**DEEP LEARNING ASSIGNMENT:**

**TensorFlow/Keras Assignment**

**Part A: TensorFlow and the Low Level API**

The Fashion-MNIST is an image dataset consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image (784 pixel values in total), associated with a label from 10 different classes.

The following are the set of classes in this classification problem (the associated integer class label is listed in brackets).

* T-shirt/top (0)
* Trouser (1)
* Pullover (2)
* Dress (3)
* Coat (4)
* Sandal (5)
* Shirt (6)
* Sneaker (7)
* Bag (8)
* Ankle boot (9)

1. **Section I: TensorFlow binary classifier**

In this part, the task is to build a binary two-layer classifier using the TensorFlow low level API (in graph mode).

The architecture of the model is as following:

a. Layer 1: 100 neurons (ReLu activation function)

b. Layer 2: 1 neuron (Sigmoid activation function)

c. Learning Rate: 0.01 with Gradient Descent

1. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL\_Assignment1\_Part\_A\_I.ipynb .

2 functions have been defined:

\_ loadData : accepts a list of 2 values as input for specifying the 2 selected classes used for the binary classification. The function returns 4 tensorflow objects which corresponds to the training images, the training labels, the test images, and the test labels.

\_ neuron\_layer: defines the neurons process computation with at the end the activation function (optional).

Beside the 2 functions, the architecture of the model is implemented in different sequential scopes. The defined scopes are:

\_ neuronal\_net: defines the structure of the neuronal network (using the neuron\_layer function)

\_ loss: defines the cross entropy used to compute the loss

\_ train: defines the optimizer used to train the model

\_ accuracy: establishes the prediction calculation and checks its correctness

Finally, after the construction of the model, a session is created in order to train the model.

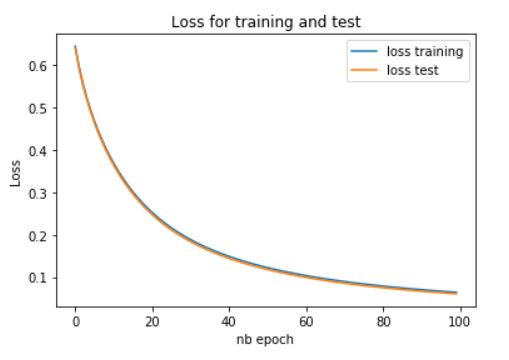
For more information on the code implementation, please, refers to the comments present in the ipynb file.

1. Result Analysis:

The binary classification has been performed for the classes Coat (4) and Sneaker (7) and the training session has been iterated for 100 times (number of epochs = 100).

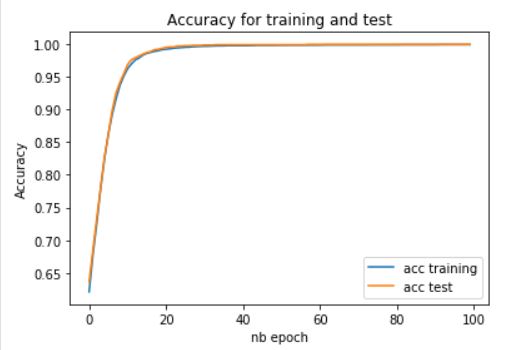
After 100 iterations, we reach an accuracy of 0.999 for the test.

Here are the results of the model:



The loss falls very quickly in the first 40 iterations. Then, it goes into a plateau.

We can notice as well that losses for the training and the test are aligned which leads to the conclusion that our model is not overfitted. Moreover, the value of this loss is extremely low after 100 iterations. Therefore, our model has a very high rate of correct predictions.



Here, the accuracy takes off very quickly in the first 20 iterations. Then, it goes to a plateau.

Again, the accuracy for the training and the test are aligned which leads to the conclusion that our model is not overfitted and as previously mentioned, the accuracy is extremely high.

1. **Section II: TensorFlow multi-class classifier**

In this part, the task is to build a full multi-class classifier (all 10 classes for Fashion MNIST). Using TensorFlow’s low level API (in graph mode).

The architecture of the model is as following:

a. Layer 1: 300 neurons (ReLu activation functions).

b. Layer 2: 100 neurons (ReLu activation function)

c. Layer 3: Softmax Layer

d. Learning rate: 0.01 for this problem with Gradient Descent

1. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL\_Assignment1\_Part\_A\_II.ipynb .

All the functions, scopes and session are mostly identical to Part A:section I. The noticeable differences are:

\_ In the loadData function, no classes section is available since the classification should be done for all the classes and those classes are one hot encoded.

\_ In the neuron\_layer function, a softmax activation function has been added

\_ In the “neuronal\_net” scope from the model architecture, the numbers of layers have changed (3 instead of 2). So is the number of neurons.

For more information on the code implementation, please, refers to the comments present in the ipynb file.

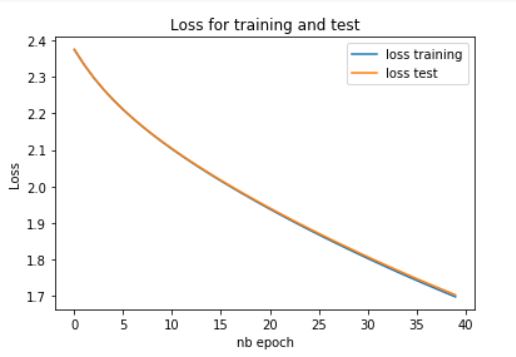
1. Result Analysis:

For this part, the training session has been iterated for 40 times (number of epochs = 40).

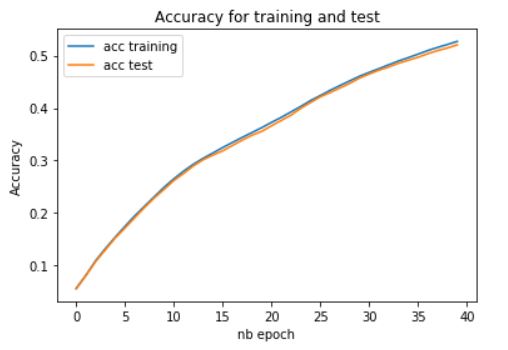
After 40 iterations, we reach an accuracy of 0.5207 for the test.

The model training runtime is 55.8 sec.

Here are the graphs showing the loss and the accuracy of the test and training accuracy after each epoch:



We can notice that the curve for the losses for the training and the test are very much aligned indicating that that our model is not overfitted. But in comparison to the Part A : Section I, we can see that this time the losses value is pretty high. However, the curves have not reached to a plateau (and the training and test curve are not very different). This means that our model needs to be trained a little bit more to get a better result of the number of losses.



Here, the accuracy takes off gradually during the 40 iterations.

The accuracy curves for the training and the test are almost similar. But we can notice that at the end, the values of the training accuracy are slightly higher than the ones of the test. This means that our model is probably about to be overfitted even though the values are still acceptable.

Finally, we can see that the value of the accuracy for both training and test are quite low (nearly 50%). The number of epochs could therefore be increased in order to get a better result of accuracy and loss. But we have to keep in mind that the risk is that we can have a model which is overfitted.

1. **Section III: TensorFlow multi-class classifier with mini batches**

In this part, the task is to take the same architecture model in the Part A: Section II; but using this time, the mini-batch gradient descent.

Indeed, the model training runtime was a bit long for the previous part (around 56 seconds). The usage of the mini-batch gradient will definitely improve the run time.

1. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL\_Assignment1\_Part\_A\_III.ipynb .

All the functions, scopes and session are mostly identical to Part A:section II. The noticeable differences are:

\_ A new function has been implemented: get\_mini\_batches. This function is called during the run session in order to split the training data set accordingly to the size of the batch.

\_ In the training run session “sess” the model is trained using the mini-batch methodology. Therefore, a new iteration loop is added inside the epoch iteration in order to go through each mini-batch and train the model in each of them.

For more information on the code implementation, please, refers to the comments present in the ipynb file.

1. Result Analysis:

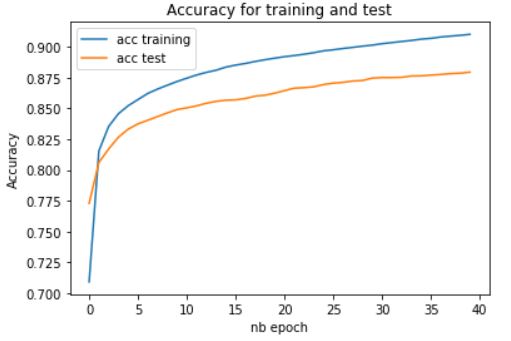
For this part, the training session has been iterated for 40 times (number of epochs = 40) and different mini-batch size have been experimented:

* batch\_size = 100
* batch\_size = 500
* batch\_size = 1000
  1. *mini-batch gradient (batch\_size=100)*

After 40 iterations, we reach an accuracy of 0.88 for the test.

The model training runtime is 92.2 sec.

Here are the graphs showing the accuracy of the test and training accuracy after each epoch:



The first observation is that with a batch size of 100, we reach quickly to a good accuracy after 1 or 2 epochs (around 0.825). This contrasts to the previous section in which after 40 epochs, we are not able to have a proper accuracy (around 0.5). This shows the obvious interest of the mini-batch methodology: we can reach quickly to a better performance with this methodology.

The second point is that we can notice a lot of “noises” in the test accuracy (the test curve is not smooth) which is due to the high variation of the errors when the batch size is small.

Without any surprise, after 40 epoch the training model is extremely overfitted: The value between the training accuracy and the test accuracy is very especially after 2 epochs.

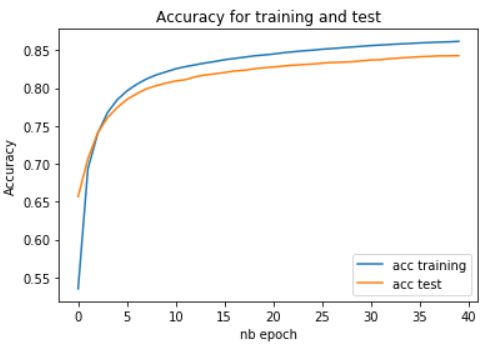
The mini-batch gradient is very powerful methodology which should be used with few numbers of epochs.

* 1. *mini-batch gradient (batch\_size=500)*

After 40 iterations, we reach an accuracy of 0.84 for the test.

The model training runtime is 36.6 sec.

Here are the graphs showing the accuracy of the test and training accuracy after each epoch:



Just like the previous observation, with a batch size of 500, we reach quickly to a good accuracy after 3 epochs (around 0.76).

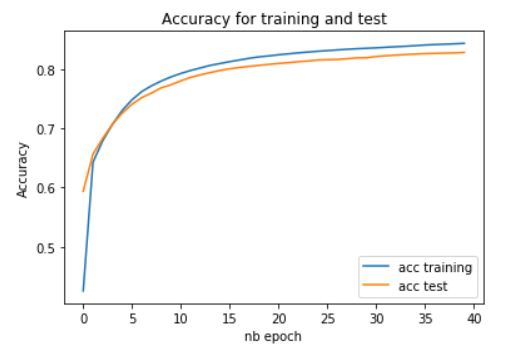
In comparison to the training model with the batch size of 100, we need more epochs to reach to a good accuracy and the test accuracy curve is less noisy. The model needs more time to converge to a good accuracy

* 1. *mini-batch gradient (batch\_size=1000)*

After 40 iterations, we reach an accuracy of 0.83 for the test.

The model training runtime is 27.8 sec.

Here are the graphs showing the accuracy of the test and training accuracy after each epoch:



Just like the previous observations, with a batch size of 1000, we reach quickly to a good accuracy after 4 epochs (around 0.72).

Just like before, a bigger batch size implies that we need more epochs to converge to a good accuracy, but at the same time, the accuracy is more stable and less noisy.

**Part B: Keras -- High Level API**

The objective of Part B is to explore the problem of image classification is using a dataset called the notMNIST dataset. The dataset contains images of letters from A – J inclusive.

In this dataset, some pre-processing work has been completed and it has also been normalized.

1. **Section I: Keras Softmax classifier**

In this part, the task is to use Keras to build a SoftMax classifier.

1. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL\_Assignment1\_Part\_B\_I.ipynb .

1 function has been defined:

\_ loadData : The function returns 4 numpy arrays which corresponds to the training images, the training labels, the test images, and the test labels.

Beside this function, the model is implemented using Keras function:

\_ keras.models.Sequential: list of keras neurons layers defining the number of neurons, their connections, their activation function and eventually their input data. In this section 2 layers has been defined:

* L1 with 512 neurons fully-connected and relu activation function
* L2 with 10 neurons fully-connected and Softmax activation function

\_ model.compile: defines the optimizer, the cross entropy and the type of metrics used to compute the loss as followed:

* Optimizer: Adam Optimization algorithm
* Loss: the sparse categorical cross entropy
* Metrics: accuracy

\_ model.fit: defines the training model runs configuration. In this section the configuration has been done with the following parameters:

* Number of epoch =3
* Batch size = 256
* Validation split : 10 % of the training data

\_ model.evaluate: Evaluates the trained model in term of loss and accuracy

For more information on the code implementation, please, refers to the comments present in the ipynb file.

1. **Section II: Keras with various layers of neurons**

In this part, the task is to study the performance of various model of softmax classifiers using different numbers of neurons and layers.

1. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL\_Assignment1\_Part\_B\_II.ipynb .

All the function and Keras functions are almost identical to Part B: section I. The noticeable differences are:

\_ In the keras.models.Sequential, various models are set with different layers and numbers of neurons .

For more information on the code implementation, please, refers to the comments present in the ipynb file.

1. Result Analysis:

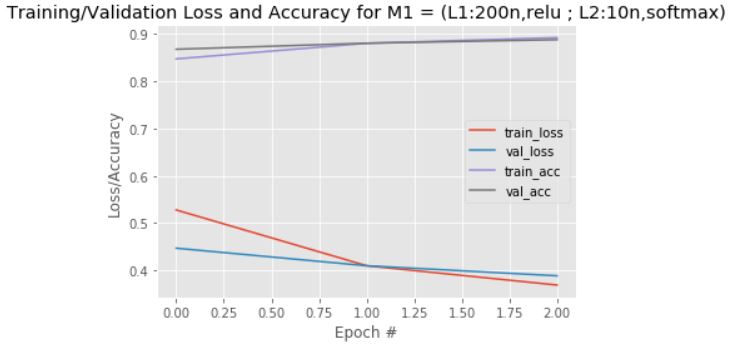
Three models of neuronal networks have been studied.

* 1. *Model 1: (L1:200n,relu ; L2:10n,softmax;)*

In this model the architecture which have been implemented is the following:

* L1 200 Neurons L2 Softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:



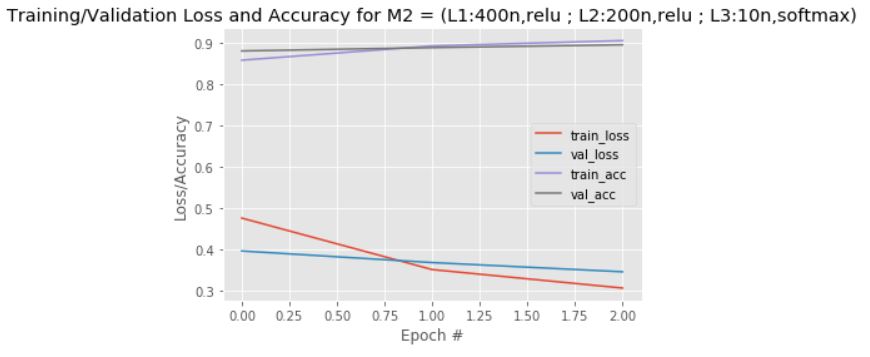
We can see that after 1 epoch, the model starts to overfit. The accuracy which can be obtained for 1 epoch is around 0.88.

* 1. *Model 2 : (L1:400n,relu ; L2:200n,relu ; L3:10n,softmax)*

In this model the architecture which have been implemented is the following:

* L1 400 Neurons L2 200 Neurons L3 Softmax

Here is the graph showing the loss and accuracy of the training and the validation data after each epoch:



Again after 1 epoch, the model starts to overfit. The accuracy which can be obtained for 1 epoch is around 0.9.

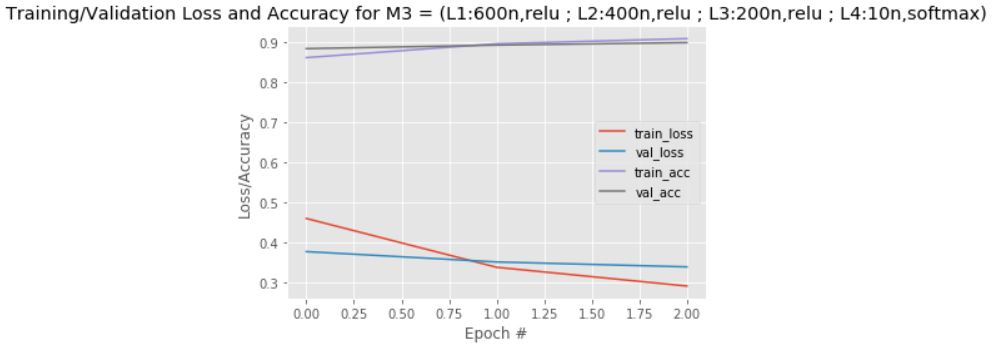
The performance seems to be slightly better in comparison to the 1st model with 2 layers

* 1. *Model 3 : (L1:600n,relu ; L2:400n,relu ; L3:200n,relu ; L4:10n,softmax)*

In this model the architecture which have been implemented is the following:

* L1 600 Neurons L2 400 Neurons L3 200 Neurons L4 Softmax

Here is the graph showing the loss and accuracy of the training and the validation data after each epoch:

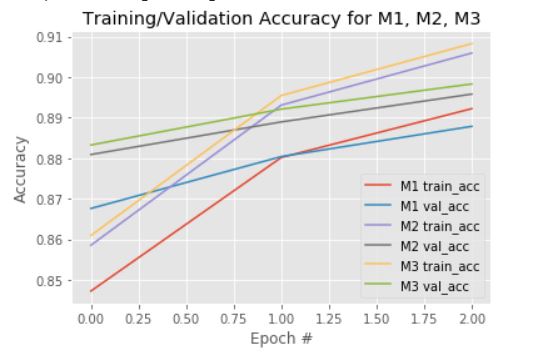


We can see the same behavior: the model starts to overfit after 1 epoch. The accuracy which can be obtained for 1 epoch is around 0.9

The performance seems to be a bit better if the number of layers and neurons increases.

Without any surprise, for the 3 models, the overfitting starts after 1 epoch. Indeed, the batch size is identical for all of them; as a result, the overfitting behavior appears at the same time for all the model.

The following graphics shows a comparison of the accuracy (training and validation) for the 3 models:



As we have correctly guessed earlier, the performance is higher if we increase the number of neurons and layers in a model.

1. **Section III: Keras using Drop out and regularization techniques**

In the previous part, we have noticed that we can get overfitting issues when the number of epochs is high. Fortunately, various techniques could be used to avoid the problem of overfitting: The Drop out and the regularization techniques.

1. Quick code explanation:

All the function and Keras functions are almost identical to Part B: section II. The noticeable differences are:

\_ In the keras.models.Sequential, the models M2 and M3 are used and for both models we introduce the drop out or the regularization techniques .

For more information on the code implementation, please, refers to the comments present in the ipynb file.

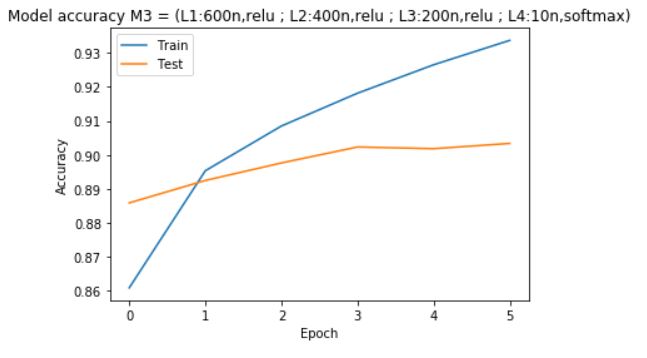
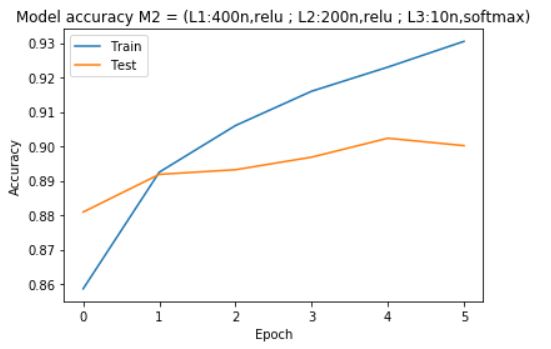
1. Result Analysis:

Various models of M2 and M3 neuronal networks have been studied inserting the drop out or the regularization.

1. *Regular Model*

The analysis has been conducted to M2 and M3 architectures from Part B: Section II.

The accuracy results for the 2 models are presented below:

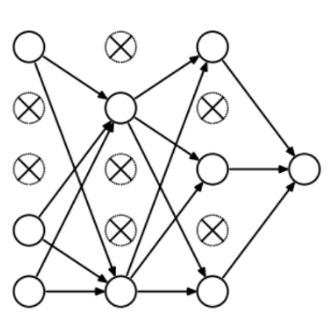


For both models, the overfitting is extremely high after on epoch and the accuracy at 1 epoch is less around 0.89.

The drop out and regularization techniques can be used here to correct this issue.

1. *Drop Out technique*

The idea of the Drop Out technique is to basically remove, depending on a fixed percentage value, some nodes in one particular neurons layer as the following :



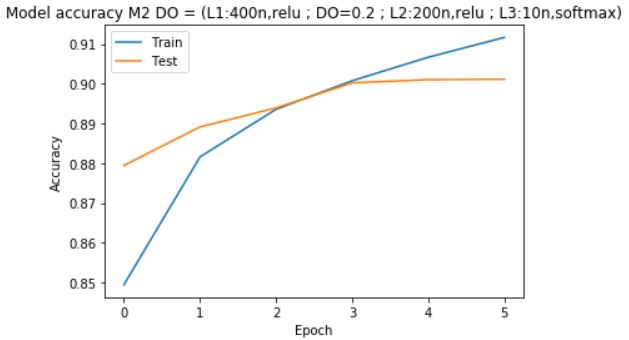
# *Image from An Overview of Regularization Techniques in Deep Learning (with Python code), Shubham Jain,* [*link*](https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/)

The method has been applied to M2and M3 model architecture.

For M2 DO the architecture is:

* L1:400n+relu ; DO=0.2 ; L2:200n+relu ; L3:10n+softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:

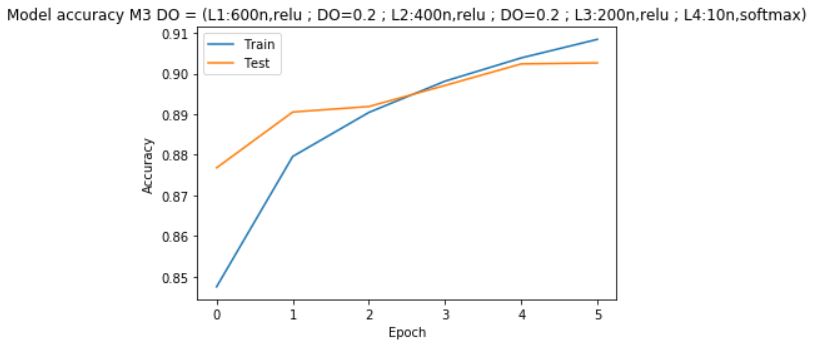


We can see that the overfitting starts only after 3 epochs and the accuracy reaches to 0.9 at 3 epochs. This results is significantly better to the M2 model which was overfitting after 1 epoch with an accuracy of 0.89.

For M3 DO the architecture is:

* L1:600n+relu ; DO=0.2 ; L2:400n+relu ; DO=0.2 ; L3:200n+relu ; L4:10n+softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:



The result is quite similar to M2 DO: We have an accuracy of 0.9 after 3 or 4 epochs and the model starts to overfits after 4 epochs. Those provides a significant improvement compared to the M3 model.

1. *L1 Regularization technique*

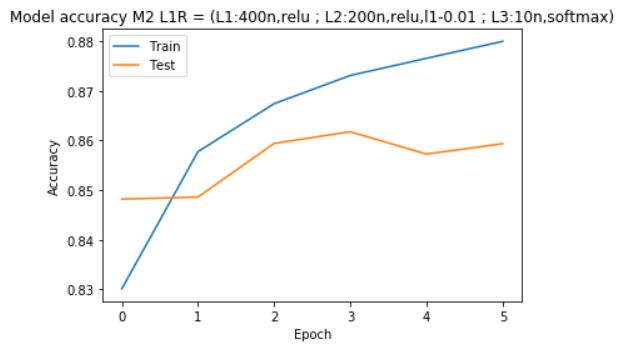
The idea of the L1 regularization technique is to change some of the weights of the feature by significantly decreasing them which leads to “ignore” purposely some parameters features in some nodes. The difference between l1 and l2 regularization is that l1 is considering the absolute value of the weight.

The method has been applied to M2and M3 model architecture.

For M2 L1R the architecture is:

* L1:400n+relu+l1=0.01 ; L2:200n+relu+l1=0.01 ; L3:10n+softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:



Despite that we can observe a beginning of an overfitting after 1 epoch, this model is still better than the M2 model since that the divergence in term of accuracy starts after 3 epochs unlike the M2 model where the divergence starts after 1 epoch.

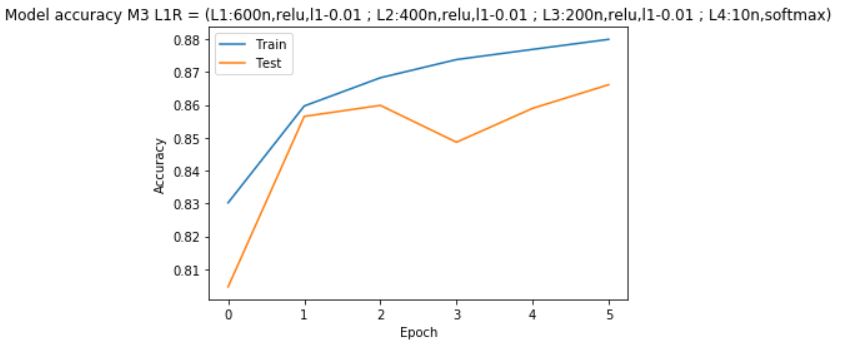
However, the M2 L1R model does not offer a better model in comparison to the M2 DO model since that its accuracy is pretty low (0.86) and overfitting starts quite early.

It could be interesting to change the l1 parameters to a much more lower value to see if it improves significantly the model.

For M3 L1R the architecture is:

* L1:600n+relu+l1=0.01; L2:400n+relu+l1=0.01 ; L3:200n+relu+l1=0.01 ; L4:10n+softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:



The result is quite similar to M2 L1R: We have a pretty low accuracy of 0.86 after 2 epochs and the model starts to overfits after 3 epochs.

The l1 regularization technique does not provide a very strong improvement in comparison to the Drop Out technique; but the improvement is still visible if we compare with the M3 original model.

Ideally, some more tests can be conducted by lowering the value of l1 regularization to see if better results could be achieved.

1. *L2 Regularization technique*

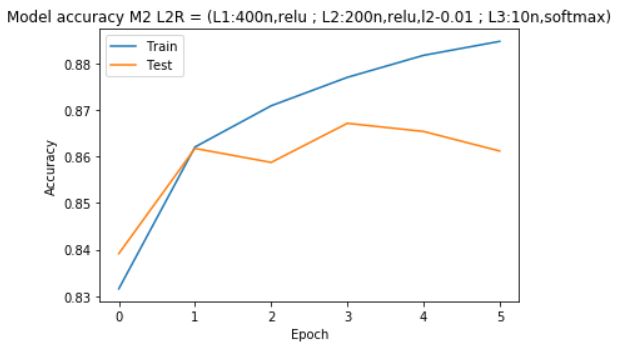
The idea of the L2 regularization technique is to change some of the weights of the feature by significantly decreasing them which leads to “ignore” purposely some parameters features in some nodes. The difference between l1 and l2 regularization is that l2 is considering the squared value of the weight.

The method has been applied to M2 and M3 model architecture.

For M2 L2R the architecture is:

* L1:400n+relu+l2=0.01 ; L2:200n+relu+l2=0.01 ; L3:10n+softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:



For this M2 L2R model, the behavior is quite similar to the M2 L1R model: we can observe a beginning of an overfitting after 1 epoch but this model is still better than the M2 model since that the divergence in term of accuracy starts after 3 epochs.

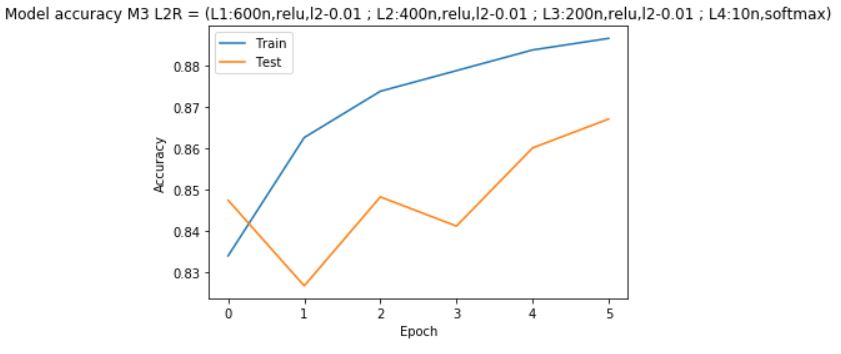
Again, this model does not offer a better model in comparison to the M2 DO model since that its accuracy is pretty low (0.86) and overfitting starts quite early.

It could be interesting to change the l2 parameters to a much more lower value to see if it improves significantly the model.

For M3 L2R the architecture is:

* L1:600n+relu+l2=0.01; L2:400n+relu+l2=0.01 ; L3:200n+relu+l2=0.01 ; L4:10n+softmax

Here is the graph showing the loss and accuracy of the training and the validation after each epoch:



This result is far worse than the other models. This probably due to the fact that l2 regularization has been applied to all layers and the model has “lost” its coherency to fit the training data. However, we can notice that at the end of the epoch 5, the test accuracy is joining the training accuracy. The model could be trained a little bit longer to obtain probably a better result.

Jut like the l1 regularization technique, the l2 regularization technique does not provide a very strong improvement in comparison to the Drop Out technique; but the improvement is still visible if we compare with the M3 original model.

Ideally, some more tests can be conducted by lowering the value of l2 regularization or to increase the number of epoch (especially for M3 L2R model) to see if better results could be achieved.

**Part C: Research**

Deep learning has benefitted from a range of algorithmic advances over the last 10 years. One of the big improvement done in those past years for the algorithm performance as an optimizer in Deep Learning is the Adam Optimization Algorithm.

In this part, we will focus on the Adam Optimization Algorithm technique and explain its concept.

1. **Adam Optimization Algorithm Overview**

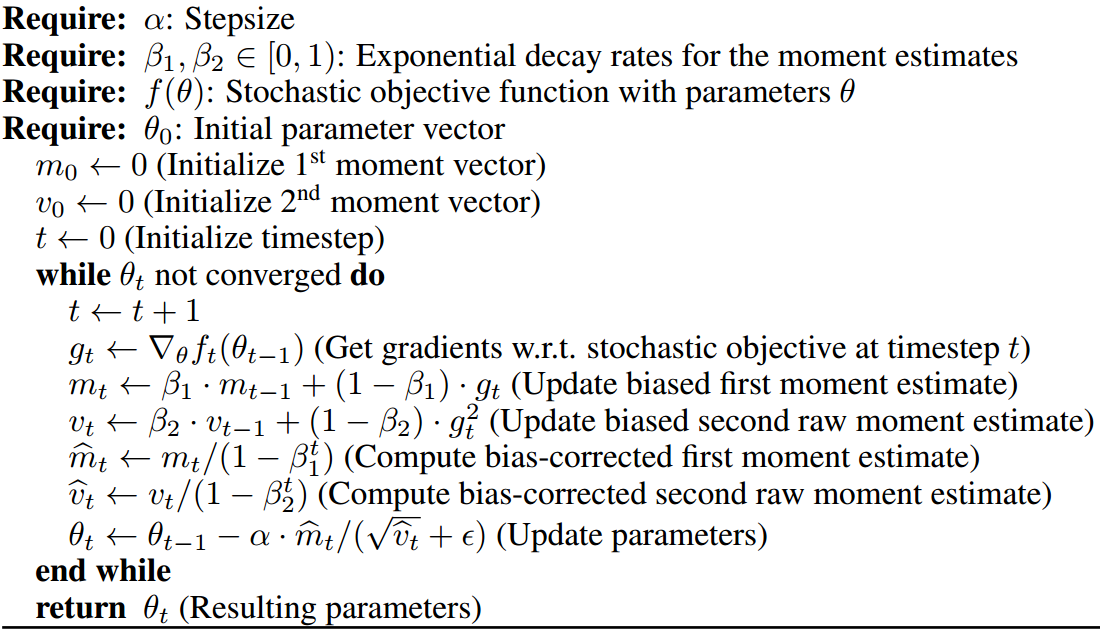
ADAM is an algorithm proposed by **Diederik P. Kingma** and **Jimmy Lei Ba** in 2015. The names comes from “adaptive moment estimation” and it is part of the Stochastic Gradient Descent based optimization. But unlike the regular SGD, the ADAM does not maintain a fixed learning rate: This learning rate is adapted after each timestep.

The concept of the Adam method is to combine the strength of 2 widely used optimization algorithm at that time:

* Adaptive Gradient Algorithm (AdaGrad) very well adapted for wide spread gradient problems (such as language).
* and Root Mean Square Propagation (RMSProp) very well adapted for noisy gradients problem since it maintains an adaptive per-parameter learning rates

The idea is to use the same principle of RMSProp by using an adaptative parameter learning rates based on the 1st order moment (gradient mean) average and extend it at the same time on the 2nd order moment (element-wise squared gradient). At the end, after correction of the moments bias, the final weight is updated proportionally to the learning rate and the 1st order moment divided by the square root of the 2nd order moment.

You can find below the pseudo code of the ADAM algorithm:



*Pseudocode from “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”,* **Diederik P. Kingma**, **Jimmy Lei Ba**

In this pseudo code, we can see the usual steps of a SGD (lines 1, 2,3, 5 and 7 inside the while loop, after the initialization steps):

1) in line 1, the timestep is updated after each iteration

2) In line 2, gt, the gradient is computed

3) In line 3, mt, the 1st order moment is calculated using the gradient mean

4) In line 4, vt, the 2nd order moment is calculated using the gradient squared: This is the true added value of the Adam optimization technique which makes its difference to other SGD algorithm.

5) In line 5 and line 6, mt and vt are both corrected. Indeed, these bias corrections provides more weight on the last computed gradient compared to their arbitrary initialized value.

6) In line 7, the parameters are updated according the computed bias-corrected moments mt and vt. You can note that the 2nd order moment is square-rooted since its gradient value was initially squared.

An important point to be highlighted in this pseudo code is that the ADAM algorithm takes 3 hyperparameters:

* α: the learning rate parameter
* β1: the decay rate of 1st-order moment
* β2: the decay rate of 2nd-order moment

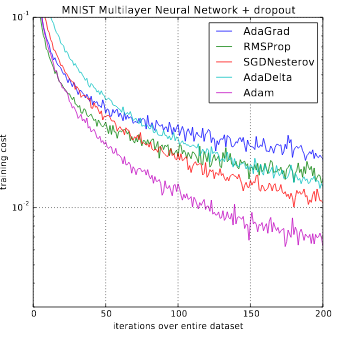
You can notice that the parameters update are mostly dependent on the learning rate α which makes easier to select the scale of the learning rate.

1. **Adam Optimization Strength and Weakness**

ADAM Algorithm is very used in the Deep Learning optimization process and becomes a very popular optimizer over the past few years.

In his paper “*An overview of gradient descent optimization algorithms*” (2016), Sebastien Ruder specified that If “the data is sparse”, the best results could be achieved by an “adaptive learning-rate methods”.

Here are a graph, comparing the losses of different algorithm in comparison to Adam:



*Figure from “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”,* **Diederik P. Kingma**, **Jimmy Lei Ba**

This is more particularly the case of Adam optimizer. Indeed, Kingma and Ba (“*Adam: a Method for Stochastic Optimization*”) , demonstrates that “its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser”.

Moreover, some derivative of the Adam algorithm has been created to rely on its algorithm strength such as:

* AdaMax: In this algorithm, the idea is to replace the 2nd order moment with a more generic form of pth order moment and more specifically with the “infinityth” order.
* and Nadam: In this algorithm, the idea is to mix the strength of Adam and Nesterov momentum methodology. Indeed, for Adam, the moments are updated with respect to the current gradient which in fact is not dependent. Therefore, the concept is to calculate the moments using a Nesterov momentum coefficient for the gradient. This algorithm performs actually better than Adam.

However, despite his great strength, Adam algorithm has shown some limitation. Indeed, their paper “ON THE CONVERGENCE OF ADAM AND BEYOND” (2018), Sashank J. Reddi, Satyen Kale and Sanjiv Kuma, from Google have demonstrated that Adam algorithm does not always converge. Many flows and mistakes were present to prove its convergence.

Some algorithm has tried to correct this conv**er**gence mistake such as Amsgrad, but this algorithm was not been proven as more effective than Adam.

In conclusion, despite its flaw, Adam (and its derived algorithm) is probably one of the most used methodology optimization for the Deep Learning application and its simplicity in its usage made this algorithm among the most popular ones. Even if Adam cannot be applied in some of area, people are still continuing to work on it in order to bring Adam results as the most optimal ones.