Neural Network for Handwritten Digits Recognition

Contents

1 Introduction

Neural networks are used as a method of deep learning, one of the many subfields of artificial intelligence. The main idea is to simulate the way the human brain works, though ~~in a~~ much more simplified form. Individual 'neurons' are connected in layers, with weights assigned to determine how the neuron responds when signals are propagated through the network. With three types of layers and complicated connection between each neuron, we are able to let our network find an appropriate way to deal with a kind of classification problem. We only need to set up those layers like playing LEGO, and input a certain amount of data for the network to learn. Finally, the network is going to achieve in classifying any randomly data input. In this article, we will implement a small subsection of object recognitionHandwritten Digit Recognition, using **PyTorch**, an open source machine learning framework that accelerates the path from research prototyping to production deployment. We are going to take hand-drawn images as input of the numbers 0-9 and build and train a neural network to recognize and predict the correct label for the digit displayed.

2 Prerequisites

Here we will need some basic knowledge about mathematics.

- Linear Algebra (Dot product, Matrix multiplication)
- Permutation and Combination
- Series of numbers
- Boolean

Moreover, let us introduce three main types of layers in Neural Network.

2.1 Fully-Connected Layer

Fully connected layers are an essential component of Convolutional Neural Networks (CNNs), which have been proven very successful in recognizing and classifying images for computer vision. In this layer, every 'neurons' is connected with its input and output 'neurons'. The operation in this layer is first multiplying the weight with the input data, and then we usually let the result go through a function named **Activation Function** before passing to the next 'neurons'.

ReLU, **Sigmoid**, **tanh**, **Linear(null)** ~~are~~ all activation function. But we usually use ReLU, because it is the most effective. The function represents as follow:

$$\mathrm{ReLU}(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

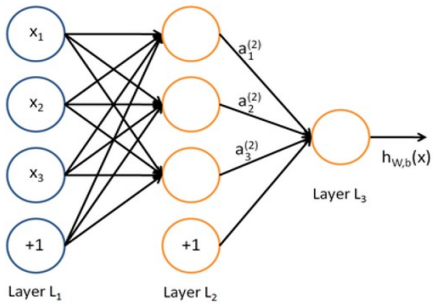

Figure 1: Fully-connected Layers

Example 2.1. As in Figure 1, we accomplish the following calculation through those layers:

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + {}_1b^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + {}_2b^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + {}_3b^{(1)})$$

$$h_{W,b}(x) = a_1^{(3)} = f({}_1W_{11}^{(2)}a^{(2)} + {}_1W_{12}^{(2)}a^{(2)} + {}_1b^{(2)})$$

2.2 Convolution Layers

Before introducing the convolution layers, it would be better to state some definitions.

Deftnition 2.1. (kernel) Each convolution operation has a kernel which could be a any matrix smaller than the original image in height and width. Kernel is a function, whose elements are all weights.

Deftnition 2.2. (Convolution) Steps of convolution in the convolution layers in details are described as follows:

- Put the first element of the kernel at every pixel of the image (element of the image matrix). Then each element of the kernel will stand on top of an element of the image matrix.
- Multiply each element of the kernel with its corresponding element of the image matrix
- Sum up all product outputs and put the result at the same position in the output matrix as the center of kernel in image matrix.

Remark. For the pixels on the border of image matrix, some elements of the kernel might stands out of the image matrix and therefore does not have any corresponding element from the image matrix.
In this case, you can eliminate the convolution operation for these position which end up an output matrix smaller than the input (image matrix) or we can apply padding to the input matrix (based on the size of the kernel we might need one or more pixels padding.

Example 2.2. The following example is about the first case, where we do not apply padding to the input matrix. Our kernel keep moving from left to right and from up to down. Usually, we call this moving as 'Window Sliding'. The default stride is 1, but we are able to change the stride.
The example shows in Figure 2.

Example 2.3. The following example is about the second case, where we apply padding to the input matrix. Although in this example we add zero padding on the left of the input matrix, we usually add them on the right.
The example shows in Figure 3.

If the size of the input imagine is large, we need to extract patches from the original picture, and do convolution with each patch. The kernel (or filter or feature detector) only looks at one chunk of an image at a time, then the filter moves to another patch of the image, and so on.

Deftnition 2.3. (patch) Patches are partitions of the original image matrix. We are going to averagely partition the imagine matrix into blocks, and each block is one patch. It is the input to the kernel.

Remark. Kernels (or filters) only process one patch at a time, rather than the whole image. This is because we want filters to process small pieces of the image in order to detect features (edges, etc). This also has a nice regularization property, since we're estimating a smaller number of parameters, and those parameters have to be 'good' across many regions of each image, as well as many regions of all other training images.

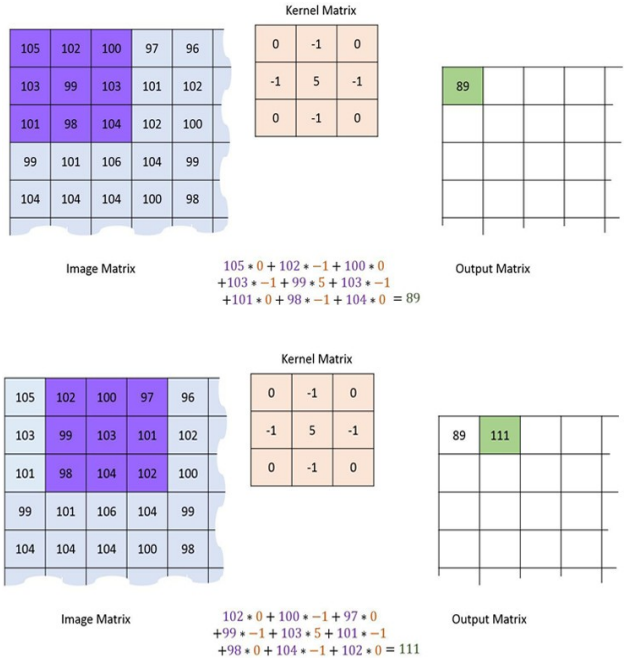

Figure 2: Convolution Calculation

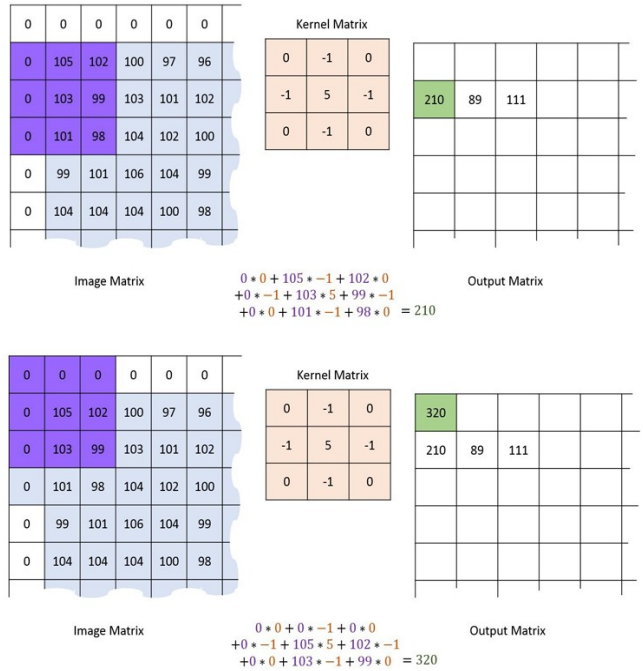

Figure 3: Convolution Calculation on borders

2.3 Dropout Layers

Normally some deep learning models use Dropout on the fully connected layers. Actually, it is a kind of fully connected layer. More capacity we add on our model (More layers, or more neurons) more prone to over-fit it becomes. Dropout is a technique used to improve over-fit on neural networks, we should use Dropout along with other techniques like L2 Regularization.

In Figure 4, we can see that there is 'p=0.5'. This is the probability of each neuron being thrown away. From the expectation formula, we know that about half of neurons are going to exist.

$$E(X) = \sum_{k=1} x_k p = \frac{1}{2} n \, , \, x_k = 1$$

We can also change the value of p, but the default value is 0.5. Since during training we randomly choose some neurons and let them be deactivated. This improve generalization because we force our layer to learn with different neurons the same 'concept'.

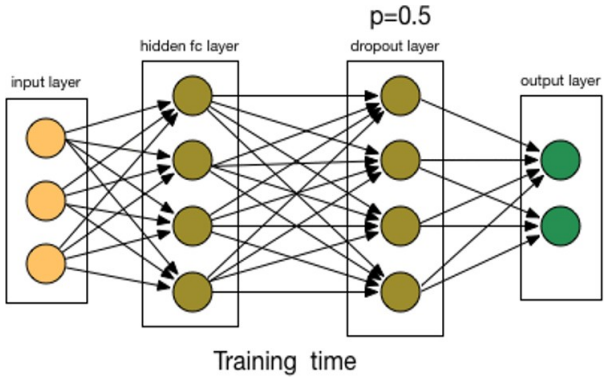

Figure 4: Dropout Layer

3 Methodology

3.1 Stochastic Gradient Decent(SGD)

Stochastic Gradient Decent is an approximation for gradient decent. If we have to use every data in our training set in every iteration,the speed of the iteration will become very slow when the size of data is large. Hence, we come up with the idea of stochastic gradient decent.

To distinguish between right and wrong after we get the output results from our network, we need to compare them with our right answers. Hence, we define the 'Cost Function' to value the output results.

Deftnition 3.1. (Cost Function)

$$J(W, b; x, y) = \frac{1}{2} \| h_{W,b}(x) - y \|^2$$

Where $h_{W,b}(x)$ is the output of our neural network (which is a predicted value from parameters W and b), y is the correct answer of output, W is weight.

Deftnition 3.2. (Cross Entropy) In logistic regression, the most commonly used cost function is cross entropy.

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}(y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

Where m is the number of training data, the meanings of other symbols are similar to the previous definition.

Deftnition 3.3. (Cost Function in Neural Network) Cost Function in Neural Network is very similar to the cost function in the logistic regression. Logistic regression is a special case of neural network(network without hidden layers).

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K}(y_k^{(i)}\log h_\theta(x^{(i)}) + (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)]$$

Where K denotes the number of classes in multiple categories.
There is one more summation term in the formula because the outputs of neural network are not single values in general.

In the SGD algorithm, we will not focus on the whole training set during each training. Nevertheless, we randomly choose a part of training set (sample) to calculate the cost function, and then randomly choose another sample in the remaining training set for the next training. We are going to repeat this process until the training set becomes empty. And then repeat this whole process over and over again.

Deftnition 3.4. (Batch) In gradient descent, a batch is the total number of examples you use to calculate the gradient in a single iteration. So far, we' ve assumed that the batch has been the entire data set.

Remark. We usually set powers of two for the batch size, because CPU is going to perform better when dealing with this kinds of numbers.

3.2 L2 Regularization

In machine learning, if there are too many parameters or the model is too complicated, it is easy to cause overfiting.

Deftnition 3.5. overftt The neural network can almost perfectly apply classification on the training data after training, but behave bad when dealing with the test data in practical. Which means the network has the problems of bad training approximation and generalization.

Deftnition 3.6. (L2 Regularization) L2 Regularization can be viewed as the penalty term of cost function, it gives restrictions for some parameters in cost function.

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

Where C_0 denote the original cost function, λ is the regularization coefficient. The second term is L2 regularized item, which is the sum of all w^2, divided by the size of the training sample set n. (The coefficient $\frac{1}{2}$ is just for convenient when taking derivation.)

Let us prove that why L2 regularization can avoid overfitting.

Proof. Take the partial derivative of w and b respectively:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

We find that L2 regularized item has no effect on the update of b, but it has effect on the update of w.

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w$$
$$= (1 - \frac{\eta\lambda}{n}) w - \eta \frac{\partial C_0}{\partial w}$$

Before using L2 regularization, the coefficient of w in the derivative result is 1, but now it is $1 - \frac{\eta\lambda}{n}$. Because η, λ, n are all positive, $1 - \frac{\eta\lambda}{n}$ less than 1. The effect is reducing, and this is exactly why weight decay. For smaller weights, they lower the complexity of the neural network.

Remark. In other words, weights decay will push weights to zero during training.

For the stochastic gradient decent based on mini batch, the formula for update of w and b are little different.

$$w \to (1 - \frac{\eta\lambda}{n}) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}$$
$$b \to \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$

3.3 Analyzing Code

Importing classes and models for constructing our neural network.

- **_future** Here we are going to import **print_function** from this toolbox, so that we can use the print function in Python3 when we are in Python2 environment.
- **argparse** The argparse module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires.
- **torch** It provides us with lots of classes and models to help us construct and train our neural network. The torch package contains data structures for multi-dimensional tensors and mathematical operations over these are defined. Additionally, it provides many utilities for efficient serializing of Tensors and arbitrary types, and other useful utilities.

Then we need to set some **Hyper Parameters**. We can change these parameters for training in order to improve our neural network afterwards.

Deftnition 3.7. (epoch) The number of epochs is the number of times you go through the full data set. We usually set it as 50, it cannot be too small.

3.3.1 Step2: Importing the MNIST Dataset

The MNIST data set, and it is a classic in the machine learning community. This data set is made up of images of handwritten digits, 28*x*28 pixels in size.

We can use **matplotlib.pyplot** to plot some picture in this data set. See figure 5 and 6.

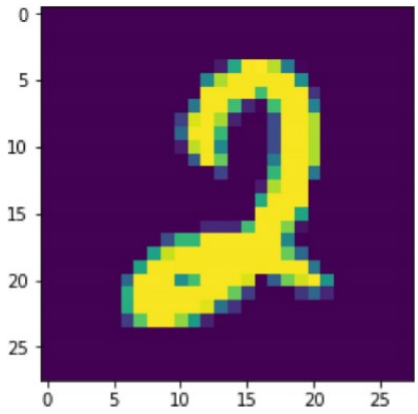

Figure 5: Handwritten digit ' 2'

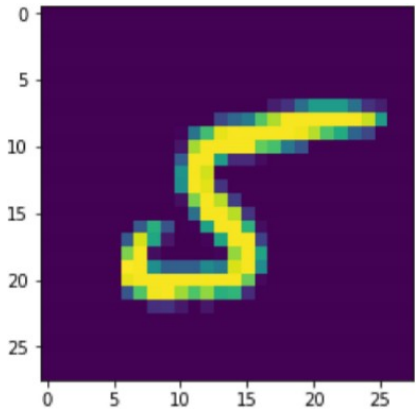

Figure 6: Handwritten digit ' 5'

In latex, it should be `2', instead of '2'

3.3.2 Step3: Deftning the Neural Network Architecture

The architecture of the neural network refers to elements such as the number of layers in the network, the number of units in each layer, and how the units are connected between layers. Different architectures can yield dramatically different results, we need to improve our network by trying different architectures.

We consider our architecture as:

$$input - - > conv1 - - > maxpool2d - - > relu - - > conv2 - - > dropout - - > maxpool2d - - > relu - - > view(-1, 320) - - > fc1 - - > relu - - > dropout - - > fc2 - - > output$$

- **conv1** and **conv2** are both convolution layers.
- **maxpool2d** is going to take the maximum value in every sub-matrix to create a smaller matrix.
- **relu** is an activation function, which we have introduced before.
- **dropout** is dropout layers, we use the default $p = 0.5$.
- **view** is meant to reshape the tensor. Here we need our data to be one dimensional.
- **fc1** and **fc2** are both fully connected layers.

Let us print out those parameters we have set as follows:

```
model = Net()
print(model)
```

```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Figure 7: Parameters in our Neural Network

4 Results

See the attach files for fully code and detailed output.

4.1 Experiment One

In the first experiment, we set the **batch size** to be 256, the **test batch size** to be 1000, and the **epochs** to be 50. The final result is showed in Figure 8:

```
Test set: Average loss: 0.0423, Accuracy: 9865/10000 (98%)
```

Figure 8: Parameters in our Neural Network

4.2 Experiment Two

In the second experiment, we set the **batch size** to be 512, the **test batch size** to be 1000, and the **epochs** to be 50. The final result is showed in Figure 9:

```
Test set: Average loss: 0.0605, Accuracy: 9803/10000 (98%)
```

Figure 9: Parameters in our Neural Network

Reference

@miscdigitalocean.com, title = How To Build a Neural Network to Recognize Handwritten Digits with TensorFlow, author = Ellie Birbeck, howpublished = https://www.digitalocean.com/community/tutorials/how-to-build-a-neural-network-to-recognize-handwritten-digits-with-tensorflow

@miscmachinelearninguru.com, title = Image Filtering, howpublished = http://machinelearninguru.com/computer_vision/basics/convolution/image_convolution_1.html

@miscleonardoaraujosantos.gitbooks.io, title = Dropout Layer, howpublished = https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/dropout_layer.html

@miscleonardoaraujosantos.gitbooks, title = Stochastic Gradient Descent, howpublished = https://developers.google.com/machine-learning/crash-course/reducing-loss/stochastic-gradient-descent