Submitted To:

Prof. PKJ Mohapatra

Submitted By: (2 Members)

Shobhit (19CS06008)

Rohit Agarwal (19CS06002)

School of Electrical Sciences

MTECH Computer Science and Engineering

IIT Bhubaneswar

Machine Learning Project

Problem Statement:

Implement the sign language recognition using an image dataset where the image contains the signs of the digits and the output should be the corresponding digit. Implement the supervised classification. The dataset file are attached within the same folder. Create train, test and validation data for correct evaluation of the neural network and plot the training and the validation loss. Note that this is just a demo of sign recognition where we consider only signs of digits, we can always increase the signs in dataset and train the model accordingly. Implement using Convolutional Neural Network as well.

Keywords:

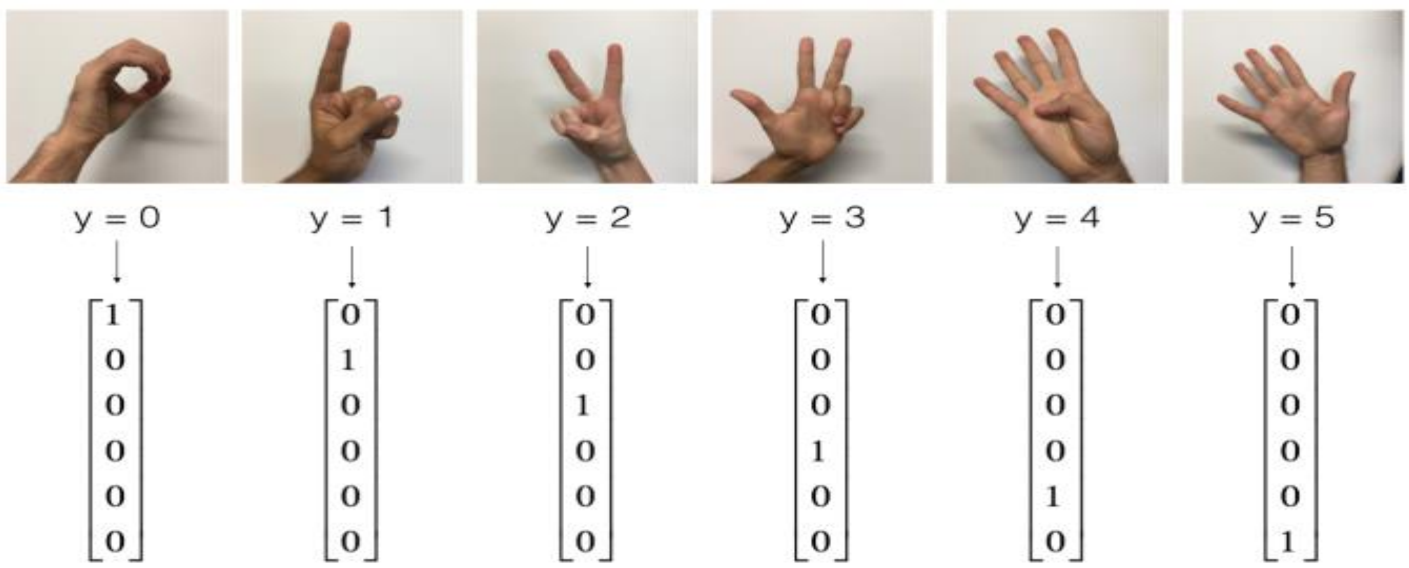
Mini Batches, Backpropagation Algorithm, Feature Scaling, RMSProp, Momentum, Adam Optimizer, Forward Propagation, Convolutional Neural Network, One Hot Encoding, Softmax Activation, Xavier Initialization.

What to Expect:

1) Convolutional Neural Network is also implemented for this using keras library.
2) The implementation does not include any library functions and is purely a python code.
3) Different optimizers were used to check the implementation.
4) Vectorized Implementation of the data.

Theory: (A short details of concepts what are implemented)

Format of dataset: The dataset that we have considered here is an image which is a hand sign of digits where the background is of white and the hand shows the digit. For example, if the image shows three fingers than the output should be 3. So the dataset is labelled with digit showed i.e. number of fingers in the image. The example of the dataset is as follows



$$\begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\1\\0\\0\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\1\\0\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\0\\1\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\0\\0\\1\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\0\\0\\0\\1\end{bmatrix}$$

Weights Initialization: The random weight initialization can lead to vanishing gradient problem or explosion of gradients. The multiplication of the term

$$\sqrt{\frac{1}{size^{[l-1]}}}$$

Where $size^{[l-1]}$ is the number of neurons in the previous layer

This weights are set neither too much bigger than 1, nor too much less than 1, so it avoids the vanishing gradient problem as well as exploding gradient. The main objective is to set the parameters in such a way so that the variance is minimized.

Optimizers: The different optimizers used are gradient descent with momentum, RMSProp and combination of both i.e. Adam Optimizer. The momentum is added to avoid the results to get stuck in the local optimum and the oscillations are not too big so that they converge faster and the oscillations are in the right direction. RMSProp adds a root mean square and Adam optimizer combines these two. Different optimizers can be used to check which optimizer can be best in different scenarios. The equations are:

$$w_3 = w_3 - \alpha * (\frac{vdw_3}{\sqrt{sdw_3} + \varepsilon})$$

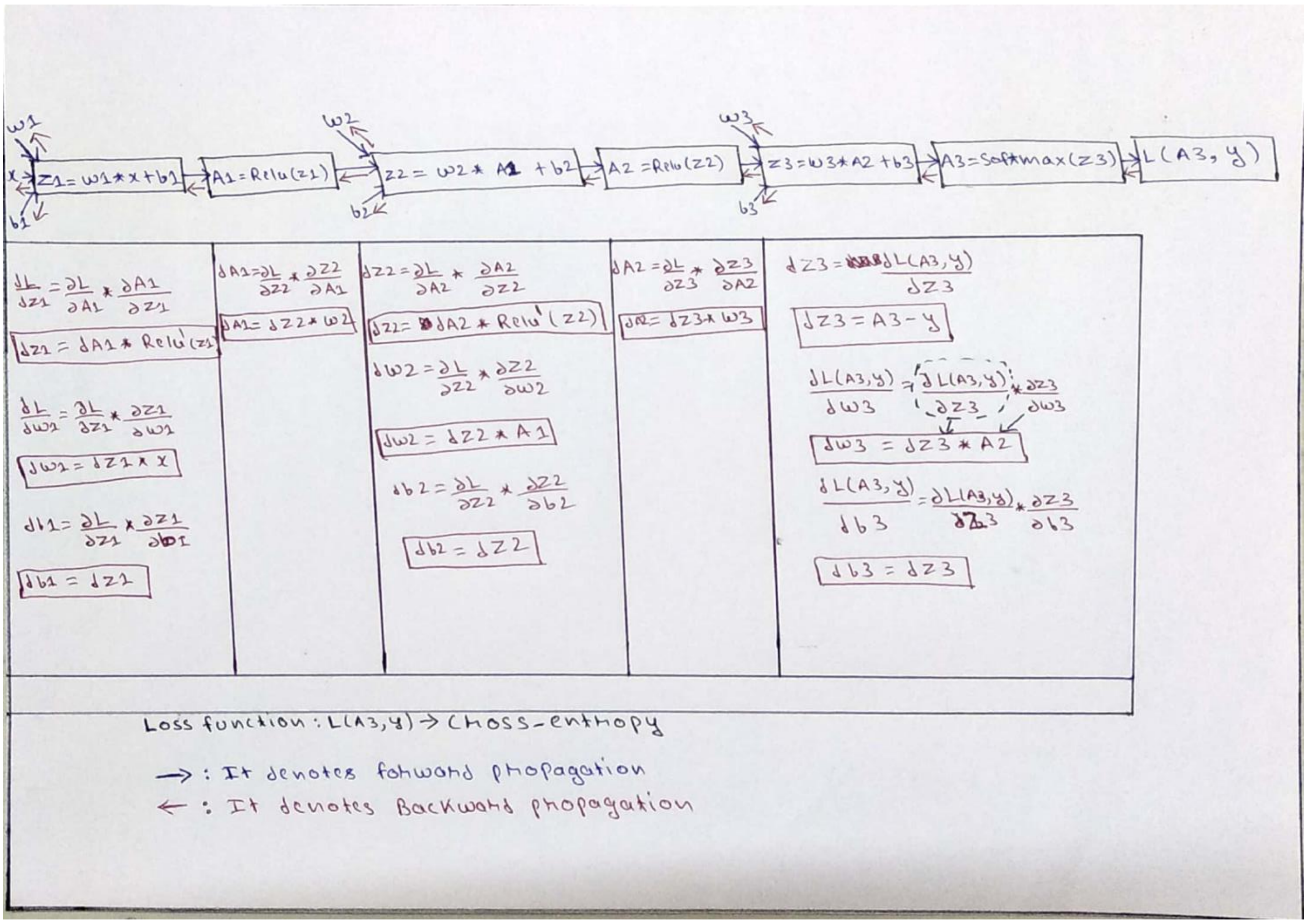
Where $vdw_3 = \beta_1 * (vdw_3) + (1 - \beta_1)dw_3$

$sdw_3 = \beta_2 * (sdw_3) + (1 - \beta_2){dw_3}^2$

$\varepsilon = 10^{-8}$ *is to avoid divide by zero error*

Feature Scaling: The image is converted into a column vector and all the column are stacked together to form a matrix of dimensions (n_x, m) where n_x is the number of pixels height * width * channels and m is the number of training example or in our case since we have used the mini batch it is the mini batch size. The pixel values range from 0 to 255 but it can lead to divergence thus we have to divide the number by 255 so that the range becomes 0 to 1 we have in avoiding number explosions.

Back and Forward Propagation: The following handwritten computation graph image shows how the computations flow from the input layer to the output layer and back from output to the input to reduce the loss function used. The loss function used here is the cross entropy formula which is same as the sigmoid function only the fact changes is that the number of classes increase in the output label since we use a softmax function. The 'dxxx' in the graph represent the gradients backwards which needs to subtracted. The computation is the exact calculation which is taking place to calculate the output and includes all the activation functions and the derivatives to calculate the gradients. The computation graph of the simple neural network is as follows:



Softmax Function and One Hot Encoding: Softmax turn logits (numeric output of the last linear layer of a multi-class classification neural network) into probabilities by take the exponents of each output and then normalize each number by the sum of those exponents so the entire output vector adds up to one - all probabilities should add up to one. Cross entropy loss is usually the loss function for such a multi-class classification problem. The equation is:

$$\sigma(z)_i = \frac{e^{-\beta z_i}}{\sum_{j=1}^{K} e^{-\beta z_j}} \quad , for\ i = 1, \dots , K$$

One hot encoding is important when it comes to softmax as only the class which the image belongs is set to one rest all are made 0. Thus the name one hot where only one entry is hot i.e. set. The one hot encoding is the example of one – all multi class classification. The above dataset example shows the one hot encoding example.

Results: (Artificial Neural Networks)

Training Data Size: 972 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5.

Validation Data Size: 108 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5.

Testing Data Size: 120 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (20 pictures per number).

Hyperparameters:

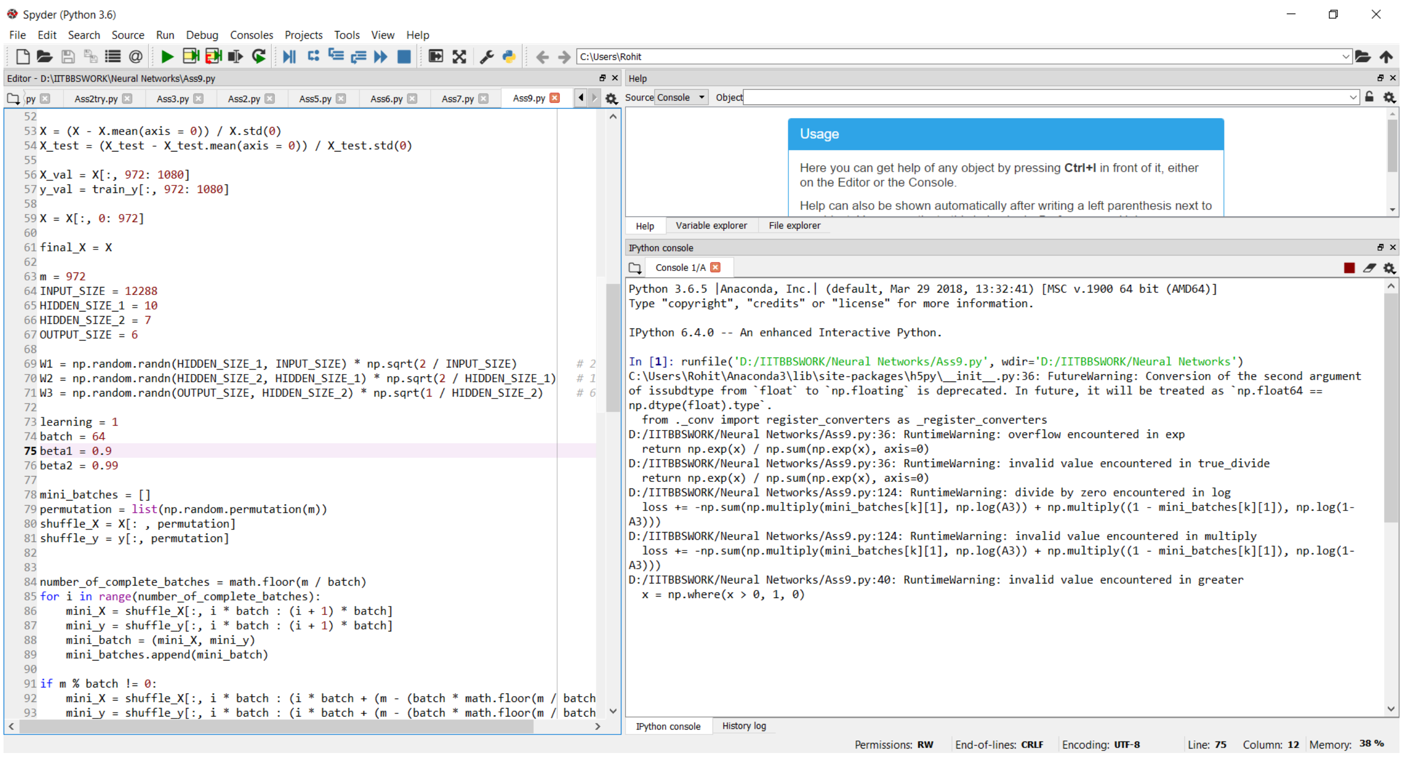
The hyperparameters which are selected are based on trial and basis and considered which avoids the problem of high variance and high bias. The hyperparameters which suits the based were considered and the set was normally of the standard and the selection was mainly from these sets. So below we present the set and how we came up with those values.

Beta1 = 0.9, Beta2 = 0.99, epsilon = 10^{-8}

These are the values which are taken from the Adam Optimizer base paper as it is [2].

Learning rate: 0.0075

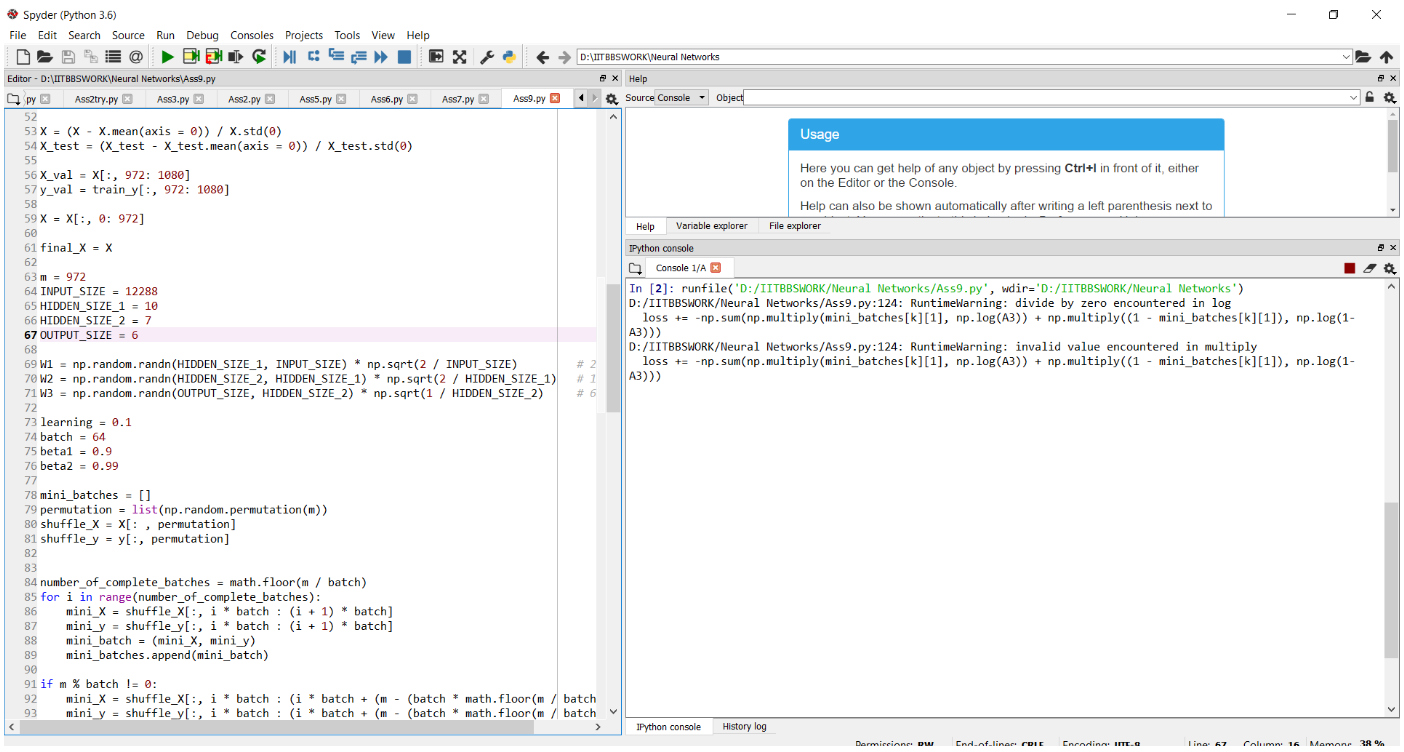
The different learning rates we considered were from the set {1, 0.1, 0.01, 0.001}. The value that we chose was to accomplish a faster learning as the dataset is not huge and we need an optimum value. Thus to learn faster we took the value ‘1’ from the set. The learning rate ‘1’ resulted in number out of boundary (number of neurons were decided in later stages of development as one needs to be kept constant and tuning of other parameters needs to be done around it. So we decided first to find learning rate and later tune other parameters based on it.). Thus we got ‘nan’ errors while forwarding propagating and hence resulted in removal of ‘1’ from the set. In the image you can see the overflow errors when learning rate is 1.


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Rohit
Editor - D:\IITBBSWORK\Neural Networks\Ass9.py
py Ass2try.py Ass3.py Ass2.py Ass5.py Ass6.py Ass7.py Ass9.py
Help
Source Console Object
Usage
Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.
Help can also be shown automatically after writing a left parenthesis next to
Help Variable explorer File explorer
IPython console
Console 1/A
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 75 Column: 12 Memory: 38 %

```

X = (X - X.mean(axis = 0)) / X.std(0)
X_test = (X_test - X_test.mean(axis = 0)) / X_test.std(0)

X_val = X[:, 972: 1080]
y_val = train_y[:, 972: 1080]

X = X[:, 0: 972]

final_X = X

m = 972
INPUT_SIZE = 12288
HIDDEN_SIZE_1 = 10
HIDDEN_SIZE_2 = 7
OUTPUT_SIZE = 6

W1 = np.random.randn(HIDDEN_SIZE_1, INPUT_SIZE) * np.sqrt(2 / INPUT_SIZE)         # 2
W2 = np.random.randn(HIDDEN_SIZE_2, HIDDEN_SIZE_1) * np.sqrt(2 / HIDDEN_SIZE_1)   # 1
W3 = np.random.randn(OUTPUT_SIZE, HIDDEN_SIZE_2) * np.sqrt(1 / HIDDEN_SIZE_2)     # 6

learning = 1
batch = 64
beta1 = 0.9
beta2 = 0.99

mini_batches = []
permutation = list(np.random.permutation(m))
shuffle_X = X[: , permutation]
shuffle_y = y[:, permutation]


number_of_complete_batches = math.floor(m / batch)
for i in range(number_of_complete_batches):
    mini_X = shuffle_X[:, i * batch : (i + 1) * batch]
    mini_y = shuffle_y[:, i * batch : (i + 1) * batch]
    mini_batch = (mini_X, mini_y)
    mini_batches.append(mini_batch)

if m % batch != 0:
    mini_X = shuffle_X[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
    mini_y = shuffle_y[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
```

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')
C:\Users\Rohit\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the second argument
of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
D:/IITBBSWORK/Neural Networks/Ass9.py:36: RuntimeWarning: overflow encountered in exp
  return np.exp(x) / np.sum(np.exp(x), axis=0)
D:/IITBBSWORK/Neural Networks/Ass9.py:36: RuntimeWarning: invalid value encountered in true_divide
  return np.exp(x) / np.sum(np.exp(x), axis=0)
D:/IITBBSWORK/Neural Networks/Ass9.py:124: RuntimeWarning: divide by zero encountered in log
  loss += -np.sum(np.multiply(mini_batches[k][1], np.log(A3)) + np.multiply((1 - mini_batches[k][1]), np.log(1-
A3)))
D:/IITBBSWORK/Neural Networks/Ass9.py:124: RuntimeWarning: invalid value encountered in multiply
  loss += -np.sum(np.multiply(mini_batches[k][1], np.log(A3)) + np.multiply((1 - mini_batches[k][1]), np.log(1-
A3)))
D:/IITBBSWORK/Neural Networks/Ass9.py:40: RuntimeWarning: invalid value encountered in greater
  x = np.where(x > 0, 1, 0)
```

Surprisingly ‘0.1’ also gave ‘nan’ errors because the learning rate is too large. But one parameters needs to be fixed so that others can be find out based on it. So we eliminated ‘0.1’ from the set and we were left with two values {0.01 and 0.001}. In the image you can see the invalid value encountered when using learning rate as 0.1.


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
D:\IITBBSWORK\Neural Networks
Editor - D:\IITBBSWORK\Neural Networks\Ass9.py
py Ass2try.py Ass3.py Ass2.py Ass5.py Ass6.py Ass7.py Ass9.py
Help
Source Console Object
Usage
Help Variable explorer File explorer
IPython console
Console 1/A
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 67 Column: 16 Memory: 38 %

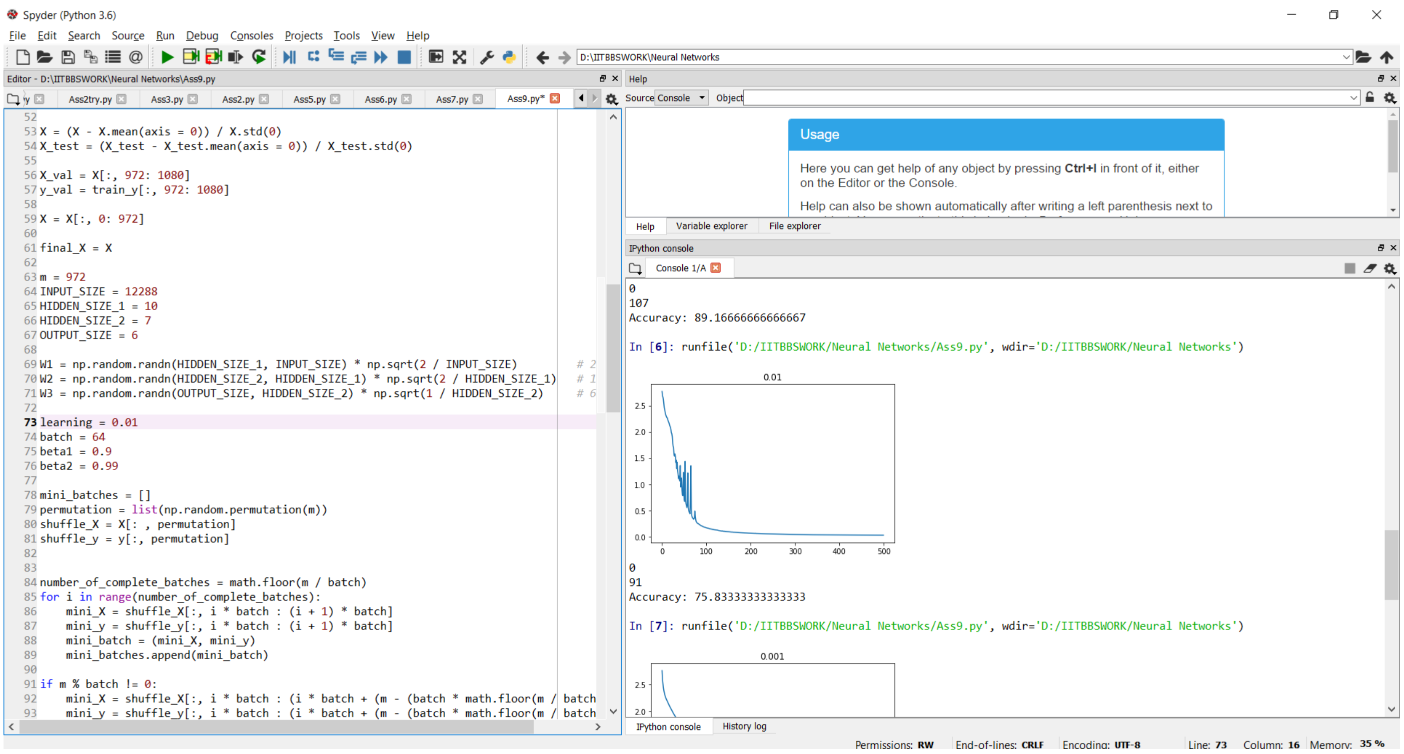
```python

X = (X - X.mean(axis = 0)) / X.std(0)
X_test = (X_test - X_test.mean(axis = 0)) / X_test.std(0)

X_val = X[:, 972: 1080]
y_val = train_y[:, 972: 1080]

X = X[:, 0: 972]

final_X = X

m = 972
INPUT_SIZE = 12288
HIDDEN_SIZE_1 = 10
HIDDEN_SIZE_2 = 7
OUTPUT_SIZE = 6

W1 = np.random.randn(HIDDEN_SIZE_1, INPUT_SIZE) * np.sqrt(2 / INPUT_SIZE)         # 2
W2 = np.random.randn(HIDDEN_SIZE_2, HIDDEN_SIZE_1) * np.sqrt(2 / HIDDEN_SIZE_1)   # 1
W3 = np.random.randn(OUTPUT_SIZE, HIDDEN_SIZE_2) * np.sqrt(1 / HIDDEN_SIZE_2)     # 6

learning = 0.1
batch = 64
beta1 = 0.9
beta2 = 0.99

mini_batches = []
permutation = list(np.random.permutation(m))
shuffle_X = X[: , permutation]
shuffle_y = y[:, permutation]


number_of_complete_batches = math.floor(m / batch)
for i in range(number_of_complete_batches):
    mini_X = shuffle_X[:, i * batch : (i + 1) * batch]
    mini_y = shuffle_y[:, i * batch : (i + 1) * batch]
    mini_batch = (mini_X, mini_y)
    mini_batches.append(mini_batch)

if m % batch != 0:
    mini_X = shuffle_X[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
    mini_y = shuffle_y[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
```

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to

```
In [2]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')
D:/IITBBSWORK/Neural Networks/Ass9.py:124: RuntimeWarning: divide by zero encountered in log
  loss += -np.sum(np.multiply(mini_batches[k][1], np.log(A3)) + np.multiply((1 - mini_batches[k][1]), np.log(1-
A3)))
D:/IITBBSWORK/Neural Networks/Ass9.py:124: RuntimeWarning: invalid value encountered in multiply
  loss += -np.sum(np.multiply(mini_batches[k][1], np.log(A3)) + np.multiply((1 - mini_batches[k][1]), np.log(1-
A3)))
```

So we tried on both the values and calculated the accuracy (The number of neurons were randomly chosen as 12288, 10, 7, 6). The '0.01' value had a promising curve of loss for such randomly chosen parameters as shown in the image below


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
D:\IITBBSWORK\Neural Networks
Editor - D:\IITBBSWORK\Neural Networks\Ass9.py
py Ass2try.py Ass3.py Ass2.py Ass5.py Ass6.py Ass7.py Ass9.py*
Help
Source Console Object
Usage
Help Variable explorer File explorer
IPython console
Console 1/A
0.01
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200
300
400
500
0.001
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 73 Column: 16 Memory: 35 %

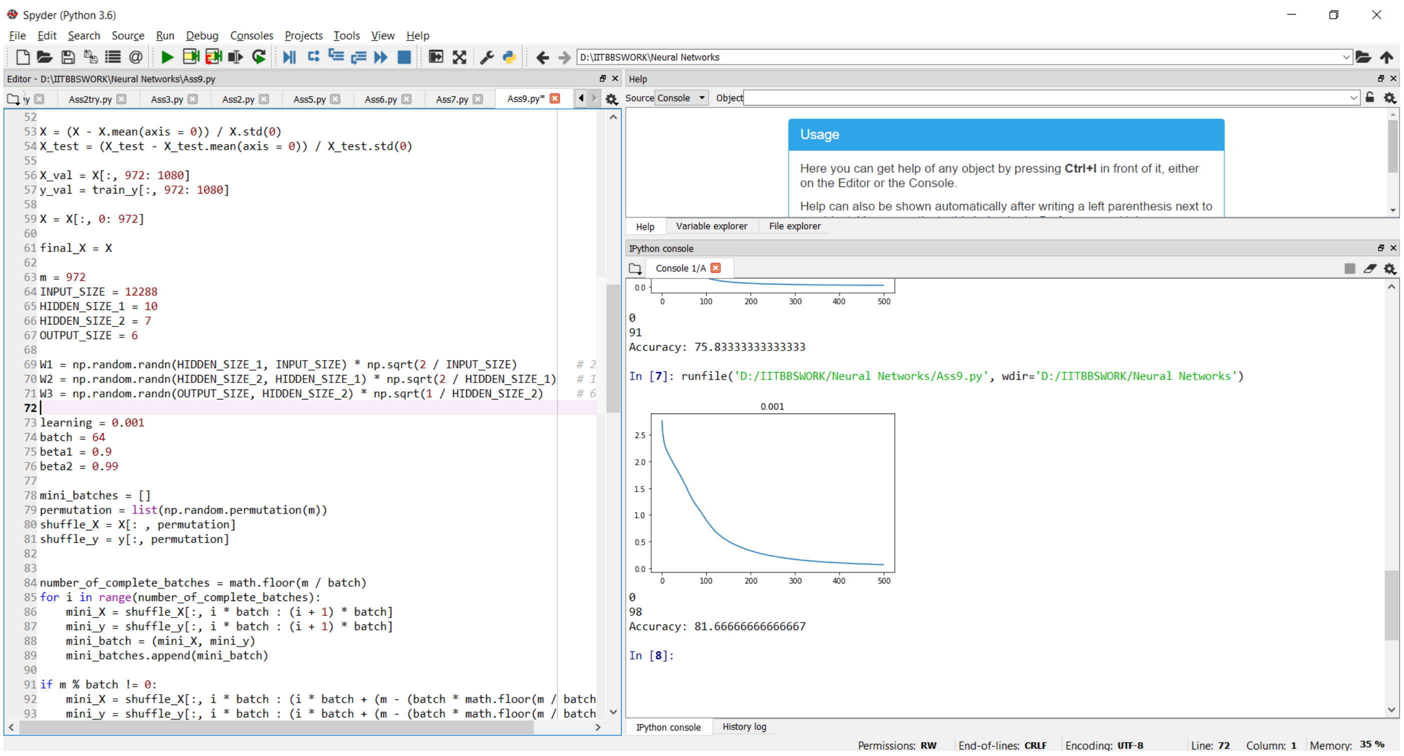
```python

X = (X - X.mean(axis = 0)) / X.std(0)
X_test = (X_test - X_test.mean(axis = 0)) / X_test.std(0)

X_val = X[:, 972: 1080]
y_val = train_y[:, 972: 1080]

X = X[:, 0: 972]

final_X = X

m = 972
INPUT_SIZE = 12288
HIDDEN_SIZE_1 = 10
HIDDEN_SIZE_2 = 7
OUTPUT_SIZE = 6

W1 = np.random.randn(HIDDEN_SIZE_1, INPUT_SIZE) * np.sqrt(2 / INPUT_SIZE)         # 2
W2 = np.random.randn(HIDDEN_SIZE_2, HIDDEN_SIZE_1) * np.sqrt(2 / HIDDEN_SIZE_1)   # 1
W3 = np.random.randn(OUTPUT_SIZE, HIDDEN_SIZE_2) * np.sqrt(1 / HIDDEN_SIZE_2)     # 6

learning = 0.01
batch = 64
beta1 = 0.9
beta2 = 0.99

mini_batches = []
permutation = list(np.random.permutation(m))
shuffle_X = X[: , permutation]
shuffle_y = y[:, permutation]


number_of_complete_batches = math.floor(m / batch)
for i in range(number_of_complete_batches):
    mini_X = shuffle_X[:, i * batch : (i + 1) * batch]
    mini_y = shuffle_y[:, i * batch : (i + 1) * batch]
    mini_batch = (mini_X, mini_y)
    mini_batches.append(mini_batch)

if m % batch != 0:
    mini_X = shuffle_X[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
    mini_y = shuffle_y[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
```

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to

```
0
107
Accuracy: 89.16666666666667

In [6]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')

0
91
Accuracy: 75.83333333333333

In [7]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')
```

The '0.001' had an accuracy greater than '0.01' value and a loss function as shown below:


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
D:\IITBBSWORK\Neural Networks
Editor - D:\IITBBSWORK\Neural Networks\Ass9.py
Ass2try.py
Ass3.py
Ass2.py
Ass5.py
Ass6.py
Ass7.py
Ass9.py*
Help
Source
Console
Object
Usage
Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.
Help can also be shown automatically after writing a left parenthesis next to
Help
Variable explorer
File explorer
IPython console
Console 1/A
0.001
IPython console
History log
Permissions: RW
End-of-lines: CRLF
Encoding: UTF-8
Line: 72
Column: 1
Memory: 35 %

```python
X = (X - X.mean(axis = 0)) / X.std(0)
X_test = (X_test - X_test.mean(axis = 0)) / X_test.std(0)

X_val = X[:, 972: 1080]
y_val = train_y[:, 972: 1080]

X = X[:, 0: 972]

final_X = X

m = 972
INPUT_SIZE = 12288
HIDDEN_SIZE_1 = 10
HIDDEN_SIZE_2 = 7
OUTPUT_SIZE = 6

W1 = np.random.randn(HIDDEN_SIZE_1, INPUT_SIZE) * np.sqrt(2 / INPUT_SIZE)        # 2
W2 = np.random.randn(HIDDEN_SIZE_2, HIDDEN_SIZE_1) * np.sqrt(2 / HIDDEN_SIZE_1)   # 1
W3 = np.random.randn(OUTPUT_SIZE, HIDDEN_SIZE_2) * np.sqrt(1 / HIDDEN_SIZE_2)     # 6

learning = 0.001
batch = 64
beta1 = 0.9
beta2 = 0.99

mini_batches = []
permutation = list(np.random.permutation(m))
shuffle_X = X[: , permutation]
shuffle_y = y[:, permutation]


number_of_complete_batches = math.floor(m / batch)
for i in range(number_of_complete_batches):
    mini_X = shuffle_X[:, i * batch : (i + 1) * batch]
    mini_y = shuffle_y[:, i * batch : (i + 1) * batch]
    mini_batch = (mini_X, mini_y)
    mini_batches.append(mini_batch)

if m % batch != 0:
    mini_X = shuffle_X[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
    mini_y = shuffle_y[:, i * batch : (i * batch + (m - (batch * math.floor(m / batch
```

```
0
91
Accuracy: 75.83333333333333

In [7]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')

0
98
Accuracy: 81.66666666666667

In [8]:
```

Thus selecting between the two values became difficult as ‘0.01’ showed lesser accuracy but the model was not tuned for other parameter it can be considered as a good one but ‘0.001’ had a better accuracy for a not tuned model. So we decided to keep a learning rate closer to 0.01 which shows better than results as 0.001 can be considered as kind of overfitting for not at all tuned model. Thus we ended up with a value closer to 0.01 and between 0.01 and 0.001 which was not in the set. This is how we decided to keep 0.0075 as our learning rate.

Mini Batch Size: 64

The best practice is to keep the mini-batch size as power of 2. Thus we considered 2 values here i.e. 32 and 64. The value 16 would have been too small degrading the benefits of vectorization of inputs provided by python internally and 128 would have been too large considering the size of dataset. Since the number of iterations were 1000 we decided to keep 64 as the batch size so that it gets train more properly. This is of no big concern because we are applying Adam optimizer technique which would definitely dampen the oscillations i.e. errors.

Iterations: 1000

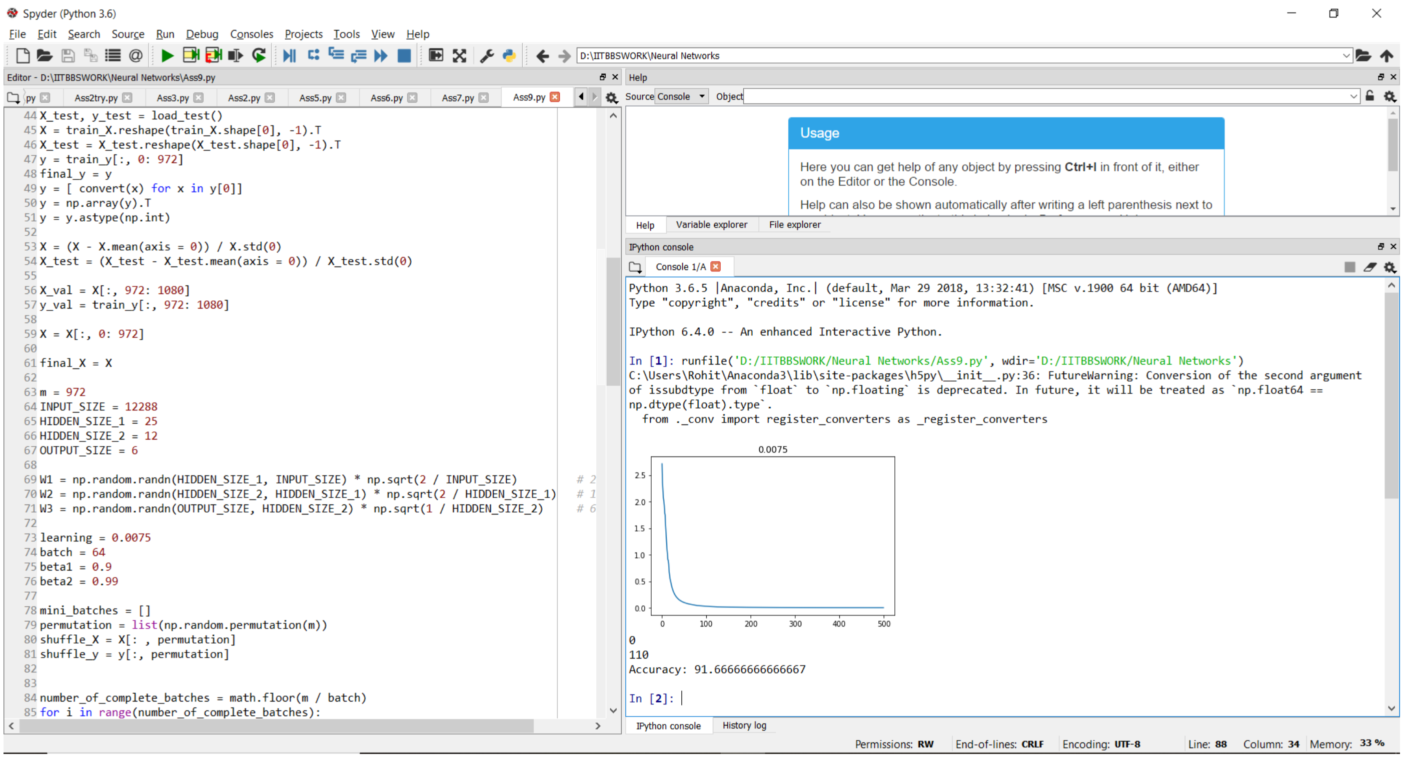
This parameter was considered as it is and no changes were made. Since the dataset size is small and number of iterations generally don’t play a huge impact when dataset is small we kept it throughout the tuning phase as 1000.

Number of layers: 3 (Excluding Input)

The number of layers denotes the complexity of features that needs to be learned and decision boundary it needs to create such that the classification is done erroneously. Since the image dataset is quite simple to be learned because here we consider that the image background is white only (no other colour image) and number of fingers shown in the image denotes the digits we kept the number of hidden layers as 2 and an output layer with 6 nodes (6 classes) and we varied the number of nodes in the hidden layer to increase the accuracy of the model.

Number of Neurons: 12288, 100, 50, 6

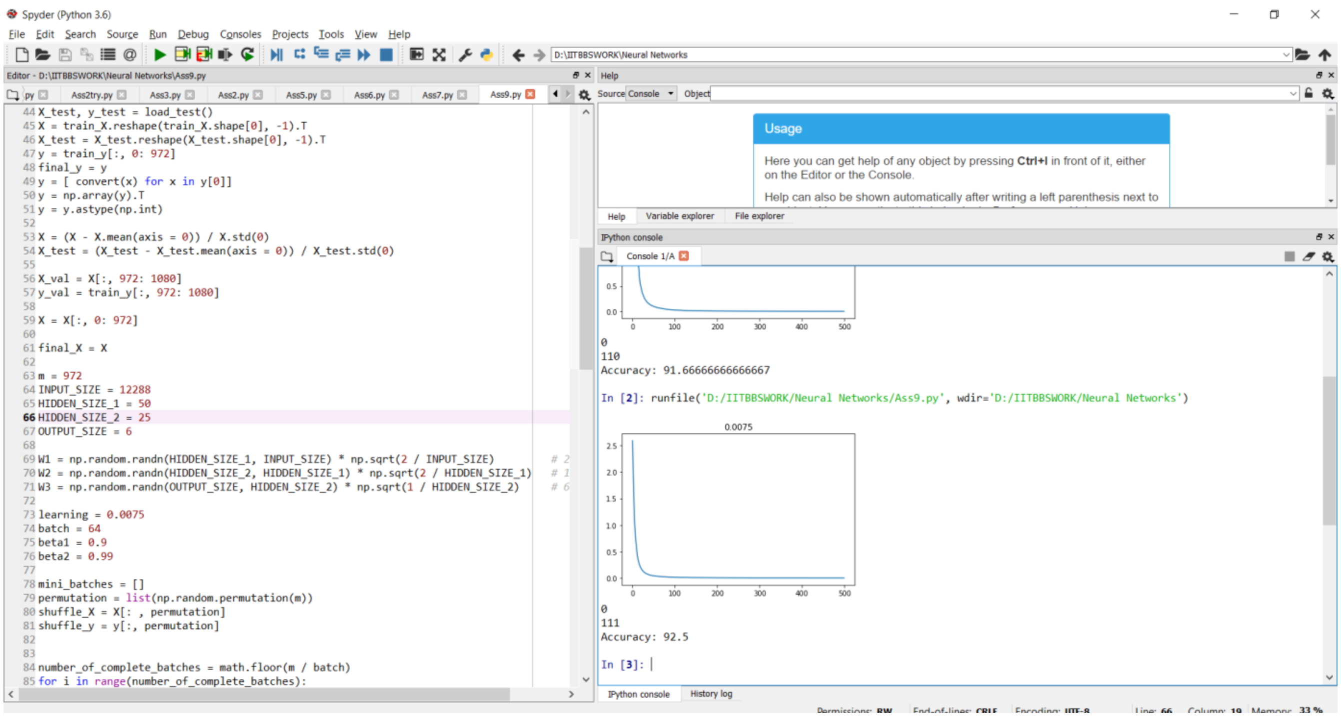
The number of neurons in the hidden layers needs to be tuned properly so that the learning of the complex decision boundary is optimum. Thus choosing number of neurons is important. The method that we applied for choosing considers the time for execution, high variance problem, and good testing accuracy. The values of hyperparameters that we kept constant here are learning rate, number of layers and iterations. Thus we started with a model having neurons (12288, 500, 100, 6). Since this is a pure python code and no usage of GPU libraries, the computation time was exceptionally huge since the matrix multiplication was of matrices having dimensions (12288, 500), (500, 100), (100, 6) and (12288, 64) (here 64 is the batch size). So we planned to start from low neurons and double the neurons in each trial just so the trade-off of the computation time and the testing accuracy. The testing accuracy should be made high so that there is not any high variance problem if the image apart from dataset is given. (Note: The image should follow the same distribution as that of training dataset else there is no sense in giving that as an input.). The output had a size 6 because the number of classes were 6 we tried to just double the neurons from back. So the 2nd hidden layer had 6 * 2 = 12 neurons. The 1st hidden layer had 12 * 2 ~ 25 and the input layer will have the number of pixels as in the image dataset. The results are shown in the image which shows the tuned values of hyperparameters i.e. number of neurons in each layer and we find that the accuracy of testing is 91.33 which is not bad considering the training accuracy of almost 99 percent. But this results may signify a hint of high variance problem and needs to be reduced. The computation time for this was low and thus there is no harm in increasing the number of neurons in the hidden layers.


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
D:\IITBBSWORK\Neural Networks
Editor - D:\IITBBSWORK\Neural Networks\Ass9.py
py Ass2try.py Ass3.py Ass2.py Ass5.py Ass6.py Ass7.py Ass9.py
Help
Source Console Object
Usage
Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.
Help can also be shown automatically after writing a left parenthesis next to
Help Variable explorer File explorer
IPython console
Console 1/A
0.0075
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 88 Column: 34 Memory: 33 %

```python
X_test, y_test = load_test()
X = train_X.reshape(train_X.shape[0], -1).T
X_test = X_test.reshape(X_test.shape[0], -1).T
y = train_y[:, 0: 972]
final_y = y
y = [ convert(x) for x in y[0]]
y = np.array(y).T
y = y.astype(np.int)

X = (X - X.mean(axis = 0)) / X.std(0)
X_test = (X_test - X_test.mean(axis = 0)) / X_test.std(0)

X_val = X[:, 972: 1080]
y_val = train_y[:, 972: 1080]

X = X[:, 0: 972]

final_X = X

m = 972
INPUT_SIZE = 12288
HIDDEN_SIZE_1 = 25
HIDDEN_SIZE_2 = 12
OUTPUT_SIZE = 6

W1 = np.random.randn(HIDDEN_SIZE_1, INPUT_SIZE) * np.sqrt(2 / INPUT_SIZE)          # 2
W2 = np.random.randn(HIDDEN_SIZE_2, HIDDEN_SIZE_1) * np.sqrt(2 / HIDDEN_SIZE_1)    # 1
W3 = np.random.randn(OUTPUT_SIZE, HIDDEN_SIZE_2) * np.sqrt(1 / HIDDEN_SIZE_2)      # 6

learning = 0.0075
batch = 64
beta1 = 0.9
beta2 = 0.99

mini_batches = []
permutation = list(np.random.permutation(m))
shuffle_X = X[: , permutation]
shuffle_y = y[:, permutation]


number_of_complete_batches = math.floor(m / batch)
for i in range(number_of_complete_batches):
```

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')
C:\Users\Rohit\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the second argument
of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  from ._conv import register_converters as _register_converters

0
110
Accuracy: 91.66666666666667

In [2]:
```

Thus again we used the same logic of doubling the neurons in each layer and we ended up with 25 * 2 = 50 neurons and 12 * 2 ~ 25 neurons (these 12 * 2 ~ 25 approximations are in the sense to make computations for us simpler since doubling becomes easier). At this moment the computation time have decreased a bit and the accuracy of testing is 92.5 but the model has again a hint of high variance since the difference in testing and training accuracy is considerable. The computation time has not gone bad so we can try to increase the neurons again by doubling it. The results are shown in the below image i.e. the loss v/s iteration curve and the hyperparameters.



```
X_test, y_test = load_test()
X = train_X.reshape(train_X.shape[0], -1).T
X_test = X_test.reshape(X_test.shape[0], -1).T
y = train_y[:, 0: 972]
final_y = y
y = [ convert(x) for x in y[0]]
y = np.array(y).T
y = y.astype(np.int)

X = (X - X.mean(axis = 0)) / X.std(0)
X_test = (X_test - X_test.mean(axis = 0)) / X_test.std(0)

X_val = X[:, 972: 1080]
y_val = train_y[:, 972: 1080]

X = X[:, 0: 972]

final_X = X

m = 972
INPUT_SIZE = 12288
HIDDEN_SIZE_1 = 50
HIDDEN_SIZE_2 = 25
OUTPUT_SIZE = 6

W1 = np.random.randn(HIDDEN_SIZE_1, INPUT_SIZE) * np.sqrt(2 / INPUT_SIZE)          # 2
W2 = np.random.randn(HIDDEN_SIZE_2, HIDDEN_SIZE_1) * np.sqrt(2 / HIDDEN_SIZE_1)    # 1
W3 = np.random.randn(OUTPUT_SIZE, HIDDEN_SIZE_2) * np.sqrt(1 / HIDDEN_SIZE_2)      # 6

learning = 0.0075
batch = 64
beta1 = 0.9
beta2 = 0.99

mini_batches = []
permutation = list(np.random.permutation(m))
shuffle_X = X[: , permutation]
shuffle_y = y[:, permutation]


number_of_complete_batches = math.floor(m / batch)
for i in range(number_of_complete_batches):
```

```
0
110
Accuracy: 91.66666666666667

In [2]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')

0
111
Accuracy: 92.5

In [3]:
```

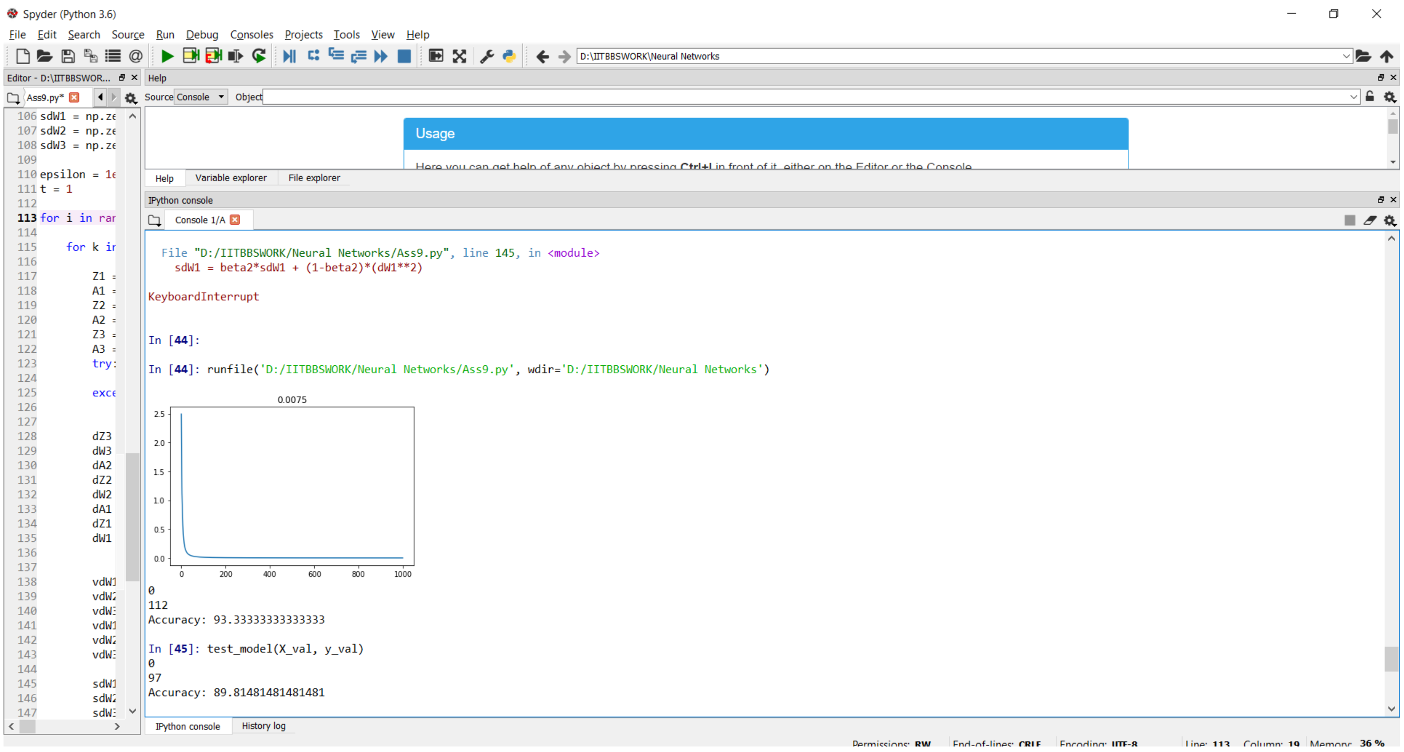
So next we doubled the number of neurons in the layers from (50, 25) to (100, 50) and once again obtained the result by keeping the hyperparameters constant mentioned above. The results we got are described below and the image shows the same. But at this moment of time the computation time have risen so much that we did not think to further increase the size of hidden layers and the testing accuracy was considerable. There could have been more improvement possible we could have increased the number of neurons but the dataset size is small and we have not considered GPUs in our implementation. So the computation time has increased too much and not feasible to increase anymore neurons in the hidden layers.

The training accuracy is: 99.53%

The validation accuracy is: 89.81%

The testing accuracy is: 93.33% (106 correct out of 120)

The training loss v/s iteration graph



```
sdW1 = np.ze
sdW2 = np.ze
sdW3 = np.ze

epsilon = 1e
t = 1

for i in rar

    for k ir

        Z1 =
        A1 =
        Z2 =
        A2 =
        Z3 =
        A3 =
        try:

        exce


        dZ3
        dW3
        dA2
        dZ2
        dW2
        dA1
        dZ1
        dW1


        vdW1
        vdW2
        vdW3
        vdW1
        vdW2
        vdW3

        sdW1
        sdW2
        sdW3
```

```
  File "D:/IITBBSWORK/Neural Networks/Ass9.py", line 145, in <module>
    sdW1 = beta2*sdW1 + (1-beta2)*(dW1**2)

KeyboardInterrupt


In [44]:

In [44]: runfile('D:/IITBBSWORK/Neural Networks/Ass9.py', wdir='D:/IITBBSWORK/Neural Networks')

0
112
Accuracy: 93.33333333333333

In [45]: test_model(X_val, y_val)
0
97
Accuracy: 89.81481481481481
```

Convolutional Neural Network:

The changes with respect to the Artificial Neural Network implemented before are:

1) Model
2) L1, L2 Regularization used.
3) Loss Function is Categorical Cross entropy
4) Batch Normalization (Normalizing channels)

The model of Convolutional Neural Network is as follows:

1) Convolution (kernels = 32, Kernel Size = (5, 5), strides = (1, 1), padding = valid)
2) Batch Normalization(axis = 3)
3) Activation = relu
4) MaxPooling(kernel size = (2, 2))
5) Convolution(kernels = 64, kernel size = (7, 7), strides = (2, 2), padding = valid)
6) Batch Normalization(axis = 3)
7) Activation = relu
8) MaxPooling(kernel size = (2, 2))
9) Flatten
10) Dense(576, activation = relu)
11) Dense(6, activation = softmax)

Hyperparameters:

Learning rate = 0.001

Epochs = 40

Validation = 20%

Batch Size = 32

All the hyperparameters chosen here are standard choices and there were no trials done on this. Here the only thing mattered was the model which has to architect properly.

Model:

The number of convolution stack and layers were decided from [4].

The model selection was greatly based on VGG 16 architecture where the basic outlines of VGG were considered while designing the model. The model has a symmetry which has proved to be a better model when compared to AlexNet and LeNet. The symmetry is based on number of kernels in each layer of VGG [6] where the kernels increases by a factor of 2 in each consecutive convolution layer. Thus in this model we started off with a kernel of 32 and doubled the number of kernel to 64 in the next convolution layer which is same as in VGG. The VGG has also the max pooling layer with strides and kernel size as (2, 2) which is kept exactly the same. The activation function used is relu same as VGG to provide non-linearity while predicting the class. The difference from that of VGG is the following: padding = 'valid', kernel size, strides and batch normalization.

The padding when kept as valid reduces the size of the image after convolution operation whereas when kept as same the size of image does not decrease. The main features which needs to be predicted largely lie in the middle of the image and not on the boundaries of image. Thus preserving the boundaries pixels by adding the padding makes no sense, thus it makes no sense to add padding here. Thus we have changed that to padding = 'valid', so that the number of trainable parameters decreases.

The next parameter that we changed when compared to VGG is the kernel size. The common choice of kernel in the first layer is 3 x 3 or 5 x 5. The fact that we chose 5 x 5 is that initially the model predicts higher level features and minute details are left. But in our image dataset there are no minute details we need to be captured as the dataset contains a very simple images which does not require minute details. Additionally, using higher kernel size reduces the number of parameters when padding is kept valid which is in our case so we decided to choose a kernel size of 5 x 5. As you can follow the AlexNet paper[7] which has 11 x 11 as its first kernel size this is again due to the fact the first layer need not capture complex functions.

The next we changed the strides as that of in VGG due to the following fact. The parameter needed to be changed because VGG 16 has actually 16 convolution stack but our model has only 2 stack thus changing parameters became important. The strides were made larger in the next convolution layer from (1, 1) to (2, 2) was due to the following facts: Smaller strides lead to large overlaps which means the Output Volume is high. Larger strides lead to lesser overlaps which means lower output volume. So these are the advantages of higher strides: 1) Lesser Memory needed for output. 2) Processing of output is easier as the volume is smaller. 3) The Output Spatial dimensions are smaller which when given as an input to the next layer simplify the model at least. 4) It avoids Overfitting especially in case of image processing having a large no. of attributes.

The next addition we did was batch normalization: This was just an additional layer and removal of it here does not really change anything. But in larger networks batch normalization helps in training the layers faster. Because it makes the mean and variance small of the values produced by the neurons. Thus learning features for next layer converge faster.

The number of neurons considered in the dense layer is the 1 / 4th of the parameters after flattening the images. The logic of choosing the 576 was same as that of doubling but here instead of starting from back we started from 2304 which is the number of parameters we obtained after flattening. We reduced it by factor 2 and then again by 2 so that the computation time and accuracy trade-off is accurate. In the results it can be seen that for each epoch the time taken is very less and it is due to the fact of considering the trade-off. The technique of doubling and halves proved to be beneficial in this report in the previous sections. Thus the same technique is applied here as well.

The following results were obtained from the training (Jupyter Notebook):

```
Train on 864 samples, validate on 216 samples Epoch 1/40 864/864
[==============================] - 2s 3ms/step - loss: 1.9029 - accuracy: 0.5336 - val_loss: 2.5406
- val_accuracy: 0.2130
Epoch 2/40 864/864
[==============================] - 1s 709us/step - loss: 0.5750 - accuracy: 0.8056 - val_loss:
2.4198 - val_accuracy: 0.2176
Epoch 3/40 864/864
[==============================] - 1s 702us/step - loss: 0.3627 - accuracy: 0.8912 - val_loss:
3.0847 - val_accuracy: 0.2917
Epoch 4/40 864/864
[==============================] - 1s 715us/step - loss: 0.2000 - accuracy: 0.9514 - val_loss:
2.9038 - val_accuracy: 0.3194
Epoch 5/40 864/864
[==============================] - 1s 703us/step - loss: 0.1384 - accuracy: 0.9699 - val_loss:
2.7738 - val_accuracy: 0.3287
Epoch 6/40 864/864
[==============================] - 1s 706us/step - loss: 0.0936 - accuracy: 0.9803 - val_loss:
2.4674 - val_accuracy: 0.1944
Epoch 7/40 864/864
```

```
[==============================] - 1s 708us/step - loss: 0.0605 - accuracy: 0.9931 - val_loss:
1.6870 - val_accuracy: 0.4120
Epoch 8/40 864/864
[==============================] - 1s 698us/step - loss: 0.0463 - accuracy: 0.9965 - val_loss:
0.5479 - val_accuracy: 0.7917
Epoch 9/40 864/864
[==============================] - 1s 707us/step - loss: 0.0384 - accuracy: 0.9988 - val_loss:
1.5183 - val_accuracy: 0.5926
……………………………………………………………………………
Epoch 40/40 864/864
[==============================] - 1s 704us/step - loss: 0.0128 - accuracy: 0.9953 - val_loss:
0.1384 - val_accuracy: 0.9444
120/120 [==============================] - 0s 492us/step

Loss = 0.12692996338009835
Test Accuracy = 0.9583333134651184
```

Validation Accuracy = 94.44%

Testing Accuracy = 95.83%

Conclusion: The results we got from the artificial neural network and convolutional are quite good. More training examples can be considered for more promising results. The convolutional neural network proved to be a better options with less number of parameters and more accuracy.

References:

[1] Khan, Rafiqul Zaman & Ibraheem, Noor. (2012). Hand Gesture Recognition: A Literature Review. International Journal of Artificial Intelligence & Applications (IJAIA). 3. 161-174. 10.5121/ijaia.2012.3412

[2] Kingma, Diederik & Ba, Jimmy. (2014). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.

[3] Bheda, Vivek & Radpour, Dianna. (2017). Using Deep Convolutional Networks for Gesture Recognition in American Sign Language.

[4] Yamashita, R., Nishio, M., Do, R.K.G. *et al.* Convolutional neural networks: an overview and application in radiology. *Insights Imaging* **9,** 611–629 (2018).

[5] https://www.sicara.ai/blog/2019-10-31-convolutional-layer-convolution-kernel

[6] Tammina, Srikanth. (2019). Transfer learning using VGG-16 with Deep Convolutional Neural Network for Classifying Images. International Journal of Scientific and Research Publications (IJSRP). 9. p9420. 10.29322/IJSRP.9.10.2019.p9420.

[7] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E. (2017-05-24). "ImageNet classification with deep convolutional neural networks" (PDF). Communications of the ACM. 60 (6): 84–90. doi: 10.1145/3065386. ISSN 0001-0782