Training Report Day-3

Ankur (URN: 2302473)
Department of Computer Science and Engineering
Guru Nanak Dev Engineering College Ludhiana

June 30, 2025

Abstract

This report outlines the implementation and step-by-step training of a Convolutional Neural Network (CNN) for image classification using the MNIST dataset. The training process is divided into two primary phases: the forward phase and the backward (backpropagation) phase. The forward phase involves propagating inputs through the layers of the network — convolution, pooling, and softmax — to compute predictions and evaluate performance using cross-entropy loss. A softmax layer is implemented from scratch and integrated into the network to convert raw outputs into class probabilities. In the backward phase, gradient calculations are performed starting from the loss function and propagated backward through the softmax layer, preparing the groundwork for learning via weight updates. Mathematical derivations for softmax gradients are explored, followed by their incorporation into code. Initial results show low accuracy (10%) and a loss of approximately 2.3, which aligns with expectations for a randomly initialized network.

1 Training Overview

The training of any neural network mainly has two phases

1. A **Forward Phase**, where input is propagated forward through the network. During the forward phase, each layer will cache any data (like inputs, intermediate values, etc) it'll need for the backward phase. This means that any backward phase must be preceded by a corresponding forward phase.

2. A **Backward Phase**, where the gradients are backpropagated through the network and the weights are updated. During the backward phase, each layer will receive a gradient and also return a gradient. It will receive the gradient of loss concerning its outputs ($\frac{\partial L}{\partial out}$) and return the gradient of loss concerning its inputs ($\frac{\partial L}{\partial in}$)

The groundwork for the forward phase was laid down on day 2, except for the softmax function. Which I implemented today

1.1 Softmax

```python
import numpy as np

class Softmax:
    # A standard fully connected layer with softmax activation.
    def __init__(self, input_len, nodes):
        self.weights = np.random.randn(input_len, nodes)/ input_len
        self.biases = np.zeros(nodes)


    def forward(self, inputs):
        '''
        Performs a forward pass of the softmax layer using the given input.
        Returns a 1d numpy array containing the respective probability values.
        - input can be any array with any dimensions.
        '''
        inputs = inputs.flatten()


        totals = np.dot(inputs, self.weights) + self.biases
        exp = np.exp(totals)

        return exp / np.sum(exp, axis=0)
```

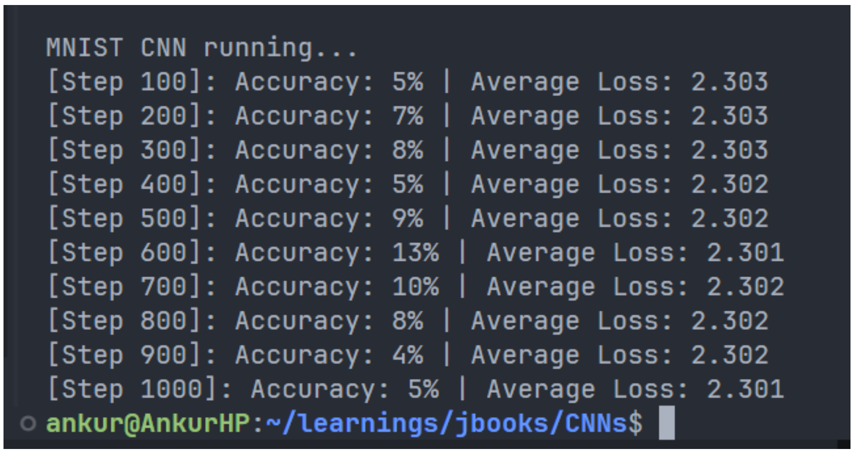
2 The Forwared Phase

The following code implements the forward phase of training a CNN over the mnist data.

```
import numpy as np
from tensorflow.keras.datasets import mnist
from conv import Conv3x3
from maxpool import MaxPool2
from softmax import Softmax

# Load MNIST data
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Initialize the layers
conv = Conv3x3(8)                       # 28x28x1 input -> 26x26x8 after conv
pool = MaxPool2()                       # 26x26x8 -> 13x13x8 after pooling
softmax = Softmax(13*13*8, 10)          # Fully connected layer: input 13*13*8 -> 10 output classes

def forward(image, label):
    '''
    Performs a full forward pass of the CNN on a single image.
    Returns the predicted probabilities, accuracy (1 or 0), and loss value.
    '''
    # Normalize input: bring pixel values from [0, 255] -> [-0.5, 0.5]
    out = conv.forward((image / 255) - 0.5)
    out = pool.forward(out)
    out = softmax.forward(out)

    # Cross-entropy loss
    loss = -np.log(out[label])

    # Accuracy check: predicted class == true label?
    acc = 1 if np.argmax(out) == label else 0

    return out, acc, loss

print("MNIST CNN running...")
```

```python

# Training stats
total_loss, num_correct = 0, 0

# Train on first 1000 images
for i, (image, label) in enumerate(zip(train_images[:1000], train_labels
    [:1000])):
    _, acc, loss = forward(image, label)

    total_loss += loss
    num_correct += acc

    # Print every 100 steps
    if (i % 100 == 99):
        print('[Step %d]: Accuracy: %d%% | Average Loss: %.3f' %
              (i + 1, num_correct, total_loss / 100))

        # Reset stats
        total_loss = 0
        num_correct = 0
```

Output



```
MNIST CNN running...
[Step 100]: Accuracy: 5% | Average Loss: 2.303
[Step 200]: Accuracy: 7% | Average Loss: 2.303
[Step 300]: Accuracy: 8% | Average Loss: 2.303
[Step 400]: Accuracy: 5% | Average Loss: 2.302
[Step 500]: Accuracy: 9% | Average Loss: 2.302
[Step 600]: Accuracy: 13% | Average Loss: 2.301
[Step 700]: Accuracy: 10% | Average Loss: 2.302
[Step 800]: Accuracy: 8% | Average Loss: 2.302
[Step 900]: Accuracy: 4% | Average Loss: 2.302
[Step 1000]: Accuracy: 5% | Average Loss: 2.301
ankur@AnkurHP:~/learnings/jbooks/CNNs$
```

Figure 1: Output for forward phase

At this point, the model is doing as expected. As for randomly initialized weights, and a way of correcting itself, one would expect the network to be as good as random guessing. That's why the accuracy is around 10% and loss is around 2.3 as $-\ln(0.1) = 2.302$

Next, I implemented the backward phase so that the network could learn.

3 The Backward Phase (Backpropagation)

I started from the end and worked my way towards the beginning. In this project, the loss function that I am using is *cross-entropy loss*, which is $L = -ln(p_c)$.

3.1 Backprop: Softmax

The first thing we need to calculate is the input to the Softmax layer's backward phase, $\frac{\partial L}{\partial out_s}$ where out_s is the output from the Softmax layer: a vector of 10 probabilities.

$$\frac{\partial L}{\partial out_s(i)} = \begin{cases} 0 & \text{if } i \neq c \\ -\frac{1}{p_c} & \text{if } i = c \end{cases}$$

First, let's calculate the gradient of $out_s(c)$ concerning the totals (the values passed into the softmax activation). Let t_i be the total for class i. Then we can write $out_s(c)$ as:

$$out_s(c) = \frac{e^{t_c}}{\sum_i e^{t_i}} = \frac{e^{t_c}}{S}$$

where $S = \sum_i e^{t_i}$

Now, consider some class k such that $k \neq c$. We can rewrite $out_s(c)$ as:

$$out_s(c) = e^{t_c} S^{-1}$$

And then we can use the chain rule to derive:

$$\begin{aligned} \frac{\partial out_s(c)}{\partial t_k} &= \frac{\partial out_s(c)}{\partial S} \times \frac{\partial S}{\partial t_k} \\ &= -e^{t_c} S^{-2} \times \frac{\partial S}{\partial t_k} \\ &= -e^{t_c} S^{-2}(e^{t_k}) \\ &= \boxed{\frac{-e^{t_c} e^{t_k}}{S^2}} \end{aligned}$$

that was assuming that $k \neq c$, Now I will derive for c using the Quotient rule of derievative as both e^{t_c} and S are in terms of t_c,

$$\begin{aligned} \frac{\partial out_s(c)}{\partial t_c} &= \frac{S e^{t_c} - e^{t_c} e^{t_c}}{S^2} \\ &= \boxed{\frac{e^{t_c}(S - e^{t_c})}{S^2}} \end{aligned}$$

Hence, we have found out that,

$$\frac{\partial out_s(k)}{\partial t_k} = \begin{cases} \frac{-e^{t_c} e^{t_k}}{S^2} & \text{if } k \neq c \\ \frac{e^{t_c}(S - e^{t_c})}{S^2} & \text{if } k = c \end{cases}$$

Hence, now I will incorporate this knowledge into the softmax function's code.

```python
class Softmax:
  # ...

  def backprop(self, d_L_d_out):
    '''
    Performs a backward pass of the softmax layer.
    Returns the loss gradient for this layer's inputs.
    - d_L_d_out is the loss gradient for this layer's outputs.
    '''
    # We know only 1 element of d_L_d_out will be nonzero
    for i, gradient in enumerate(d_L_d_out):
      if gradient == 0:
        continue

      # e^totals
      t_exp = np.exp(self.last_totals)

      # Sum of all e^totals
      S = np.sum(t_exp)

      # Gradients of out[i] against totals
      d_out_d_t = -t_exp[i] * t_exp / (S ** 2)
      d_out_d_t[i] = t_exp[i] * (S - t_exp[i]) / (S ** 2)

      # ... to be continued
```