Google Classroom Code: mhxgl24

Convolution Neural Networks

Deep Learning (DS-5006)

Dr. Adeel Mumtaz

Lecture 8

Fall, 2022

National University Of Computer and Emerging Sciences

Contents

- Last Lecture
  - Torchvision
    - ImageFolder Dataset
    - Image transformations
    - Conversion transforms
    - Normalization Transforms
    - Data Augmentation
- Problems in using ANN Architecture for images
  - Pixel Flattening
  - Too many parameters
- CNN (Theory)
  - History
    - LeNet
    - AlexNet
  - CNNs / ConvNets Architecture
    - Convolution Layer
      - Striding
      - Padding
    - Pooling Layer
    - Flattening Layer
    - Multi-Channel Convolutions
    - Multiple Filters
    - ReLU Layer
    - FC Layer
    - Dropout Layer
- CNN (Code)
  - Imports
  - Datasets/Dataloaders
  - Model Class
  - Training Loops
  - Results
- Graded Home Task-5

TORCHVISION

Introduction

- Package Containing Computer Vision Popular
  - Datasets
    - Like MNIST, ImageNet, CIFAR
    - Inherit from the original Dataset class
    - Can be naturally used with a DataLoader
  - Model architectures
    - Including its pretrained weights
    - AlexNet, VGG, ResNet (ResNet18, ResNet34, ResNet50, ResNet101, ResNet152), and Inception V3
  - Image transformations
    - Transformations based on images (either in PIL or PyTorch shapes)
    - Transformations based on Tensors
    - Conversion transforms to convert from tensor **ToPILImage** and from PIL image **ToTensor**

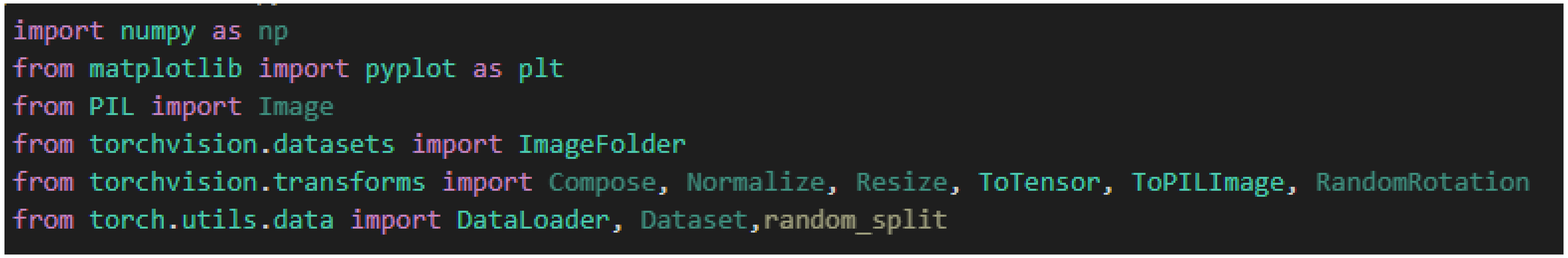
Image Transformations

- ToTensor()
  - Convert a PIL Image or numpy.ndarray to tensor
- ToPILImage()
  - Convert a tensor or an ndarray to PIL Image

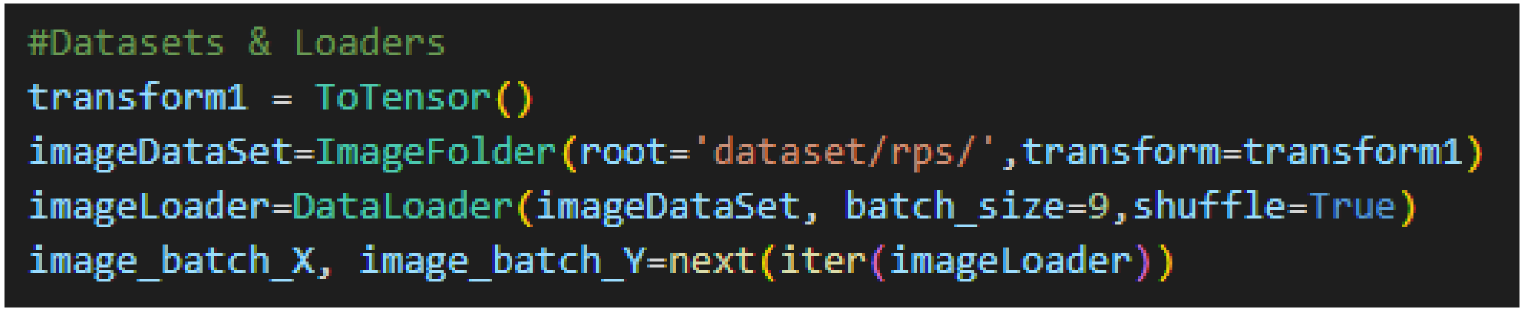
Image Datasets

- ImageFolder
  - a generic dataset that you can use with your own images organized into sub-folders
  - Each sub-folder named after a class and containing the corresponding images
  - The "Rock-Paper-Scissors" dataset is organized like that:
    - Inside the rps folder there are three sub-folders named after the three classes (rock, paper, and scissors)

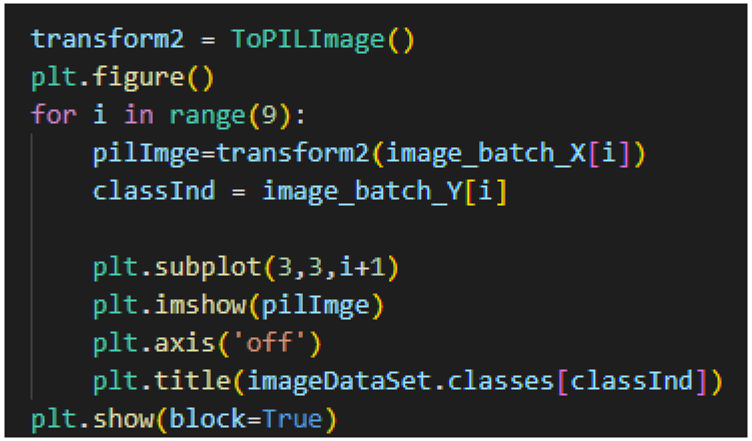
ImageFolder & Conversion Transforms



```python
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image
from torchvision.datasets import ImageFolder
from torchvision.transforms import Compose, Normalize, Resize, ToTensor, ToPILImage, RandomRotation
from torch.utils.data import DataLoader, Dataset,random_split
```

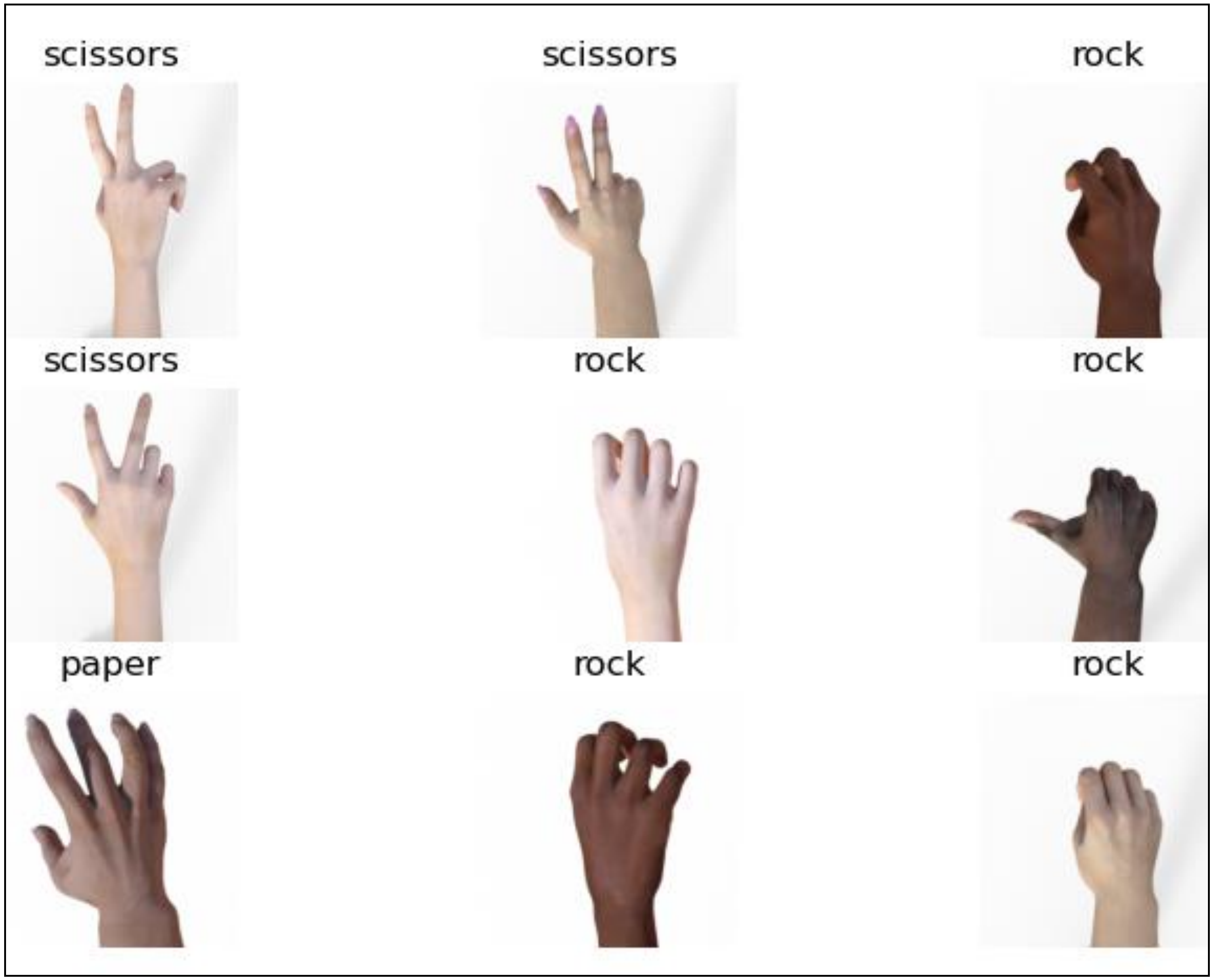


```python
#Datasets & Loaders
transform1 = ToTensor()
imageDataSet=ImageFolder(root='dataset/rps/',transform=transform1)
imageLoader=DataLoader(imageDataSet, batch_size=9,shuffle=True)
image_batch_X, image_batch_Y=next(iter(imageLoader))
```



```python
transform2 = ToPILImage()
plt.figure()
for i in range(9):
    pilImge=transform2(image_batch_X[i])
    classInd = image_batch_Y[i]

    plt.subplot(3,3,i+1)
    plt.imshow(pilImge)
    plt.axis('off')
    plt.title(imageDataSet.classes[classInd])
plt.show(block=True)
```

Output



Standardization Transform

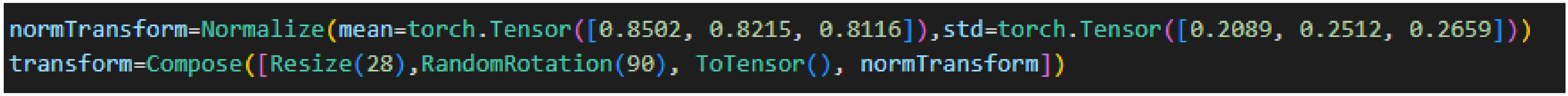
- Similar to sklearn **standardScalar**
- Each Image Dataset has it's own mean and std for each image channel (for training set only)
- **Normalize**(mean=torch.Tensor([0.8502, 0.8215, 0.8116]),std=torch.Tensor([0.2089, 0.2512, 0.2659]))
- Remember always use the training set to compute statistics for standardization

Resize transforms

- Resize the input image to the given size
- torchvision.transforms.Resize(size)

Compose transformations

- from torchvision.transforms import **Compose**
- No need to apply transformations one by one
- Compose can combine several transformations into a single, big, composed, transformation.



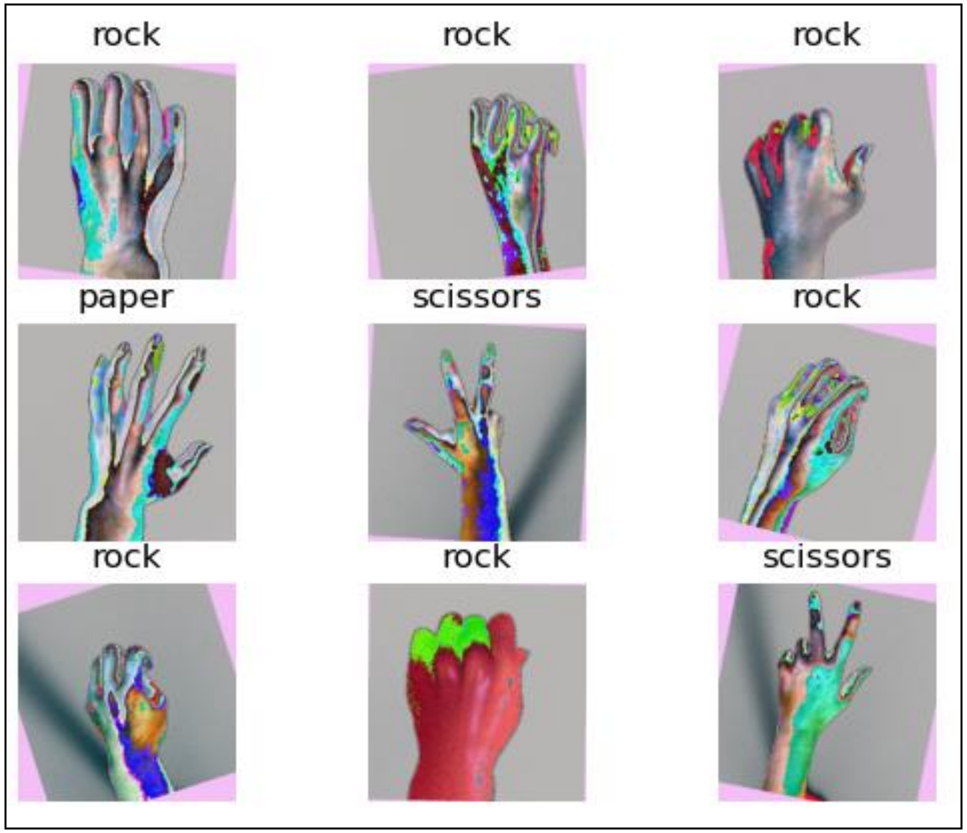
```
normTransform=Normalize(mean=torch.Tensor([0.8502, 0.8215, 0.8116]),std=torch.Tensor([0.2089, 0.2512, 0.2659]))
transform=Compose([Resize(28),RandomRotation(90), ToTensor(), normTransform])
```

Data augmentation

- These transformations modify the training images in many different ways:
  - rotating, shifting, flipping, cropping, blurring, zooming in, adding noise to it, erasing parts of it…
- Why we need it?
  - clever technique to expand a dataset (augment it) without collecting more data
  - deep learning models are very data-hungry, requiring a massive amount of examples to perform well
- Not suited for every task
  - In **object detection**, you shouldn't do anything that changes its position, like flipping or shifting.

Example

```
normTransform=Normalize(mean=torch.Tensor([0.8502, 0.8215, 0.8116]),std=torch.Tensor([0.2089, 0.2512, 0.2659]))
transform=Compose([Resize([500, 500]), RandomHorizontalFlip(p=.5), RandomAutocontrast(0.5),RandomRotation(20),ToTensor(),normTransform])

imageDataSet=ImageFolder(root='dataset/rps/',transform=transform)
imageLoader=DataLoader(imageDataSet, batch_size=9,shuffle=True)
image_batch_X, image_batch_Y=next(iter(imageLoader))
```

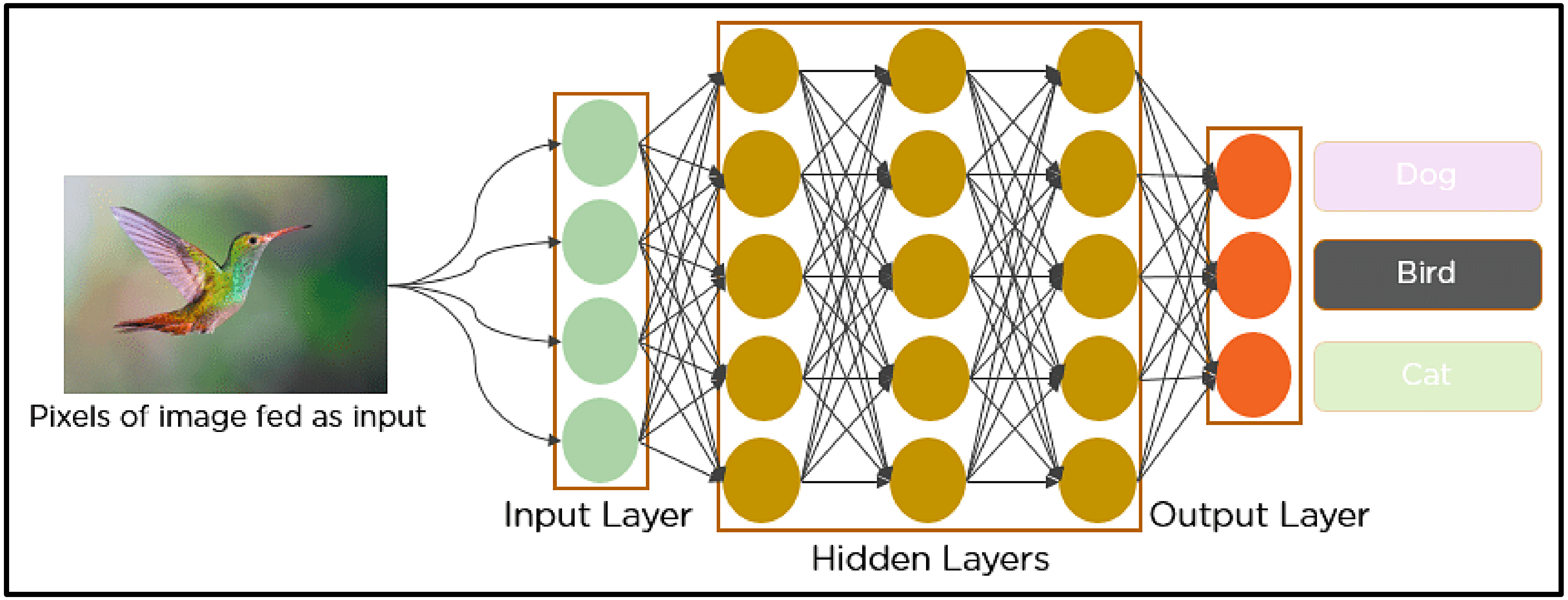


Graded Home Task-4

- Build your own dataset class for RPS dataset
  - Don’t use ImageFolder
  - Use Pillow to load images from folders of each class
  - Make a list of images & labels manually
- Add a compose transforms to your dataset class
  - Resize, RamdomRotation, ToTensor, Normalize
- Create train & test loaders
  - Only do Resize & ToTensor transform for validation loader
  - You can create a flag in your dataset class
- Take a batch of images from your loader and display it

PROBLEMS OF ANN FOR IMAGES

Pixel Flattening

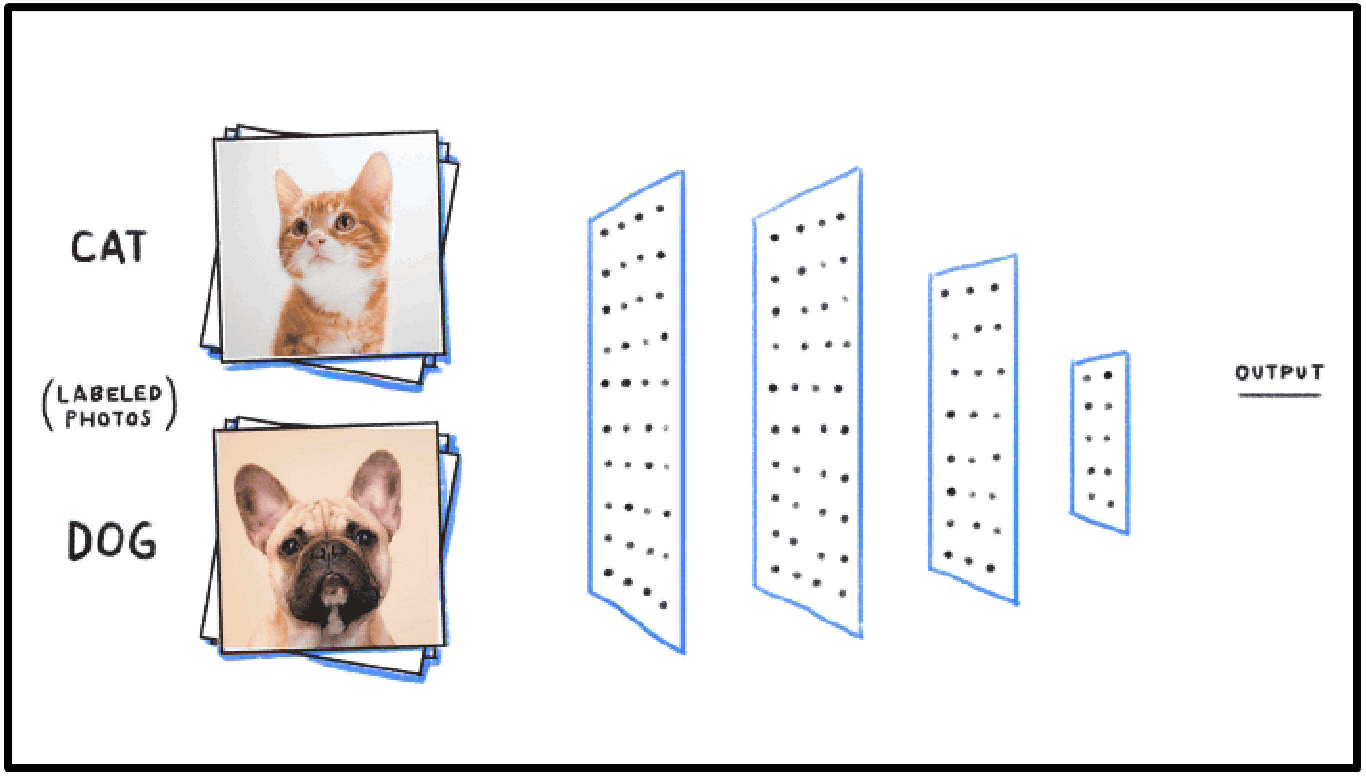


MLP/ANN/FF Problem-1

- MLPs (Multilayer Perceptron) use one perceptron for each input (e.g. pixel in an image)
- the amount of weights rapidly becomes unmanageable for large images.
- It includes **too many parameters** because it is fully connected.
  - Each node is connected to every other node in next and the previous layer, forming a very dense web
  - Resulting in redundancy and inefficiency.
- As a result, difficulties arise whilst training
  - **Overfitting** can occur which makes it lose the ability to generalize

MLP/ANN/FF Problem-2

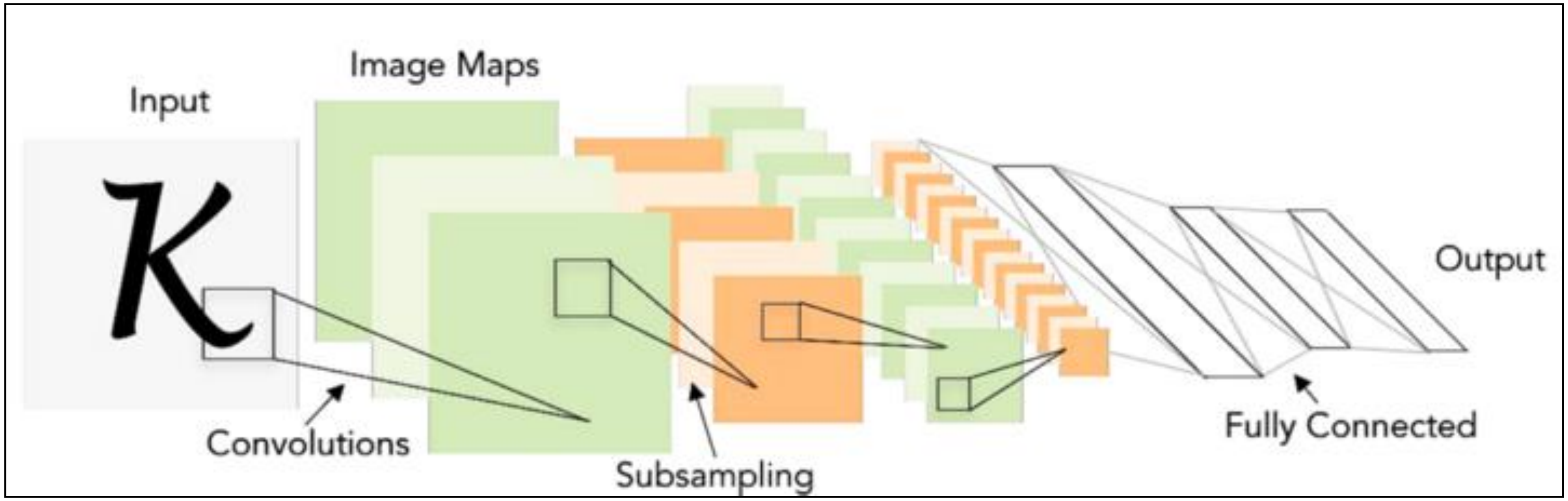
- MLPs will react differently to an input (images) and its shifted version
- They are not translation invariant.
- Example:
  - If a picture of a cat appears in the top left of the image in one picture and the bottom right of another picture
  - MLP will treat them differently
- We considered each pixel as an individual, independent, feature, thus **losing spatial information while flattening the image.**



CNN (THEORY)

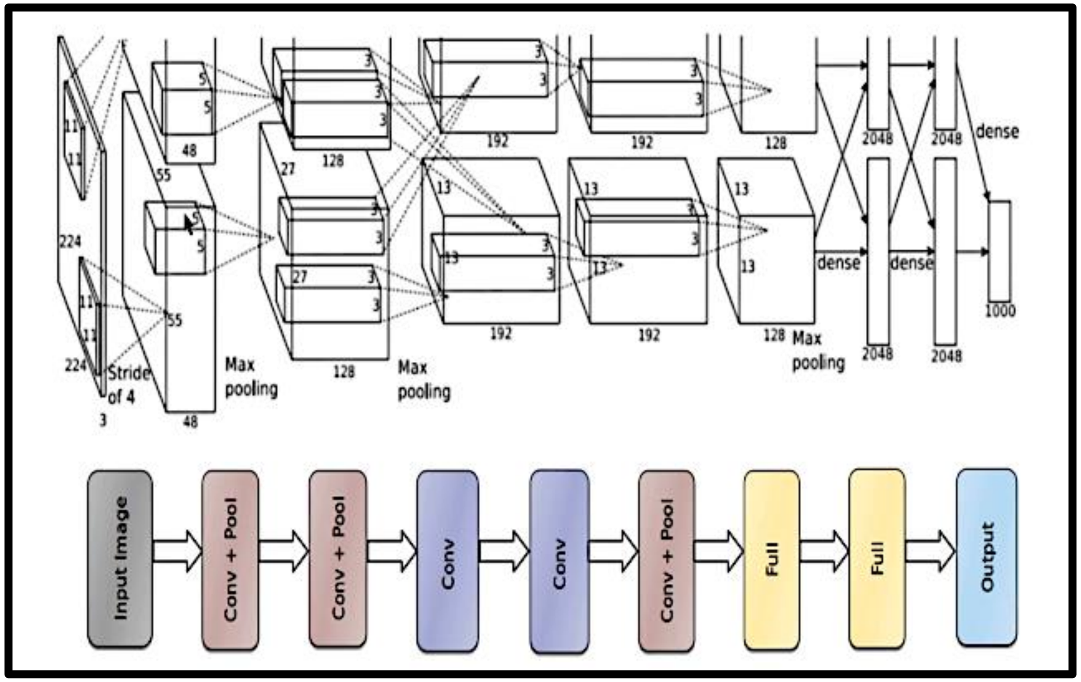
History (LeNet-1988)

- Yann LeCun, director of Facebook's AI Research Group, is the pioneer of convolutional neural networks.
  - He built the first convolutional neural network called **LeNet in 1988.**
  - LeNet was used for character recognition tasks like reading zip codes and digits.

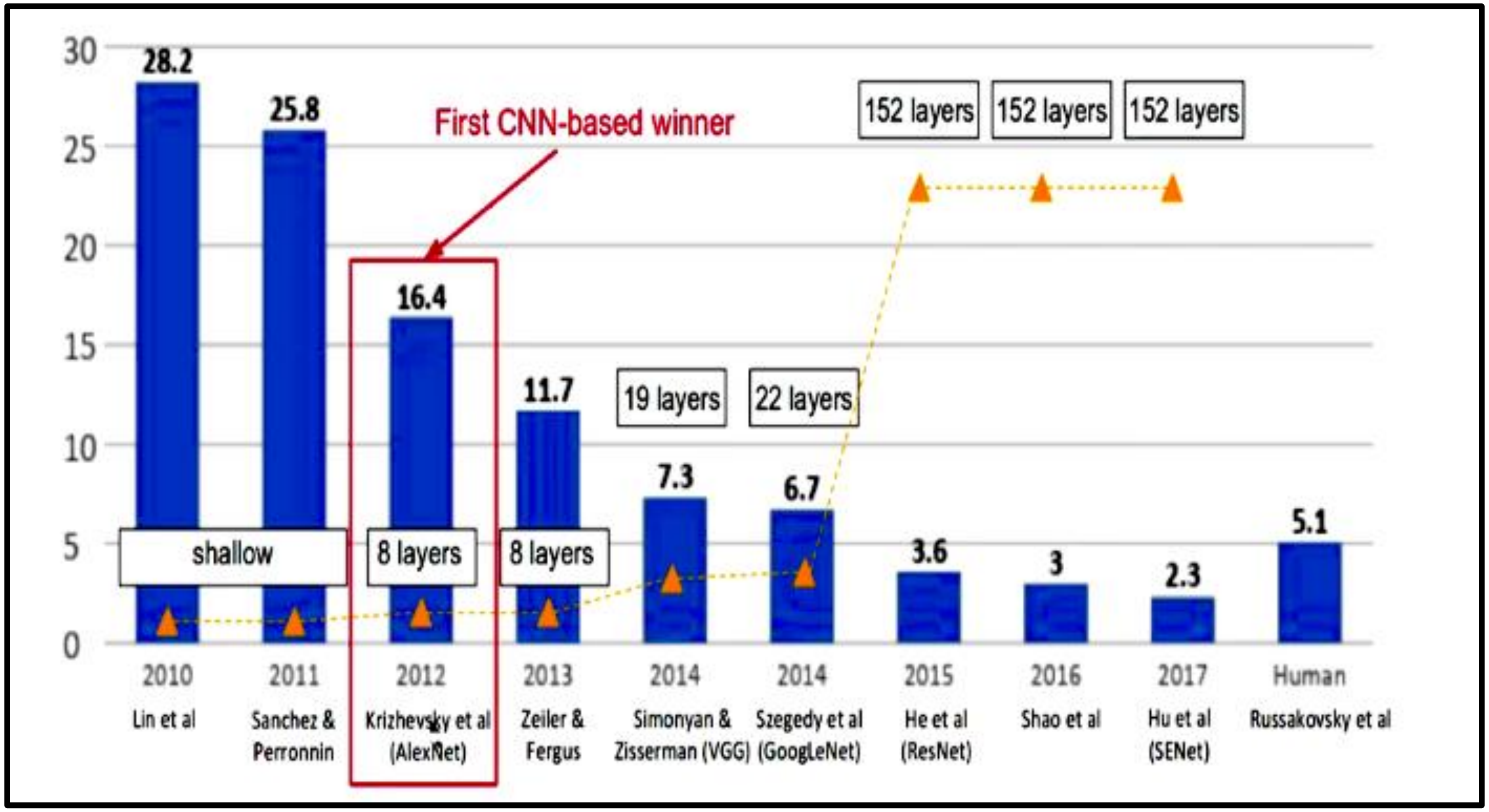


History (AlexNet-2012)

- AlexNet was the **first winner of the ImageNet challenge** and was based on a CNN,
- Since 2012, every year's challenge has been won by a CNN
- Reduced the error from 25.8% to 16.4% which was a significant improvement at that time



History (2012~)

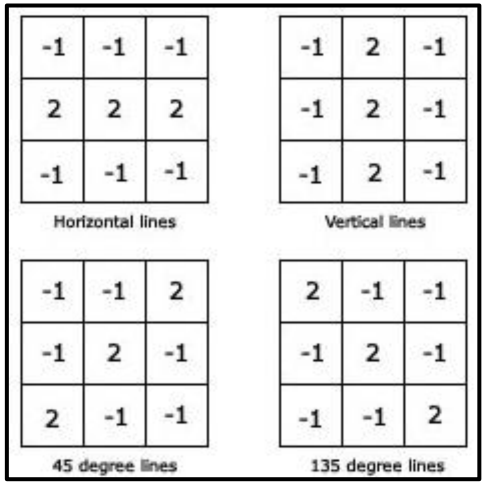


Convolutional Neural Network?

- A Convolutional neural network (CNN) is a neural network that has one or more convolutional layers and are used mainly for **image processing**
  - Convolution operation on the two functions gives output in a form of a third function that shows how the shape of one function is being influenced, modified by the other function.

Convolutional Neural Network (ConvNets)?

- Convolution matrix is also called a kernel or filter
- Image processing operations through convolution between a kernel and an image:
  - Blurring
  - Sharpening
  - Edge detection, and more are



-1	-1	-1
2	2	2
-1	-1	-1

Horizontal lines

-1	2	-1
-1	2	-1
-1	2	-1

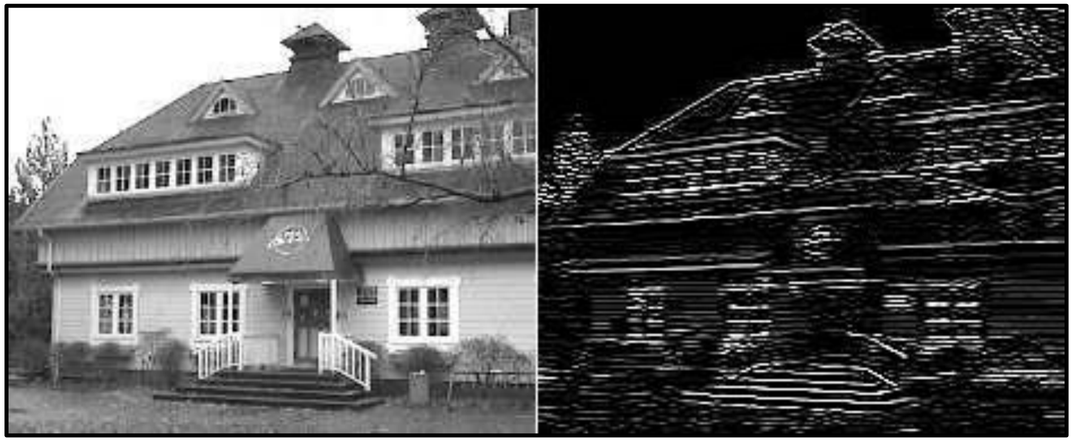
Vertical lines

-1	-1	2
-1	2	-1
2	-1	-1

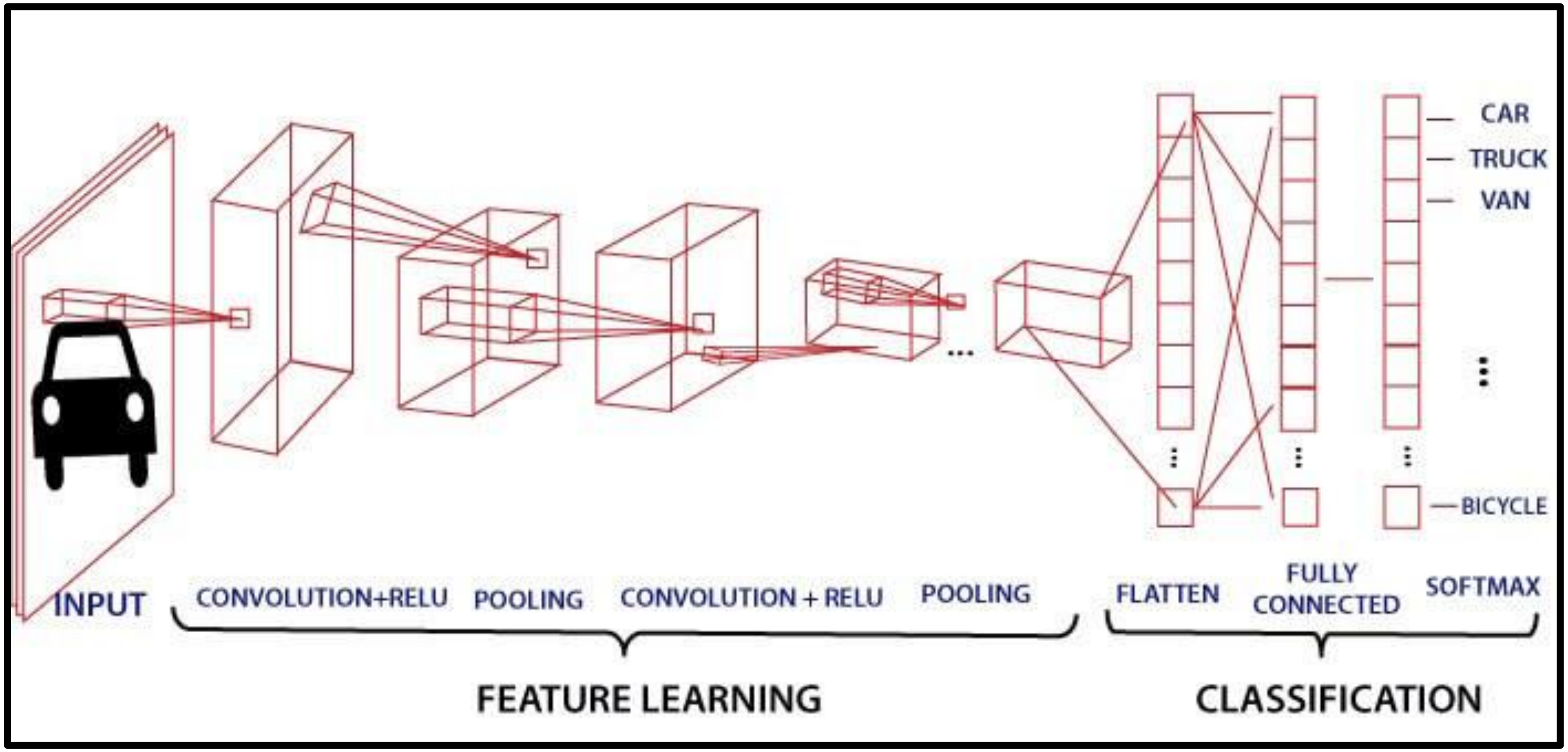
45 degree lines

2	-1	-1
-1	2	-1
-1	-1	2

135 degree lines

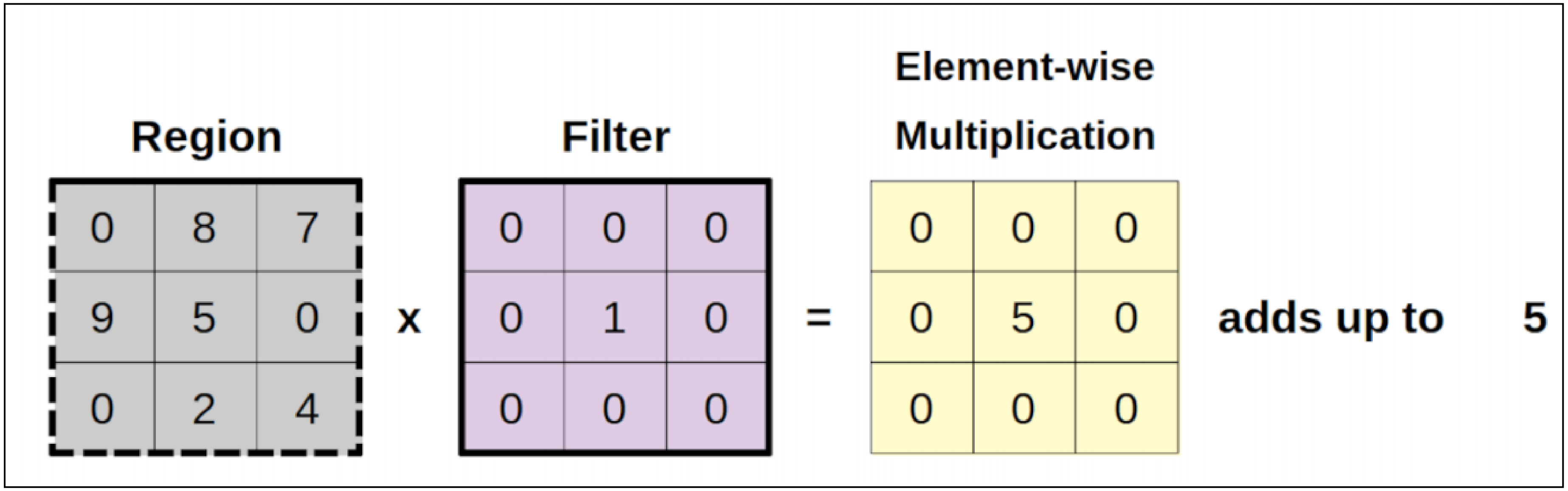


CNNs / ConvNets Architecture



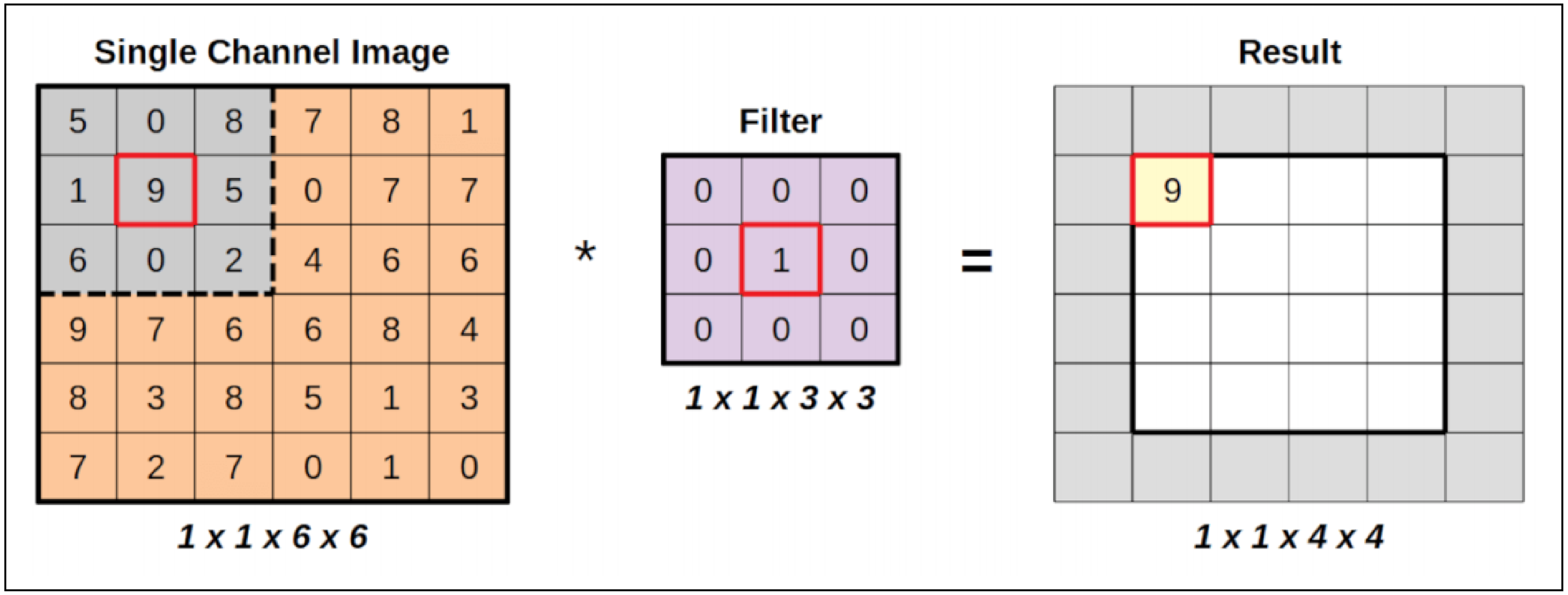
Convolution Layer

- Convolution refers to the **filtering** process



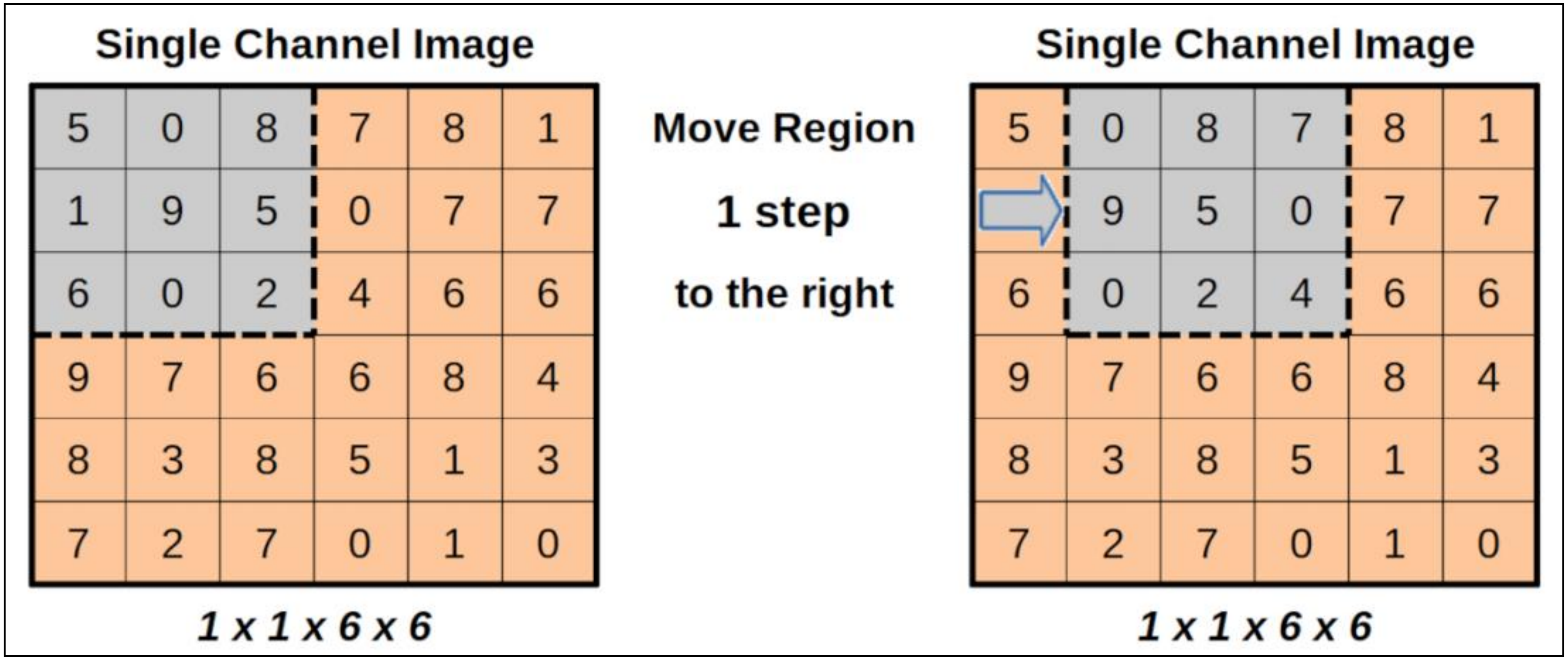
Convolution Layer

- Convolution refers to the filtering process
- Doing a convolution produces an image with a **reduced size**.



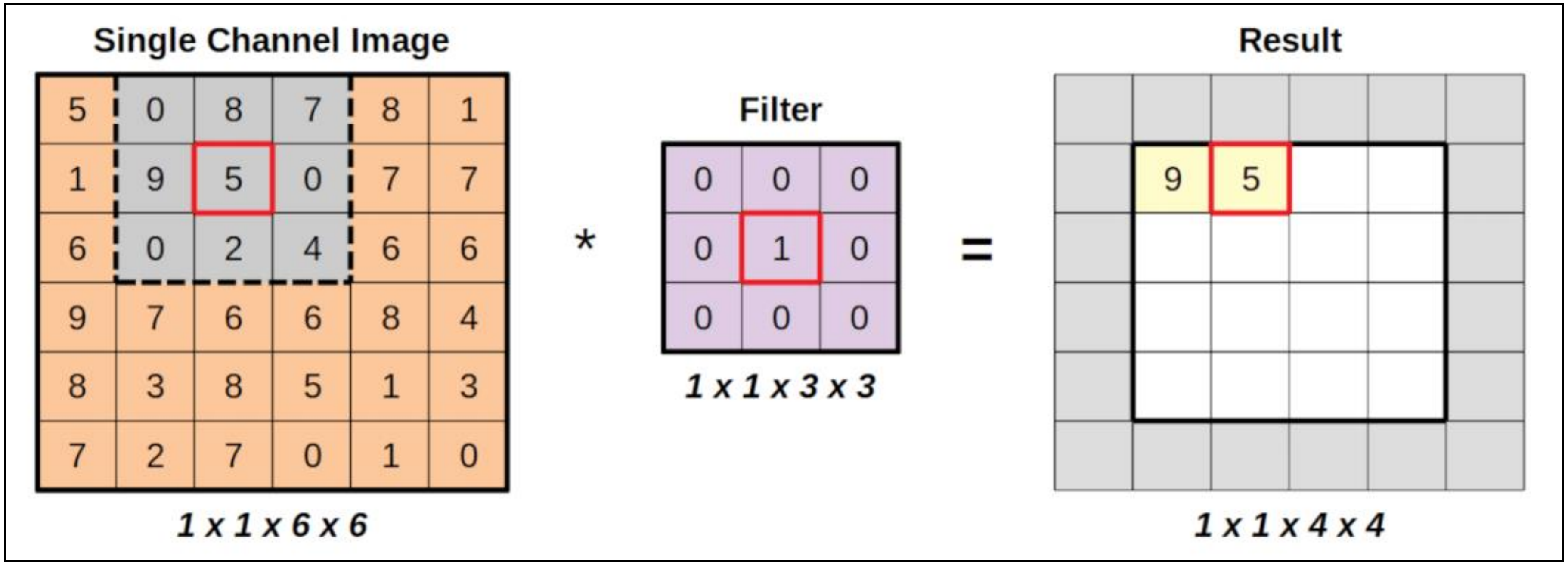
Convolution Layer

- Move the region one step to the right, that is, we change the receptive field, and apply the filter again



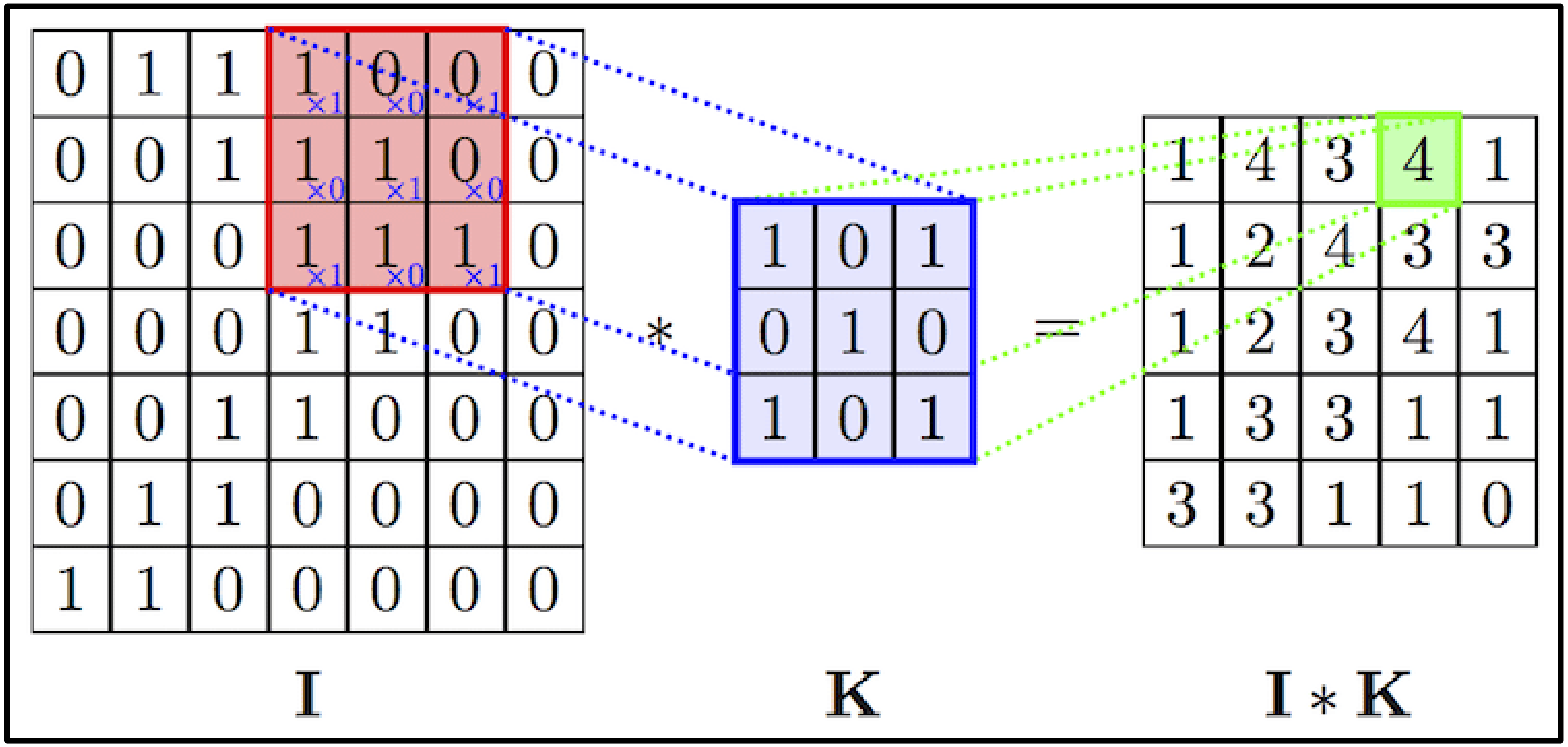
Convolution Layer

- Move the region one step to the right, that is, we change the receptive field, and apply the filter again



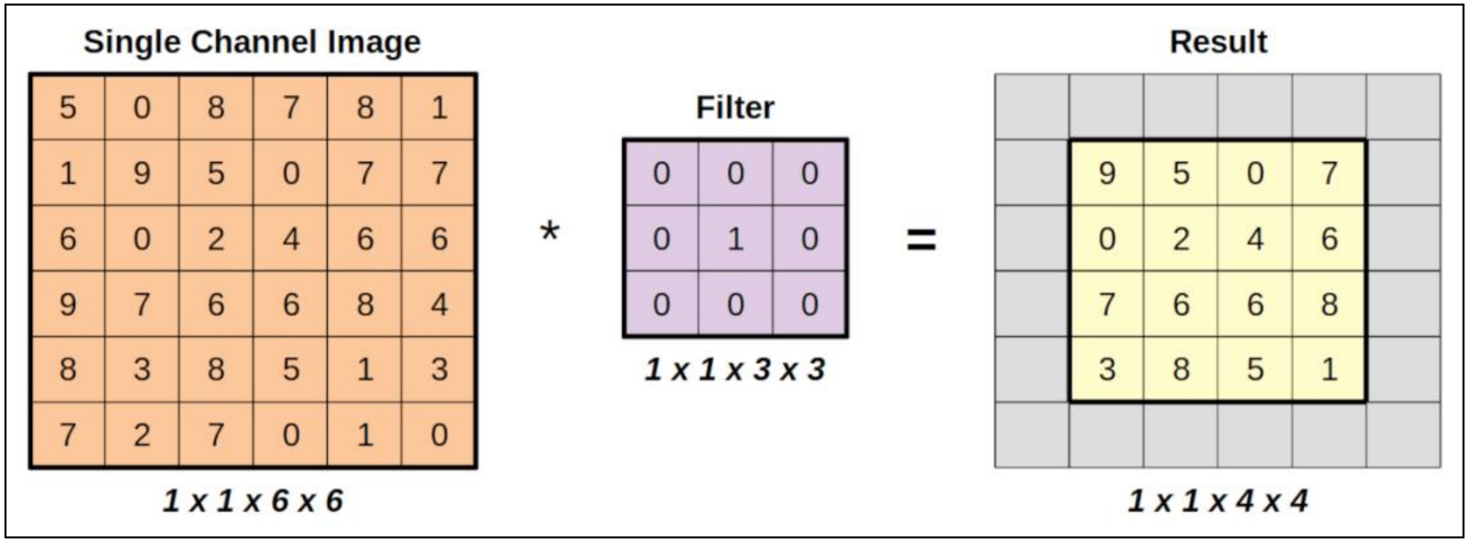
Convolution Layer

- Convolution refers to the filtering process



Convolution Layer

- Fully Convolved
- How much smaller is it going to be?
  - The bigger the filter, the smaller the resulting image
  - **reduction** is equal to the filter size minus one

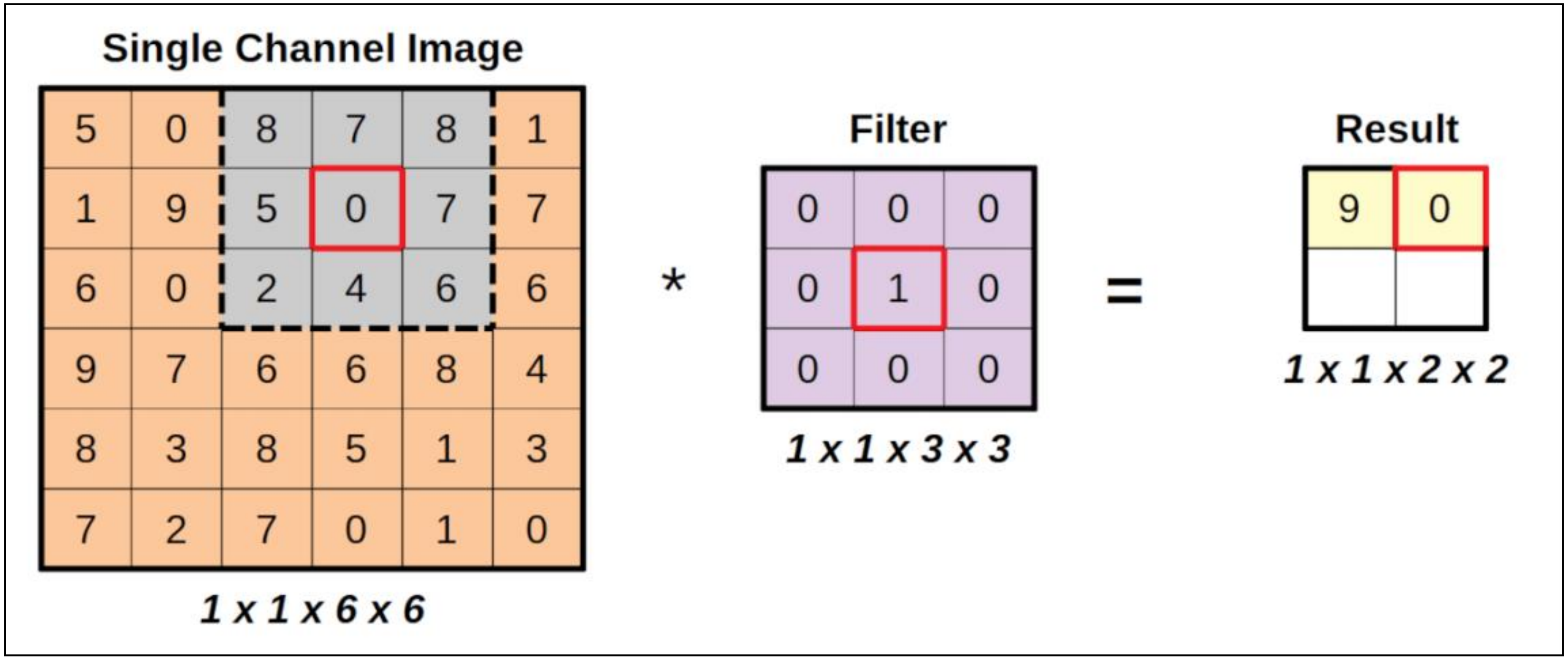


$$(h_i, w_i) * (h_f, w_f) = (h_i - (h_f - 1), w_i - (w_f - 1))$$

$$(h_i, w_i) * f = (h_i - f + 1, w_i - f + 1)$$

Striding

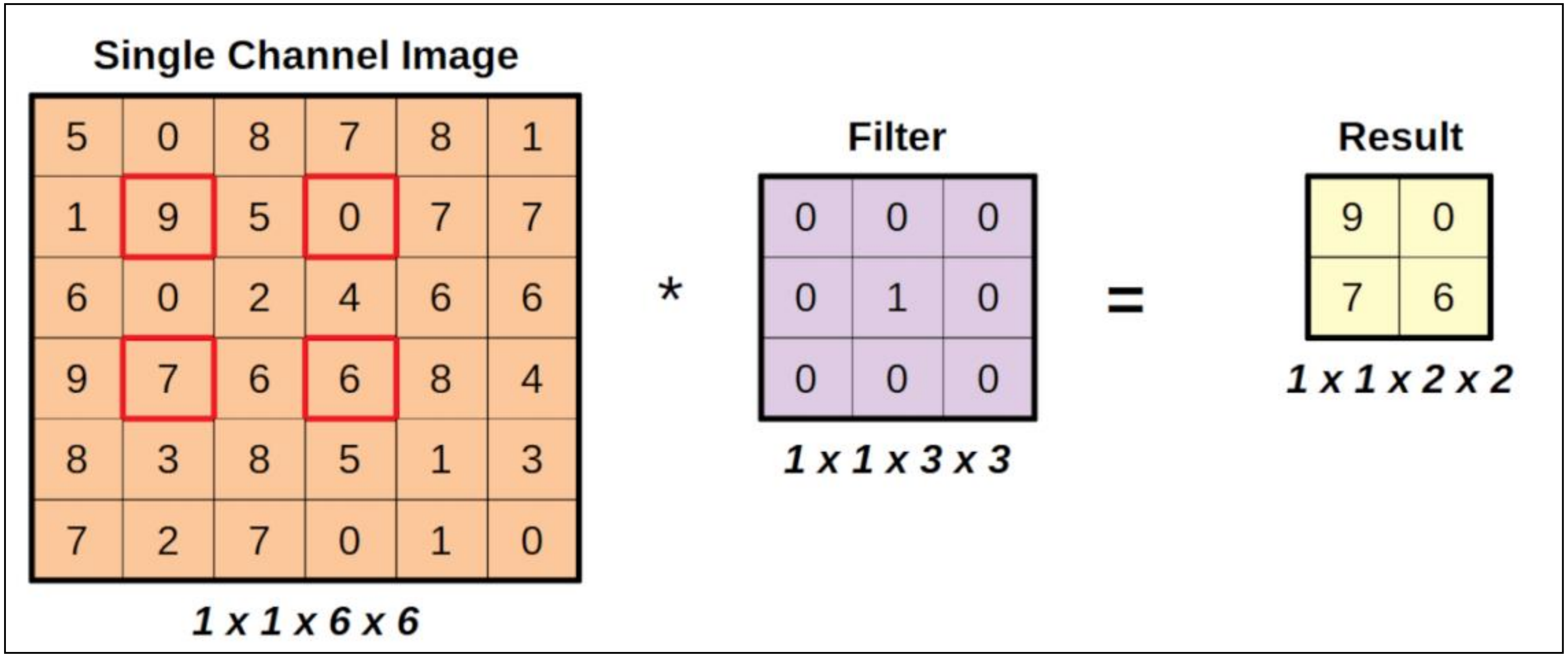
- So far, we've been moving the region of interest one pixel at a time:
  - a stride of one



A stride of 2

Striding

- Stride of 2
  - only **four** valid operations
- The identity kernel may be boring
  - but it highlight the inner workings of the convolutions.
  - It is clear in the figure above where the pixel values in the resulting image come from



A stride of 2

Striding

- The bigger the stride, the smaller the resulting image
- So far, the operations we performed have been shrinking the images. What about **restoring them to their original size**?

$$(h_i, w_i) * f = \left(\frac{h_i - f + 1}{s}, \frac{w_i - f + 1}{s}\right)$$

Equation 5.3 - Shape after a convolution with stride

Padding

- Padding means stuffing.
- We need to stuff the original image so it can sustain the "attack" on its size
  - We may simply **add zeros around it**
  - This simple trick can be used to preserve the original size of the image

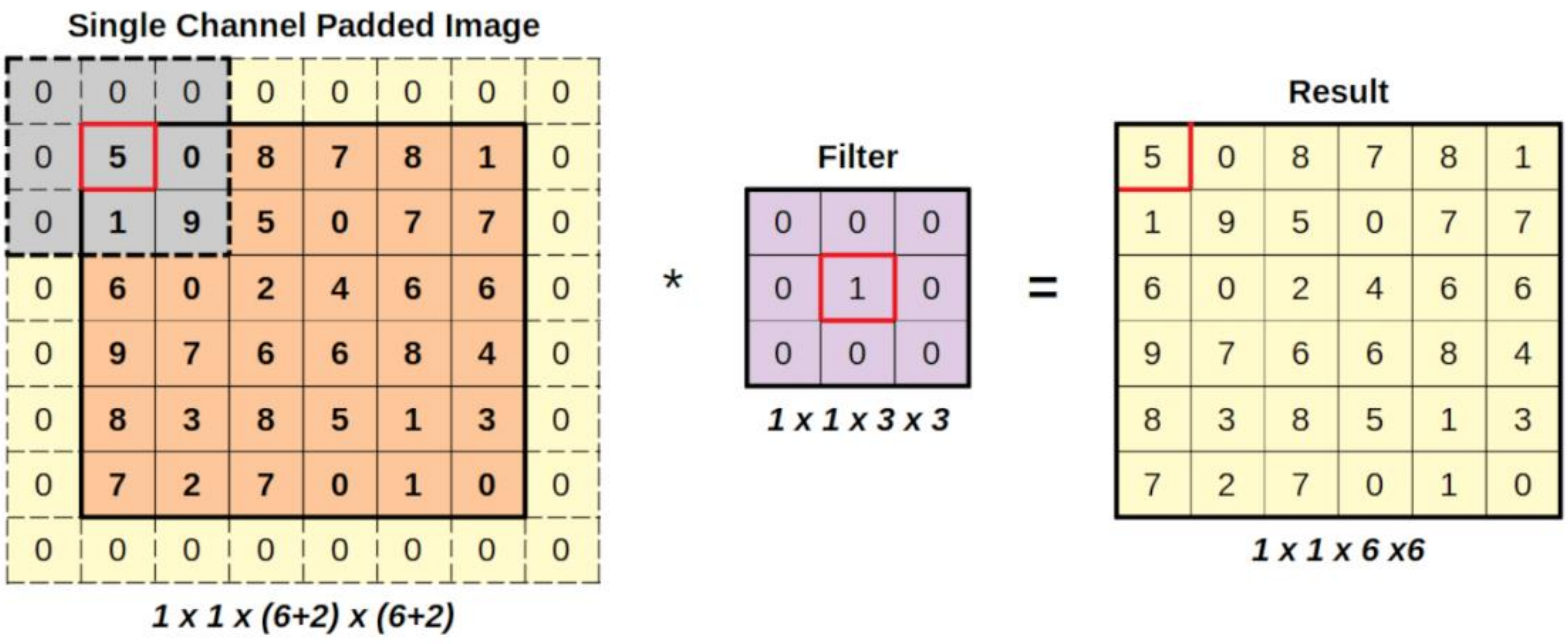
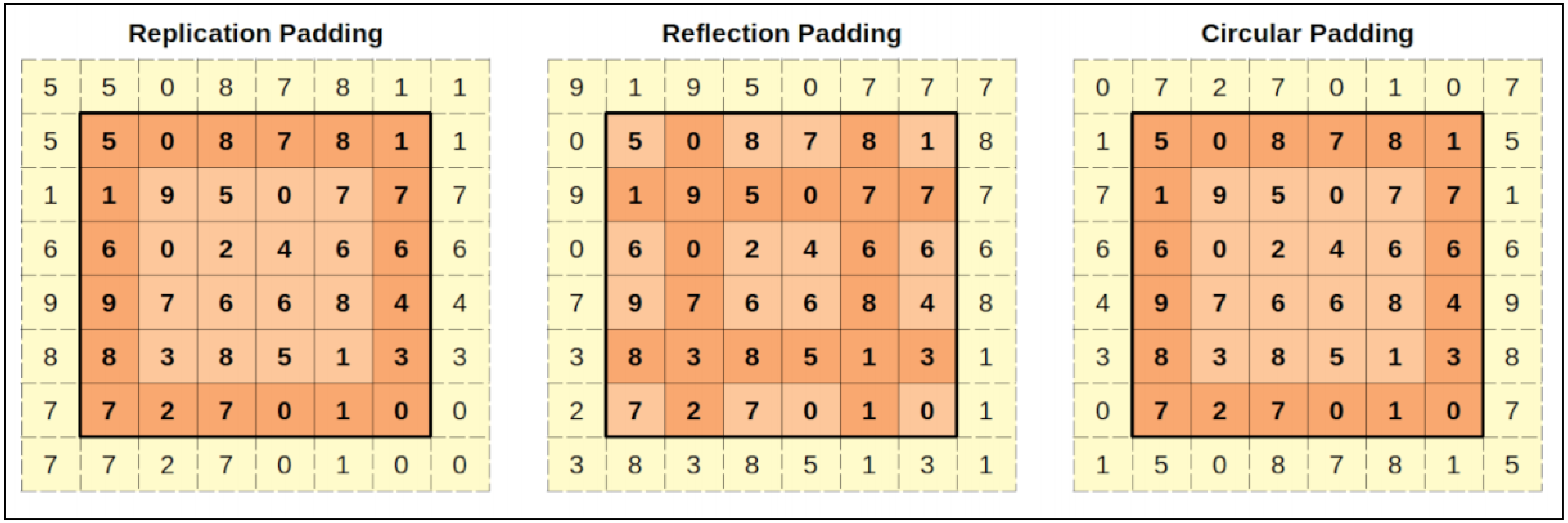


Figure 5.11 - Zero-padded image

Padding Modes

- In the **replication** padding, the padded pixels will have the same value as the closest real pixel



Replication Padding

5	5	0	8	7	8	1	1
5	5	0	8	7	8	1	1
1	1	9	5	0	7	7	7
6	6	0	2	4	6	6	6
9	9	7	6	6	8	4	4
8	8	3	8	5	1	3	3
7	7	2	7	0	1	0	0
7	7	2	7	0	1	0	0

Reflection Padding

9	1	9	5	0	7	7	7
0	5	0	8	7	8	1	8
9	1	9	5	0	7	7	7
0	6	0	2	4	6	6	6
7	9	7	6	6	8	4	8
3	8	3	8	5	1	3	1
2	7	2	7	0	1	0	1
3	8	3	8	5	1	3	1

Circular Padding

0	7	2	7	0	1	0	7
1	5	0	8	7	8	1	5
7	1	9	5	0	7	7	1
6	6	0	2	4	6	6	6
4	9	7	6	6	8	4	9
3	8	3	8	5	1	3	8
0	7	2	7	0	1	0	7
1	5	0	8	7	8	1	5

Convolution Size after Padding

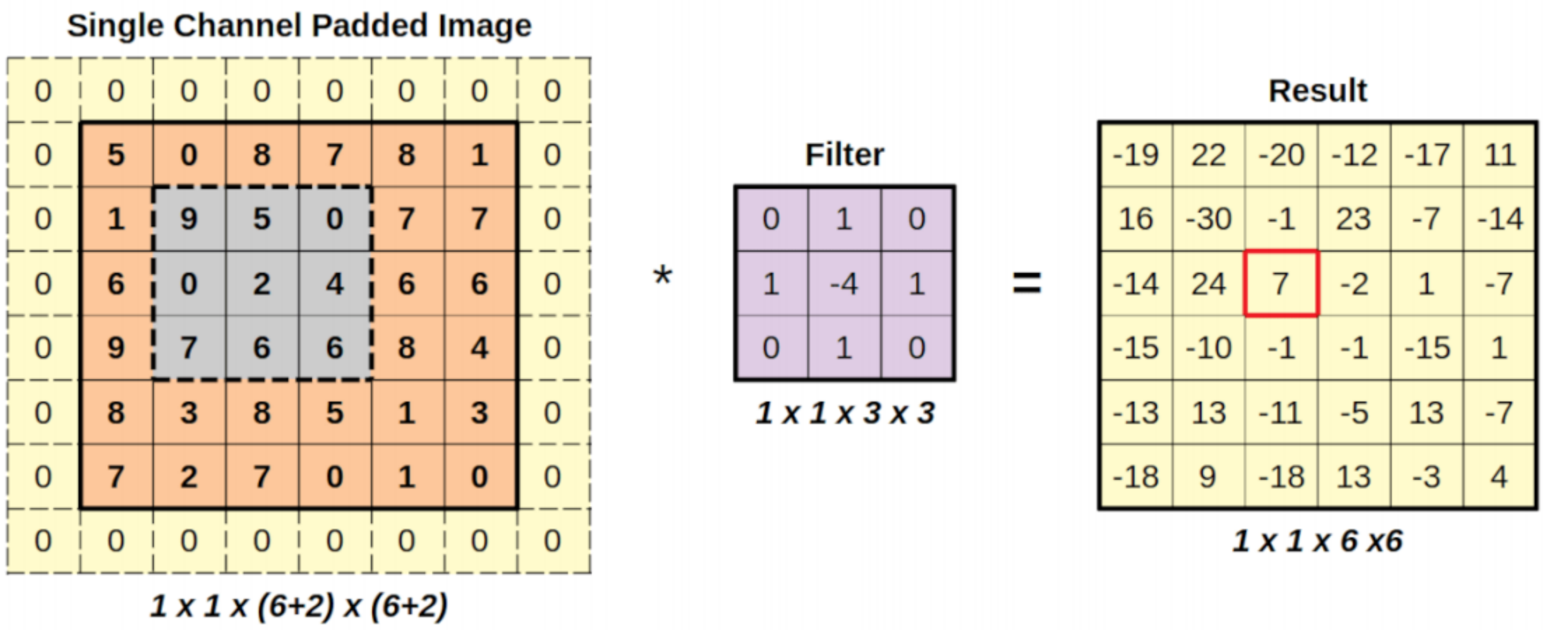
- In the **replication** padding, the padded pixels will have the same value as the closest real pixel

$$(h_i, w_i) * f = \left(\frac{(h_i + 2p) - f}{s} + 1, \frac{(w_i + 2p) - f}{s} + 1 \right)$$

Equation 5.4 - Shape after a convolution with stride and padding

A Real filter

- Edge Detector



Single Channel Padded Image

0	0	0	0	0	0	0	0
0	5	0	8	7	8	1	0
0	1	9	5	0	7	7	0
0	6	0	2	4	6	6	0
0	9	7	6	6	8	4	0
0	8	3	8	5	1	3	0
0	7	2	7	0	1	0	0
0	0	0	0	0	0	0	0

1 x 1 x (6+2) x (6+2)

*

Filter

0	1	0
1	-4	1
0	1	0

1 x 1 x 3 x 3

=

Result

-19	22	-20	-12	-17	11
16	-30	-1	23	-7	-14
-14	24	7	-2	1	-7
-15	-10	-1	-1	-15	1
-13	13	-11	-5	13	-7
-18	9	-18	13	-3	4

1 x 1 x 6 x6

Figure 5.13 - Convolving a padded image - no shrinking!

A Real filter

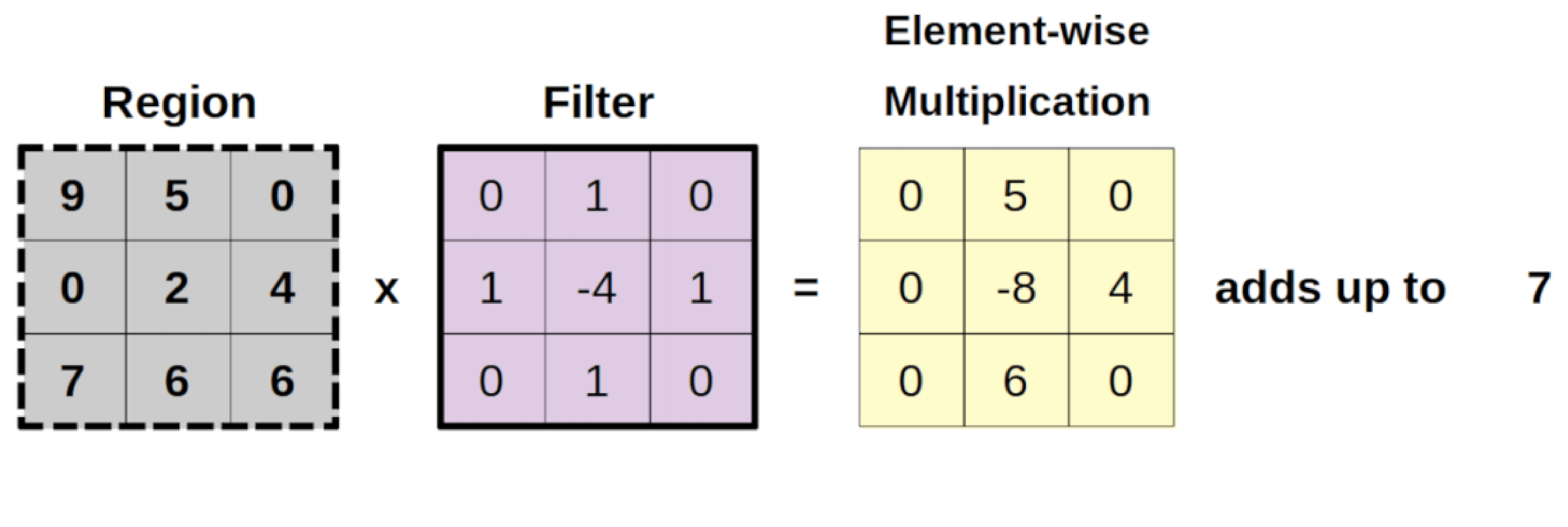


Figure 5.14 - Element-wise multiplication - edge filter

Pooling Layer

- Splits the image into tiny chunks, performs an operation on each chunk
  - Each chunk yields a single value
  - Puts the chunks together as the resulting image and shrink image
- **Max-Pooling** with a kernel size of two (stride is also a parameter of pooling)
- Besides max-pooling, **average pooling** is also fairly common

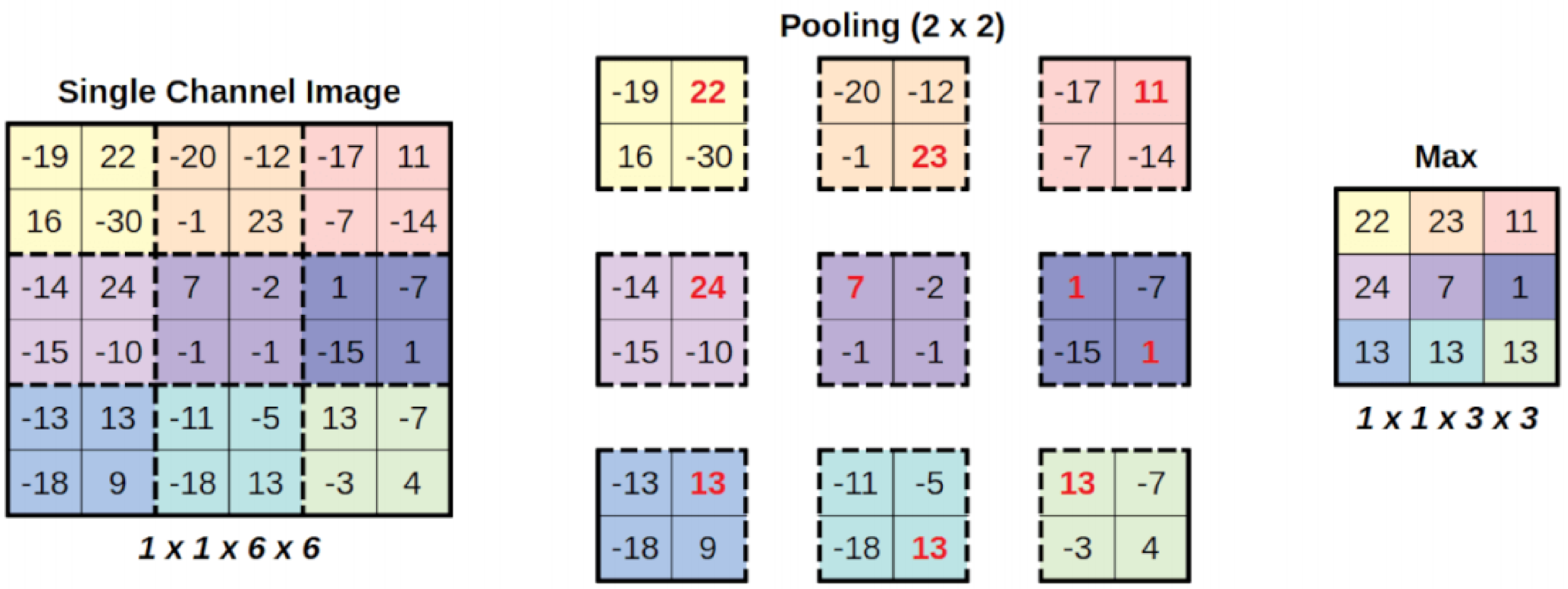
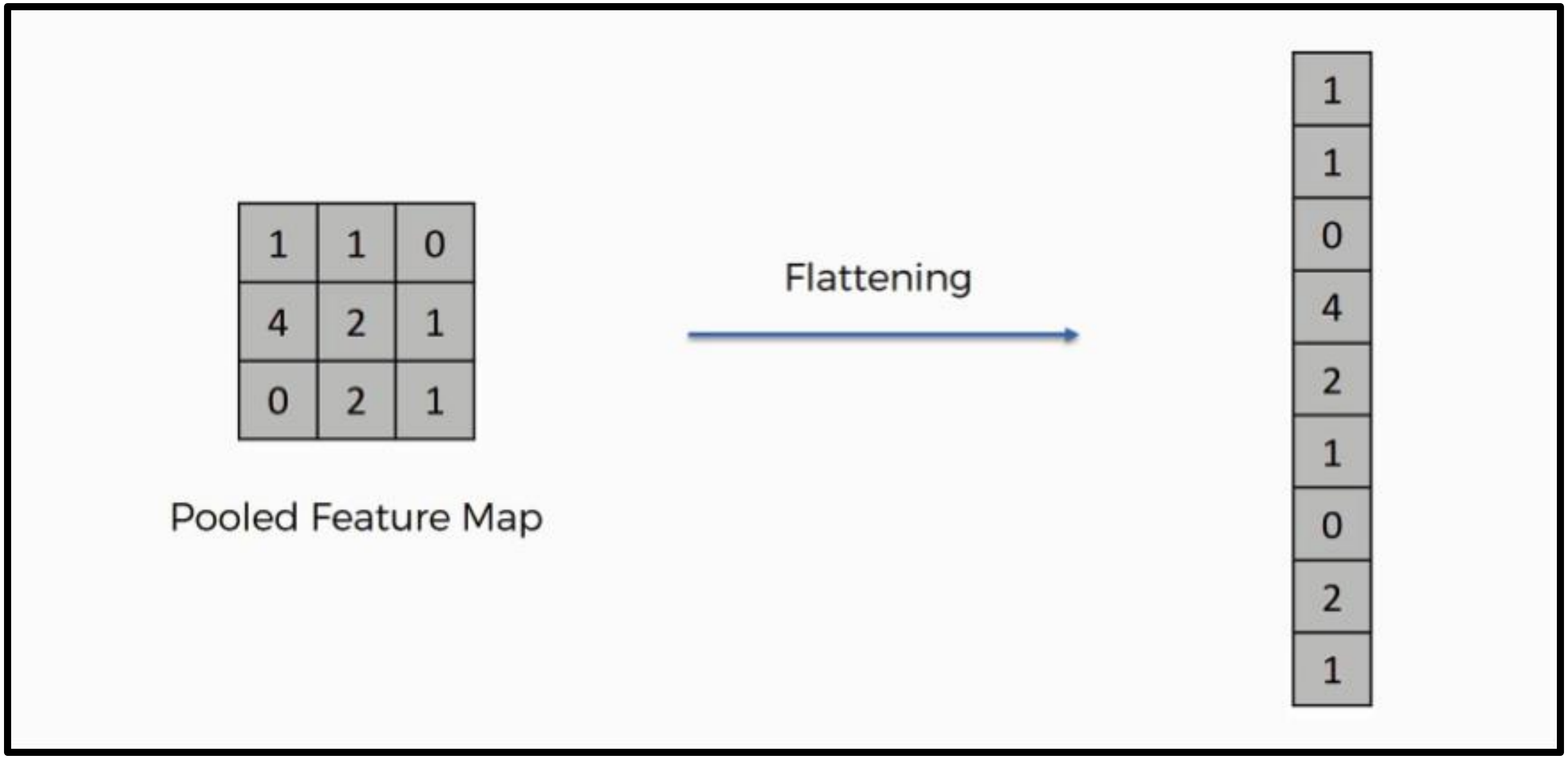


Figure 5.15 - Max pooling

Flattening Layer

- It simply flattens a tensor by **collapsing** dimensions

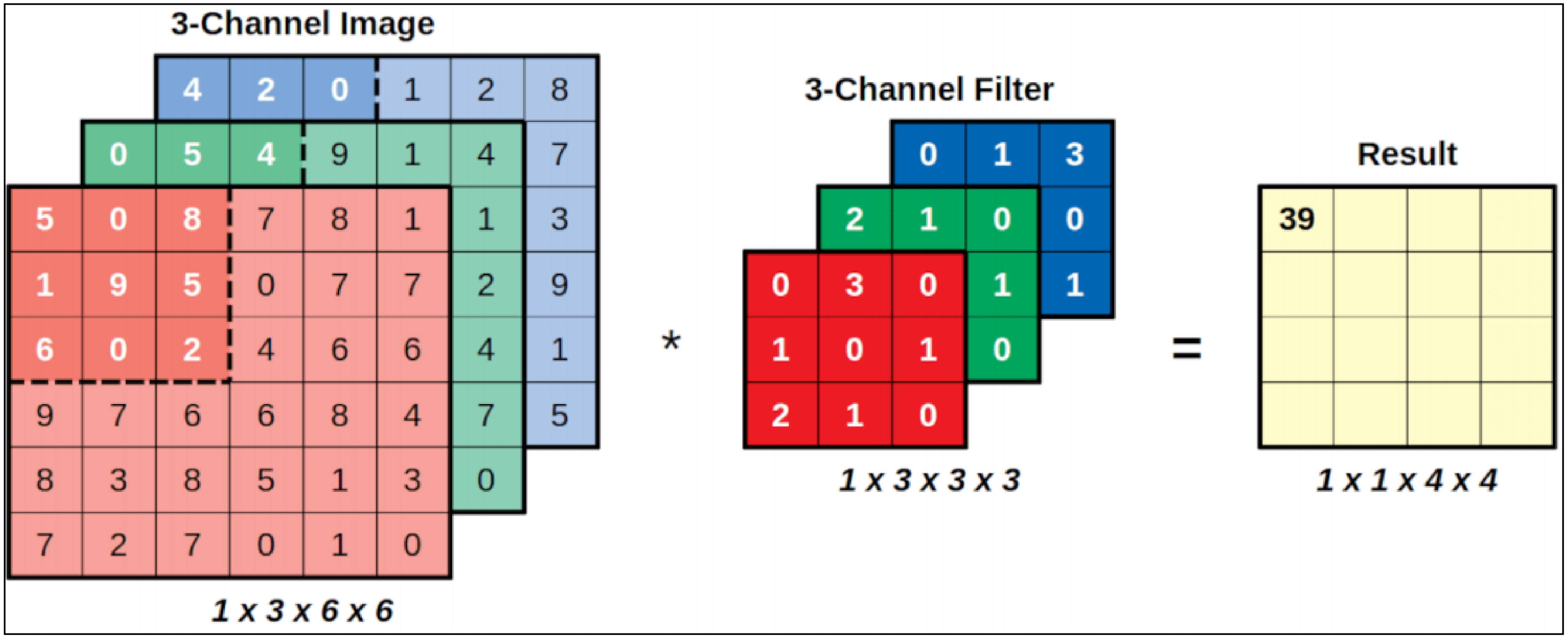


Multi-Channel Convolutions

- So far , there was a single channel image and a single channel filter.
- For three-channel images, a **three-channel filter** is used
- In Generalized 2D convolutions
  - Every filter has as many channels as the image it is convolving
- Convolving a three-channel filter over a three-channel image still produces a single value

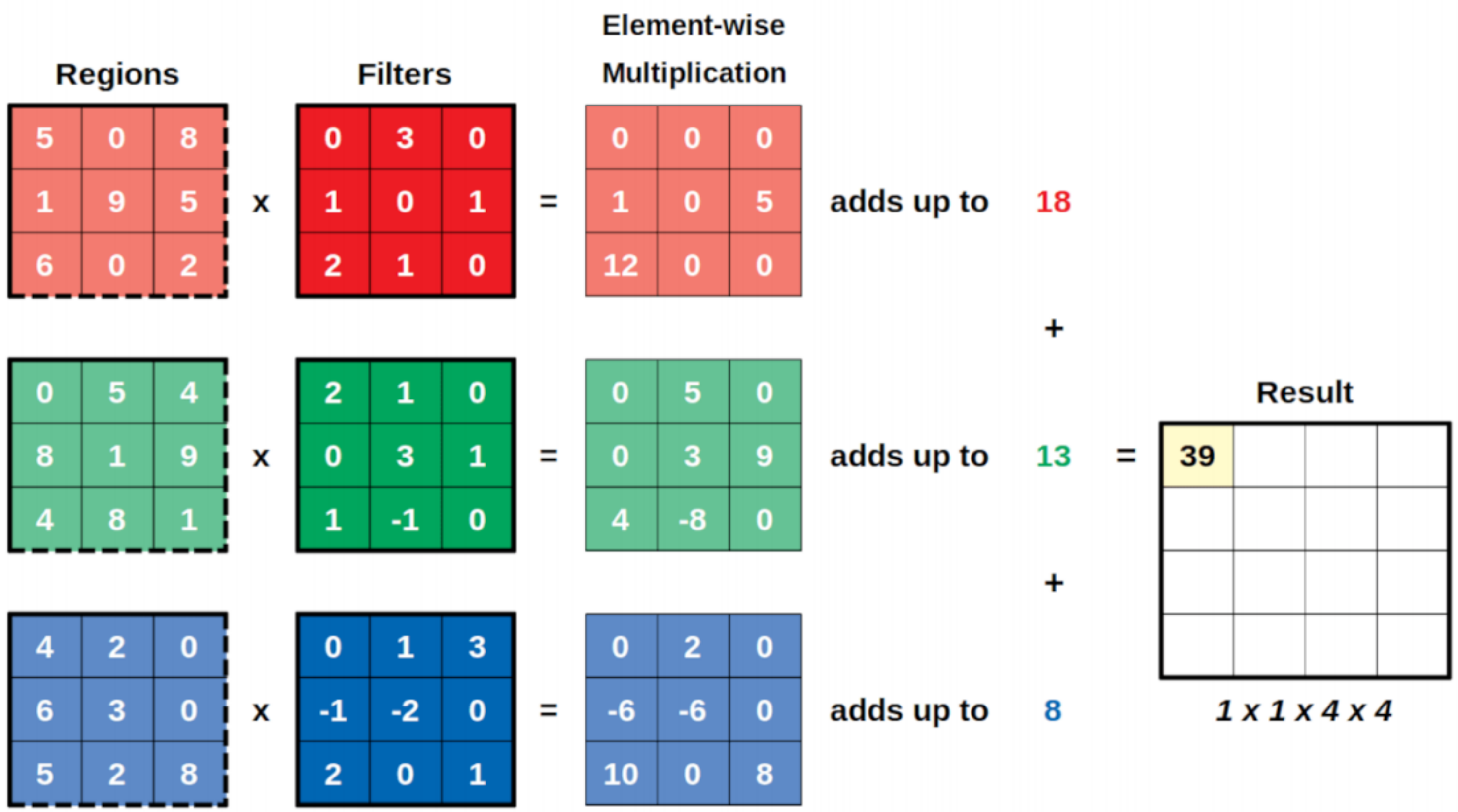
Multi-Channel Convolutions

- Think of it as performing three convolutions, each corresponding to the element-wise multiplication of the matching region/channel and filter/channel,
  - Resulting in three values, one for each channel.
  - **Adding up the results** for each channel produces the expected single value.

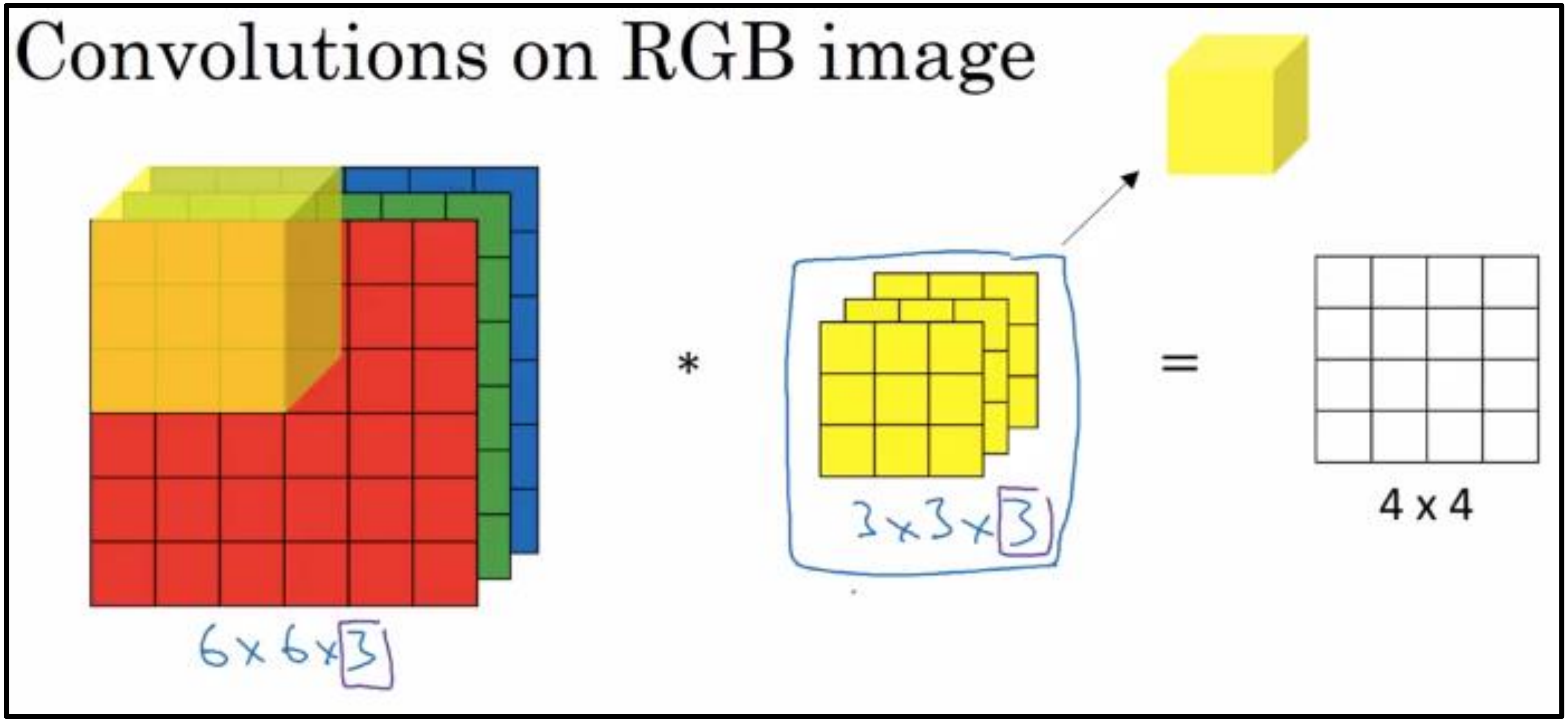


Multi-Channel Convolutions

- Think of it as performing three convolutions, each corresponding to the element-wise multiplication of the matching region/channel and filter/channel,
  - Resulting in three values, one for each channel.
  - Adding up the results for each channel produces the expected single value.

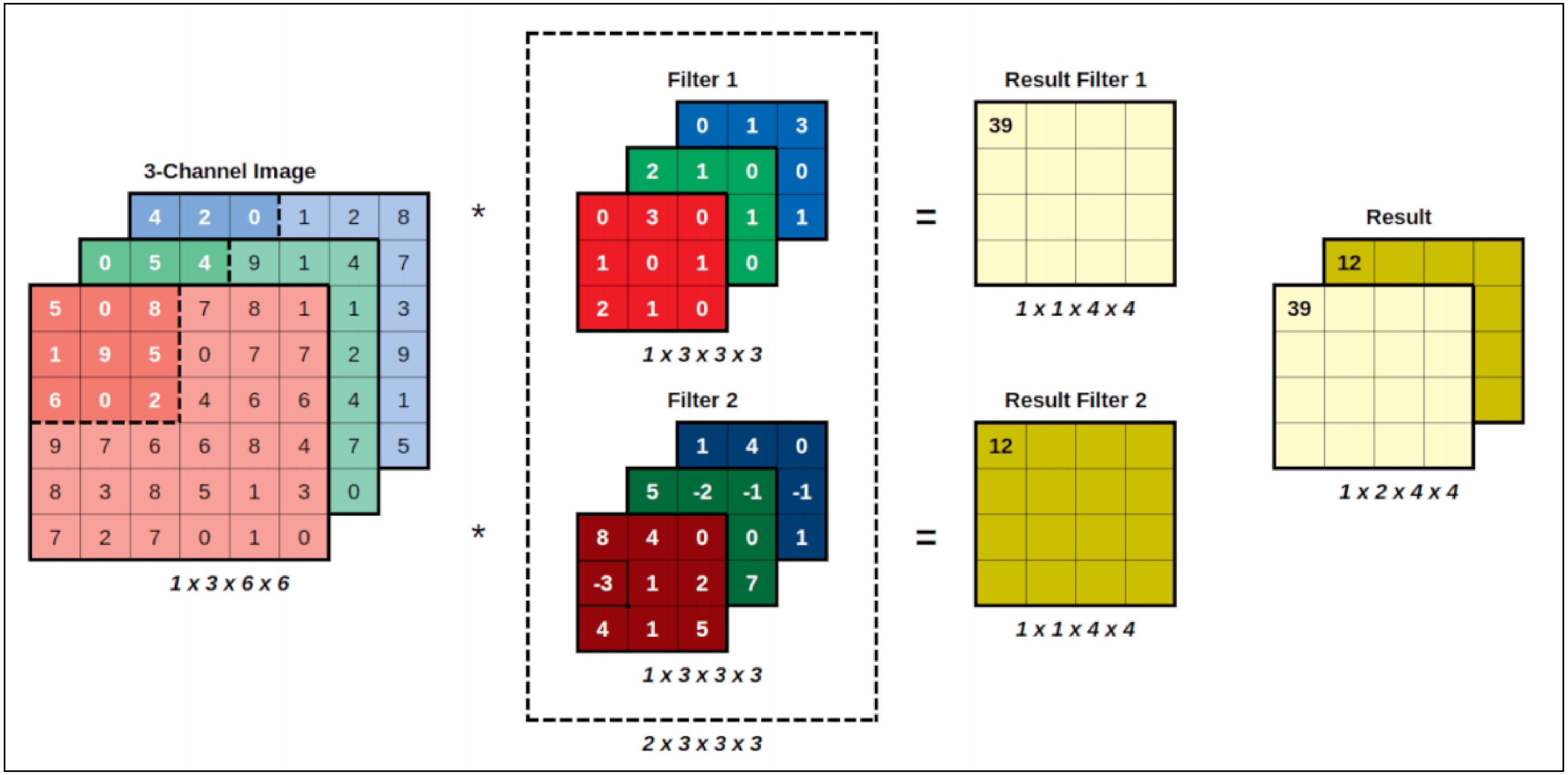

Regions
Filters
Element-wise Multiplication
5 0 8 / 1 9 5 / 6 0 2 x 0 3 0 / 1 0 1 / 2 1 0 = 0 0 0 / 1 0 5 / 12 0 0 adds up to 18
+
0 5 4 / 8 1 9 / 4 8 1 x 2 1 0 / 0 3 1 / 1 -1 0 = 0 5 0 / 0 3 9 / 4 -8 0 adds up to 13
+
4 2 0 / 6 3 0 / 5 2 8 x 0 1 3 / -1 -2 0 / 2 0 1 = 0 2 0 / -6 -6 0 / 10 0 8 adds up to 8
=
Result
39
1 x 1 x 4 x 4

Multi-Channel Convolutions



Multiple Filters

- The convolution produces as many channels as there are filters
- Example:
  - **Two filters over three channels**



PyTorch Conv2D Layer

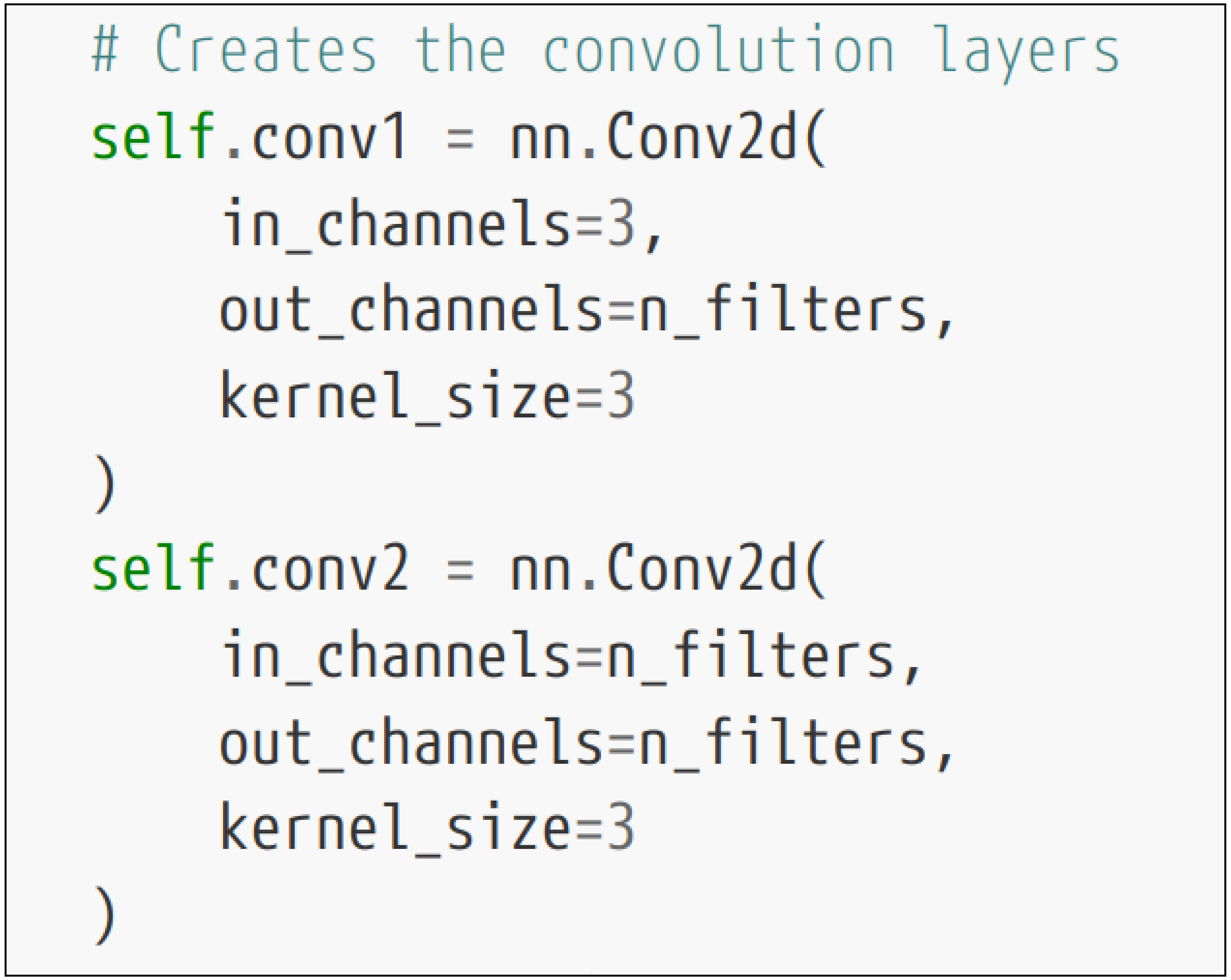
- PyTorch's convolution module
  - nn.Conv2d
- many arguments
  - **in_channels**: number of channels of the input image
  - **out_channels**: number of channels produced by the convolution
  - **kernel_size**: size of the (square) convolution filter/kernel
  - **stride**: the size of the movement of the selected region
- There is no argument for the kernel/filter itself, there is only a kernel_size argument
  - **Learned by the module**
- It is possible to produce multiple channels as output

PyTorch Conv2D Layer

```
conv = nn.Conv2d(
    in_channels=1, out_channels=1, kernel_size=3, stride=1
)
conv(image)
```

```
conv_multiple = nn.Conv2d(
    in_channels=1, out_channels=2, kernel_size=3, stride=1
)
conv_multiple.weight
```

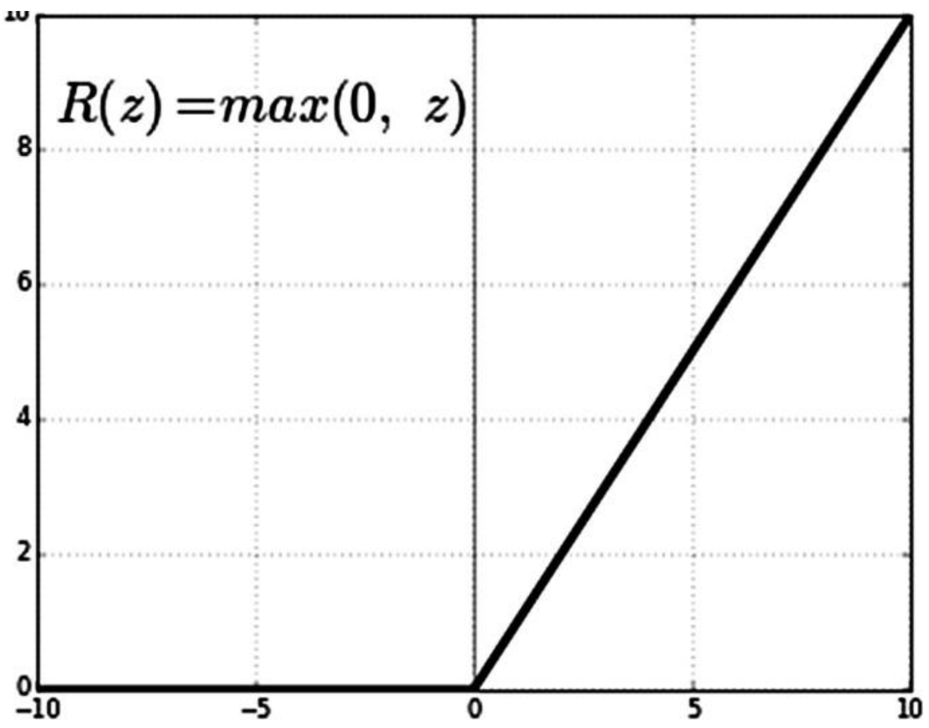
PyTorch Conv2D Layer



```
# Creates the convolution layers
self.conv1 = nn.Conv2d(
    in_channels=3,
    out_channels=n_filters,
    kernel_size=3
)
self.conv2 = nn.Conv2d(
    in_channels=n_filters,
    out_channels=n_filters,
    kernel_size=3
)
```

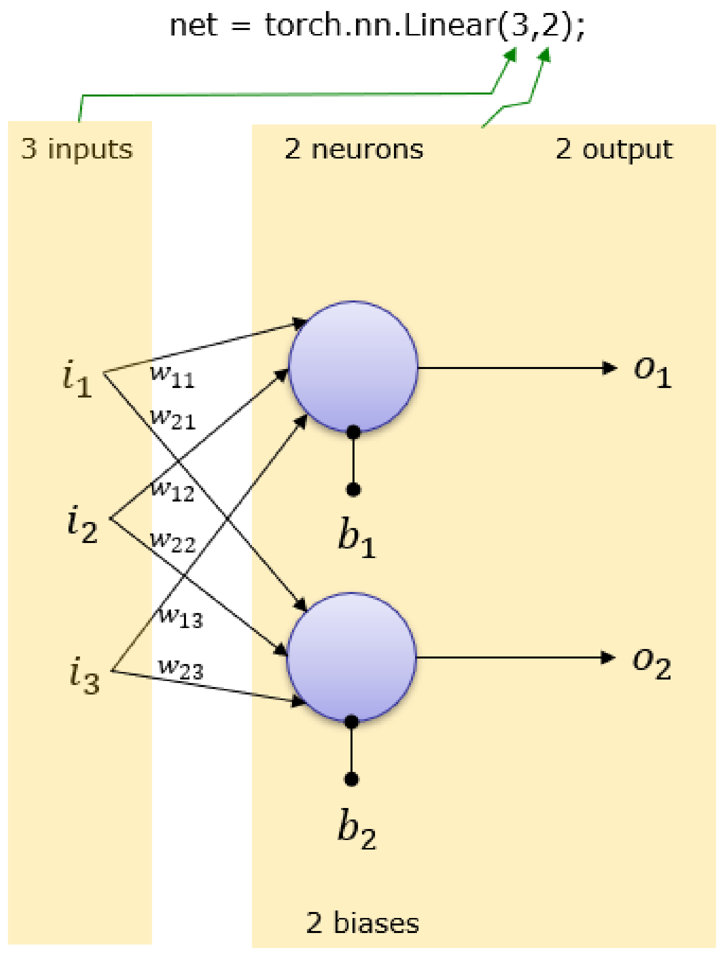
ReLU Activation layer

- Applies the rectified linear unit function element-wise
  - Input: can have any number of dimensions
  - Output: same shape as the input



Fully Connected Layer

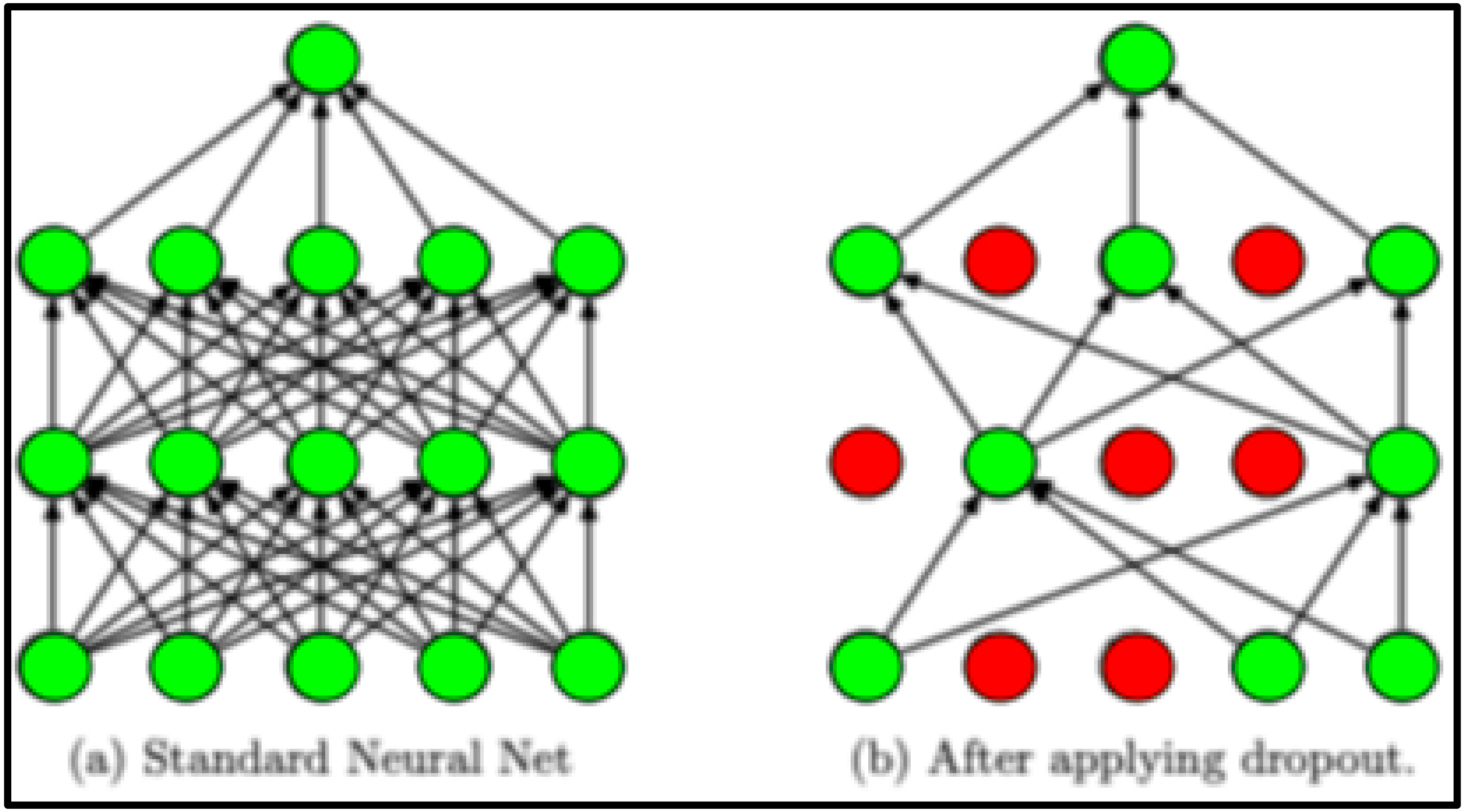
- The linear layer or Dense layer



Dropout Layer

- Dropout is very important deep learning layer
- It is used as a **regularizer**
  - Prevent **Overfitting** by forcing the model to find more than one way to achieve the target
  - Some features are randomly denied to model to achieve the target in a **different** way.
- Without dropout
  - Model may end up relying on a handful of features only because these features were found to be more relevant in the training set

Dropout Layer



Dropout Layer (Example)

```
dropping_model = nn.Sequential(nn.Dropout(p=0.5))
```

```
spaced_points = torch.linspace(.1, 1.1, 11)
```

```
tensor([0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000,
        0.8000, 0.9000, 1.0000, 1.1000])
```

```
dropping_model.train()
output_train = dropping_model(spaced_points)
output_train
```

```
tensor([0.0000, 0.4000, 0.0000, 0.8000, 0.0000, 1.2000, 1.4000,
        1.6000, 1.8000, 0.0000, 2.2000])
```

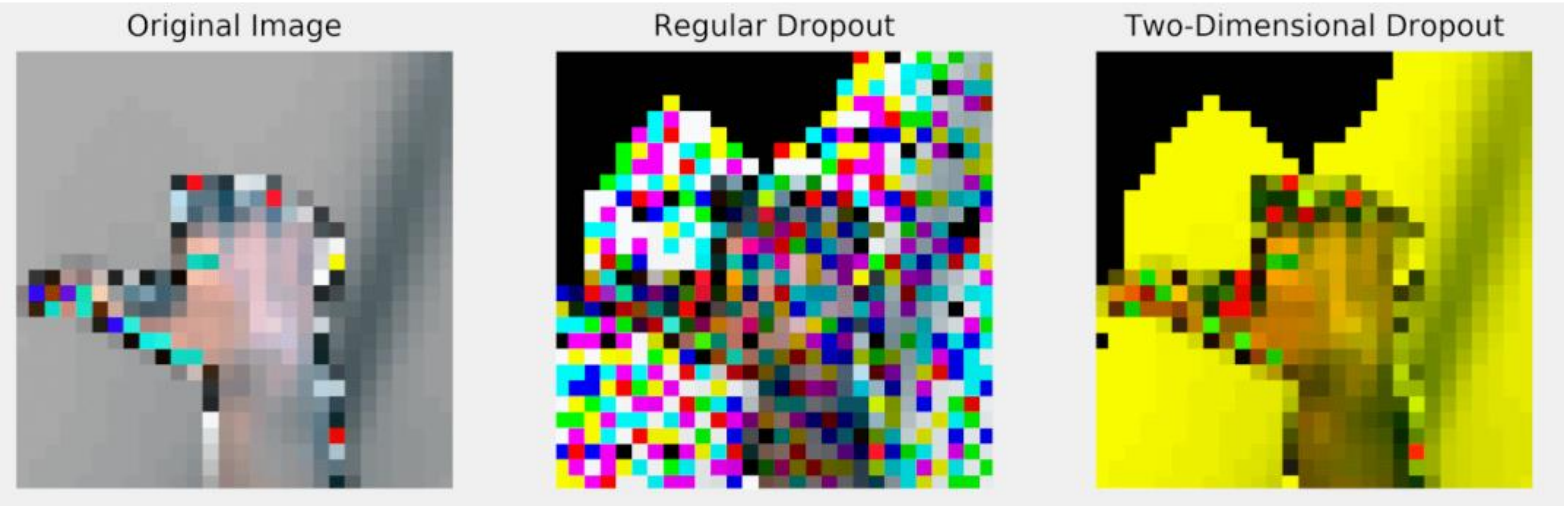
Dropout Facts

- Works on features not weights
- Works only when model is in training mode
- Drops feature according to the probability
- Rescale the remaining features

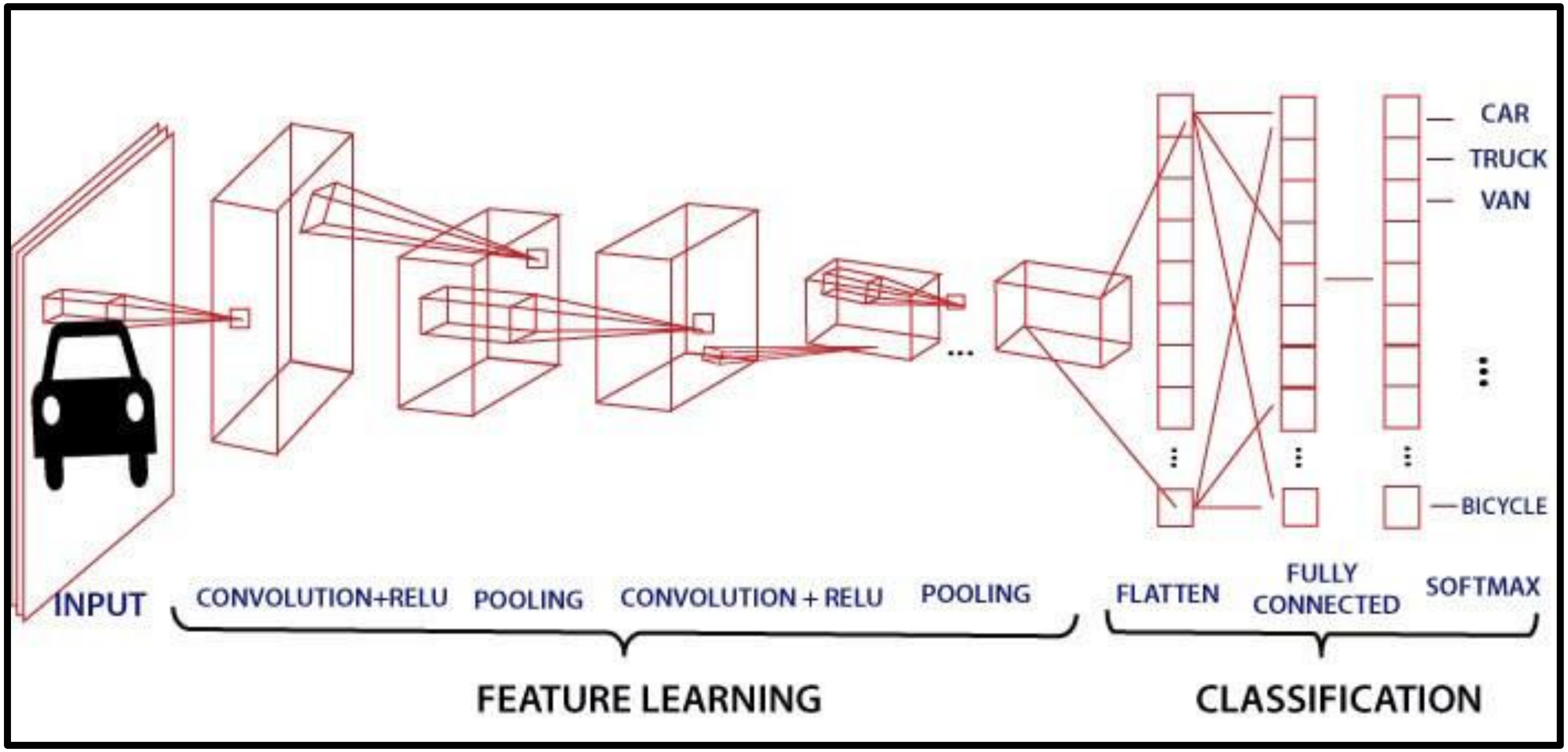
2D Dropout Layer

- used with convolutional layers
- It drops entire channels
- **nn.Dropout2d**
  - If a convolutional layer produces 10 channels
  - Two-dimensional dropout with a probability of 50% would drop five filters (on average),
  - Remaining filters would have all their pixel values left untouched

2D Dropout Layer



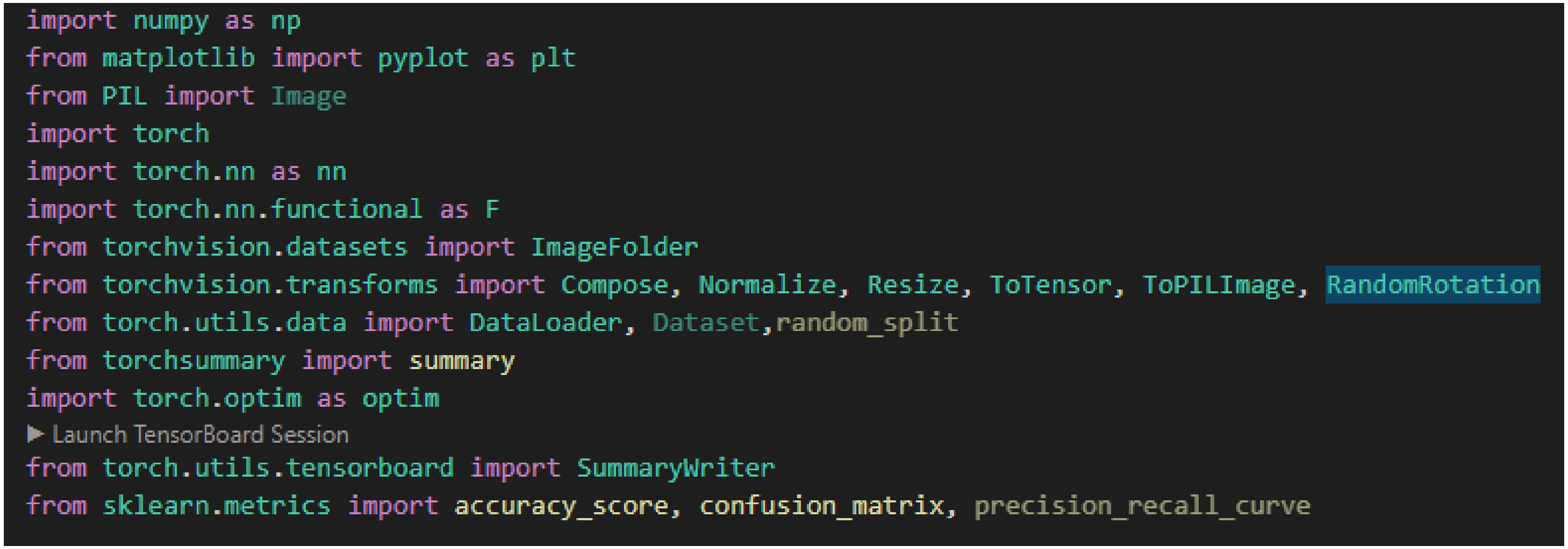
CNNs / ConvNets Typical Architecture



RPS Image Classification

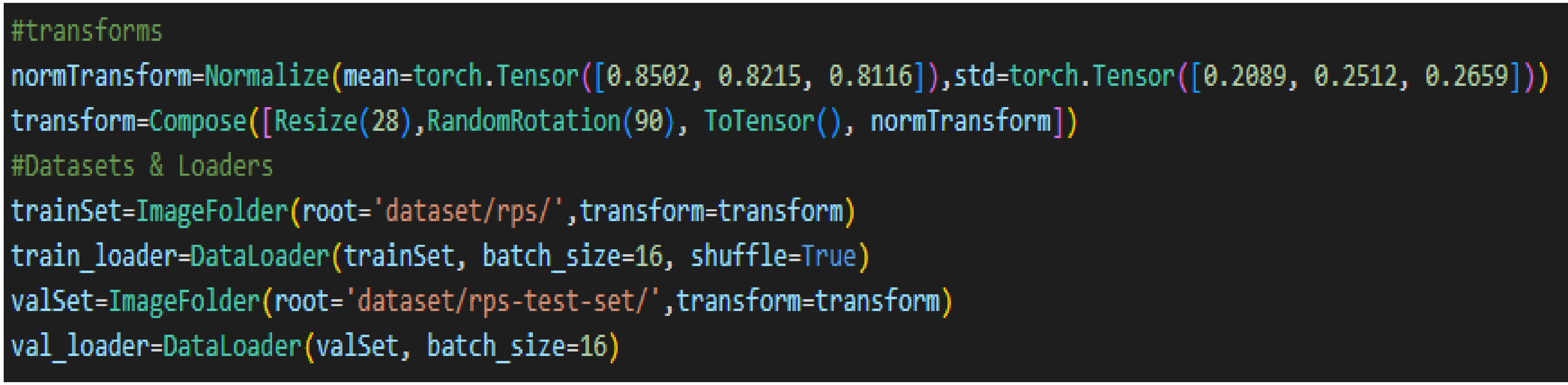
CNN (CODE)

Imports



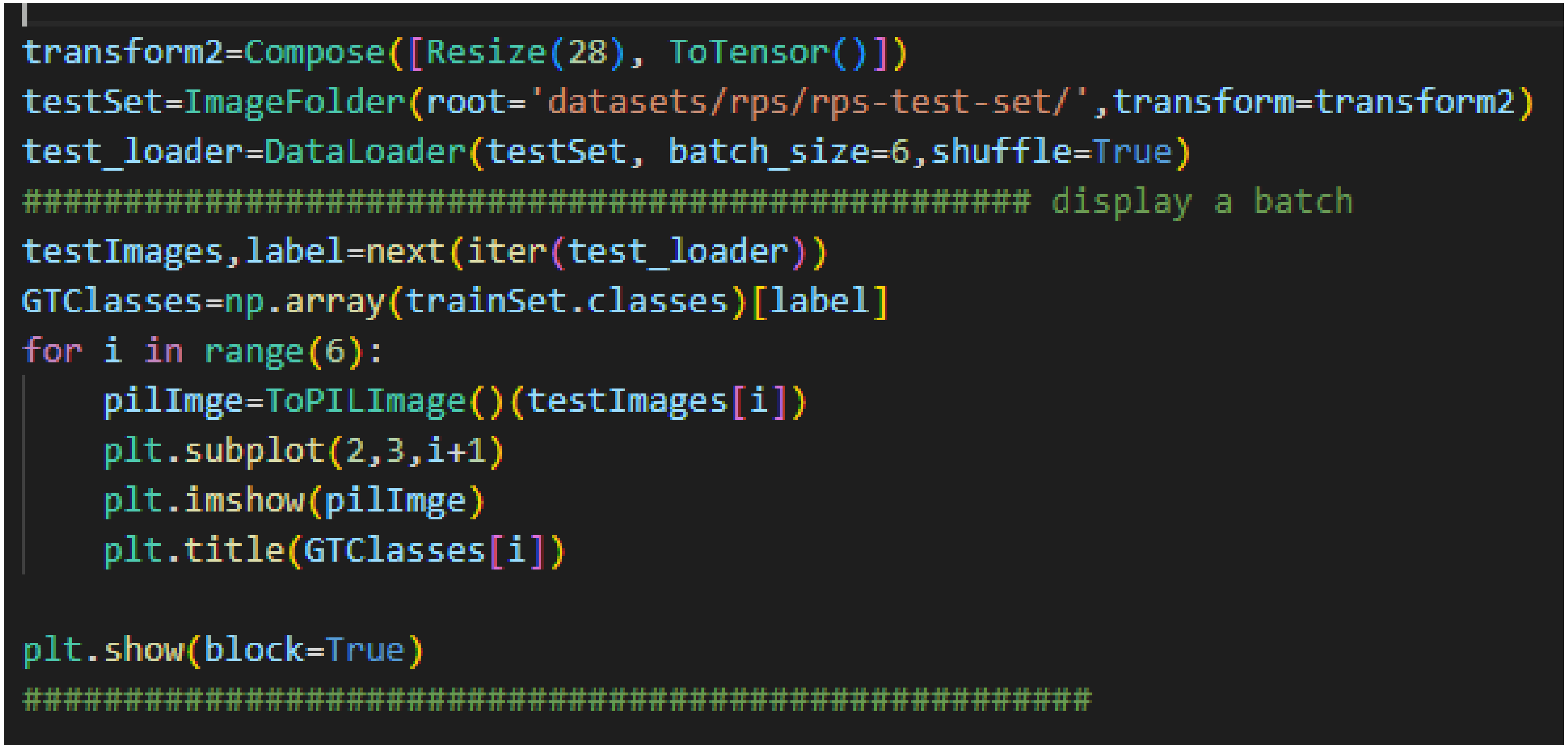
```
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import ImageFolder
from torchvision.transforms import Compose, Normalize, Resize, ToTensor, ToPILImage, RandomRotation
from torch.utils.data import DataLoader, Dataset,random_split
from torchsummary import summary
import torch.optim as optim
▶ Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter
from sklearn.metrics import accuracy_score, confusion_matrix, precision_recall_curve
```

Datasets/Dataloaders



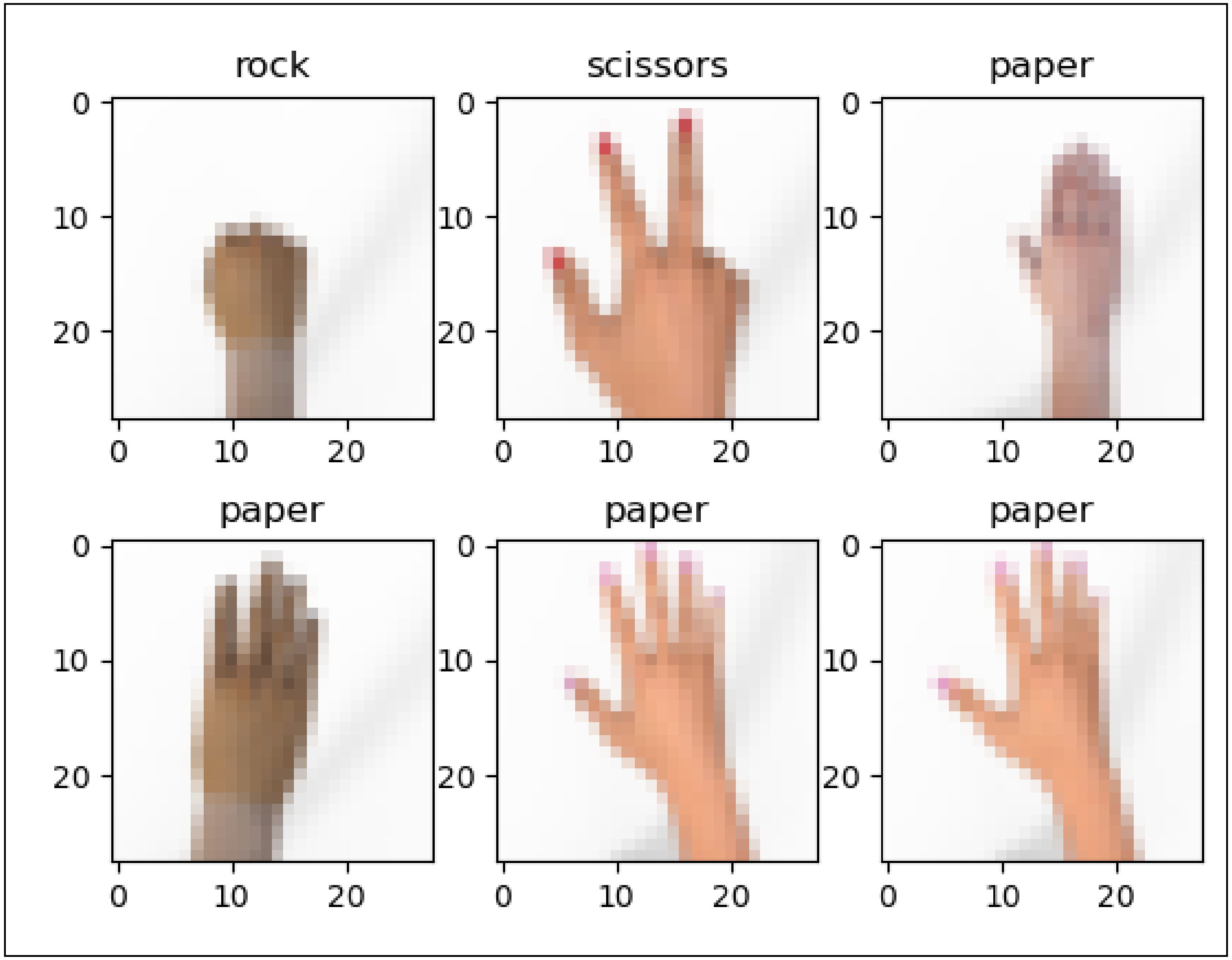
```
#transforms
normTransform=Normalize(mean=torch.Tensor([0.8502, 0.8215, 0.8116]),std=torch.Tensor([0.2089, 0.2512, 0.2659]))
transform=Compose([Resize(28),RandomRotation(90), ToTensor(), normTransform])
#Datasets & Loaders
trainSet=ImageFolder(root='dataset/rps/',transform=transform)
train_loader=DataLoader(trainSet, batch_size=16, shuffle=True)
valSet=ImageFolder(root='dataset/rps-test-set/',transform=transform)
val_loader=DataLoader(valSet, batch_size=16)
```

Displaying a test batch

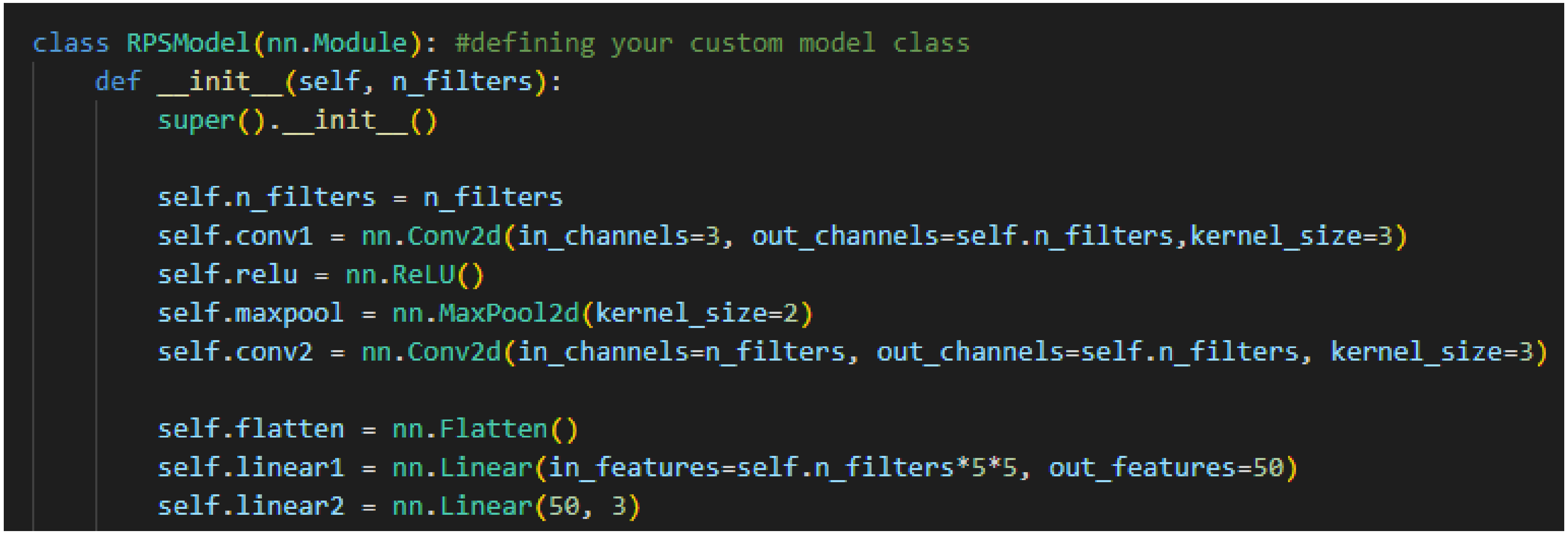


```
transform2=Compose([Resize(28), ToTensor()])
testSet=ImageFolder(root='datasets/rps/rps-test-set/',transform=transform2)
test_loader=DataLoader(testSet, batch_size=6,shuffle=True)
############################################################ display a batch
testImages,label=next(iter(test_loader))
GTClasses=np.array(trainSet.classes)[label]
for i in range(6):
    pilImge=ToPILImage()(testImages[i])
    plt.subplot(2,3,i+1)
    plt.imshow(pilImge)
    plt.title(GTClasses[i])

plt.show(block=True)
##############################################################
```

Displaying a test batch

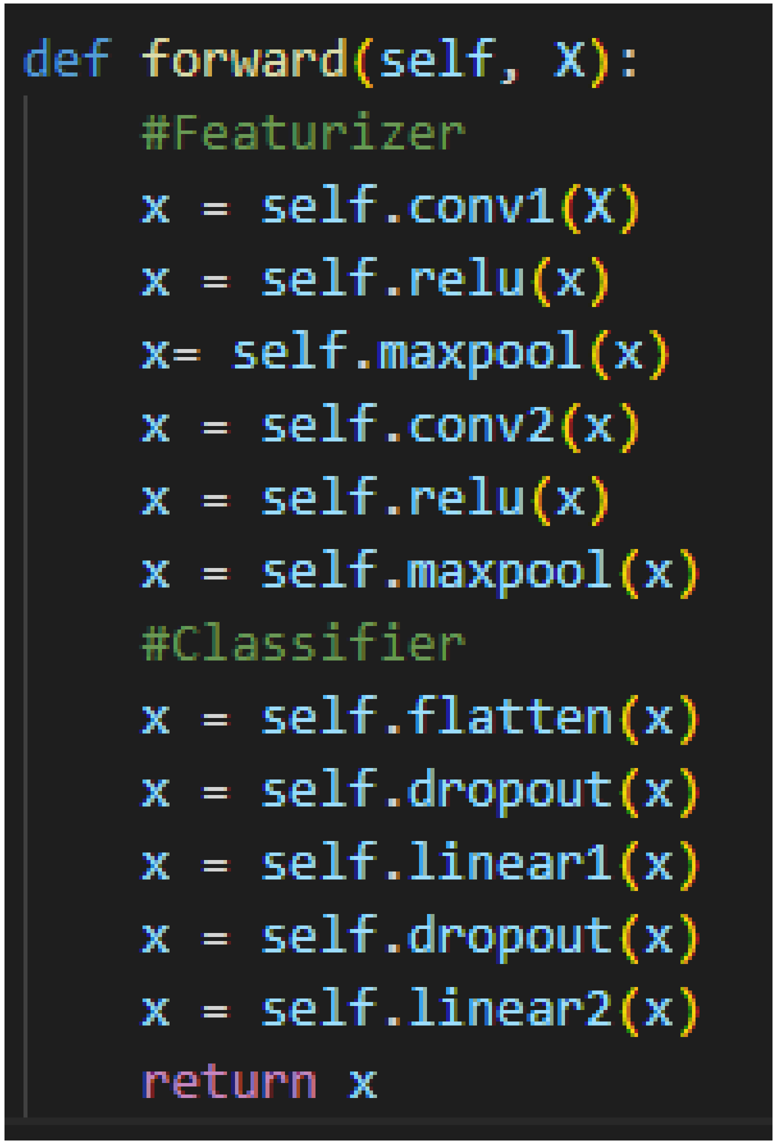


Model Class



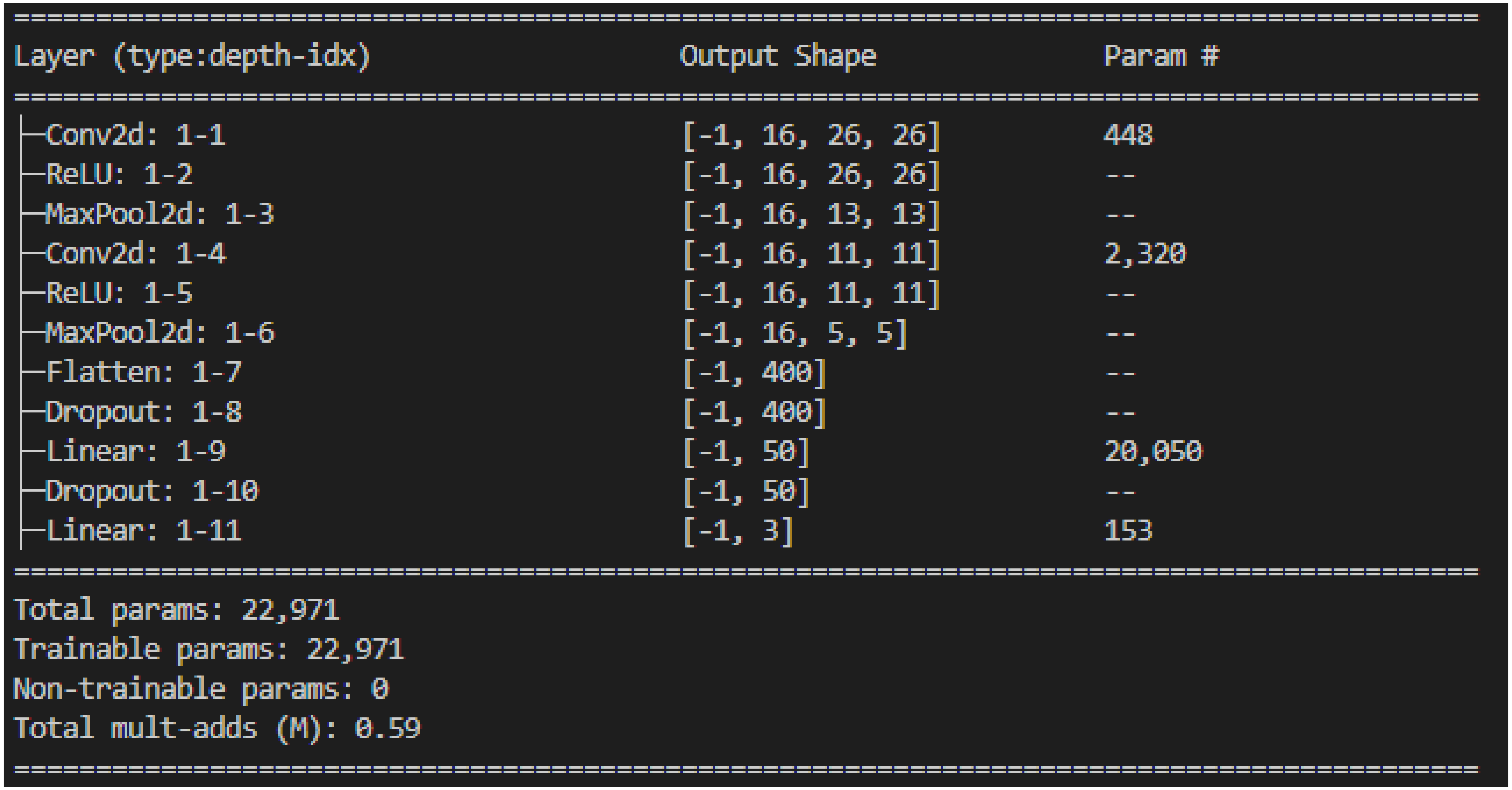
```
class RPSModel(nn.Module): #defining your custom model class
    def __init__(self, n_filters):
        super().__init__()

        self.n_filters = n_filters
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=self.n_filters,kernel_size=3)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2)
        self.conv2 = nn.Conv2d(in_channels=n_filters, out_channels=self.n_filters, kernel_size=3)

        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(in_features=self.n_filters*5*5, out_features=50)
        self.linear2 = nn.Linear(50, 3)
```

Model Class



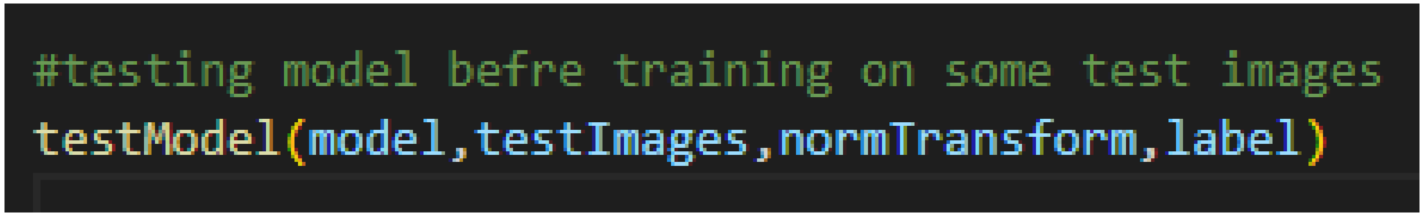
```
def forward(self, X):
    #Featurizer
    x = self.conv1(X)
    x = self.relu(x)
    x= self.maxpool(x)
    x = self.conv2(x)
    x = self.relu(x)
    x = self.maxpool(x)
    #Classifier
    x = self.flatten(x)
    x = self.dropout(x)
    x = self.linear1(x)
    x = self.dropout(x)
    x = self.linear2(x)
    return x
```

Model Parameters

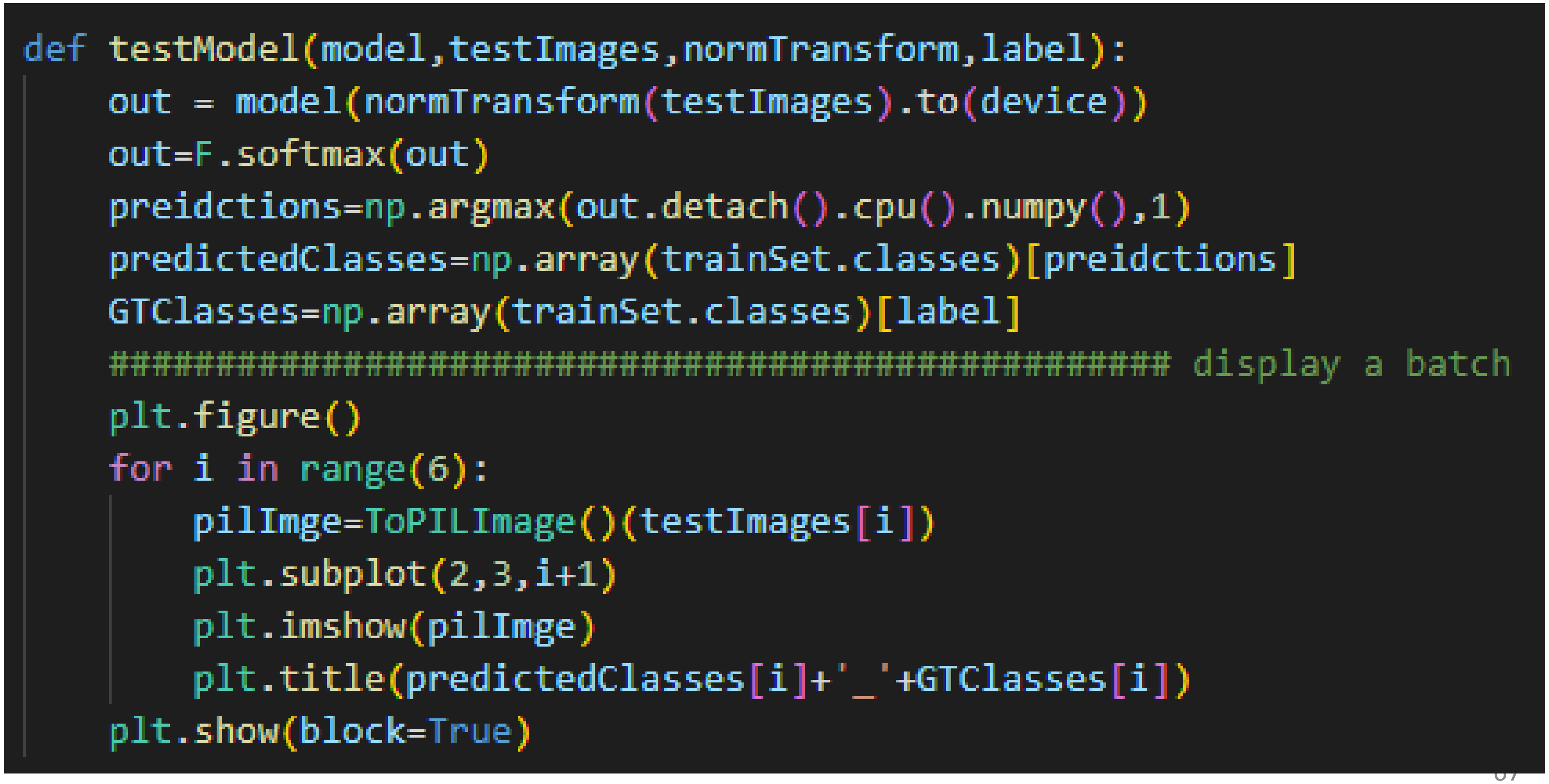


```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Conv2d: 1-1                            [-1, 16, 26, 26]          448
├─ReLU: 1-2                              [-1, 16, 26, 26]          --
├─MaxPool2d: 1-3                         [-1, 16, 13, 13]          --
├─Conv2d: 1-4                            [-1, 16, 11, 11]          2,320
├─ReLU: 1-5                              [-1, 16, 11, 11]          --
├─MaxPool2d: 1-6                         [-1, 16, 5, 5]            --
├─Flatten: 1-7                           [-1, 400]                 --
├─Dropout: 1-8                           [-1, 400]                 --
├─Linear: 1-9                            [-1, 50]                  20,050
├─Dropout: 1-10                          [-1, 50]                  --
├─Linear: 1-11                           [-1, 3]                   153
==========================================================================================
Total params: 22,971
Trainable params: 22,971
Non-trainable params: 0
Total mult-adds (M): 0.59
==========================================================================================
```

Testing Model on a batch before any training

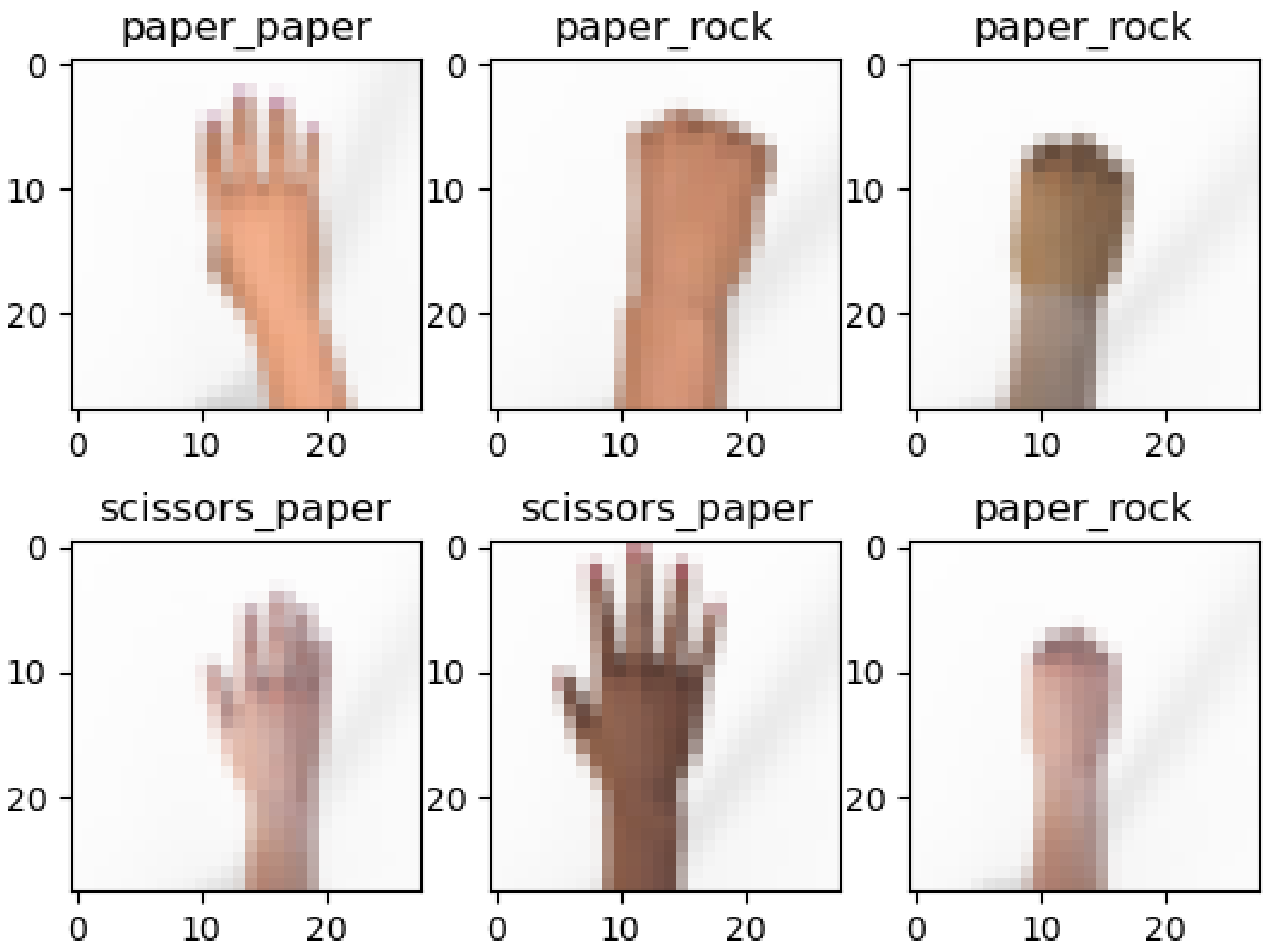


```python
#testing model befre training on some test images
testModel(model,testImages,normTransform,label)
```

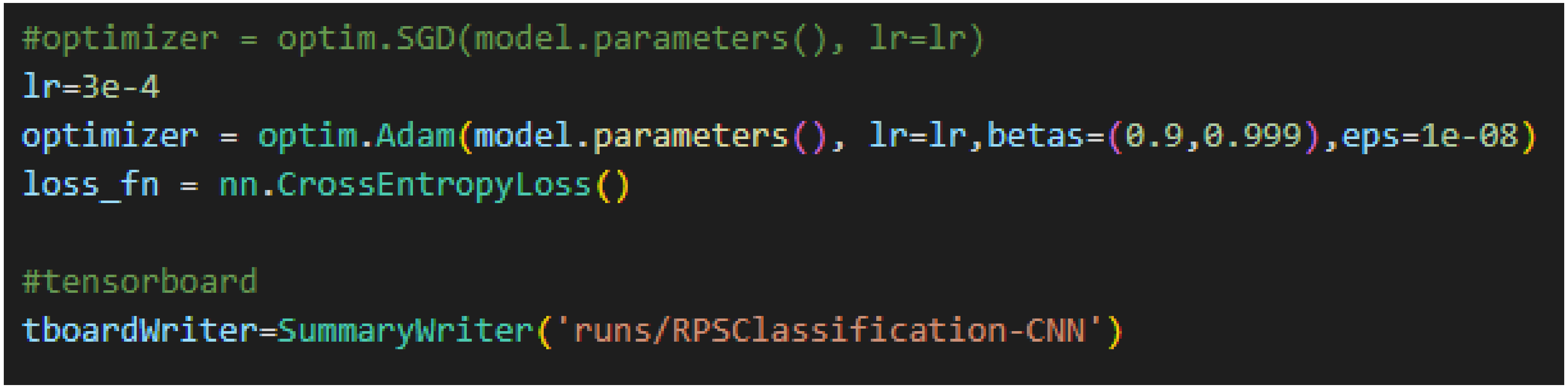


```python
def testModel(model,testImages,normTransform,label):
    out = model(normTransform(testImages).to(device))
    out=F.softmax(out)
    preidctions=np.argmax(out.detach().cpu().numpy(),1)
    predictedClasses=np.array(trainSet.classes)[preidctions]
    GTClasses=np.array(trainSet.classes)[label]
    ################################################### display a batch
    plt.figure()
    for i in range(6):
        pilImge=ToPILImage()(testImages[i])
        plt.subplot(2,3,i+1)
        plt.imshow(pilImge)
        plt.title(predictedClasses[i]+'_'+GTClasses[i])
    plt.show(block=True)
```

Testing Model on a batch before any training

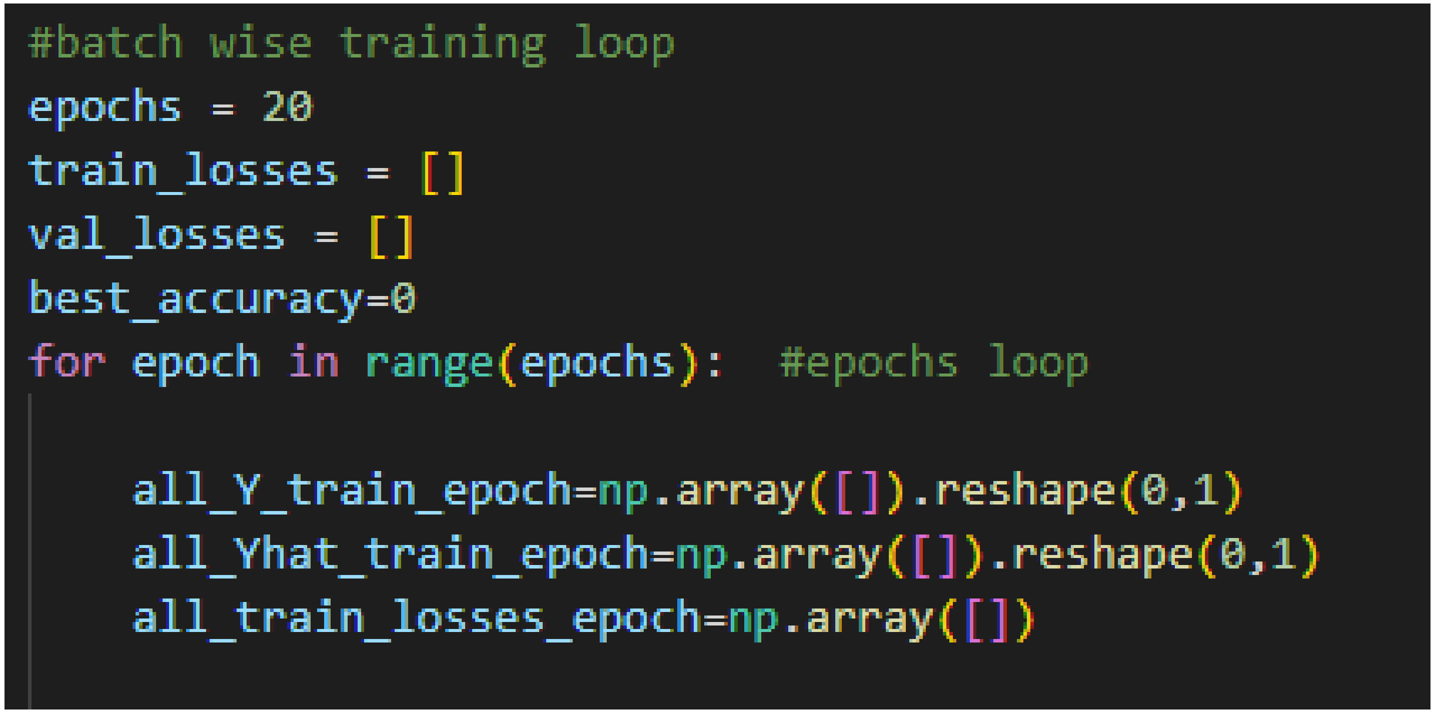


Loss & Optimizer



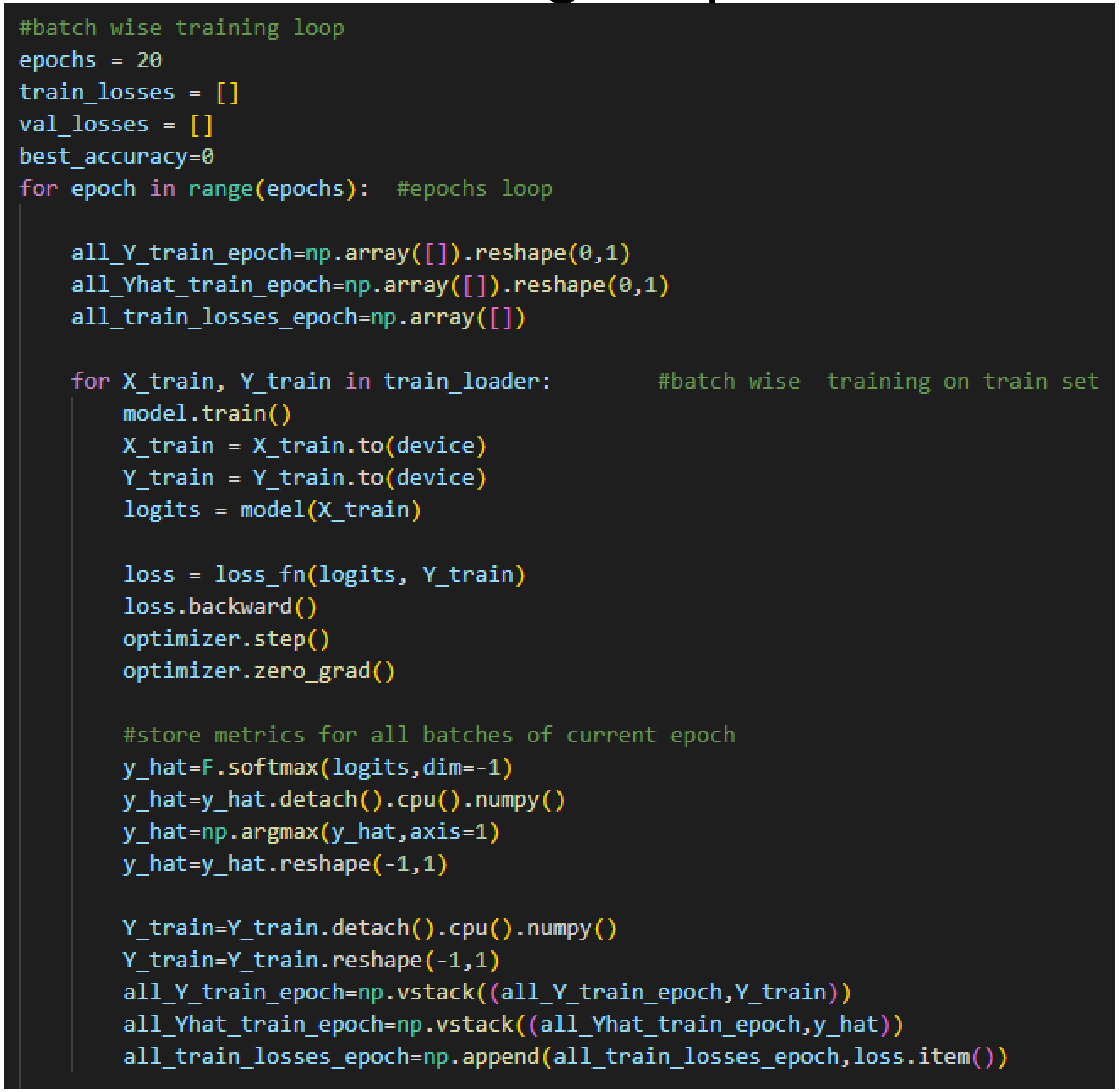
```
#optimizer = optim.SGD(model.parameters(), lr=lr)
lr=3e-4
optimizer = optim.Adam(model.parameters(), lr=lr,betas=(0.9,0.999),eps=1e-08)
loss_fn = nn.CrossEntropyLoss()

#tensorboard
tboardWriter=SummaryWriter('runs/RPSClassification-CNN')
```

Training Loop
(Exactly Same to IRISDataset That we did)



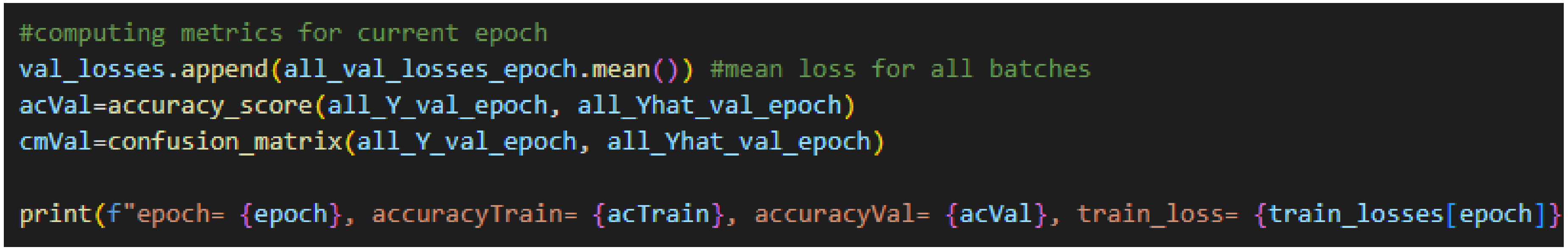
```
#batch wise training loop
epochs = 20
train_losses = []
val_losses = []
best_accuracy=0
for epoch in range(epochs):  #epochs loop

    all_Y_train_epoch=np.array([]).reshape(0,1)
    all_Yhat_train_epoch=np.array([]).reshape(0,1)
    all_train_losses_epoch=np.array([])

```

Training Loop



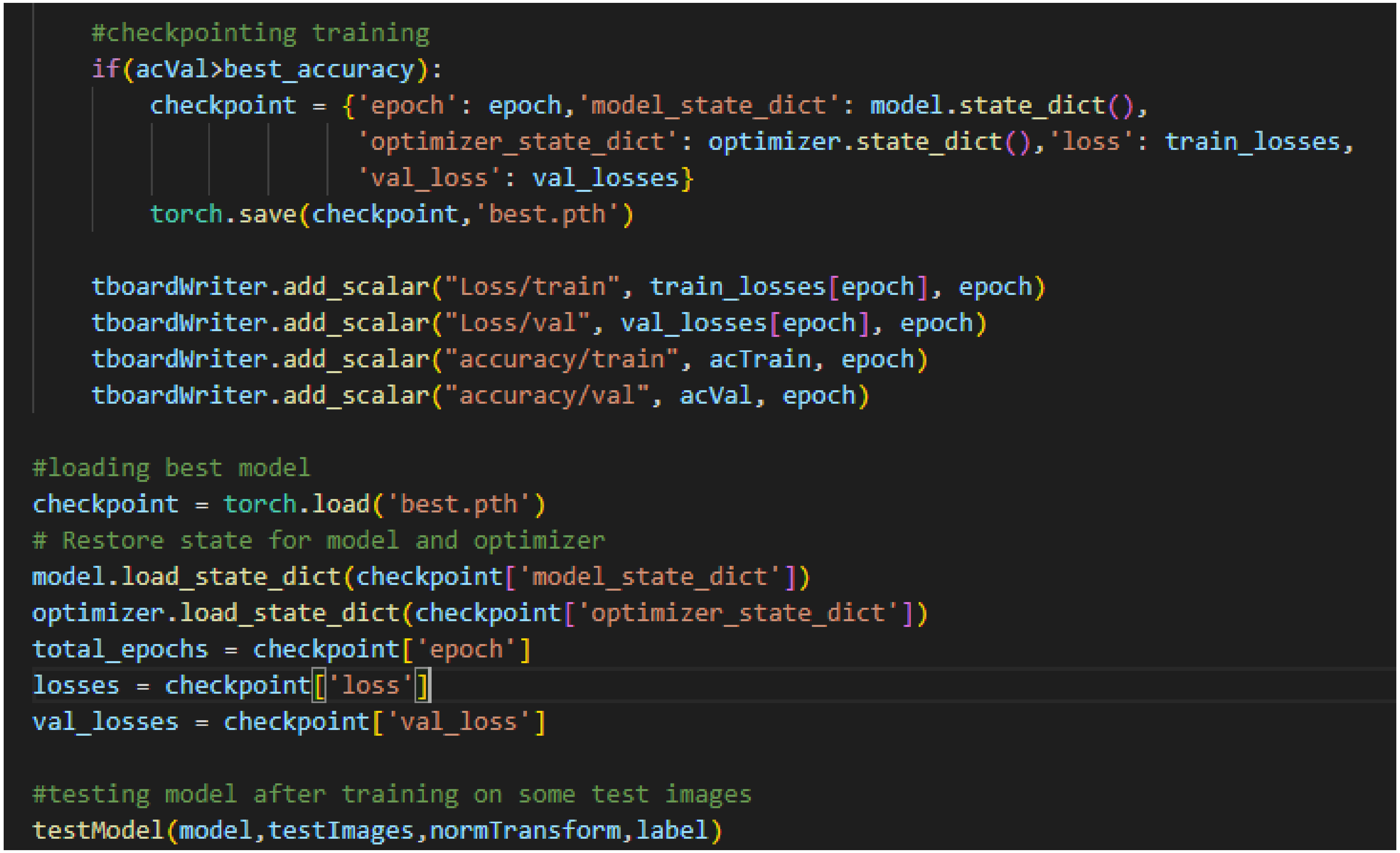
```
#batch wise training loop
epochs = 20
train_losses = []
val_losses = []
best_accuracy=0
for epoch in range(epochs):  #epochs loop

    all_Y_train_epoch=np.array([]).reshape(0,1)
    all_Yhat_train_epoch=np.array([]).reshape(0,1)
    all_train_losses_epoch=np.array([])

    for X_train, Y_train in train_loader:        #batch wise  training on train set
        model.train()
        X_train = X_train.to(device)
        Y_train = Y_train.to(device)
        logits = model(X_train)

        loss = loss_fn(logits, Y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        #store metrics for all batches of current epoch
        y_hat=F.softmax(logits,dim=-1)
        y_hat=y_hat.detach().cpu().numpy()
        y_hat=np.argmax(y_hat,axis=1)
        y_hat=y_hat.reshape(-1,1)

        Y_train=Y_train.detach().cpu().numpy()
        Y_train=Y_train.reshape(-1,1)
        all_Y_train_epoch=np.vstack((all_Y_train_epoch,Y_train))
        all_Yhat_train_epoch=np.vstack((all_Yhat_train_epoch,y_hat))
        all_train_losses_epoch=np.append(all_train_losses_epoch,loss.item())
```

Training Loop



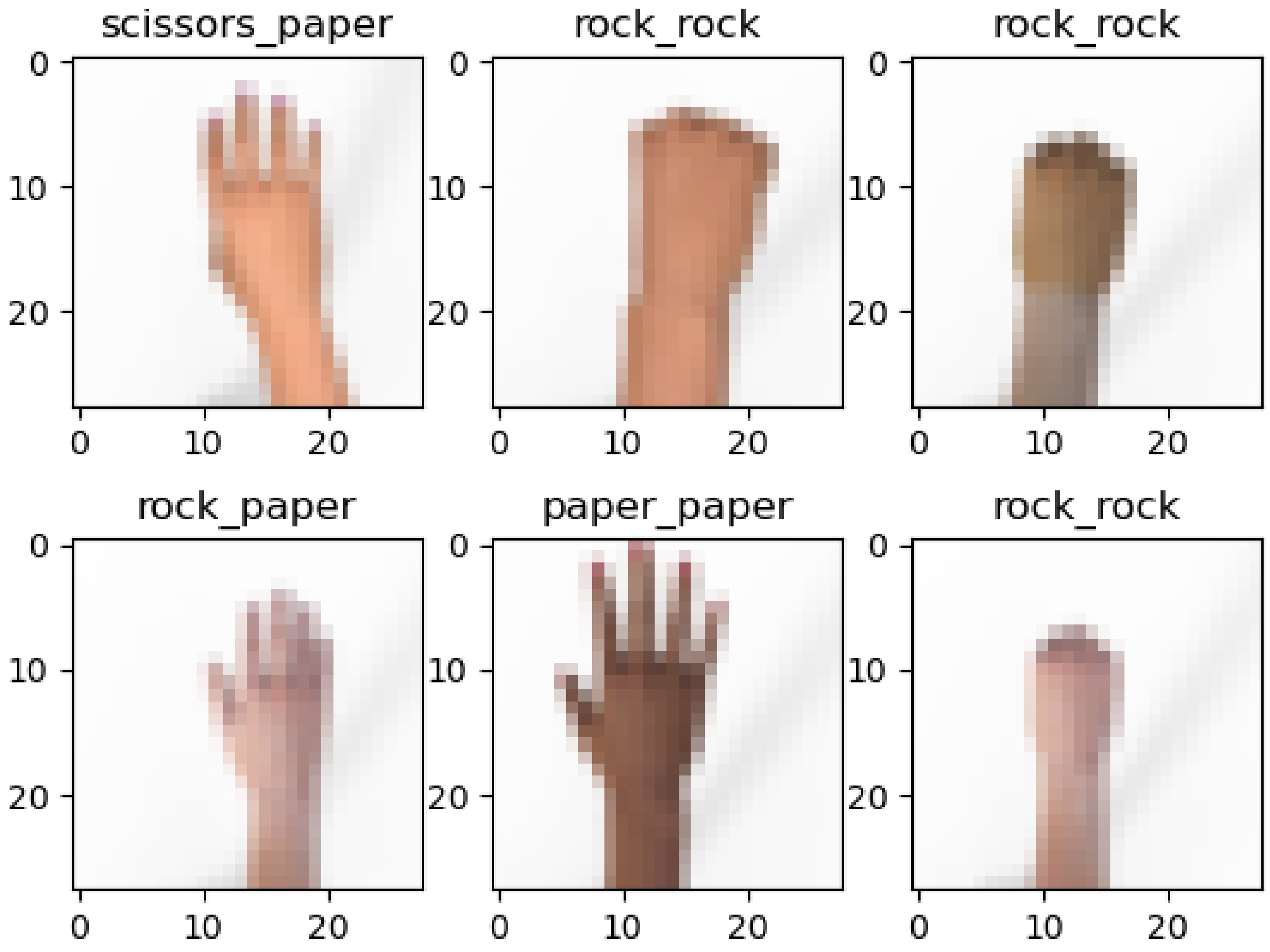
```
#computing metrics for current epoch
val_losses.append(all_val_losses_epoch.mean()) #mean loss for all batches
acVal=accuracy_score(all_Y_val_epoch, all_Yhat_val_epoch)
cmVal=confusion_matrix(all_Y_val_epoch, all_Yhat_val_epoch)

print(f"epoch= {epoch}, accuracyTrain= {acTrain}, accuracyVal= {acVal}, train_loss= {train_losses[epoch]}
```

Checkpointing & Testing



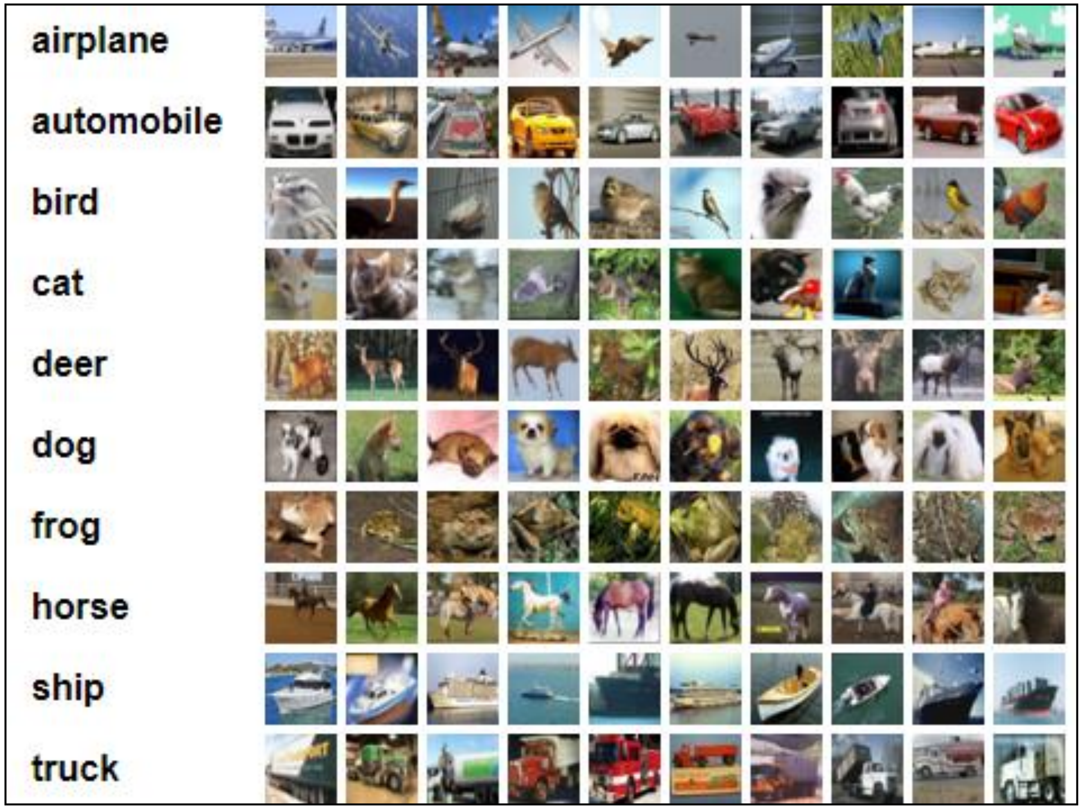
```
    #checkpointing training
    if(acVal>best_accuracy):
        checkpoint = {'epoch': epoch,'model_state_dict': model.state_dict(),
                      'optimizer_state_dict': optimizer.state_dict(),'loss': train_losses,
                      'val_loss': val_losses}
        torch.save(checkpoint,'best.pth')

    tboardWriter.add_scalar("Loss/train", train_losses[epoch], epoch)
    tboardWriter.add_scalar("Loss/val", val_losses[epoch], epoch)
    tboardWriter.add_scalar("accuracy/train", acTrain, epoch)
    tboardWriter.add_scalar("accuracy/val", acVal, epoch)

#loading best model
checkpoint = torch.load('best.pth')
# Restore state for model and optimizer
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
total_epochs = checkpoint['epoch']
losses = checkpoint['loss']
val_losses = checkpoint['val_loss']

#testing model after training on some test images
testModel(model,testImages,normTransform,label)
```

Testing Model on a batch after training



Graded Home Task-5

- The **CIFAR-10** dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class
- The dataset is divided into five training batches and one test batch, each with 10000 images

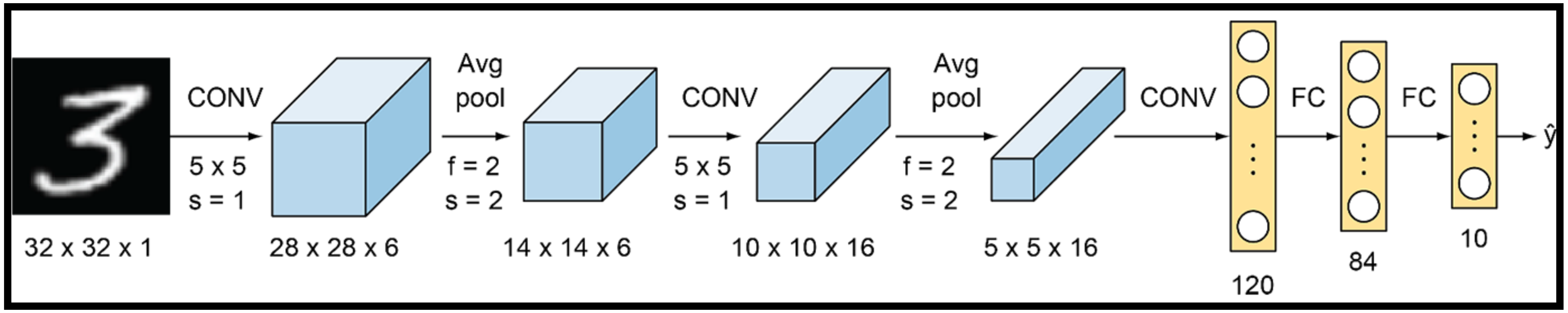


Graded Home Task-5

- Use torchvision.datasets.CIFAR10 to load the train & validation datasets
- Create normalization transform using CIFAR10 means & std
- Create data augmentation transforms
  - RandomCrop
  - RandomHorizontalFlip
  - RandomRotation

Graded Home Task-5

- Create your own model class for LeNet or AlexNet
  - Use tanh or ReLU activations



- Report results after 50 epochs
  - Train/Val accuracy
  - Confusion matrix