COMP4660/8420 Lab 4
Deep Learning

Part 1: Deep Learning Basics

1.1 General

1. What is the definition of deep learning?

2. Describe the differences in feature generation between Deep Learning and traditional shallow machine learning techniques.

1.2 Convolutional Neural Networks

3. What type of tasks do convolutional neural networks excel at?

4. In a convolutional neural network, how does the local receptive field of a neuron differ to that of another neuron in the same activation map?

5. What is the purpose of a pooling layer in a convolutional neural network?

6. Determine which value would be output from a pooling layer for the following input and the following pooling functions. For each, assume a size of 2x2 and stride of 2.

4	3	5	1
2	1	3	1
1	1	2	3
0	0	1	2

   a. Max-pooling
   b. Average pooling

7. The following questions refer to the filter (kernel) of a convolutional layer within a convolutional neural network.

   a. How many filters are there in a layer?

   b. What data does the filter convolve with? How is this computed?

   c. How many times are the weights of a single filter applied to the input of that convolutional layer?

   d. How many values would it output each time it is applied?

   e. Where is the output stored?

8. What is the purpose of DropOut in a convolutional neural network?

Part 2: Deep Learning in Python/PyTorch

2.1 Convolutional Neural Network

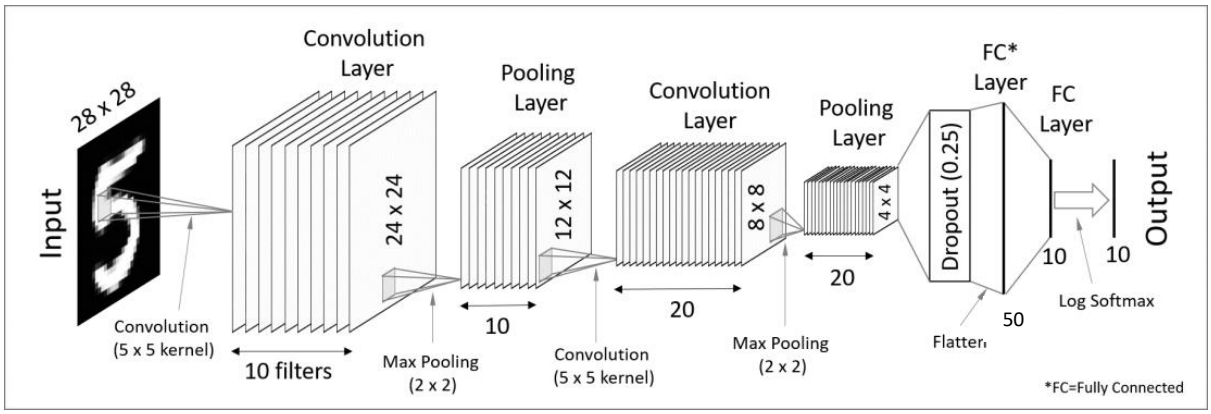
This task will help you build a basic convolutional neural network with PyTorch.

Download "task1_cnn.py" from Wattle. To execute the script, navigate to your directory and type the following command in your terminal (if you are using one of the lab machines in CSIT, please use Anaconda3 Shell as your terminal):

```
python3 task1_cnn.py

--------------------
Note on this script

Task1_cnn.py accepts 8 optional arguments. A full list of arguments can be found below:

GENERAL ARGUMENTS:
-h                    --help                show help message and exit

REQUIRED ARGUMENTS:
(none)

OPTIONAL ARGUMENTS:
[--batch-size               input batch size for training (default:64)             ]
[--test-batch-size          input batch size for testing (default: 1000)           ]
[--epochs                   number of epochs to train (default: 10)                ]
[--lr                       learning rate (default: 0.01)                          ]
[--momentum                 SGD momentum (default: 0.5)                            ]
[--no-cuda                  disables CUDA training (default: False)                ]
[--seed                     random seed (default: 1)                               ]
[--log-interval             how many batches to wait before logging training status
                            (default: 100)                                         ]

So, for example, to run this script with a different learning rate of 0.005 and 500
training epochs, you can type "python3 task1_cnn.py --lr 0.005 --epochs 500"
```

For this task, we will build a CNN to classify hand written digits using the popular MNIST (http://yann.lecun.com/exdb/mnist/) dataset, which contains a training set of 60,000 labelled images and a test set of 10,000 labelled images.

The two functions in class `Net` (`__init__(self)` and `forward(self, x)`) are intentionally blank, so try to **implement these two functions** to define a CNN with two conv layers followed by a dropout, then two fully connected layers. The forward pass of the CNN can be defined with a ReLU activation function for each hidden layer and a dropout after the first fully connected layer. An example of the structure of the CNN is drawn as below:

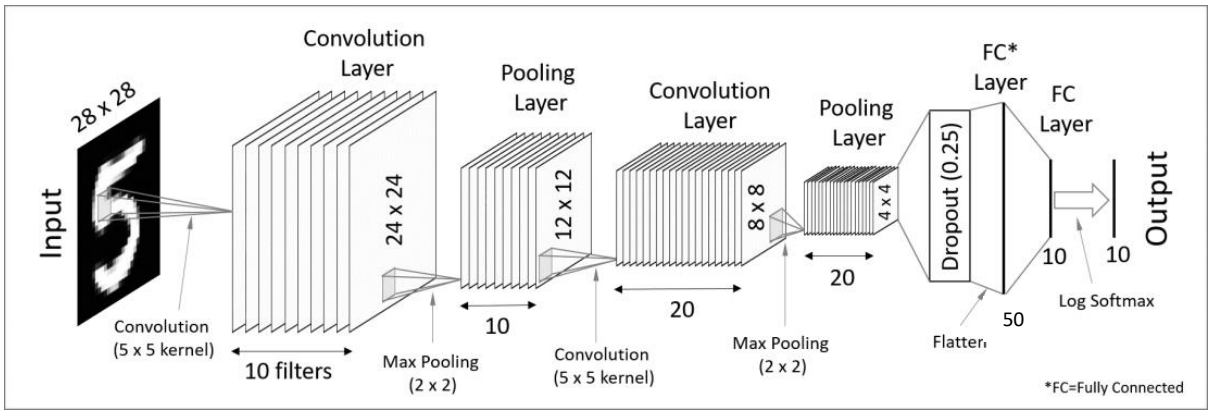


Hint: Please refer to *Appendix 1. Understanding task1_cnn.py* if you don't know how to start or have trouble understanding "task1_cnn.py".

Once you have built the CNN, **experiment with the number of channels and the kernel sizes** and investigate their impacts on the performance of the classification.

Appendix 1. Understanding task1_cnn.py

In this task, we will build a CNN to classify hand written digits. We will use the popular MNIST dataset. An example of the structure of the CNN is provided below.



This CNN takes a 28 x 28 pixel, greyscale, input image, fed through several layers, one after the other, and finally gives an output vector, which contain the log probability (since we will use the Negative Log Likelihood loss function) that the input was one of the digits 0 to 9.

Training the network means that you have a dataset of matching input-output pairs. So if you give a hand written digit of a 5 as an input, you will know what the expected output is, in this case a vector of zeros with a one at index 5 (this is also called one-hot encoding).

A typical training procedure for a neural network is also applied here as follows:

1. Define the neural network which has some learnable parameters, often called weights.
2. Iterate over the dataset or inputs (could also be done as batches).
3. Process the input through the network and calculate the output.
4. Compute the loss (how far the calculated output differed from the correct output)
5. Propagate the gradients back through the network.
6. Update the weights of the network according to a simple update rule. Such as:

   weight = weight - learning_rate * gradient

Step 1. Define the network

The most convenient way of defining our network is by creating a new class which extends `nn.Module.` The `Module` class simply provides a convenient way of encapsulating the parameters, and includes some helper functions such as moving data parameters to GPU, etc.

A network is usually defined in two parts. First we initialize all the functions that we will use (these can be reused multiple times). And then in the required `forward`() function we "connect" our network together using the components defined in the initialize function as well as any of the Tensor operations. We can also use all the activation functions, such as ReLu and SoftMax, which is provided in the `torch.nn.functional` package.

Note that since we are only using built in functions we do not have to define the backward function (which is where the gradients are computed), since this is automatically determined by the `autograd` package. Also, if we want to train our network using the GPU, we can achieve this by simply calling `net.cuda()`.

The learnable parameters of the model are returned by `net.parameters(),` and for interest sake you can view the size of each layer's weights, and retrieve the actual weight values for the kernels that are used as below.

```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's weights size
print(params[0][0,0])  # conv1's weights for the first filter's kernel
```

Step 2. Iterate over dataset or inputs

The input that the network must be a `autograd.Variable,` as is the output. But first, how do we process our dataset in a simple way to iterate over it?

Loading data

This script uses the `DataLoader` class to load datasets. It can automatically divide our data into matches as well as shuffle it among other things. It can be used to load supplied or custom datasets, that can be defined using the `Dataset` class.

```
Mini note on batching for PyTorch

torch.nn only supports mini-batches The entire torch.nn package only supports inputs that
are a mini-batch of samples, and not a single sample.
For example, nn.Conv2d will take in a 4D Tensor of nSamples x nChannels x Height x Width.
```

Since we use MNIST data here, we can get the already processes dataset for free in `torchvision` which can be installed as http://pytorch.org/ shows. Using `torchvision` and `DataLoader`, we can create our training and test dataset as follows:

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

batch_size = 100
```

```
train_dataset = datasets.MNIST(root='./data/', train=True, transform=transforms.ToTensor(),
download=True)
test_dataset = datasets.MNIST(root='./data/', train=False, transform=transforms.ToTensor(),
download=True)

# batch the data for the training and test datasets
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)

print(train_loader.__len__()*train_loader.batch_size, 'train samples')
print(test_loader.__len__()*test_loader.batch_size, 'test samples\n')
```

Iterating over the dataset

In order to run several epochs we will define a new function `train()` to run our training loop.

When we are busy training our network we need to set it in "training mode", this effectively only means that we would like the Dropout and BatchNorm layers to be active (we generally turn them off when running our test data). We do this by simply call `net.train()`. Again, if we want all our data on the GPU, to increase performance, we will convert all our Tensors to their GPU version using `data.cuda()`. And finally, remember our network module requires the input to be of type `Variable`, so we simply cast our image and target to that type. The first part of our `train()` function will look as follows:

```
def train(epoch):
    net.train() # set the model in "training mode"

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data.cuda()), Variable(target.cuda())
```

Step 3. Process input through the network and get output

Since we have already done all of the difficult work in setting up the network. This is literally only one line, which we will add in the loop at the end of the last code snippet.

```
output = net(data)
```

Since we defined our network to use the Log Softmax at the end, the output will the contain the Log of the probability that the input was for each of the digits from 0 to 9. The reason we used the Log Softmax, is because we will use the Negative Log Likelihood loss function, which expects the Log Softmax as input.

Step 4. Compute the loss

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

When designing a neural network, you have a choice between several loss functions, some of are more suited certain tasks. Since we have a classification problem, either the Cross Entropy loss or the related Negative Log Likelihood (NLL) loss can be used. In this example we will use the NLL loss. Since the softmax can be interpreted as the the probability the that the input belongs to one of the output classes, and this probability is between 0 and 1, when taking the log of that value, we find that the value increases (and is negative), which is the

opposite of what we want, so we simply negate the answer, hence the Negative Log Likelihood. The internal formula for the loss is as follows:

$$L_i = -\log(\frac{e^{f_{yi}}}{\sum_j e^{f_j}})$$

In PyTorch there is a built in NLL function in `torch.nn.functional` called `nll_loss`, which expects the output in log form. That is why we calculate the Log Softmax, and not just the normal Softmax in our network. Using it as simple as adding one line to our training loop, and providing the network output, as well as the expected output.

```
loss = F.nll_loss(output, target)
```

Step 5. Propagate the gradient back

In this step we only *calculate* the gradients, but we don't use them yet. That happens in the next step. We would like to calculate the gradients of the loss relative to the input, so in order to do this just leverage the power of PyTorch's `autograd` and call the `.backward()` function on the loss variable. We however first need to clear the existing gradients, otherwise gradients will be accumulated to existing gradients. For this we will use `.zero_grad()`, before we will call the `.backward()` function. For now we add two more lines to our training loop:

```
optimizer.zero_grad()
loss.backward()
```

Step 6. Update the weights of the network

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

We will define our optimizer directly after our model, as follows:

```
import torch.optim as optim
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

Now within our `train()` function we must remember to zero our gradients before calling `.backward()` as well as tell our optimizer to "step", meaning that the weights will be updated using the calculated gradients according to our rule. Since our entire training loop is finished now, here is the function in its entirety:

```
def train(epoch):
    net.train() # set the model in "training mode"

    for batch_idx, (data, target) in enumerate(train_loader):
        # data.cuda() loads the data on the GPU, which increases performance
        data, target = Variable(data.cuda()), Variable(target.cuda())

        optimizer.zero_grad()   # necessary for new sum of gradients
        output = net(data)      # call the forward() function (forward pass of network)
```

```
        loss = F.nll_loss(output, target) # use negative log likelihood to determine loss
        loss.backward()         # backward pass of network (calculate sum of gradients for
graph)
        optimizer.step()        # perform model parameter update (update weights)
```

Testing our trained network

Now that we have finished training our model, we will probably also want to test how well our model was generalized by applying it to on our test dataset. For this we essentially copy our training function and just modify it to set the model in "evaluation mode" using `net.eval()`, which will turn Dropout and BatchNorm off. We would also like to accumulate the loss and print out the accuracy. Here is the code for the `test()` function

```
def test():
    net.eval()  # set the model in "testing mode"
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        if args.cuda:
            data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target) # volatile=True,
since the test data should not be used to train
        output = net(data)
        test_loss += F.nll_loss(output, target, size_average=False).data[0]
#fsize_average=False: # sum up batch loss, instead of average losses
        pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-
probability
        correct+= pred.eq(target.data.view_as(pred)).long().cpu().sum() # to operate on
variables they need to be on the CPU again

    test_dat_len = len(test_loader.dataset)
    test_loss /= test_dat_len

    # print the test accuracy
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss, correct, test_dat_len, 100. * correct / test_dat_len))
```

Repeat

Usually our model is not trained very well after only running it once. We would therefore want to train (and test) our model for several epochs.

```
for epoch in range(1, epochs):
    train(epoch)
    test()
```