Training Report Day-4

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

June 26, 2025

1 Backprop: Softmax

Yesterday, I started working on backpropagation for the softmax layer, and today I continued with that. Ultimately, we want to calculate the loss against weights, biases, and input.t

- We will use weights gradient $\frac{\partial L}{\partial w}$ to update our layer's weights.
- We will use biases gradient $\frac{\partial L}{\partial b}$ to update our layer's biases.
- We will return the input gradient $\frac{\partial L}{\partial input}$ from the backprop method so that the next (actually previous) layer can use it.

So next I worked on finding the necessary middleware for calculating the abovementioned gradients,
We know that

$$t = w * input + b$$

Therefore the required gradients are as follows:

$$\frac{\partial t}{\partial w} = input$$

$$\frac{\partial t}{\partial b} = 1$$

$$\frac{\partial t}{\partial input} = w$$

Therefore we can say,

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial t} \cdot \frac{\partial t}{\partial w}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial t} \cdot \frac{\partial t}{\partial b}$$

$$\frac{\partial L}{\partial \text{input}} = \frac{\partial L}{\partial \text{out}} \cdot \frac{\partial \text{out}}{\partial t} \cdot \frac{\partial t}{\partial \text{input}}$$

Next, I translated the above logic into code and incorporated it into the *softmax class's backprop method*. After that, I updated the weights and biases of the softmax layer.

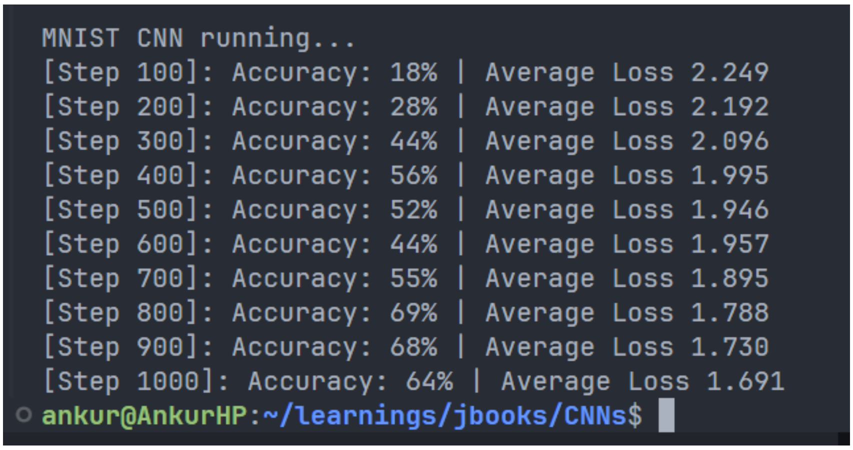
```
# Update weights / biases
    self.weights -= learn_rate * d_L_d_w
    self.biases -= learn_rate * d_L_d_b
    return d_L_d_inputs.reshape(self.last_input_shape)
```

Reshaping to *last_input_shape* ensures that this layer returns gradients for its input in the same format that the input was originally given to it.

The backpropagation for the softmax layer has been set up. At this point, I trained and tested the model by utilizing the backprop method of the softmax layer.

```
# Imports and setup here
# ...

def forward(image, label):
  # Implementation excluded
  # ...

def train(im, label, lr=.005):
  '''
  Completes a full training step on the given image and label.
  Returns the cross-entropy loss and accuracy.
  - image is a 2d numpy array
  - label is a digit
  - lr is the learning rate
  '''
  # Forward
  out, acc, loss = forward(im, label)

  # Calculate the initial gradient
  gradient = np.zeros(10)
  gradient[label] = -1 / out[label]

  # Backprop
  gradient = softmax.backprop(gradient, lr)
  # TODO: backprop MaxPool2 layer
  # TODO: backprop Conv3x3 layer

  return loss, acc

print('\n\nMNIST CNN running...')

# Train!
loss = 0
num_correct = 0
for i, (im, label) in enumerate(zip(train_images[0:1000], train_labels
    [0:1000])):
  if i % 100 == 99:
    print(
      '[Step %d]: Average Loss %.3f | Accuracy: %d%%' %
      (i + 1, loss / 100, num_correct)
    )
    loss = 0
    num_correct = 0

  l, acc = train(im, label)
  loss += l
  num_correct += acc
```

The results show that the network is learning. The accuracy is going high and loss is going down. Going forward, I will apply backpropagation for the *max-pool* layer and the *conv* layer as well.



```
MNIST CNN running...
[Step 100]: Accuracy: 18% | Average Loss 2.249
[Step 200]: Accuracy: 28% | Average Loss 2.192
[Step 300]: Accuracy: 44% | Average Loss 2.096
[Step 400]: Accuracy: 56% | Average Loss 1.995
[Step 500]: Accuracy: 52% | Average Loss 1.946
[Step 600]: Accuracy: 44% | Average Loss 1.957
[Step 700]: Accuracy: 55% | Average Loss 1.895
[Step 800]: Accuracy: 69% | Average Loss 1.788
[Step 900]: Accuracy: 68% | Average Loss 1.730
[Step 1000]: Accuracy: 64% | Average Loss 1.691
ankur@AnkurHP:~/learnings/jbooks/CNNs$
```

Figure 1: Output for backprop:softmax

2 Backprop: Maxpool

A Max Pooling layer can't be trained because it doesn't have any weights, but we still need to implement a *backprop()* method for it to calculate gradients.

During the forward pass, the Max Pooling layer takes an input volume and halves its width and height dimensions by picking the max values over 2x2 blocks. The backward pass does the opposite: we'll double the width and height of the loss gradient by assigning each gradient value to where the original max value was in its corresponding 2x2 block.

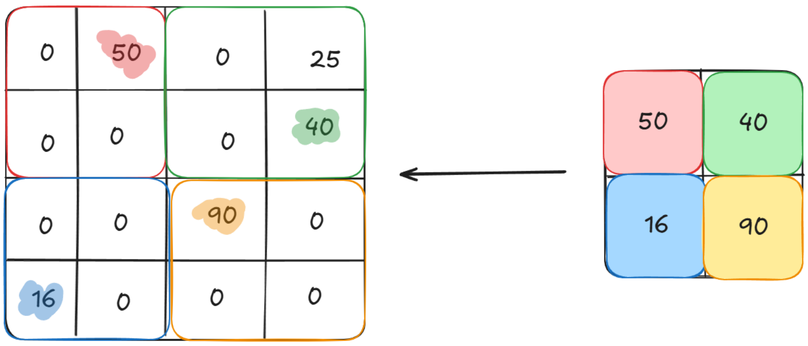


Figure 2: Backprop: maxpool

Each gradient value is assigned to where the original max value was, and every other value is zero. An input pixel that isn't the max value in its 2x2 block would have zero marginal effect on the loss because changing that value slightly wouldn't change the output at all! In other words, $\frac{\partial L}{\partial input} = 0$ for non-max pixels. On the other hand, an input pixel that is the max value would have its value passed through to the output, so $\frac{\partial output}{\partial input} = 1$. Hence $\frac{\partial L}{\partial input} = \frac{\partial L}{\partial output}$

3 Backprop: Conv

We're primarily interested in the loss gradient for the filters in our conv layer, since we need that to update our filter weights. We already have $\frac{\partial L}{\partial out}$ for the conv layer, so we just need $\frac{\partial out}{\partial filters}$. To calculate that, we ask ourselves this: how would changing a filter's weight affect the conv layer's output? The reality is that changing any filter weights would affect the entire output image for that filter since every output pixel uses every pixel weight during convolution.

By observing how the $out(i, j)$ gets affected by changing any filter weight, we can derive the required gradient.

$$
\begin{aligned}
\text{out}(i, j) &= convolve(image, filter) \\
&= \sum_{x=0}^{3} \sum_{y=0}^{3} \text{image}(i + x, j + y) \cdot \text{filter}(x, y)
\end{aligned}
$$

$$
\frac{\partial \text{out}(i, j)}{\partial \text{filter}(x, y)} = \text{image}(i + x, j + y)
$$

Therefore we can write

$$
\frac{\partial L}{\partial \text{filter}(x, y)} = \sum_i \sum_j \frac{\partial L}{\partial \text{out}(i, j)} \cdot \frac{\partial \text{out}(i, j)}{\partial \text{filter}(x, y)}
$$

Add this to the conv3x3 class.

```
class Conv3x3
  # ...

  def backprop(self, d_L_d_out, learn_rate):
    '''
    Performs a backward pass of the conv layer.
    - d_L_d_out is the loss gradient for this layer's outputs.
    - learn_rate is a float.
    '''
    d_L_d_filters = np.zeros(self.filters.shape)

    for im_region, i, j in self.iterate_regions(self.last_input):
      for f in range(self.num_filters):
        d_L_d_filters[f] += d_L_d_out[i, j, f] * im_region

    # Update filters
    self.filters -= learn_rate * d_L_d_filters

    # We aren't returning anything here since we use Conv3x3 as
    # the first layer in our CNN. Otherwise, we'd need to return
    # The loss gradient for this layer's inputs, just like every
    # another layer in our CNN.
    return None
```

4 Testing the model

Finally, I trained the model on the first 1000 images from the MNIST dataset over 10 epochs. The driver code, along with the output, is as follows.

```
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
  ()

```

```python
# 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

def train(im, label, lr=.005):
  '''
  Completes a full training step on the given image and label.
  Returns the cross-entropy loss and accuracy.
  - image is a 2d numpy array
  - label is a digit
  - lr is the learning rate
  '''
  # Forward
  out, acc, loss = forward(im, label)

  # Calculate initial gradient
  gradient = np.zeros(10)
  gradient[label] = -1 / out[label]

  # Backprop
  gradient = softmax.backprop(gradient, lr)
  gradient = pool.backprop(gradient)
  gradient = conv.backprop(gradient, lr)


  return loss, acc

print('\n\nMNIST CNN running...\n')

# Train!
epochs = 10

for epoch in range(epochs):
  print('--- Epoch %d ---' % (epoch + 1), end=' ')


  loss = 0
  num_correct = 0
```

```python
  for i, (im, label) in enumerate(zip(train_images[0:1000], train_labels
    [0:1000])):
    if i == 999:
      print(
          'Accuracy: %d%% | Average Loss %.3f' %
          (num_correct/10, loss / 1000)
        )
      loss = 0
      num_correct = 0

    l, acc = train(im, label)
    loss += l
    num_correct += acc


# Test the CNN
print('\n--- Testing the CNN ---')
loss = 0
num_correct = 0
for im, label in zip(test_images[:1000], test_labels[:1000]):
  _, acc, l = forward(im, label)
  loss += l
  num_correct += acc

num_tests = 1000
print('Test Loss: ', loss / num_tests)
print('Test Accuracy: ', (num_correct / num_tests)*100, "%")
```

Output

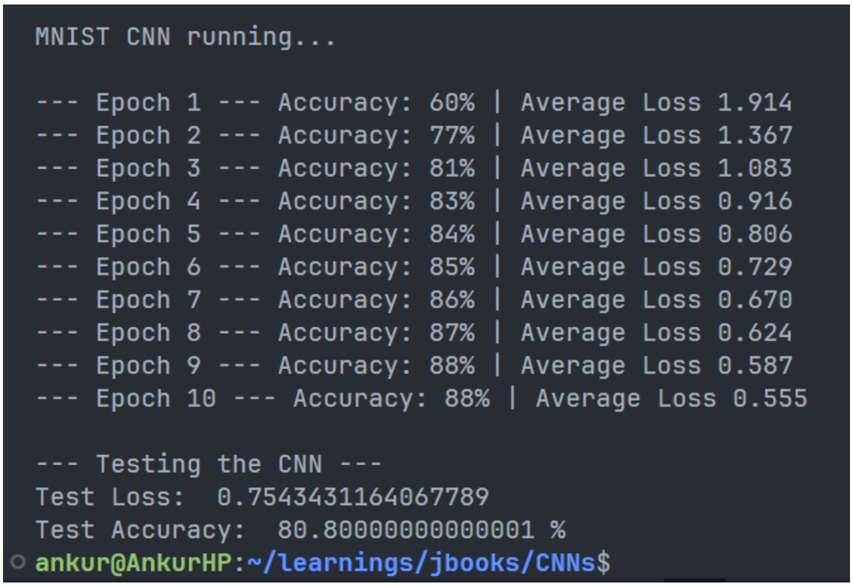


Figure 3: Results of training and testing