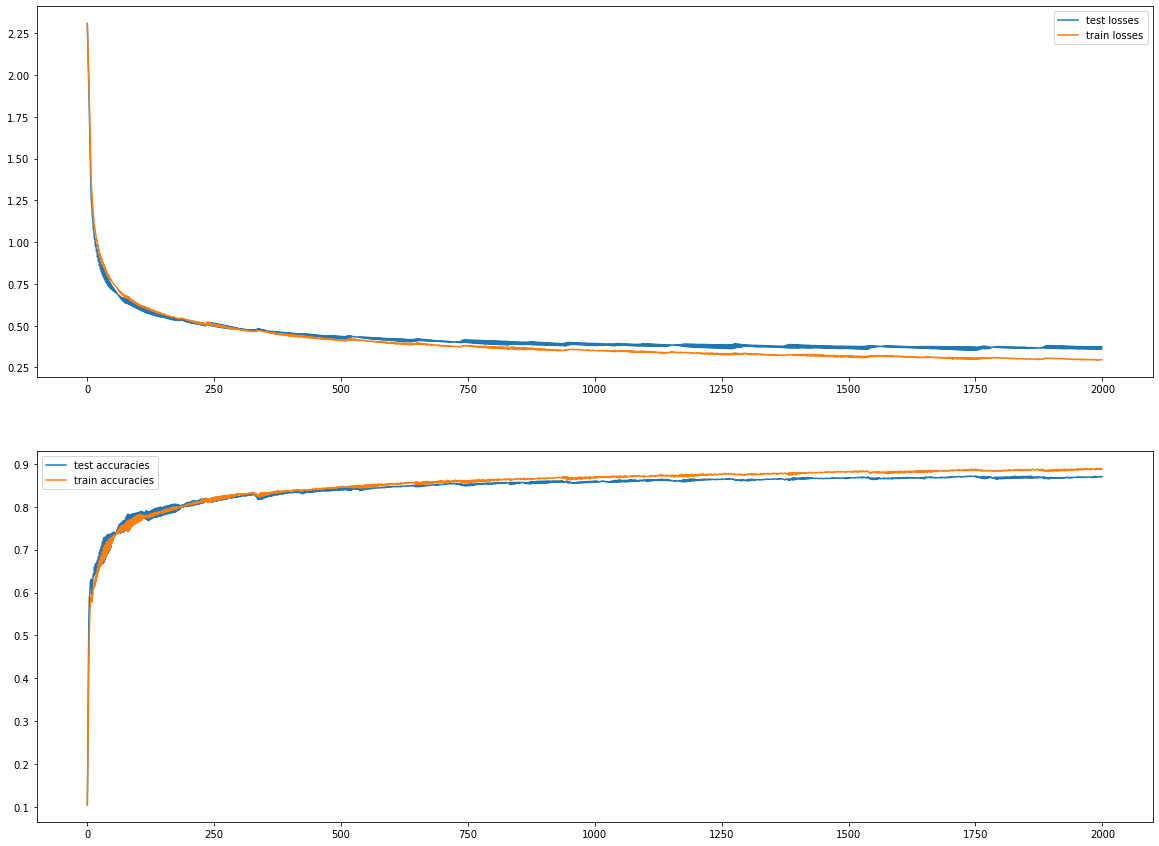
nn **=** NeuralNetwork**(**network**,** 0.5**)**

nn**.**train\_network**(**2000**,** tr\_x**,** tr\_y**,** te\_x**,** te\_y**,** cross\_entropy**,** accuracy**)**

**return** nn

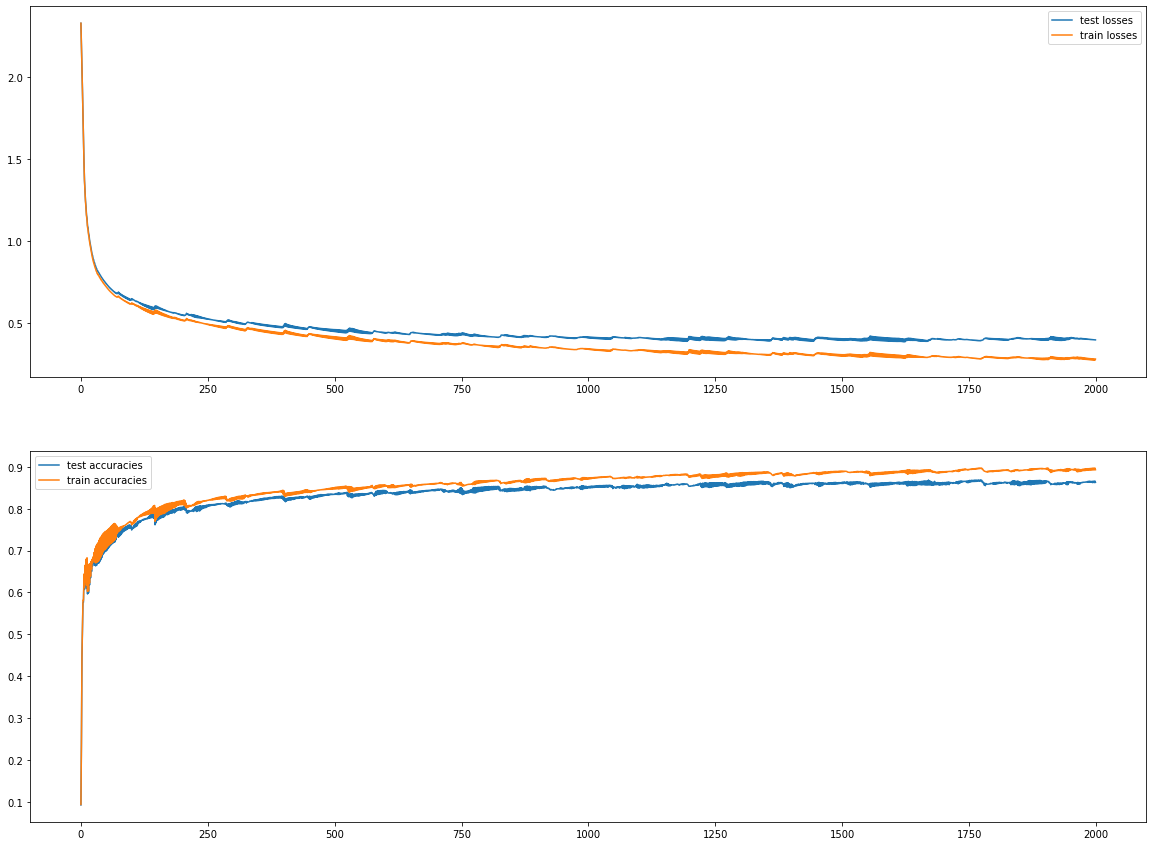
In the above implementation, 0.5 is the dropout probability.

The training and the test losses and accuracies were plotted with dropout.



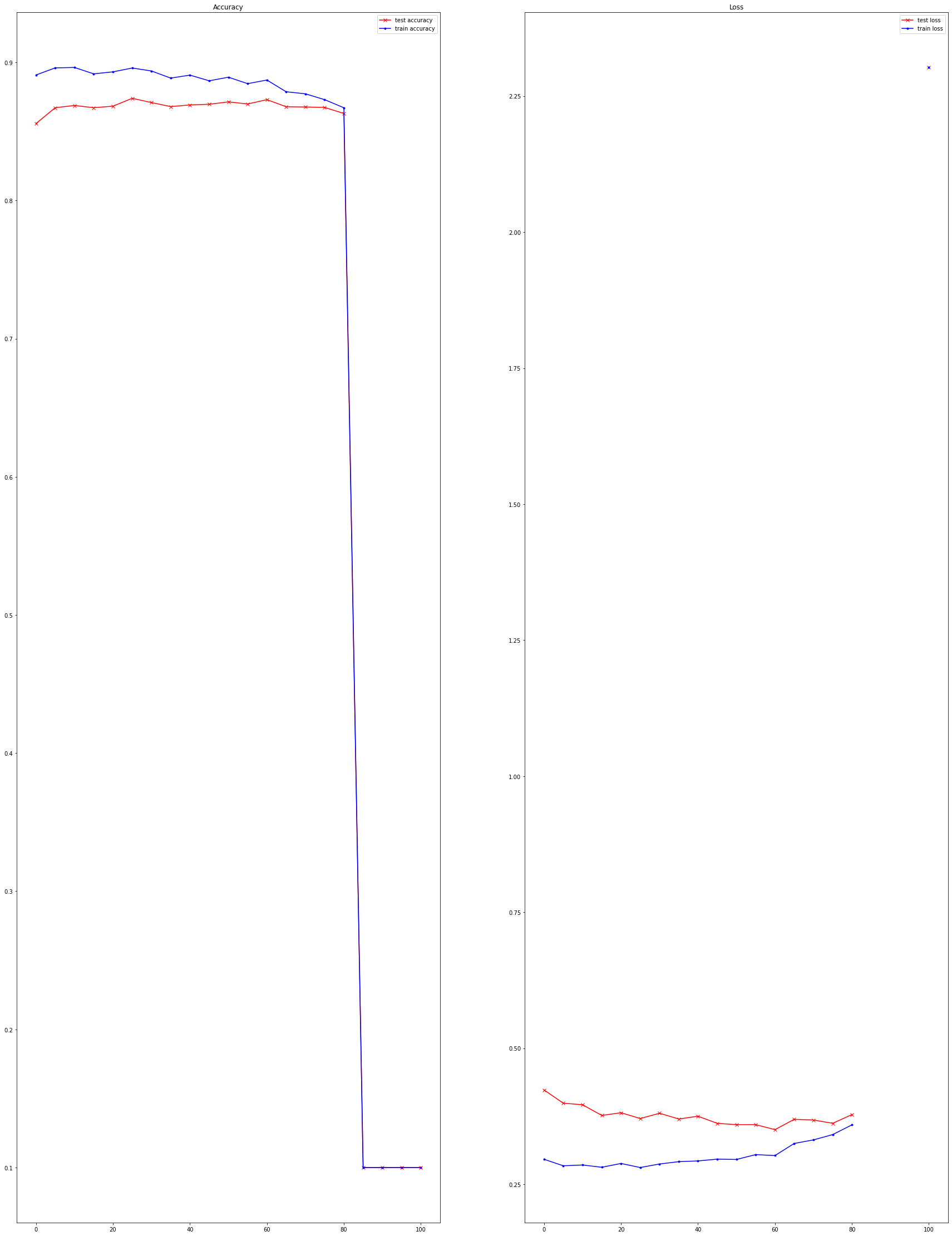
We start seeing some overfitting beyond 750 iterations, but not too much.

When the same thing is tried without dropout, the results are much worse



Without dropout, we see overfitting starting with 250 iterations, and the level of overfitting becomes much worse than the dropout case.

As an additional test, the dropout percentage was varied, and the losses and accuracies plotted.



From the above we can observe that the train losses are their best between 15 and 25%, and beyond that the train losses start to become worse. However, the test losses keep on decreasing all the way upto 60% dropout beyond which there is no improvement in the test losses. As the dropout rate goes higher, the gap between training and test losses decrease – this is consistent with theory as dropout is expected to reduce the level of overfitting. Beyond 80% dropout, the accuracy tanks. This is also consistent with theory as beyond this point, too many inputs are zero for the neural network to be able learn anything, and the neural network just doesn’t have enough information to converge on anything.

### Question 1\_3\_1

Implementation Details:

The mean absolute error is implemented as such:

# This function finds the mean absolute error

**def** mean\_absolute\_error**(**y\_true**:**tf**.**Tensor**,** y\_pred**:**tf**.**Tensor**)->**tf**.**Variable**:**

y\_true **=** tf**.**transpose**(**y\_true**)**

logging**.**debug**(**y\_true**.**shape**,** y\_pred**.**shape**)**

abs\_dist **=** tf**.**math**.**abs**(**tf**.**math**.**subtract**(**y\_true**,** y\_pred**))**

**return** tf**.**reduce\_mean**(**abs\_dist**)**

The forward pass is implemented as below:

**def** q\_1\_3\_1**():**

tr\_x**,** tr\_y**,** te\_x**,** te\_y **=** get\_fashion\_mnist\_data**()**

tr\_x\_noisy **=** get\_noisy\_images**(**tr\_x**)**

te\_x\_noisy **=** get\_noisy\_images**(**te\_x**)**

network **=** **[**

***# Layer 1, 128 Nodes***

**(**make\_layer**([**128**,** 784**],** "L1 weights"**),**

make\_layer**([**128**,** 1**],** "L1 bias"**),**

tf**.**nn**.**relu**,**

**False,),**

***# Layer 2, 64 nodes***

**(**make\_layer**([**64**,** 128**],** "L2 weights"**),**

make\_layer**([**64**,** 1**],** "L2 bias"**),**

tf**.**nn**.**relu**,**

**False,),**

***# Layer 3, 32 nodes***

**(**make\_layer**([**32**,** 64**],** "L3 weights"**),**

make\_layer**([**32**,** 1**],** "L3 bias"**),**

tf**.**nn**.**relu**,**

**False,),**

***# Layer 4, 64 nodes***

**(**make\_layer**([**64**,** 32**],** "L4 weights"**),**

make\_layer**([**64**,** 1**],** "L4 bias"**),**

tf**.**nn**.**relu**,**

**False,),**

***# Layer 5, 128 nodes***

**(**make\_layer**([**128**,** 64**],** "L5 weights"**),**

make\_layer**([**128**,** 1**],** "L5 bias"**),**

tf**.**nn**.**relu**,**

**False,),**

***# Layer 6, 784 nodes***

**(**make\_layer**([**784**,** 128**],** "L6 weights"**),**

make\_layer**([**784**,** 1**],** "L6 bias"**),**

tf**.**nn**.**sigmoid**,**

**False,),**

**]**

nn **=** NeuralNetwork**(**network**)**

nn**.**train\_network**(**3000**,** \

tr\_x\_noisy**,** tr\_x**,** \

te\_x\_noisy**,** te\_x**,** \

mean\_absolute\_error**,** mean\_square\_error\_image**,** \

use\_argmax**=False)**

**return** nn

**import** matplotlib**.**pyplot **as** plt

nn **=** q\_1\_3\_1**()**

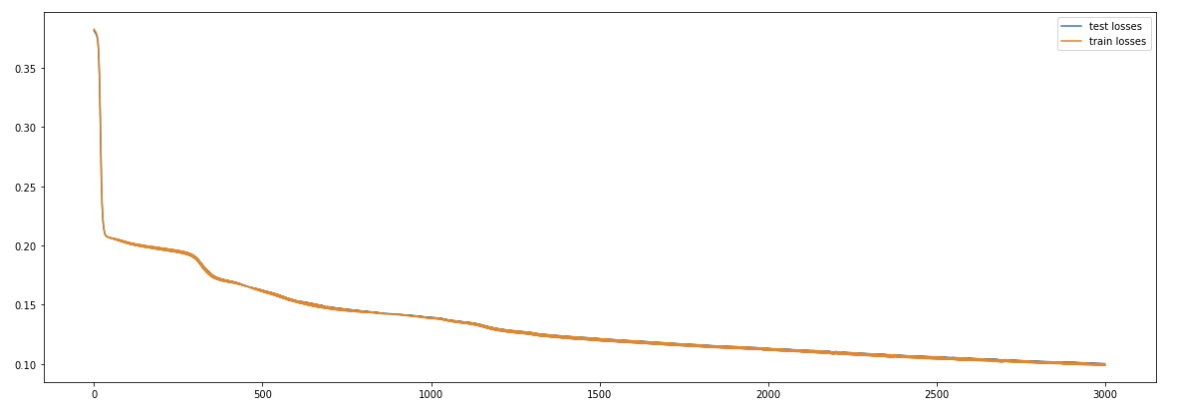
tr\_x**,** tr\_y**,** te\_x**,** te\_y **=** get\_fashion\_mnist\_data**()**

te\_x\_noisy **=** get\_noisy\_images**(**te\_x**)**

denoised **=** nn**.**predict**(**te\_x\_noisy**,** **None)**

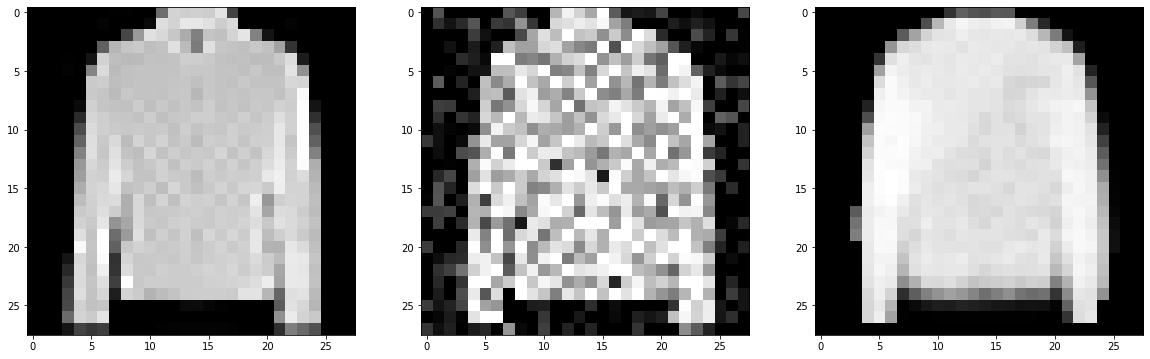
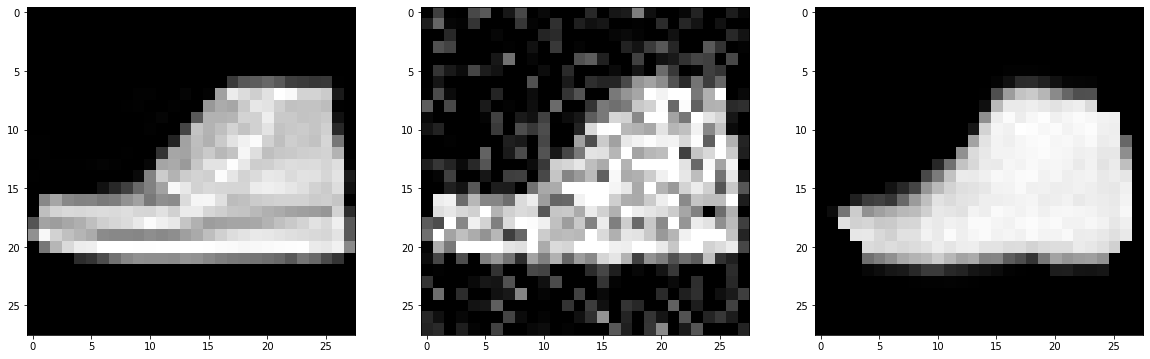
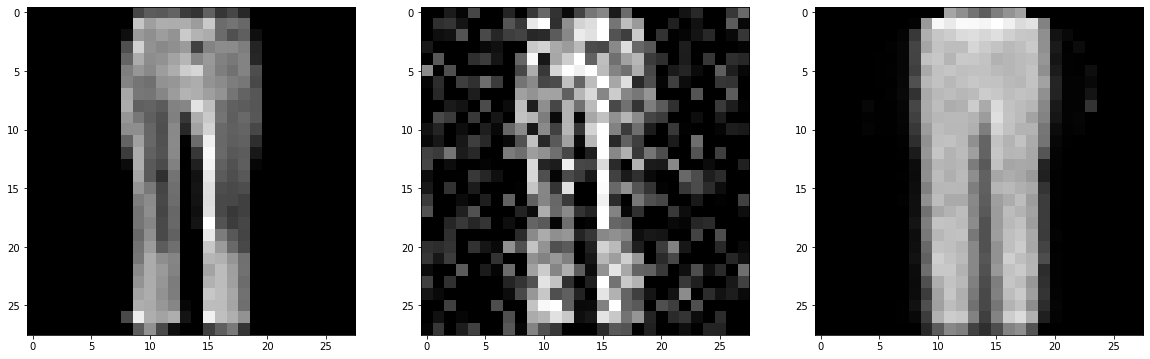
**print(**denoised**.**shape**)**

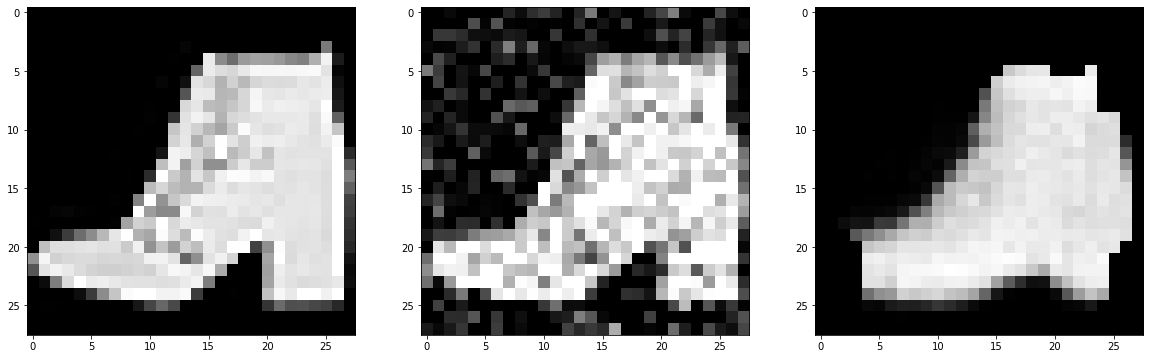
A denoising autoencoder was implemented and the losses plotted:



Here we can see the effects of the non-convexity of the function. The first few iterations lead to a large decrease in the loss, but then we see areas of very slow progress interspersed with some areas of better progress. No overfitting is observed.

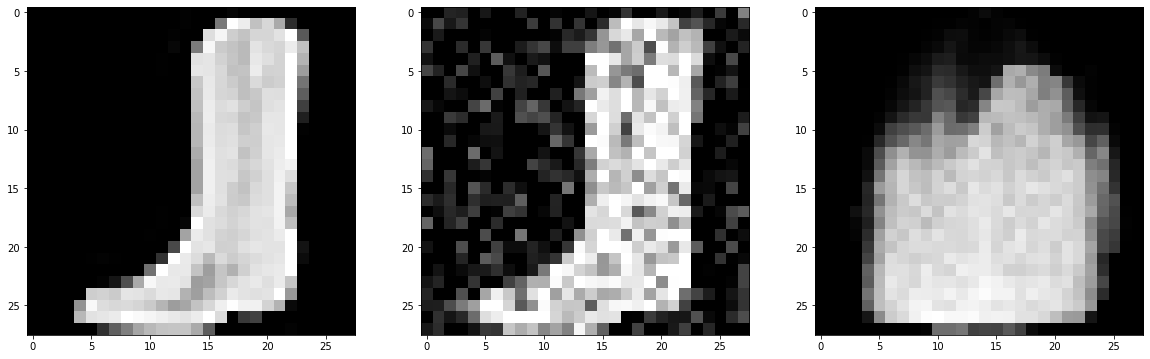
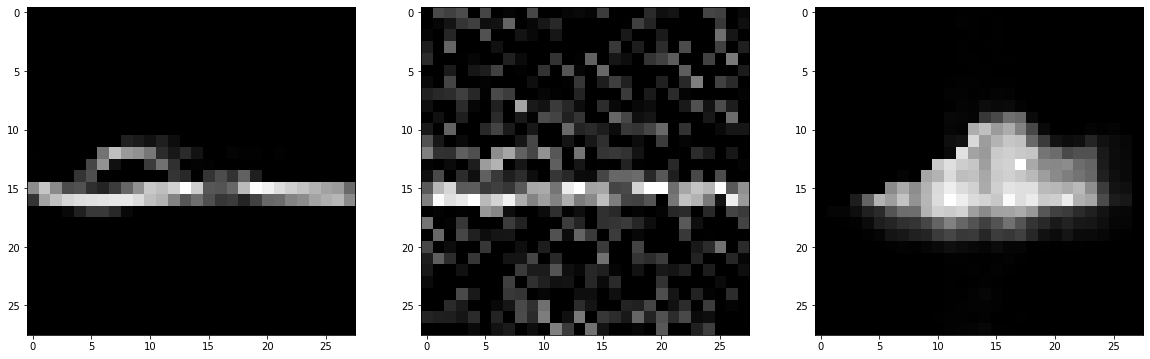
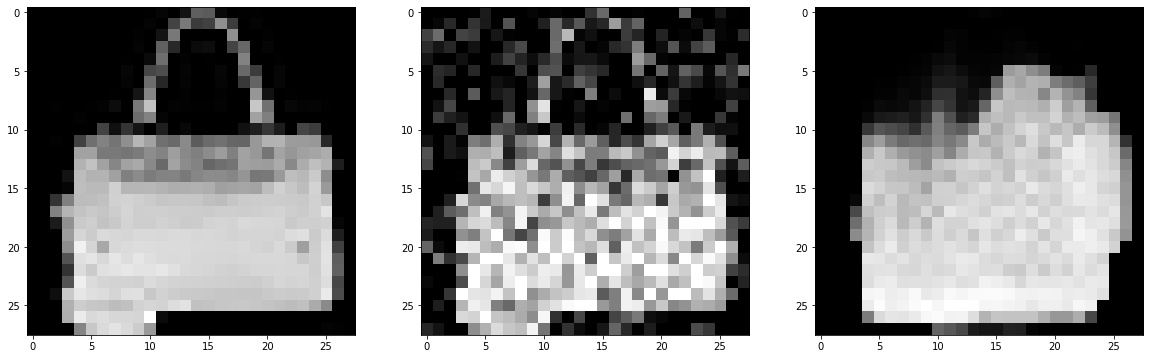
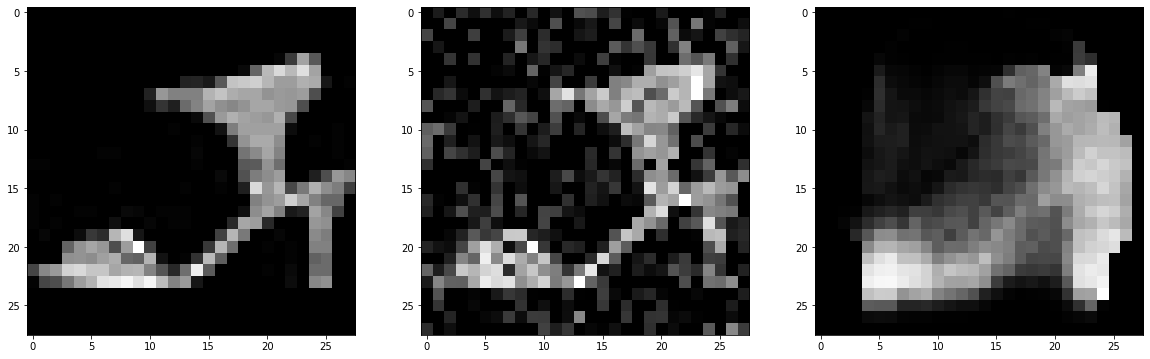
Some sample images are shown below, the three images in each row are the original image, the image with noise, and the reconstructed image. We see that for some images, it does a relatively good job of reconstruction.



However, it often gets things wrong as in the images below. A few things to notice here is that

1. Where the regions have small features, like straps of bags, straps of signals, the autoencoder struggles to reconstruct the original image. While for images, a DNN based auto-encoder is not the right choice as it ignores any spatial relationship between pixels and just approaches everything by the histogram approach, the same phenomenon has also been noted in literature with CNN based auto-encoders which also struggle with small features while getting larger features right.
2. In some instances it gets things completely wrong, eg. reconstructing a boot as a bag.

# Autoencoders

Traditional, auto-encoders take form where they learn a representation of the true data instances D and represent it in a lower dimensional space. This part of the autoencoder is called the encoder. The decoder then takes this lower-dimensional representation and then tries to reconstruct the original input.

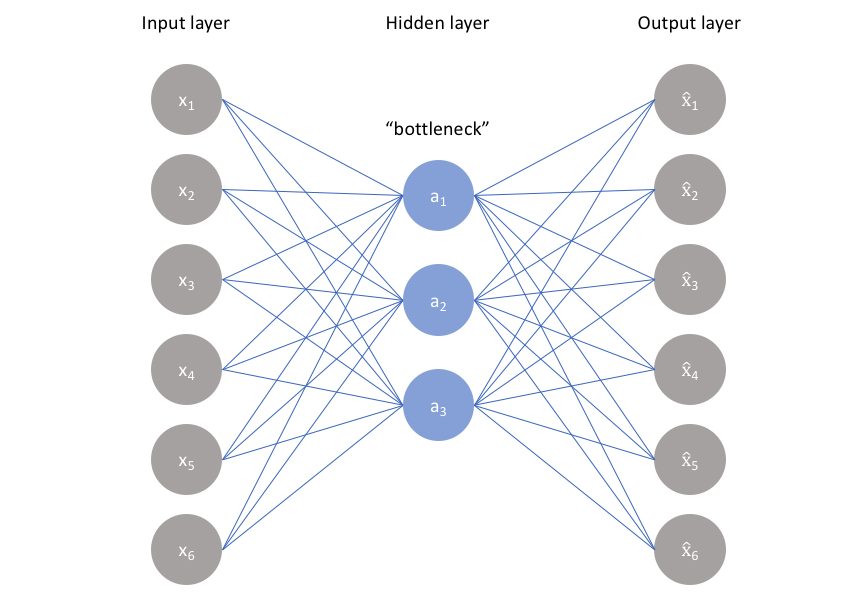


Figure 1https://www.jeremyjordan.me/autoencoders/

The above is not strictly true as the bottleneck can be of two types

1. It can have fewer number of parameters than the original input layer. This type of autoencoders are called undercomplete auto-encoders
2. It can have an even bigger space than the original input. This type of autoencoders is called overcomplete autoencoders. In over-complete auto-encoders, special care must be taken so that the network doesn’t just duplicate the input as the output, because the network will try to do just that if it has a chance.

The use of undercomplete hidden layer is most common[[1]](#footnote-1), however over-complete auto-encoders produce good results in denoising tasks.

Applications of autoencoders include

1. Denoising – this is one of the most common uses. The training method here is to inject noise into images, and let the autoencoder try to produce the original image by denoising. The loss function used is usually a MSE of each pixel of the original image and the denoised output.
2. Dimensionality Reduction and Feature Extraction. Given a large number of features, the bottleneck can be said to be a mapping onto a smaller set of features that describes the instance. In this model, the decoder can be discarded after training, and the output of the bottleneck layer passed to other modules. This use is similar to PCA, except that there is a non-linearity involved, and PCA is strictly linear.
3. Video and image compression and decompression. However, use of Autoencoders for image and video compression is not very popular as better methods exist.
4. Converting grayscale images to color, and vice versa.

Potential Disadvantages of Traditional Autoencoders

1. The images generated by autoencoders are often blurry, especially towards the periphery. This is also a problem with variational autoencoders.
2. Autoencoders are good at generating large features, like eyes, mouth etc, but not good at generating smaller features like hair etc.
3. The latent space is discrete and there is no continuous probability distribution function by which it can be modeled. Hence it is difficult to generate new instances by discarding the encoder part of the network and providing randomized inputs to the decoder as we will not know how to model those inputs.
4. Often there are gaps in the latent space, and the classes in the latent space are not always separable.

The last two problems are illustrated in the following diagram:

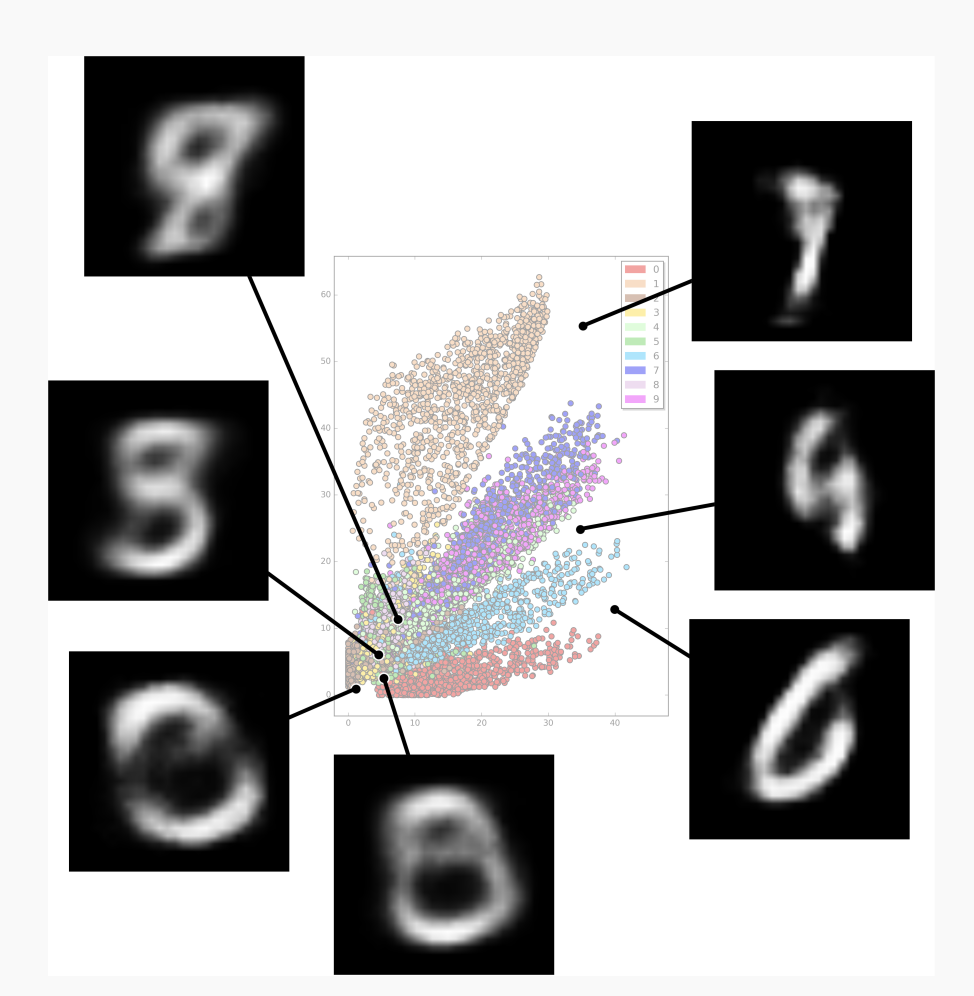


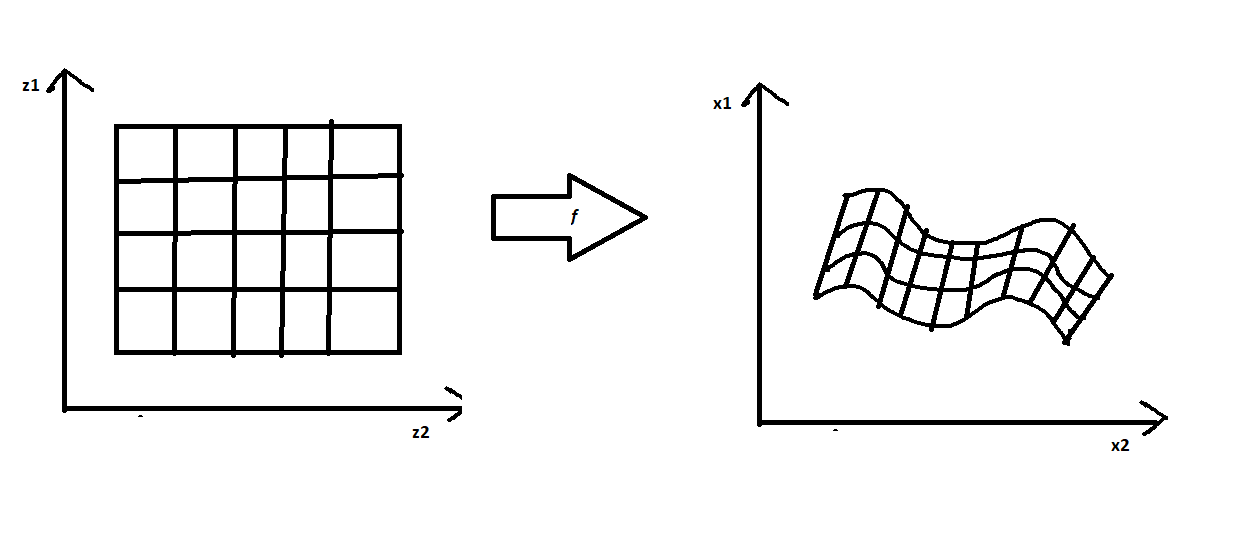
Figure Latent space representation for MNIST dataset. <https://towardsdatascience.com/generating-images-with-autoencoders-77fd3a8dd368>

# Variational Autoencoders[[2]](#footnote-2)

Traditional, auto-encoders take form where they learn a representation of the true data instances D and represent it in a lower dimensional space. This part of the autoencoder is called the encoder. The decoder then takes this lower-dimensional representation and then tries to reconstruct the original input.

In some ways, the lower dimensional representation contains the essence of the data and from this lower representation (which we shall call *z* from now on) contains the essence of the data, and from this *z* it is possible to generate new instances which closely resemble the input data. These also have analogies with PCA, however, PCA doesn’t have any non-linearity and auto-encoders have a non-linearity which may be able to represent data better.

Conceptually we can think of a function *f* that maps *z* 🡪 X where X closely resembles the input data.



In case of auto-encoders and variational auto-encoders, *f* is a neural network.

Consider the case of generating hand-written digits. The generating model should recognize all the interdependencies of the pixels, and should produce an image that makes sense. For example, if the left half of the picture looked like a 5, and the right half of the picture looks like a 3, by some measures of similarity the final distribution would still have high correlation but would be completely useless. It makes intuitive sense if the generator first decides that it is going to draw a ‘3’ and then maps that decision to a set of pixels. Furthermore, there could be additional decisions that it might need to make first – like the font, slant etc. These decisions are what correspond to *z* in our discussion.

Ideally we don’t want to hand-code the features of *z* and we would like the model to learn these characteristics itself. However, in some cases the learned *z*’s can be augmented by additional parameters that could be hand-coded.[[3]](#footnote-3)

Therefore, the crux of the problem is to find a *latent encoding z* that can represent the data, and find a model *f* that can generate new instances X that resemble closely the input data. Typically *f* is a neural network.

*Once z and f have been learned, we can then discard the encoder, and just retain the decoder. The way to generate new instances after the training phase is to randomly sample for z over some probability density function and pass it to the decoder to generate new instances.*

This last part is different from traditional auto-encoders where traditionally there is no way to sample random instances of the latent variable *z* and pass that to the decoder to generate new instances. In traditional auto-encoders, *z* is arrived at by passing the input data through a deep learning model like a neural-network, and there is a 1:1 correspondence between an input data instances and its latent representation.

It is often said that Variational Auto-encoders are a Bayesian spin on auto-encoders. The mathematical basis of variational auto-encoders is very different from classical autoencoders. They are called autoencoders because the apparatus used for training has similarities with traditional auto-encoders. Unlike denoising auto-encoders, we don’t need the input data after the training phase is over, and we can sample Z over P(Z) and get by with it.

The learning objective in variational autoencoders is to learn a mapping from some latent real value *z* to a possibly complex distribution on X̂ , where X̂ is similar to data X.

Here we are assuming that P(X̂) is similar to P(X), since that is what we want to achieve, and in this context, P(X̂) and P(X) can be used interchangeably.

We can write P(X) as follows.

where

We also assume that is something simple, as this allows us to easily model the distribution. Usually it is chosen to be a Gaussian, but it can be a different distribution. We also have a function *f* which can take an arbitrary *Z* and create a generated data point X̂. This function *f*, as we have said earlier, is the decoder.

The goal is to learn if there is a set of underlying latent representations Z and a function *f* which allow us to do that. Typically we will want to learn a probability distribution over Z, and then sample from the distribution and pass the sample through *f* to get our generated output.

As an example, if we are generating faces, we can have two latent variables identity and expression, and *f* can sample from these two variables and produce an output face. We further assert that both identity (Barak Obama, Donald Trump, etc. ) and mood (happy, sad, etc.) are governed by a simple probability distribution function, and in the training phase, we will learn the parameters of those probability distribution function. The form of those distribution functions is pre-determined (eg. We first choose whether we assert those are Gaussians or Bernoulli). *f* is usually a neural network.

However, for this to work, we need to first know the distribution Z. But we don’t even know what Z’s look like before we have trained the model, and since we don’t know that, therefore P(Z|X) is intractable. This becomes a further problem because we need the values of Z to train *f*.

Schematically, what we want is:

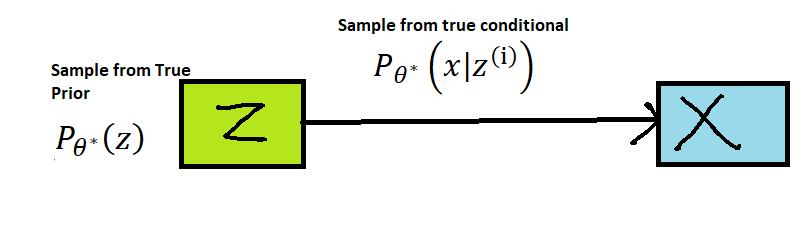


Figure 3Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Now the challenge is that without knowing what the Z’s we need to estimate the parameters of .

VAE’s allow us to do that by assuming that there is a known prior distribution over Z, usually a diagonal Gaussian, that closely resembles the probability distribution of the original data.

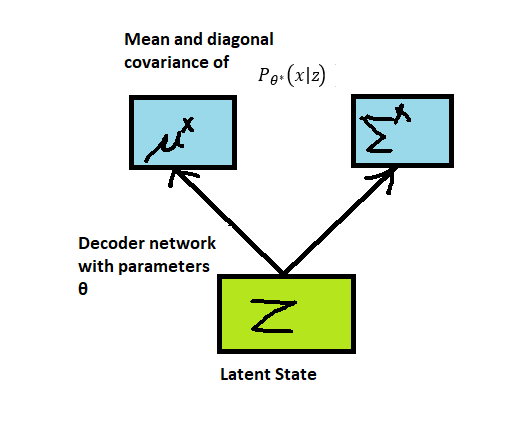


Figure 4Fei-Fei Li, Andrej Karpathy and Justin Johnson, CS231n, Stanford University

The decoder network predicts the means and covariances of the multi-variate diagonal Gaussian. We need to learn the latent variables Z for a particular set of input X, and we can use Baye’s rule as below[[4]](#footnote-4).

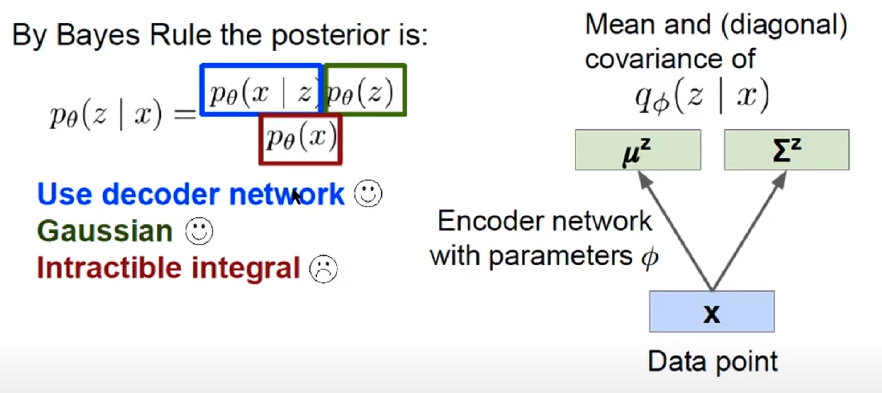


Figure 5Fei-Fei Li, Andrej Karpathy and Justin Johnson, CS231n, Stanford University

Here we have assumed to be Gaussian. We can find out from the decoder network during training. However is an intractable integral.

To circumvent this challenge, it is assumed that there is some other distribution similar to the original distribution, encoder network is used to estimate . The encoder network is another neural network, and it outputs a set of means and variances of a multivariate diagonal Gaussian distribution.

The next step is to make similar to . Witten in a different form, the goal is to maximize the log likelihood of when Z is sampled over . This can be written as:

Which can be re-written as

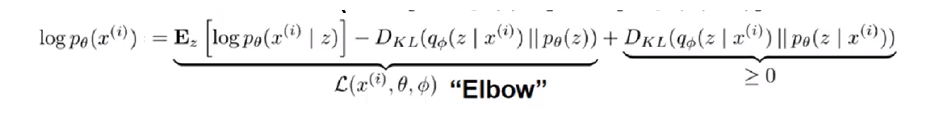


Figure 6 Fei-Fei Li, Andrej Karpthy, Justin Johnson, CS231n, Stanford University

Where is the expectation.

The first two terms are called the Elbow or evidence lower bound. The third term, by definition of KL Divergence is 0 or positive. Therefore, maximizing the first two terms results in maximizing the lower bound of this log likelihood.

Maximizing the lower-bound is not the same as maximizing the log likelihood itself, but gives us some room and makes the problem simpler.

This method of optimization is called variational optimization, where variational distribution is introduced and that is employed to learn the parameters of the original distribution

is parametrized with the encoder network, and with the decoder network, and the process of training will ensure that approximates . The encoder outputs a set of means and variances which describe the approximate posterior distribution which should be as close to the original prior distribution .

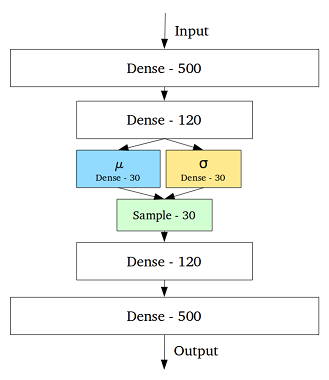
The challenge with the model where the decoder samples vectors from the distribution output by the encoder. This causes an issue with backpropagation since a sampling operation is not deterministic and makes it difficult to pass the gradients backwards in backpropagation. This is resolved by the use of the *reparameterization trick[[5]](#footnote-5).* This is achieved by making the randomness of the sample independent of the learnt distribution, and then we can treat the randomness as an extraneous factor.

Ideally z is a real vector sampled from the latent space such that .

However, with the reparameterization trick, instead of sampling z, z is chosen as follows: where

Since now is a vector randomly sampled over the unit normal distribution which is independent of input, and instead of sampling, we just multiply the output mean and variance with this number, backpropagation can now proceed as usual. can now be treated as external noise.

To put it all together, a typical VAE in the training phase may look like this[[6]](#footnote-6):



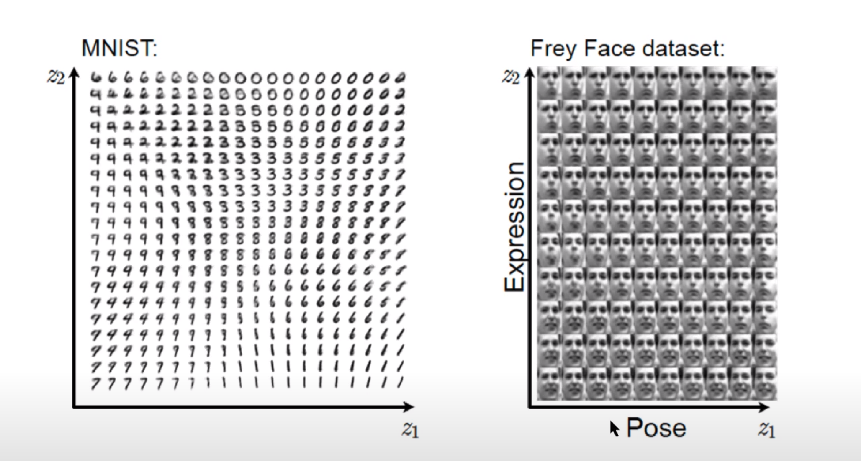


Figure 7 Aaron Courville, Deep Learning Summer School, 2015

The above shows how generation works for VAEs. In the above the input vectors Z’s were changed and used to produce images. We can see a continuous predictable change in the output image as the latent representation is changed.

Some uses of VAE’s are[[7]](#footnote-7)

1. Image and video generation
2. Superresolution, to enhance the resolution of an image
3. Forecasting from static images
4. Image in-painting

There are several extensions to VAEs, some of which are:

1. Semi Supervised VAEs
2. Conditional VAE - Traditional VAEs work on unlabeled data, but if labeled data is given, VAE's can learn to generate images of a particular class label.
3. Importance-Weighted VAE - It weights the random variables differently to get better results.
4. Denoising VAE
5. Inverse Graphics Network - It uses a VAE latent representation to modify graphics like changing the angle of lighting etc.
6. Adversarial Autoencoders - Combines GAN and VAE

Advantages of VAE

1. The latent space is continuous. This allows us to vary one each parameter in a predictable way to get a predictable output. For example, we can have a variable called ‘happiness’ in face generation, and very it from ‘sad’ to ‘happy’, and get different degrees of happiness in the image.
2. The latent space is better separated than traditional VAEs

Disadvantages of VAE

1. VAE’s produce blurred outputs, especially towards the periphery of the image. The most common explanation of this is that this is the result of taking Gaussian which makes things imprecise. However, some other studies suggest that this is because that the choice of the probability distribution function may not represent the distribution of the data. [[8]](#footnote-8)
2. We force a simple probability distribution on the data and the latent features, but the actual distribution may be complex and a simple Gaussian may not be able to model it well.

# Adam Optimization Algorithm

One practical problem with gradient descent is that it can often take too long to converge. There are many different problems associated with gradient descent, and many different approaches have been proposed to make gradient descent. Each of those algorithms have some pros, but have some drawbacks as well. Subsequent algorithms have tried to remedy those drawbacks, and the sequence of proposed models resulted in Adam, which has been very successful in practice. Although the question is specifically about the Adam algorithm, going through all the different algorithms will allow for a nuanced understanding of the motivation behind Adam.

If we take the simplest case of logistic regression, the gradient update step takes the following form:

And

Note that is dependent on , and if is 0, then the derivative is also zero. *This means that if a particular feature is sparse, most of the time, we will get the derivative as 0, and progress in learning along that feature may be very slow.*

Another consequence of the mathematics of the gradient descent is that good progress is made in areas where the gradient is steep, but very little progress is made in areas where the gradient is more flat.

In an ideal case, the perfect function for fastest convergence is a bowl shaped function, but in reality that is hardly the case, and the surface may be very complex. Consider the relatively simple gradient plot with a single variable:

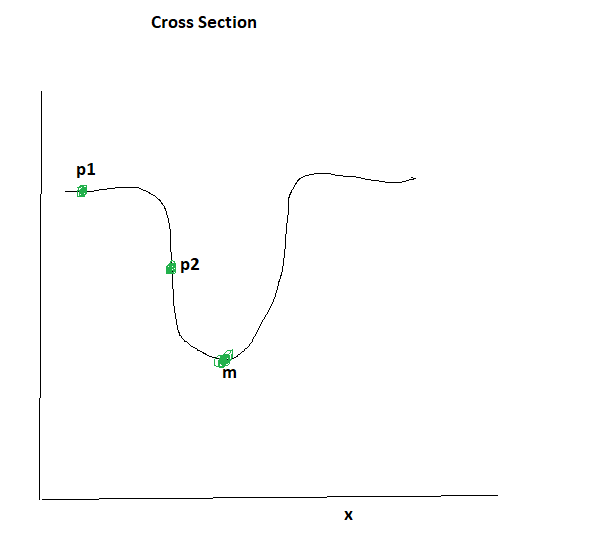


Figure 8 Gradient Descent at different Slopes

Here, when we start off with P1, the gradient is almost flat, and we will make very little progress. When we have reached P2, we will make rapid progress towards the minima m.

The cost functions of neural networks are non-convex functions, and these problems are very common in neural networks.

The case with multiple features is even more complex. One of the features may have a different contour than the other. *Learning may be very fast along one feature, but very slow in the other direction, and may even oscillate in the other direction.*

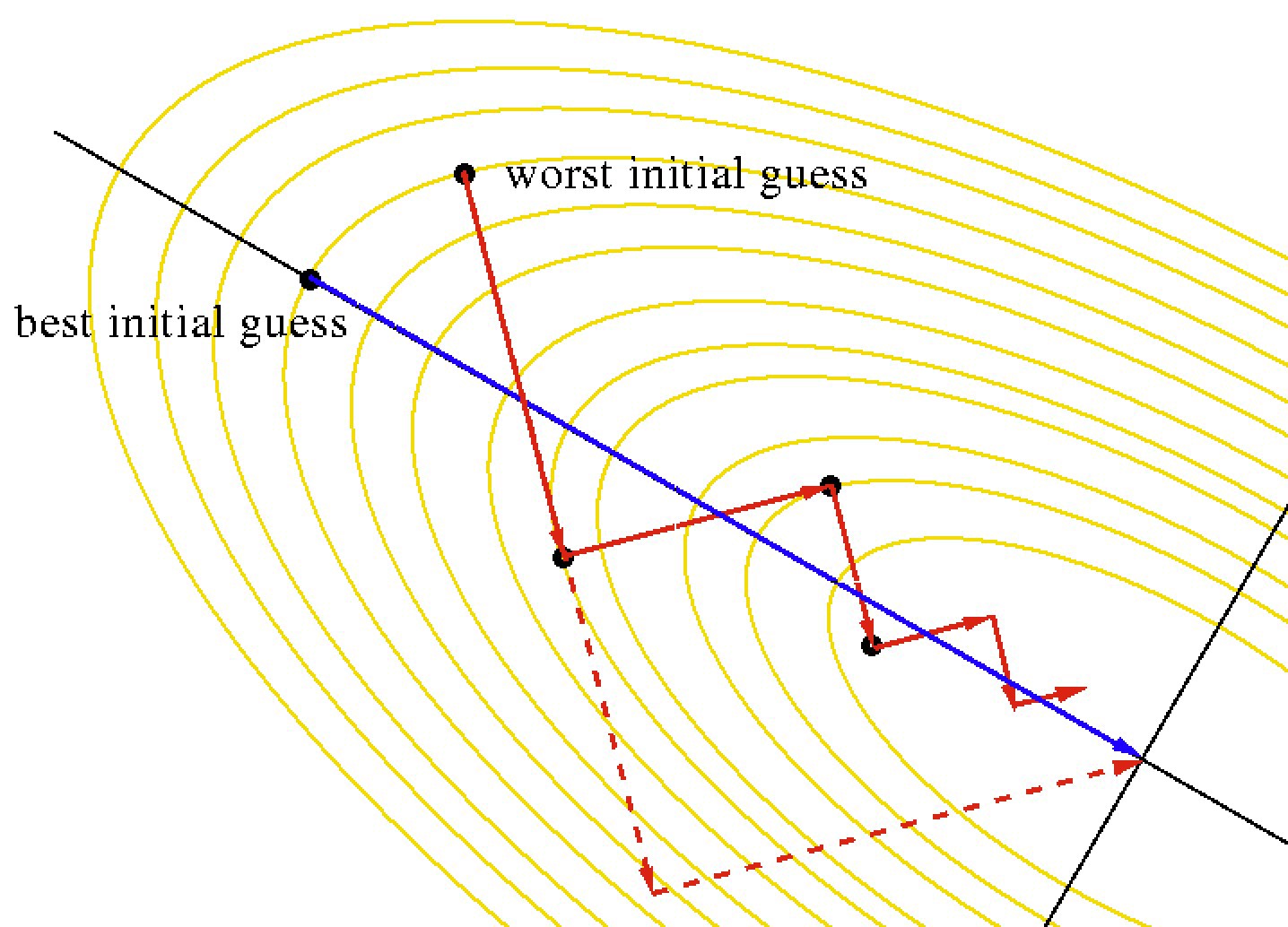


Figure 9 <https://towardsdatascience.com/learning-parameters-part-4-6a18d1d3000b>

The Adam optimization algorithm can best be explained by explain other techniques to solve this problem, as Adam has evolved from these methods, and is a combination of some of these.

**Gradient Descent with Momentum** is a very simple method that tries to solve the problems of slow gradient descent when the initial point is in a relatively flat area of the curve, as P1 in Figure 8 Gradient Descent at different Slopes. Here the problem is that since the surface has a low gradient, initially, the rate of optimization will be very slow. The main idea to mitigate this problem is to use the history of gradient updates. If we have made several small updates in a particular direction, then it is very likely that that is the direction in which the minima lies. We leverage this idea by increasing the learning rate if we have made several improving updates in one direction. Mathematically, this is achieved by taking the exponentially moving average, and use that to scale the learning rate. The update rule for gradient descent with momentum look like these:

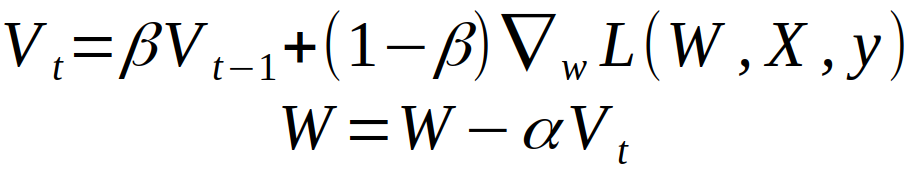


Figure 10 <https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d>

The first term is just the exponential weighted average of the gradients over the iterations, and we always update each weighted by this.

Gradient Descent with Momentum helps us move faster in areas where the gradient is low. It also helps us achieve faster convergence when there are sparse features, because it takes the weight of more than one instances while updating the gradient. Lastly, in Figure 9 https://towardsdatascience.com/learning-parameters-part-4-6a18d1d3000b , we took a very zig zag route along one axis, and not along the other, and the momentum term would dampen the zig zagging across that direction somewhat, and at the same time, the learning along the other direction would be faster.

While initially, momentum would cause the convergence to proceed faster towards the minima, it could have the drawback of overshooting the minima because we already have momentum in that direction. *This could lead to some back and forth iterations near the minima.* This is illustrated in the contour map below, as we reach the minima, we start a lot of oscillations back and forth.

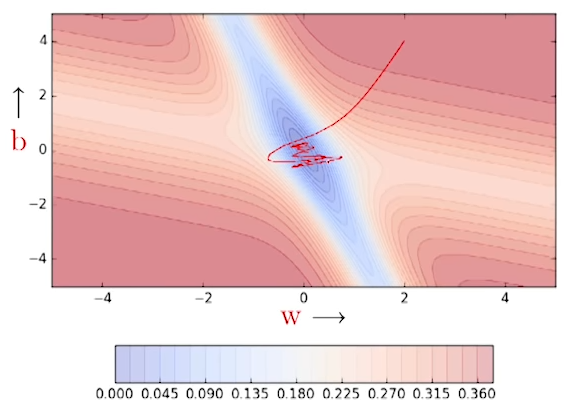


Figure 11 Slides from NPTEL Deep Learning, https://www.youtube.com/watch?v=ewN0vFYFJ7A&list=PLyqSpQzTE6M9gCgajvQbc68Hk\_JKGBAYT&index=37

Another aspect of the same figure above is to see the sigmoids, as we cross over from one end of the minima to the other end, the sigmoids change direction.

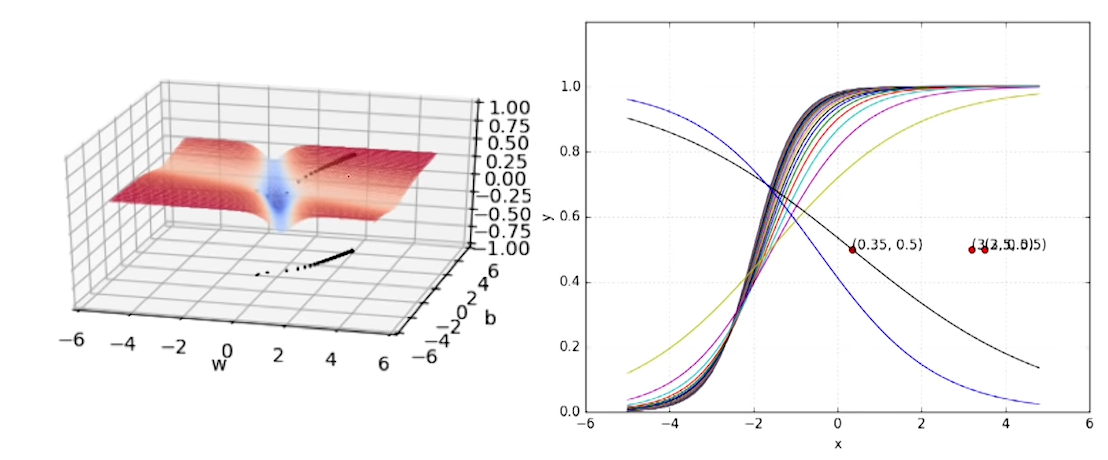
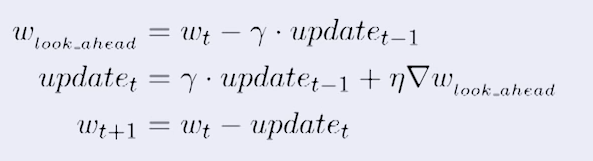


Figure 12 Slides from NPTEL Deep Learning Course, <https://www.youtube.com/watch?v=ewN0vFYFJ7A&list=PLyqSpQzTE6M9gCgajvQbc68Hk_JKGBAYT&index=37>

To remedy this back and forth near the minima, **Nesterov Accelerated Gradient Descent** is used. This method updates the gradient descent slightly; the idea here is to look before leaping, and if the update will make us cross the minima, we will remedy it. Mathematically, it is implemented by calculating the gradient not at the current position, but at the next position where we expect it to be, essentially making it a 3 step process as below.



Here is the term we get from the momentum. We add this to get the expected point where we would land, and then compute the gradient. If we have crossed the minima at the expected point, then the gradient would change direction, and would change sign and would have a dampening effect on the step. The overall result is that the oscillations around the minima would be decreased.

These concepts will be helpful in understanding Adam and other optimization heuristics.

The classical gradient descent update equations have a learning rate, and we usually expect it to be fixed. However, there are several disadvantages to a fixed learning rate. If we fix this learning rate to be too low, then the model has a better chance of converging, but the learning may be very slow. On the other hand, if we fix this learning rate to be too high, then it might actually start diverging. Moreover, in different parts of the surface, we may need different values of the gradient. Therefore adaptively changing the gradient will be advantageous. The following algorithms adaptively change the learning rate. Furthermore, the following algorithms will try to use a different learning rate for each feature. Sparse features may need a higher learning rate and dense features may need a lower learning rate because of reasons we discussed earlier.

**AdaGrad** or **Adaptive Gradient Algorithm** tries to decay the learning rate for the parameters according to their update history. If the weights for the particular parameter has been updated very often, then it decays its learning rate. This achieves uniformity between the convergence of sparse and dense features. Mathematically, adagrad equations take the following form:

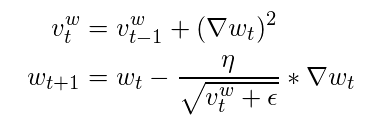
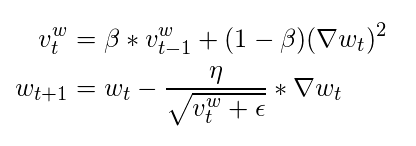


Figure 13 Adagrad Equation, <https://towardsdatascience.com/learning-parameters-part-5-65a2f3583f7d>

*Here we see that the square of the update is accumulated, and that is used to scale down the learning rate . The problem with AdaGrad is that at some point of time, may become so small that the gradients are no longer updated, and the model may never converge.*

**RMSProp** tries to remedy this problem of AdaGrad. RMSProp was proposed by G. Hinton in a coursera course, and has worked quite well. The idea in RMSProp is to prevent the denominator in the above equation to become too large. Instead of accumulating the gradients as a simple summation of squares, RMSProp uses an exponentially weighted average of the gradients, and uses that to scale the learning rate. A small value is added to avoid any division by zero problems.



One thing to note for both AdaGrad and RMSProp is the use of the square root in the denominator. Theoretically, not using the square root should work just as well, but practically it has been found that things work better when the square root is present.

The **Adam** algorithm is also known as **Adaptive Moment Estimation** combines both momentum and RMSProp into a single update step, and the combination of the two converges much faster than either of them. In addition, Adam employs bias correction to get more accurate estimations at the beginning of the training phase.

Adam also has two hyperparameters and , which control the momentum and RMSProp components respectively. Adam algorithm update step can be expressed by a series of expressions.

The first equation is the momentum part of the update.

The second equation is the RMSProp part

Now bias correction is applied on both and :

,

Finally the update step is performed as below:

As we can see above, the term captures the momentum aspect, and the learning rate is scaled by a factor consistent with RMSProp.

The algorithm is called Adaptive Moment Estimation because it uses the first moment and second moment of the gradients to come up with the update rule, and hence the name.

There has been some papers to show some theoretical datasets where Adam failed to converge [[9]](#footnote-9), but for real world problems, Adam works very well.

There have been papers which show that a Nesterov Accelerated Gradient Descent can perform as well as Adam, but it requires carefully setting the hyperparamters, and in practice, Adam performs well on a wide array of problems with little or no hyperparameter tuning.

1. <https://www.youtube.com/watch?v=bggWQ14DD9M> [↑](#footnote-ref-1)
2. Doersch, Tutorial on Variational Autoencoders, <https://arxiv.org/pdf/1606.05908.pdf> [↑](#footnote-ref-2)
3. Doersch, Carl. "Tutorial on variational autoencoders." *arXiv preprint arXiv:1606.05908* (2016). [↑](#footnote-ref-3)
4. <https://www.youtube.com/watch?v=psaDtRj_8oA&t=587s> [↑](#footnote-ref-4)
5. Salimans, Tim, Diederik Kingma, and Max Welling. "Markov chain monte carlo and variational inference: Bridging the gap." International Conference on Machine Learning. PMLR, 2015. [↑](#footnote-ref-5)
6. <https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf> [↑](#footnote-ref-6)
7. Doersch, Carl. "Tutorial on variational autoencoders." arXiv preprint arXiv:1606.05908 (2016). [↑](#footnote-ref-7)
8. Zhao et. al, Towards a Deeper Understanding of Variational Autoencoding Models, <https://arxiv.org/pdf/1702.08658.pdf> [↑](#footnote-ref-8)
9. Reddi,, et.al., On the Convergence of Adam and Beyond, <https://openreview.net/pdf?id=ryQu7f-RZ> [↑](#footnote-ref-9)