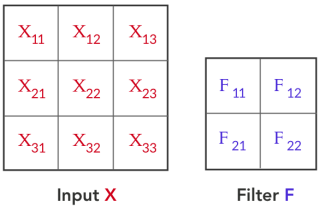
Student Name: Arash Sajjadi
Student ID: 400422096

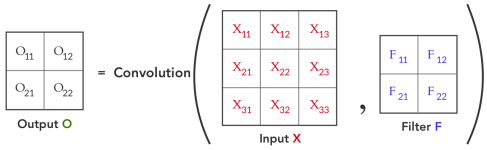
Artificial Neural Network (ANN)
Assignment II

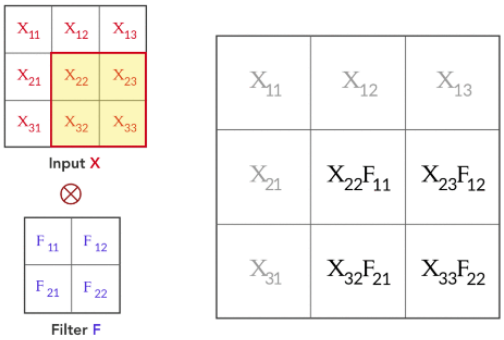
Question 1

Lets assume the function f is a convolution between Input X and a Filter F. Input X is a 3×3 matrix, and Filter F is a 2×2 matrix, as shown below:



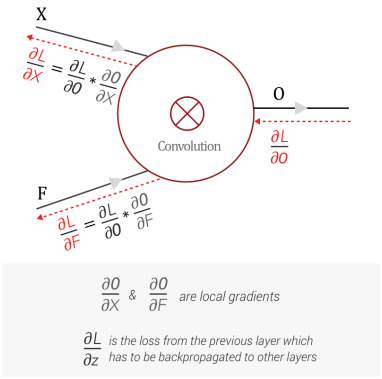
Convolution between Input X and Filter F, gives us an output O. This can be represented as:





$$
\begin{aligned}
O_{11} &= X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22} \\
O_{12} &= X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22} \\
O_{21} &= X_{21}F_{11} + X_{22}F_{12} + X_{31}F_{21} + X_{32}F_{22} \\
O_{22} &= X_{22}F_{11} + X_{23}F_{12} + X_{32}F_{21} + X_{33}F_{22}
\end{aligned}
$$

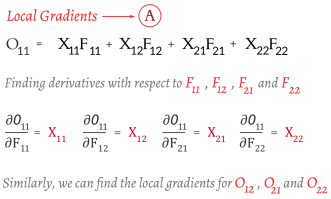
This gives us the forward pass! Let's get to the Backward pass. As mentioned earlier, we get the loss gradient with respect to the Output O from the next layer as $\frac{\partial L}{\partial O}$, during the Backward pass. And combining with our previous knowledge using the Chain rule and Backpropagation, we get:



As seen above, we can find the local gradients $\frac{\partial O}{\partial X}$ and $\frac{\partial O}{\partial F}$ with respect to Output O., And with loss gradient from previous layers $\frac{\partial L}{\partial O}$ and using the chain rule, we can calculate $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial F}$.

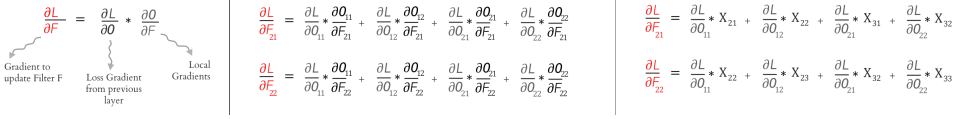
So let's find the gradients.

Step 1: Finding the local gradient $\frac{\partial O}{\partial F}$: This means we have to differentiate Output Matrix O with Filter F. From our convolution operation,



Step 2: Using the Chain rule:

As described in our previous examples, we need to find $\frac{\partial O}{\partial F}$ as:



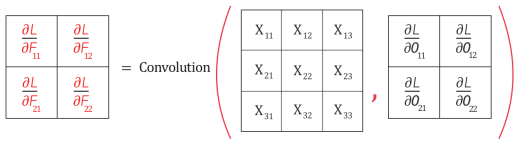
We can describe it as a convolution operation between input X and loss gradient $\frac{\partial L}{\partial O}$ as shown below:

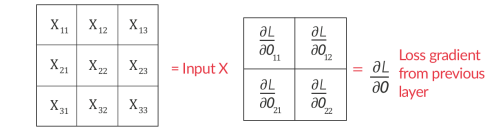
Finding $\frac{\partial L}{\partial X}$:

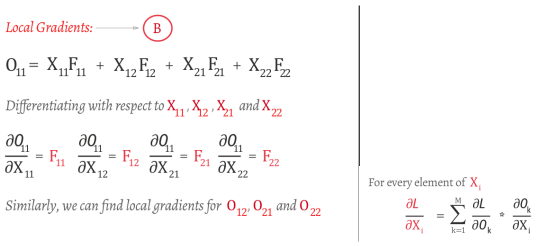
Step 1: Finding the local gradient $\frac{\partial o}{\partial X}$: (Similar to how we found the local gradients earlier, we can find $\frac{\partial o}{\partial X}$ as:)

Expanding this and substituting from Equation B, we get:

even this can be represented as a convolution operation.







$$\frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial O_{11}} * F_{11}$$

$$\frac{\partial L}{\partial X_{12}} = \frac{\partial L}{\partial O_{11}} * F_{12} + \frac{\partial L}{\partial O_{12}} * F_{11}$$

$$\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial O_{12}} * F_{12}$$

$$\frac{\partial L}{\partial X_{21}} = \frac{\partial L}{\partial O_{11}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{11}$$

$$\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial O_{11}} * F_{22} + \frac{\partial L}{\partial O_{12}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{12} + \frac{\partial L}{\partial O_{22}} * F_{11}$$

$$\frac{\partial L}{\partial X_{23}} = \frac{\partial L}{\partial O_{12}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{12}$$

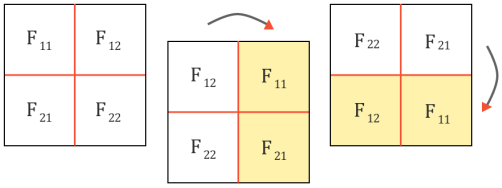
$$\frac{\partial L}{\partial X_{31}} = \frac{\partial L}{\partial O_{21}} * F_{21}$$

$$\frac{\partial L}{\partial X_{32}} = \frac{\partial L}{\partial O_{21}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{21}$$

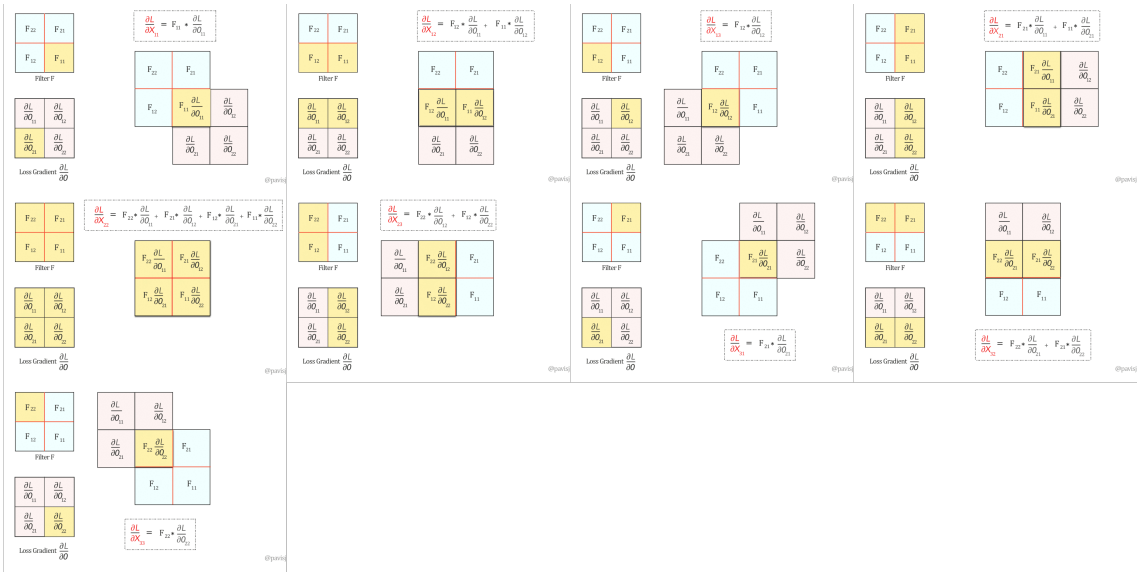
$$\frac{\partial L}{\partial X_{33}} = \frac{\partial L}{\partial O_{22}} * F_{22}$$

$\frac{\partial L}{\partial X}$ can be represented as 'full' convolution between a 180-degree rotated Filter F and loss gradient $\frac{\partial L}{\partial O}$

Flipping 180 degrees:



Now, let us do a 'full' convolution between this flipped Filter F and $\frac{\partial L}{\partial O}$, which can be visualized as below: (It is like sliding one matrix over another from right to left, bottom to top)



The full convolution above generates the values of $\frac{\partial L}{\partial X}$ and hence we can represent $\frac{\partial L}{\partial X}$ as:

$$\frac{\partial L}{\partial X} = \begin{bmatrix} \frac{\partial L}{\partial X_{11}} & \frac{\partial L}{\partial X_{12}} & \frac{\partial L}{\partial X_{13}} \\ \frac{\partial L}{\partial X_{21}} & \frac{\partial L}{\partial X_{22}} & \frac{\partial L}{\partial X_{23}} \\ \frac{\partial L}{\partial X_{31}} & \frac{\partial L}{\partial X_{32}} & \frac{\partial L}{\partial X_{33}} \end{bmatrix} = \text{Full Convolution} \left(\begin{bmatrix} F_{22} & F_{21} \\ F_{12} & F_{11} \end{bmatrix}_{\text{Filter F}}, \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix}_{\text{Loss Gradient } \frac{\partial L}{\partial O}} \right)$$

Well, now that we have found $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial F}$, we can now come to this conclusion

Both the Forward pass and the Backpropagation of a Convolutional layer are Convolutions

Summing it up: CNN uses back-propagation and the back propagation is not a simple derivative like ANN but it is a convolution operation as given below.

Backpropagation in a Convolutional Layer of a CNN

Finding the gradients:

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input X}, \text{Loss gradient } \frac{\partial L}{\partial O}\right)$$

$$\frac{\partial L}{\partial X} = \text{Full Convolution}\left(\text{180° rotated Filter F}, \text{Loss Gradient } \frac{\partial L}{\partial O}\right)$$

Question 2

This expression refers to the time when the weights of the neural network have a specific initialization, so in the process of updating the weights, all the weights change in the same way and may

not learn many complexities. It is better to use random initial weighting for the neural network to deal with this problem.

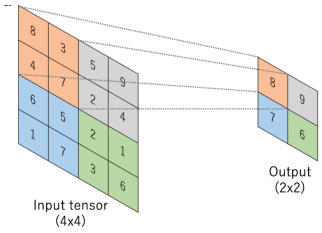
Question 3

Benefits of the pooling layers: Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of learning parameters and the amount of computation performed in the network. The pooling layer summarises the features present in a region of the feature map generated by a convolution layer.

Drawbacks of the pooling layers: A perfect pooling method can extract only valuable information and discard inappropriate features. Despite these advantages of the pooling layer, it still has some drawbacks as losing features and reducing the spatial resolution, which could affect the accuracy of the classification systems.

Are you willing to use these layers? Definitely, to use convolutional networks, we must use pooling layers correctly and sufficiently. Adding a pooling layer after the convolutional layer is a typical pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model.

Can you use these layers frequently? As I mentioned, the use of these pooling layers should be correct. If we use it repeatedly, we will lose valuable information.



Question 4

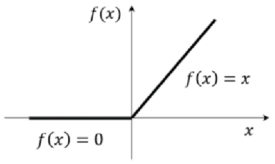
When we use the quadratic cost, learning is slower when the neuron is unambiguously wrong than it is later on as the neuron gets closer to the correct output. In contrast, with cross-entropy, learning is faster when the neuron is unambiguously wrong.

Also, I prefer to use the cross-entropy cost for classification problems. The reason behind that is the cross-entropy loss helps to distinguish between wrong and very wrong predictions. (Also correct predictions and correct predictions with high confidence)

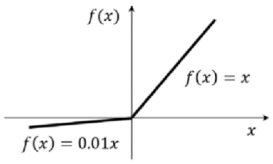
Question 5

I checked various sources to answer your question. If I want to answer in one sentence, the answer entirely depends on the problem and the data we work with. It is impossible to give a general answer. However, many sources believe that ReLU is faster. Many believe leaky ReLU performs better because it does not lose negative information. But the experience of practical work with data shows that it is not necessarily so.

For the second question, leaky ReLU prevents gradient vanishing. Because in ReLU, all numbers less than zero go to a constant value of 0. As a result, we are facing the phenomenon of vanishing gradients.



ReLU activation function



LeakyReLU activation function

Question 6

1. **Report the depth effect of different hidden layers**: My experience working with these data showed that three convolutional layers are suitable. In addition, the use of batchNorm2 and Dropout2d increased the final accuracy to a reasonable extent
2. **Use batch normalization for convolution layers and report its effects**: Done ✓
3. **Use different architectures to find the optimum model for achieving maximum accuracy**: Done ✓
4. **The confusion matrix should be calculated and analyzed for your model**: Done ✓

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sn
import pandas as pd
from prettytable import PrettyTable
PTable= PrettyTable()
from IPython.display import clear_output
PTable.field_names = ["Episode","Step", "Loss"]
```

[2]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
num_epochs = 8
batch_size = 32
learning_rate = 0.001
```

[3]:
```
# dataset has PILImage images of range [0, 1].
# We transform them to Tensors of normalized range [-1, 1]
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# CIFAR10: 60000 32x32 color images in 10 classes, with 6000 images per class
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                          shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                         shuffle=False)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(train_loader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
```

```
Files already downloaded and verified
Files already downloaded and verified
```

torch.Size([4, 3, 32, 32]) torch.Size([4, 6, 14, 14]) torch.Size([4, 16, 5, 5])

[4]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3,   64,  3)
        self.batchNorm1=nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64,  128, 3)
        self.batchNorm2=nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, 3)
        self.batchNorm3=nn.BatchNorm2d(256)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 4 * 4, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 10)
        self.Dropout = nn.Dropout2d(p=0.1)
    def forward(self, x):
        x = self.pool(F.relu(self.batchNorm1(self.conv1(x))))
        x = self.pool(F.relu(self.batchNorm2(self.conv2(x))))
        x = self.Dropout(x)
        x = self.pool(F.relu(self.batchNorm3(self.conv3(x))))
        x = x.view(-1, 64 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)

model = CNN().to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[5]:
```
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [4, 3, 32, 32] = 4, 3, 1024
        # input_layer: 3 input channels, 6 output channels, 5 kernel size
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 500 == 0:
          PTable.add_row([str(epoch+1)+"/"+str(num_epochs),str(i+1)+"/
 ↪"+str(n_total_steps),round(loss.item(),4)])
          clear_output()
          print(PTable.get_string(header=True))
          #print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
 ↪{n_total_steps}], Loss: {loss.item():.4f}')



print('Finished Training')
PATH = './cnn.pth'
torch.save(model.state_dict(), PATH)
```

```
+---------+-----------+--------+
| Episode |    Step   |  Loss  |
+---------+-----------+--------+
|   1/8   |  500/1563 | 1.0783 |
|   1/8   | 1000/1563 | 1.3102 |
|   1/8   | 1500/1563 | 0.6962 |
|   2/8   |  500/1563 | 0.9929 |
|   2/8   | 1000/1563 | 1.2287 |
|   2/8   | 1500/1563 | 0.8617 |
|   3/8   |  500/1563 |  0.78  |
|   3/8   | 1000/1563 | 0.578  |
|   3/8   | 1500/1563 | 1.399  |
|   4/8   |  500/1563 | 0.6582 |
|   4/8   | 1000/1563 | 0.7764 |
|   4/8   | 1500/1563 | 1.1192 |
|   5/8   |  500/1563 | 0.6798 |
|   5/8   | 1000/1563 | 0.5597 |
```

```
|   5/8   | 1500/1563 | 0.4598 |
|   6/8   |  500/1563 | 0.3155 |
|   6/8   | 1000/1563 | 0.5189 |
|   6/8   | 1500/1563 | 0.6751 |
|   7/8   |  500/1563 | 0.6727 |
|   7/8   | 1000/1563 | 0.4385 |
|   7/8   | 1500/1563 | 0.3012 |
|   8/8   |  500/1563 | 0.386  |
|   8/8   | 1000/1563 | 0.4715 |
|   8/8   | 1500/1563 | 0.4819 |
+---------+-----------+--------+
Finished Training
```

[6]:
```python
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    n_class_correct = [0 for i in range(10)]
    n_class_samples = [0 for i in range(10)]
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

        for i in range(batch_size):
            if i>labels.shape[0]-1:
              break
            label = labels[i]
            pred = predicted[i]
            if (label == pred):
                n_class_correct[label] += 1
            n_class_samples[label] += 1

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network: {acc} %')

    for i in range(10):
        acc = 100.0 * n_class_correct[i] / n_class_samples[i]
        print(f'Accuracy of {classes[i]}: {acc} %')
```

```
Accuracy of the network: 76.6 %
Accuracy of plane: 80.2 %
Accuracy of car: 87.1 %
Accuracy of bird: 62.0 %
Accuracy of cat: 57.4 %
```

```
Accuracy of deer: 68.4 %
Accuracy of dog: 74.8 %
Accuracy of frog: 87.8 %
Accuracy of horse: 80.7 %
Accuracy of ship: 81.2 %
Accuracy of truck: 86.4 %
```

[7]:
```
# Confusion matrix construction
confusion_matrix = torch.zeros(10, 10)
with torch.no_grad():
    for i, (images, labels) in enumerate(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        for t, p in zip(labels.view(-1), preds.view(-1)):
                confusion_matrix[t.long(), p.long()] += 1
# Confusion matrix visualization
array = confusion_matrix.numpy()
df_cm = pd.DataFrame(array, index = [i for i in classes],
                  columns = [i for i in classes])
plt.figure(figsize = (10,7))
sn.heatmap(df_cm, annot=True ,fmt='g')
plt.ylabel("True labels")
plt.xlabel("Predicted labels")
plt.title("Confusion Matrix")
plt.show()
```