Neural Networks and Deep Learning – Summer Term 2019

Team - 09	
Name: Rajib Chandra Das Matriculation Number: 1140657 Email: stu218517@mail.uni-kiel.de	Name: M M Mahmudul Hassan Matriculation Number: 1140658 Email: stu218518@mail.uni-kiel.de

Exercise sheet 4

Exercise 1 (Learning in neural networks):

a)

- **Loss Function:** Machines learn by means of a loss function. It's a method of evaluating how well specific algorithm models the given data. If predictions deviate too much from actual results, loss function would cough up a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction. Therefore, a loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. A loss function is used for parameter estimation, and the event in question is some function of the difference between estimated and true values for an instance of data.
- **Stochastic gradient descent:** It is an iterative method for optimizing an objective function with suitable smoothness properties. It is called stochastic because the method uses randomly selected samples to evaluate the gradients, hence SGD can be regarded as a stochastic approximation of gradient descent optimization. In gradient descent, a batch is the total number of examples we use to calculate the gradient in a single iteration. A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches. By choosing examples at random from our data set, we could estimate a big average from a much smaller one. Stochastic gradient descent takes the idea to the extreme-it uses only a single example per iteration. Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random.
- **Mini-batch:** It is actually a compromise between full-batch iteration and generalized SGD. Often regarded as small subset of the training data. The size of mini batch is chosen according to hardware resources and algorithmic requirements. A mini-batch is typically between 10 and 1000 examples chosen at random.
- **Regularization:** In mathematics, statistics, machine learning regularization is a process of introducing additional information in order to solve an ill posed problem or to prevent overfitting. The goal in machine learning is to minimize the generalization error. Regularization is a technique which makes slight modifications to the learning algorithm generalizes better. This in turn improves the model's performance on the unseen data as well.
- **Dropout:** Dropout is a regularization technique to reduce the risk of overfitting, mostly applied in fully connected layers of neural networks. This is one of the most interesting types of regularization techniques. It also produces very good results and is

consequently the most frequently used regularization technique in the field of deep learning. At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections. So each iteration has a different set of nodes and this result in a different set of outputs. It can also be thought of as an ensemble technique in machine learning. Ensemble models usually perform better than a single model as they capture more randomness. Similarly, dropout also performs better than a normal neural network model This probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. Dropout can be applied to both the hidden layers as well as the input layers. Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

- **Batch-Normalization:** We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers that are changing all the time, and get 10 ties or more improvement in the training speed. Batch normalization reduces the amount by what the hidden unit values shift around.
- **Learning with momentum:** Neural network momentum is a simple technique that often improves both training speed and accuracy. Training a neural network is the process of finding values for the weights and biases so that for a given set of input values, the computed output values closely match the known, correct, target values. Learning with momentum is a technique applied in gradient descent learning to improve convergence. For small learning rates, gradient descent based learning too large, the weight update may overshoot, leading to an oscillating loss function. In stochastic gradient descent the true gradient of the loss function is approximated by the average gradient calculated on a small mini-batch of training examples. Thus, the weight changes will not be perpendicular to the isocontours of the loss function, and take different directions at each weight update step. If the learning rate is small enough, this erratic behavior of the weight updates will still lead to the local minimum of the loss function. Learning with momentum is a compromise that smoothes the erratic behavior of the mini-batch updates, without slowing down the learning too much.
- **Data augmentation:** We do augmentation before we feed the data to the model. We have two options. One option is to perform all the necessary transformations beforehand, essentially increasing the size of our dataset. The other option is to perform these transformations on a mini-batch, just before feeding it to our machine learning model. The first option is known as offline augmentation. This method is preferred for relatively smaller datasets, as we would end up increasing the size of the dataset by a factor equal to the number of transformations we perform. The second option is known as online augmentation, or augmentation on the fly. This method is preferred for larger datasets, as we can't afford the explosive increase in size. Instead, we would perform transformations on the mini-batches that we would feed to our model. Some machine learning frameworks have support for online augmentation, which can be accelerated on the GPU. For example, if we have training dataset so called list of images we can perform following transformations to increase the size of our datasets:
    - Flip,
    - Rotation,
    - Scale,
    - Crop,
    - Translation and
    - Gaussian Noise.

- **Unsupervised pre training/supervised fine-tuning:** Training deep feed-forward neural networks can be difficult because of local optima in the objective function and because complex models are prone to overfitting. Unsupervised pre-training initializes as discriminative neural net from one which was trained using an unsupervised criterion, such as a deep belief network or a deep autoencoder. This method can sometimes help with both the optimization and the overfitting issues.
- **Deep learning:** Deep learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. Deep learning use multiple layers to progressively extract higher level features from raw input. For example, in image processing, lower layers may identify edges, while higher layer may identify human meaningful items such as digits/letters or faces.

b) Most import activation Function:

i. Binary Step Function:
   a. Mathematical Equation: f(x) = 0 if x>= 0
   b. Use case: It is used while creating a binary classifier.

ii. Linear Function:
   a. Mathematical Equation: f(x) = ax
   b. Use case: It is used for Regression related tasks.

iii. Sigmoid Function:
   a. Mathematical Equation: f(x)=1/(1+e^-x)
   b. Use case: classify the values to particular classes.

iv. Tanh:
   a. Mathematical Equation: tanh(x)=2/(1+e^(-2x)) -1
   b. Use case: Classification.

v. ReLU:
   a. Mathematical Equation: f(x)=max(0,x)
   b. Use case: Regression, but Non-Negative.

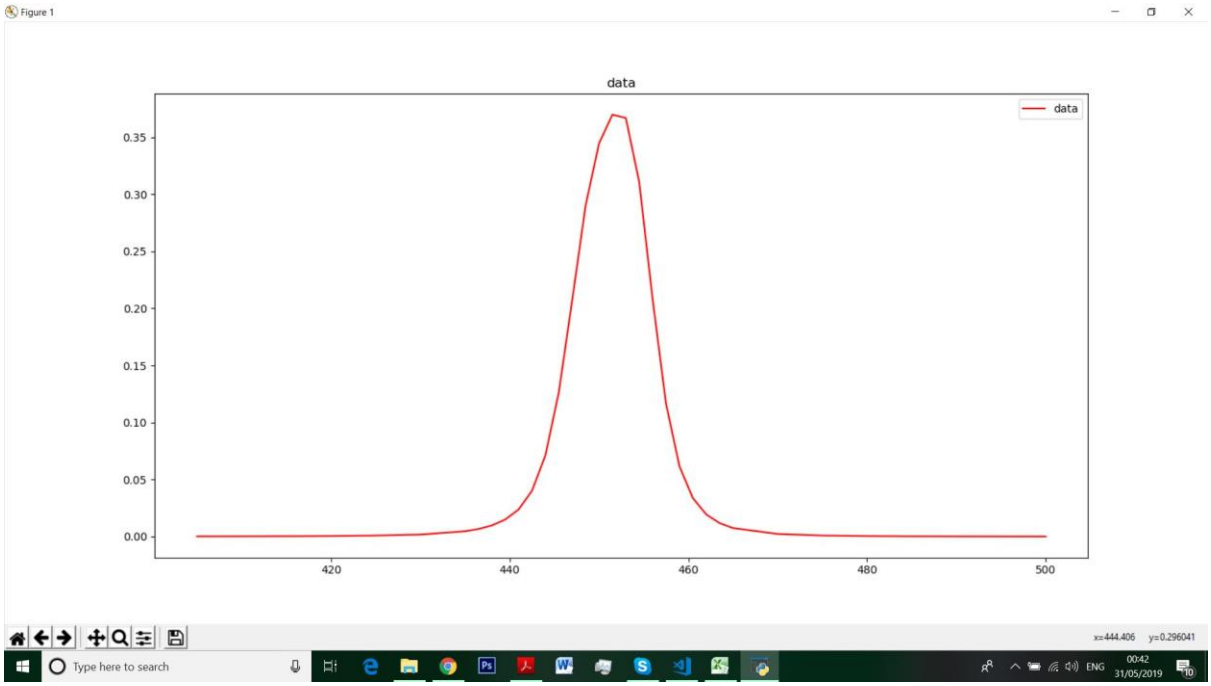
vi. Softmax:
   a. Mathematical Equation:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

   b. Use case: Classification.

Exercise 2 (Multi-layer perceptron – regression problem):

After plotting Eckerle 4 Dataset we have following graph:



First Fix: We need to set number of hidden layers to solve this nonlinear regression problem.

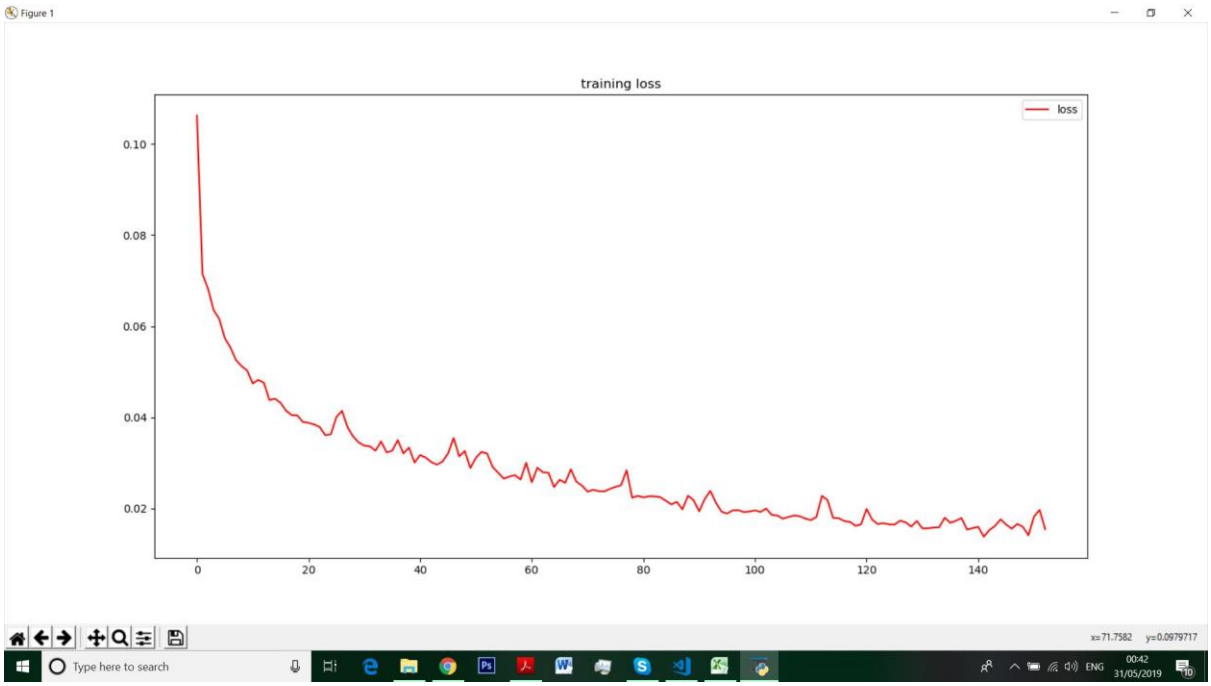
```
    hidden_layer_size = (128, 64, 64)
```

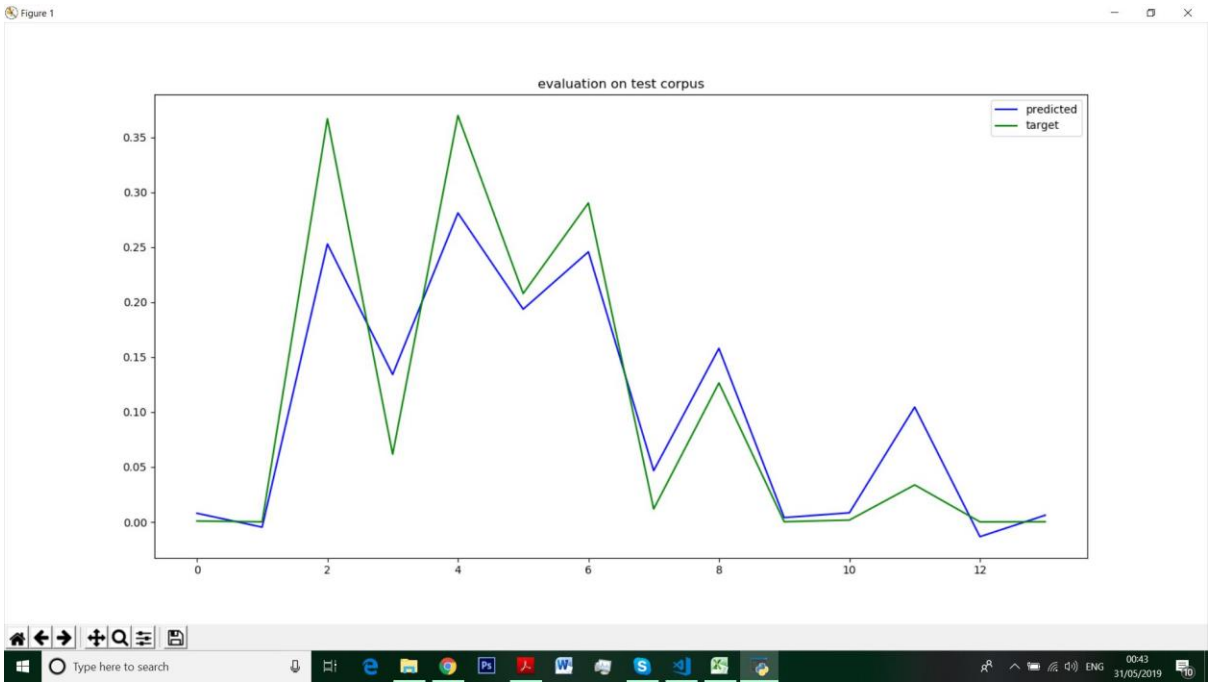
- We are modeling a network for multilayer perceptron.
- It has a input layer , an output layer and 3 hidden layers.
- First, Second and Third hidden layer contains 128, 64, 64 neurons respectively.

Second Fix: We need to set the parameter for **MLPRegressor.**

```
    mlp = MLPRegressor(hidden_layer_sizes=hidden_layer_size,
                       activation='relu',
                       solver='adam',
                       learning_rate='adaptive',
                       max_iter=256,
                       learning_rate_init=0.001,
                       alpha=0.001,
                       batch_size=2)
```

- We are using activation function ReLU. ReLU is the most commonly used activation function in neural networks. ReLU stands for rectified linear unit. Mathematically, it is defined as $y = \max(0,x)$.
- We are using 'adam' as solver.
- We are setting our initial learning rate to 0.001 and regularization coefficient to 0.001.





We have obtained the output using those fix.

```
➜  files python.exe exercise2.py
number of network inputs: 1
number of network outputs: 1
number of network layers: 5
Training set score: 0.472071
Training set loss: 0.032697
Test set score: 0.392114
```

After analyzing a lot of combinations of the setting this is our best solution so far. It's true that there exist other combinations which the error rate can be improved.

Reference: http://gonzalopla.com/deep-learning-nonlinear-regression/

Exercise 2 (Parameters of a multi-layer perceptron – digit recognition):

a) **Modifying Learning rate:** We have following configuration for modifying learning rates.

```
numEpochs = 30
batchSize = 10
learningRates = [0.001, 0.002, 0.003, 0.004, 0.005, 0.01, 0.05, 0.1, 0.5, 1,
5, 10, 15]
numRepetitions = 4
hiddenLayerSizes = (100)
```

```
net = nn.MLPClassifier(  hidden_layer_sizes=hiddenLayerSizes,
                         activation='logistic',
                         solver='sgd',
                         alpha=0.0,
                         batch_size=batchSize,
                         learning_rate='constant',
                         learning_rate_init=learningRate,
                         max_iter=numEpochs, shuffle=False )
```

So, we are trying to modify learning rate in a neural network with 1 hidden layer consisting 100 neurons. It will try 4 iterations using each learning rates and run for 30 epochs.

Learning rate	Training accuracy	Validation accuracy	Test accuracy	Mean Training Accuracy	Mean Validation accuracy	Mean Test Accuracy
0.001000	0.960200 +/- 0.000548	0.960650 +/- 0.000229	(0.956125 +/- 0.001252	0.9602	0.96065	0.956125
0.002000	0.977535 +/- 0.000172	0.971200 +/- 0.000354	0.968575 +/- 0.001130	0.977535	0.9712	0.968575
0.003000	0.985030 +/- 0.000397	0.974250 +/- 0.001004	0.972850 +/- 0.000890	0.98503	0.97425	0.97285
0.004000	0.990045 +/- 0.000331	0.975925 +/- 0.000955	0.975500 +/- 0.001134	0.990045	0.975925	0.9755
0.005000	0.993180 +/- 0.000272	0.976450 +/- 0.000808	0.976350 +/- 0.000942	0.99318	0.97645	0.97635
0.010000	0.998990 +/- 0.000171	0.977550 +/- 0.000743	0.977500 +/- 0.000187	0.99899	0.97755	0.9775
0.050000	0.999990 +/- 0.000010	0.977700 +/- 0.000704	0.978975 +/- 0.000687	0.99999	0.9777	0.978975

0.100000	1.000000 +/- 0.000000	0.976825 +/- 0.001672	0.978250 +/- 0.001146	1	0.976825	0.97825
0.500000	0.924880 +/- 0.013709	0.924475 +/- 0.010511	0.919825 +/- 0.011458	0.92488	0.924475	0.919825
1.000000	0.859745 +/- 0.026271	0.867700 +/- 0.026893	0.862275 +/- 0.026091	0.859745	0.8677	0.862275
5.000000	0.096840 +/- 0.000000	0.100900 +/- 0.000000	0.097400 +/- 0.000000	0.09684	0.1009	0.0974
10.000000	0.096870 +/- 0.003908	0.097150 +/- 0.003262	0.098400 +/- 0.005720	0.09687	0.09715	0.0984
15.000000	0.098820 +/- 0.000312	0.099075 +/- 0.000043	0.099300 +/- 0.002252	0.09882	0.099075	0.0993

After running all those learning rates we have got following graphs:

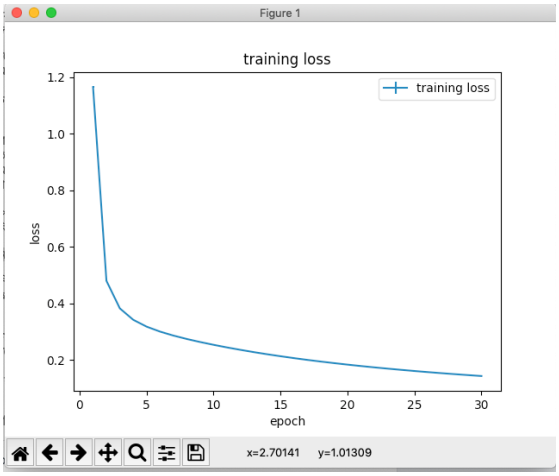


Fig 1: Learning rate =0.001

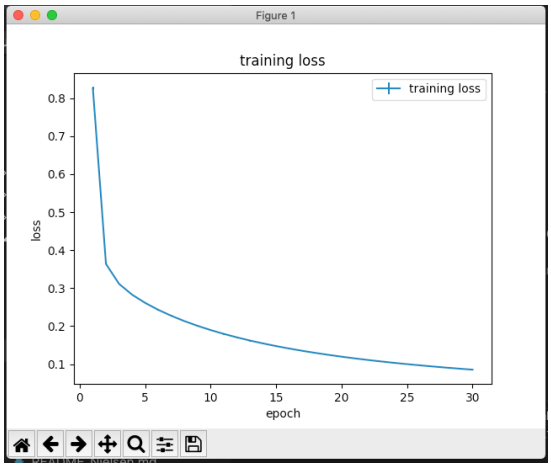


Fig 2: Learning rate =0.002

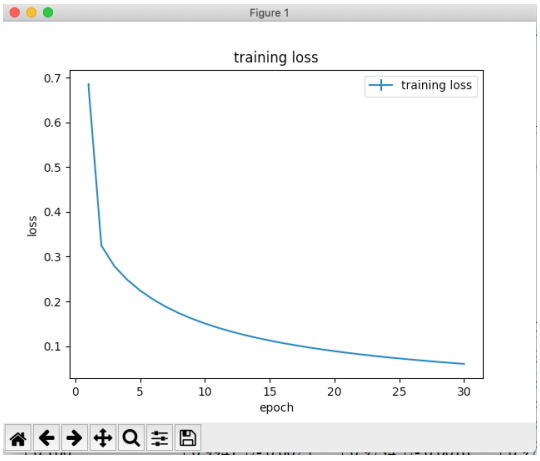


Fig 3: Learning rate =0.003

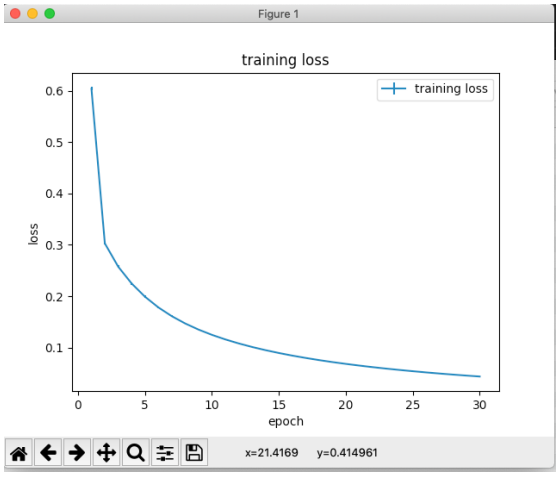


Fig 2: Learning rate =0.004

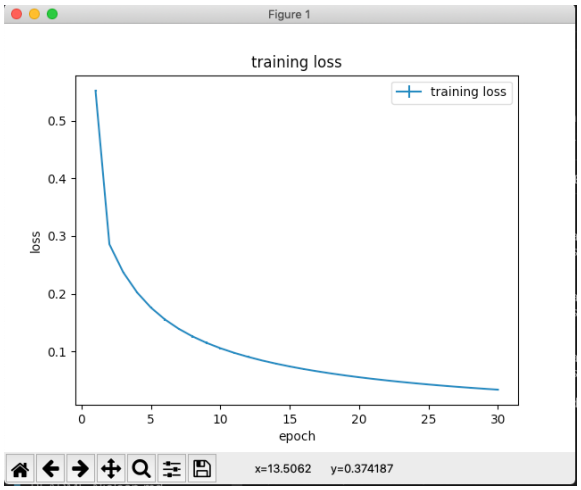


Fig 5: Learning rate =0.005

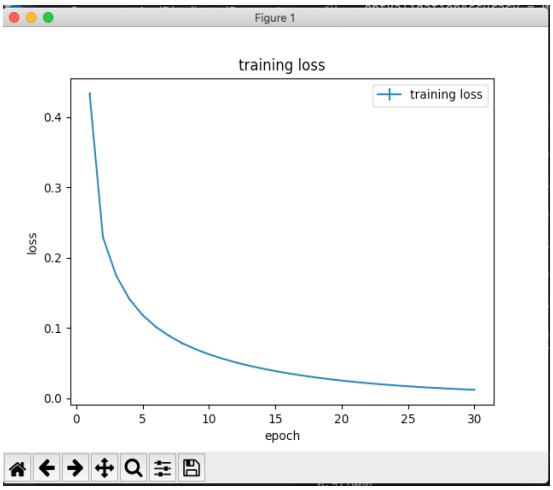


Fig 6: Learning rate =0.01

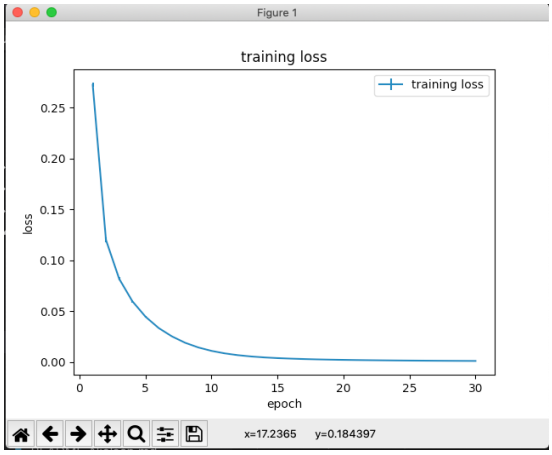


Fig 7: Learning rate =0.05

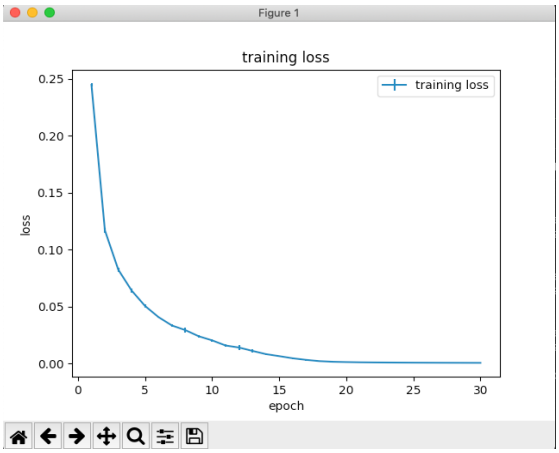


Fig 8: Learning rate =0.1

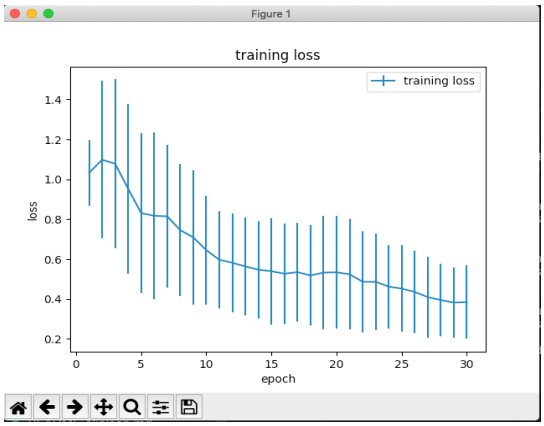


Fig 9: Learning rate =0.5

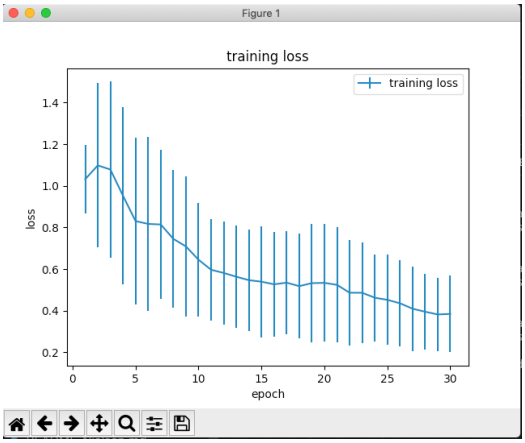


Fig 10: Learning rate =1

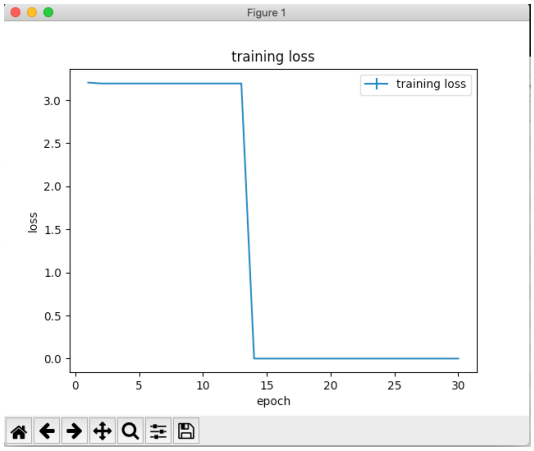


Fig 11: Learning rate =5

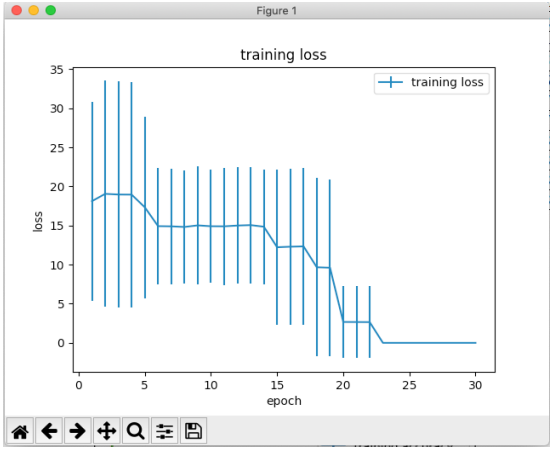
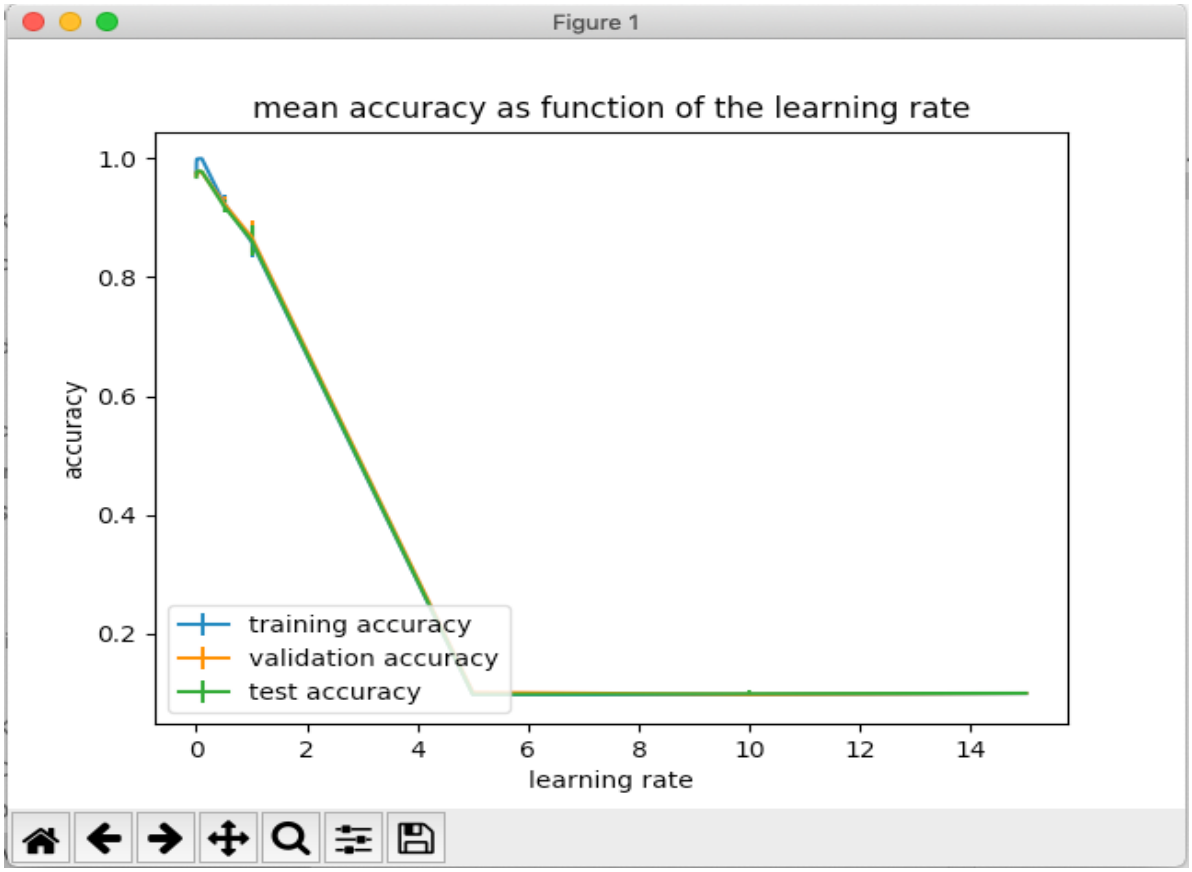


Fig 12: Learning rate =10



Findings: So far we have found that, learning rate 0.05 is the best setting for this problem. For learning rate lower than 0.05 the convergence is slow. And for other learning rates which are greater than 0.05 , the convergence is not stable. So we can assume that we can set 0.05 learning rate to continue our next analysis.

b) Modifying number of neurons: We are setting following parameters to analyze this problem:
   - Learning rate = 0.05
   - Epochs = 30
   - For each layer number of iteration = 4
   - Number of neurons: [5, 25, 50, 75, 100, 150, 200, 500]

Number of Neurons	Training accuracy	Validation accuracy	Test accuracy
5	0.874950 +/- 0.013722	0.869350 +/- 0.013658	0.866050 +/- 0.015014
25	0.973485 +/- 0.003486	0.951325 +/- 0.000545	0.948550 +/- 0.001812
50	0.998870 +/- 0.000266	0.971350 +/- 0.001464	0.970175 +/- 0.000763
75	0.999960 +/- 0.000024	0.976175 +/- 0.001599	0.974400 +/- 0.000628
100	0.999980 +/- 0.000014	0.978400 +/- 0.000458	0.977950 +/- 0.000658
150	1.000000 +/- 0.000000	0.981150 +/- 0.000832	0.980250 +/- 0.000730
200	1.000000 +/- 0.000000	0.981575 +/- 0.000626	0.981375 +/- 0.001132
500	1.000000 +/- 0.000000	0.982300 +/- 0.000381	0.981650 +/- 0.000885

We have found following graphs for this setting:

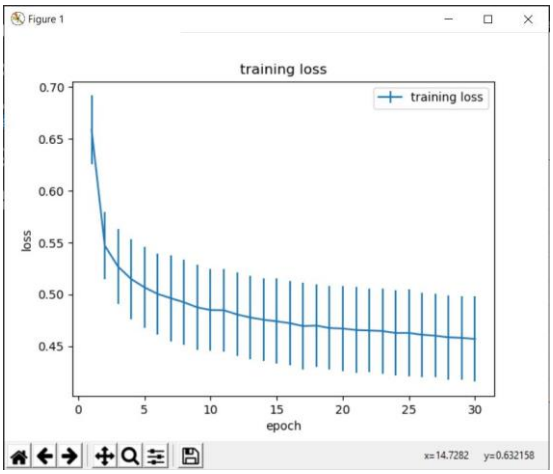


Fig1: Number of neurons = 5

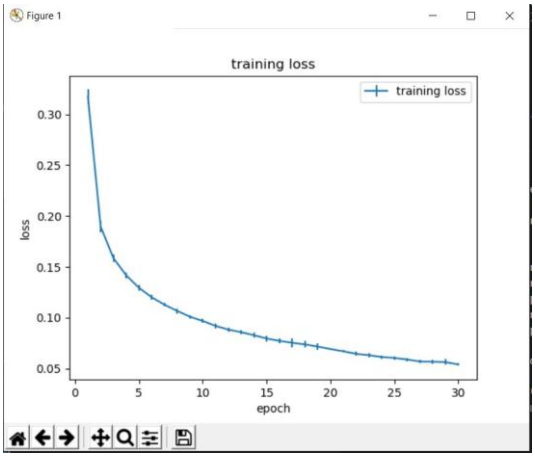


Fig2: Number neurons = 25

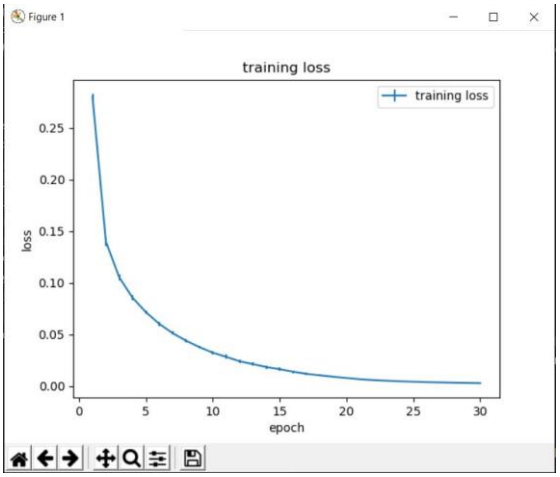


Fig 3: Number of neurons = 50

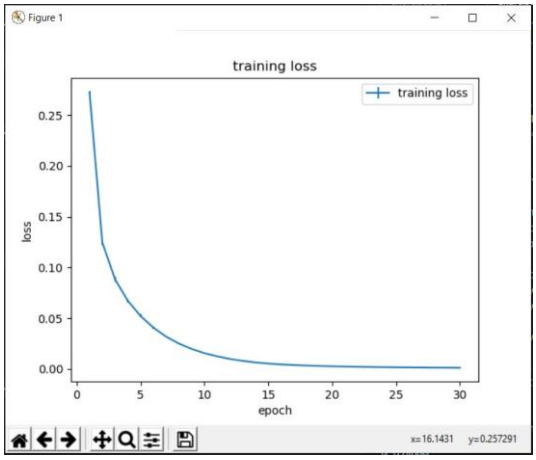


Fig 4: Number neurons = 75

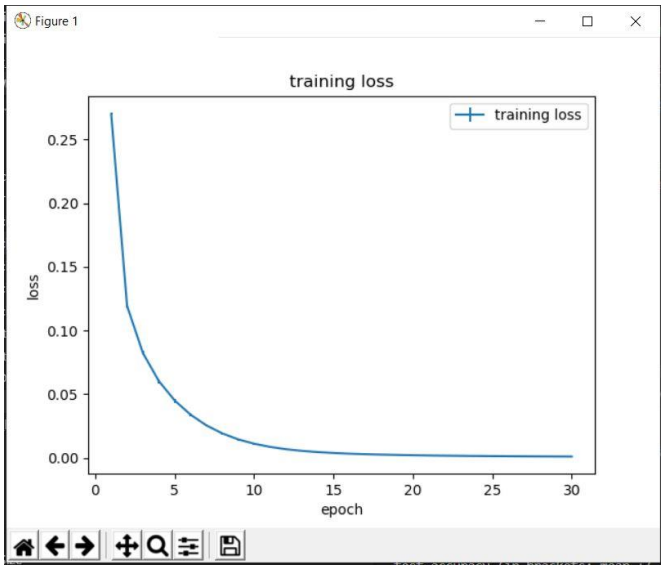


Fig 5: Number of neurons = 100

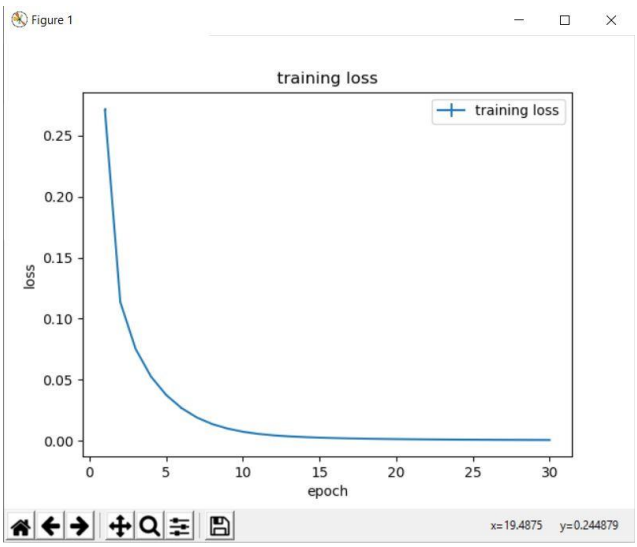


Fig 6: Number neurons = 150

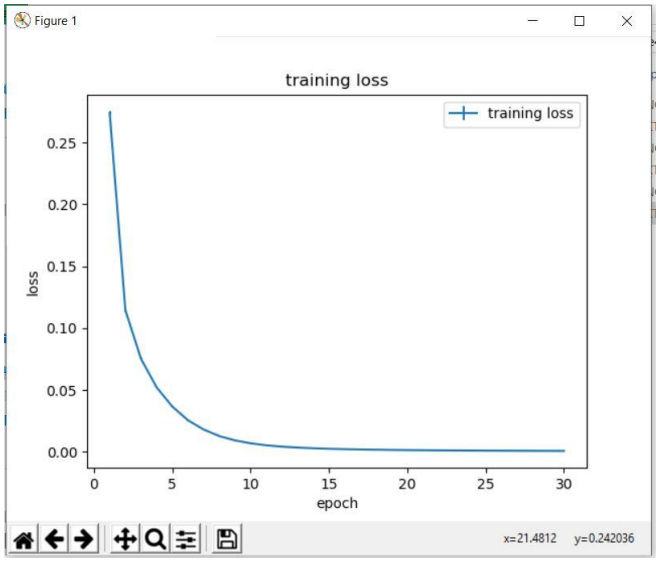


Fig 7: Number of neurons = 200

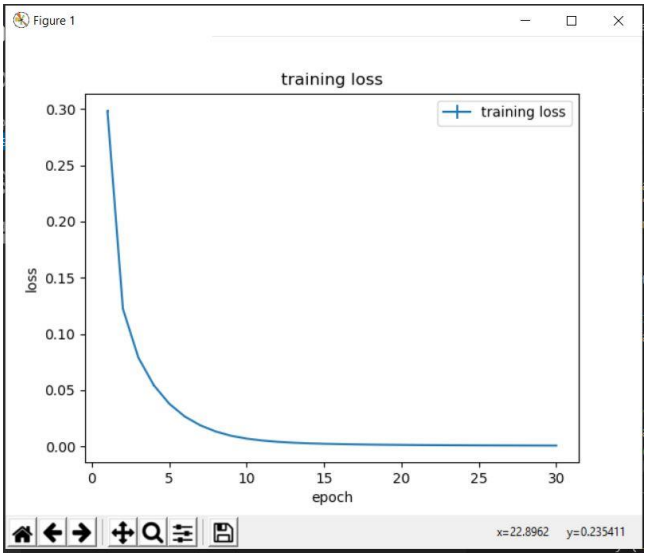
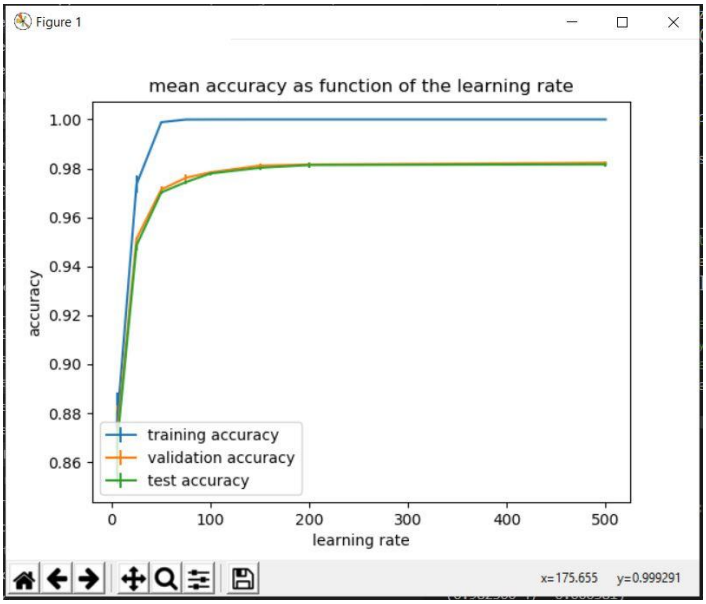


Fig 8: Number neurons = 500



Findings: After getting all those statistics we have realized that, if we use 200 or more neurons, then we will get most accurate results. If we increase more neurons then we don't find noticeable improvements. But if we increase number of neurons then it takes much time to calculate the results. So we are choosing 100 neurons to continue our next analysis.

c) **Modifying number of layers:** We are setting following parameters to analyze this problem:

Learning rate = 0.05
Number of neurons per hidden layer: 100
Epochs = 30
For each layer number of iteration = 2

Number of hidden layer	Training accuracy	Validation accuracy	Test accuracy
1	0.999990 +/- 0.000010	0.978450 +/- 0.000850	0.978600 +/- 0.000900
2	1.000000 +/- 0.000000	0.979350 +/- 0.000150	0.978850 +/- 0.000050
3	0.995160 +/- 0.000340	0.972700 +/- 0.000800	0.973200 +/- 0.000100

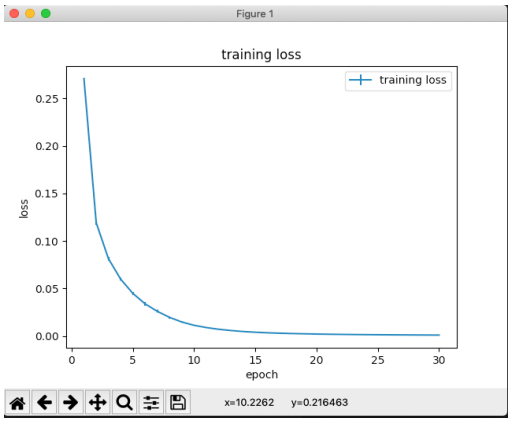


Fig: 1 Number of hidden layer 1

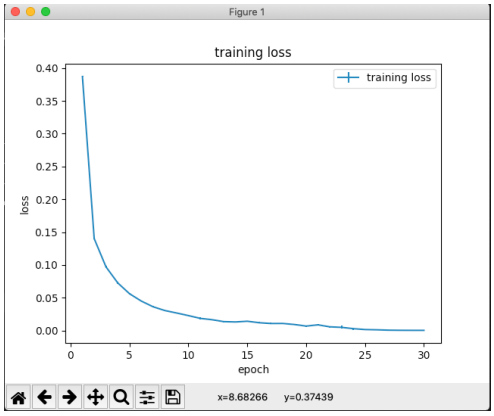


Fig: 2 Number of hidden layer 2

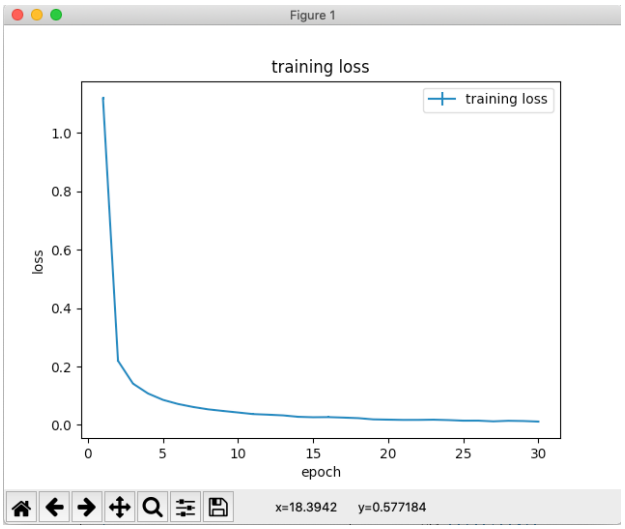


Fig: 3 Number of hidden layer 3

Findings: After analysis we are sure that, one hidden layer is the best option to solve this problem. Increasing number of hidden layers doesn't help us.

D) Modifying Solver technique:

Learning rate = 0.05
Number of neurons per hidden layer: 100
Epochs = 30
For each layer number of iteration = 4
Solver technique: sgd/adam

Solver	Training accuracy	Validation accuracy	Test accuracy
sgd	0.999980 +/- 0.000014	0.979425 +/- 0.000606	0.978200 +/- 0.000644
adam	0.924455 +/- 0.003378	0.927725 +/- 0.003159	0.920075 +/- 0.003407

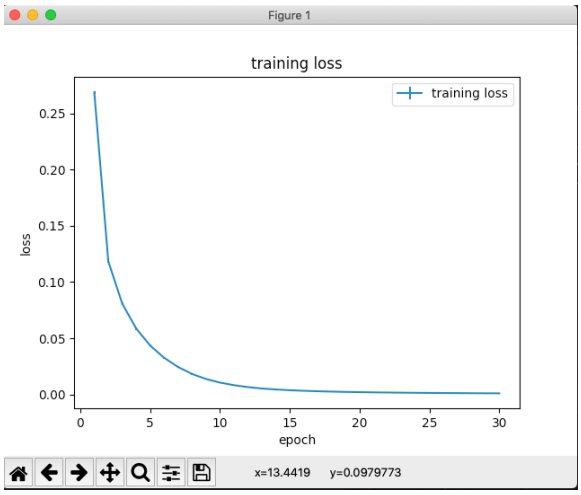


Fig : 1 Solver Technique 'sgd'

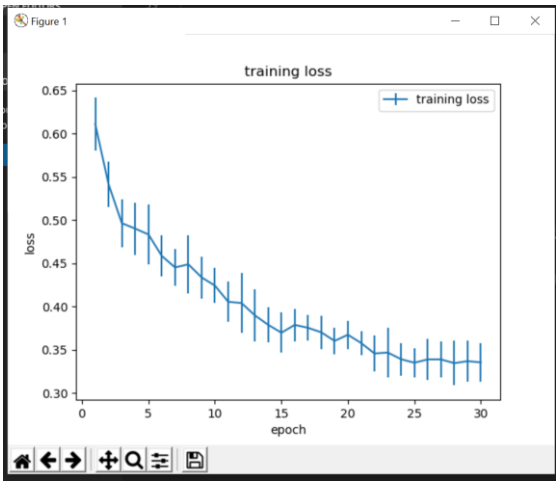


Fig : 2 Solver Technique 'adam'

Findings: After analysis we are sure that, stochastic gradient descent is the best option for this problem. So we can continue our next analysis using 'sgd' as solver technique.

E) Modifying Activation Function:

Learning rate = 0.05
Number of neurons per hidden layer: 100
Epochs = 30
For each layer number of iteration = 1
Solver technique: sgd
Activation Function: relu / logistic/ tanh

Solver Technique	Activation function	Training accuracy (mean +/- std)	Validation accuracy (mean +/- std)	Test accuracy (mean +/- std)
sgd	Logistic	0.999980	0.978300	0.978800
	Tanh	0.967280	0.960200	0.955000
	relu	0.976880	0.961600	0.954100

We have found following graphs:

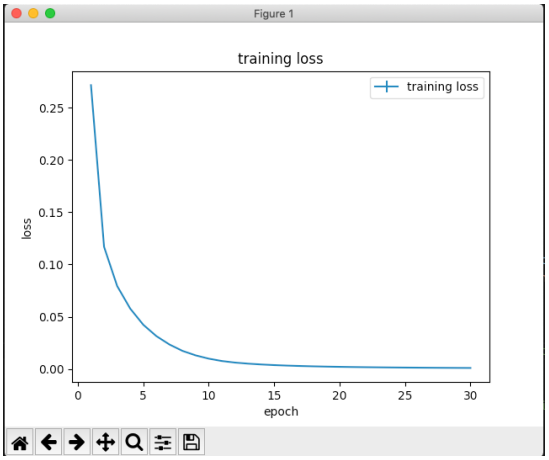


Fig 1: 'logistic' as activation function

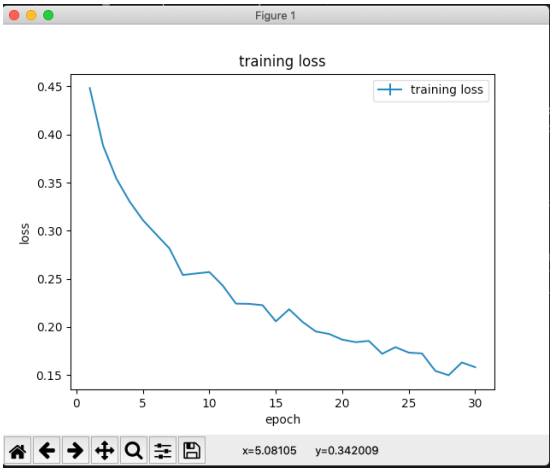


Fig 2: 'tanh' as activation function

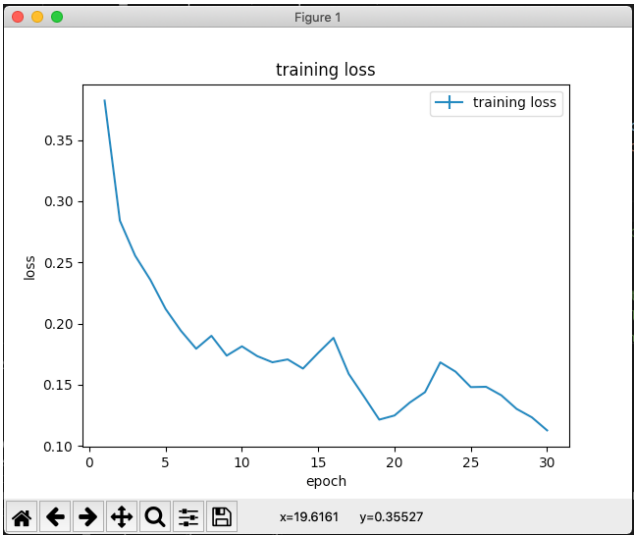


Fig 3: 'relu' as activation function

Findings: After analysis we have found that, 'logistic' activation function gives us the fastest performance. If we set 'relu' and 'tanh' as activation function then we don't get stable convergence.