

## Machine Learning Assignment 2 REPORT

We are implementing a modification of LeNet-5 convolution neural network architecture on MNIST dataset in this assignment.

### Layering:

**Conv1** → ReLu1 → **Maxpool1** → **Conv2** → ReLu2 → **Maxpool2** → **FC1** → ReLu3  
→ **FC2** → ReLu4 → **FC3** → **SoftMax**

**Conv1** is a convolution layer takes input images of size  $1*32*32$ . It has 6 convolution kernels of  $5*5$ . It has no padding and the size of its output feature is  $6*28*28$ . It is followed by a ReLu1 activation function.

**ReLu1** is rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero.

**Maxpool1** is the max-pooling layer with size and stride of 2. Its output feature size is  $6*14*14$ .

**Conv2** has 16 convolution kernels of  $5*5$ . It has no padding and output feature size is  $16*10*10$ . It is followed by a ReLu2 activation function.

**Maxpool2** is similar to Maxpool1, with size of  $2*2$  and output feature map of  $16*5*5$ .

**FC1** has 120 convolution kernels of  $5*5$ . It has no padding and output feature size is  $120*1*1$ . C5 is followed by a ReLu3 activation function.

**FC2** is fully connected to the C5 output feature vector of size  $120*1*1$  and has 84 output channels. The output feature of F6 is 84. F6 is followed by a ReLu4 activation function.

**FC3** is fully connected to the F6 output feature vector of size 84. It has 10 output features which corresponds to the 10 classes for digit 0-9.

**SoftMax** is the last layer which normalizes or “squashes” the input into a probability distribution consisting of 10 probabilities proportional to the exponentials of the input numbers.

### Implementation:

We first shuffle the training data first then get chunks from it. That way we will get the random subset with size of our full training data. Each of those random subsets will be fed to the networks, and then the gradients of that minibatch will be propagate back to update the parameters/weights of the networks.

First, we get the gradients of each layer of the networks for the current minibatch. Then we use that gradients to update the weights of each layers of the networks. To update, we just add the gradient of particular weight matrix to our existing weight matrix. To make the learning better, we scale the gradients with a learning rate hyperparameter. If we think visually, the gradient of a function is the direction of our step, whereas the learning rate is how far we take the step.

We iterate every data point in our minibatch, then feed it to the network and compare the output with the true label from the training data. The error is defined by the difference of the probability of true label with the probability of our prediction. This is so, because we’re implicitly using the SoftMax output with Cross Entropy cost function.

Because this is a Minibatch Gradient Descent algorithm, we then accumulate all the information of the current minibatch. We use all those information of the current minibatch to do the backpropagation, which will yield gradients of the networks' parameters.

### Forward Propagation:

#### For Convolution layer:

Convolutional operation as a matrix multiplication, as essentially at every patch of images, we apply a filter on it by taking their dot product, that is matrix multiplication between the input and the weight. To do this, we will use a utility function called `im2col`, which essentially will stretch our input image depending on the filter, stride, and width.

#### For Max-Pooling layer:

Pooling layer makes use of the summarization operation by taking the maximum value of each image patch. However, instead of getting the maximum value directly, we did an intermediate step: getting the maximum index first. This is because the indices are useful for the backward computation.

#### For Fully Connected Layer:

The objective of a fully connected layer is to take the results of the convolution/pooling process and use them to classify the image into a label. The output of convolution/pooling is flattened into a single vector of values, each representing a probability that a certain feature belongs to a label.

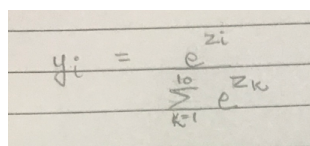
### Backward Propagation:

#### For Fully Connected Layer and SoftMax:

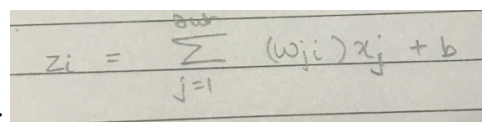
In this implantation we are using cross-entropy loss,  $L$  as our cost function. Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0.

The SoftMax function, or normalized exponential function is a function that takes as input a vector of  $K$  real numbers, and normalizes it into a probability distribution consisting of  $K$  probabilities proportional to the exponentials of the input numbers.

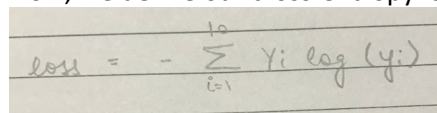
Let our outputs  $y_i$  (multi-class labels) from the SoftMax layer be equal to:


$$y_i = \frac{e^{z_i}}{\sum_{k=1}^{10} e^{z_k}}$$

where  $z_i = w^T x$ , that is:


$$z_i = \sum_{j=1}^{10} (w_{ji}) x_j + b$$

Now, we define our cross-entropy loss negative log likelihood as:


$$\text{loss} = - \sum_{i=1}^{10} y_i \log(y_i)$$

, here 10 represents the no. of classes in our model.

$$\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \text{ and } \frac{\partial L}{\partial x}$$

Our goal here is to find: where L is loss, w is weights, b is bias and x in input to the layer.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w}$$

We use chain rule, to find the partial derivative of loss w.r.t weight, bias and input. After working out the math we reach at the value of

$$\frac{\partial L}{\partial w} = \sum_{j=1}^{\text{Output } 10} \sum_{i=1} (y_i - \hat{y}_i) x_j$$

, which is used as gradient descent to update the weights and  $\partial L / \partial x$  is send to Max-Pooling layer.

### For Max-Pooling layer:

Backpropagation of the pooling layer computes the error which is acquired by this single value “winning unit”. To keep track of the “winning unit” its index noted during the forward pass and used for gradient routing during backpropagation.

Maxpool layer is similar to ReLu, because that’s essentially what max operation do in backpropagation. We let the gradient pass through when the ReLu result is non-zero, and otherwise we block the gradient by setting it to zero.

We basically upscale our input coming from the FC layers,  $\partial L / \partial x$  and only let the max values of the weights of the filters stay put and change all other values to zero. Then this updated differential is sent to the Convolution layer.

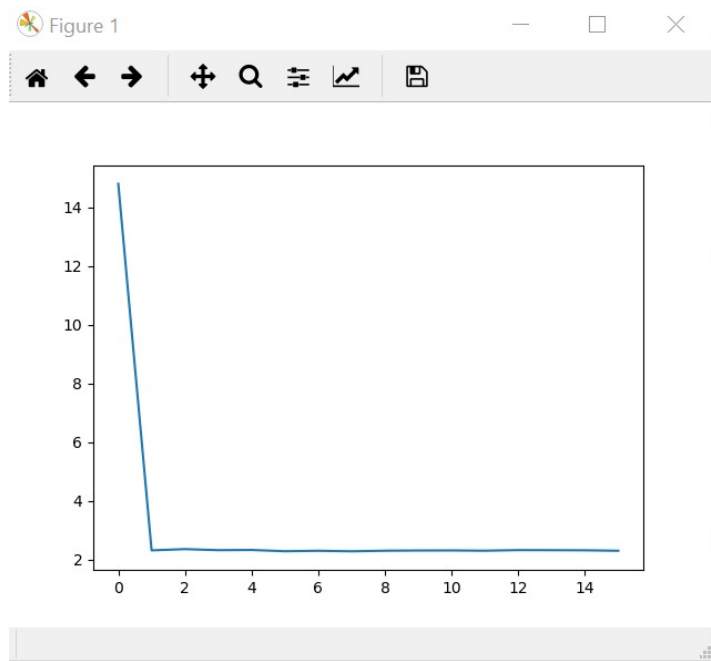
### For Convolution layer:

We approach the convolution layer as kind of normal feed forward layer, which is just the matrix multiplication between the input and the weight. To do this, we will use a utility function called im2col, which essentially will stretch our input image depending on the filter, stride, and width. Now, for back propagation we derive the partial derivative of the layer. As the bias is added to each of our filter, we’re accumulating the gradient to the dimension that represent of the number of filters. Now we calculate  $\partial W$ , gradient of filters and  $\partial X$ , the input gradient.

At the end, we’re getting the gradient of the stretched image (using im2col). To undo this, and get the real image gradient, we’re going to de-im2col that using operation called col2im to the stretched image. And hence, we have our image input gradient,  $\partial X$ .

### Analysis

As shown below the graph depicts, error rate vs epochs. In our model we use learning rate = 0.0001 as that’s the optimal one. Otherwise the error rate doesn’t dip below 2.3



We only check accuracy at the end of the training to increase speed in our model and make it robust.