

# Week 04-05: Intro to Deep Learning

## Intensive Summary

### 1\_Image\_classification\_CIFAR10\_CNN

We load the data set from CIFAR10 (The images in CIFAR-10 are of size  $32 \times 32 \times 3$ ) which is a common data set in torchvision dataset and transform it into tensor and resize then we split the data set into 40000 and 10000.

```
trainvalset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
trainset, valset = torch.utils.data.random_split(trainvalset, [40000, 10000])

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size, shuffle=False)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
```

By subclassing nn.Module, we are defining a neural network. The function `__init__` allows us to define the constructor of this class, while the `__forward__` function defines steps of how this neural network will operate. Then, we create an example of CNN and move it to the device. Our models consists of 8 steps as follow:

1. Convolute (3 input channels , 6 output channels, kernel size =  $5 \times 5$ ,  $32 \times 32$  input 2d matrix,  $28 \times 28$  output 2d matrix) then us ReLU as activation function
2. Applies Max pooling (kernel size =  $2 \times 2$ , stride = 2,  $28 \times 28$  input 2d matrix,  $14 \times 14$  output 2d matrix)
3. Convolute (6 input channels , 16 output channels, kernel size =  $5 \times 5$ ,  $14 \times 14$  input 2d matrix,  $10 \times 10$  output 2d matrix) then use ReLU as activation function
4. Applies Max pooling kernel size =  $2 \times 2$ , stride = 2 (tensor size input =  $10 \times 10$  output =  $5 \times 5$ ) and then flatten 16 channels,  $5 \times 5$  tensor into array of 400 values
5. Applies Linear transformation turn 400 features to 120 features
6. Applies Linear transformation turn 120 features to 84 features
7. Applies Linear transformation turn 84 features to 10 features
8. Applies the Softmax function by row

```

class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(400, 120) # dense input 400 (16*5*5), output 120
        self.fc2 = nn.Linear(120, 84) # dense input 120, output 84
        self.fc3 = nn.Linear(84, 10) # dense input 84, output 10
        self.softmax = torch.nn.Softmax(dim=1) # perform softmax at dim[1] (batch,class)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # relu as an activation function
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, start_dim=1) # flatten all dimensions (dim[1]) except batch
        (dim[0])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.softmax(x)
        return x

net = CNN().to(device)

```

Print the summary of all inputs and outputs of every layer in this neural network when the input size is 32\*3\*32\*32.

```

from torchinfo import summary as summary_info
print(summary_info(net, input_size = (32, 3, 32, 32))) # (batchsize,channel,width,height)

```

As we can see the output of this NN is 10 because our data must be classified into any of the 10 classes.

```

=====
Layer (type:depth-idx)                   Output Shape          Param #
=====
CNN                                     [32, 10]              --
├─Conv2d: 1-1                          [32, 6, 28, 28]       456
├─MaxPool2d: 1-2                       [32, 6, 14, 14]       --
├─Conv2d: 1-3                          [32, 16, 10, 10]      2,416
├─MaxPool2d: 1-4                       [32, 16, 5, 5]        --
├─Linear: 1-5                          [32, 120]             48,120
├─Linear: 1-6                          [32, 84]              10,164
├─Linear: 1-7                          [32, 10]              850
└─Softmax: 1-8                       [32, 10]              --
=====
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
Total mult-adds (M): 21.06
=====
Input size (MB): 0.39

```

```
Forward/backward pass size (MB): 1.67
Params size (MB): 0.25
Estimated Total Size (MB): 2.31
=====
```

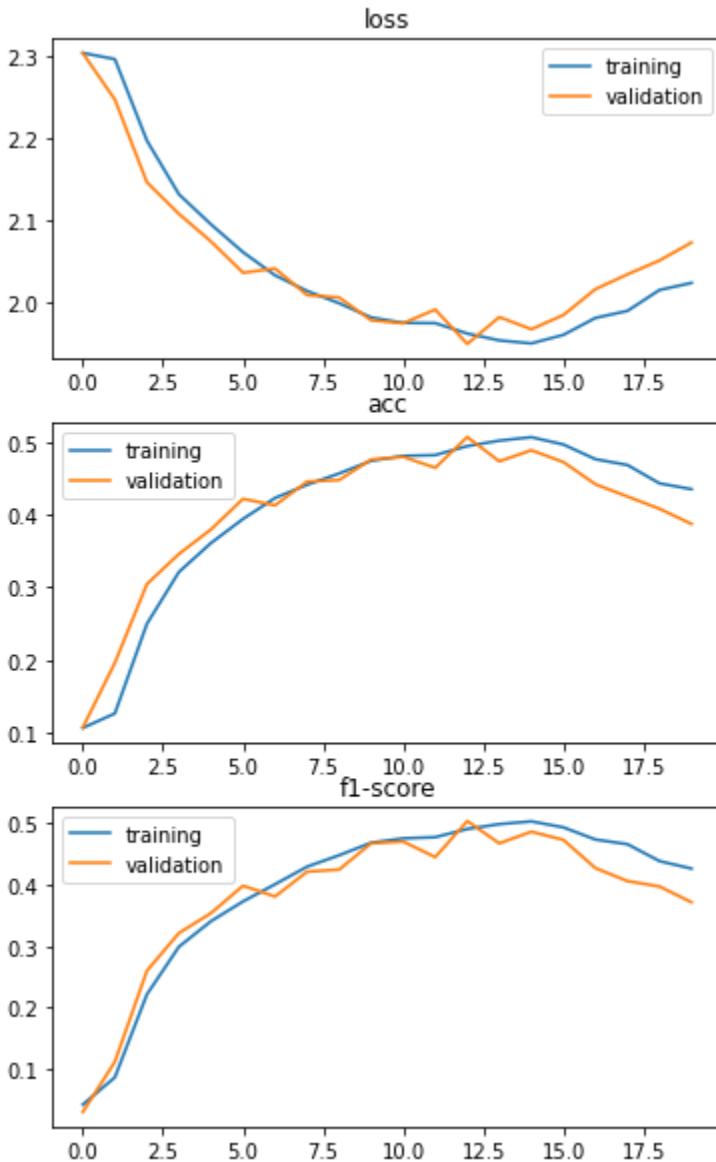
We set CrossEntropyLoss function to be matrix function and optimizer to be SGD.

```
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=1e-2, momentum=0.9)
```

Set the epochs = 20. Set the model to training mode loop iterates over batches size of the data by using **tqdm** (display a progress bar). then we move the data to the device. Set the gradient of all parameters to zero. Run the data through the NN to get the output and calculate the loss then compute the gradient for all parameters with respect to the loss. Update the weight. Use our model to predict an evaluation set and collect the result(loss, accuracy, f1-score). And then we plot the graph.

```
epochs = 20
history_train = {'loss':np.zeros(epochs), 'acc':np.zeros(epochs), 'f1-score':np.zeros(epochs)}
history_val = {'loss':np.zeros(epochs), 'acc':np.zeros(epochs), 'f1-score':np.zeros(epochs)}
min_val_loss = 1e10
PATH = './CNN_CIFAR10.pth'
for epoch in range(epochs): # loop over the dataset multiple times
    print(f'epoch {epoch + 2} \nTraining ...')
    y_predict = list()
    y_labels = list()
    training_loss = 0.0
    n = 0
    net.train() # set model to training mode
    for data in tqdm(trainloader):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer.zero_grad() # set the gradient of all model parameters to zero
        outputs = net(inputs) # forward pass
        loss = criterion(outputs, labels) # calculate loss from forward pass
        loss.backward() # gradient are computed for all parameters with respect to the loss
        optimizer.step() # update weights
```

X axis represents epoch while y axis represents loss (figure 1), accuracy (figure 2), or f1-score (figure 3). We expect loss to decrease and the graph shows that loss plummets and reaches a minimum before arriving at the last epoch. The graph also shows that the loss of training and validation set do not differ significantly, meaning that our model is not overfitted. On the other hand, we expect the accuracy and f1-score to increase and the graphs shows that both skyrocket and reach maximum before the last epoch. Similar to the loss graph, both accuracy and f1-score of the training and validation set do not differ significantly, another evidence that advocates that our model is not overfitted.



We now use our model to predict testing sets, collect the results, then use a confusion matrix to evaluate our model. Show the result.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

print('testing ...')
y_predict = list()
y_labels = list()
test_loss = 0.0
n = 0
with torch.no_grad():
    for data in tqdm(testloader):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        y_labels += list(labels.cpu().numpy())
        y_predict += list(outputs.argmax(dim=1).cpu().numpy())
        n+=1

# print statistics
test_loss /= n
print(f"testing loss: {test_loss:.4}" )
report = classification_report(y_labels, y_predict, digits = 4)
M = confusion_matrix(y_labels, y_predict)
print(report)
disp = ConfusionMatrixDisplay(confusion_matrix=M)
```

```
testing loss: 1.953
           precision    recall  f1-score   support

     0       0.5345       0.5880       0.5600        1000
     1       0.5241       0.7290       0.6098        1000
     2       0.4115       0.3720       0.3908        1000
     3       0.3542       0.3230       0.3379        1000
     4       0.4986       0.3640       0.4208        1000
     5       0.4062       0.4850       0.4421        1000
     6       0.5854       0.6000       0.5926        1000
     7       0.5797       0.5820       0.5808        1000
     8       0.5421       0.5990       0.5691        1000
     9       0.6378       0.4050       0.4954        1000

 accuracy                   0.5047        10000
  macro avg       0.5074       0.5047       0.4999        10000
 weighted avg       0.5074       0.5047       0.4999        10000
```

## 2\_Image\_classification\_Animal\_EfficientNetV2

First we create list of transforms compose of 7 operation:

1. Random rotation 30 degree
2. Random crop image to size 224\*224
3. Horizontal flip the given image with 0.5 probability
4. Make the data to tensor
5. Normalize each channel by transforming a color value of [0,255] to a tensor with a range of [0,1]. We input means and standard deviations of each channel respectively.

```
1 transform_train = transforms.Compose(  
2     [transforms.Resize((230,230)),  
3       transforms.RandomRotation(30,),  
4       transforms.RandomCrop(224),  
5       transforms.RandomHorizontalFlip(),  
6       transforms.RandomVerticalFlip(),  
7       transforms.ToTensor(),  
8       transforms.Normalize(mean=[0.507, 0.487, 0.441], std=[0.267, 0.256, 0.276]) #nomalize imagenet pretrain  
9     ])  
10  
11 transform = transforms.Compose(  
12     [transforms.Resize((224,224)),  
13       transforms.ToTensor(),  
14       transforms.Normalize(mean=[0.507, 0.487, 0.441], std=[0.267, 0.256, 0.276])  
15     ])  
16  
17 batch_size = 32
```

Define class AnimalDataset

1. Define parameters: `img_dir` and `transforms` (in which default=None)
2. Using `super().__init__()` to inherit constructors of class `dataset`
3. In loops, `self.input_dataset` will collect the image path and its label.
4. Define the method of `__len__` (return len of `self.input_dataset`)
5. Define the method of `__getitem__` (return transformed image and label number of that picture)

```
1 class AnimalDataset(Dataset):
2
3     def __init__(self,
4                 img_dir,
5                 transforms=None):
6
7         super().__init__()
8         label_image = ['butterfly', 'cat', 'chicken', 'cow', 'dog', 'elephant', 'horse', 'sheep', 'spider', 'squirrel']
9         self.input_dataset = list()
10        label_num = 0
11        for label in label_image:
12            _, _, files = next(os.walk(os.path.join(img_dir, label)))
13            for image_name in files:
14                input = [os.path.join(img_dir, label, image_name), label_num] # [image_path, label_num]
15                self.input_dataset.append(input)
16            label_num += 1
17
18        self.transforms = transforms
19
20    def __len__(self):
21        return len(self.input_dataset)
22
23    def __getitem__(self, idx):
24        img = Image.open(self.input_dataset[idx][0]).convert('RGB')
25        x = self.transforms(img)
26        y = self.input_dataset[idx][1]
27        return x, y
```

Set pretrain weight to be IMAGENET1K\_V1, model to be EfficientNet, and weight of our model to be `pertain_weight`. Set output layers to be probabilities of 10 classes.

```
1 pretrain_weight = torchvision.models.EfficientNet_V2_S_Weights.IMAGENET1K_V1
2 net = torchvision.models.efficientnet_v2_s(weights = pretrain_weight)
3 net.classifier[1] = nn.Linear(1280, 10)
4 net = net.to(device)
```

Summarize our model and print all of the layers in the model.




```
1 summary(net, (3, 224, 224), batch_size = 64)
```

```
-----
      Layer (type)                Output Shape          Param #
=====
      Conv2d-1                    [64, 24, 112, 112]      648
      BatchNorm2d-2               [64, 24, 112, 112]       48
      SiLU-3                      [64, 24, 112, 112]        0
      Conv2d-4                    [64, 24, 112, 112]    5,184
      BatchNorm2d-5               [64, 24, 112, 112]       48
      SiLU-6                      [64, 24, 112, 112]        0
      StochasticDepth-7           [64, 24, 112, 112]        0
      FusedMBConv-8               [64, 24, 112, 112]        0
      Conv2d-9                    [64, 24, 112, 112]    5,184
      BatchNorm2d-10              [64, 24, 112, 112]       48
      SiLU-11                     [64, 24, 112, 112]        0
      StochasticDepth-12          [64, 24, 112, 112]        0
      FusedMBConv-13              [64, 24, 112, 112]        0
      Conv2d-14                   [64, 96, 56, 56]    20,736
      BatchNorm2d-15              [64, 96, 56, 56]       192
      SiLU-16                     [64, 96, 56, 56]        0
      Conv2d-17                   [64, 48, 56, 56]     4,608
      BatchNorm2d-18              [64, 48, 56, 56]        96
      FusedMBConv-19              [64, 48, 56, 56]        0
      Conv2d-20                   [64, 192, 56, 56]    82,944
      BatchNorm2d-21              [64, 192, 56, 56]       384
      SiLU-22                     [64, 192, 56, 56]        0
      ...
Forward/backward pass size (MB): 20629.03
Params size (MB): 77.02
Estimated Total Size (MB): 20742.80
-----
```



Next, we set the criterion to be CrossEntropyLoss and the optimizer to be SGD. Sets up a learning rate scheduler for an optimizer. The learning rate is adjusted during training using a stepwise decay strategy. Every 7 epochs, the learning rate is reduced by a factor of 0.5. The rest of the steps are similar to the first classification model.



```
1 criterion = nn.CrossEntropyLoss()
2 optimizer = optim.SGD(net.parameters(), lr=0.02, momentum=0.9)
3 scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.5)
```