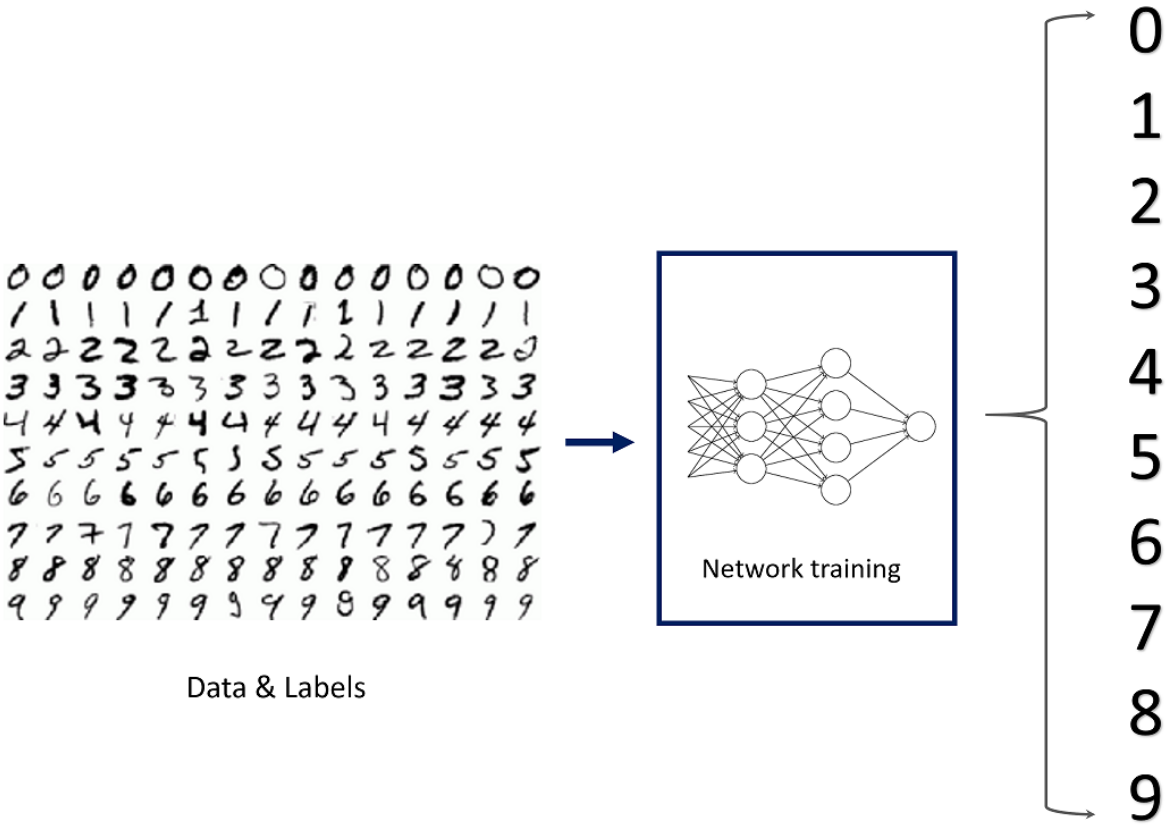


Computer Vision Project Report

Image Classification on Mnist Dataset using CNN

Sajal Rastogi
2018KUCP1132

Introduction

In this project we will learn about how to create a deep neural network such as Convolutional neural network for image classification from scratch with the help of mathematical libraries such as numpy.

Observations

1. CNN drastically decreases the number of parameters
2. Increasing the number of neurons in the hidden layer increases the time for training drastically.
3. For simplicity I have trained the model 3 times for 50 epochs with 20 neurons in hidden layer 1 and 10 neurons in the final layer.
4. Loss was constantly decreasing so one can train model for more iterations.
5. Accuracy on training set after 3x50 iterations: 64.3
6. Accuracy on test set: 59%

Dataset

For simplicity and small size of image i am using a mnist digit dataset.

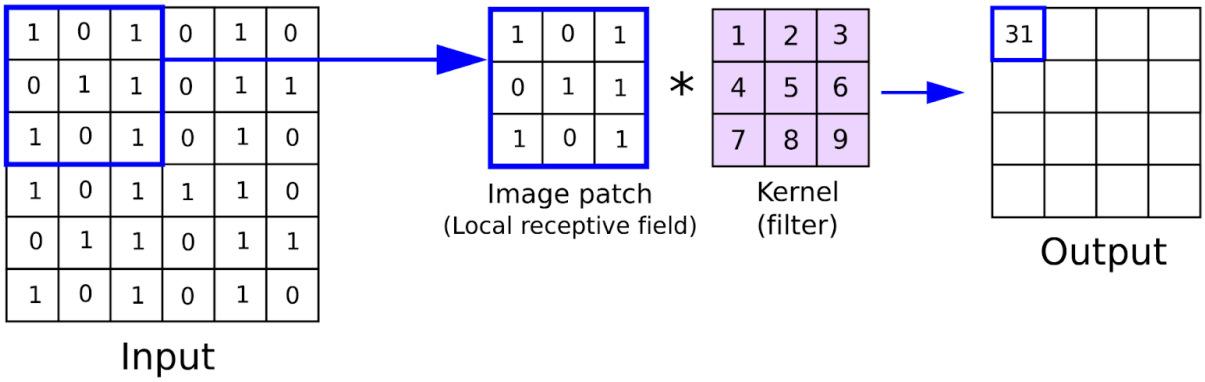
```
# We only use the first 1k examples of each set in the interest of time.
train_images = mnist.train_images()[:1000]
train_labels = mnist.train_labels()[:1000]
test_images = mnist.test_images()[:1000]
test_labels = mnist.test_labels()[:1000]
```

Libraries

1. Numpy
2. mnist

Components for CNN

Convolution Layer



A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

Implementation of Convolution Layer

We will be creating a class for this layer we need to do following operations using same Convolution object:

1. Initialize layer
2. Forward pass
3. backprop

1. Initialization

Set the filters and size of filters.

```
def __init__(self, num_filters):
    """
     num_fileters: total number of filters
     filters: 3d array with dimentions (num_filters, 3, 3)
    """
    self.num_filters = num_filters
    self.filters = np.random.randn(num_filters, 3, 3) / 9
```

2. Forward Pass

Iterate regions: creates a window/region where each filter is going to be multiplied. Creates a blue box type resion as in the above image .

```
def get_regions(self, image):
    """
    # image: matrix of image
    # getting regions for filters to convolve
    """
    h, w = image.shape
    regions_list = []
    for i in range(h - 2):
        for j in range(w - 2):
            im_region = image[i:(i + 3), j:(j + 3)]
            regions_list.append([im_region,i,j])
    return regions_list
```

Forward : function for forward pass of filters over the input matrix. Output contains final feature map after convolution.

For our case size will be:

28x28x1 → 26x26x8 (height,width,num_filters)
Image feature map

```
def forward(self, input):
    """
    Forward pass for convolution layer
    input: input matrix
    return: output after convolution operation
    """

    self.last_input = input
    h, w = input.shape
    output = np.zeros((h - 2, w - 2, self.num_filters))
    regions_list = self.get_regions(input)
    for im_region, i, j in regions_list:
        output[i, j] = np.sum(im_region * self.filters, axis=(1, 2))

    return output
```

3. Backward Pass

This function takes in 2 arguments :
d_out: gradient loss
Lr : learning rate.

We need to update filters according to the type of feature which are necessary for our model. So we are again cutting these regions out and from gradient that we have received we are trying to find out → d_out/d_filter

New_filter = old_filter - lr* d_out/d_filter.

```
def backprop(self, d_out, lr=0.001):
    """
    d_out: gradient loss
    lr: learning rate
    return: None
    """
    d_filters = np.zeros(self.filters.shape)
    regions_list = self.get_regions(self.last_input)
    for im_region, i, j in regions_list:
        for f in range(self.num_filters):
            d_filters[f] += d_out[i, j, f] * im_region

    # Update filters
    self.filters -= lr * d_filters
    return None
```

Activation Layer

These layers are used in order to add more non linearity and make model more robust to data that can not separated linearly.

Implementation

```python
class Sigmoid:
    """
    class for calculating forward and backward pass
    with sigmoid as activation function
    """
    def __init__(self):
        self.lastf = None

    def sigmoid(self,x):
        return 1/(1+np.exp(-x))

    def forward(self,x):
        self.last = self.sigmoid(x)
        return self.last

    def backprop(self,x):
        sig = self.sigmoid(self.last)
        return x * sig * (1 - sig)
```

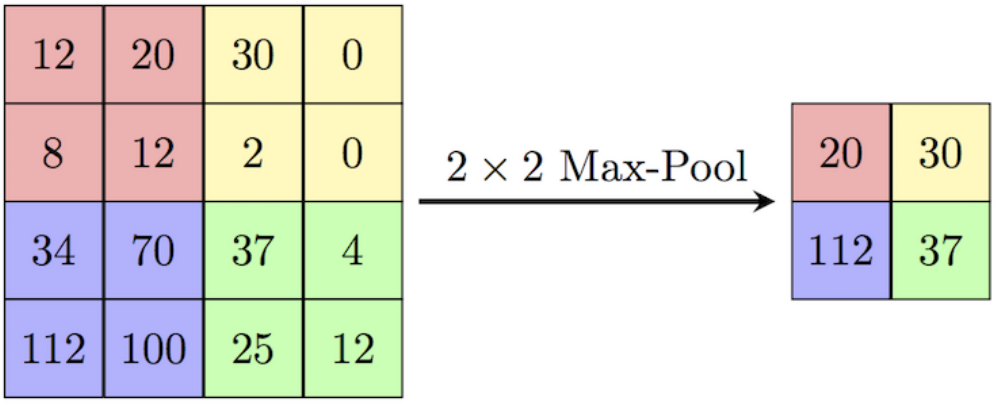
Calculations for derivative of sigmoid:

$$
\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] = \frac{d}{dx}(1+e^{-x})^{-1} \\
&= -1*(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{-e^{-x}}{-(1+e^{-x})^2} \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\frac{e^{-x}+(1-1)}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\left[\frac{(1+e^{-x})}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right] \\
&= \frac{1}{1+e^{-x}}\left[1 - \frac{1}{1+e^{-x}}\right] \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}
$$

Pooling Layer

- Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.
- The pooling layer summarizes the features present in a region of the feature map generated by a convolution layer.

We are going to use **MAXPOOL LAYER.**



Implementation of MaxPool Layer

1. Forward Pass

```
def forward(self, input):
    '''
    Performs a forward pass of the maxpool layer using the given input.
    Returns a 3d numpy array with dimensions (h / 2, w / 2, num_filters).
    input is a 3d numpy array with dimensions (h, w, num_filters)
    '''

    self.last_input = input
    h, w, num_filters = input.shape
    output = np.zeros((h // 2, w // 2, num_filters))

    regions_list = self.get_regions(input)
    for im_region, i, j in regions_list:
        output[i, j] = np.amax(im_region, axis=(0, 1))

    return output
```

As in the above image of maxpool we can see that the size of the image is halved when maxpool is applied. We are doing the same with our input. We first get the regions and find max in that region.

2. Backprop

```
def backprop(self, d_out):
    """
    backprop
    d_out: incoming gradient loss
    return: outgoing gradient loss
    """

    d_input = np.zeros(self.last_input.shape)
    regions_list = self.get_regions(self.last_input)
    for im_region, i, j in regions_list:
        h, w, f = im_region.shape
        amax = np.amax(im_region, axis=(0, 1))

        for i2 in range(h):
            for j2 in range(w):
                for f2 in range(f):
                    if im_region[i2, j2, f2] == amax[f2]:
                        d_input[i + i2, j + j2, f2] = d_out[i, j, f2]

    return d_input
```

d_out: gradient loss that need to be propagated.

To calculate loss in this layer for propagation to before layers we need to only consider max values since only they are involved in giving the output.

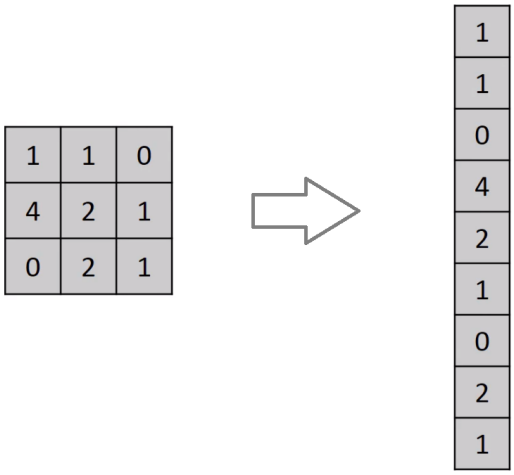
So mathematically it would look somewhat like :

New matrix loss == 0 for non max value
Gradient value for max values.

Above code is replication of this mathematics

Flatten Layer

Converts a x-d output to a single vector of m-size.



Dense Layer / FullyConnected Layer as Hidden layer

Operations done till now are kind of preprocessing steps for converting a image with too many features to a flatten array which can be fed in input to a neural network that can be used to predict the class of each image.

Implementation of Dense Layer

1. Initialization
2. Forward pass
3. backprop

1. Initialization

We need to initialize some variable before we can start creating this layer.

Number of neurons that will be there in this layer.
Weights that this layer will be holding.
Biases
Input

2. Forward Pass

Mathematics:
Output of a Neuron - : activation_function(wx+ b)

For our problem we are using sigmoid as activation function and we have weight matrix of 2d shape so that can be represented as:

$$z1 = w.T * x + b$$
$$a1 = 1/(1 + e^{-z1})$$

For our case size of weight matrix : 1352x20

Bias	: 20
Z1	: 20
A1	: 20

```
def forward(self,input):
    """
    forward pass for normal neural network
    input: input matrix
    return: a = sigmoid(w.T * x+b)
    """
    self.lastinputshape = input.shape
    input = input.flatten()
    self.lastinput = input
    z1 = np.dot(self.weights.T,input) + self.biases
    self.total = z1
    a1 = 1/(1+np.exp(-z1))
    self.last = a1
    return a1
```

3. Backward Pass

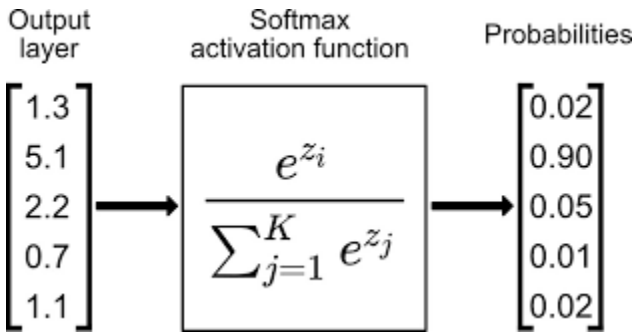
d_out: gradient loss
Lr: learning rate

Mathematical aspects of this part will be explained in later sections

```
def backprop(self,d_out,lr):
    """
    Backprop for dense layer
    d_out: incoming gradient losss
    lr: learning rate
    """
    x = self.total
    dervative = (np.exp(-x))/((np.exp(-x)+1)**2)

    dtd = d_out*dervative
    db = 1
    dldw = np.matmul(self.lastinput[np.newaxis].T,dtd[np.newaxis])
    dldb = dtd

    dldinp = np.matmul(self.weights,dtd)

    # updating weights and biases
    # w = w - lr*d/dw
    # b = b - lr*d/db
    self.weights -= lr*dldw
    self.biases -= lr*dldb

    return dldinp.reshape(self.lastinputshape)
```

Dense Layer / FullyConnected Layer for output:

This layer has softmax function as its activation function instead of sigmoid for bringing the probability of each class in between 0 and 1. Number of output neurons depends upon number of labels in dataset.



Implementation of Dense Layer Softmax:

1. Initialization

   We will be need to initialize weights and biases as we did in previous section. .

2. Forward pass

   Input : matrix from previous layers
   Mathematical view of this layer:

   $z2 = w.T * x + b$
   Then this z2 is passed in the below expression

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for j = 1,...,K}$$

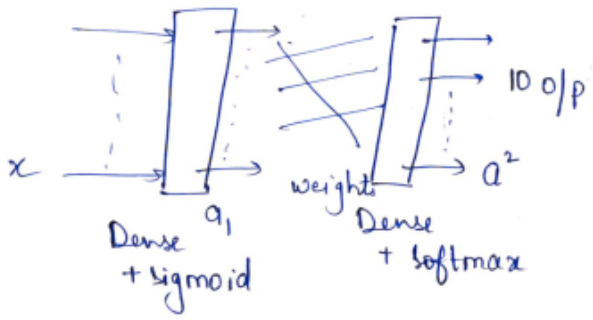
```
def forward(self, input):
    '''
    Performs a forward pass of the softmax layer using the given input.
    Returns a 1d numpy array containing the respective probability values.
    - input can be any array with any dimensions.
    '''
    self.last_input_shape = input.shape
    self.last_input = input
    input_len, nodes = self.weights.shape
    z2 = np.dot(input, self.weights) + self.biases
    self.last_totals = totals

    # activation function : softmax
    exp = np.exp(z2)
    return exp / np.sum(exp, axis=0)
```

3. Backward pass

```
def backprop(self, d_out, lr):
    for i, gradient in enumerate(d_out):
        if gradient == 0:
            continue
        t_exp = np.exp(self.last_totals)

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

        # changing only set value
        d_outdz = -t_exp[i] * t_exp / (S ** 2)
        # change the value of k == c
        d_outdz[i] = t_exp[i] * (S - t_exp[i]) / (S ** 2)

        #derivative
        d_outdw = self.last_input
        d_outdb = 1
        d_outdinp = self.weights

        # Gradients of loss against totals
        dg = gradient * d_outdz

        d_L_d_w = np.matmul(d_outdw[np.newaxis].T , dg[np.newaxis])
        d_L_d_b = dg * d_outdb
        d_L_d_inputs = np.matmul( d_outdinp , dg)

        # Update weights / biases
        self.weights -= (lr * d_L_d_w)
        self.biases -= (lr * d_L_d_b)

        # it will be used in previous pooling layer
        # reshape into that matrix
        return d_L_d_inputs.reshape(self.last_input_shape)
```

Let's Look at the math of backpropagation with help of below image.

For our case since we have 2 layers →.



For Layer 1 →

$z^1 = w^1 \cdot T x + b^1$

$a^1 = \text{sigmoid}(z^1)$

for layer 2 →

$z^2 = w^2 \cdot T \cdot a^1 + b^2$

$a^2 = \text{softmax}(z^2)$.

$\hat{y} = a^2$

Now for backprop of layer 2 →

Cost function $= \Sigma -y \log \hat{y}$

$$\frac{\partial \text{Cost}}{\partial w^2} = \frac{\partial \text{cost}}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z^2} * \frac{\partial z^2}{\partial w^2}$$

$$\frac{\partial \text{Cost}}{\partial \hat{y}} = \Sigma - \frac{y}{\hat{y}}$$

Since our classes our all mutually exclusive gradient would look like [0 0 . . . . 1-4 . . . . 0].

$$\frac{\partial \hat{y}}{\partial z^2} = \begin{cases} \text{softmax}(z^2_i) * (1 - \text{softmax}(z^2_j)) & i = j \\ -\text{Softmax}(z^2_i) * \text{softmax}(z^2_j) & i \neq j \end{cases}$$.

$$\frac{\partial z^2}{\partial w^2} = a^T$$

Similarly → $\frac{\partial z^2}{\partial b} = 1$ $\frac{\partial z^2}{\partial input} = w$

$$\frac{\partial cost}{\partial w^2} = \frac{-y}{\hat{y}} * \frac{\partial \hat{y}}{\partial z^2} * a^T$$

Similarly = $\frac{\partial cost}{\partial b}$ = → $*1$ $\frac{\partial cost}{\partial input}$ $* w$

For layer 1 →

$$\frac{\partial cost}{\partial w^1} = \frac{\partial C}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z^2} * \frac{\partial z^2}{\partial a^1} * \frac{\partial a^1}{\partial z^1} * \frac{\partial z^1}{\partial w^1}$$

Since we have calculated this part already we can send this to backward direction

$$\frac{\partial a^1}{\partial z^1} = sig(z^1) * (1 - sig(z^1))$$

Similarly we can have

$$\frac{\partial cost}{\partial w^1}, \frac{\partial cost}{\partial b}, \frac{\partial cost}{\partial x}.$$

Defining the objects of each class

```
conv = ConvLayer(8)                          # 28x28x1 -> 26x26x8
act = Sigmoid()
pool = MaxPool()                             # 26x26x8 -> 13x13x8
dense = Dense(20,13 * 13 * 8)                # 13x13x8 -> 20
softmax = Softmax(20, 10)                    # 20 -> 10
```

Compiled Model Forward Step

Each Forward Step Conveys:

Convolution layer → Activation Layer → MaxPool Layer → Dense Layer (Sigmoid) → Dense Layer (SoftMax)

```
def forward(image, label):
    '''
    Completes a forward pass of the CNN and calculates the accuracy and
    cross-entropy loss.
    - image is a 2d numpy array
    - label is a digit
    '''
    out = conv.forward((image / 255))
    out = act.forward(out)
    out = pool.forward(out)
    out = dense.forward(out)
    out = softmax.forward(out)

    # Calculate cross-entropy loss
    loss = -np.log(out[label])
    acc = 1 if np.argmax(out) == label else 0

    return out, loss, acc
```

Compiled Model Backward Step

We need to propagate the error in exactly the reversed way as that error has reached the final layer.

Gradient error/loss will be propagated in reverse fashion so it will go from:

Softmax layer → dense layer → MaxPool Layer → Activation Layer → Convolution Layer

Initial gradient passed to softmax layer will look like below:

Since labels can be from 0 to 9: vector size = 10

[0 , 0 , 0, 0 , 0 , 1.4 , 0 , 0 , 0, 0]

Why only one position is non zero: because cross entropy error will be 0 for all the labels which are not activated and will be activated only for 1 label.

```
def backward(gradient,lr):
    """
    Backward Controller for entire model
    gradient: loss
    lr: learning rate
    """
    gradient = softmax.backprop(gradient, lr)
    gradient = dense.backprop(gradient,lr)
    gradient = pool.backprop(gradient)
    gradient = act.backprop(gradient)
    gradient = conv.backprop(gradient, lr)
```

Train Function for controlled forward and backward Steps

Im: image matrix

Label: class of image (0-9)

Lr: learning rate

```
def train(im, label, lr=0.01):
    """
    Controller for forward , backward and loss of entire model
    im: image matrix
    label: label/class of image
    lr: learning rate
    """
    out, loss, acc = forward(im, label)
    gradient = np.zeros(10)
    gradient[label] = -1 / out[label]
    backward(gradient,lr)
    return loss, acc
```

Training Model 150 epochs.

l : loss of each sample
Loss: sum of all loss for each sample
Acc: correct = 1 , incorrect = 0
For each sample we are training our model and trying to find loss and accuracy of our model.

```
for epoch in range(150):
    print('--- Epoch %d ---' % (epoch + 1))

    loss = 0
    num_correct = 0
    for i, (im, label) in enumerate(zip(train_images, train_labels)):
        l, acc = train(im, label)
        loss += l
        num_correct += acc
    print(loss/1000,num_correct/1000)
```

Snapshots During Training

```
--- Epoch 1 ---
2.2744832082039 0.161
--- Epoch 2 ---
2.272352411588011 0.164
--- Epoch 3 ---
2.2699872764662654 0.168
--- Epoch 4 ---
2.267356193957076 0.173
--- Epoch 5 ---
2.264422607489012 0.177
--- Epoch 6 ---
2.261144863863248 0.187
--- Epoch 7 ---
2.25747455873526 0.197
--- Epoch 8 ---
2.2533554701760736 0.207
--- Epoch 9 ---
2.2487233559204616 0.223
--- Epoch 10 ---
2.2435051802386643 0.236
--- Epoch 11 ---
2.237618505307607 0.244
--- Epoch 12 ---
2.230966152300031 0.252
```

```
--- Epoch 36 ---
1.6756619553526262 0.536
--- Epoch 37 ---
1.645224171198259 0.551
--- Epoch 38 ---
1.615528582255598 0.554
--- Epoch 39 ---
1.5866020577154762 0.565
--- Epoch 40 ---
1.5584487817393353 0.57
--- Epoch 41 ---
1.531063698431820 0.583
--- Epoch 42 ---
1.5044326697658623 0.592
--- Epoch 43 ---
1.4785346330011662 0.604
--- Epoch 44 ---
1.4533493160944286 0.613
--- Epoch 45 ---
1.4288549538337116 0.619
--- Epoch 46 ---
1.4050299089074236 0.627
--- Epoch 47 ---
1.3818545148312866 0.633
--- Epoch 48 ---
1.3593085718928242 0.639
--- Epoch 49 ---
1.3373735673667744 0.642
--- Epoch 50 ---
1.3160325339517702 0.643
```

We can see that loss was constantly decreasing which shows that we can train for more epochs.

Test Accuracy

```
loss = 0
num_correct = 0
for im, label in zip(test_images, test_labels):
    _, l, acc = forward(im, label)
    loss += l
    num_correct += acc

num_tests = len(test_images)
print('Test Loss:', loss / num_tests)
print('Test Accuracy:', num_correct / num_tests)
```

```
Test Loss: 1.3955537269638751
Test Accuracy: 0.591
```

THANK YOU