NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE4483 Artificial Intelligence and Data Mining

Continuous Assessment – Project (Option 2)

Student: ZENG JUNHAO

Matric No: U1520619D

Tutorial Group: FC2

1 Objectives

This project aims to build classifiers on the MNIST dataset following provided guidelines and appreciate the process. Specifically, the classifiers are: 1) a simple neural network 2) a simple convolutional neural network (CNN) 3) a complex CNN to boost accuracy above 99%.

2 MNIST Dataset

2.1 Download & Preprocess

MNIST is an image dataset of handwritten digits. There are 60000 images for training and 10000 for testing. All images are 28x28 and grayscale (meaning there is one channel and the pixel value is from 0 to 255). The source files, including images and labels, are encoded in `IDX` format. To illustrate, Fig 2.1 shows the first 12 lines in `train-images-idx3-ubyte`

```
1   0000 0803 0000 ea60 0000 001c 0000 001c
2   0000 0000 0000 0000 0000 0000 0000 0000
3   0000 0000 0000 0000 0000 0000 0000 0000
4   0000 0000 0000 0000 0000 0000 0000 0000
5   0000 0000 0000 0000 0000 0000 0000 0000
6   0000 0000 0000 0000 0000 0000 0000 0000
7   0000 0000 0000 0000 0000 0000 0000 0000
8   0000 0000 0000 0000 0000 0000 0000 0000
9   0000 0000 0000 0000 0000 0000 0000 0000
10  0000 0000 0000 0000 0000 0000 0000 0000
11  0000 0000 0000 0000 0312 1212 7e88 af1a
12  a6ff f77f 0000 0000 0000 0000 0000 0000
```

Fig 2.1

```python
datasets.MNIST(mnist_dir, train=True, download=True,
 transform=transforms.Compose([
     transforms.ToTensor(),
     transforms.Normalize((0.1307,), (0.3081,)) # no.
 ]))
```

Fig 2.2

`IDX` presents data by 2-byte groups. In the 1st row, the 3rd + 4th group is 0x0000ea60 = 60000, representing the size. The 5th and 6th groups are 0x0000001c = 28, representing image width and height. Starting from 2nd row, each byte ($2^8 = 256$) represents a pixel.

With `Pytorch` installed, `torchvision` provides an easy way to download and preprocess the MNIST data. Fig 2.2 is the code, where `transforms.Normalize` normalize pixels within [-1, 1]. 0.1307 and 0.3081 are mean and standard deviation of the dataset. After transformation, each sample of the dataset has two elements. The first is a normalized tensor of size [1, 28, 28] and the second is the label. For example, we print the first sample, shown partially below, which is digit 5.

```python
for i in range(len(transformed_dataset)):
    sample = transformed_dataset[i]
    # print(i, sample[0].size(), sample[1].size())
    print(i, sample[1], sample[0])
    if i == 0:
        break
```

```
0 tensor(5) tensor([[[-0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242,
          -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242, -0.4242],
```

2.2 DataLoader

Organizing data into batches is to feed them into networks so that they are processed in groups. Essentially, this makes the input tensor a 4D array with an additional dimension as batch size. Shuffling is to introduce randomness in picking the training data. `Pytorch` enables setting these two features easily with `DataLoader`, shown in Fig 2.3.

```
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(mnist_dir, train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=batch_size, shuffle=True, **kwargs)
```

Fig 2.3 DataLoader

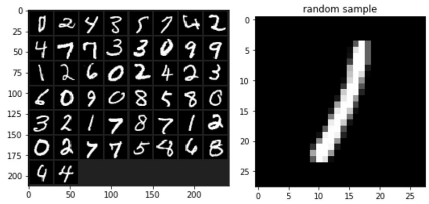


Fig 2.4 MNIST visualization

In this project, we set batch size for training and testing to be 50 and 100. Each sample of `DataLoader` is a tuple of batched data and corresponding labels. Fig 2.4 shows the first batch and a random image using `pyplot`. ** Note that in provided source code, the random sample is not in displayed in grayscale, in order to do so, we changed imshow function with `plt.imshow(npimg, cmap='gray')`.

3 A Simple Neural Network

3.1 Network Structure

A very simple neural network was tried first. It has only one fully connected (FC) layer. In terms of input vector, the size should be 28x28 = 784 since each image is rolled out into a 1D vector. In terms of output, there should be 10 neurons since 10 different labels (digits from 0 - 9). Activation function of the FC layer is `softmax`, which firstly raises each output to the power of `e`, then divided each one by the sum of all outputs, so that outputs sum up to 1 and the output vector can represent a categorical probability distribution. In terms of implementation with `Pytorch`, a child class of `nn.Module` is used to construct the network structure. `nn.Linear` is the FC layer and the `view` function flatten the image where the -1 argument serves like a placeholder left for the library to calculate, essentially that reshapes the [50, 1, 28, 28] input tensors into [50, 784].

\

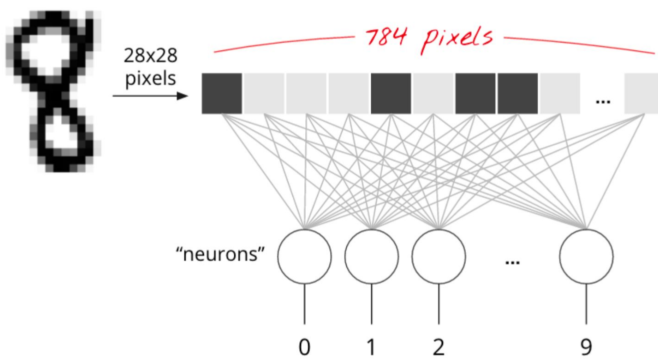


Fig 3.1 Network Structure

```
class simpleFC(nn.Module):
    def __init__(self, input_size=784, num_classes=10):
        super().__init__()
        self.fc1 = nn.Linear(input_size, num_classes)

    def forward(self, x):
        x = x.view(-1, 784)
        out = self.fc1(x)
        return F.log_softmax(out, dim=1)

model = simpleFC().to(device)
```

Fig 3.2 Pytorch implementation

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

Fig 3.3 Softmax

3.2 Loss Function

Cross entropy loss is used: $H(y, \hat{y}) = -\sum_i y_i log \hat{y}_i$. y is a one-hot vector representing output labels and $\hat{y}$ is the probability distribution after `softmax`. Therefore, if the computed probability is high for the actual label, the cross entropy loss is low, and vice versa. In `Pytorch`, such loss function can be either defined by `nn.NLLLoss()` given a `LogSoftmax` layer, or defined by `nn.CrossEntropyLoss()` without activation functions.

3.3 Optimizer

An optimizer sets the policy for back propagation algorithm to update weights. Stochastic Gradient Descent (SGD) optimizer is used. It only updates all the weights after loss of a batch of images is calculated. In `Pytorch`, optimizers are pre-implemented in torch.optim module. It is simply called by passing the model and hyperparameters like learning rate (lr).

Momentum: Momentum accelerates gradient descent in relevant direction and reduces oscillations shown in Fig 3.4 & Fig 3.5. It works by defining a hyperparameter called $momentum$, that leads to $V = momentum * V + (1 - momentum) * gradient\_of\_parameters$, so that the update in parameters is $-learning\_rate * V$. In this way, current step is reinforced by V if gradient step is in the same direction and reduced if direction changes.

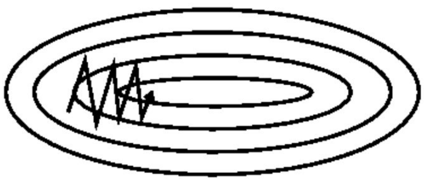


Fig 3.4 Without momentum

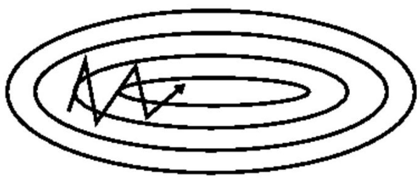


Fig 3.5 With momentum

3.4 Training & Testing

In `Pytorch`, as for training, we set model to training mode by calling `train()`. For each batch, we initialize gradients to 0 and feed the batch into model. Then loss is computed with previously defined loss function by passing the output and label. Gradient updates are computed by calling `backward()` on the loss and are actually updated on weights by calling `step()` on the optimizer. Every 100 batches or 5000 images, the loss is evaluated.

```
model.train()
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    output = model(data)
    loss = loss_function(output, target)
    loss.backward()
    optimizer.step()
    if batch_idx % 100 == 0:
        print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()))
```

Fig 3.6 Training implementation

```
Train Epoch: 1 [0/60000 (0%)]	Loss: 2.650063
Train Epoch: 1 [5000/60000 (8%)]	Loss: 0.581665
Train Epoch: 1 [10000/60000 (17%)]	Loss: 0.693524
Train Epoch: 1 [15000/60000 (25%)]	Loss: 0.562518
Train Epoch: 1 [20000/60000 (33%)]	Loss: 0.366446
Train Epoch: 1 [25000/60000 (42%)]	Loss: 0.369256
Train Epoch: 1 [30000/60000 (50%)]	Loss: 0.590143
Train Epoch: 1 [35000/60000 (58%)]	Loss: 0.539863
```

Fig 3.7 Part of the console outputs

As for testing, we set model to testing by `eval()`. After getting model output, index of highest output probability for an image is then the predicted label. In `Pytorch`, this is done by calling `torch.max(input=output.data, dim=1)`. `dim` indicates which axis is fixed in search of maximum (in this case the row), and the second return value is the indexes thus predictions of the batch. Testing accuracy is then computed by dividing the number of correct samples (=sum of predictions since one-hot) by total sample number.
The hyperparameters and results below.

epoch	batch_size	optimizer	learning rate	momentum	final loss	testing accuracy
5	50	SGD	0.001	0.9	0.368408	92.100%

4 A Simple CNN

4.1 Network Structure

The simple CNN provided by the guideline is used. Based on the given code, how each layer processes a single image input is visualized below. ** Note that in actual training phase, a batch of 50 images is fed into the network. For all convolutional filters, there's no padding and the stride is 1.

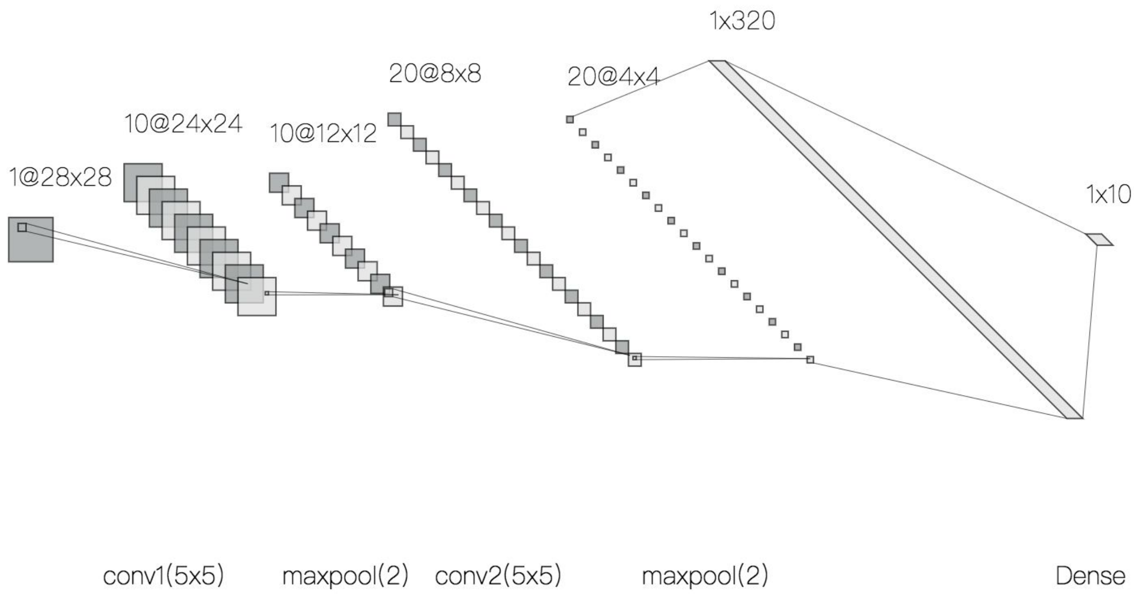


Fig 4.1 Visualization of the simple CNN

First, 10 convolutional filters of size [5, 5] are applied on the image, resulting in 10 feature maps of size 28 - 5 + 1 = 24 by 24. They are then max-pooled by 2x2 filters with stride 2, so the size is halved resulting in 10 12x12 maps. Then another convolutional layer is applied. Here, the output channel is 20 and the kernel size is 5, so there are 20 kernels of size [10, 5, 5] and the output should have the size of [20, 12 - 5 + 1 = 8, 8]. Following is the same max pooling which halves the feature map to [20, 4, 4]. **Flattening this map into a 1D array, the dimension is thus 20x4x4 = 320, which then goes through a FC layer to have output size of 10. This explains where the 320 comes from in `nn.Linear(320, 10)`.**

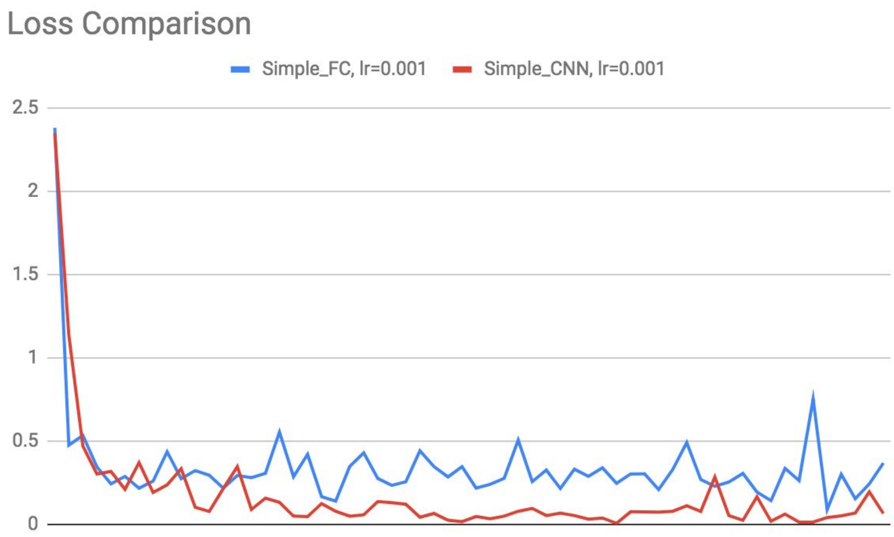
As for activation function, after each max pooling layer, `ReLU` is applied. After the final FC layer, `softmax` is applied.

4.2 Training & Testing

Same optimizer, loss function, training and testing functions previously defined are used. Below shows the hyperparameters and results.

epoch	batch_size	optimizer	learning rate	momentum	final loss	testing accuracy
5	50	SGD	0.001	0.9	0.065379	98.280%

Compared to the simple FC, this simple CNN performs much better. The final loss is smaller and the testing accuracy is higher. Below the loss plot of these two models are also plotted to visualize that CNN is more effective.



5 Fine-tune CNN

In this section, we record how we manipulated with CNN in order to achieve a testing accuracy higher than 99%.

5.1 Provided CNN

We first tried the CNN provided in the guideline. Its structure is very similar to the simple CNN. The difference is that it has two FC layers instead of one. The first FC layer has 50 neutrons and the second FC layer has 10 neutrons. In addition, the first FC layer is activated by `ReLU` and goes through a `Dropout` layer before feeding into the second FC.

`Dropout` layer is applied to prevent overfitting since FC layers occupy most of parameters. It works as follow: in each training iteration, every neuron has a probability (0.5 by default) to be dropped out, meaning they are removed in the training but added back after this iteration so their weights are not updated. Since not all neurons are trained on all data, this approach reduces overfitting and accelerate training process as well.
In this approach, the provided CNN architecture is not altered. The strategy is to train for more epochs and if the loss seems to converge, learning rate is lowered by multiplying with 0.1, in hope of approaching the optim. Below is what have been tried.

Epochs	Learning rate	Final loss	Testing accuracy
5	0.001	0.155010	97.700%
5	0.0001	0.085790	98.530%
5	0.00001	0.073976	98.540%

This approach is not effective in the end. By decreasing learning rate from 0.0001 to 0.00001 and training 5 more epochs, testing accuracy only improves 0.01%. Therefore, it is decided to modify the CNN architecture in order to boost the accuracy above 99%.

5.2 Complex CNN

We want to make the CNN more complex to achieve higher accuracy, so we do following:

- **Another convolution layer**
  When doing this, we realize that the current kernel size is 5, with two convolution layers and two max pooling layers, the size of a single feature map is only [4, 4], which is too small for a kernel to learn. So we decide to first change all kernel size to be 3. For the ease of calculation, we take a “same” padding approach, meaning to set the padding such that the feature map has the same size after a convolution layer. Solving 2 * padding - kernel_size +1 = 0, we thus set padding to be 1 for all convolutional layers. Currently, output channels of feature maps are only 10 and 20. Since deeper feature maps increase number of parameters, for the three convolutional layers, output channels are raised to 32, 64, 64 in sequence.

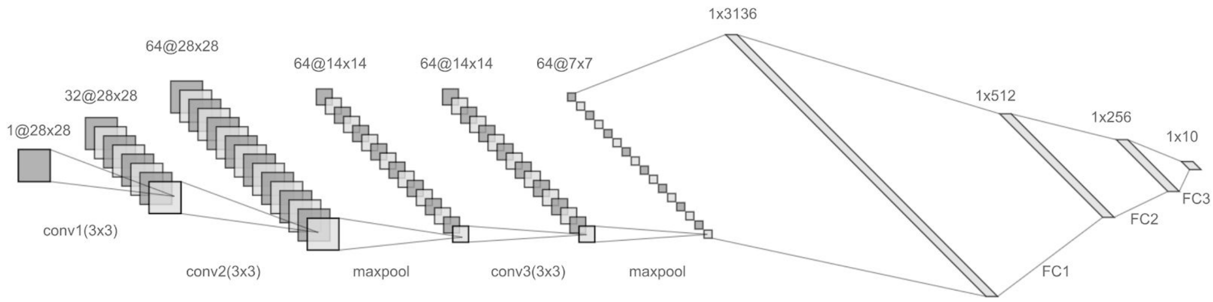
- **Keep the same max pooling layers**
  Although another convolutional layer is added, we do not add more max pooling layers. This is because with a “same” padding approach, after going through 2 convolutional layers and 2 max pooling layers, the size of feature map is [7, 7], which is not divisible by 2. In addition, a feature map of size less than 7 is already very small. So we decide to only implement max pooling layers on the last two convolutional layers.

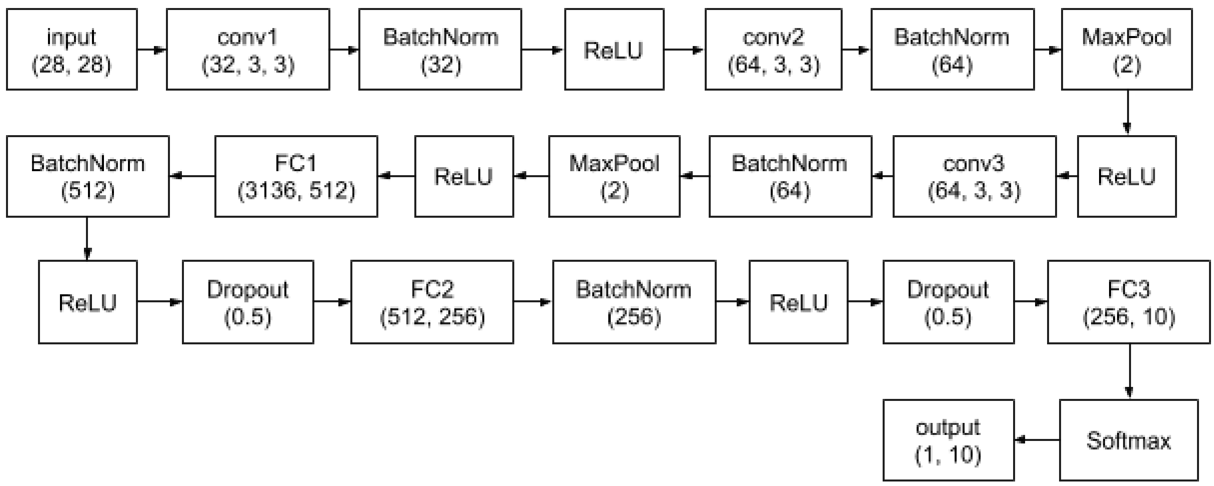
- **Another FC layer**
  After 3 convolutional layers and 2 max pooling layers, the feature maps are flattened into a 1D vector of dimension 64*(28/2/2)*(28/2/2) = 3136. To have one more FC

layer, we need to decide the number of neurons x, y in (3136, x) -> (x, y) -> (y, 10). Just for trials, we set x = 512 and y = 256.

- **Batch normalization layers**
  Before the activation in each convolutional layer and FC layer (except for the last FC layer), we add a batch normalization (BN) layer. BN normalizes its input by subtracting the batch mean and dividing the batch standard deviation. It also introduces two trainable parameters to perform linearization to add flexibility after normalization, which are adjusted by the gradient descent process. In this way, it is proven to prevent zero activations / vanishing gradients caused by internal covariate shift. It also speeds up the training process.

Based on the analysis above, diagrams below illustrate the proposed CNN structure.





In terms of training, same training and testing functions are used. The optimizer is SGD and the momentum is 0.9. But through learning experiments, we realized for very complex networks we could start with a relatively high learning rate so we started training this CNN for 2 epochs with lr=0.01 then continued to train for 2 epochs with lr=0.001. (*`Pytorch` implementation and training logs are attached in later sections.*). Results are shown below.

epochs	optimizer	learning rate	momentum	final loss	testing accuracy
2	SGD	0.01	0.9	0.062739	99.020%
2	SGD	0.001	0.9	0.067749	99.320%

In fact in the first 2 epochs with lr=0.01, we already achieved our objective - a testing accuracy higher than 99%. Using a lower learning rate and training for 2 more epochs, we finally improve the accuracy to 99.320%. Till now, objectives of this project have all been successfully met.

Learning Rate Experiments

We checked the loss behaviours of different learning rates (lr) on different models, namely a simple FC, a simple CNN (as provided) and a complex MLP with following flow:

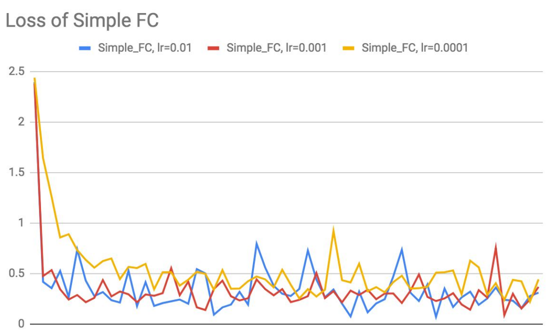
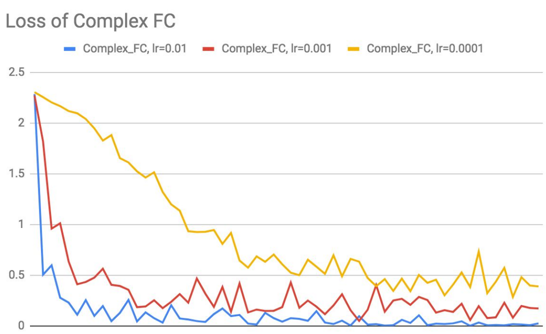
In `Pytorch`, the complex MLP is built as below.

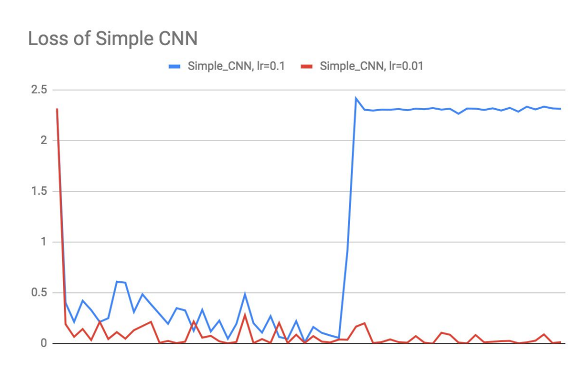
```
class simpleFC(nn.Module):
    def __init__(self, input_size=784, num_classes=10):
        # declaring operations
        super().__init__() # calling the inint method of
        self.fc1 = nn.Linear(input_size, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes) # fully-co

    def forward(self, x):
        # defines the forward pass
        x = x.view(-1, 784) # here we represent a MNIST i
        # 784 is the total number of pixels in a 28 by 28
        x = self.fc1(x) # weighted sum of input and netwo
        x = F.relu(x)
        x = F.dropout(x, p=0.2)
        x = self.fc2(x)
        x = F.relu(x)
        x = F.dropout(x, p=0.2)
        out = self.fc3(x)
        return F.log_softmax(out, dim=1) # return activat

model = simpleFC().to(device)
```

For the simple FC and complex MLP, we tested with lr = 0.01, lr = 0.001 and lr = 0.0001. For the simple CNN, we tested with lr=0.1, lr=0.01. Below are the results and discussion.

Loss Graphs	Analysis
	**lr=0.01**: gradient descent is the fastest yet loss fluctuates heavily, as it is hard to locate near the optimized point with a large step **lr=0.001**: a proper learning rate for this model. However, due to simple structure and feed forward style, the converged loss is higher than a complex FC or CNN. **lr=0.0001**: gradient descent is the slowest as each step is too small. Loss may further reduce given more epochs.
	**lr=0.01**: fastest gradient descent and a good learning curve with loss almost converging to 0. Though the lr is high for a simple FC, given this complex ANN has a lot of weights to update, it has the room for such lr to learn without overshooting optim. The testing accuracy is 97.860%. **lr=0.001**: lr is a bit small making gradient descent slow yet an acceptable testing accuracy of 95.970%. **lr=0.0001**: lr is too small and loss is still high in the end. The testing accuracy is only 89.530%.

	**lr=0.1**: lr is too high such that the gradient descent has overshot the optim. The testing accuracy is only 11.350%. **lr=0.01**: lr is appropriate. The gradient descent is fast and the final loss converges to 0. The testing accuracy is 97.260%.

Questions Answering

1. What is the depth (no. of units) for the input layer of the (neural) network that processes MNIST images?

$$28 * 28 = 784$$

2. What is the maximum spatial size of a convolutional filter that can be applied to a training image from the MNIST dataset (assuming the image has no padding)?

$$[28, 28]$$

3. What is the size of the output feature map after applying 2 convolutional layers (one after another) with 5x5 filters (for both layers) to the MNIST images (assuming the inputs and feature maps have no padding)?

$$28-5+1-5+1 = 20 \Rightarrow [20, 20]$$

4. Apply the 3x3 convolutional filter with a stride S=1 and no padding to the following input image and compute the output (Note: the code for this is available in the IPython notebook)

```
Image:
[[1 2 3 4 3]
 [1 3 4 3 1]
 [0 1 5 1 0]
 [1 3 4 3 1]
 [2 4 3 2 2]]

Kernel:
[[ 0 -1  0]
 [-1  5 -1]
 [ 0 -1  0]]

Output:
[[ 7  6  5]
 [-6 15 -6]
 [ 5  6  7]]
```

5. Apply the same filter on the input image as in the previous question, but this time

(a) With a stride S=2, no padding

```
Image:
[[1 2 3 4 3]
 [1 3 4 3 1]
 [0 1 5 1 0]
 [1 3 4 3 1]
 [2 4 3 2 2]]

Kernel:
[[ 0 -1  0]
 [-1  5 -1]
 [ 0 -1  0]]

Output:
[[7 5]
 [5 7]]
```

(b) With a stride S=1 and zero padding

```
Image padded:
[[0 0 0 0 0 0 0]
 [0 1 2 3 4 3 0]
 [0 1 3 4 3 1 0]
 [0 0 1 5 1 0 0]
 [0 1 3 4 3 1 0]
 [0 2 4 3 2 2 0]
 [0 0 0 0 0 0 0]]

Image:
[[1 2 3 4 3]
 [1 3 4 3 1]
 [0 1 5 1 0]
 [1 3 4 3 1]
 [2 4 3 2 2]]

Kernel:
[[ 0 -1  0]
 [-1  5 -1]
 [ 0 -1  0]]

Output:
[[ 2  3  5 11 10]
 [ 1  7  6  5 -1]
 [-3 -6 15 -6 -3]
 [ 0  5  6  7  0]
 [ 5 12  5  2  7]]
```

Above outputs are obtained from modified codes in `conv_filter.ipynb`, shown below

```
###########################################################
# You need to correctly compute the output image size given:
# 1) input image size (W1, H1)
# 2) Image padding P
# 3) Stride S
# 4) F: kernel size
# modify code here:
W2 = (W1 - F + 2*P)//S + 1 # width of the output
H2 = (H1 - F + 2*P)//S + 1 # height of the output
###########################################################
```

```
###########################################################
# You need to correctly account for the stride other than 1
# modify code here:
output[y,x]=(kernel*image_padded[S*y:S*y+F,S*x:S*x+F]).sum()
###########################################################
```

6. Train a simple fully connected network with only 1 fully-connected layer on the MNIST image data, so the resulting network has a {fc -> Softmax} architecture. Use the learning rate of 0.001 and momentum = 0.9. Train it for 5 epochs. Report the accuracy you can achieve

on the test image set. What is the number of parameters in this network? Provide your calculations.

Accuracy of the network on the test images: 92.100%; correct: 9210 out of 10000
Number of parameters: (784+1) * 10 = 7850

7. Train a Convolutional Neural Network (simpleCNN) on the MNIST data using the following network architecture: {conv1 (10 5x5 filters) -> MaxPool -> conv2 (20 5x5 filters) -> MaxPool -> ReLU -> fc1 -> Softmax}. Use the learning rate of 0.001 and momentum = 0.9. Train it for 5 epochs. Report the accuracy on the test image set.

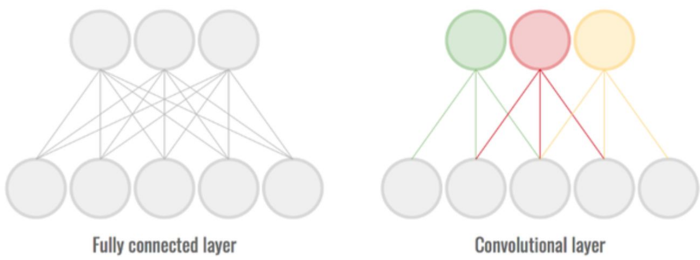
Accuracy of the network on the test images: 98.280%; correct: 9828 out of 10000

8. What is the total number of parameters in the simple CNN? Provide your calculations.

(5*5*1+1)*10+(5*5*10+1)*20+(320+1)*10 = 8490

9. Compare the number of parameters of the networks in Q6 and Q7. Which network has more parameters? Compare the performance on the test image set and explain the difference in accuracy, if any.

The simple CNN has more parameters. The simple CNN has a higher accuracy (98.280%) than the simple FC network (92.100%). One reason for better performance is trivial in that the CNN has around 1500 more parameters than the FC network. But the more important reason is that the CNN has a special structure which is more suitable for images. First, CNN uses convolutional layers. Spatially close points on images or feature maps are usually correlated and the way kernels are applied leverages this local spatial coherence. However, in a multilayer perceptron (MLP), the 2D image input is rolled out into a 1D vector so this spatial information is lost. The figure below visualizes this point.



Second, the simple CNN uses max pooling layers, which not only reduces the size of feature maps and parameters to learn, it also provides a translation invariance to the internal representation spatially. By pooling the maximum regional points, the network would address to the issue of whether an object appears or not, instead of focusing on where the object is. Therefore, it is very helpful for image classification tasks.

10. [Optional] Finetune the architecture of the CNN and report the best accuracy you could achieve on the test image set. It would be great that if you can achieve an accuracy better than 99%. Specify the network used.

Please refer to section 5.2 Complex CNN.

Implementation of the Complex CNN & its Training Logs

```
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64*7*7, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)
        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(64)
        self.bn4 = nn.BatchNorm1d(512)
        self.bn5 = nn.BatchNorm1d(256)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.max_pool2d(x, 2)
        x = F.relu(x)
        x = self.conv3(x)
        x = self.bn3(x)
        x = F.max_pool2d(x, 2)
        x = F.relu(x)
        x = x.view(-1, 64*7*7)
        x = self.fc1(x)
        x = self.bn4(x)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        x = self.bn5(x)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.fc3(x)

        return F.log_softmax(x, dim=1)

cnn_model_2 = CNN().to(device)
```

```
Train Epoch: 1 [0/60000 (0%)]	Loss: 2.358605
Train Epoch: 1 [5000/60000 (8%)]	Loss: 0.362447
Train Epoch: 1 [10000/60000 (17%)]	Loss: 0.115301
Train Epoch: 1 [15000/60000 (25%)]	Loss: 0.122175
Train Epoch: 1 [20000/60000 (33%)]	Loss: 0.116554
Train Epoch: 1 [25000/60000 (42%)]	Loss: 0.191841
Train Epoch: 1 [30000/60000 (50%)]	Loss: 0.034423
Train Epoch: 1 [35000/60000 (58%)]	Loss: 0.022370
Train Epoch: 1 [40000/60000 (67%)]	Loss: 0.054307
Train Epoch: 1 [45000/60000 (75%)]	Loss: 0.221910
Train Epoch: 1 [50000/60000 (83%)]	Loss: 0.079650
Train Epoch: 1 [55000/60000 (92%)]	Loss: 0.160576
Train Epoch: 2 [0/60000 (0%)]	Loss: 0.039653
Train Epoch: 2 [5000/60000 (8%)]	Loss: 0.091880
Train Epoch: 2 [10000/60000 (17%)]	Loss: 0.045071
Train Epoch: 2 [15000/60000 (25%)]	Loss: 0.017580
Train Epoch: 2 [20000/60000 (33%)]	Loss: 0.040908
Train Epoch: 2 [25000/60000 (42%)]	Loss: 0.016423
Train Epoch: 2 [30000/60000 (50%)]	Loss: 0.039192
Train Epoch: 2 [35000/60000 (58%)]	Loss: 0.051257
Train Epoch: 2 [40000/60000 (67%)]	Loss: 0.102503
Train Epoch: 2 [45000/60000 (75%)]	Loss: 0.010123
Train Epoch: 2 [50000/60000 (83%)]	Loss: 0.010963
Train Epoch: 2 [55000/60000 (92%)]	Loss: 0.062739
Training finished

[0/100]	Correct: 99	Total: 100
[10/100]	Correct: 1090	Total: 1100
[20/100]	Correct: 2084	Total: 2100
[30/100]	Correct: 3072	Total: 3100
[40/100]	Correct: 4058	Total: 4100
[50/100]	Correct: 5053	Total: 5100
[60/100]	Correct: 6041	Total: 6100
[70/100]	Correct: 7030	Total: 7100
[80/100]	Correct: 8024	Total: 8100
[90/100]	Correct: 9009	Total: 9100
Accuracy of the network on the test images: 99.020%; correct: 9902 out of 10000

Train Epoch: 1 [0/60000 (0%)]	Loss: 0.041984
Train Epoch: 1 [5000/60000 (8%)]	Loss: 0.071085
Train Epoch: 1 [10000/60000 (17%)]	Loss: 0.014451
Train Epoch: 1 [15000/60000 (25%)]	Loss: 0.014497
Train Epoch: 1 [20000/60000 (33%)]	Loss: 0.035567
Train Epoch: 1 [25000/60000 (42%)]	Loss: 0.026986
Train Epoch: 1 [30000/60000 (50%)]	Loss: 0.014952
Train Epoch: 1 [35000/60000 (58%)]	Loss: 0.013512
Train Epoch: 1 [40000/60000 (67%)]	Loss: 0.044437
Train Epoch: 1 [45000/60000 (75%)]	Loss: 0.022795
```

```
Train Epoch: 1 [50000/60000 (83%)]	Loss: 0.014681
Train Epoch: 1 [55000/60000 (92%)]	Loss: 0.188442
Train Epoch: 2 [0/60000 (0%)]	Loss: 0.042125
Train Epoch: 2 [5000/60000 (8%)]	Loss: 0.024907
Train Epoch: 2 [10000/60000 (17%)]	Loss: 0.099192
Train Epoch: 2 [15000/60000 (25%)]	Loss: 0.167722
Train Epoch: 2 [20000/60000 (33%)]	Loss: 0.020995
Train Epoch: 2 [25000/60000 (42%)]	Loss: 0.022316
Train Epoch: 2 [30000/60000 (50%)]	Loss: 0.026142
Train Epoch: 2 [35000/60000 (58%)]	Loss: 0.010289
Train Epoch: 2 [40000/60000 (67%)]	Loss: 0.046773
Train Epoch: 2 [45000/60000 (75%)]	Loss: 0.063486
Train Epoch: 2 [50000/60000 (83%)]	Loss: 0.022127
Train Epoch: 2 [55000/60000 (92%)]	Loss: 0.067749
Training finished

[0/100]	Correct: 99	Total: 100
[10/100]	Correct: 1096	Total: 1100
[20/100]	Correct: 2093	Total: 2100
[30/100]	Correct: 3086	Total: 3100
[40/100]	Correct: 4080	Total: 4100
[50/100]	Correct: 5067	Total: 5100
[60/100]	Correct: 6059	Total: 6100
[70/100]	Correct: 7055	Total: 7100
[80/100]	Correct: 8043	Total: 8100
[90/100]	Correct: 9038	Total: 9100
Accuracy of the network on the test images: 99.320%; correct: 9932 out of 10000
```

Reference

An overview of gradient descent optimization algorithms
http://ruder.io/optimizing-gradient-descent/index.html#momentum

Dropout in Convolutional Neural Network
https://en.wikipedia.org/wiki/Convolutional_neural_network#Dropout

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
https://arxiv.org/pdf/1502.03167.pdf

What is the difference between a convolutional neural network and a multilayer perceptron?
https://www.quora.com/What-is-the-difference-between-a-convolutional-neural-network-and-a-multilayer-perceptron

Pytorch Documentation https://pytorch.org/docs/stable/index.html

Publication-ready NN-architecture schematics http://alexlenail.me/NN-SVG/LeNet.html