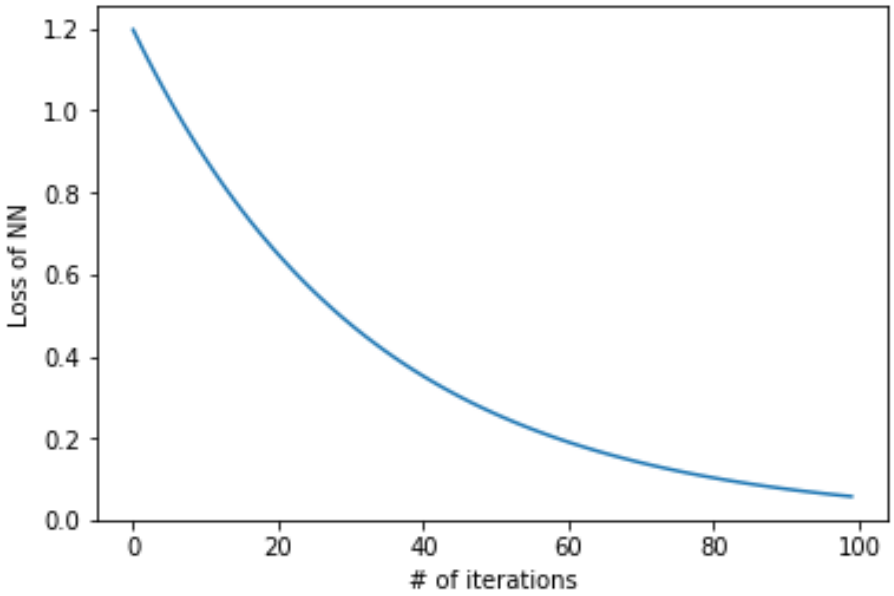
Deepayan Bhadra - Problem Set 1 Solutions

Solution to Q1: No hidden layers

This network contains an input layer and an output layer, with no nonlinearities. The output is just a linear combination of the input.

We have $a_1^1 = \sum_{k=1}^{d} w_{1k}^0 a_k^0 + b_1^1 = {w^0}^T a^0 + b_1^1$
For one sample, the loss function is expressed as $C = \frac{1}{2}(y - a_1^1)^2$
We compute the derivatives wrt weight and bias as follows.
$\nabla C_{w^0} = -(y - a_1^1)a^0, \quad \nabla C_{b_1^1} = -(y - a_1^1)$ using the chain rule

```
import numpy as np
import matplotlib.pyplot as plt
import math

d = 10 # No. of input neurons
alp = 1e-4 # Hyperparameter
a0 = np.random.randn(d,1) # Input data
y = 7*np.random.randn(1,1)+3+np.random.normal(0,1,(1,1)) # Random output data

def compute_loss(y,w,b):
       return np.sum(np.power(y-a1,2))/2

def gradient_step(w,b,a1,y): # Based on equations above

    w += alp*(y-a1)*a0
    b += alp*(y-a1)
    return w,b

loss_vec = []
num_iterations = 100 # Hyperparameter

for i in range(num_iterations):

    a1 = np.matmul(w.transpose(),a0)+b
    loss_vec.append(compute_loss(y,w,b))
    w,b = gradient_step(w,b,a1,y)

plt.plot(loss_vec)
plt.ylabel('Loss of Neural Network')
plt.xlabel('# of iterations')
```



Initially, I used $\alpha = 1e-4$. Then the learning rate was too low. After changing it to 1e-3, the convergence was much faster. As the network learns and adjusts the weights and biases, the discrepancy between the actual and the predicted output decreases as depicted in the loss plot.

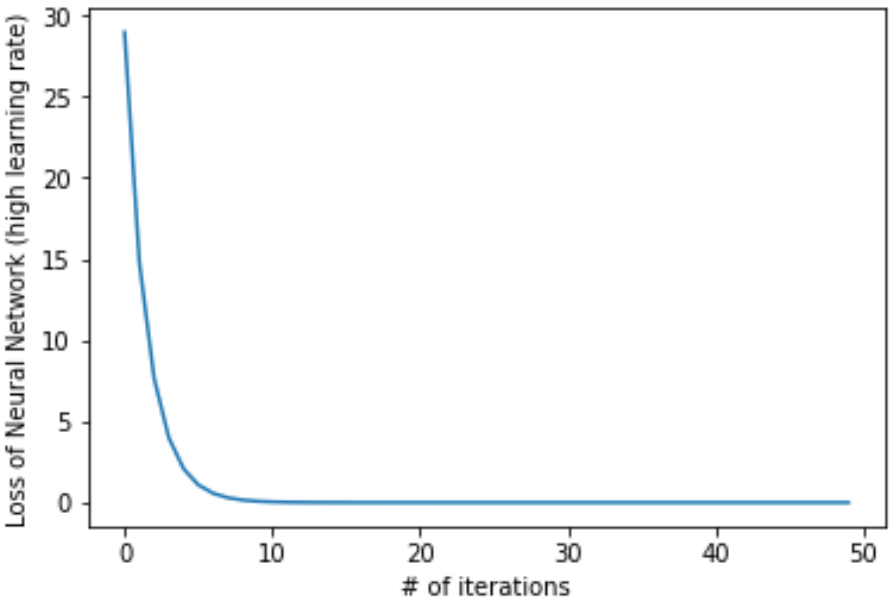
Solution to Q2: Shallow Network

Now implement a fully connected shallow neural network with a single hidden layer, and a ReLU nonlinearity. This should work for any number of units in the hidden layer and any sized input (but still just one output unit). Here, the individual weights and biases have been explicitly treated. In a regular NN implementation, they'd be treated as one black box containing all the weights/biases.

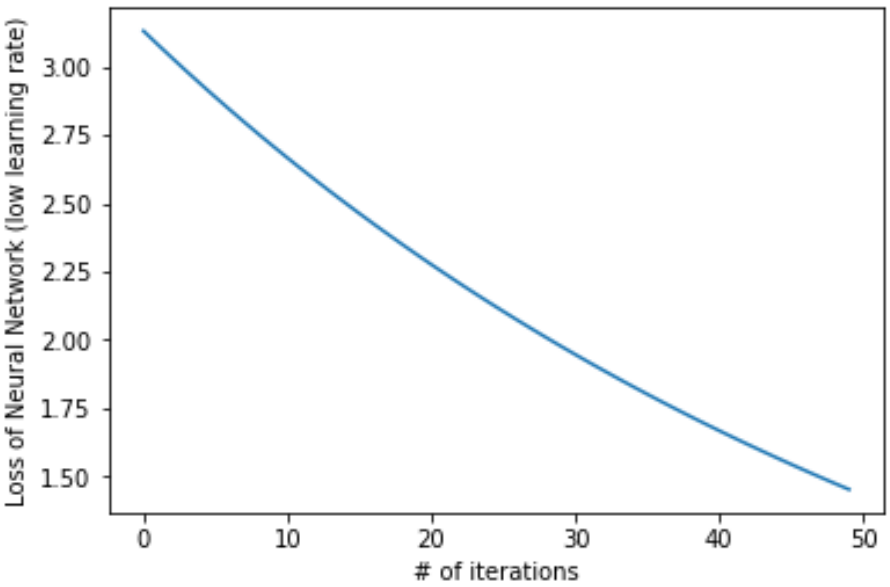
We have $a_1^2 = \sum_{k=1}^{d} w_{1k}^2 a_k^1 + b_1^2 = w^2 a^1 + b_1^2$
The loss function is expressed as $C = \frac{1}{2}(y - a_1^2)^2$
We compute the derivatives wrt weight and bias as follows.
$\nabla C_{w^2} = -(y - a_1^2)(a^1)^T, \quad \nabla C_{b_1^2} = -(y - a_1^2)$

```
import numpy as np
import matplotlib.pyplot as plt
import math

d = 20 # Change number of input neurons here
```

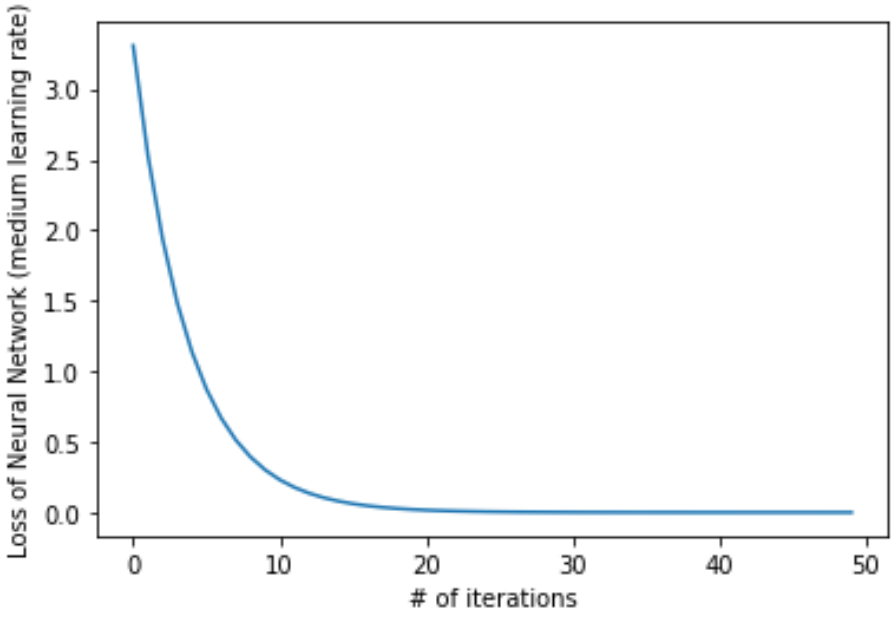
```
m = 10 # Change number of hidden units here
alp = 1e-3 # Hyperparameter
W1 = np.random.randn(m,d) # Weights b/w the input and hidden layer
W2 = np.random.randn(1,m) # Weights b/w the hidden and output layer
a0 = np.random.randn(d,1) # Input data
b1 = np.random.randn(m,1) # Bias in hidden layer
b2 = np.random.randn(1,1) # Output bias
y = np.random.randn(1,1) # Actual output

def feedforward(W1,W2,b1,b2): # Forward Pass
    z1 = np.matmul(W1,a0)+b1
    a1 = np.maximum(0,z1)
    a2 = np.matmul(W2,a1)+b2
    return z1,a1,a2

def compute_loss(y,a2): # Discrepancy between predicted and actual output
       return np.sum(np.power(y-a2,2))/2

def backprop(W1,W2,b1,b2,a1,a2,z1): # Adjust weights and biases
    W2 += alp*(y-a2)*a1.transpose()
    b2 += alp*(y-a2)
    a1_deriv = np.array(reluDerivative(z1))
    b1 += (y-a2)*alp*(np.matmul(W2,np.diagflat(a1_deriv))).transpose()
    W1 += (y-a2)*alp*(a0.dot(W2).dot(np.diagflat(a1_deriv))).transpose()

    return W1,W2,b1,b2

def reluDerivative(x):
     x[x<=0] = 0
     x[x>0] = 1
     return x

loss_vec = []
num_iterations = 50
z1,a1,a2 = feedforward(W1,W2,b1,b2)

for i in range(num_iterations):
    loss_vec.append(compute_loss(y,a2))
    W1,W2,b1,b2 = backprop(W1,W2,b1,b2,a1,a2,z1)
    z1,a1,a2 = feedforward(W1,W2,b1,b2)
plt.plot(loss_vec)
plt.ylabel('Loss of Neural Network')
plt.xlabel('# of iterations')
```



The convergence is quick due to the high learning rate. If we reduce α to $1e-4$ from $1e-3$, the number of iterations markedly increases.



If we average these two learning rates, we get something midway as expected.



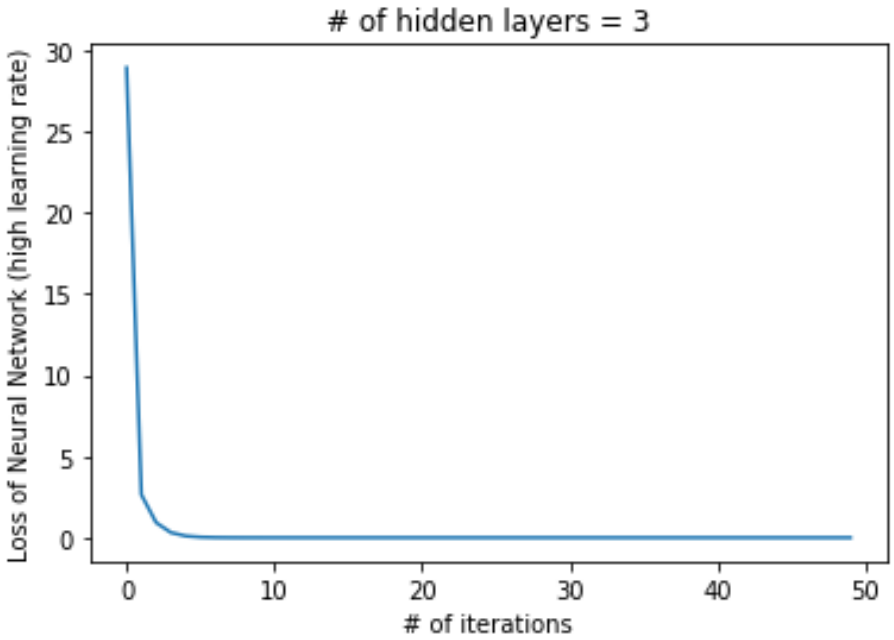
Solution to Q3: General Deep Network

```
num_layers = 5 # Change np. of layers (including input and output) here
d,m,n,p,o = 20,10,7,4,1 # Change no. of input neurons, hidden units and outputs
layer_dims = [d,m,n,p,o]
x = np.random.randn(d,1)
y = np.random.randn(1,1)
alp = 1e-3

# Helper Functions
def create_random_wt(num_layers, layer_dims):
    num_hidden = num_layers-2
    W = []
    W.append(np.random.randn(layer_dims[1],layer_dims[0]))
    for l in range(1,num_hidden):
        W.append(np.random.randn(layer_dims[l+1],layer_dims[l]))
    W.append(np.random.randn(layer_dims[-1],layer_dims[-2]))
    # compute biases if required
    return W

def create_random_bias(num_layers,layer_dims): # For the hidden and output layers
    num_bias = num_layers-1
    b = []
```

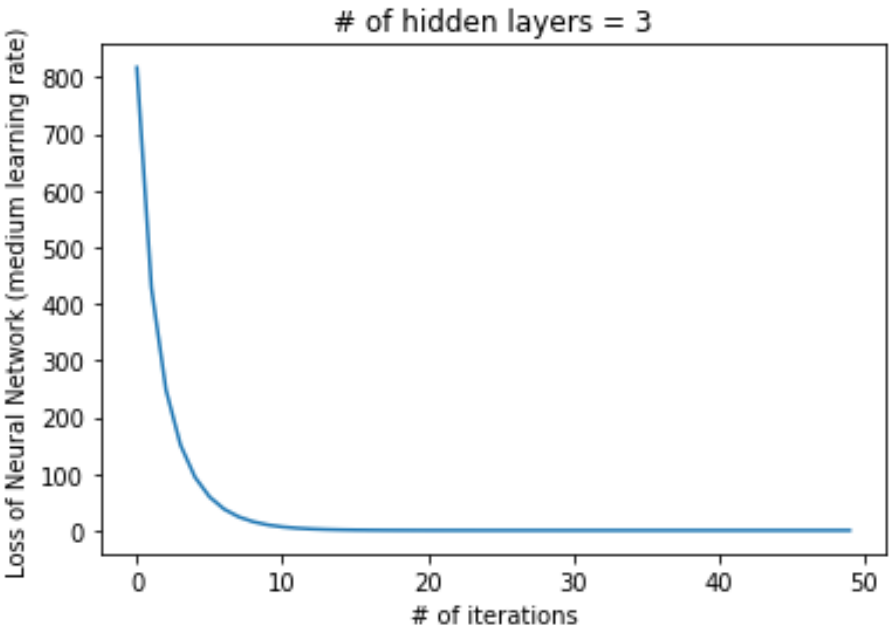
```
    b.append(np.random.randn(layer_dims[1],1))
    for l in range(1,num_bias):
        b.append(np.random.randn(layer_dims[l+1],1))
    return b

W = create_random_wt(num_layers, layer_dims)
b = create_random_bias(num_layers,layer_dims)

def compute_loss(y,a2): # Discrepancy between predicted and actual output
       return np.sum(np.power(y-a2,2))/2

def feedforward(W,b): # Forward Pass
    z,a = [],[] # Weighted input and activation vectors
    z.append(np.matmul(W[0],x)+b[0])
    for l in range(1,len(W)):
        a.append(np.maximum(0,z[l-1])) # ReLu
        z.append(np.matmul(W[l],a[l-1])+b[l])
    a.append(z[l]) # Since no ReLu at the output layer
    return z,a

def reluDerivative(x):
     x[x<=0] = 0
     x[x>0] = 1
     return x

loss_vec = []
num_iterations = 50
z,a = feedforward(W,b)

delta = [None]*len(W)   # Will store all the layer errors
nabla_b = [None]*len(W) # Will store all the bias updates
nabla_W = [None]*len(W) # Will store all the weight updates

def backprop(W,b,a,z):
    delta[len(W)-1] = -(y-a[len(W)-1])
    nabla_b[len(W)-1] = delta[len(W)-1]
    nabla_W[len(W)-1] = np.matmul(a[len(W)-2],delta[len(W)-1]).transpose()

    # Computing the gradients. (BP1) to (BP4) in Nielsen

    for i in range(len(W)-2,-1,-1):
        delta[i] = np.matmul(W[i+1].transpose(),delta[i+1])*np.array(reluDerivative(z[i]))
        nabla_b[i] = delta[i]
        nabla_W[i] = np.matmul(delta[i],a[i-1].transpose())
    nabla_W[0] = np.matmul(delta[0],x.transpose())
```

```
    # Updating the weights and biases

    for i in range(0,len(W)):
        W[i]-=alp*nabla_W[i]
        b[i]-=alp*nabla_b[i]
    return W,b

for i in range(num_iterations):
    loss_vec.append(compute_loss(y,a[len(W)-1]))
    W,b = backprop(W,b,a,z)
    z,a = feedforward(W,b)
plt.plot(loss_vec)
plt.ylabel('Loss of Neural Network')
plt.xlabel('# of iterations')
```

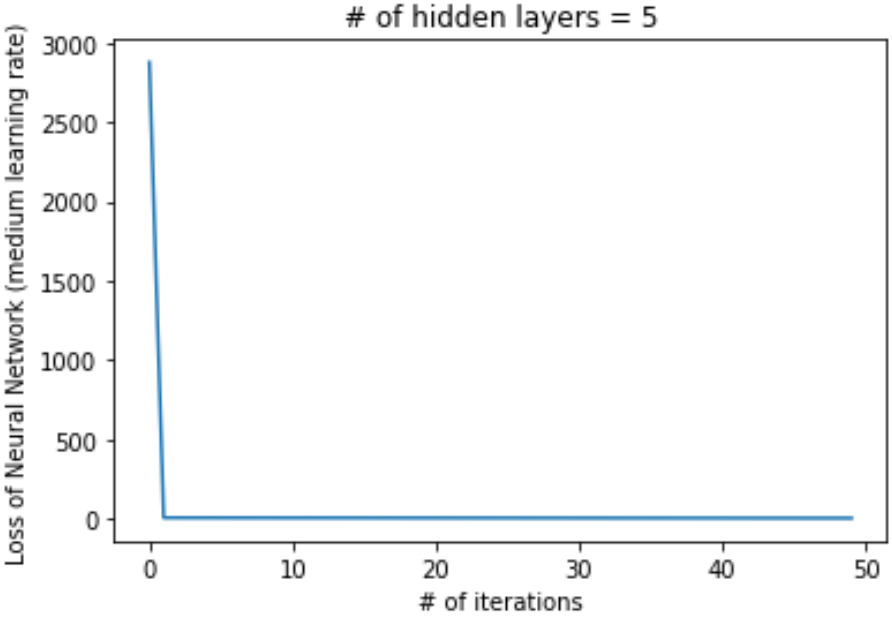
With a high learning rate α, the convergence is quick again as expected.



As the rate is lowered, the time taken to learn increases.



Now, as the number of hidden layers is increased to 5, the effect on convergence is pronounced (as expected with reasonable increase)



It's more difficult to choose hyperparameters for deep networks as compared to shallow ones.

Solution to Q4: Cross Entropy Loss

```
# This will implement the loss as given in the question

def cross_entropy(X,Y):
    Xm = X - np.max(X,axis = 1, keepdims = True)
    softmax = np.divide(np.exp(Xm),np.sum(np.exp(Xm),axis = 1,keepdims = True))
    y = -math.log(np.mean(np.sum(softmax*Y,axis=1,keepdims = True)))
    return y

# I have experimented with the SoftPlus non-linearity which is a smoothed form of the RELU.

def softplus(X):
    y = np.zeros(X.shape)
    idxN, idxP = X<0, X>=0
    xn,xp = X[idxN],X[idxP]
    y[X<0] = [math.log(1+np.exp(x)) for x in xn]
    y[X>=0] = xp+[math.log(np.exp(-x)+1) for x in xp];
    return y

# This  will implement the loss of a neural network

def net_objective(W,D,L):
    temp = D
    for i in range(0,len(W)):
        temp = softplus(np.dot(temp,W[i]))
    y = cross_entropy(temp,L);
    return y

# I have used the MNIST dataset for this part

def load_mnist():
    import h5py
    f = h5py.File("mnist.h5")
    x_test = f["x_test"]
    x_train = f["x_train"]
    y_test = f["y_test"]
    y_train = f["y_train"]

    with x_test.astype("float"):
        x_test = x_test[:]
```

```
    with y_test.astype("float"):
        y_test = y_test[:]
    with x_train.astype("float"):
        x_train = x_train[:]
    with y_train.astype("float"):
        y_train = y_train[:]
    return x_test,x_train,y_test,y_train

x_test,x_train,y_test,y_train = load_mnist()


D = x_train
W = [None] * 2 # 2 hidden layers for demonstration
W[0] = np.random.randn(784,100)
W[1] = np.random.randn(100,10)
y_train += np.ones((60000,))

# One HOT Encoding
s = pd.Series(y_train)
L = pd.get_dummies(s).values

print("""The objective value using the random weights and MNIST
      training data is {r:5.3f}""".format(r=net_objective(W,D,L)))
```

Solution to Q5: Convolution Neural Networks (CNNs)

```
d = 20 # Change number of input neurons here
m = 10 # Change number of hidden units here
alp = 1e-3 # Hyperparameter
W1 = np.random.randn(m,d) # Weights b/w the input and hidden layer
W2 = np.random.randn(1,m) # Weights b/w the hidden and output layer
a0 = np.random.randn(d,1) # Input data
b1 = np.random.randn(m,1) # Bias in hidden layer
b2 = np.random.randn(1,1) # Output bias
y = np.random.randn(1,1) # Actual output

K = 2 # Change network parameter here

# First additional constraint
```

```
W1 = [[0 if abs(i - j) > K else W1[i][j] for j in range(len(W1[i]))] for i in range(len(W1))
W1 = np.array(W1) # Converting the list back to a matrix.

# Second additional constraint of weight sharing
W1 = [[next((W1[j2][k2] for j2 in range(len(W1)) for k2 in range(len(W1[j2])) \
if j1 - k1 == j2 - k2 and abs(j1 - k1) <= K), W1[j1][k1]) for k1 in range(len(W1[j1]))] \
    for j1 in range(len(W1))]
W1 = np.array(W1) # Converting the list back to a matrix.

def feedforward(W1,W2,b1,b2): # Forward Pass
    z1 = np.matmul(W1,a0)+b1
    a1 = np.maximum(0,z1)
    a2 = np.matmul(W2,a1)+b2
    return z1,a1,a2

def compute_loss(y,a2): # Discrepancy between predicted and actual output
       return np.sum(np.power(y-a2,2))/2

def backprop(W1,W2,b1,b2,a1,a2,z1): # Adjust weights and biases
    W2 += alp*(y-a2)*a1.transpose()
    b2 += alp*(y-a2)
    a1_deriv = np.array(reluDerivative(z1))
    b1 += (y-a2)*alp*(np.matmul(W2,np.diagflat(a1_deriv))).transpose()

    # Forcing the extra constraints below

    W1 += (y-a2)*alp*(a0.dot(W2).dot(np.diagflat(a1_deriv))).transpose()
    W1 = [[0 if abs(i - j) > K else W1[i][j] for j in range(len(W1[i]))] \
           for i in range(len(W1))]
    W1 = np.array(W1) # Converting the list back to a matrix.

    W1 = [[next((W1[j2][k2] for j2 in range(len(W1)) for k2 in range(len(W1[j2])) \
    if j1 - k1 == j2 - k2 and abs(j1 - k1) <= K), W1[j1][k1]) for k1 in range(len(W1[j1]))]
        for j1 in range(len(W1))]
    W1 = np.array(W1) # Converting the list back to a matrix.

    return W1,W2,b1,b2

def reluDerivative(x):
     x[x<=0] = 0
     x[x>0] = 1
     return x

loss_vec = []
num_iterations = 50
z1,a1,a2 = feedforward(W1,W2,b1,b2)
```

```
for i in range(num_iterations):
    loss_vec.append(compute_loss(y,a2))
    W1,W2,b1,b2 = backprop(W1,W2,b1,b2,a1,a2,z1)
    z1,a1,a2 = feedforward(W1,W2,b1,b2)
plt.plot(loss_vec)
plt.ylabel('Loss of CNN')
plt.xlabel('# of iterations')
plt.title('High learning rate of 1e-3')
```