Machine Learning HW#3

UIN: 231003486

Name: Chan-Wei Hu

Question 1: Maximum likelihood estimate

(a.i) The handwritten derivation is as follow.

Let $\theta = \{\mu, \sigma^2\}$, and the data $\mathcal{X} = \{x_1, \ldots, x_N\}$

$p(\mathcal{X}|\theta) = \prod_{n=1}^{N} p(x_n|\theta) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x_n-\mu)^2}{2\sigma^2}\right)$, define $L(\theta|\mathcal{X}) \equiv p(\mathcal{X}|\theta)$

if we take $\log L(\theta|\mathcal{X})$

$$\longrightarrow \log L(\theta|\mathcal{X}) = -\frac{n}{2}\log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2$$

take first order derivative with respect to μ

$\therefore \frac{\partial L(\theta|\mathcal{X})}{\partial \mu} = \frac{1}{\sigma^2}\sum_{n=1}^{N}(x_n-\mu) = 0 \Rightarrow$ if $\mu = \frac{1}{N}\sum_{n=1}^{N} x_n$, then $\frac{\partial L(\theta|\mathcal{X})}{\partial \mu} = 0$

then we take first order derivative w.r.t σ^2

$$\therefore \frac{\partial L(\theta|\mathcal{X})}{\partial \sigma^2} = -\frac{N}{\sigma^2} + \frac{1}{(\sigma^2)^2}\sum_{n=1}^{N}(x_n-\mu)^2 = \frac{N}{(\sigma^2)^2}\left(\sigma^2 - \frac{1}{N}\sum_{n=1}^{N}(x_n-\mu)^2\right)$$

$$\Rightarrow \sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n-\mu)^2$$

thus, we prove that $\hat{\mu} = \frac{1}{N}\sum_{n=1}^{N} x_n$ and $\hat{\sigma}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n-\hat{\mu})^2$

(a.ii) Code as follows.

```python
# CSCE 633 - Machine Learning
# HW3 - Question 1
import pdb
import numpy as np
import pandas as pd

data = pd.read_csv('Q1_Data.csv', header=None)
data = data.to_numpy()

# calculate mean and variance
mean = np.sum(data) / len(data)
var = np.sum(np.power(data - mean, 2)) / len(data)

print('Mean: %.5f, Variance: %.5f' % (mean, var))
```

and I got mean=1.07116, variance=0.08841.

(b)

$$N_1, N_2, N_3 \sim \text{Multinomial}\left(N_1+N_2+N_3,\ p=\{(1-\phi)^2, \phi^2, 2\phi(1-\phi)\}\right)$$

$$\therefore \ell(\phi) = \log\left(f(N_1, N_2, N_3 \mid (1-\phi)^2, \phi^2, 2\phi(1-\phi))\right)$$

$$= \log\left(\frac{(N_1+N_2+N_3)!}{N_1!\,N_2!\,N_3!}\left[(1-\phi)^2\right]^{N_1} \phi^{2\cdot N_2} \cdot \left[2\phi(1-\phi)\right]^{N_3}\right)$$

$$= N_1 \cdot \log\left[(1-\phi)^2\right] + N_2 \cdot \log(\phi^2) + N_3 \cdot \log(2\phi(1-\phi))$$

$$= 2N_1 \cdot \log(1-\phi) + N_3 \cdot \log(1-\phi) + 2N_2 \cdot \log(\phi) + N_3 \cdot \log(\phi) + \cancel{\log 2}$$

$$= (2N_1+N_3) \cdot \log(1-\phi) + (2N_2+N_3) \cdot \log(\phi)$$

$$\frac{\partial \ell(\phi)}{\partial \phi} = \frac{-2N_1 - N_3}{1-\phi} + \frac{2N_2+N_3}{\phi} = 0$$

$$\Rightarrow \frac{\partial \ell(\phi)}{\partial \phi} = \frac{-2N_1\phi - N_3\phi}{\phi(1-\phi)} - \frac{2N_2+N_3-2N_2\phi-N_3\phi}{\phi(1-\phi)} = 0$$

$$\therefore \text{we got } \phi = \frac{2N_2+N_3}{2N_1+2N_2+2N_3} \quad \#$$

Question 2: Machine learning for facial recognition

(a) I wrote a function to randomly choose the samples from each class. As follows.

```python
def show_random_img(self, spc):
    # spc: samples per class
    # show random examples for each class
    class_num = len(set(self.y))
    # initial figure

    fig, axs = plt.subplots(spc, class_num,
                            figsize=(15, spc*2))

    for c in range(class_num):
        idx = np.where(self.y == c)[0]
        random_img = self.X[np.random.choice(idx, spc)]
        axs[0, c].set_title(EMOTION_MAP[c])
        for i in range(spc):
            img = random_img[i].reshape(48, 48)
            axs[i, c].axis('off')
            axs[i, c].imshow(img, cmap='gray', vmin=0, vmax=255)
```

Visualization:

(b)

A function is implemented to help calculate the data distribution.

```
def data_distribution(self):
    # Help calculate data distribution
    unique_label = set(self.y)

    for i in unique_label:
        samples_of_class = np.sum(self.y == i)
        print('%s: %d samples' % (EMOTION_MAP[i], samples_of_class))
```

Emotion	Angry	Disgust	Fear	Happy	Sad	Surprise	Neutral
# of samples	3995	436	4097	7215	4830	3171	4965

Total: 28709 samples

(c) Image classification with FNNs

(c.i) For first setup, please refer to the following table. I will use four combinations for the experiment. (GPU for training: RTX 2070 super)

I ran for 1000 epochs to fairly compare between each setting. Note that I use cross-entropy loss as my criterion, but in my loss plot, I took the average of the original loss over total number of data.

	# of layers	# nodes per layer	activation	dropout	batch norm
Setting 1	1 hidden layers	4096	ReLU	no	yes
Setting 2	2 hidden layers	4096	ReLU	no	yes
Setting 3	2 hidden layers	4096	SELU	no	yes
Setting 4	2 hidden layers	4096	ReLU	yes	yes

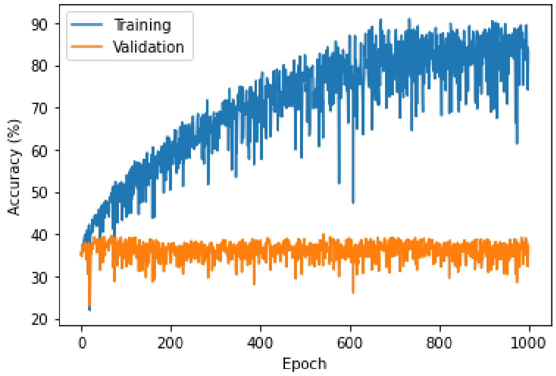
For setting 1:

Total training time is 2892.28 seconds ~ 50 minutes

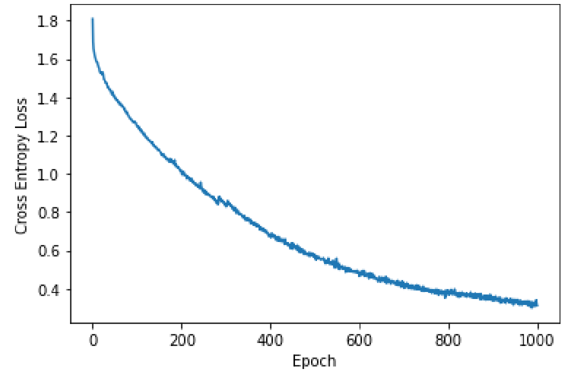
Total parameters can be displayed as follows. I use a library called `pytorch_model_summary` to help calculate the parameters for each layer.

```
-----------------------------------------------------------------------
      Layer (type)        Input Shape         Param #     Tr. Param #
=======================================================================
          Linear-1          [1, 2304]       9,441,280       9,441,280
            ReLU-2          [1, 4096]               0               0
     BatchNorm1d-3          [1, 4096]           8,192           8,192
          Linear-4          [1, 4096]          28,679          28,679
            ReLU-5             [1, 7]               0               0
     BatchNorm1d-6             [1, 7]              14              14
=======================================================================
Total params: 9,478,165
Trainable params: 9,478,165
Non-trainable params: 0
-----------------------------------------------------------------------
```

<table>
<tr><th>Training & validation accuracy</th><th>Loss curve</th></tr>
<tr><td>



At epoch 1000:
Training acc = 83.315%
Validation acc = 37.03%
</td><td>



Final loss at epoch 1000: 0.312
</td></tr>
</table>

Note that the best validation accuracy was at epoch 542, which gave me 40.13%. From the loss curve, I found that it is hard for this setting to converge to optimal solution (even it can, it might take a long time). I think this is because the model has no capability to represent such a high dimensional data.

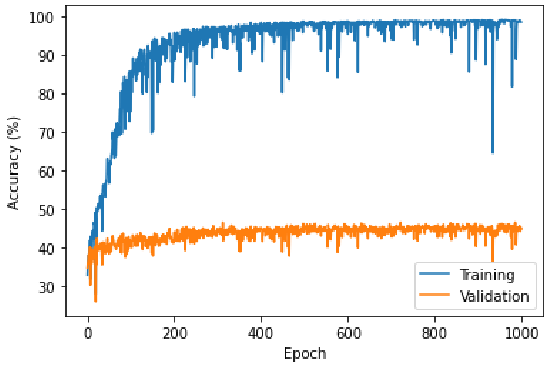
For setting 2:

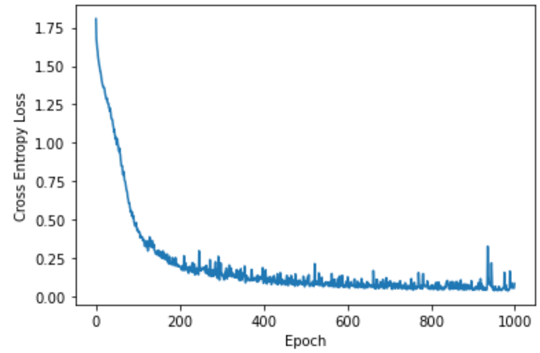
Total training time is 10469.81 seconds ~ 3 hours

Same as above, parameters can be shown as…

Layer (type)	Input Shape	Param #	Tr. Param #
Linear-1	[1, 2304]	9,441,280	9,441,280
ReLU-2	[1, 4096]	0	0
BatchNorm1d-3	[1, 4096]	8,192	8,192
Linear-4	[1, 4096]	16,781,312	16,781,312
ReLU-5	[1, 4096]	0	0
BatchNorm1d-6	[1, 4096]	8,192	8,192
Linear-7	[1, 4096]	28,679	28,679
ReLU-8	[1, 7]	0	0
BatchNorm1d-9	[1, 7]	14	14

Total params: 26,267,669
Trainable params: 26,267,669
Non-trainable params: 0

<table>
<tr><th>Training & validation accuracy</th><th>Loss curve</th></tr>
<tr><td>

At epoch 1000:
Training acc = 98.373%
Validation acc = 44.72%</td><td>

Final loss at epoch 1000: 0.084</td></tr>
</table>

As the result indicates, when increasing hidden layers, our network is able to learn better representations, and thus can increase the accuracy in validation data. Note that the best validation accuracy was at epoch 312, which gave me 45.19%.

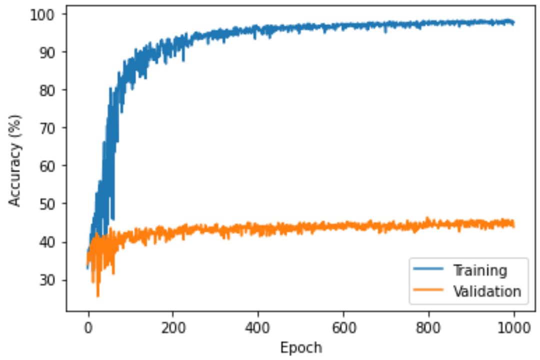
For setting 3:

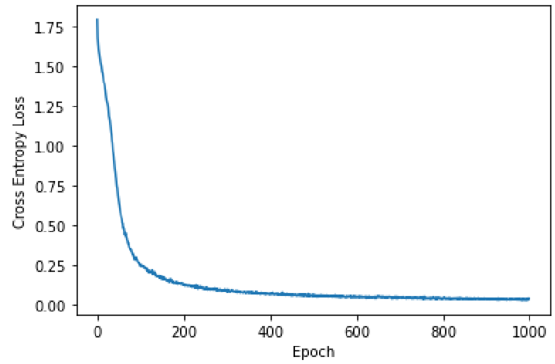
In this setting, I use SELU as my activation function, which ensures that the outputs have zero mean and unit standard deviation, and thus can make our network converge faster. Let's see if it really helps!

Total training time is 10642.56 seconds ~ 3 hours

Parameters:

```
------------------------------------------------------------------------
      Layer (type)          Input Shape         Param #     Tr. Param #
========================================================================
          Linear-1           [1, 2304]       9,441,280       9,441,280
            SELU-2           [1, 4096]               0               0
     BatchNorm1d-3           [1, 4096]           8,192           8,192
          Linear-4           [1, 4096]      16,781,312      16,781,312
            SELU-5           [1, 4096]               0               0
     BatchNorm1d-6           [1, 4096]           8,192           8,192
          Linear-7           [1, 4096]          28,679          28,679
            SELU-8              [1, 7]               0               0
     BatchNorm1d-9              [1, 7]              14              14
========================================================================
Total params: 26,267,669
Trainable params: 26,267,669
Non-trainable params: 0
------------------------------------------------------------------------
```

<table>
<tr><th>Training & validation accuracy</th><th>Loss curve</th></tr>
<tr><td>

At epoch 1000:
Training acc = 97.712%
Validation acc = 43.717%</td><td>

Final loss at epoch 1000: 0.038</td></tr>
</table>

Compare to setting 2, the loss curve seems to be more stable, as well as the curve of training & validation accuracy. However, the model converges at 500 epochs, which is almost the same in setting 2. Thus, regarding convergence, it seems that SELU only helps on **stability**. Note that the best validation accuracy was at epoch 798, which gave me 46.336%.

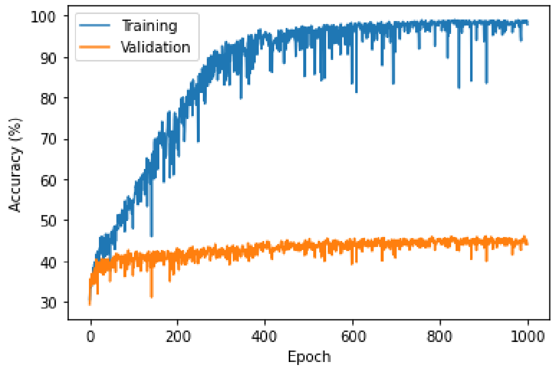
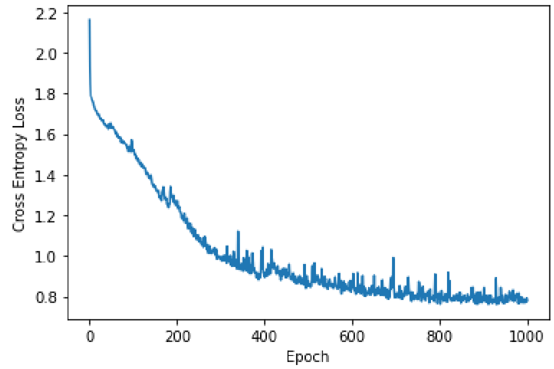
For setting 4:

In this setting, I use ReLU for activation, and apply the dropout in the output layer.

Total training time is 11160.81 seconds ~ 3 hours

Parameter:

```
----------------------------------------------------------------------
      Layer (type)        Input Shape        Param #     Tr. Param #
======================================================================
          Linear-1        [1, 2304]        9,441,280       9,441,280
            ReLU-2        [1, 4096]                0               0
     BatchNorm1d-3        [1, 4096]            8,192           8,192
          Linear-4        [1, 4096]       16,781,312      16,781,312
            ReLU-5        [1, 4096]                0               0
     BatchNorm1d-6        [1, 4096]            8,192           8,192
          Linear-7        [1, 4096]           28,679          28,679
            ReLU-8           [1, 7]                0               0
     BatchNorm1d-9           [1, 7]               14              14
        Dropout-10           [1, 7]                0               0
======================================================================
Total params: 26,267,669
Trainable params: 26,267,669
Non-trainable params: 0
----------------------------------------------------------------------
```

Training & validation accuracy	Loss curve
 At epoch 1000: Training acc = 97.851% Validation acc = 44.107%	 Final loss at epoch 1000: 0.779

From the results shown above, by using dropout, the model tends to converge slower than setting 2 (with the same setting but no dropout). I think this is because dropout randomly eliminates some nodes, and thus more noise is added to the network. Note that the best validation accuracy was at epoch 994, which is 46.113%.

(c.ii)

Recall the settings from (c.i):

	# of layers	# nodes per layer	activation	dropout	batch norm
Setting 1	1 hidden layers	4096	ReLU	no	yes
Setting 2	2 hidden layers	4096	ReLU	no	yes
Setting 3	2 hidden layers	4096	SELU	no	yes
Setting 4	2 hidden layers	4096	ReLU	yes	yes

The testing accuracy:

	Epoch of best model	Best validation accuracy	Testing accuracy
Setting 1	542	40.13%	37.81%
Setting 2	312	45.19%	44.246%
Setting 3	798	46.336%	44.915%
Setting 4	994	46.113%	46.141%

As the result shown, setting 4 (with dropout at output layer) performs the best among all. This is not surprising because dropout regularizes the network, and thus slightly boosts the performance. For setting 2 and 3, I conclude SELU performs better than ReLU in this dataset. However, it is not guaranteed that SELU will always be good, since the testing accuracy is pretty close.

Code for (c):

In `model.py`

```
import torch.nn as nn

class FNN(nn.Module):
    def __init__(self, hidden_size):
        super(FNN, self).__init__()
        self.hidden_size = hidden_size

        # layers
        self.fc1 = nn.Sequential(
            nn.Linear(in_features = 2304,
                      out_features = 4096,),
            nn.ReLU(),
            #nn.SELU(),
            nn.BatchNorm1d(4096),
            #nn.Dropout(),
        )

        self.fc2 = nn.Sequential(
            nn.Linear(in_features = 4096,
                      out_features = 4096,),
            #nn.SELU(),
            nn.ReLU(),
            nn.BatchNorm1d(4096),
            #nn.Dropout(),
        )

        self.fc3 = nn.Sequential(
            nn.Linear(in_features = 4096,
```

```
                    out_features = 7,),
                #nn.SELU(),
                nn.ReLU(),
                nn.BatchNorm1d(7),
                nn.Dropout(),
            )

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)

        return x
```

In `EmotionDataset.py` (Contains the class to perform dataloader supported in PyTorch)

```
class EmotionDataset(Dataset):
    def __init__(self, images, labels=None, transforms=None,
resize=False):
        self.X = images
        self.y = labels
        self.transforms = transforms
        self.resize = resize

    def __len__(self):
        return (len(self.X))

    def __getitem__(self, i):
        data = self.X[i, :]
        if self.resize:
```

```
            data =
np.asarray(data).astype(np.float32).reshape(1, 48, 48)
        if self.transforms:
            data = self.transforms(data)

        if self.y is not None:
            return (data, self.y[i])
        else:
            return data
```

In `misc.py` (contains some useful functions, such as read_data -> read data from csv)

```
import os
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# read the csv file
def read_data(data_dir):
    # start time
    t0 = time.process_time()

    train_data = pd.read_csv(os.path.join(data_dir,
'Q2_Train_Data.csv'))
    val_data = pd.read_csv(os.path.join(data_dir,
'Q2_Validation_Data.csv'))
    test_data = pd.read_csv(os.path.join(data_dir,
'Q2_Test_Data.csv'))

    # get data and labels
    # training data
```

```
    train_img = train_data.iloc[:, 1].apply(lambda x:
x.split())
    train_img = [list(map(int, train_img[i])) for i in
range(len(train_data))]
    train_img = np.asarray(train_img, dtype=np.float32)
    train_label = train_data.iloc[:, 0]

    # validation data
    val_img = val_data.iloc[:, 1].apply(lambda x: x.split())
    val_img = [list(map(int, val_img[i])) for i in
range(len(val_data))]
    val_img = np.asarray(val_img, dtype=np.float32)
    val_label = val_data.iloc[:, 0]

    # testing data
    test_img = test_data.iloc[:, 1].apply(lambda x: x.split())
    test_img = [list(map(int, test_img[i])) for i in
range(len(test_data))]
    test_img = np.asarray(test_img, dtype=np.float32)
    test_label = test_data.iloc[:, 0]

    print('Data Processing Done. Time elapsed: %.2f sec\n'
                                % (time.process_time()-t0))

    return train_img, train_label, val_img, val_label,
test_img, test_label

# plot the accuracy curve
def plot_acc(curve_list, curve_label):
    assert len(curve_list) == len(curve_label)

    data_len = len(curve_list)
```

```
    for i in range(data_len):
        plt.plot(range(len(curve_list[i])), curve_list[i], 
label=curve_label[i])
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')

    plt.legend()
    plt.show()
```

In `main.py` (main code for training procedure)

```
import os
import pdb
import time
import torch
import pandas as pd
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from torchvision.transforms import transforms
from torch.utils.data import DataLoader
import pytorch_model_summary as pms

# import my function
from EmotionDataset import EmotionDataset
from model import FNN, CNN
from misc import read_data, plot_acc

# read data first
data_dir = './'
```

```
# start doing training
HIDDEN_SIZE = 4096
MAX_EPOCH = 1000
batch_size = 64
net = 'FNN'
phase = 'test'
checkpoint = os.path.join('./model', net)
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')

if phase == 'train':
        for epoch in range(1, MAX_EPOCH+1):
            # start traning for one epoch
            running_loss = train(model, trainloader,
valloader, optimizer, criterion)
            # get the average loss
            train_loss.append(running_loss/len(trainloader))

            # test the result
            train_acc = test(model, trainloader)
            val_acc = test(model, valloader)
            Acc_train.append(train_acc)
            Acc_val.append(val_acc)

            # store the model and print out result
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                best_epoch = epoch
                PATH = os.path.join(checkpoint,
'set4_best.pt')
                torch.save(model.state_dict(), PATH)
```

```
                print('[Epoch %d] loss: %.3f, training acc: %.3f, val acc: %.3f -> Model saved!' % \
                  (epoch, running_loss/len(trainloader), train_acc, val_acc))
            else:
                print('[Epoch %d] loss: %.3f, training acc: %.3f, val acc: %.3f' % \
                (epoch, running_loss/len(trainloader), train_acc, val_acc))


        print('Training Done. Best validation acc = %.3f at epoch %d.\n Time elapsed: %.2f sec\n'
                                        % (best_val_acc, best_epoch, time.process_time()-t0))


        test_acc = test(model, testloader)
        print('Testing accuracy = %.3f' % test_acc)


        # plot
        plot_acc([Acc_train, Acc_val], ['Training', 'Validation'])


        plt.plot(range(MAX_EPOCH), train_loss)
        plt.xlabel('Epoch')
        plt.ylabel('Cross Entropy Loss')
        plt.show()
```

In `main.py` (function for training one epoch)

```
def train(net, trainloader, valloader, optimizer, criterion):
    Acc_train, Acc_val, train_loss = [], [], []
    best_val_acc = 0
    net.train()
```

```
    running_loss = 0
    for data in trainloader:
        # data pixels and labels to GPU if available
        inputs, labels = data[0].to(device, 
non_blocking=True), data[1].to(device, non_blocking=True)

        # set the parameter gradients to zero
        optimizer.zero_grad()
        outputs = net(inputs)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    train_loss.append(running_loss/len(trainloader))

    return running_loss
```

In `main.py` (function for testing the network)

```
def test(net, testloader):
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            inputs, labels = data[0].to(device, 
non_blocking=True), data[1].to(device, non_blocking=True)
            outputs = net(inputs)
```

```
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

(d) Image classification with CNNs

(d.i) The hardware environment is the same as in FNN experiments.

I ran for 1000 epochs to fairly compare between each setting. Note that I use cross-entropy loss as my criterion, but in my loss plot, I took the average of the original loss over total number of data.

In addition, in this CNN experiment, I fixed the number of fully connected layers to 3 layers. This is to compare the performance from only conv layers without any effect from FC layers.

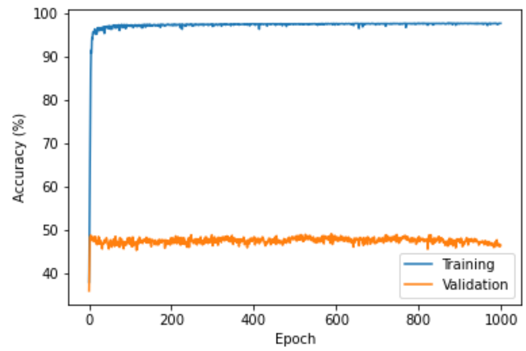
	# of conv layers	kernel size	stride size	activation	2D dropout	batch norm
Setting 1	1	3	1	ReLU	no	yes
Setting 2	1	5	1	ReLU	no	yes
Setting 3	3	3	1	ReLU	no	yes
Setting 4	3	3	1	ReLU	yes	yes

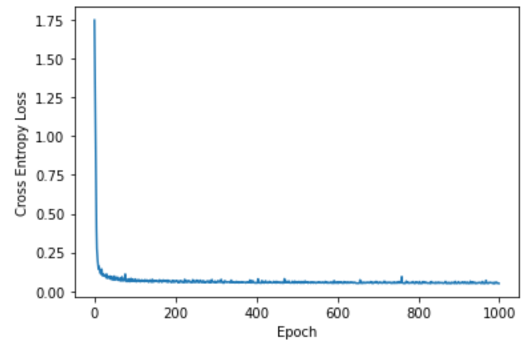
For setting 1:

Time elapsed for training: 3121.59 seconds ~ 52 mins.

Parameters:

```
-----------------------------------------------------------------------
      Layer (type)          Input Shape         Param #     Tr. Param #
=======================================================================
          Conv2d-1       [1, 1, 48, 48]             160             160
            ReLU-2      [1, 16, 48, 48]               0               0
       MaxPool2d-3      [1, 16, 48, 48]               0               0
     BatchNorm2d-4      [1, 16, 24, 24]              32              32
          Linear-5            [1, 9216]       2,359,552       2,359,552
            ReLU-6             [1, 256]               0               0
          Linear-7             [1, 256]          16,448          16,448
            ReLU-8              [1, 64]               0               0
          Linear-9              [1, 64]             455             455
           ReLU-10               [1, 7]               0               0
=======================================================================
Total params: 2,376,647
Trainable params: 2,376,647
Non-trainable params: 0
-----------------------------------------------------------------------
```

<table>
<tr><th>Training & validation accuracy</th><th>Loss curve</th></tr>
<tr><td>

At epoch 1000:
Training acc = 97.687%
Validation acc = 46.42%</td><td>

Final loss at epoch 1000: 0.05</td></tr>
</table>

From the results, the observation is that CNNs definitely performs better than FNNs with only one conv layer! Moreover, the convergence is fast (within 100 epochs, the model will converge to the solution)!

Note that the best validation accuracy is at epoch 589 with accuracy = 49.07 %.

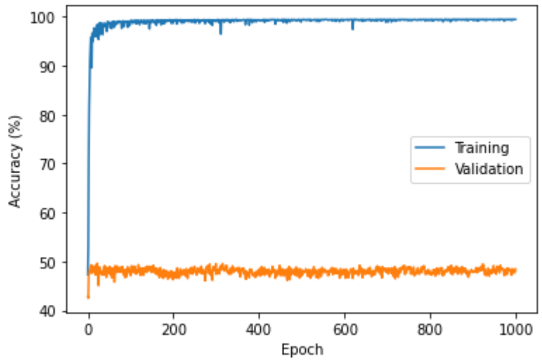
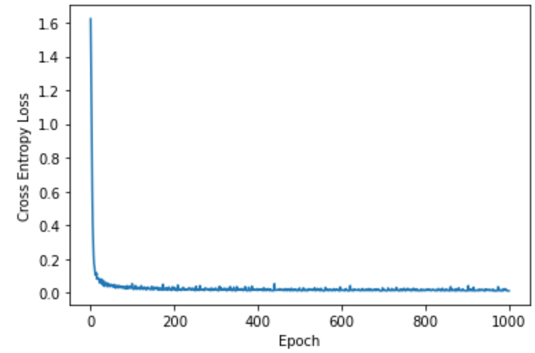
For setting 2:

I increase the kernel size from 3 to 5, and thus the local receptive field increases. Let's see how this affects the performance.

Time elapsed for training: 3307.94 seconds ~ 1 hour

Parameters:

```
------------------------------------------------------------------------
      Layer (type)          Input Shape         Param #     Tr. Param #
========================================================================
          Conv2d-1       [1, 1, 48, 48]             416             416
            ReLU-2      [1, 16, 48, 48]               0               0
       MaxPool2d-3      [1, 16, 48, 48]               0               0
     BatchNorm2d-4      [1, 16, 24, 24]              32              32
          Linear-5            [1, 9216]       2,359,552       2,359,552
            ReLU-6             [1, 256]               0               0
          Linear-7             [1, 256]          16,448          16,448
            ReLU-8              [1, 64]               0               0
          Linear-9              [1, 64]             455             455
           ReLU-10               [1, 7]               0               0
========================================================================
Total params: 2,376,903
Trainable params: 2,376,903
Non-trainable params: 0
------------------------------------------------------------------------
```

Training & validation accuracy	Loss curve
 At epoch 1000: Training acc = 99.446% Validation acc = 48.45%	 Final loss at epoch 1000: 0.012

Since I increase the kernel size, the model should learn more accurate. The reason is that the receptive field increases. The result shows that the accuracy indeed slightly increase! Note that the best validation accuracy is at epoch 23 with accuracy = 49.624 %.

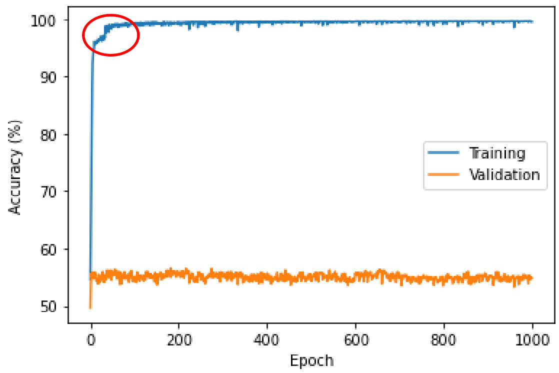
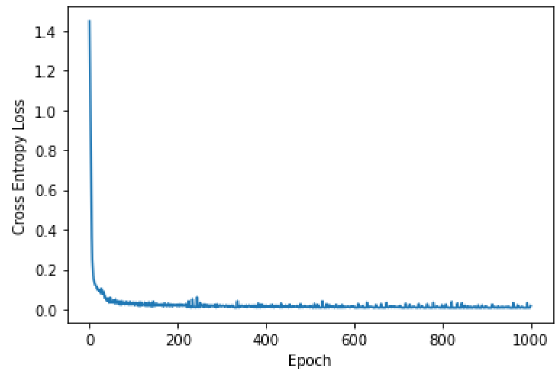
For setting 3:

I increase the # of conv layer to 3 layers, with the filter size increase by 16 for each layer (red circle).

Time elapsed for training: 3762.19 seconds ~ 1 hour

Parameters:

```
------------------------------------------------------------------------
        Layer (type)          Input Shape         Param #     Tr. Param #
========================================================================
            Conv2d-1      [1, 1, 48, 48]             160             160
              ReLU-2     [1, 16, 48, 48]               0               0
         MaxPool2d-3     [1, 16, 48, 48]               0               0
       BatchNorm2d-4     [1, 16, 24, 24]              32              32
            Conv2d-5     [1, 16, 24, 24]           4,640           4,640
              ReLU-6     [1, 32, 24, 24]               0               0
         MaxPool2d-7     [1, 32, 24, 24]               0               0
       BatchNorm2d-8     [1, 32, 12, 12]              64              64
            Conv2d-9     [1, 32, 12, 12]          18,496          18,496
             ReLU-10     [1, 64, 12, 12]               0               0
        MaxPool2d-11     [1, 64, 12, 12]               0               0
      BatchNorm2d-12       [1, 64, 6, 6]             128             128
           Linear-13           [1, 2304]         590,080         590,080
             ReLU-14            [1, 256]               0               0
           Linear-15            [1, 256]          16,448          16,448
             ReLU-16             [1, 64]               0               0
           Linear-17             [1, 64]             455             455
             ReLU-18              [1, 7]               0               0
========================================================================
Total params: 630,503
Trainable params: 630,503
Non-trainable params: 0
------------------------------------------------------------------------
```

Training & validation accuracy	Loss curve
 At epoch 1000: Training acc = 99.484% Validation acc = 54.89%	 Final loss at epoch 1000: 0.018

With only one convolutional layer, it may not be sufficient for out model to learn a good representation, and thus stacking more layers may help boost the accuracy. Apparently, the result indicates that this indeed helps!! Note that the best validation accuracy is at epoch 183 with accuracy = 56.645 %.

In the red circle, there is a weird drop of the training accuracy. I think it is because I didn't do any regularization or something else to stabilize the process. However, I am not sure.

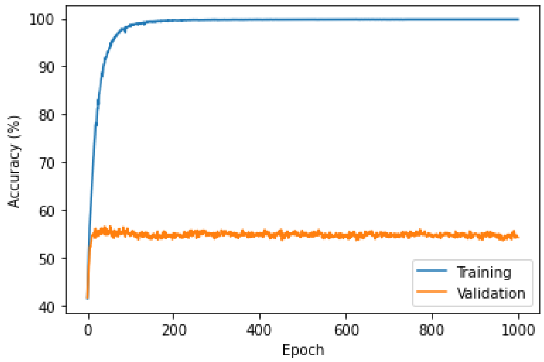
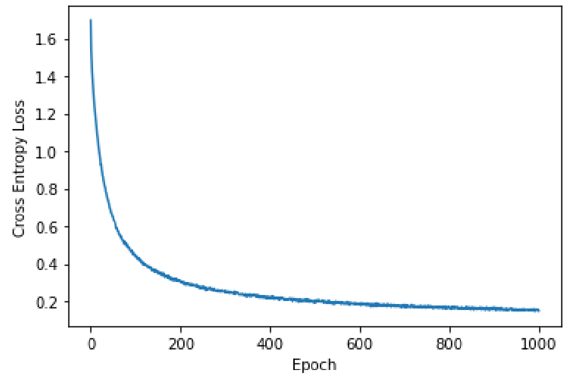
For setting 4:

In this setting, I enable dropout in my model to help prevent overfitting and help add some noise to the network to increase stability.

Time elapsed for training: 2964.53 seconds ~ 50 mins

Parameter:

```
----------------------------------------------------------------------
       Layer (type)          Input Shape          Param #     Tr. Param #
======================================================================
          Conv2d-1       [1, 1, 48, 48]              160             160
            ReLU-2      [1, 16, 48, 48]                0               0
       MaxPool2d-3      [1, 16, 48, 48]                0               0
     BatchNorm2d-4      [1, 16, 24, 24]               32              32
       Dropout2d-5      [1, 16, 24, 24]                0               0
          Conv2d-6      [1, 16, 24, 24]            4,640           4,640
            ReLU-7      [1, 32, 24, 24]                0               0
       MaxPool2d-8      [1, 32, 24, 24]                0               0
     BatchNorm2d-9      [1, 32, 12, 12]               64              64
      Dropout2d-10      [1, 32, 12, 12]                0               0
         Conv2d-11      [1, 32, 12, 12]           18,496          18,496
           ReLU-12      [1, 64, 12, 12]                0               0
      MaxPool2d-13      [1, 64, 12, 12]                0               0
    BatchNorm2d-14        [1, 64, 6, 6]              128             128
      Dropout2d-15        [1, 64, 6, 6]                0               0
         Linear-16            [1, 2304]          590,080         590,080
           ReLU-17             [1, 256]                0               0
         Linear-18             [1, 256]           16,448          16,448
           ReLU-19              [1, 64]                0               0
         Linear-20              [1, 64]              455             455
           ReLU-21               [1, 7]                0               0
======================================================================
Total params: 630,503
Trainable params: 630,503
Non-trainable params: 0
----------------------------------------------------------------------
```

<table>
<tr><th>Training & validation accuracy</th><th>Loss curve</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>At epoch 1000:
Training acc = 99.784 %
Validation acc = 54.31 %</td><td>Final loss at epoch 1000: 0.149</td></tr>
</table>

By adding the dropout, the loss curve becomes smoother. Compared to setting 3, this can achieve the same results with stability. The best validation accuracy is at epoch 54 with accuracy = 56.617 %.

In conclusion, since FNNs are all fully connected layers, the parameters are usually more than CNNs'. Furthermore, FNNs have no spatial information of the image, while CNNs are able to handle because of the using of convolution.

(d.ii)

Recall the settings from (d.i):

	# of conv layers	kernel size	stride size	activation	2D dropout	batch norm
Setting 1	1	3	1	ReLU	no	yes
Setting 2	1	5	1	ReLU	no	yes
Setting 3	3	3	1	ReLU	no	yes
Setting 4	3	3	1	ReLU	yes	yes

The testing accuracy:

	Epoch of best model	Best validation accuracy	Testing accuracy
Setting 1	589	49.07 %	47.59 %
Setting 2	23	49.624 %	47.67 %
Setting 3	183	56.645 %	57.48 %
Setting 4	54	56.617 %	55.95 %

From the final result, the usage of conv layers really helps boost the accuracy. Another advantage is that CNNs have less parameters than FNNs, which make CNNs easier to optimize.

Code for (d):

In `model.py`:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv1 = nn.Sequential(         # input shape: (1,
48, 48)
            nn.Conv2d(in_channels = 1,
                      out_channels = 16,
                      kernel_size = 3,
                      stride = 1,
                      padding = 1,),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2),  # output shape:
(16, 24, 24)
            nn.BatchNorm2d(16),
            nn.Dropout2d(),
        )

        self.conv2 = nn.Sequential(         # input shape:
(16, 24, 24)
            nn.Conv2d(in_channels = 16,
                      out_channels = 32,
                      kernel_size = 3,
                      stride = 1,
                      padding = 1,),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2),  # output shape:
(64, 12, 12)
            nn.BatchNorm2d(32),
```

```
            nn.Dropout2d(),
        )

        self.conv3 = nn.Sequential(         # input shape:
(64, 12, 12)
            nn.Conv2d(in_channels = 32,
                      out_channels = 64,
                      kernel_size = 3,
                      stride = 1,
                      padding = 1,),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2),  # output shape:
(64, 6, 6)
            nn.BatchNorm2d(64),
            nn.Dropout2d(),
        )



        self.fc1 = nn.Sequential(
            nn.Linear(16 * 24 * 24, 256),
            nn.ReLU(),
        )

        self.fc2 = nn.Sequential(
            nn.Linear(256, 64),
            nn.ReLU(),
        )

        self.fc3 = nn.Sequential(
            nn.Linear(64, 7),
            nn.ReLU(),
```

```
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

(g) Feature design

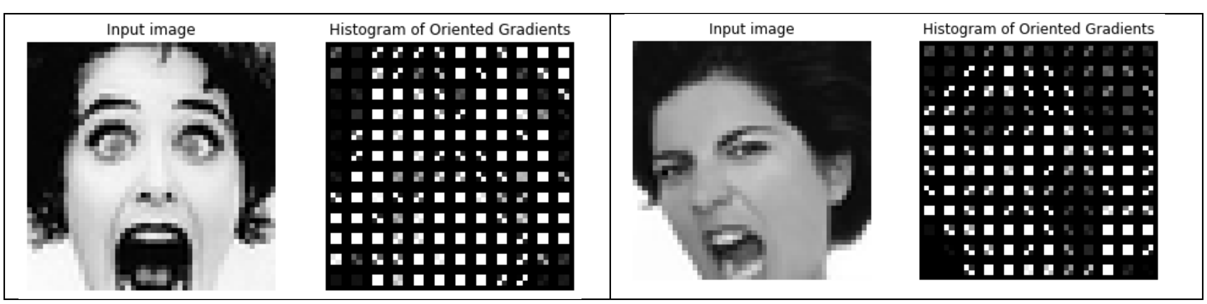
First, I use HOG to extract features of each image, and then using such features to train my FNN. In this question, I use the setting 4 in (c.i).

	# of layers	# nodes per layer	activation	dropout	batch norm
Setting 1	1 hidden layers	4096	ReLU	no	yes
Setting 2	2 hidden layers	4096	ReLU	no	yes
Setting 3	2 hidden layers	4096	SELU	no	yes
Setting 4	2 hidden layers	4096	ReLU	yes	yes

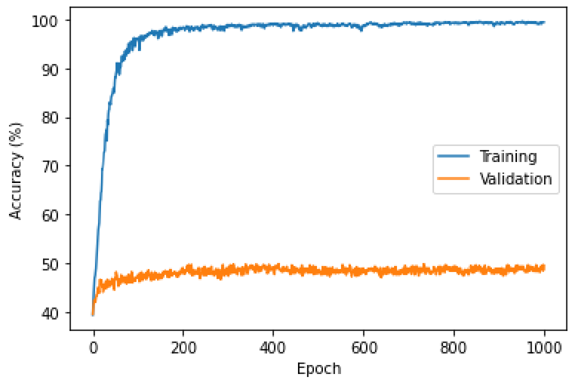
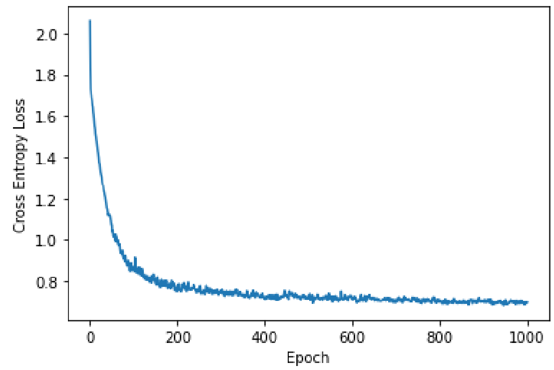
I use `from skimage.feature import hog` to perform HOG computation. As follows:

```
img = np.asarray(train_img[i]).reshape(48, 48)
fd, data = hog(img, orientations=4, pixels_per_cell=(4, 4),
               cells_per_block=(1, 1), visualize=True)
train_img_HOG[i] = np.asarray(data).astype(np.float32).reshape(48*48)
```

Some examples from this parameter setting:



After training with HOG features, the experiment result is:

Training & validation accuracy	Loss curve
 At epoch 1000: Training acc = 99.429 % Validation acc = 49.54 %	 Final loss at epoch 1000: 0.698

The best validation accuracy is **49.875 %,** occurred at epoch 413. Recall that in setting 4 of (c.i), the best validation accuracy is 46.113%.

Let's look into the testing accuracy.

setting 4 with raw image	setting 4 with HOG feature
46.141 %	49.763 %

Apparently, HOG feature helps a lot! This is because HOG extracts gradient information from the image, which is useful. However, directly train from raw image may include lots of redundant information, and thus degrades the performance.