EECS445 Project 2

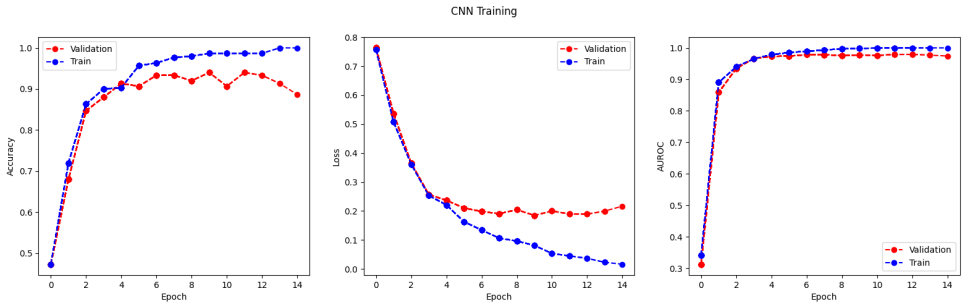
uniqname: duanzq

5 Challenge

We first created a default model. Challenge class in model/challenge.py was implemented in the same way as Target class in model/target.py. And we defined loss function, optimizer, and patience as follows:

```
    # TODO: define loss function, and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    #TODO: define patience for early stopping
    patience = 5
    curr_patience = 0
```



To test the performance of the model, similar to **test_cnn.py**, **test_challenge.py** was created.

```
import torch
from dataset import get_train_val_test_loaders
from model.challenge import Challenge

from train_common import *
import utils

def main():

    tr_loader, va_loader, te_loader, _ = get_train_val_test_loaders(
        task="target",
        batch_size=config("challenge.batch_size"),
    )

    model = Challenge()
    criterion = torch.nn.CrossEntropyLoss()

    print("Loading challenge...")
    model, start_epoch, stats = restore_checkpoint(model,
config("challenge.checkpoint"))

    axes = utils.make_training_plot()

    evaluate_epoch(
```

```
        axes,
        tr_loader,
        va_loader,
        te_loader,
        model,
        criterion,
        start_epoch,
        stats,
        include_test=True,
        update_plot=False,
    )


if __name__ == "__main__":
    main()
```

The performance of the default model is given below:

Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
9	0.9981	0.9776	0.7084	0.66

Note that to measure the performance, execute **train_challenge.py** first, then select the epoch with the lowest validation loss, and finally execute **test_challenge.py**.

5.1 Learning rate

Learning rate	Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
1e-2	19	0.5	0.5	0.5	0.5
5e-3	8	0.9851	0.9822	0.592	0.49
Default: 1e-3	9	0.9981	0.9776	0.7084	0.66
5e-4	13	0.9992	0.9799	0.734	0.67
1e-4	43	0.9991	0.9684	0.7308	0.68
5e-5	68	0.9965	0.9657	0.7244	0.68

It can be concluded that when **Learning rate** = 5e-4, Test AUROC reaches maximum: 0.734.

5.2 Optimizer

Optimizer	Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
Default: Adam(lr=1e-3)	9	0.9981	0.9776	0.7084	0.66
Adam(lr=1e-3, weight_decay=0.01)	11	0.9998	0.9812	0.7292	0.64
SGD(lr=1e-3, momentum=0.9)	29	0.9939	0.9739	0.7164	0.66
SGD(lr=1e-3, momentum=0.9, weight_decay=0.01)	35	0.996	0.9748	0.72	0.67
RMSprop(lr=1e-3)	14	0.9701	0.9788	0.594	0.51
RMSprop(lr=1e-3, weight_decay=0.01)	20	0.979	0.9879	0.6088	0.51

It can be concluded that when **Optimizer** = Adam(lr=1e-3, weight_decay=0.01), Test AUROC reaches maximum.

Combined with result in 5.1, we select **Optimizer** = Adam(lr=5e-4, weight_decay=0.01). Its performance is as follows:

Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
16	0.9996	0.9799	0.7308	0.66

5.3 Activation function

Based on the default model, several common activation functions were used to test whether it can improve Test AUROC.

Activation function	Code	Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
Default: RELU	F.relu()	9	0.9981	0.9776	0.7084	0.66
ELU	F.elu(*alpha=1*)	9	1.0	0.9772	0.6848	0.6
SoftPlus	F.softplus(*beta=1*)	15	0.9949	0.966	0.752	0.67
LReLU	F.leaky_relu(*negative_slope=0.01*)	9	0.999	0.9788	0.7112	0.64
Sigmoid	torch.sigmoid()	69	0.9944	0.9783	0.5944	0.52
Tanh	torch.tanh()	9	0.9948	0.9739	0.6528	0.59

Although the default **Activation function**: *RELU* is commonly used in many well-known CNNs, such as AlexNet and VGG, it seems that SoftPlus is the best in this case since it results in the highest Test AUROC: 0.752.

Combined with result in 5.2, we select **Optimizer** = Adam(lr=5e-4, weight_decay=0.01) and **Activation function**: SoftPlus. Its performance is as follows:

Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
20	0.9957	0.96	0.766	0.71

5.4 Data augmentation

Based on the default model, we tested different augmentation techniques.

	Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
Rotation (keep original)	7	1.0	0.981	0.7368	0.68
Grayscale (keep original)	5	1.0	0.9563	0.7832	0.69
Grayscale (discard original)	8	0.9907	0.8284	0.7736	0.69
Rotation and Grayscale (keep original)	5	0.9957	0.9577	0.7552	0.67
Rotation and Grayscale (discard original)	4	0.8614	0.8492	0.706	0.62

It can be concluded that Grayscale (keep original) is the best augmentation technique.

Again, combined with result in 5.3, with **Optimizer** = Adam(lr=5e-4, weight_decay=0.01) and **Activation function**: SoftPlus, and **Augmentation technique**: Grayscale (keep original), the model's performance is as follows:

Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
8	0.9946	0.9118	0.778	0.7

5.5 Transfer Learning

In the script **train_challenge.py**, we added following line in the front.

```
from train_target import freeze_layers, train
```

Also, we added following code right below the line: model = Challenge()

```
    num_layers = 2 # TODO
    print("Loading source...")
    model, _, _ = restore_checkpoint(
        model, config("source.checkpoint"), force=True, pretrain=True
    )
    freeze_layers(model, num_layers)
    train(tr_loader, va_loader, te_loader, model, "./checkpoints/challenge/",
num_layers)
    return
```

num_layers indicates the number of layers to be frozen. It can be changed in the range 0 - 3, which corresponds to "Freeze no layers" - "Freeze all CONV layers".

In the script **train_target.py**, we also modified the optimizer in the function train().

```
    criterion = torch.nn.CrossEntropyLoss()
    #optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    optimizer = torch.optim.Adam(model.parameters(), lr=5e-4, weight_decay=0.01)
```

When executing the script **train_challenge.py**, enter 'y' for the question: "Augmented data found, would you like to use it? y/n". It indicates that **Augmentation technique**: Grayscale (keep original) is applied.

And enter 10 for the question "Which epoch to load from? Choose in range [1, 20]." since it's the epoch with the lowest validation loss for Source().

	Epoch with the lowest validation loss	TRAIN AUROC	VAL AUROC	TEST AUROC
Freeze all CONV layers	205	0.8049	0.8539	0.7828
Freeze first two CONV layers	24	0.975	0.9632	0.8
Freeze first CONV layer	13	0.9946	0.949	0.7984
Freeze no layers	11	0.9908	0.9586	0.7844
No Transfer Learning (in section 5.4)	8	0.9946	0.9118	0.778

From the table, we can conclude that Freezing first two CONV layers is the best when **Transfer Learning**.

Based on **Transfer Learning** that freezes first two CONV layers, we also tried to add a new layer nn.Dropout() to test whether it can improve the performance further. Define the dropout layer as:

```
self.dropout = nn.Dropout(p=0.1, inplace=False)
```

Note that the parameter p=0.1 will be changed. And the following line of code is added right before z = *self*.fc_1(z)

```
z = self.dropout(z)
```

p	Epoch with the lowest validation loss	TRAIN AUROC	VAL AUROC	TEST AUROC
0.2	24	0.9464	0.9751	0.7756
0.1	14	0.9375	0.9627	0.7932
0.05	12	0.9308	0.9694	0.7572
0.01	17	0.9564	0.9596	0.7964

From the table, all **TEST AUROC** < 0.8. Thus, we do not add the dropout layer.

5.6 Model architecture

Design the class Challenge() in **challenge.py** as follows:

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

        ## TODO: define each layer
        self.conv1 = nn.Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.pool = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0)
        self.conv2 = nn.Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv3 = nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.conv4 = nn.Conv2d(64, 8, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        self.fc_1 = nn.Linear(2048, 2)
        ##

        self.init_weights()

    def init_weights(self):
        for conv in [self.conv1, self.conv2, self.conv3, self.conv4]:
            C_in = conv.weight.size(1)
            nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5 * 5 * C_in))
            nn.init.constant_(conv.bias, 0.0)

        ## TODO: initialize the parameters for [self.fc_1]
        nn.init.normal_(self.fc_1.weight, 0.0, 1 / sqrt(32))
        nn.init.constant_(self.fc_1.bias, 0.0)
        ##
```

```
    def forward(self, x):
        N, C, H, W = x.shape
        activ_func = F.softplus
        ## TODO: forward pass
        z = activ_func(self.conv1(x))
        z = self.pool(z)
        z = activ_func(self.conv2(z))
        z = self.pool(z)
        z = activ_func(self.conv3(z))
        z = activ_func(self.conv4(z))
        z = torch.reshape(z, (N, 2048))
        z = self.fc_1(z)

        return z
```

The design of the model uses VGG for reference. Thus, we set kernel_size=(3, 3), stride=(1, 1), padding=(1, 1) in nn.Conv2d. And we only set two layers of nn.MaxPool2d in the function forward() to avoid dropping too much data.

Then the **Optimizer** is modified to the default one.

```
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

With **Activation function**: *SoftPlus* and **Augmentation technique**: Grayscale (keep original), we got the performance:

Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
13	0.9695	0.8571	0.85	0.75

Optimizer which works best for the default model in section 5.3 was also tested .

```
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=5e-4, weight_decay=0.01)
```

With **Activation function**: *SoftPlus* and **Augmentation technique**: Grayscale (keep original), we got the performance:

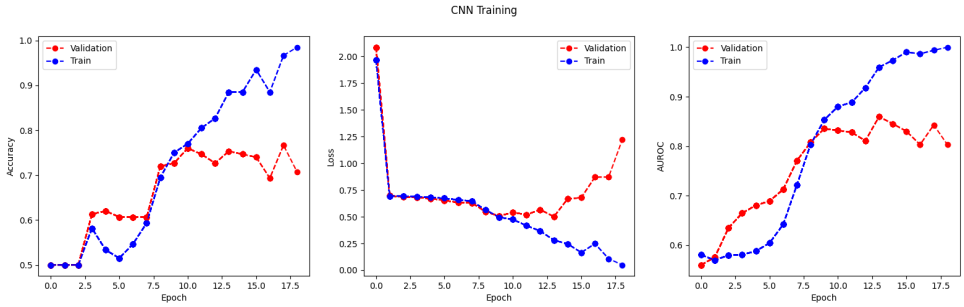
Epoch with the lowest validation loss	Train AUROC	Validation AUROC	Test AUROC	Test Accuracy
8	0.8721	0.866	0.7716	0.71

It's worth noting that the performance becomes worse.

Actually, the design of the model took a lot of time to modify and test. For example, more nn.Conv2d layers can be added and the number of filters at each layer can be changed, and so on. But this process is omitted since it's too long, and its method is similar.

5.7 Predict Challenge

Comparing the best models in section 5.5 and 5.6, we can conclude that the model in section 5.6 works better since its **Test AUROC** = 0.85 > 0.8. The following figure is its training plot.



Then the script **predict_challenge.py** was executed. Note that epoch 13 was selected. The challenge is done!