

Deep Learning Assignment 1

Note:

1. For brevity and ease of comparison I have presented the graphs in tabular format and hence the images might look small. Please zoom as and when needed.
2. In certain occasion during the PART A, accuracy model of my model tends to be more than expected. I think its because I have used `tf.round` in the calculate accuracy function. Ideally this function should be used in binary classification and `tf.argmax` would have been a better fit. However, with `tf.round` accuracy should have ideally dropped which hasn't happened. I haven't found the answer to this behavior but shall love to hear if you have an answer.
3. Despite this, I think nothing whatsoever shall get impacted in training and testing the model as I believe even with this accuracy the overall accuracy trend shall remain same.
4. Will my marks be deducted for volunteering to let you know this beforehand? 😊

Part 1 1

Below is the detailed documentation of a SoftMax classifier with 10 SoftMax neurons based on TensorFlow 2.1.

- Forward pass:

$$A = W^T X + b$$

$$[\lambda_1, \lambda_2, \dots, \lambda_n] \begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^m \\ \vdots & \ddots & & \vdots \\ x_n^1 & x_n^2 & \dots & x_n^m \end{bmatrix} + b$$

- The above image shows how the vector multiplications are to be done by multiplying the weights and input vector. The idea is here is very similar to machine learning where we try to **find out the more important features among the less important ones**.
- In neural networks this is done with the help of a weight matrix, which essentially tells how much would a feature contribute to the output value. And assignment of these weight matrices is then

optimized by performing a partial derivation of the loss function with respect to weights and biases. This is something we shall discuss in further detail below.

- Below is the code implementation of the above equation:

```
def forward_pass(x, w_T, b):

    '''Taking the transpose to allow matrix multiplication (60,000 * 784) ==> (784 * 60,000)'''
    x = tf.transpose(x)

    """ We need to multiply each training example by the weights and add bias"""
    y_pred = tf.matmul(w_T, x) + b

    """Pipe the results through the softmax activation function. """
    y_pred_softmax = softmax(y_pred)

#     print('***50+y_pred'+ '**50)
#     print(y_pred)
#     print('\n')

#     print('***50+y_pred_softmax'+ '**50)
#     print(y_pred_softmax)
#     print('\n')

    return y_pred_softmax
```

- In the given problem below are the shapes:

```
*****
Shape of x ==> (60000, 784)
Shape of x transpose==> (784, 60000)
Shape of W ==> (10, 784)
Shape of y_pred ==> (10, 60000)
Shape of y_pred_softmax ==> (10, 60000)
*****
```

- So essentially, a forward pass just tells us what would the output be for the all the 60,000 images given the weights and biases. Now, the catch is, each of the 10 neurons might come up with a different number which could be a positive or negative number (Like this one below). Which makes it a little difficult about which of the neurons we should trust on. This is where SoftMax comes in.
- Below Image shows the predicted pre-activation values by each neuron (over the validation data). It can be clearly observed that values distributed over the number line.

```
*****y_pred*****
tf.Tensor(
[[[-0.1549698 -0.63514316 1.0338681 ... 0.29420584 0.63779694
 -0.16447693]
 [-0.5918318 -1.0979135 1.4933379 ... -0.27001536 1.2650856
 -0.17851506]
 [-0.01592596 1.7101347 1.2753543 ... 0.6204642 0.9835879
 -0.07420595]
 ...
 [ 0.3257482 -0.9478956 -0.5272741 ... -0.6094035 -0.2051818
 0.2881257 ]
 [ 0.79592794 1.0878099 -0.3698877 ... 0.14669168 -0.76271844
 0.03840249]
 [ 1.0663555 0.58064395 0.24591137 ... 0.01577252 -0.38360205
 0.36142257]], shape=(10, 10000), dtype=float32)
```

- [SoftMax Activation function:](#)
- Below is the formula that represents the SoftMax Activation function. **Now, what this function essentially does is** - finds the neuron who is the most confident about the target class of the given data (image in our case). If we see at the equation below, it is the ratio of the exponential of the value given by j^{th} neuron to the summation of the exponential of the value given by each neuron. Exponential is done to restrict the value in between 0 and 1.
- So, long cut short, **SoftMax converts the value given out by each neuron (which can be positive or negative) into a probability value (between 0 to 1) which in turn is used to find the target class which corresponds to the max probability.**

- Mathematical Equation for a SoftMax Activation function:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

- Below is the code implementation of the SoftMax Activation function:

```
def softmax(y_pred):
    """Compute softmax values for each of the image in X .
    And convert each predicted value into a probability between 0 and 1, summing to 1
    """
    return tf.exp(y_pred) / tf.reduce_sum(tf.exp(y_pred), axis=0)
```

- Below is the post SoftMax activation value for the validation dataset:

```
*****y_pred_softmax*****
tf.Tensor(
[[0.02013392 0.04252557 0.08225834 ... 0.11804266 0.13563056 0.05583263]
 [0.03437398 0.00594652 0.59080684 ... 0.09606602 0.3412622 0.0625214 ]
 [0.01969389 0.5084085 0.01889278 ... 0.03517255 0.0255686 0.05313301]
 ...
 [0.14057674 0.00334839 0.00384398 ... 0.02702069 0.01401768 0.12347834]
 [0.16348557 0.03708254 0.00791791 ... 0.21894222 0.02236186 0.1198169 ]
 [0.30028787 0.01418647 0.00789507 ... 0.03792913 0.02489325 0.20662381]], shape=(10, 10000), dtype=float32)
```

Calculate Accuracy:

- As we can see from the output above, the **SoftMax layer outputs the probabilities** of an image belonging to a certain class but these are still numbers (float 32 – between 0 and 1) and cannot be used to compare with the true value which are hot encoded in terms of 1/0.
- So, in order to convert the SoftMax values from float to 1/0 and then to allow calculate accuracy we use the `tf.round` function which rounds the float values to the nearest integer.
- If we compare the SoftMax output above with the rounded output below we can see the values more 0.5 have been rounded to 1.0 and the values less than 0.5 have been rounded to 0.
- Now, in order to compare the predicted values (predictions) with the true values (y) we use a `tf.equal` function which compares the given values and return True when values are equal and False otherwise. So, this `tf.equal` function returns a Boolean array of dimension same as y true. Then, we cast this Boolean array in 1/0 with the help of a `tf.cast` function.
- Finally, we find accuracy but summing this NumPy array and dividing with total number of instances (10k in this case). We use `tf.reduce_mean` for this job.
- Below is the code Implementation:

```
def calculate_accuracy(x, y, w, b):
    '''Pass the given input through the network'''
    y_pred_softmax = forward_pass(x, w, b)

    """Round the predictions to the nearest integer to either 1. or 0."""
    predictions = tf.round(y_pred_softmax)

    """ tf.equal will return a boolean array: True if prediction correct, False otherwise
        tf.cast converts the resulting boolean array to a numerical array
        1 if True (correct prediction), 0 if False (incorrect prediction) """
    predictions_correct = tf.cast(tf.equal(predictions, y), tf.float32)

    """Finally, we just determine the mean value of predictions_correct"""
    accuracy = tf.reduce_mean(predictions_correct)

    # print(''*50+'predictions (rounded)'+''*50)
    # print(predictions)
    # print('Accuracy of this prediction is ==>',accuracy)

    return accuracy
```

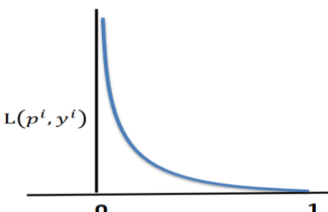
- Below is the output for the validation dataset when predicted values are rounded to nearest integer:

```
*****predictions (rounded)*****
tf.Tensor(
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]], shape=(10, 10000), dtype=float32)
: Test Loss :1.2929039001464844 Test Accuracy : 0.9107099771499634
*****
```

- Below is the Accuracy prediction for the validation dataset:

```
Accuracy of this prediction is ==> tf.Tensor(0.89999, shape=(), dtype=float32)
```

- [Cross Entropy Loss:](#)
- Now, why do we need to calculate loss!! **We can think of this value is like a guiding torch** for us, which shows us if we are going towards the goal or away from the goal. The more the loss, the more ill-defined the weights are biases are. **So, in more technical terms**, this loss is then used to find partial derivatives i.e. what will be the change in average loss for small change in weights or biases. And these partial derivatives are also called as gradients. And we use `GradientTape()` class to calculate and store these partial derivatives. These gradients are then passed on to the optimizer which then chooses the best weights and biases to be used to minimize loss. The optimizer used by us for this problem is **an Adam Optimizer**. (A detailed description is presented on the research section)
- Below is how a Cross Entropy Loss function per Image (L) looks like. Here we multiply the one hot encoded true value with the log of predicted values. Essentially, we will get an output for only one of the 10 (as we have 10 output neurons in this case) values ($-y \cdot \log(y_pred)$) as only one of the y true will be 1 and rest be 0. So, since we take a negative log, it will ensure that greater is the value of the prediction lesser is the loss. You can see below negative log graph for more insight.

$$L(p^i, y^i) = -\sum_{j=1}^c y_j^i \log(p_j^i)$$


- Below Equations shows the Cross-Entropy Loss Function per Image and the graph of a negative log.

- This is how average cross Entropy loss function looks like.

$$C = \frac{1}{m} \sum_{i=1}^m L(p^i, y^i)$$

- Below is the code implementation:

```
def cross_entropy(y, y_pred):
    '''Sometimes the softmax values (probability) outputted by a neuron can be very very close to 0 or 0
    In this case the log(0) ==> not defined and will result in NaN values.
    This can be handled using this function clip_by_value which replaces every value less than threshold
    with the minimum value (1e-10)'''

    y_pred = tf.clip_by_value(y_pred, 1e-10, 1.0)

    """Calculate the cross entropy loss per image"""
    loss_per_image = - tf.reduce_sum( y * tf.math.log(y_pred), axis=0 )

    """Calculate the average loss"""
    average_loss = tf.reduce_mean(loss_per_image)

    # print('***50+Cross Entropy Loss per Image'+***50)
    # print(loss_per_image)
    # print('***50+Average Cross Entropy Loss'+***50)
    # print(average_loss)

    return average_loss
```

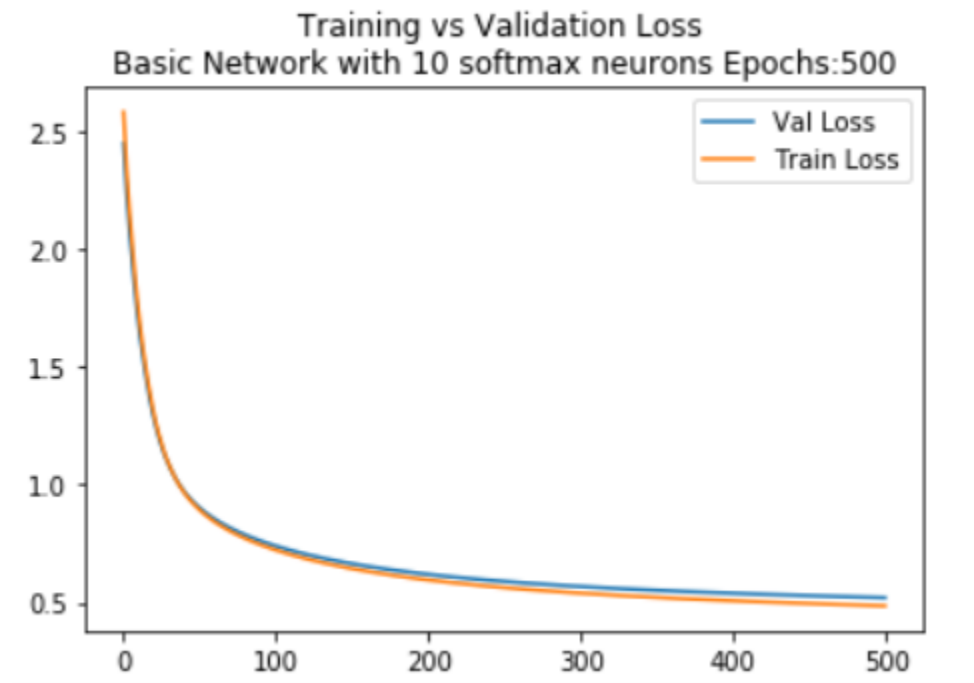
- Below are the Loss calculated per image and average loss.

```
*****Cross Entropy Loss per Image*****
****
tf.Tensor([3.0592182 1.5473728 1.6265501 ... 1.9634833 1.8330247 2.381402 ], shape=(60000,), dtype=float32)
*****Average Cross Entropy Loss*****
***
tf.Tensor(2.5096307, shape=(), dtype=float32)
```

- Below plot shows the performance of the neural network train and validation dataset across 500 epochs. As can be observed, both the losses substantially drop during the initial 50 epochs and then the descend becomes gradual. Both the losses start stagnating around 0.5 after 350 epochs with a very minute drop in loss then after.
- Post 350 epochs, although the loss continues to decline minutely however the train and the validation loss also continue to diverge. At this point the train loss is at 0.501 and the validation loss is 0.518.
- The performance after this point may be considered as an indication that the model is overfitting but the difference even at 500 epochs (as stated below) does not seem to be a clear indication. In my opinion the model does not clearly seem to overfit even at 500 epochs. However, as we see the loss starts to stagnate at 350, we can restrict the number of iterations at 350 and maintain the generalization of the model.**
- At 500 epochs:

Train Loss: 0.4860666

Test Loss: 0.5207719206809998

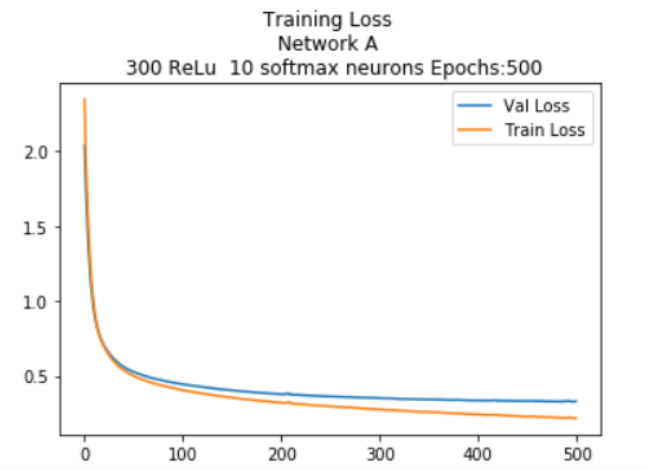


PART 121

- Below is the code for `forward_pass` function in the Network A:

```
42 """Forward pass for 1_2_1"""
43 def forward_pass(x, w2,w1, b):
44
45     x = tf.transpose(x)
46
47
48     """Layer 1 - ReLu Layer with 300 neurons"""
49     y_pred_layer1 = tf.matmul(w1, x) + b
50     y_pred_relu = tf.maximum(y_pred_layer1, 0)
51
52
53     """Layer 2 - Sigmoid Layer with 10 neurons"""
54     # We need to mutliply the output of the previous layer by the weights of this layer and add bias
55     y_pred_layer2 = tf.matmul(w2, y_pred_relu) + b
56
57
58     # Pipe the results through the sigmoid activation function.
59     y_pred_softmax = softmax(y_pred_layer2)
60
61
62     return y_pred_softmax
63
```

- Below is the performance graph for the Network A:



PART 122

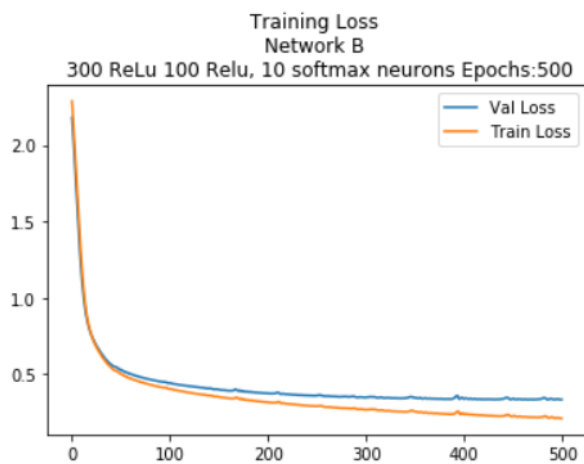
- Below is the code for `forward_pass` function in the Network B:

```

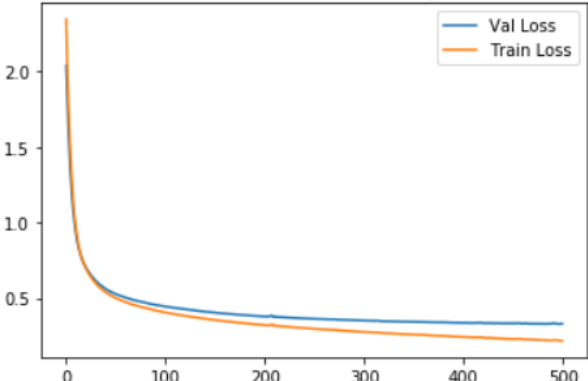
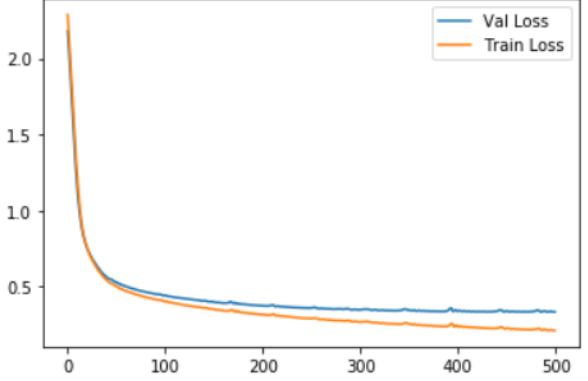
39 """Forward pass for 1_2_2"""
40 def forward_pass(x, w3_T, w2_T, w1_T, b):
41
42
43
44     """Layer 1 - With 300 ReLu Neurons"""
45     # We need to mutliply the flattened INPUT by the weights of this layer and add bias
46     y_pred_layer1 = tf.matmul(w1_T, x) + b
47     y_pred_relu_layer1 = tf.maximum(y_pred_layer1, 0)
48
49
50     """Layer 2 - With 100 ReLu Neurons"""
51     # We need to mutliply the output of the previous layer by the weights of this layer and add bias
52     y_pred_layer2 = tf.matmul(w2_T, y_pred_relu_layer1) + b
53     y_pred_relu_layer2 = tf.maximum(y_pred_layer2, 0)
54
55
56     """Layer 3 - Softmax Layer with 10 neurons"""
57     # We need to mutliply the output of the previous layer by the weights of this layer and add bias
58     y_pred_layer3 = tf.matmul(w3_T, y_pred_relu_layer2) + b
59
60     '''Pipe the results through the softmax activation function. '''
61     y_pred_softmax = softmax(y_pred_layer3)
62
63
64     return y_pred_softmax
65

```

- Below is the performance for the Network B:



- A comparison of performance of both Network A and Network B:

Network A	Network B
<p>Training Loss Network A 300 ReLu 10 softmax neurons Epochs:500</p>  <p>Train Loss : 0.21766794 Val Loss : 0.3302423059940338 Train Accu : 0.98500335 Val Accu : 0.9775400161743164</p>	<p>Training Loss Network B 300 ReLu 100 Relu, 10 softmax neurons Epochs:500</p>  <p>Train Loss : 0.20935807 Val Loss : 0.33287954330444336 Train Accu : 0.98555833 Val Accu : 0.9775300025939941</p>

My Comments:

- Both the network architectures exhibit similar performance. However, if we see the minute level, we will notice a small difference in train and Val loss of Network A and B respectively. Network B seems to be doing little better in terms of loss because of its ability to tune more weight and bias coefficient.
- The greater the number of hidden layers and more the number of neurons, the more your model is expected to perform.
- However, having said this, with more network complexity comes greater training cost in terms of memory and computation.
- Hence, if the difference in losses would be so miniscule I would try not to go for a deeper network.
- One more observation in both the charts, the model tends to perform well on the training data however does not perform equally well on the test or validation data. This means the model hasn't generalized well or is 'Overfitting'.
- I shall discuss this in detail in the next section.

PART 3

When does a Network Overfit?

- Overfitting occurs when your model seems to perform well on the training data while it does not perform so well on the unseen data (validation or test data). Basically, the model isn't generalizing well. **In other words, the model has learned specific data or image patterns in the training set which does not help predict the image class on the test data.**
- We can validate Overfitting by looking at the performance metrics, such as accuracy or loss. Usually the validation or test metric stops to improve after a particular epoch although the train metric during the same range of epochs seems to improve.
- **So, when we see the validation loss to stagnate or increase while the train loss is still reducing; it's a clear case of over fitting.** Your model is over trained to find the best fit on the train data and hence cannot generalize.

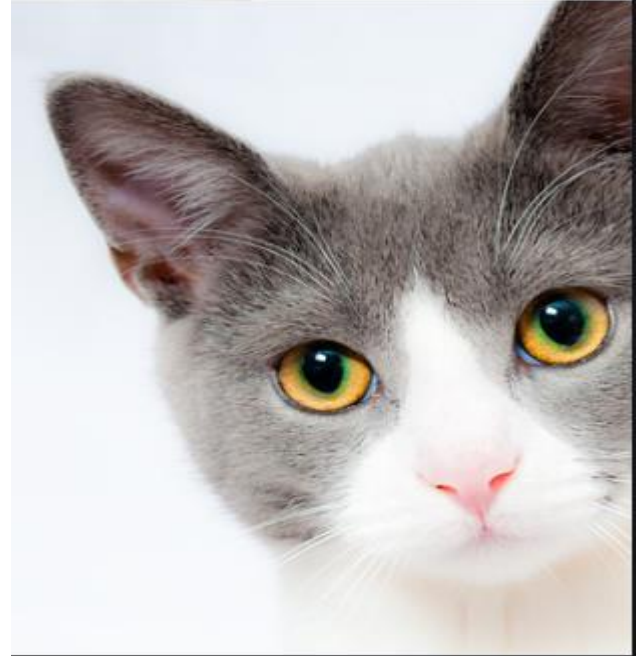
What are the ways to overcome over fitting?

- **Get more data:** This is one of the most obvious solutions. But unfortunately, not every company is Facebook, Google or Amazon to have access to massive volume of data. In applications such as Medical, Defense or Health there is a constraint to the volume of data you have access to. If that's the case, this is definitely not the solution for you.
- **Memory Reduction:** Try to reduce your model's memory of the training set so that the model does not get too much dependent on the training data. This process is also called as 'Normalization'.
- In Normalization we force the model to avoid being over dependent on certain inputs and hence the weights.

What's the Intuition behind 'Normalization'?

- Let's assume we are training a neural network for identify cats. And while training most of the images **contains a detailed image of a cats' face** (front picture). In this case, the model will start relying more on facial features of the cat and would think much about the rest of the features such as a tail which would be evident in a full image.
- So, now when you test your model, and present a facial picture of a cat as seems to be going well. **But, when you present a complete picture of a cat facing the other side**, although clearly recognizable for a human that it's a cat but the model will fail to recognize. Mainly because it has been over reliant on the facial features such as eyes or whiskers which were barely visible in the test image.

When a model gets over trained on certain features of an image class.

	
<p>Me: What's this? Model: Ohh! That's a cat.</p>	<p>Me: What's this? Model: I don't know!! I have never seen something like this!</p>

THIS IS OVERFITTING. 😊

Now in order to **GENERALIZE** we need to **NORMALIZE**.

- This is done by forcing the model to reduce the magnitude of weights by adding a normalization value to the loss function.

Ways of Normalization:

- **L1: Sum of absolute value of weights are added** to the average loss hence in order to reduce loss the model is forced to reduce magnitude of weights.
- Below is the equation for L1

$$C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m ((h(x^i) - y^i))^2 + \delta \sum_{j=1}^n \lambda_j$$

- Implementation of L1

```
def cross_entropy_wth_L1(y, y_pred, w3, w2, w1, regularization_rate):
    loss = - tf.reduce_sum( y * tf.math.log(y_pred), axis=0 )

    regularization_factor = regularization_rate / 2

    regularization = regularization_factor * ( tf.reduce_sum(tf.abs(w1)) + tf.reduce_sum(tf.abs(w2))
                                              + tf.reduce_sum(tf.abs(w3)) )

    out = tf.reduce_mean(loss) + regularization

    return out

def cross_entropy(y, y_pred, w3, w2, w1, regularization_rate):
    y_pred_ = tf.clip_by_value(y_pred, 1e-10, 1.0)

    return cross_entropy_wth_L1(y, y_pred, w3, w2, w1, regularization_rate)
```

L2: Similar to L1 but in this case instead of absolute values of weights, square of weights is used to penalize the model. This is stronger form (more aggressive) of normalization as the model is heavily penalized for bigger weights.

- Below is the equation for L2

$$C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m ((h(x^i) - y^i))^2 + \delta \sum_{j=1}^n \lambda_j^2$$

- Implementation for L2

```
def cross_entropy_wth_L2(y, y_pred, w3, w2, w1, regularization_rate):
    regularization_factor = regularization_rate / 2

    loss = - tf.reduce_sum( y * tf.math.log(y_pred), axis=0 )

    regularization = regularization_factor * ( tf.reduce_sum(tf.square(w1)) + tf.reduce_sum(tf.square(w2))
                                              + tf.reduce_sum(tf.square(w3)) )

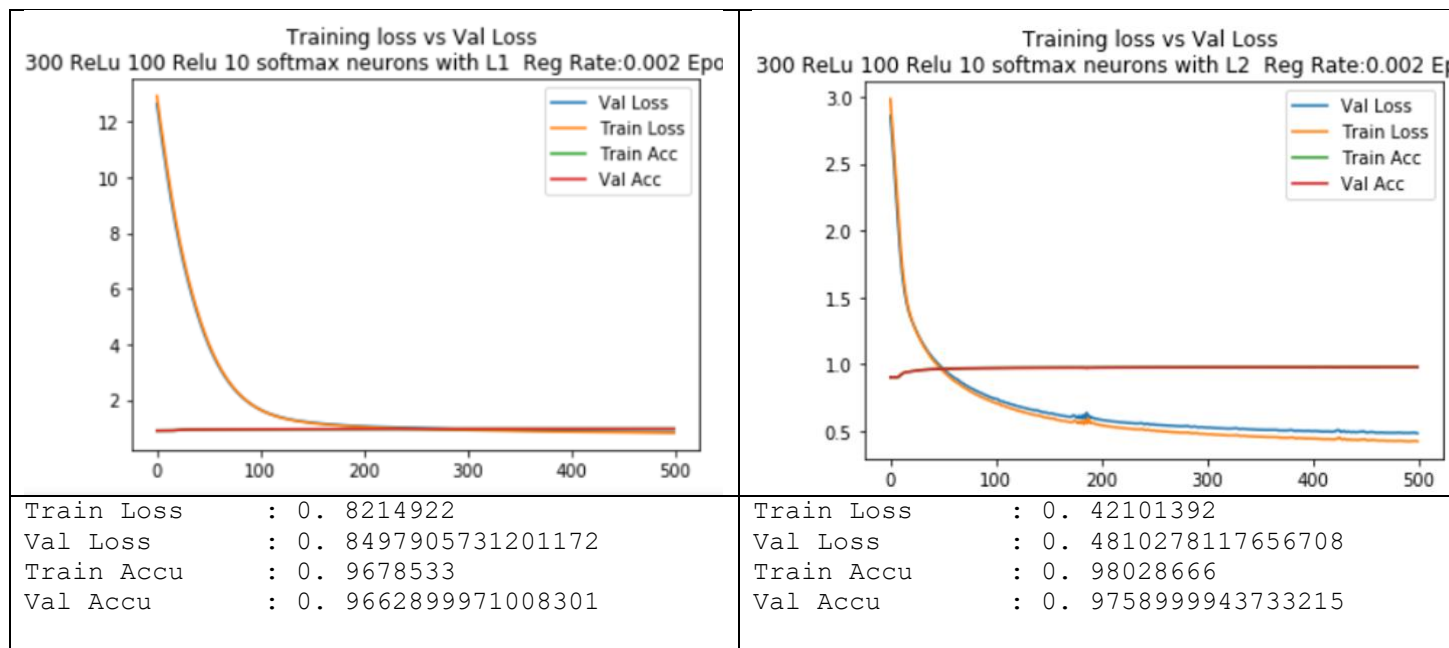
    out = tf.reduce_mean(loss) + regularization

    return out

def cross_entropy(y, y_pred, w3, w2, w1, regularization_rate):
    y_pred_ = tf.clip_by_value(y_pred, 1e-10, 1.0)

    return cross_entropy_wth_L2(y, y_pred, w3, w2, w1, regularization_rate)
```

- Below are the performance evaluation and comparison of model with both L1 and L2 Regularization.



My Comments:

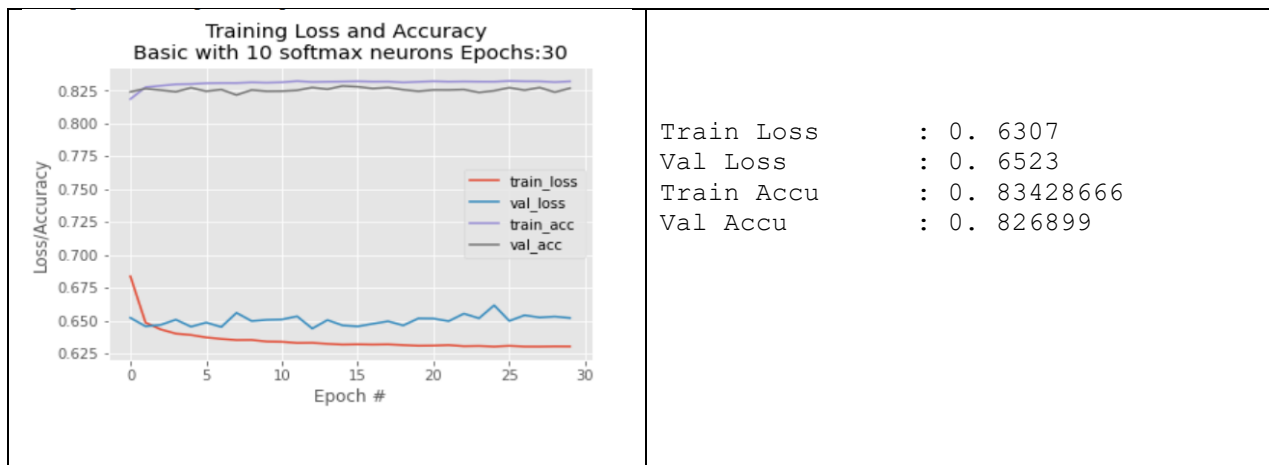
- As already explained above about **why is L2 more aggressive** in penalizing the model and bring down magnitude of weights as compared to L1, we shall try to reflect on the concept in the above findings.
- Both the networks have same architecture and have been passed with same 'regularization rate'.
- However, **if you see in L1, loss starts to descend from 12 and in L2 the loss starts to descend from 3.0**. This is primarily because L2 has been more aggressive in forcing the model to reduce the magnitude of weights which in turn helps model to avoid overfit and supports generalizing.
- If we look at the values presented below the graphs, in 500 epochs, **L2 brings down the train and Val loss to 0.48 and 0.42 respectively**. On the contrary, **L1 brings down the train and Val loss to 0.82 and 0.84 respectively**. **L2 helps the model to converge to optimum faster than L1** while also avoiding over fitting.
- There is also a caveat to this**, while L1 converges slowly it also is more careful while descending towards the optimum. If you look at the difference between the values test and validation loss of L1 and L2 below, L1 appears to be doing better.
- Which Regularization to use is fine balance between speed and accuracy**. L1 is slow by more convergent however L2 is fast but there is a little more difference between train and test loss as compared to L1.
- In practice, L2 is more often used than L1.**

PART B – Keras – High Level API

Building a basic SoftMax Classifier:

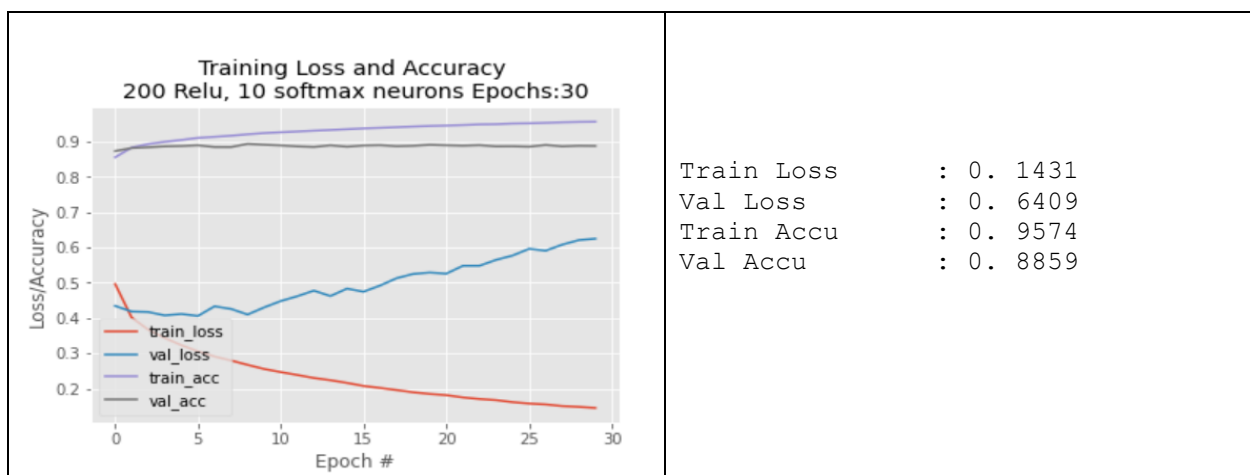
```
"""Configure the architecture here.."""

#Model with only Softmax Layer - Softmax classifier
model=tf.keras.models.Sequential([layers.Dense(10, activation=tf.nn.softmax, input_shape= (784,) ) ])
```



Building a 2 layer fully connected network:

```
# Model with Softmax and ReLu Layers #L1 200 Neurons L2 Softmax
model=tf.keras.models.Sequential([layers.Dense(200, activation=tf.nn.relu, input_shape= (784,) ) ,
                                  layers.Dense(10, activation=tf.nn.softmax, input_shape= (784,) ) ])
```



- Is there an improvement in accuracy over the SoftMax model?
- Of course, there has been an improvement in terms of accuracy and loss in both test and train data.
- However, despite the less accuracy and more loss, **the graph of the basic model seen to be a lot more promising than 2 layered model.**
- **If we see the graph of the 2 layered model, that is a clear case of overfitting** with both accuracy and loss of test and train data resp diverging. Over the 30 epochs, the train loss drops to as low as 0.1413 where as the test loss shoots to 0.6409. The model has been overfitting and fails to generalize and hence performs relatively poor on test data as compared to train matrix.
- Building Deeper Networks (Below are the possible option of layered architecture)

```
# Model with Softmax and ReLu Layers #L1 400 Neurons L2 200 Neurons L3 Softmax
model=tf.keras.models.Sequential([ layers.Dense(400, activation=tf.nn.relu, input_shape= (784,) ) ,
                                   layers.Dense(200, activation=tf.nn.relu ) ,
                                   layers.Dense(10, activation=tf.nn.softmax)
                                   ])

# Model with Softmax and ReLu Layers #L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 Softmax
model=tf.keras.models.Sequential([ layers.Dense(600, activation=tf.nn.relu, input_shape= (784,)),
                                   layers.Dense(400, activation=tf.nn.relu ) ,
                                   layers.Dense(200, activation=tf.nn.relu ) ,
                                   layers.Dense(10, activation=tf.nn.softmax)
                                   ])

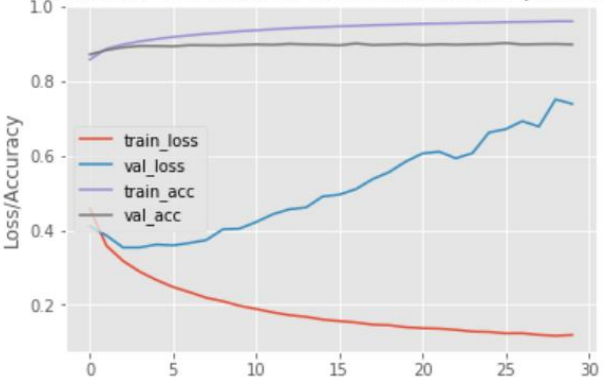
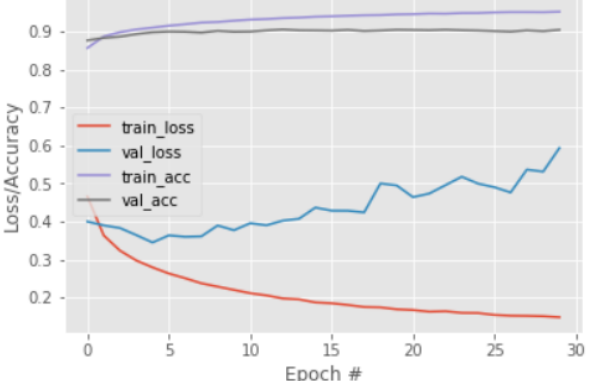
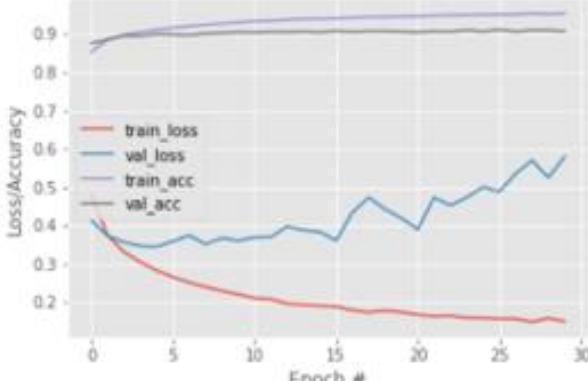
# Model with Softmax and ReLu Layers #L1 800 Neurons 600 Neurons L2 400 Neurons L3 200 Neurons L4 Softmax
model=tf.keras.models.Sequential([ layers.Dense(800, activation=tf.nn.relu, input_shape= (784,)),
                                   layers.Dense(600, activation=tf.nn.relu ) ,
                                   layers.Dense(400, activation=tf.nn.relu ) ,
                                   layers.Dense(200, activation=tf.nn.relu ) ,
                                   layers.Dense(10, activation=tf.nn.softmax)
                                   ])

```

Compare and contrast:

- Below graphs and metrics detail the performance of the deep architecture mentioned above.
- **Comparison in terms of loss:** if we look at the loss of all three models, it is evident that although 3 layered (3L) architecture could manage to reduce the loss to as low as 0.1189 but the it has given a poor show in terms of validation loss at 0.7409.
- Theoretically, more the number of neurons and hence more the number of connections less the models tend to overfit. We can see a clear example of it in the below cases. **Deeper models have managed to perform better in terms of loss and have been less prone to overfitting. With denser models, the validation loss drops at 0.7409, 0.5939, 0.5802 for 3L, 4L, 5L models respectively.**
- **Comparison in terms of Accuracy:**
- if we look at the accuracy of all three models, it is evident that although 3 layered (3L) architecture could manage to improve the train accuracy to as high as 0.9622 but the it has given a poor show in terms of validation accuracy at 0.8999.
- Theoretically, more the number of neurons and hence more the number of connections, less the models tend to overfit. Also, in this case, deeper models have managed to perform better in terms of loss and have been less prone to overfitting. **With denser models, the validation accuracy moves to 0.8999, 0.9049, 0.9071 at for 3L, 4L, 5L models respectively.**

- Also, it is important to note that there isn't much difference in terms of performance between 4L and 5L models. However, in such cases where performance is almost the same, **it is wise to choose a less dense network (4L in this case) as with denser network comes greater computational and operational cost.**

<p>Training Loss and Accuracy 400 ReLu, 200 Relu, 10 softmax neurons Epochs:30</p> 	<p>Train Loss : 0. 1189 Val Loss : 0. 7409 Train Accu : 0. 9622 Val Accu : 0. 8999</p>
<p>Training Loss and Accuracy 600 ReLu, 400 ReLu, 200 Relu, 10 softmax neurons Epochs:30</p> 	<p>Train Loss : 0. 1482 Val Loss : 0. 5939 Train Accu : 0. 9526 Val Accu : 0. 9049</p>
<p>Training Loss and Accuracy 800 ReLu, 600 ReLu, 400 ReLu, 200 Relu, 10 softmax neurons Epochs:30</p> 	<p>Train Loss : 0. 1503 Val Loss : 0. 5802 Train Accu : 0. 9535 Val Accu : 0. 9071</p>

PART B – ii

Regularization:

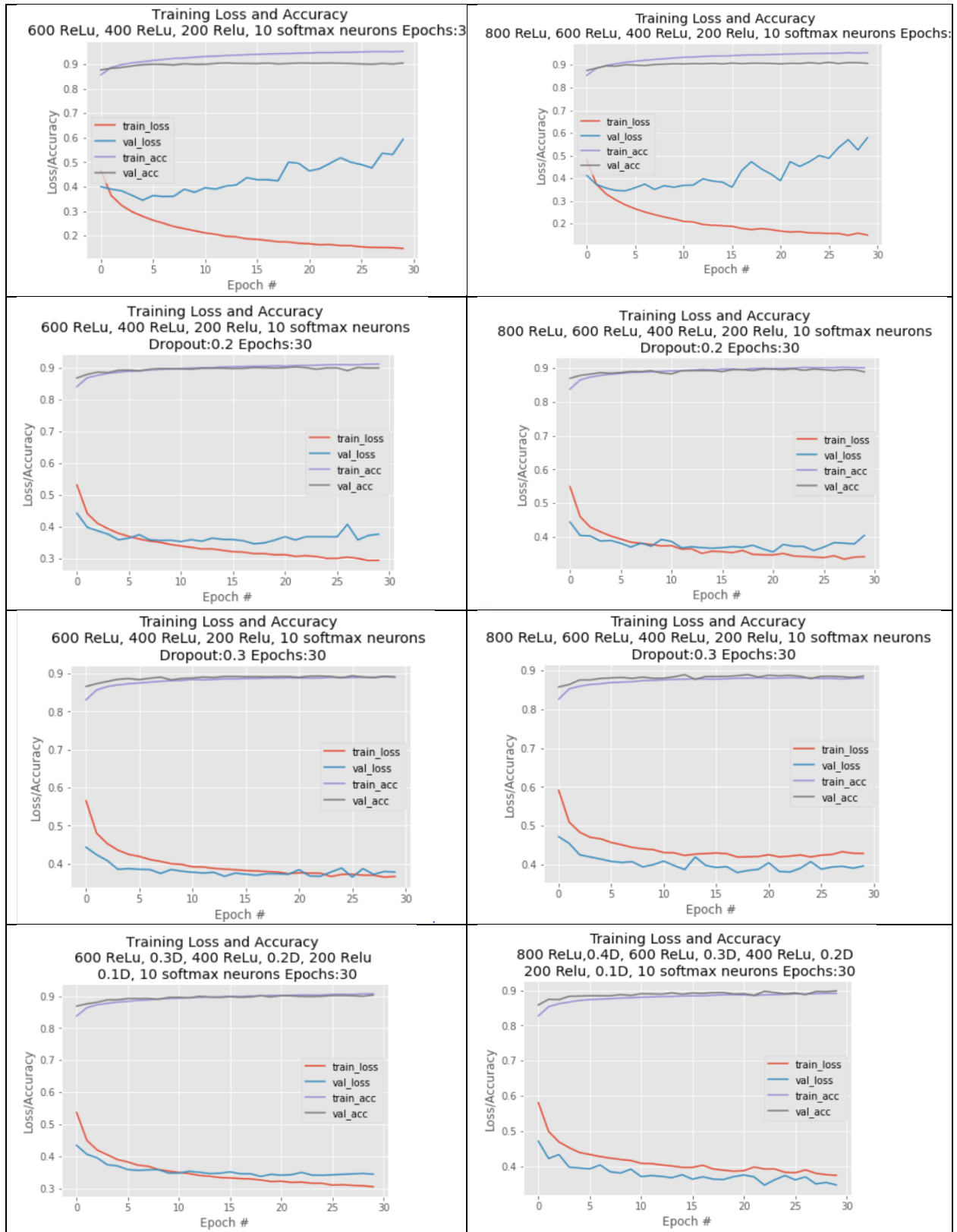
- We have discussed Regularization in great detail in the sections above. Drop out is another type of regularization.
- The idea is to **discourage the model to rely too much on the output of few neurons while ignoring the rest**. This is done by randomly dropping the connections between neurons with a given probability. **By dropping the neurons, I mean, the values of weights connecting those neurons in the weight matrix is set to zero, thus by passing those neurons.**
- In the models below, I have used a drop out probability of 0.2 and 0.3 across the architecture and in the **last experiment I have tried to drop more neurons in layers with more connections and less in the less dense layers.**
- For instance, the drop out in the first layer with weight dimension at (800*600) is kept at 0.4, (600*400) at 0.30, (400*200) at 0.2, (200*10) at 0.1. This ensures the regularization is more rigorously done in the layers with more connections.

Below are my comments:

- **4D (without drop out)** - clear case of overfitting as discussed in detail in the section above.
- **4D (Drop out = 0.2)** - Improvement over the base model but model still tends to overfit which can be seen by the divergence in the loss graph below.
- **4D (Drop out = 0.3)** - Further improvement over the previous model. Although the validation loss dropped just a little from 0.3848 to 0.3781 but the difference between the losses (test and train) which can also be called as divergence has reduced significantly from approx. 9 points to 2 points. Drop out has improved the problem of overfitting.
- **4D (Drop out = 0.3/0.2/0.1)** - In this setting we have tried higher drop out ratio in denser layers and lower in less dense ones. As already discussed above this technique was expected to work better than using common drop out ratio. After this testing the performance was as expected. With this setting, **the validation loss further reduced from 0.3781 to 0.3443 and the validation accuracy further improved from 0.8911 to 0.9034.**
- Similar effect of dropout was observed in the 5L network as well. But, the 4L model seemed to perform similar to 5L both in terms of accuracy and loss. **In such cases it is rather a better choice to perform normalization in a less dense network and improve the performance as denser networks also mean more computational and memory cost.**

Below are the accuracies and loss obtained in different settings:

4D	loss: 0.1482 - accuracy: 0.9526 - val_loss: 0.5939 - val_accuracy: 0.9049
4D WITH dROP 0.2-	loss: 0.2937 - accuracy: 0.9102 - val_loss: 0.3848 - val_accuracy: 0.9003
4D WITH DROP 0.3 -	loss: 0.3661 - accuracy: 0.8904 - val_loss: 0.3781 - val_accuracy: 0.8911
4D WITH DROP 0.3/0.2/0.1 -	loss: 0.3054 - accuracy: 0.9069 - val_loss: 0.3443 - val_accuracy: 0.9034
5D	loss: 0.1503 - accuracy: 0.9535 - val_loss: 0.5802 - val_accuracy: 0.9071
5D WITH dROP 0.2 -	loss: 0.3411 - accuracy: 0.9008 - val_loss: 0.4044 - val_accuracy: 0.8888
5D WITH DROP 0.3-	loss: 0.4286 - accuracy: 0.8799 - val_loss: 0.3963 - val_accuracy: 0.8853
5D WITH DROP 0.4/0.3/0.2/0.1 -	loss: 0.3744 - accuracy: 0.8913 - val_loss: 0.3469 - val_accuracy: 0.8982



Research

Adam Optimizer

Before we understand Adam Optimizer, there are some prerequisites that we must understand to know what led to the dawn of this optimizer. But for now, just understand this, Adam (Adaptive Moment estimation) is a modern optimizer which mostly finds its use in deep neural networks. Adam was first published in 2014 and then presented in [ICLR 2015](#). **Adam is largely a combination of two of its predecessor optimizers: SGD momentum and RMS prop.**

What's an optimizer?

- Optimizer is an algorithm or a set of mathematical equations that are used to optimize gradient descent. Or in other words, find the values of the weight and bias matrices for the next epoch. This in turn supports learning.
- Neural Networks or machine learning models work on the basic idea of first initializing the weights randomly (with a particular distribution), pass the data through the model, find the loss and try to minimize the loss. And this loop continues for ever.
- Now, the fun and the mostly challenging part here is called as learning which happens when you update the weights and biases given the loss and pass the data through the network again. The algorithm who does this difficult but critical part is called as 'Optimizer'.

How many optimizers has the world seen so far?

- Well, there are tons of Optimizer papers been published over the years and many even go unnoticed. But there is one common thing among each of them – 'They all claim to be the best and solve the problem of learning'.
- However, although many of them present interesting and promising results but very few of them (algorithms) succeed to generalize. In other - a very few of them tend to work well on most machine learning or deep learning problems (data sets).
- Of the very few who managed to get the limelight and attention of the world and came up with real break throughs were Stochastic Gradient Descent (SGD) with momentum, RMS prop, Adam, AdaGrad, AdaMax and few more.

What is Adaptive Learning Rate?

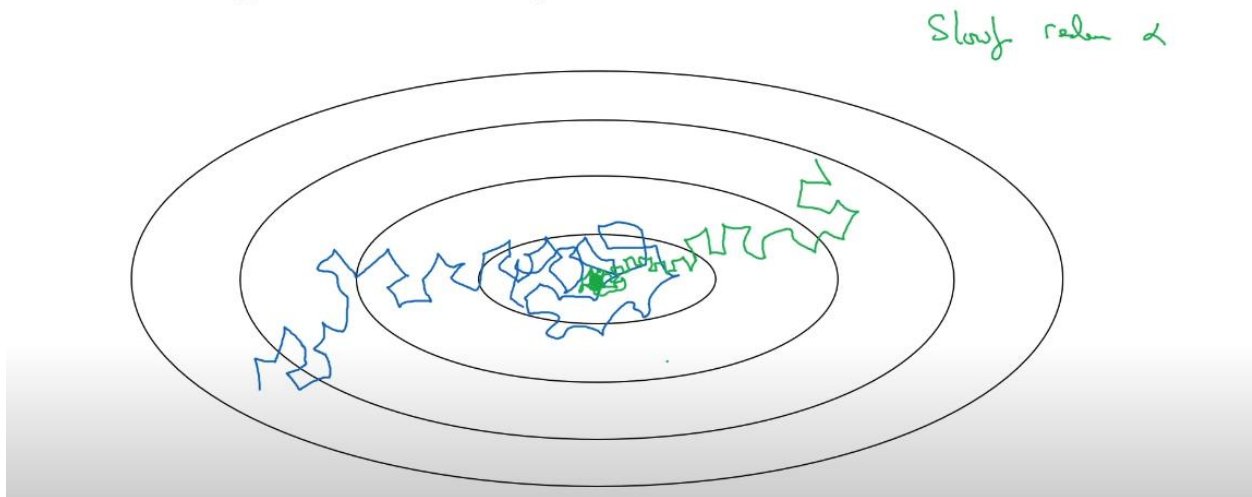
Challenges:

1. A low learning rate leads to a painfully slow convergence to the optima, which is not acceptable in most of the cases due to the high volume of data.
2. A high learning rate solves the above problem but in turn comes up with a new problem of never reaching to the global optima. In such cases the gradient descent just hovers around the optima but never reaches one.

Solution:

1. We need an algorithm that starts fast and ends slow. In other words, an algorithm that initially starts with a high learning rate thus allowing faster convergence and then methodically slowing down as it reached the optima and thus allowing convergence. This technique is called as Adaptive Machine learning (ADL).
2. Adam Optimizer is something that incorporates ADL in a way by finely balancing between Velocity and Momentum of the Gradient Decent (GD).

Learning rate decay



Source: Lecture on Learning Rate Decay by Andrew Ng.

What is Sparse Gradient?

This phenomenon occurs when there is a very less information to extract from the gradient. In other words, the gradient is too small to infer anything. If you imagine a ball rolling down a hill and descending to the steepest slope (gradient), **then sparse gradient is when the ball hits a plain ground (with almost zero slope) and does not know what to do or go next.** This is something that happens very often in deep learning problems.

What's the problem with SGD momentum?

SGD with Momentum is a very popular optimization technique which focuses on reducing vertical oscillations and tune more towards moving to the horizontal direction (This is just an intuitive understanding; in real time these parameters are in high dimensional space and difficult to imagine). This helps the algorithm prevent over shooting. **Although, SGD with momentum is a popular but it has been criticized for its inability to navigate through sparse gradients (discussed above).**

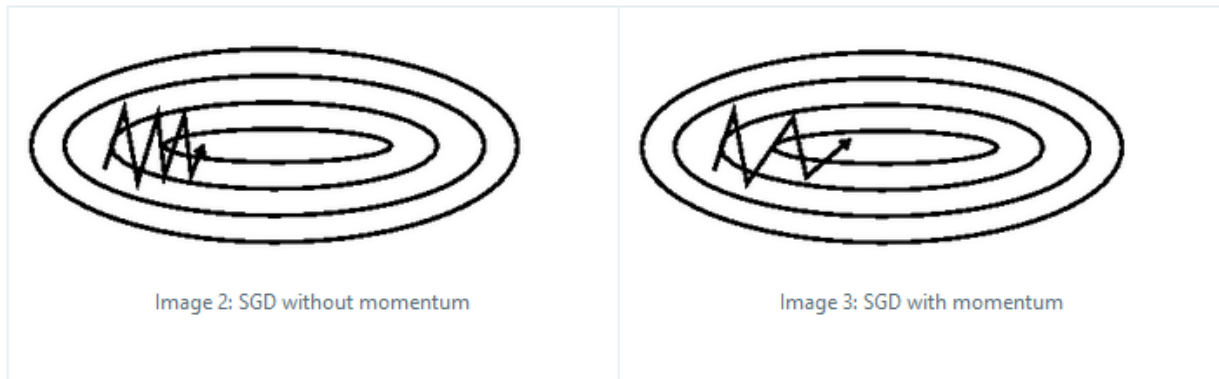


Image Source: medium.com

Let's see what happens technically:

Below is the equation from Adam Optimizer which was inspired from SGD with Momentum.

In the equation below, when the algorithm will hit the sparse matrix gradient (g_t) will be very less. At $\beta = 0.9$, the **second part of the equation will be even lesser ($0.1 * g_t$)**, and hence m_t will remain almost the same and hence the algorithm won't update the coefficients much.

$$m_t = \beta m_{t-1} + \eta g_t$$

Notation:

β = update factor

m = exponential moving average of the partial gradient at batch t

η = Learning Rate

conclusion: SGD -M doesn't work well in sparse gradient.

What is a Moment?

Moment is nothing but the expected value of a variable to its n^{th} power. So, the second moment of X is expected value at X^2 .

$$m_n = E[X^n]$$

m — moment, X -random variable.

Adaptive Moment Estimation (Adam)

Adam is another method that helps to calculate the adaptive learning rates for each parameter (weights and biases). **In addition to storing the exponentially decaying squared moving averages of the gradients** (V_t) as done in RMS prop; the algorithm also stores the exponentially decaying moving averages of the gradients (first moment - m_t) as is done in SGD with momentum.

- Below are the notations used in the equation:

m^t =first moment, calculated taking exp moving average of the gradient.

v^t = second moment, calculated taking exp moving average of the **squared** gradient.

β_1 = hyper learning parameter for moving averages

β_2 = hyper learning parameter for squared moving averages

g = gradient evaluated on the current mini batch (t)

- Equations used to calculate the moving averages:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Moving averages of gradient and squared gradient.

- Since m and v are the estimates of the first and second moment. Below condition should be fulfilled and only then we can say that the algorithm is an unbiased estimator.

$$E[m_t] = E[g_t]$$

$$E[v_t] = E[g_t^2]$$

- But during the initial run, because the gradients are more filled with zero and hence the algorithm also seems to be biased towards zero.

This is what happens:

$$\begin{aligned}
 m_0 &= 0 \\
 m_1 &= \beta_1 m_0 + (1 - \beta_1) g_1 = (1 - \beta_1) g_1 \\
 m_2 &= \beta_1 m_1 + (1 - \beta_1) g_2 = \beta_1 (1 - \beta_1) g_1 + (1 - \beta_1) g_2 \\
 m_3 &= \beta_1 m_2 + (1 - \beta_1) g_3 = \beta_1^2 (1 - \beta_1) g_1 + \beta_1 (1 - \beta_1) g_2 + (1 - \beta_1) g_3
 \end{aligned}$$

- The above can be written in terms of equation:

$$m_t = (1 - \beta_1) \sum_{i=0}^t \beta_1^{t-i} g_i$$

- In the above equation β_1 is a tunable constant (rarely changed from the default value of 0.9). If we see, the equation carefully, most of the contribution will be done by the second part of the equation which has powers of β_1 which moves towards 0 with greater powers of β_1 . This in turn reduce the value of m_t towards 0. **This is what makes the estimator biased towards zero in the initial phases from initialization.**
- **In order to avoid this bias or skew, we need to correct the values** given by this estimator as below. We divide the equation by $(1 - \beta^t)$ where t is the mini batch number.

$$\begin{aligned}
 \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
 \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}
 \end{aligned}$$

Bias corrected estimators for the first and second moments.

- This makes complete sense, as in the initial phase let's say at, $t = 1$, $\beta_1 = 0.9$

Here m_t will be divided with 0.1 which shall increase the value of m_t and will as a result move the algorithm away from 0. As the algorithm continues to train, the dividing factor will increase gradually and move towards 1 leaving the m_t unchanged.

- The only thing after this that is left to do is to put the above numbers in the update rule below.

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where,

η = step size or learning rate

ϵ = a very small number usually defaulted to 10^{-9} , just in case the denominator ($\sqrt{\hat{v}_t}$) drops to 0 and the division goes 'undefined'.

The best part:

The authors propose the value β_1 β_2 and ϵ be used with the default values of 0.9, 0.99, 10^{-8} and hence there isn't much to tune. Happy times!!

Problems Solved:

1. **Made adaptive learning very intuitive:** While adaptive learning was still a problem to be completely solved, Adam comes up very well on this.
2. **Unlike SGD-M, the moving averages and hence the update rule is not a function of the magnitude of the gradient and therefore helps in navigating through sparse gradients.**
3. **Adam also retains the goodness of RMS prop while solving the problem in SGD-M and hence can be used in diverse applications.**

