Khalid Hasan
M03543550

Assignment 2
Deep Learning, CSC737

1 Introduction

This assignment is intended to implement the back-propagation algorithm for a three-layer network (input, hidden, and output) along with evaluating the implementation by displaying both the training error and validation error over time for each experiment. This invokes two sections of the solution: Section 2 describes the details of coding implementation and Section 3 presents the related experimental analysis.

2 Software Implementation

2.1 Model Parameters Initialization

The software initializes the model parameters weight and bias for each layer except the output layer which are normalized random numbers. The shape of the parameters matrix is based on each layer's incoming and outgoing signals. If the number of incoming and outgoing signals of a layer are L_{in} and L_{out} then the shapes of weight and bias matrices are (L_{out}, L_{in}) and (L_{out}), respectively.

2.2 Defining the Activation Function

For the activation functions, the user can use any of linear, identity, sigmoid, and relu functions. The software utilizes the related activation functions of PyTorch and customized functions for their derivatives.

2.3 Defining the Loss Function

For this assignment, two types of loss functions, *mse_loss* and *ce_loss*, are implemented: Mean squared error (MSE) for linear output, and Cross-Entropy (CE) for classified output. Both functions take two arguments: expected output *y* and predicted output *y_hat* matrices and return with scalar average loss value and the derivative of loss with respect to the output (dLdy_hat).

For the MSE function: $loss = mean[\frac{1}{2}(y - y\_hat)^2]$ and $dLdy\_hat = y\_hat - y$

For the CE function: $loss = -mean[y * log(y\_hat) + (1-y) * log(1-y\_hat)]$ and $dLdy\_hat = -(\frac{y}{y\_hat} - \frac{1-y}{(1-y\_hat)})$

2.4 Defining the forward pass

The forward pass method takes an input tensor and returns the output signal. It calculates the accumulated and activation signals at each layer and then stores them to be used in the next layers. The whole process is implemented in an iterative way. The following algorithm steps demonstrate the forward pass implementation.

1. Initialize cache for accumulated and activation signals
2. Repeat through hidden to output layer
   (a) Get activation signal of previous layer a_prev from the cache
   (b) Calculate the accumulated signal for the current layer z_cur
   (c) Calculate the activation signal for the current layer a_cur using the given activation function g for the current layer: a_cur = g(z_cur)
   (d) Store the current accumulated and activation signal z_cur and a_cur in the cache
3. Return the latest activation signal from the cache which is the predicted output

2.5 Defining the backward pass

The backward pass functionality is implemented in the *backward* method which takes the loss with respect to the predicted value *dLdy_hat* as an input argument and implements the back-propagation algorithm to calculate errors at each layer and subsequently updates the weight and bias parameters in a reversed order. The following algorithm describes the implementation steps of the backward pass.

1. Initialize a variable del_next which will track the error from next layer, initially it starts with dLdy_hat
2. Iterate in a backward manner from the last to first layer to calculate gradients in each layer
   (a) Get the accumulated signal z_cur, the activation function g for the current layer, and the activation signal of current layer a_cur from the cache generated during forward pass.
   (b) Calculate the error for current layer: del_cur = del_next * g'(z_cur)
   (c) Calculate the loss w.r.t the weight of current layer : dLdW = del_cur * a_cur
   (d) Calculate the loss w.r.t the bias of the current layer
   (e) Update the error del_next to be used in the next iteration: del_next = del_cur * weight at the current layer
   (f) Update the weight and bias parameters based on gradient descent

3 Experimental Analysis

For the experimental analysis, the torch.randn function is used for generating the training and validation instances. The torch.randn returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution).

3.1 How do the different activation functions affect the weight updates in the mean squared loss function (regression task)?

3.1.1 Experimental Setup

This experiment uses 2000 and 500 training and validation instances respectively. The number of features, neurons in hidden layer, and outputs are 2, 20, and 5 respectively. The learning rate equivalent to 0.01 is used as a hyper-parameter. For this experiment, at hidden and output layers, two sets of combined activation functions have been used: (Relu, Identity) and (Sigmoid, Identity). The identity function is used at output layer since we want linear result.

3.1.2 Plots & Observations

Figure 1 shows the training and validation error resulted per epoch for the combination of activation functions. Initially, Relu gives more training and validation error compared to Sigmoid. However, for Sigmoid at the hidden layer, it takes more iteration to minimize error compared to Relu though the loss tends to 0 for both activation functions.

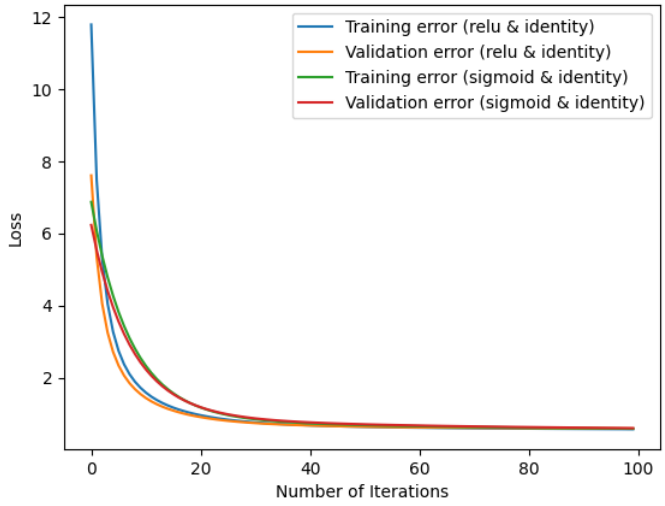

Figure 1: Training and validation error per iteration for (Relu & Identity) and (Sigmoid & Identity) activation functions

3.1.3 Inference Statements

From the observations, we can say that if want to minimize the number of iterations then we would like to choose relu in hidden layer for regression task.

3.2 How do the different activation functions affect the weight updates in cross-entropy loss function (multi-label classification task)?

3.2.1 Experimental Setup

This experiment uses 1200 and 300 training and validation instances respectively. The number of features, neurons in hidden layer, and outputs are 2, 20, and 5 respectively. The learning rate equivalent to 0.01 is used as a hyper-parameter. For this experiment, different combination of activation functions in both hidden and output layer have been tried out.

3.2.2 Plots & Observations

When the Relu is used at output layer, there is no output for loss and we get not-a-number (nan) loss. This makes sense since it only produces positive values and is not a smooth function, which can lead to unstable predictions. So, the Sigmoid function is used at output layer for this experiment.

Figure 2 visualizes the training and validation error per epoch for the combination of activation functions (Sigmoid, Sigmoid) and (Relu, Sigmoid). It clearly shows using Relu in hidden layer optimizes cost at a greater speed in contrast to Sigmoid.

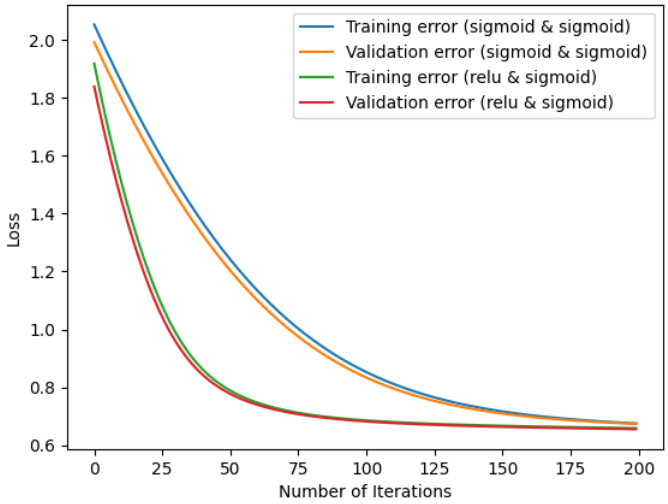

Figure 2: Training and validation error per iteration for (Relu & Sigmoid) and (Sigmoid & Identity) activation functions

3.2.3 Inference Statements

From this experiment, we can realize that relu would be a considerable candidate for choosing an activation function at hidden layer.

3.3 Result Validation with Autograd

3.3.1 Regression task

For regression task, the loss function $loss = mean[\frac{1}{2}(y - y\_hat)^2]$ and loss w.r.t output $dLdy\_hat = y\_hat - y$ are used. After comparing the gradient side by side generated by both this software implementation and the Autograd for similar input features and parameters, I see there is a ratio of 2.5 between this implementation and the Autograd. If we use $\frac{dLdy\_hat}{5}$ for backward calculation as an error w.r.t output, both provides similar gradient. I reckon there has some computation inside Autograd logic that is not present for the current assignment. That's why the console print of the difference checking in gradient generated by both implementation displays *false*.

3.3.2 Classification Task

For classification task, the loss function $loss = -mean[y * log(y\_hat) + (1 - y) * log(1 - y\_hat)]$ and the loss w.r.t output $dLdy\_hat = -(\frac{y}{y\_hat} - \frac{1-y}{(1-y\_hat)})$ are used. For this task, I have found there is a ratio of 5.0 between this implementation and the Autograd. That's why I think for the same reason mentioned in Subsection 3.3.1 the console print displays *false* for the gradient difference checking for the classification task.