Introduction

In this study, a two-layer artificial neural network is trained on 'Digits Dataset' from sklearn. The neural network is defined and trained using two methodologies –

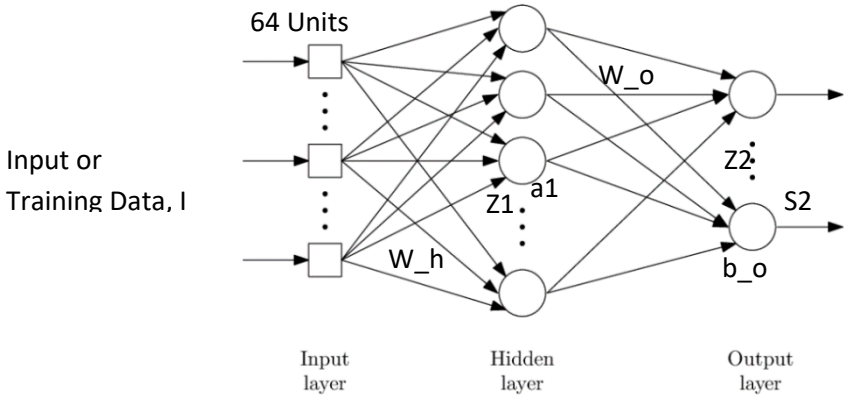
(i) Using Numpy where the gradients are calculated analytically and implemented using Python's Numpy module, and weight updates are also done by implementing gradient descent using Numpy.

(ii) Using Pytorch, where the neural network is defined in Pytorch itself and the gradient calculation as well the weight updates are done using Pytorch.

The accuracy, training and testing losses vs epoch results are plotted for both the methodologies, and computation time for both cases is also noted.

Methodology - Numpy

The experimental setup for this study was implemented in Python. The dataset used was 'Digits Dataset' from 'sklearn.datasets' library. This dataset consists of 8X8 images of handwritten digits corresponding to 10 classes, i.e., each image which belongs to a digit from 0 to 9 consists of 64 features. The network that we will be implementing has – 1 input layer, 1 hidden layer with hyperbolic tangent activation function, and 1 output layer with softmax, as shown below,



Here, input layer units are assigned values corresponding to each feature value of the image of a digit. If the number of hidden units are 1000 (say), then,
(i) W_h will be a 1000X64 matrix of hidden layer weights.
(ii) b_h will be a vector of 1000 components representing bias for hidden layer.
(iii) Z1 will be a vector of 1000 components corresponding to each data point (or image).
(iv) a1 is tanh activation of Z1. Therefore, it will be a vector of 1000 components corresponding to each data point.
(v) W_o will be a 10X1000 matrix of outer layer weights.
(vi) b_o will be a vector of 10 components representing bias for outer layer.
(vii) Z2 will be a vector of 10 components corresponding to each data point.
(viii) S2 will be a vector of 10 components corresponding to each data point, which is softmax applied to Z2.
(ix) I is a vector of 64 components representing one data point or image.

It is important to note the equations which define each layer, and which result in a prediction corresponding to each data point. These are called feed forward equations as in this case, information is flowing from input to output in a unidirectional way. These equations are as below,

i. Hidden layer:
For n^{th} data point,

$$Z1_n = W\_h\, I_n + b\_h$$
$$a1_n = tanh(Z1_n)$$

ii. Outer layer:
For n^{th} data point,

$$Z2_n = W\_o\ a1_n + b\_o$$
$$S2_n = softmax(Z2_n)$$

Where softmax function is defined as,

$$softmax(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}\ for\ j = 1, 2, \ldots K\ , for\ K\ classes$$

It should be noted that the output of the softmax function can be treated as probabilities corresponding to each class. From these probabilities, we can construct a one hot encoded vector for 1 of K classes, representing prediction for that data point.
The python code snippets for feed forward equations and for softmax function are as follows,

```
# Equation of Layer
def compute(weights,input,bias):
  # We are calculating these vectors for the whole dataset
  # Not for one data point
  c1=np.dot(weights, input.T)
  c2=np.tile(bias,(input.shape[0],1)).T
  z=c1+c2
  z=z.T
  return z
```

```
# activation
tanh = lambda x : np.tanh(x)
```

```
# Feed Forward
def feed_forward(inputs,W_h,W_o,b_h,b_o):
  z1=compute(W_h,inputs,b_h)
  a1=tanh(z1)
  z2=compute(W_o,a1,b_o)
  a2=softmax(z2)
  return z1,a1,z2,a2
```

```
# defining softmax function
# Here, we are performing softmax operation on entire dataset
# Using vectorized coding
def softmax(z):
  exp_mat=np.exp(z)
  sum_exp=np.sum(exp_mat,axis=1)
  sftmax=exp_mat.T/sum_exp
  return sftmax.T
```

There can be different approaches to define the loss function for a neural network depending upon the type and context of the problem being modelled. For a multi-class classification task such as the one in this report, a Cross Entropy Loss function is very useful and strongly recommended, because this function can take probabilities as input and return high loss values when low probabilities are predicted for correct classes. This function is defined as follows,

$$\mathcal{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{k,n}\ln\left(S_{k,n}\right)$$

Where,

$t_{k,n}$: Either 1 or 0. It is 1 if k^{th} class for n^{th} data point is the true class, otherwise 0.

$S_{k,n}$: Predicted probability for k^{th} class for n^{th} data point

N: Total number of observations or data points in training set;

K : Total number of classes. Its value is 10 corresponding to digits from 0 to 9

Python implementation for the above loss function is,

```
# Function to Calculate Average CE Loss
def averageCE(pred_mat,train_1hot):
  log_score=np.log(pred_mat)
  avg_CE=log_score*train_1hot
  avg_CE=-(np.sum(avg_CE))/(train_1hot.shape[0])
  return avg_CE
```

In order to train our neural network, we need to implement Backpropagation Algorithm. Using, chain rule, we can calculate the gradient of the loss function with respect to all unit elements, weights and biases that are present in the network architecture.

First, although the gradient of the loss function with respect to the softmax of the outer layer predictions can be calculated, but we can directly calculate the gradient of the loss function with respect to outer layer predictions (before softmax) by first substituting the value of outer layer predictions in the loss function. Thus, the loss function can now be written as,

$$\mathcal{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{10} t_{k,n}\ln\left(\frac{e^{z_{k,n}}}{\sum_{j=1}^{10} e^{z_{j,n}}}\right)$$

The gradient of the loss function with respect to outer layer units (Z2) is given as,

$$\frac{d\mathcal{L}}{d(\mathbf{Z2})_{\mathbf{n,i}}} = S_{i,n} - t_{i,n} \qquad \text{i= 1, 2, . . . 10}$$

Where, i represents the i^{th} class out of 10 classes, and n represents the n^{th} data point. Therefore, the above gradient is a N X 10 matrix which is calculated in python as below,

```
# Defining function which gives gradient wrt to outer layer units (Z2)

def gradCE_out(pred_mat,train_1hot):
  grad= pred_mat - train_1hot

  # Here pred_mat is N X 10 matrix of softmax of Z2
  # train_1hot is N X 10 matrix of hot encoded training targets
  return grad
```

Now, the gradient of the loss function with respect to outer layer bias can be derived using chain rule, and is given as,

$$\frac{d\mathcal{L}}{db\_o} = \frac{1}{N}\sum_{n=1}^{N} S_{i,n} - t_{i,n} \qquad \text{i= 1, 2, . . . 10}$$

This gradient is a 10 X 1 vector.

which in Python is calculated as,

```
# Gradient of loss fn. with respect to outer layer bias

def grad_bo(pred_mat,train_1hot):
  grad=np.sum(gradCE_out(pred_mat,train_1hot),axis=0)/(train_1hot.shape[0])
  return grad
```

Next, the gradient of the loss function with respect to outer layer weights can be written as,

$$\frac{d\mathcal{L}}{dW\_o} = \frac{1}{N}\sum_{n=1}^{N} \frac{d\mathcal{L}}{d(Z2)_{\boldsymbol{n}}} \; [a1_n{}^T]$$

Where $\frac{d\mathcal{L}}{d(Z2)_{\boldsymbol{n}}}$ is a column vector (or jacobian) of 10 components i.e. a 10 X 1 vector

and $a1_n{}^T$ is a row vector of 1000 components i.e. a 1 X 1000 matrix
Therefore, this gradient is a 10 X 1000 matrix and is implemented in python as below,

```
# Gradient of loss fn. with respect to outer layer weights

def grad_Wo(pred_mat,train_1hot,a1):
  d1=gradCE_out(pred_mat,train_1hot)
  grad=(d1.T)@(a1)/(train_1hot.shape[0])
  return grad
```

Gradient of the loss function with respect to hidden layer activations, $a1$ is given by,

$$\frac{d\mathcal{L}}{d(a1)_{\mathrm{n,p}}} = \frac{d\mathcal{L}}{d(Z2)_{\boldsymbol{n},i}} \; W\_o$$

Where $\frac{d\mathcal{L}}{d(Z2)_{\boldsymbol{n},i}}$ as noted before, is a N X 10 matrix.

Therefore, the gradient $\frac{d\mathcal{L}}{d(a1)_{\mathrm{n,p}}}$ is a N X 1000 matrix.

Therefore, the gradient of the loss function with respect to hidden layer units, Z1 can be written as,

$$\frac{d\mathcal{L}}{d(Z1)_{\mathrm{n,p}}} = \frac{d\mathcal{L}}{d(Z2)_{\boldsymbol{n},i}} \; W\_o \; * \; \mathrm{sech}^2\big((Z1)_{\mathrm{n,p}}\big)$$

Where * denotes element-wise multiplication.

This gradient is also a N X 1000 matrix.

These are implemented in python as follows,

```
def d_tanh(x):
  d_=(((np.cosh(x))**(-2)))
  return d_
```

```
# Gradient with respect to z1

def grad_z1(pred_mat,train_1hot,W_o,z1):
  grad=(gradCE_out(pred_mat,train_1hot)@W_o)*d_tanh(z1)
  return grad
```

Now, the gradient of the loss function with respect to hidden layer bias b_h, is given by

$$\frac{d\mathcal{L}}{db\_h} = \frac{1}{N}\sum_{n=1}^{N} \frac{d\mathcal{L}}{d(Z1)_{\mathrm{n,p}}}$$

This gradient is a 1000 X 1 vector, which in Python is calculated as,

```
# Gradient of loss fn. with respect to hidden layer bias

def grad_bh(pred_mat,train_1hot,W_o,z1):
  dz1=grad_z1(pred_mat,train_1hot,W_o,z1)
  grad=np.sum((dz1),axis=0)/(train_1hot.shape[0])
  return grad
```

Lastly, the gradient of the loss function with respect to hidden layer weights can be written as,

$$\frac{d\mathcal{L}}{dW\_h} = \frac{1}{N}\sum_{n=1}^{N} \frac{d\mathcal{L}}{d(Z1)_{\boldsymbol{n}}} \; [{I_n}^T]$$

Where $\frac{d\mathcal{L}}{d(Z1)_{\boldsymbol{n}}}$ is a column vector of 1000 components i.e. a 1000 X 1 vector

and ${I_n}^T$ is a row vector of 64 components i.e. a 1 X 64 matrix. Therefore, this gradient is a 1000 X 64 matrix and is implemented in python as below,

```
# Gradient of loss fn. with respect to hidden layer weights

def grad_Wh(pred_mat,train_1hot,W_o,z1,input):
  dz1=grad_z1(pred_mat,train_1hot,W_o,z1)
  grad=(dz1.T)@(input)/(train_1hot.shape[0])
  return grad
```

Additionally, we use a function to clip the values of elements in matrices and vectors of gradients. This function is as follows,

```
# Grad Clip Function

def gradClip(x):
  grad=np.clip(x,-1,1)
  return grad
```

Now that we have all the required gradients, we make use of a function to implement gradient descent with momentum algorithm to train our neural network using training data.

```
# Defining a function which takes number of hidden units,
# and initializes the network with random weights
def Weights(hidden_units,gaus=np.random.normal):
  W_h = gaus(0,1,(hidden_units, 64))
  W_o = gaus(0,1,(10, hidden_units))
  b_h = gaus(0,1,(hidden_units))
  b_o = gaus(0,1,(10))
  return W_h,W_o,b_h,b_o
```

```
# Training the network
def network_train(train_data,train_1hot,test_data,test_1hot,test_targets,
                  hidden_units,epochs,learning_rate,momentum):
  W_h,W_o,b_h,b_o=Weights(hidden_units)
  W_hv,W_ov,b_hv,b_ov=W_h*(1e-5),W_o*(1e-5),b_h*(1e-5),b_o*(1e-5)

  learning_rate_i=learning_rate

  losses_train=np.zeros(epochs)
  losses_test=np.zeros(epochs)
  accuracy_test=np.zeros(epochs)

  for i in range(0,epochs):

    z1,a1,z2,pm_train=feed_forward(train_data,W_h,W_o,b_h,b_o)
    _,__,___,pm_test=feed_forward(test_data,W_h,W_o,b_h,b_o)

    pred_test=np.array([np.argmax(pm_test[i]) for i in range(0,len(pm_test))])
    accuracy_test[i]=(accuracy_score(test_targets,pred_test))*100

    losses_train[i]=averageCE(pm_train,train_1hot)
    losses_test[i]=averageCE(pm_test,test_1hot)

    # This neural network is prone to overflow
    # Therefore, we need to stop training if overflow occurs
    if np.max(z2)>100:
      losses_train[i+1:]=np.nan
      losses_test[i+1:]=np.nan
      accuracy_test[i+1:]=np.nan
      return losses_train,losses_test,accuracy_test,W_h,W_o,b_h,b_o

    b_hv=momentum*b_hv+learning_rate*gradClip(grad_bh(pm_train,train_1hot,W_o,z1))
    b_h=b_h-b_hv

    W_hv=momentum*W_hv+learning_rate*gradClip(grad_Wh(pm_train,train_1hot,W_o,z1,train_data))
    W_h=W_h-W_hv

    b_ov=momentum*b_ov+learning_rate*gradClip(grad_bo(pm_train,train_1hot))
    b_o=b_o-b_ov

    W_ov=momentum*W_ov+learning_rate*gradClip(grad_Wo(pm_train,train_1hot,a1))
    W_o=W_o-W_ov

  return losses_train,losses_test,accuracy_test,W_h,W_o,b_h,b_o
```

Methodology - Pytorch

The neural network is defined and trained using Pytorch as below,

```
# Defining function which trains the neural network using Pytorch
def nnet_train_py(hidden_units,epochs,learning_rate,momentum_,
                  train_data,train_1hot,test_data,test_1hot,test_targets):

  class nnet(nn.Module):

    def __init__(self):
        super(nnet, self).__init__()
        self.layer1 = nn.Linear(train_data.shape[1], hidden_units)
        self.layer2 = nn.Linear(hidden_units, 10)

    def forward(self, img_data):
        z1 = self.layer1(img_data)
        a1 = torch.tanh(z1)
        z2 = self.layer2(a1)
        return z2

  NNET=nnet()
  criterion = nn.CrossEntropyLoss()
  optimizer = optim.SGD(NNET.parameters(), lr=learning_rate, momentum=momentum_)

  losses_train=np.zeros(epochs)
  losses_test=np.zeros(epochs)
  accuracy_test=np.zeros(epochs)

  for i in range(0,epochs):
     targets_train = torch.tensor(train_1hot)
     input=torch.tensor(train_data)
     input=input.view(-1, train_data.shape[1])
     input=input.to(torch.float32)
     preds_train = NNET(input)
     losses_train[i]=criterion(preds_train,targets_train)

     targets_test = torch.tensor(test_1hot)
     input=torch.tensor(test_data)
     input=input.view(-1, test_data.shape[1])
     input=input.to(torch.float32)
     preds_test = NNET(input)
     losses_test[i]=criterion(preds_test,targets_test)
```

```
    preds_test=preds_test.detach().numpy()
    pred_test=np.array([np.argmax(preds_test[i]) for i in range(0,len(preds_test))])
    accuracy_test[i]=(accuracy_score(test_targets,pred_test))*100

    b=int(np.floor(train_data.shape[0]/20))

    for j in range(0,b):

      targets_mini = torch.tensor(train_1hot[20*j:20*j+20])
      input=torch.tensor(train_data[20*j:20*j+20])
      input=input.view(-1, train_data.shape[1])
      input=input.to(torch.float32)
      preds_mini = NNET(input)

      # updating the parameters
      loss1___ = criterion(preds_mini, targets_mini)
      loss1___.backward()
      optimizer.step()
      optimizer.zero_grad()

  return losses_train,losses_test,accuracy_test
```

Results

Analysing training loss, test loss and accuracy vs epochs for both implementations:

1) Using Numpy,

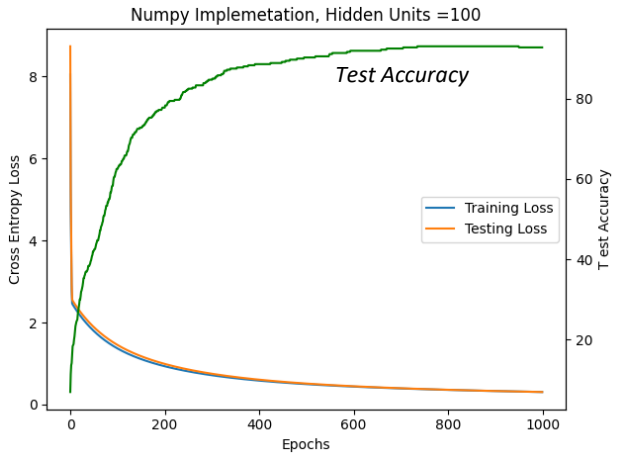


Fig 1. *Plot showing the successive change in training loss, testing loss and test accuracy with epochs for 100 hidden units using Numpy.*

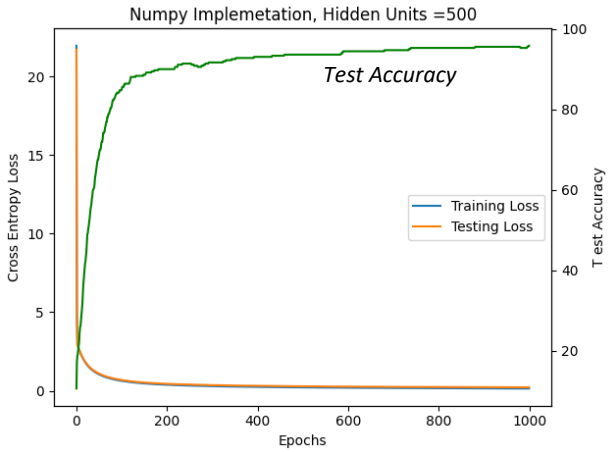


Fig 2. *Plot showing the successive change in training loss, testing loss and test accuracy with epochs for 500 hidden units using Numpy.*

Training Time on Google Colab Pro: **5 minutes**

2) Using Pytorch

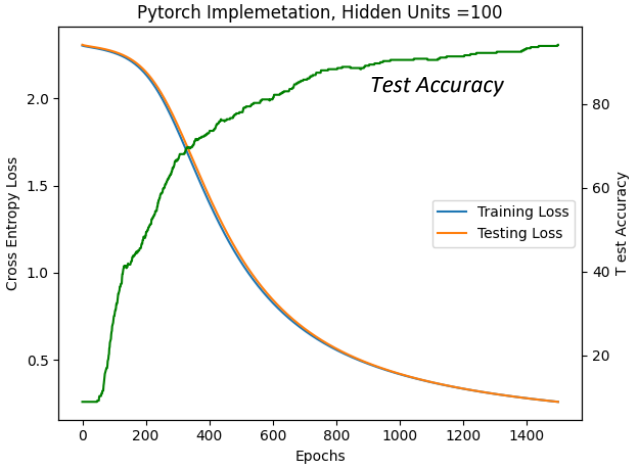


Fig 3. *Plot showing the successive change in training loss, testing loss and test accuracy with epochs for 100 hidden units using Pytorch.*

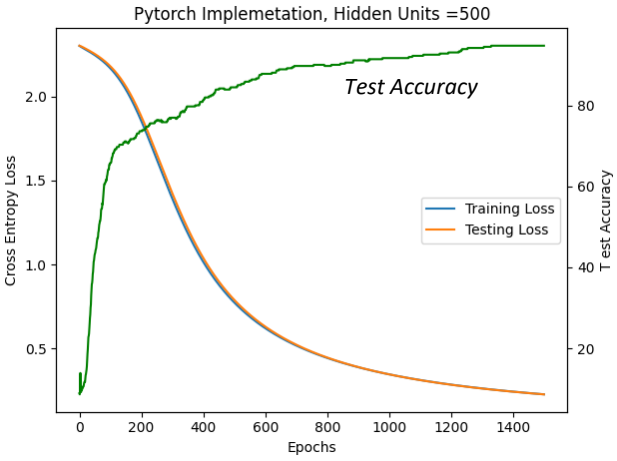


Fig 4. *Plot showing the successive change in training loss, testing loss and test accuracy with epochs for 500 hidden units using Pytorch.*

Training Time on Google Colab Pro: **1 minute**