A simple CNN model on Fashion MNIST dataset

Tam King Man 1155160072 Lai Cheuk Lam 1155159309

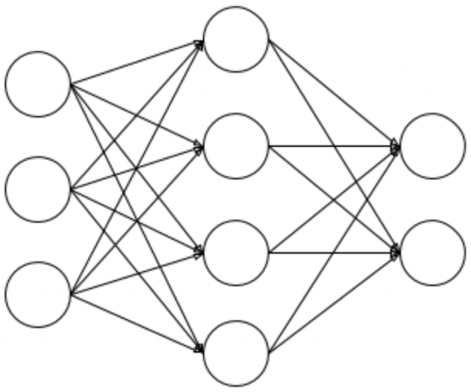
1. Background

Our project aims to provides basic deep learning background for readers to gain a brief overview of deep neural network. Topic includes the neural network architecture, Multilayer Perceptron, backpropagation, activation functions, CNN architecture, convolution, pooling, flattening, dropout, stochastic gradient descent. A working example for image classification on Fashion MNIST dataset using CNN is also illustrated in the end to provide an insight on how neural networks really works.

2. Deep Neural Network

To simulate how the brain works, the idea of neural network is proposed. In a neural network, we have input layer (the first layer), multiple hidden layers and output layer (the last layer). Each layer is composed with some neurons. The neurons in the layer i may be connect to the neurons in the layer $i + 1$ by some weighted edges. When the number of hidden layer increases, human may be difficult to follow what happens among the hidden layers and this is how the term "deep" comes from.

The most basic class of deep neural network is Multilayer Perceptron (MLP), it is a fully connected acyclic feedforward neural network. Feedforward means that the information moves in one direction through the hidden layer. Such a network might be drawn as follows:



We can formulate it in mathematical way:

Let x be the input, W_l be the weights in layer l, σ_l be the activation function in layer l. Denote the number of layers (except input layer) by L, the number of neurons in layer $l \in \{0, \dots, L\}$ by n_l. Note that $\sigma_l: \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ and $W_l \in \mathbb{R}^{n_l \times n_{l-1}}$ for $l \in \{1, \dots, L\}$. Hence, we have

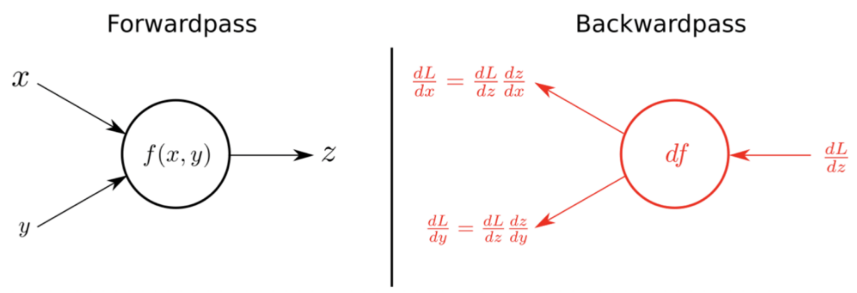
$$x \longmapsto \sigma_L(W_L \sigma_{L-1}(W_{L-1}(\dots \sigma_2(W_2 \sigma_1(W_1 x))))).$$

By this network, we may mimic the behavior of our brain. We only have one problem; how can we find the weights among each layer? We want to minimize the cost function E in weight space by gradient descent. To do gradient descent efficiently, we have backpropagation algorithm, which is an efficient use of Chain rule for derivatives.

3. Backpropagation

Backpropagation algorithm:

1) Initialize network with random weight
2) For all training samples:
    a. Input sample and compute the output (forward pass)
    b. Compare the output with correct output to get loss term
    c. For all layers (backward pass from output layer)
        - Propagate the loss term back to the previous layer
        - Update the weights between the two layers


Forwardpass
Backwardpass
x
y
$f(x,y)$
z
df
$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx}$
$\frac{dL}{dz}$
$\frac{dL}{dy} = \frac{dL}{dz}\frac{dz}{dy}$

Let a_j^l and w_{ij}^l be the value of the neurons j in the layer l and the associated weight from neurons i in the layer $l-1$ to the neuron j in the layer l respectively. Simply put, $a_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^l \sigma_l\left(a_i^{l-1}\right)$. Also, let o_j^l be the output of row j in layer l. Then $o_j^l = \sigma_l\left(a_j^l\right)$.

We first start looking at the output layer. By Chain rule, we have

$$\frac{\partial E}{\partial w_{ij}^L} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial a_j^L}\frac{\partial a_j^L}{\partial w_{ij}^L}$$

$$= \frac{\partial E}{\partial o_j}\frac{\partial}{\partial a_j^L}\left(\sigma_L\left(a_j^L\right)\right)\frac{\partial}{\partial w_{ij}^L}\left(\sum_{i=1}^{n_{L-1}} w_{ij}^L \sigma_L\left(a_i^{L-1}\right)\right) = \frac{\partial E}{\partial o_j}\sigma'_L\left(a_j^L\right)\sigma_L\left(a_i^{L-1}\right)$$

Note that we know a_j^L and a_j^{L-1} in the forward pass and we can compute the derivative of cost function and activation function directly. We know what is $\frac{\partial E}{\partial w_{ij}^L}$.

For the hidden layers,

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a_j^l}\frac{\partial a_j^l}{\partial w_{ij}^l} \triangleq \delta_j^l \frac{\partial a_j^l}{\partial w_{ij}^l} = \delta_j^l \sigma_l\left(a_i^{l-1}\right)$$

$$\delta_j^l = \frac{\partial E}{\partial a_j^l} = \frac{\partial E}{\partial o_j^l}\frac{\partial o_j^l}{\partial a_j^l} = \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^{l+1}\, \sigma_l'\left(a_j^l\right)$$

$$\frac{\partial E}{\partial w_{ij}^l} = \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^{l+1}\, \sigma_l'\left(a_j^l\right)\sigma_l\left(a_i^{l-1}\right)$$

Hence, we only need δ_i^{l+1}, w_{ji}^{l+1}, a_j^L and a_j^{L-1}. Finally, we can update the weight by $w_{ij}^l \leftarrow w_{ij}^l + \alpha \frac{\partial E}{\partial w_{ij}^l}$ where α is the learning rate.

The concept of forward pass and backward pass plays an important role in neural network. We just summary parts of the main ideas and we will go to the next chapter to discuss activation functions.

4. Activation function

When the network passes the features to the next layer, we need some non-linear activation functions to transform the features. Otherwise,

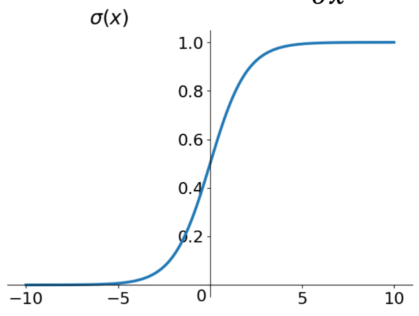
$$T_1\left(T_2\left(T_3(\dots T_n(x))\right)\right) = T(x)$$

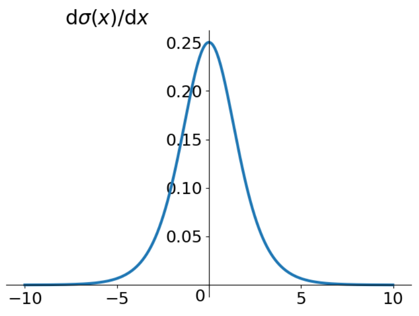
where $T_1, T_2, \dots, T_n$ are linear transformation. In other words, we can view multiple linear transformations as one linear transformation. We hereby discuss some common activation functions, which are non-linear.

- Sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial\sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$





First, we have sigmoid function, which is very common, appear in the logistic regression. However, there are some troubles for using sigmoid function as an activation function. The computational cost of the exponential term is high. Also, the output is not zero-centered, this means that the network may converge slowly. Note we pass $f = \sum w_i x_i + b$ to next layer.

$$\frac{df}{dw_i} = x_i$$

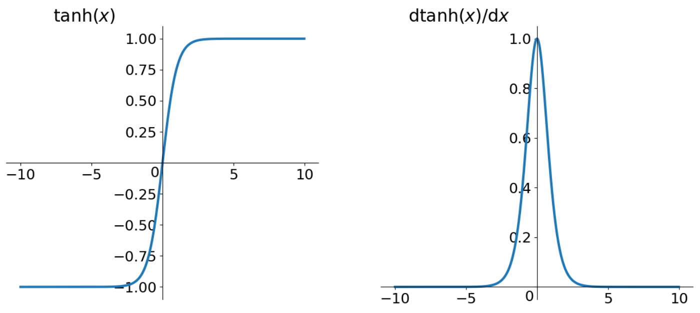
$$\frac{dL}{dw_i} = \frac{dL}{df}\frac{df}{dw_i} = \frac{dL}{df}x_i$$

Since the output always ≥ 0, i.e., $x_i \geq 0$, the gradient $\frac{dL}{dw_i}$ always has the same sign as $\frac{dL}{df}$ (all positive or all negative). Let constraint our layer to two neurons with associated weights w_1 and w_2 and $\frac{df}{dw_i}$ have same sign for $i = 1,2$. It follows that gradient descent can only move in the same direction. This may cause the weight to converge with a zig-zagging behavior and therefore in a slower speed.

Another problem is that the gradient will be vanishing quickly. The convergence speed here means the number of iterations. We can see from the graph of the derivative of sigmoid function (right graph) that $\frac{\partial\sigma(x)}{\partial x} \leq 0.25$. It follows that the gradient tends to 0 quickly when we update weightings by gradient descent. Hence, it is not suggested to use sigmoid function as the activation function.

- Tanh function

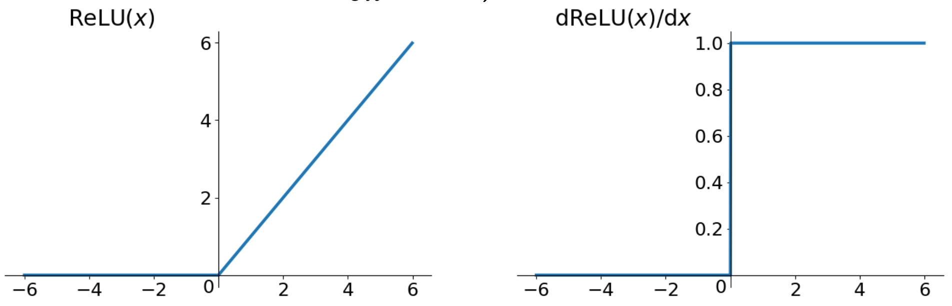
$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$\frac{\partial\,\sigma(x)}{\partial x} = 1 - \sigma^2(x)$$



Secondly, we can see from the graph of $\tanh(x)$ that the output is zero-centered. However, the heavy computation cost of exponential term and gradient vanishing problem still exist. Hence, ReLU function is proposed to solve these problems.

- ReLU function

$$\sigma(x) = \max(0, x)$$
$$\frac{\partial\,\sigma(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$



Reader may notice that the derivative of ReLU is not defined. This problem can be solved easily by some convention. i.e., $\frac{\partial\,\sigma(x)}{\partial x} = 0$ when $x = 0$. ReLU converges quickly and the computation cost is $O(1)$ since the network only need to determine the threshold. Also, it solves the gradient vanishing problem for $x > 0$. Therefore, we usually use ReLU function as the activation function. Specifically, note that ReLU is not zero-centered and sometimes it may cause some dead neurons due to bad weight initialization or high learning rate. People still use it due to its low computation cost. Some functions are proposed based on regular ReLU. For example, we have Leaky ReLU and ELU (Exponential Linear Units).

$$\text{Leaky ReLU: } \sigma(x) = \max(0.01x, x)$$

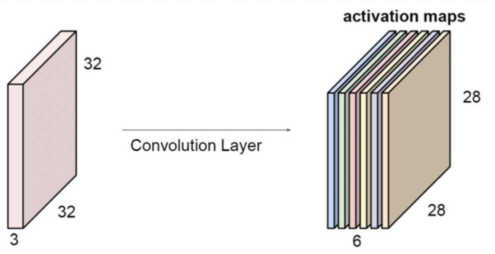
$$\text{ELU: } \sigma(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \le 0 \end{cases}$$

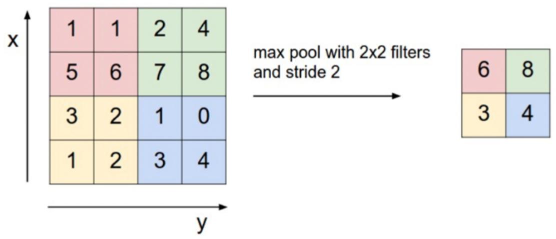
We may skip the details of them and go to CNN. In the working example, we mainly use ReLU as activation function for the neural network.

5. From NN to Convolutional NN

In deep learning, researcher builds a lot of extension on top of the neural network (NN) and develop powerful frameworks, such as Convolutional neural network (CNN). We may discuss it in this chapter. There are many image recognition models are based on the framework of CNN, which is a good starting point to learn different deep learning frameworks. A CNN arranges its neurons in three dimensions, and hence we may obtain a 3D output tensor. It is composed with three components, convolution layer, pooling and fully connected layer.

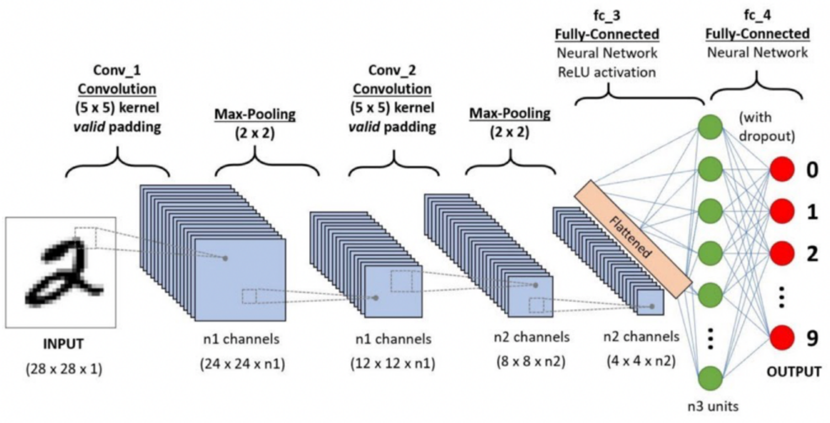
For better understanding, we use an example here.[1] For convolution layer, we have a $5 \times 5 \times 3$ filter and we convolve the filter with a $32 \times 32 \times 3$ dimension input. The filter slide over the image spatially and compute the dot products. Then we have an activation map with dimension $(32 - 5 + 1) \times (32 - 5 + 1) \times 1 = 28 \times 28 \times 1$. Since we may have many filters, we may stack the activation maps together (let's say we have 6 filters) and hence we obtain a $28 \times 28 \times 6$ tensor as the following graph.





To make the layer smaller and more manageable (downsample), Pooling layer can be applied. There are two common implementations of pooling. In max-pooling, the representative value just becomes the largest of all the units in the window, while in average-pooling, the representative value is the average of all the units in the window. Note the volume depth is preserved.

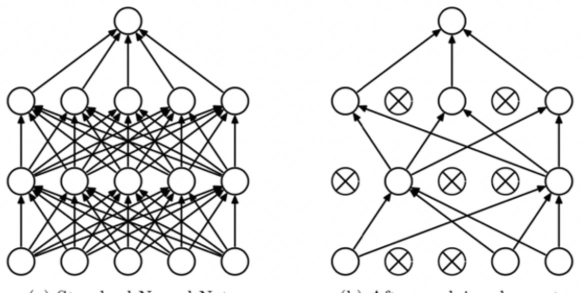
Neurons in a fully connected layer have full connections to all activations in the previous layer, as the regular NN. Their activations can hence be computed with a matrix multiplication followed by a bias offset. It is remined that we should flatten the layer into a feature vector before putting it into the neural network. The following graph is an example of the illustrate convolutional neural network. There are also many examples of CNN, e.g. AlexNet, ResNet, GoogLeNet, etc.



[1] More mathematical detail refers to the lecture note. The formal definition is omitted due to page limits.

6. Dropout

Dropout was proposed by Hinton in 2019 and became famous by the influence of AlexNet. The idea of dropout is simple, prevent neurons from co-adapting too much. In each batch, some neurons may stop forward pass (can be simply done by setting their value as 0) with a given probability p. This may reduce that some local properties dominate the network and make the network more generalized. In another word, this method can prevent overfitting.



The above figure may illustrate how dropout affect the network. We can see figure b become more simplified. Note that the crossed neurons are just removed temporarily. After forward pass, the network will continue do backpropagation on the simplified network. Then we update the uncrossed neurons, recover the crossed neurons and repeat the process. A more formal explanation and implementation may leave to readers due to page limit.

7. Stochastic gradient descent

In this chapter, we briefly introduce what is stochastic gradient descent. Our loss function can be represented as $f(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$. Then the gradient $\nabla f(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w)$. This standard form refers to batch gradient descent (the standard one), because it computes a full "batch" of gradients in each update.

For batch gradient descent (the standard one), we need to update every $\nabla f_i(w)$. It can be computationally expensive, because it requires to compute n gradients. In addition, due to the deterministic nature of the algorithm, it can easily get stuck at local minima and saddle points. On top of that, mini-batch gradient descent is proposed. That is, instead of summing up an entire batch of n gradients, only k gradients are chosen, $k < n$, i.e., $\nabla f(w) = \frac{1}{k}\sum_{i=1}^{k} \nabla f_i(w)$.

In particular, when $k = 1$, we have stochastic gradient descent, namely $\nabla f(w) = \nabla f_i(w)$. We pick $\nabla f_i(w)$ randomly from $\{1, 2, \dots, n\}$ and we verify $\mathbb{E}(\nabla f_i(w)) = \nabla f(w)$.

$$\mathbb{E}(\nabla f_i(w)) = \sum_{j=1}^{n} P(i = j) \nabla f_j(w) = \frac{1}{n}\sum_{j=1}^{n} \nabla f_j(w) = \nabla f(w)$$

Given fixed w, it satisfies $\mathbb{E}(\nabla f_i(w)) = \nabla f(w)$. We can therefore say that the stochastic gradient is an unbiased estimate of the true gradient. It reduces the time complexity from $O(n)$ to $O(1)$. Hence, we have a new weight updating rule:

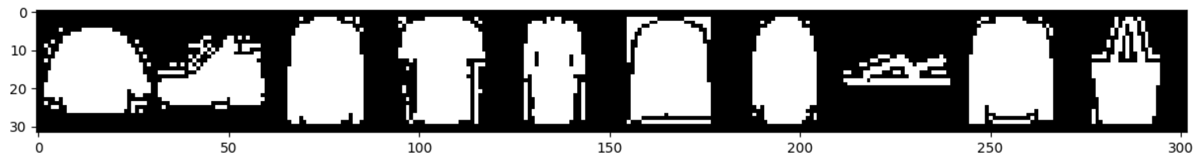
$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_t \nabla \mathrm{G}\left(\mathrm{w}^{(t)}\right)$$

8. Working Example[2]

We try to implement CNN model using Fashion MNIST dataset. Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

Class 0	T-shirt/Top
Class 1	Trouser
Class 2	Pullover
Class 3	Dress
Class 4	Coat
Class 5	Sandal
Class 6	Shirt
Class 7	Sneaker
Class 8	Bag
Class 9	Ankle Boot

We may first illustrate how the dataset looks like.



Now, we introduce our proposed CNN model for classification and discuss some problems we met during the experiments. We set our batch size as 256, epochs as 40 and learning rate as 0.05. After experiments with different setting, the batch size doesn't affect the result a lot. We set epochs as 40 because of the long training time. The network can already achieve amazing result with 40 epochs. The most interesting part here is the learning rate. With different learning rate, our result may differ a lot and converge to 10% eventually.

```
def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3)
        self.dropout = nn.Dropout(p=0.4)
        # fully connected layer
        self.fc1 = nn.Linear(3*3*128, 128)
        self.fc2 = nn.Linear(128, 10)
```

```
def forward(self, x):
         # first conv
         x = self.pool1(F.relu(self.conv1(x)))
         x = self.pool2(F.relu(self.conv2(x)))
```

[2] We have a reference on the dataset accessed on https://www.kaggle.com/code/pavansanagapati/a-simple-cnn-model-beginner-guide. Our model uses Pytorch instead of TensorFlow.

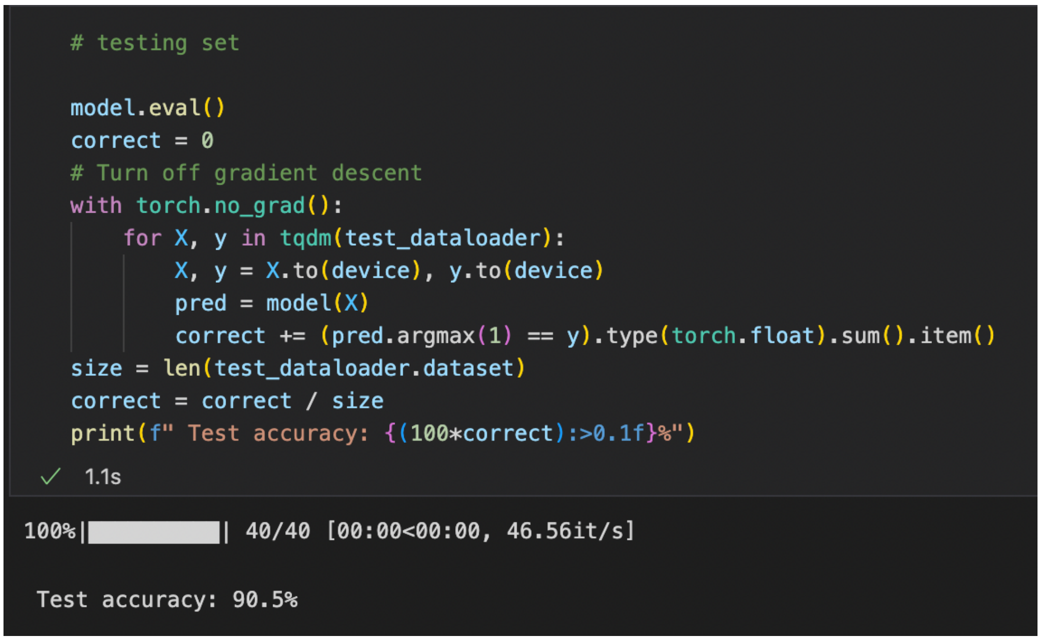
```
        x = F.relu(self.conv3(x))
        x = self.dropout(x)
        # flatten all dimensions except batch
        x = torch.flatten(x, 1)
        # fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.softmax(self.fc2(x))
        return x
```

Note that our input image is of size $28 \times 28 \times 1$. In the first convolution layer, we use 32 $3 \times 3 \times 1$ filters with ReLU function. Then we get a $(28 - 3 + 1) \times (28 - 3 + 1) \times 32 = 26 \times 26 \times 32$ tensor. We apply max-pooling with 2×2 filters and stride 2. Note the tensor is of size $13 \times 13 \times 32$ and the volume depth is unaffected.

In the second convolution layer, we use 64 $3 \times 3 \times 1$ filters with ReLU function. Then we get a $(13 - 3 + 1) \times (13 - 3 + 1) \times 64 = 11 \times 11 \times 64$ tensor. We apply max-pooling with 2×2 filters and stride 2. Note the tensor is of size $5 \times 5 \times 64$.

We do the third convolution layer again, we use 128 $3 \times 3 \times 1$ filters with ReLU function. Then we get a $(5 - 3 + 1) \times (5 - 3 + 1) \times 128 = 3 \times 3 \times 128$ tensor. We use dropout to prevent overfitting with $p = 0.4$. After flattening the tensor, we put it into the fully connected network. The first fully connected layer use ReLU function as the activation function. Finally we use softmax function to output the result.

For the loss function, we choose Cross Entropy Loss function and use Stochastic gradient descent to update the parameter with learning rate 0.05. During the experiments, we have tried different learning rate $(0.1, 0.05, 0.01)$. For smaller learning rate, the curve of training loss is smoother. If the learning rate is too small or too large, the testing accuracy is low. When $lr = 0.05$, we get the highest accuracy. Note some learning rate is unstable, that is, the accuracy stuck at 10% eventually.

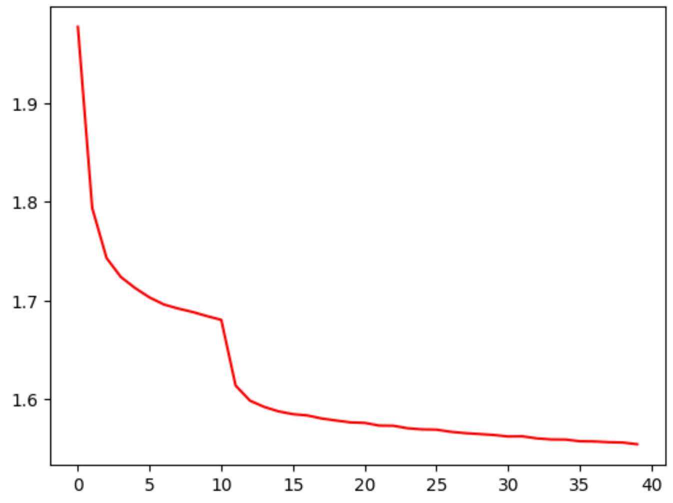


```
# testing set

model.eval()
correct = 0
# Turn off gradient descent
with torch.no_grad():
    for X, y in tqdm(test_dataloader):
        X, y = X.to(device), y.to(device)
        pred = model(X)
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
size = len(test_dataloader.dataset)
correct = correct / size
print(f" Test accuracy: {(100*correct):>0.1f}%")
```

✓ 1.1s

```
100%|██████████| 40/40 [00:00<00:00, 46.56it/s]

 Test accuracy: 90.5%
```

We achieve 90.5% testing accuracy with just 40 epochs. The result is quite impressive and how can't we believe the power of deep learning? We can see the loss function is keep decreasing. Although there is a 'angle' around the $10th$ epoch, it doesn't affect the final result a lot and we can see that the loss function tends to 1.5 stably after about 15 epochs.



For full code, readers may refer to
https://github.com/marcotam2002/MATH3320-miniproject

9. Conclusion

In the mini-project, we discuss different basic knowledge for deep learning, and extend to CNN. Finally, we try to run a CNN on a fashion MNIST dataset. We hope that this project can let other students have a brief overview on how Deep Learning looks like. There are many different aspects in deep learning, e.g. NLP, Computer Vision, etc. There are more and more learning methods. Hope this project can show you the beauty of deep learning.