Implementing Backpropagation for a Feedforward Neural Network in C

Saankhya S Mondal
Sr. No. - 17990, M.Tech in AI, Dept. of CSA, IISc, Bangalore
saankhyas@iisc.ac.in

1 Introduction

In this project, the backpropagation algorithm in feedforward neural networks is implemented using C. A data structure to store and manipulate neural network weights is created. Algorithms are designed to perform forward propagation, back propagation, and update weights. The user can specify the number of hidden layers, number of neurons in each hidden layer, the loss function, the optimizer, and the activation function to be used. Backpropagation is an algorithm used for updating weights of an artificial neural network using gradient descent optimization. Given a neural network and a loss function, the algorithm calculates the gradient of the error function with respect to each of the neural network's weights. The calculation of errors takes place in a backward direction from the output layer, through the hidden layers to the input layer. Using the obtained errors, the weights are updated. Here, the backpropagation process has been discussed in detail. For demonstration, neural network models were trained and tested on the MNIST dataset for handwritten digit recognition and the results have been shown.

2 Problem Description

Let L be the number of layers, n_l be the number of neurons in layer $l, l = 1, ..., L$, and y_i^l be the output of the neuron-i in layer l. Let w_{ij}^l be the weight of connection from neuron-i, layer l to node j, layer $(l+1)$. Let η_i^l be the net input of neuron-i in layer l. Let b_j^l be the bias. Let $X = [x_1, ..., x_n]^T$ be the input sample. The input can also be thought of as output being fed into the first layer in the network. Hence, $y_i^1 = x_i, i = 1, ..., n_1$. From layer-2 onwards, units in each layer, successively compute their outputs. The output of a typical neuron is computed as

$$\eta_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^{l-1} y_i^{l-1} + b_j^l,\ y_i^l = f(\eta_j^l)$$

The function f is any differentiable non-linear activation function. The function f' is the derivative of the activation function. Following are some of the widely used activation functions.

$$\text{Sigmoid - } f(x) = \tfrac{1}{1+e^{-x}}, f'(x) = f(x)(1 - f(x))$$

$$\text{Tanh - } f(x)\tfrac{e^x - e^{-x}}{e^x + e^{-x}}, f'(x) = 1 - f(x) * f(x)$$

$$\text{ReLU - } f(x) = max(x, 0), f'(x) = 1 \text{ if } x > 0, f'(x) = 0 \text{ if } x \leq 0$$

$$\text{Softmax - } f(x_i) = \tfrac{e^{x_i}}{\sum_{j=1}^{j=k} e^{x_j}}$$

The bias term can be considered an extra input and the output can also be written as $\eta_j^l = \sum_{i=0}^{n_{l-1}} w_{ij}^{l-1} y_i^{l-1}$ where, by notation, $w_{0j}^{l-1} = b_j^l$ and $y_0^l = 1, \forall l$. The $y_1^L, y_2^L, ..., y_{n_L}^L$ form the final outputs of the neural network. The process of computing the output is known as *forward propagation*. The weights, w_{ij}^l are the parameters of the network. Let W represent all these parameters. We are given a training set, $\mathcal{D} = \{(X_i, d_i), i = 1, 2, ...\}$. Note that d_i are the target/actual output for the given sample. We will employ empirical risk minimization. The objective function is

$$\min_W \tfrac{1}{N} \sum_{i=1}^{N} L(y(X_i, W), d_i)$$

We can use stochastic gradient descent to find the minimizer of J. In this project, only one training example is used to update the weights. Weights can also be updated after running forward propagation through a batch of training set or the entire set. Let t be the iteration number and λ be the learning rate. Then, the stochastic gradient descent update equation in terms of individual weights is given by

$$w_{ij}^l(t+1) = w_{ij}^l(t) - \lambda \frac{\partial J_s}{\partial w_{ij}^l}(W(t))$$

The loss function L can be of many kinds. This project deals with two different loss functions - the *mean square error loss* and the *cross entropy error loss*.

$$J_{mse} = \frac{1}{2}\sum_{j=1}^{n_L}(y_j^L - d_j)^2,\ J_{ce} = -\sum_{j=1}^{n_L}(d_j log(y_j^L))$$

For explaining backpropagation algorithm, let us consider loss function to be mean square error loss. We need partial derivative of J with respect to weight w_{ij}^l. Let us define $\delta_j^l = \frac{\partial J}{\partial \eta_j^l}$. These are called errors. Using chain rule of differentiation, we can write $\frac{\partial J}{\partial w_{ij}^l} = \delta_j^{l+1} y_i^l$. We can get all the needed partial derivatives if we calculate δ_j^l for all nodes. We can write $\delta_j^L = (y_j^L - d_j) f'(\eta_j^L)$ for mean square error loss and $\delta_j^L = y_j^L - d_j$ for cross entropy loss with softmax activation. For any loss function L and activation function f, $\delta_j^L = \frac{\partial L}{\partial y_i^L} f'(\eta_j^L)$. For $l = L-1, ..., 2, \delta_j^l$ can be recursively calculated as $\delta_j^l = \sum_{s=1}^{n_{l+1}} \delta_s^{l+1} w_{js}^l f'(\eta_j^l)$. The calculation of all the error terms associated with each weight and bias is called *backward propagation*. Now, there can be many ways to carry out the weight updates. In this project, I have used four different methods, namely *SDG*, *Momentum*, *RMSprop*, and *Adam*. The update equation for each of the optimization techniques are given. Note that $\beta(0.9), \beta_1(0.9)$, and $\beta_2(0.999)$ are hyper-parameters.

$$\text{SGD - } w_{ij}^l(t+1) = w_{ij}^l(t) - \lambda \delta_j^{l+1} y_i^l$$

$$\text{Momentum - } m_{ij}^l(t) = \beta m_{ij}^l(t-1) + (1-\beta)\delta_j^{l+1} y_i^l,\ w_{ij}^l(t+1) = w_{ij}^l(t) - \lambda m_{ij}^l(t)$$

$$\text{RMSprop - } v_{ij}^l(t) = \beta v_{ij}^l(t-1) + (1-\beta)(\delta_j^{l+1} y_i^l)^2,\ w_{ij}^l(t+1) = w_{ij}^l(t) - \lambda \frac{\delta_j^{l+1} y_i^l}{\sqrt{v_{ij}^l(t)} + \epsilon}$$

$$\text{Adam - } m_{ij}^l(t) = \beta_1 m_{ij}^l(t-1) + (1-\beta_1)\delta_j^{l+1} y_i^l,\ v_{ij}^l(t) = \beta_2 v_{ij}^l(t-1) + (1-\beta_2)(\delta_j^{l+1} y_i^l)^2$$

$$\hat{m}_{ij}^l(t) = \frac{m_{ij}^l(t)}{1-\beta_1^t},\ \hat{v}_{ij}^l(t) = \frac{v_{ij}^l(t)}{1-\beta_2^t},\ w_{ij}^l(t+1) = w_{ij}^l(t) - \lambda \frac{\hat{m}_{ij}^l(t)}{\sqrt{\hat{v}_{ij}^l(t)} + \epsilon}$$

3 Data Structure

A data structure was created using C for the neural network. The data structure stores the number of layers in the network, the number of neurons in each layer, the weights and biases, their gradients, the input and output to each hidden layer, the first order and second order momentum terms, and the actual output. There are a total of 12 members in the data structure. The C program has a option of using adam optimization, RMSprop, and momentum, in addition to stochastic gradient descent and hence there is a need to store the moments. Memory has been alloted dynamically to all the members. The description of all the members in the data structure are tabulated.

```
struct NeuralNet{
    int n_layers;
    int* n_neurons_per_layer;
    double*** w;
    double** b;
    double*** momentum_w;
    double*** momentum2_w;
    double** momentum_b;
    double** momentum2_b;
    double** delta;
    double** in;
    double** out;
    double* targets;
};
```

Figure 1: The Neural Network struct

Member	Data type	Description
n_layers	Integer	Stores the number of layers.
n_neurons_per_layer	Integer 1D array	Stores the number of neurons in each layer.
w	Double 3D array	Stores the weights between each neurons in each pair of layers.
b	Double 2D array	Stores the bias weights from bias unit to neurons in the next layer.
momentum_w	Double 3D array	Stores the first order moment for the weights.
momentum_b	Double 2D array	Stores the first order moment for the bias.
momentum2_w	Double 3D array	Stores the second order moment for the weights.
momentum2_b	Double 2D array	Stores the second order moment for the weights.
delta	Double 2D array	Stores the errors computed for each neuron in each layer.
in	Double 2D array	Stores the input to the activation function in each layer.
out	Double 2D array	Stores the output of the activation function in each layer.
targets	Double 1D array	Stores the actual output for a given sample. It is a one-hot vector.

Table 1: Description of the members of the struct

4 Algorithms

The C program consists of functions that were used for forward propagation, backward propagation, training, testing, and calculating error. The pseudo-code for forward propagation, backward propagation, calculating the loss, and the main algorithm (training and testing simultaneously for every epoch) has been shown.

Algorithm 1: Forward Propagation

Function `forward_prop`(*nn, activation_fun, loss*)**:**
- `// Initialize inputs to activation function to 0`
- $nn.in \leftarrow 0$
- **for** *each layer* $k \leftarrow$ *1 to n_layers* **do**
  - `// Compute the inputs before applying activation function`
  - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k]$ **do**
    - $nn.in[k][j] \leftarrow nn.in[k][j] + 1.0 * nn.b[k-1][j]$
  - **end**
  - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k-1]$ **do**
    - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k]$ **do**
      - $nn.in[k][j] \leftarrow nn.in[k][j] + nn.out[k-1][i] * nn.w[k-1][i][j]$
    - **end**
  - **end**
  - `// Apply activation based on the given activation_fun and loss(only for last layer) parameters`
  - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k]$ **do**
    - $nn.out[k][j] \leftarrow f(nn.in[k][j])$
  - **end**
- **end**

return

Algorithm 2: Backward Propagation

Function `backward_prop`(*nn, learning_rate, optimizer, loss, itr*)**:**
- `// Compute error in the last layer`
- **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[last\_layer]$ **do**
  - $nn.\delta[k][j] \leftarrow f'(nn.out[last\_layer][j]) * (nn.targets[j] - nn.out[last\_layer][j])$
- **end**
- `// Backpropagate the errors through the hidden layers till the input layer is reached`
- **for** *each layer* $k \leftarrow$ *n_layers-1 to 1* **do**
  - **for** *each neuron* $i \leftarrow$ *1 to* $nn.neurons\_per\_layer[k]$ **do**
    - $sum = 0$
    - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k+1]$ **do**
      - $sum \leftarrow sum + nn.b[k][j] * nn.\delta[k+1][j]$
    - **end**
    - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k+1]$ **do**
      - $sum \leftarrow sum + nn.w[k][i][j] * nn.\delta[k+1][j]$
    - **end**
    - $nn.\delta[k][i] \leftarrow sum * f'(nn.out[k][i])$
  - **end**
- **end**
- `// update weights and biases using SGD`
- **for** *each layer* $k \leftarrow$ *1 to n_layers* **do**
  - **for** *each neuron* $i \leftarrow$ *1 to* $nn.neurons\_per\_layer[k]$ **do**
    - **for** *each neuron* $i \leftarrow$ *1 to* $nn.neurons\_per\_layer[k+1]$ **do**
      - $dw \leftarrow nn.\delta[k+1][j] * nn.out[k][i]$
      - $nn.w[k][i][j] \leftarrow nn.w[k][i][j] + learning\_rate * dw$
    - **end**
  - **end**
  - **for** *each neuron* $j \leftarrow$ *1 to* $nn.neurons\_per\_layer[k+1]$ **do**
    - $db \leftarrow nn.\delta[k+1][j] * 1.0$
    - $nn.b[k][j] \leftarrow nn.b[k][j] + learning\_rate * db$
  - **end**
- **end**

return

Algorithm 3: Calculate Loss

Function `calc_loss`(*nn, loss*)**:**
- *loss_val* $\leftarrow 0$
- **for** *each output neuron* $i \leftarrow 1$ *to* $nn.neurons\_per\_layer[last\_layer]$ **do**
  - **if** *loss == 'mse'* **then**
    - $loss\_val \leftarrow loss\_val + (0.5) * (nn.out[last\_layer][i] - nn.targets[i])^2$
  - **else if** *loss == 'ce'* **then**
    - $loss\_val \leftarrow loss\_val - nn.targets[i] * log(nn.out[last\_layer][i])$
  - **end**
- **end**

return *loss_val*

Algorithm 4: Main Algorithm

Data: Training Data $\mathcal{D}_1 = \{(X\_train_i, d\_train_i), i = 1, 2, ...\}$, Test Data $\mathcal{D}_2 = \{(X\_test_i, d\_test_i), i = 1, 2, ...\}$

`// Initialize weights and biases`

$nn.w \leftarrow random\_initialize(), nn.b \leftarrow 0$

for *each epoch itr* $\leftarrow$ *1 to n_epochs* **do**
- `// Training step - Train the model, compute training error, and accuracy`
- $correct \leftarrow 0$
- $train\_loss \leftarrow 0$
- $shuffle\_data(X\_train, d\_train)$
- **for** *each training example* $i \leftarrow$ *1 to n_train* **do**
  - `// store the input and output of the training sample`
  - $nn.out[0] \leftarrow X\_train[i]$
  - $nn.targets \leftarrow d\_train[i]$
  - $forward\_prop(nn, activation\_fun, loss)$
  - $backward\_prop(nn, learning\_rate, optimizer, loss, itr)$
  - $train\_loss \leftarrow train\_loss + calc\_loss(nn, loss)$
  - **if** $argmax(targets) == argmax(nn.out[last\_layer])$ **then**
    - $correct \leftarrow correct + 1$
  - **end**
- **end**
- $train\_accuracy \leftarrow correct/n\_train$
- $train\_loss \leftarrow train\_loss/n\_train$
- `// Testing step - Test the model, compute test error, and accuracy`
- $correct \leftarrow 0$
- $test\_loss \leftarrow 0$
- **for** *each test example* $i \leftarrow$ *1 to n_test* **do**
  - $nn.out[0] \leftarrow X\_test[i]$
  - $nn.targets \leftarrow d\_test[i]$
  - $forward\_prop(nn, activation\_fun, loss)$
  - $test\_loss \leftarrow test\_loss + calc\_loss(nn, loss)$
  - **if** $argmax(targets) == argmax(nn.out[last\_layer])$ **then**
    - $correct \leftarrow correct + 1$
  - **end**
- **end**
- $test\_accuracy \leftarrow correct/n\_test$
- $test\_loss \leftarrow train\_loss/n\_test$
- $print(itr, train\_loss, train\_accuracy, test\_loss, test\_accuracy)$

end

Furthermore, functions were written to appropriately read and preprocess(scale and normalize inputs, convert output labels to one hot encoding vectors) training and test data from csv files. Additionally, I created C function for pseudo-random number generation using a linear congruential generator. This function randomly generates a number between -1 to 1. This function is used to randomly initialize neural network weights. Another function was devised to randomly shuffle an array. This function was used before every epoch to shuffle the training data. In the C program, there are options to

- choose the number of samples to be trained.
- choose number of layers and number of neurons in each layer.
- choose hidden layer activation functions among ReLU, sigmoid, and tanh.
- choose the objective loss function to be minimized.
- choose a suitable learning rate and number of epochs to train for.
- choose the optimization techniques namely SGD, Momentum, RMSprop, and Adam.

5 Results

Due to computational constraints, I have randomly selected only 10,000 images for training. All the 10,000 images in test set have been used for testing. I chose both mean square error(mse) loss function and cross entropy(ce) loss function as objective function. In case of mean square error loss function, sigmoid activation function was applied on all the neurons in the output layer. In case of cross entropy loss function, softmax activation function has been applied in the output layer. I have shown results for four models, two neural networks with one hidden layer containing 128 neurons trained considering objective function as both loss functions, one neural network with one hidden layer containing 256 neurons, and one neural network with two layers containing 64 and 32 neurons respectively. Cross entropy loss was used in the latter two cases. The model accuracy and loss vs epoch plots have been shown.

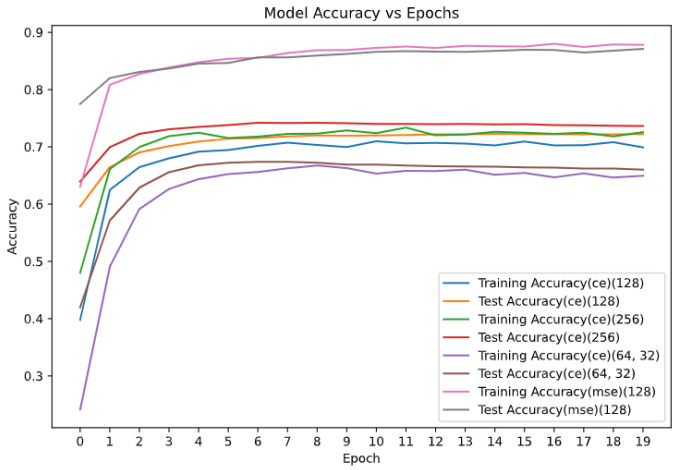

Figure 2: Model Accuracy vs Epoch

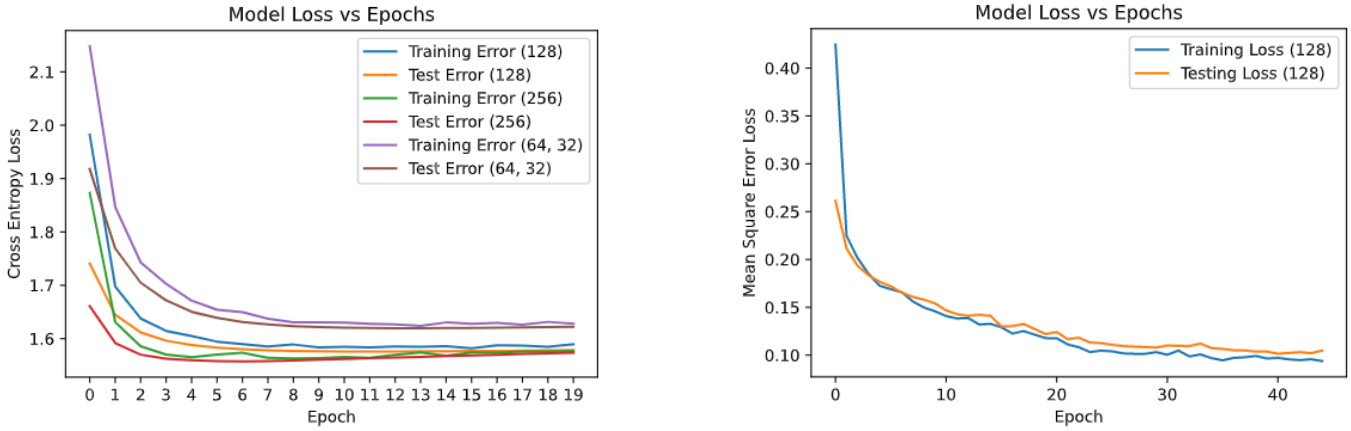

Figure 3: Model Loss vs Epoch

Adam optimization(lr=1e-4) was used for training all the models. Around 90% test accuracy was obtained for the network with one hidden layer containing 128 neurons with mean square error loss function. This was best obtained model. We often use cross entropy loss function for multiclass classification. With cross entropy loss, the network with one hidden layer containing 256 neurons achieved 74% test accuracy and performed slightly better than the other two.

6 Conclusion

To summarize, a data structure which can store neural network weights, biases, moments, gradients, input to the layers and output of the layers was created. The created data structure can only process one sample at a time. The graphical structure of a feedforward neural network was studied. Backward propagation algorithm and how the errors are backpropagated in a neural network were studied in detail. The forward propagation, backward propagation, and weight updating techniques were implemented from scratch in C. Gradient descent based optimization techniques such as 'adam', and 'rmsprop' were implemented, which improved the training process significantly. Exponential learning rate decay was performed to reduce the learning rate with increasing number of epochs. Finally for demonstration, handwritten digit classifiers (10-class classifiers) were created and trained using the MNIST dataset. The model loss and accuracy curves were observed and the results were shown.

7 Additional Points

If the same neural network architecture (cross-entropy loss case) is trained using Python frameworks like Tensorflow, the accuracy obtained is 92% which is way better than the one obtained (74%) using the C program. One of the reasons why the implementation in C is producing worse results is the weight initialization. In the framework, weight values are initialized using numbers which come from a certain distribution like normal or uniform distribution. Research has shown how using uniform distribution improves the performance of the network while using tanh activation function. Same goes for using ReLU with weights initialized using Gaussian distribution. In the code, the values which are obtained from the pseudo-random number generator do not come from any of the above distributions. The weight values may or may not be zero-mean. Due to bad initialization the neural network doesn't achieve that good a performance. The weight values may be too small and hence the weights do not get updated using gradient descent. This is the vanishing gradient problem. To bypass this problem, random initialization techniques such as Xavier or He are used which generates weight values from normal or uniform distribution and then normalizes them. The framework makes use of such methods by default. Also, the values obtained after some computation such as passing through softmax activation function or sigmoid/tanh activation may produce different results. While I was writing the code for softmax activation function, I used to get NaN or INF values. To mitigate those, I normalized the values before feeding them to softmax function. The frameworks may use some other method. There are other internal computations that take place in the framework functions which are difficult to reproduce in C.

8 References

[1] Christopher M. Bishop, Pattern Recognition and Machine Learning.

[2] Bengio, Yoshua, Ian Goodfellow, and Aaron Courville. Deep learning. Vol. 1. Massachusetts, USA:: MIT press, 2017.

[3] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[4] Linear congruential generators - Link

[5] Shuffling an array - Link