

Back Propagation Training of a MLP

Sruthikesh Surineni (ssc64)

09/22/2016

Instructor name: Dr. Erik Stone

Introduction

Classification and regression for high dimensional and large datasets is not easy. Learning them with regular machine learning algorithms is tough and there are many hyperparameters to learn a complex model. Neural networks are better in reducing the complexity of hyperparameters by using back propagation. In this project we have to classify 3 non-linearly separable data sets with high accuracy. For solving this problem we implemented a configurable multi-layer perceptron which can learn the network and generalize well.

Background

Neural networks:

In recent times neural networks have become very popular for classification and regression problems. Neural networks are very popular because of the non-linear activation functions and the hidden layers. If we design network appropriate for input data we can estimate any complex classification function using the hidden layers of neural network. Even though these hidden layers don't have the expected output their non-linear activation functions are the key to learning any complex function and generalize well any given set of data. Convolutional neural networks for object detection and recognition have been improved to an extent where machines are more accurate than humans. The reason neural networks for being very popular is the training of the network which used back propagation. Compared to regular machine learning algorithms there are very few hyper parameters to set for the network to learn a complex model.

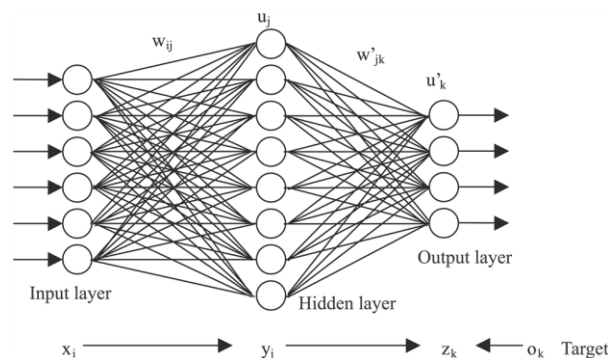


Figure 1: Sample neural network with 1 hidden and output layer

Back propagation:

For a given set of training inputs, we first forward pass the inputs through the network and compute the classification of the inputs. We compute the error of the neural network classified result with the true expected result. Using this squared error we start from the last layer and compute the gradient for that layer and we go backward to the previous layer. With the chain rule we compute the gradient for the all the layers and then we update the weights using delta rule in the opposite direction of gradient, so that we reduce the error. We run this for multiple iterations until we have reach a small error or a termination condition.

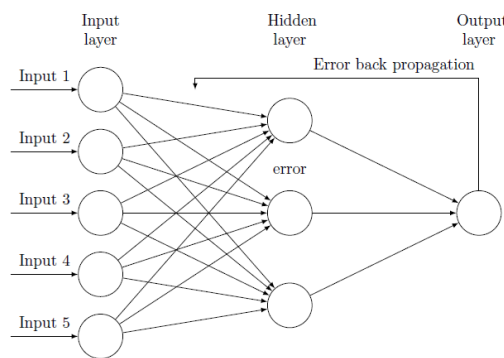


Figure 2: Back propagation of error

Initialization of weights:

The error for a particular given input problem is most likely a non-convex optimization problem where we have multiple local minimum. Considering a lot of local minimum our goal of the network is to reach the global minimum and not a local minimum because we are updating the weights in the reverse direction. In order for the network to not get stuck in a local minimum we need our weights initialized so that we can have a local minimum. Since there isn't any trivial condition for initializing the weights of the network we initialize our weights with randomly generated values depending on the input dimensions.

*Weights for a hidden layer neuron =
 $\text{rand}(\text{Number of neurons in that layer, input dimensions}) / \text{sqrt}(\text{input dimensions})$*

Learning rate:

Learning rate is the hyper parameter which is determines the amount of gradient update for each iteration. Naïve way of using this parameter is to fix the learning rate for every iteration. Ideally as we get closer to a minimum we want reduce the momentum of our weight update so that we converge fast. If we have a higher learning rate then it oscillates at

the local minimum and finally we might not converge. If we have too small learning rate then it will take very long time to converge. So, for this implementation we used exponential decay depending on the iteration count. We use this because as we run over the data multiple times we want to slow our learning so the error is small. For our runs we used $k = 20000$.

Learning rate:

$$w_{updated} = w_{prev} - \eta \partial E / \partial w$$

Exponential decay of learning rate:

$$\eta = \eta_0 e^{-k * iter}$$

k - hyperparameter

Epoch count:

Epochs is one iteration of backward pass on the whole input training data. For the current implementation we used 100000 as our maximum epochs.

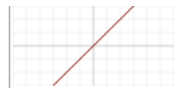
Activation function :

Activation functions are determine the output activation of each neuron whether it fires on not. As mentioned earlier non-linear activation functions provide the required complexity for learning and generalizing the input data. There are certain properties for a good non-linear functions such as anti-symmetric, zero mean. But these may not be true for the certain activation functions but may not be for all the cases. Below are the activation functions and their derivatives implemented for out neurons.

Linear (Identity) :

Linear activation is just the identity of the neuron output. This is not a really good activation function because this is not anti-symmetric.

Identity



$$f(x) = x$$


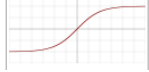
$$f'(x) = 1$$

Logistic Sigmoid:

The sigmoid non-linearity has the mathematical form as shown below. It takes a real-valued number and “squashes” it into range between 0 and 1.

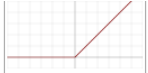
Tanh:

The tanh non-linearity has shown as below. It squashes a real-valued number to the range $[-1, 1]$.

| | | | |
|-------------------------------|---|---|--------------------------|
| Logistic (a.k.a Soft step) |  | $f(x) = \frac{1}{1 + e^{-x}}$ | $f'(x) = f(x)(1 - f(x))$ |
| TanH |  | $f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$ | $f'(x) = 1 - f(x)^2$ |

Rectified Linear units (RELU) :

RELU activation function is of the form as shown below. Recently this has become very popular because of its sparsity in activations. And this sparsity is related to biological neuron sparsity of the brain. Other nice property about RELU is that it emphasis the neuron which satisfies the forward pass as shown in derivative of RELU function.

| | | | |
|--------------------------|---|--|---|
| Rectifier ^[9] |  | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |
|--------------------------|---|--|---|

Solver:

For the backward pass we can update the gradient multiple ways. Below are the solvers implemented in this project. For all the solvers first we randomly shuffled the data every iteration so that we have good variance across iterations and batches.

VanillaGD:

Vanilla Gradient Descent (GD) is the batch training solver. In this solver we update the weights only after all the gradient for the whole training set is computed. This solver doesn't include an online learning. Once we have the delta for all the inputs we normalize it and update the gradients. Using the updated gradients we run the next epoch.

VanillaGDRand:

Vanilla Gradient Descent Rand is the same as VanillaGD, but for this solver for each batch of training data we randomly sample from the input samples. These random batch could have overlapping sample across them. Once we run the forward and backward pass for number of inputs samples, we update the weights. With the updated weights we train the network for the next epoch.

SGD:

Stochastic Gradient Descent (SGD) solver is the one which will allow for online training. For every input in the training data we forward pass the input through the network and then do a backward pass. Once we have the gradient we update the weights for immediately and then go for the next input. This solver also supports mini-batch of inputs where we find the normalized gradient for all the inputs and then update the weights. Mini-batch improves the performance of the network by doing batch update.

Experiments and Results

In all the runs we first divide the input dataset into 3 sets, train, validation and test. For all the runs we used 80% for training, 10% for validation and 10% for test.

Network structure:

Network structure for a neural network because the result of classification/ regression depends on the number of hidden layers and neurons in each layer. In this implementation for the datasets we defined the number of neurons for the first hidden layer are set to input data dimensionality. For the first 2 datasets we used only 1 hidden and an output layer. So, for the output layer since this is a classification problem, the number of neurons on the output layer is same number of classes in the input data. For semeion dataset since this a high dimensional data we used 3 hidden layers and 1 output layer. Number of neurons at each layer are shown below.

Number of neurons at each layer for semeion data = [2, 16, 32, number of classes]

Configurations:

For training the network we used various configurations. Below are some of the parameters we used in our network.

Activation functions = ['linear', 'sigmoid', 'tanh', 'relu']

Initial learning rate = [0.1, 0.01, 0.05, 0.01]

Solver = ['vanillaGD', 'vanillaGDRand', 'SGD']

We implemented all the above configurations and combinations of above configurations are ran on the datasets. Below show are some of the results for the configurations. For the first 2 datasets we used the 50000 iterations for training. For runtime reasons we ran last dataset for only 3000 iterations and because of higher dimensionality the error plateaued after few hundred iterations.

Datasets:

Three clouds:

Three clouds is 2 dimensional data set with 300 samples. In this dataset we have 3 class of data. Below shown is the three clouds with 3 classes marked differently. Classes are much cleaner, so using a smaller network could classify the data with high accuracy. Below shown are the validation errors for different configurations. Tanh is the best activation with minimum accuracy of 86% for all the solvers and learning rates.

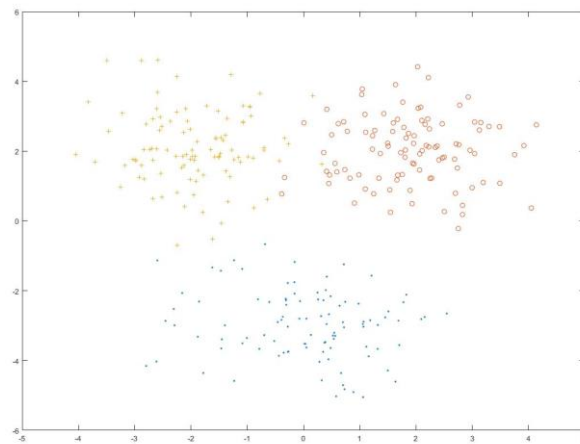


Figure 3: Three clouds dataset

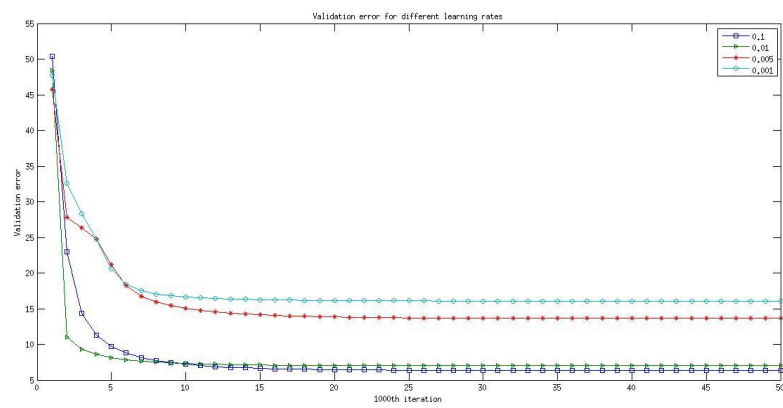


Figure 4: Validation error for different learning rates with tanh activation and vanillaGD solver

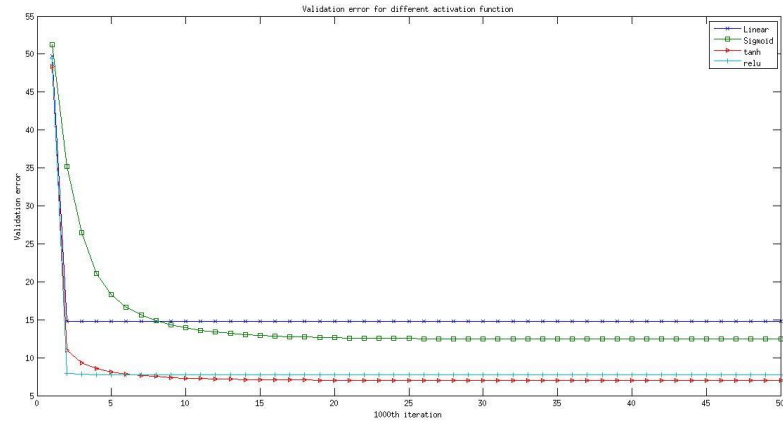


Figure 5: Validation error for different activation functions with learning rate 0.1 and vanillaGD solver

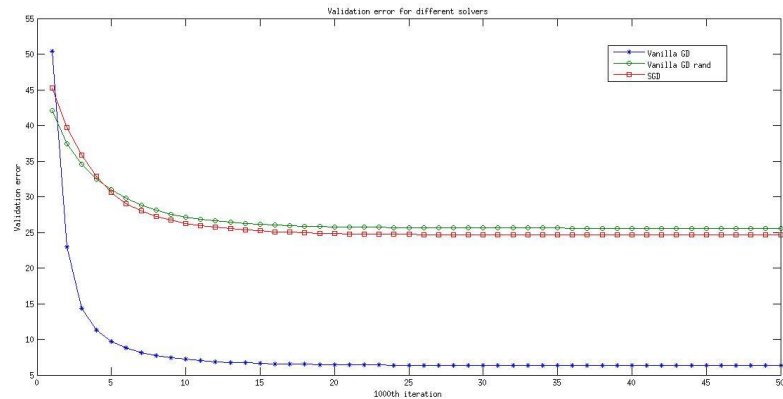


Figure 6: Validation error for different solvers with learning rate 0.1 and tanh activation

From all the above configurations linear activation saturates and doesn't improve the network because of its derivative property. Tanh performs well for all the activation and solvers.

Wine:

This is a 13 dimensional data with 178 samples. Using a similar network as the three clouds dataset we were able to classify the data accurately. Below shown are the validation errors for different configurations. Tanh is the best activation with minimum accuracy of 88% for all the solvers and learning rates.

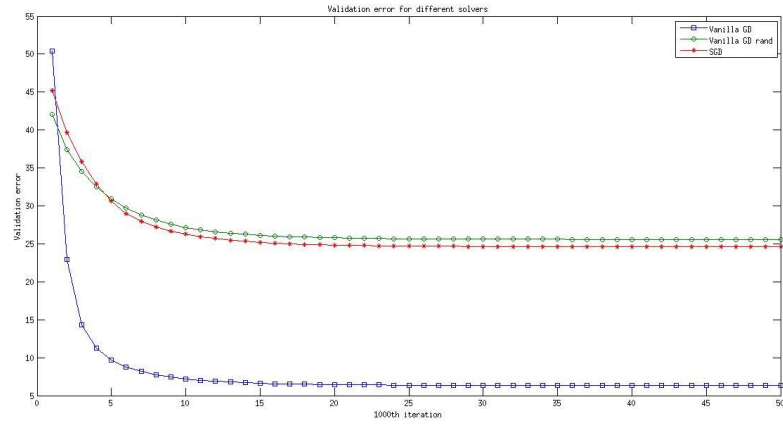


Figure 7: Validation error for different learning rates with tanh activation and vanillaGD solver

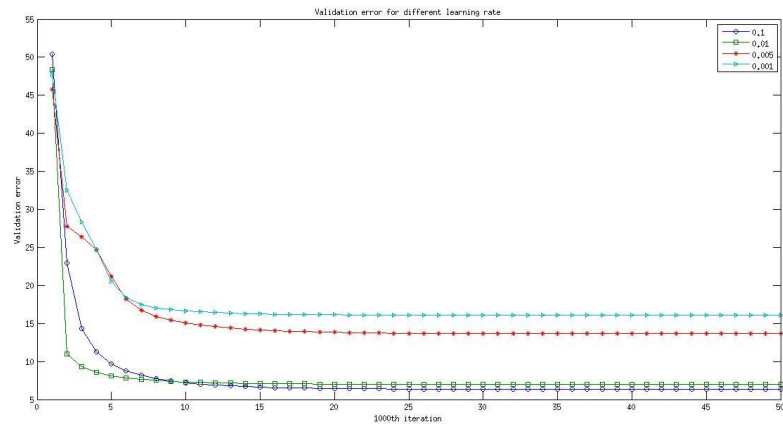


Figure 8: Validation error for different activation functions with learning rate 0.1 and vanillaGD solver

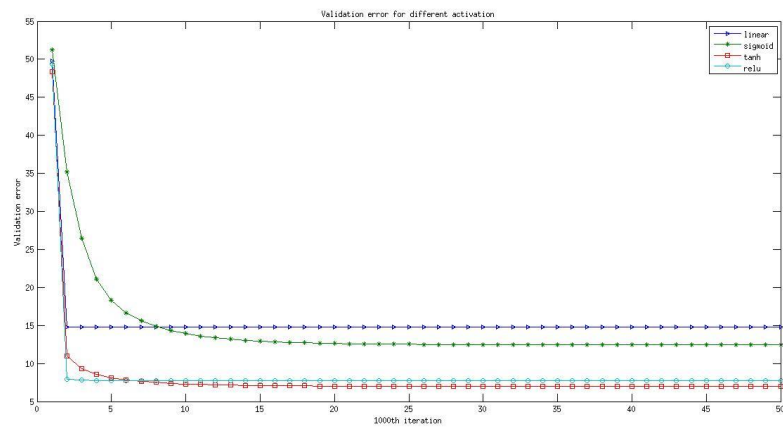


Figure 9: Validation error for different activation functions with learning rate 0.1 and vanillaGD solver

Semeion:

Semeion is image data with 265 dimensional data. Since this is a very high dimensional data we needed more layers compared to other datasets. Below shown table is the test accuracy/ 100, for this dataset. Best test data accuracy is with sigmoid activation, 0.001 learning rate and Vanilla Gradient descent solver.

| Solver | Learning rate | Linear | Sigmoid | tanh | relu |
|-----------------|---------------|---------------|---------------|---------------|---------------|
| | | Test accuracy | Test accuracy | Test accuracy | Test accuracy |
| Vanilla GD | 0.1 | 0.1006 | 0.0943 | 0.0680 | 0.0754 |
| Vanilla GD | 0.01 | 0.0967 | 0.0691 | 0.0544 | 0.1383 |
| Vanilla GD | 0.005 | 0.0277 | 0.0943 | 0.0492 | 0.1320 |
| Vanilla GD | 0.001 | 0.1320 | 0.1635 | 0.0691 | 0.0880 |
| Vanilla GD Rand | 0.1 | 0.1257 | 0.1006 | 0.1133 | 0.0628 |
| Vanilla GD Rand | 0.01 | 0.0208 | 0.1006 | 0.0479 | 0.1194 |
| Vanilla GD Rand | 0.005 | 0.0070 | 0.0817 | 0.0884 | 0.0817 |
| Vanilla GD Rand | 0.001 | 0.0759 | 0.0817 | 0.1006 | 0.0691 |
| SGD | 0.1 | 0.1572 | 0.0691 | 0.1273 | 0.1257 |
| SGD | 0.01 | 0.1006 | 0.0279 | 0.0069 | 0.1069 |
| SGD | 0.005 | 0.1006 | 0.0416 | 0.0273 | 0.0880 |
| SGD | 0.001 | 0.0943 | 0.1006 | 0.0272 | 0.0880 |

Discussion and Future work

About the best configuration:

Accuracy of semeion data is too low because of high dimensionality of input data. To improve the accuracy of this dataset we could implement convolution neural network where we can learn important features.

Since the network is not very complex using tanh is better for the first 2 datasets with small dimensionality. And also for the first 2 datasets learning rate 0.1 with exponential decay and VanillaGD solver have a consistently less validation error.

Future work:

For the future work and improving the model these are some of the features which could be useful. Accuracy for semeion data can be improved by a better neural network with appropriate hidden layers. Ensemble of multiple models is always proven to be 2% more accurate than a normal model. Solver methods implemented in this project are basic solvers,

other solvers like adagrad, adadelata etc., which are good at updated weights can be implemented for better convergence. Momentum for updating weights can be implemented so we can find the convergence faster.

MATLAB implementation code

Top file for the neural network:

```
function nn_wrapper(inFile)
% Wrapper function for neural network function

% Load input data and labels
tmp = load(inFile);
y = tmp(:,1);
X = tmp(:,2:end);

% indices = randi(size(tmp, 1), 10, 1);
% y = tmp(indices,1);
% X = tmp(indices,2:end);

% Variance of input data
disp(['Variance of input data for each dimension is :: ',num2str(var(X))]);

% Input parameters
% noOfNeuronsPerLayer = [2, 4, 8, 16, 32, length(unique(y))];
noOfNeuronsPerLayer = [size(X, 2), length(unique(y))];
trainRatio = 0.8;
testRatio = 0.1;
epoch = 50000;
errThrsd = 0.1;
maxIter = 10000;
eta = [0.1, 0.01, 0.005, 0.001];
% sigmoid, tanh, relu activation functions
% softmax function for last layer; TODO: Not sure if this works
actFnType = {'linear', 'sigmoid', 'tanh', 'relu'};
batchSize = max(1, int16(size(y,1)/ 10));
% vanillaGD - gradient descent with weight update only after all the training set is feed
forward
% vanillaGDRand - same as gradient descent but indices of every batch are randomly
generated
% SGD - stochastic gradient descent allows online mini-batch training
solver = {'vanillaGD', 'vanillaGDRand', 'SGD'};

% TODO
% Additional features
% Momentum on weights and learning rate; activation function;
% network structure;
% K-fold validation
```

```

accuracy = zeros(length(actFnType)*length(solver));
valErr = zeros(length(actFnType)*length(solver), epoch);

% nn function
for i=1:length(actFnType) % Loop for all the activation functions
    for j=1:length(solver) % Loop for all the solvers
        for k=1:length(eta) % Loop for all the learning rates
            disp(['activation : ', actFnType(i), ' and solver: ', solver(j), 'learning rate : ',
num2str(eta(k))]);
            [accuracy((i-1)*length(solver)*length(eta)+(j-1)*length(eta)+k), valErr((i-
1)*length(solver)*length(eta)+(j-1)*length(eta)+k,:)] = nn(X, y, noOfNeuronsPerLayer,
trainRatio, testRatio, epoch, errThrsd, maxIter, eta(k), actFnType{i}, batchSize, solver{j});
            disp(['Accuracy is : ', num2str(accuracy((i-1)*length(solver)*length(eta)+(j-
1)*length(eta)+k))]);
        end
    end
end
end

save(['accuracy_.mat'], 'accuracy');
save('valErr_.mat', 'valErr');

end

```

Neural network function:

```

function [accuracy, valErr] = nn(X, y, noOfNeuronsPerLayer, trainRatio, testRatio, epoch,
errThrsd, maxIter, eta, actFnType, batchSize, solver)
% Implementation of a simple neural network with back propagation

% Divide input data into train, validation and test set
[trainInd, valInd, testInd] = dividerand(size(X, 1), trainRatio, (1-trainRatio-testRatio),
testRatio); % Indices for input data split
trainX = X(trainInd,:); trainY = y(trainInd); % Training input and labels
valX = X(valInd,:); valY = y(valInd); % Validation input and labels
testX = X(testInd,:); testY = y(testInd); % test input and labels

clearvars trainInd valInd testInd; % Clear variables, not required anymore

% Data size
[nData, nDim] = size(trainX);

% Data augmentation; Note: Not sure what todo for non-image data augmentation

% Data normalization across dimensions
% zscore normalization
% Note: why not on whole input data? It is better if we just normalize
% training set. may be because we don't want to touch the test with train. Not sure to add
validation set.

```

```

trainX = zscore(trainX);
valX = zscore(valX);
testX = zscore(testX);

% Initialize input weights
W = init_weights(nDim, noOfNeuronsPerLayer, 0);

% One hot representation of labels
oneHotTrainY = oneHotEncoding(trainY);
oneHotValY = oneHotEncoding(valY);

% Train the network on training set and check the error on validation set
% to update the hyper parameters
[W, valErr] = train(trainX, valX, oneHotTrainY, oneHotValY, noOfNeuronsPerLayer, W,
epoch, errThrsd, eta, actFnType, batchSize, solver);

% Deploy the model on the test set to check the accuracy
[labels, accuracy] = test(testX, testY, noOfNeuronsPerLayer, W, actFnType);

End

```

Weight initialization:

```

% Weight Initialization; TODO: Initialization type
function out = init_weights(inpDim, noOfNeuronsPerLayer, initType)

out = cell(length(noOfNeuronsPerLayer), 1);
out{length(noOfNeuronsPerLayer), 1} = [];
for i = 1:length(noOfNeuronsPerLayer)
    if i==1 % First layer, input dimensions will be the dimensions of weights
        out{i} = rand(noOfNeuronsPerLayer(i), inpDim+1) / sqrt(inpDim);
    else % Other layers, no. of neurons in the previous layer will be the dimensions of
current layer weights
        out{i} = rand(noOfNeuronsPerLayer(i), noOfNeuronsPerLayer(i-1)+1) /
sqrt(noOfNeuronsPerLayer(i-1));
    end
end
end

% One Hot Encoding
function oneHotLabels = oneHotEncoding(labels)

valueLabels = unique(labels); % Unique labels which could may not be in sequence
nLabels = length(valueLabels); % Number of labels
nSamples = length(labels); % Number of data samples

oneHotLabels = zeros(nSamples, nLabels);
for i = 1:nLabels
    oneHotLabels(:,i) = (labels == valueLabels(i));
end

```

```

end

% Train the neural network
function [W, valErr] = train(trainX, valX, trainY, valY, noOfNeuronsPerLayer, W, epoch,
errThrsd, eta, actFnType, batchSize, solver)

[nData, nDim] = size(trainX);
etaOrig = eta;

% Preallocate cell array
deltaBatch = cell(length(noOfNeuronsPerLayer)+1, 1);
deltaBatch{length(noOfNeuronsPerLayer)+1, 1} = [];
deltaW = cell(length(noOfNeuronsPerLayer)+1, 1); 100000
deltaW{length(noOfNeuronsPerLayer)+1, 1} = [];

% Dropout scheme. If implmented correct the weights to normalize correctly.
epochIter = 1;
valErr = inf(1, epoch);
randIndices = 1:nData;

% Max no. iterations before terminating the training
while epochIter < epoch && valErr(epochIter) > errThrsd

    % Shuffle the indices every iteration to have a differnet update
    randIndices = (randperm(nData))';

    for i = 1:batchSize:nData % Online training/ batch training

        curBatch = i / batchSize + 1;

        % Generate batch indices completely random; Somehow this gives a
        % good accuracy because this way input is covered randomly
        if strcmp(solver, 'vanillaGDRand')
            if curBatch*batchSize > nData
                randIndices((curBatch-1)*batchSize+1:nData) = randi(nData, rem(nData,
batchSize), 1); % Generate random indices
            else
                randIndices((curBatch-1)*batchSize+1:curBatch*batchSize) = randi(nData,
batchSize, 1); % Generate random indices
            end
        end

        % Sample labels for forward pass
        if curBatch*batchSize > nData
            y = trainY(randIndices((curBatch-1)*batchSize+1:nData),:);
            % Forward pass
            X=forward(noOfNeuronsPerLayer, actFnType, trainX(randIndices((curBatch-
1)*batchSize+1:nData),:), W); % Feed forward phase of network
        else
            y = trainY(randIndices((curBatch-1)*batchSize+1:curBatch*batchSize),:);

```

```

    % Forward pass
    X=forward(noOfNeuronsPerLayer, actFnType, trainX(randIndices((curBatch-
1)*batchSize+1:curBatch*batchSize),:), W); % Feed forward phase of network
    end

    % Compute the error
    err = computeErr(X{end}, y);
    % Backward pass
    if strcmp(solver, 'SGD') || curBatch==1 % Online training or first batch of training
data
        deltaW = backward(actFnType, noOfNeuronsPerLayer, err, X, W);
    else % accumulate all delta
        deltaBatch = backward(actFnType, noOfNeuronsPerLayer, err, X, W);
        for k = 1:length(noOfNeuronsPerLayer) % Accumulate delta over all training
samples
            deltaW{k} = deltaW{k} + deltaBatch{k};
        end
    end
    % update weights, Overfitting: learning rate momentum, weight decay??
    % TODO: momentum for learning rate
    if strcmp(solver, 'SGD')
        W = updateWeights(noOfNeuronsPerLayer, W, deltaW, eta); % Stochastic gradient
online training
    end

end
% Batch training
if strcmp(solver, 'vanillaGD') || strcmp(solver, 'vanillaGDRand')
    for k = 1:length(noOfNeuronsPerLayer) % Accumulate delta over all training samples
        deltaW{k} = deltaW{k}./nData;
    end
    W = updateWeights(noOfNeuronsPerLayer, W, deltaW, eta); % Batch training
end

% Compute the error on validation set and decide which
% parameters are optimal and check for overfitting and termination
X=forward(noOfNeuronsPerLayer, actFnType, valX, W); % Feed forward phase of
network
% Compute the error
valErr(epochIter) = sum(sum(abs(computeErr(X{end}, valY))));
if mod(epochIter, 1000)==1
    disp(['Iteration: ', num2str(epochIter), ' Validation error: ',
num2str(valErr(epochIter))]);
end

% Annealing learning rate
% Multiple ways to do this; using the gradient difference or using
% iteration number;
% For simplicity we use iteration count
k = 2*1e-4;

```

```

    eta = etaOrig*exp(-k*epochIter); % exponential decay: CS231n

    epochIter = epochIter+1;
end
end

```

Weight update, activation, activation derivatives and error functions:

```

% Weight update function
function out = updateWeights(noOfNeuronsPerLayer, W, deltaW, eta)

out = cell(length(noOfNeuronsPerLayer), 1);
out{length(noOfNeuronsPerLayer), 1} = [];

% TODO: Other solver optimizations
% Momentum, Nesterov, Adagrad, Adadelat, Adam

% Vanilla/ Batch Gradient descent
% SGD - Online training for each sample
for i=1:length(noOfNeuronsPerLayer)
    out{i} = W{i} - eta.*deltaW{i};
end
end

% Forward pass for the activation function
function out = forward(noOfNeuronsPerLayer, actFnType, X, W)

a = cell(length(noOfNeuronsPerLayer)+1, 1);
a{length(noOfNeuronsPerLayer)+1, 1} = [];

% First layer activations are inputs to the network
a{1} = X;

for i=1:length(noOfNeuronsPerLayer)
    % Local induced field
    v = horzcat(a{i}, ones(size(a{i}, 1), 1))*W{i}';

    % activation functions
    a{i+1} = actFn(actFnType, v);
end

out = a; % Output all the activation of inputs for backpropagation
end

% Activation function
function out = actFn(actFnType, in)

thrsd = 0; % Naive activation function

```

```

switch actFnType
    case 'linear' % Identity activation
        out = in;
    case 'sigmoid' % Sigmoid activation
        out = 1./(1 + exp(-in));
    case 'tanh' % tanH activation
        out = (2./(1 + exp(-2.*in))) - 1;
    case 'relu' % RELU activation, highly sparse
        out = max(0, in);
    case 'softmax'
        out = exp(in)/sum(exp(in)); % probabilities
    otherwise
        out = in>thrsd; % Heaveside step function
end
end

% Backward pass for the activation function
function out = backward(actFnType, noOfNeuronsPerLayer, err, a, W)

delta = err.*actFnDer(actFnType, a{end}); %Last layer local gradient
for i=length(noOfNeuronsPerLayer):-1:1
    out{i} = (horzcat(a{i}, ones(size(a{i}, 1), 1))*delta); % Sum of all the delta for the batch
    of inputs
    if i>1 % Local gradient not required for first layer
        delta = actFnDer(actFnType, a{i}) .* (delta*W{i}(:,1:end-1)); % Local gradient for
    prev layer
    end
end
end

% Derivative of activation function required for backward pass
function out = actFnDer(actFnType, v)

% Derivative of activation function
switch actFnType
    case 'linear' % Identity activation
        out = 1;
    case 'sigmoid' % Sigmoid activation
        out = v.*(1-v); % Derivative
    case 'tanh' % tanH activation
        out = 1 - v.^2;
    case 'relu' % RELU activation, highly sparse
        out = v>0;
    case 'softmax' % Softmax activation function
        out = repmat(v.*(1-v), length(v), 2)*eye(length(v)) + (v*v')*(ones(length(v))-
        eye(length(v))); % TODO: Not sure if this is correct
    otherwise
        out = v; % Heaveside step function
end
end

```



```

end

% Compute error
function out = computeErr(y, trueY)
% Compute the error
out = -(trueY - y); % Derivative of error wrt current neuron y
End

```

Test function for neural network:

```

% Test function
function [labels, accuracy] = test(X, y, noOfNeuronsPerLayer, W, actFnType)

% Deploy a forward pass and classify the data
actOut = forward(noOfNeuronsPerLayer, actFnType, X, W);
% Max probability for data classification
[result, labels] = max(actOut{end}, [], 2);
% Compute the error of classification
cp = classperf(y, labels);
accuracy = cp.CorrectRate;
end

```