Deep Learning Basics

- Dr. Ahmad Altarawneh
- Department of Data Science
- Faculty of information technology, Mutah University

Outline

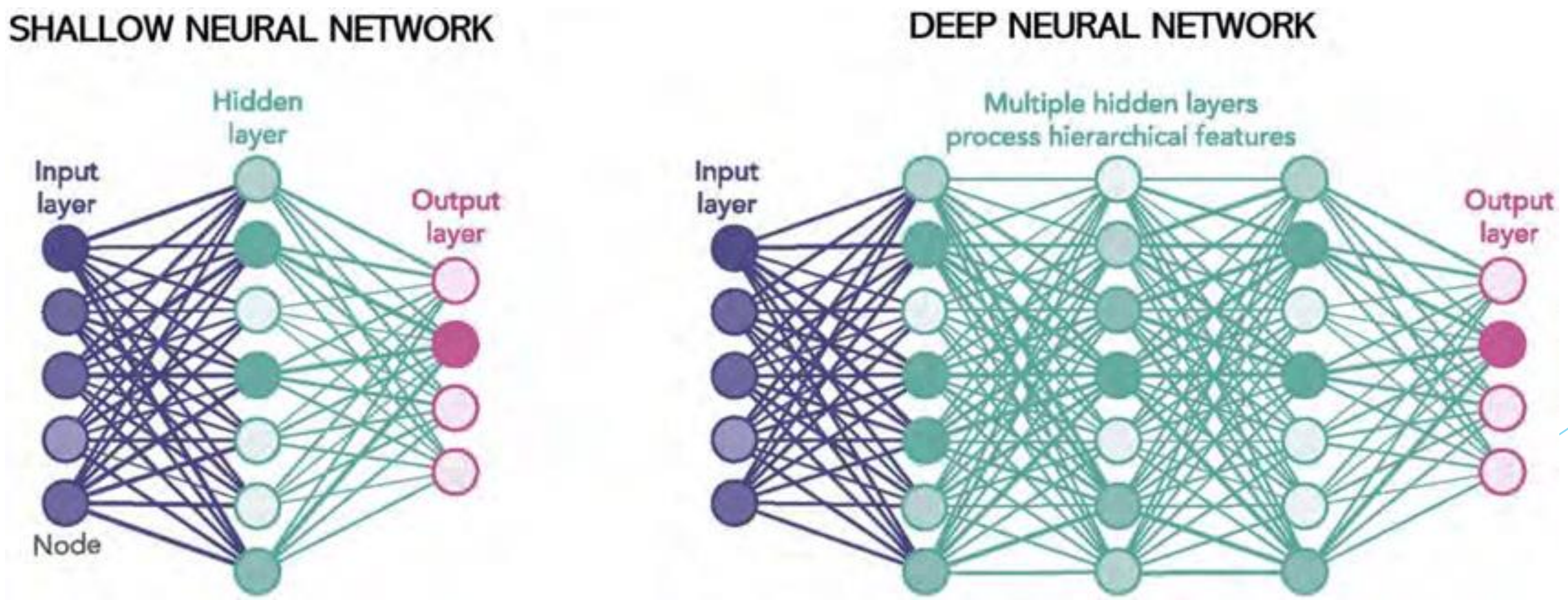
- What is Deep learning?
- Why do we need Deep learning?
- Deep Multi-layer Perceptron (DMLP)
  - Example with Pytorch
- Convolutional neural networks (CNN)
  - Example with Pytorch

What is deep learning (DL)

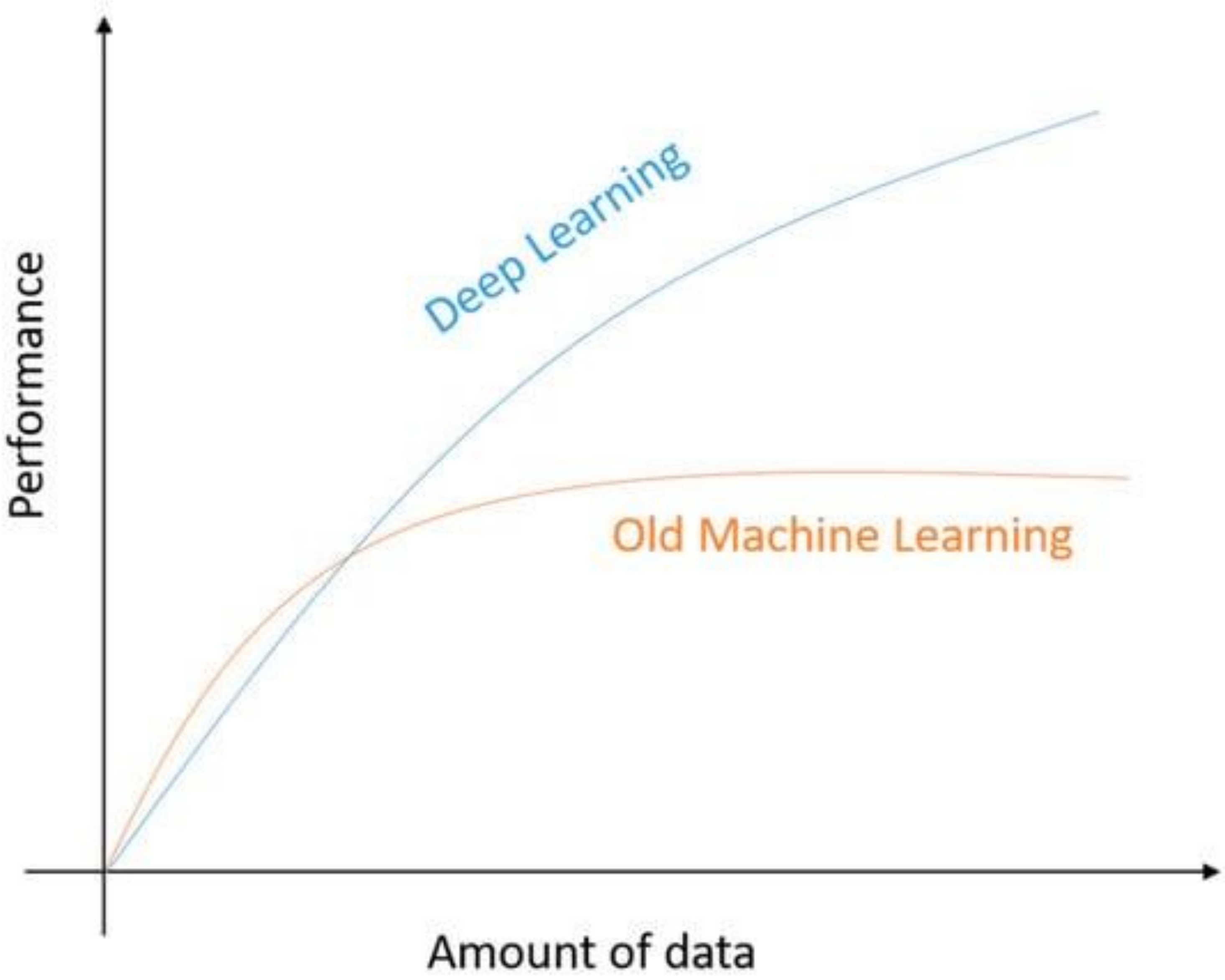
- A subset of machine learning that uses neural networks with multiple layers to model complex patterns in data
- DL excels in tasks such as image recognition, natural language processing, and speech recognition.
- A neural network with two or more hidden layers is indeed called a deep neural network (DNN).
  - This is true regardless of the specific structure or shape of the architecture (e.g., fully connected, convolutional, etc.).
- A neural network with one hidden layer is called a shallow neural network.

Why Deep learning

- Although shallow networks can capture non-linear relationships in data, it fails as the complexity and size of data grows
- More hidden layers means the model can model more complex patterns.

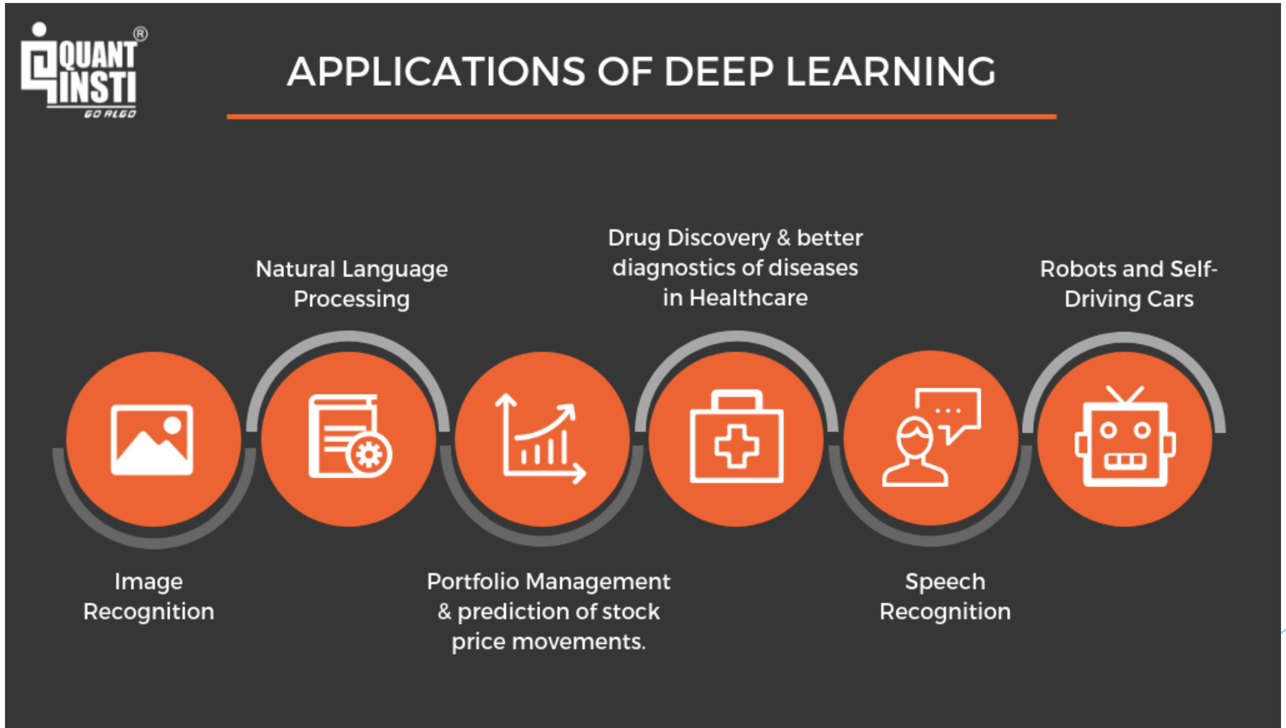


- As data grows in size, the performance of the traditional ML algorithms will not improve
  - Do not benefit from the data volume
- Deep learning methods keeps enhancing
  - More data means better performance



Applications of Deep Learning

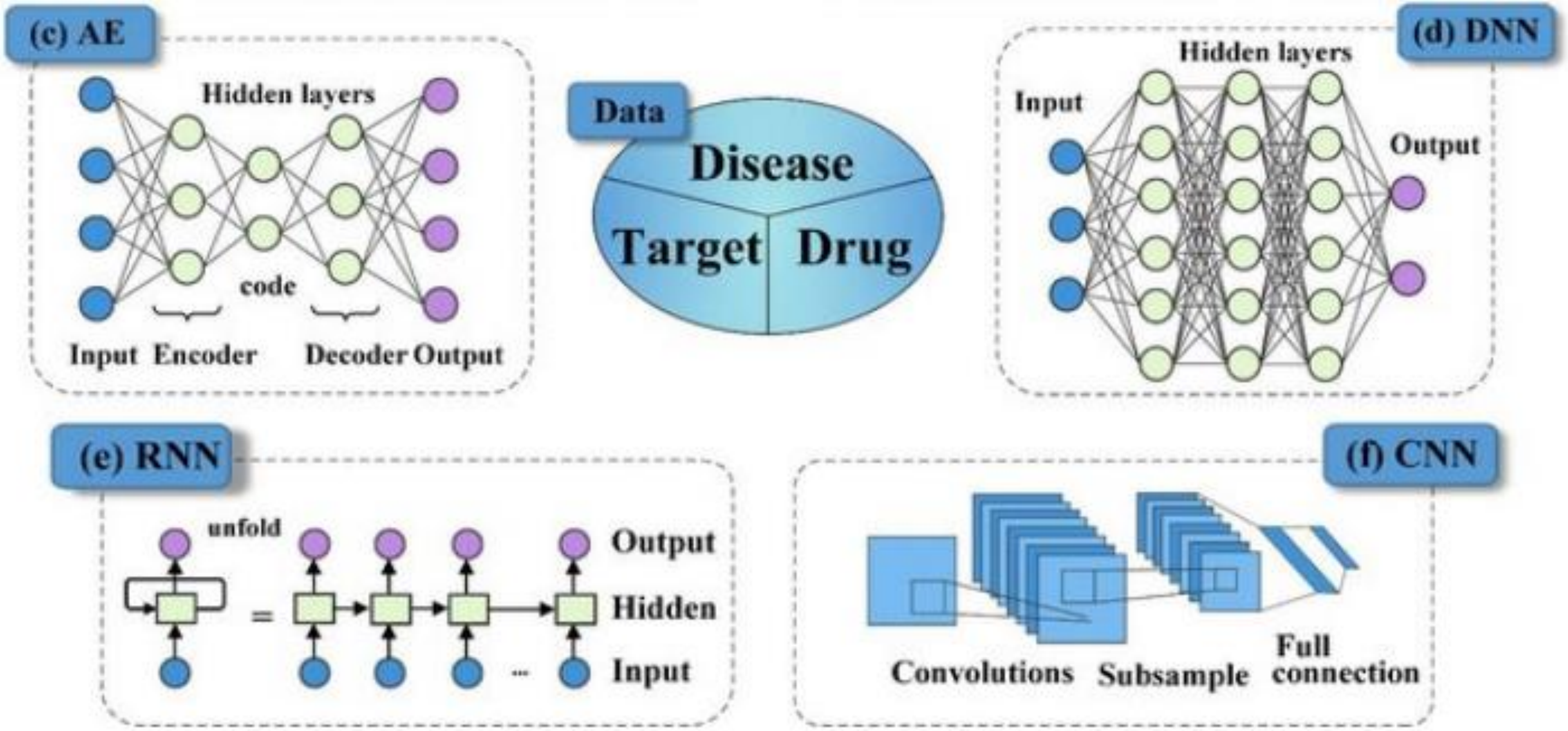
- **Computer Vision:** Image classification, object detection, facial recognition.
- **Natural Language Processing (NLP):** Chatbots, language translation, text classification.
- **Healthcare:** Disease diagnosis, drug discovery, personalized medicine.
- **Autonomous Systems:** Self-driving cars, drones, robotics.



Types of DL architectures

- **Deep MLP:** A type of neural network that consists of multiple layers of perceptrons, where each neuron is fully connected to the neurons in the subsequent layer
  - Tabular data
- **CNN:** A specialized type of neural network primarily used for processing structured grid-like data such as images.
- **Recurrent Neural Networks (RNN):** For sequential, timeseries data, like audio, texts
- **Autoencoders:** For compression, learning low dimensional representation of the data, image denoising
  - through bottleneck mechanism
- **Generative Networks:** Generate texts images, or music
- **Transformer:** Sequential data using the attention mechanism

Architectures



MLP using Pytorch

- In Pytorch building a DMLP is very simple, let us define a DMLP model

```
class MLP(nn.Module):
    def __init__(self, inputsD, outputsD):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(inputsD, 64)
        self.fc2 = nn.Linear(64, 128)
        self.fc3 = nn.Linear(128, 128)
        self.fc4 = nn.Linear(128, outputsD)

    def forward(self, X):
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = F.relu(self.fc3(X))
        X = self.fc4(X)
        return X
```

Here the last fully connected layer remains without activation as the loss function applies Softmax.

If another loss function is used, you can add the activation accordingly

Convolutional neural network

- **Convolutional Neural Networks (CNNs)** are named after the **convolution process** that occurs in the layers of the network.
- The convolution operation is a mathematical technique used to extract features from the input data (usually images) by applying filters (also called kernels) that move across the data.

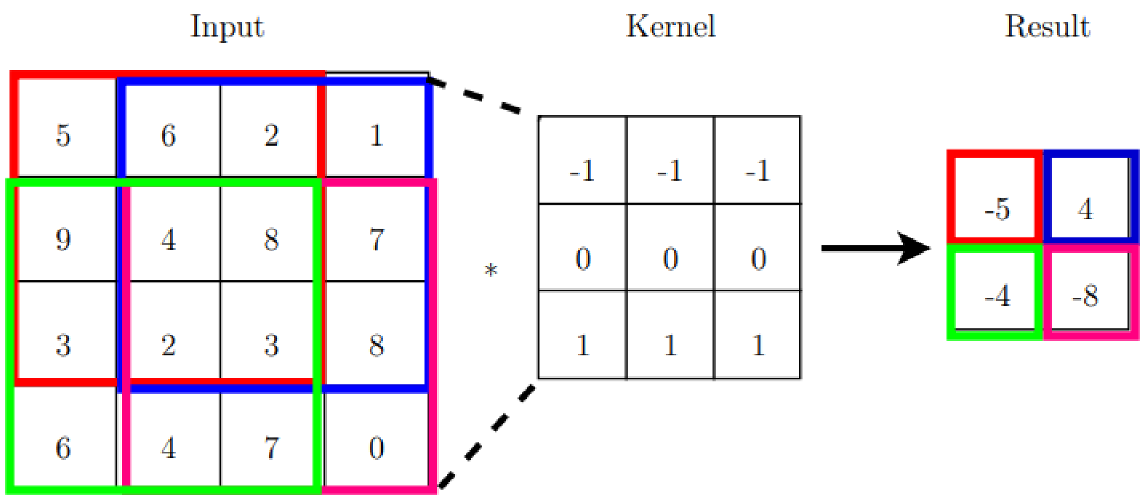
Convolution

- Convolution is an operation that combines two functions (or signals) to produce a third function that expresses how the shape of one is modified by the other
- It is similar to the cross correlation, which measures the similarity between two signals
  - In convolution the kernel should be flipped horizontally and vertically, to satisfy some mathematical properties
- The actual CNN use cross-correlation, where not flipping is needed, and it does not make significant difference in training the CNN

$$G(i,j) = h * f = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h(u,v) f(i-u, j-v)$$

How it works

- Imagine the kernel is **sliding** on top of the image, and for each step we calculate the weighted sum of the kernel values and its corresponding input values
  - The sliding step length is called the ***stride***



Original Grayscale Image

```
[[-0.64279312  0.96271155  1.29115632]
 [ 1.73101261  0.94550141  0.05336789]
 [ 0.57215926  0.21039854 -0.33334572]]
```

Convoluted Image

```
[[-0.21175006  0.09742543 -2.39730811]
 [-0.34625193 -1.06878272  0.96419665]
 [ 0.32762917 -2.19511065 -0.15130719]]
```

Convoluted Image

```
[[-1.21659707 -1.74555671  1.40724629]
 [ 0.29720501 -0.01492687  0.32592306]
 [ 0.24371565 -0.74327176  0.58953915]]
```

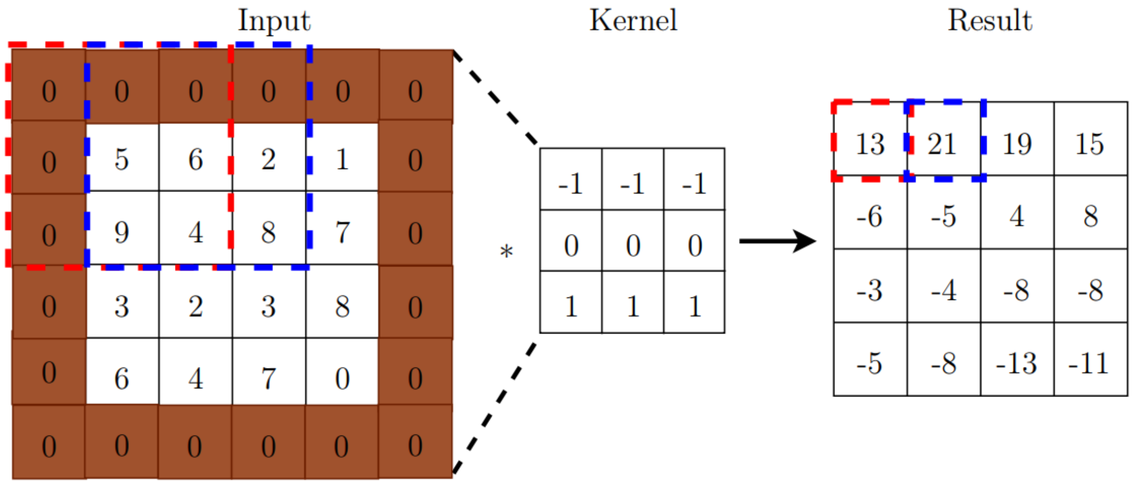
Convoluted Image

Cont.

- This operation results in another output image with a reduced size
- If one wants the same size to be preserved, we add paddings, as follows
  - **Note, stride value also affect the output shape**
- In general the following formulas are used to calculate the output shape after the convolution operation

$$\text{Output Height} = \left\lfloor \frac{\text{Input Height} + 2 \cdot \text{Padding Height} - \text{Kernel Height}}{\text{Stride Height}} + 1 \right\rfloor$$

$$\text{Output Width} = \left\lfloor \frac{\text{Input Width} + 2 \cdot \text{Padding Width} - \text{Kernel Width}}{\text{Stride Width}} + 1 \right\rfloor$$

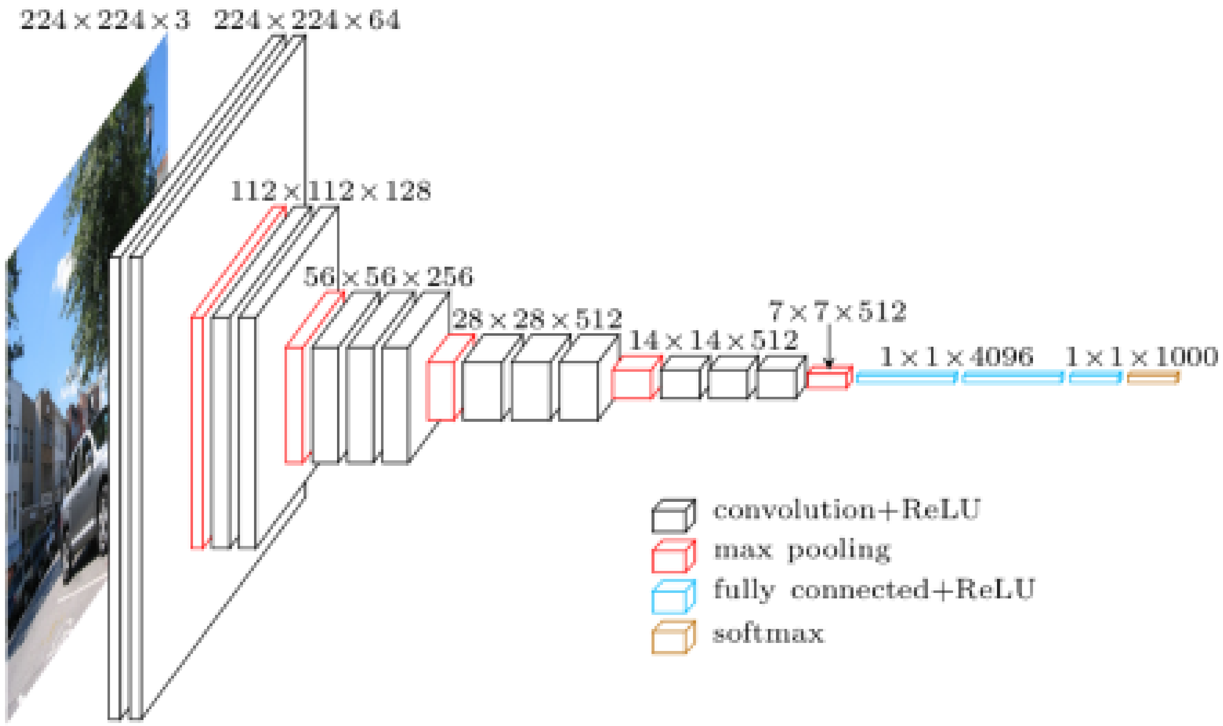


Kernels

- The depth of the kernel must align the depth of the input, which means if we have RGB image, the kernel will have 3 chennels as well.
- Likewise, If you have a **CNN layer with input of depth 50 and 100 kernels (filters)**, then each kernel will have a size of 3 X 3 X 50.
- Each of these filters is applied independently to the input and produce a feature map, each feature map is of depth 1.
  - Means layer of 100 kernels will give 100 feature maps
- These Kernel values are the weights, which will be tuned during the training
- Biases are added to each value in the feature map after the weighted sum

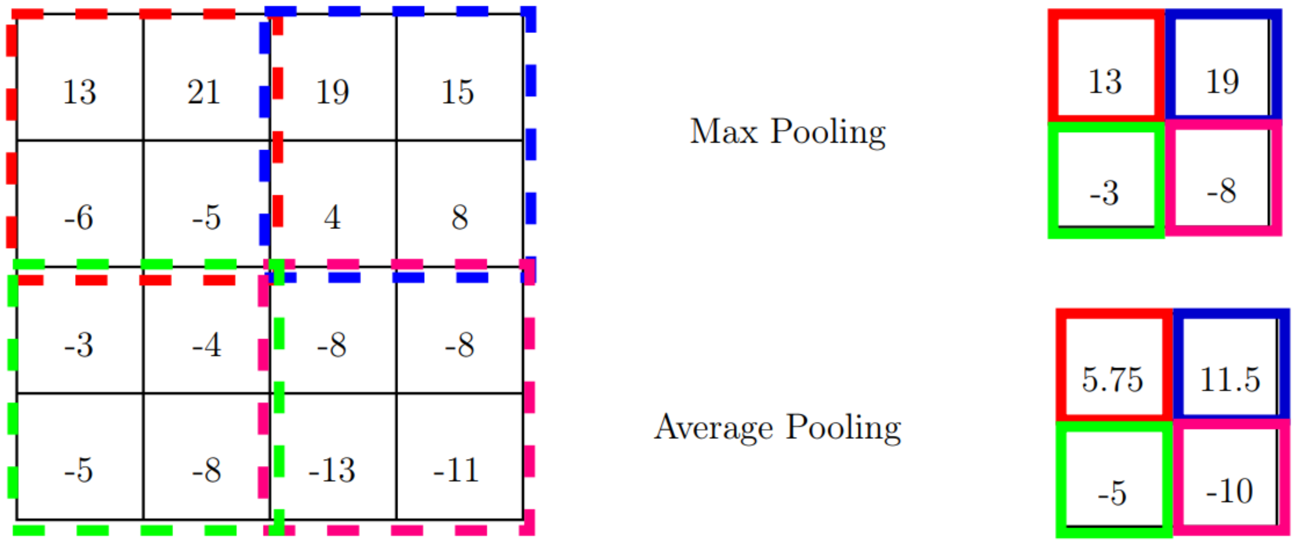
Activation

- Each Conv layer is activated by a non-linear activation function, e.g., ReLU
- The following Figure shows the architecture of the VGG network



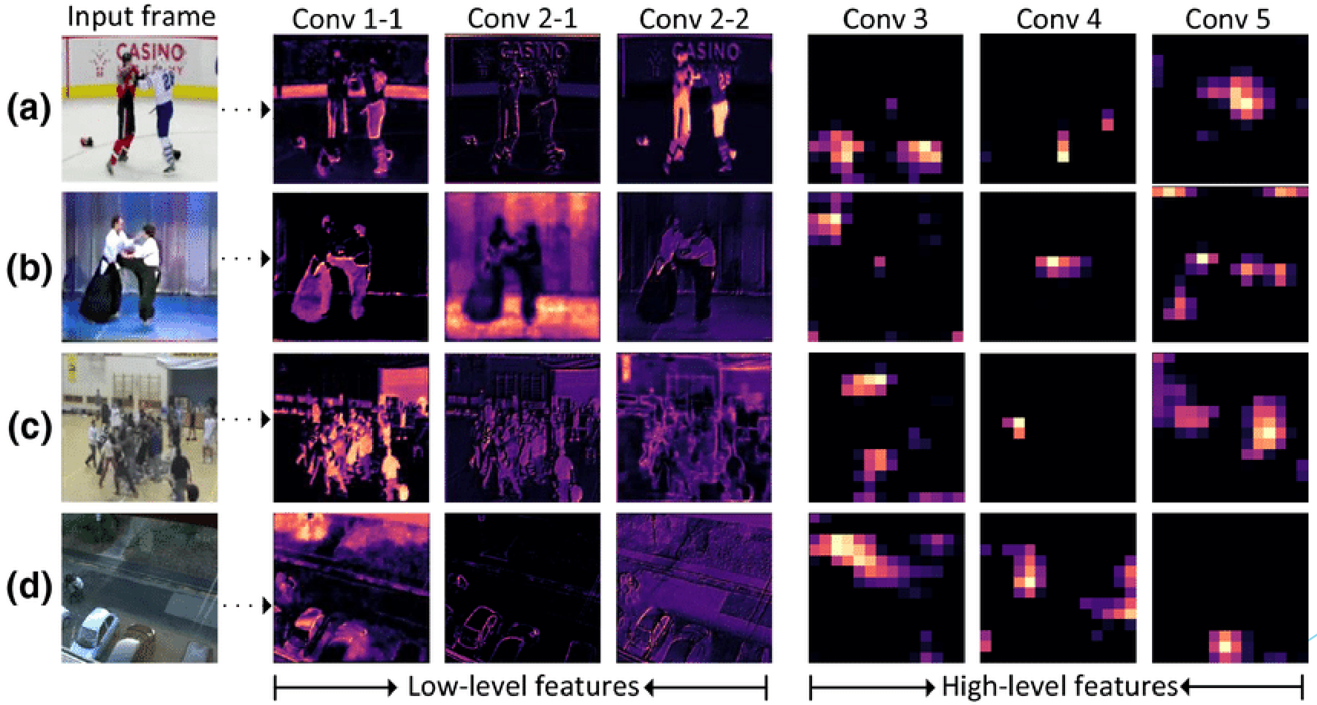
Pooling layer

- Pooling layer is used to reduce the spatial shape of the input.
- It helps in providing less dimensional data (less parameters) while preserving the most important features
- The standard pooling uses non overlapping window, although overlapping can be used
  - **Max** and **Average** pooling are popular



Feature maps

- The figure shows how different layers produce different representations of the features (feature maps)
  - High level features contains more features combined from the previous feature maps from previous layers.



Final notes

- Fine-tuning and feature extraction will be discussed in the next course (neural networks)
- Calculating the gradients and update the parameters is similar to the way we studied, but the procedure is different due to the way CNN deals with data
  - Refer to the following link to get idea about it
  - https://www.youtube.com/watch?v=z9hJzduHToc

Training CNN

- There are three ways to use CNN:
  - Training from scratch
  - Fine-tuning: Use a pretrained model and adopt it to a new dataset
  - Feature extraction: Use a pretrained model to extract features, from feature maps or from fully connected layers
- In the next slides we will see an example on training CNN from scratch, using Pytorch.

Training CNN from scratch

Example using Pytorch

- **Step 1: prepare the data**
- Here we will use the popular dataset MNIST
- The images are organized as in the following figure
  - Train folder: Contains subfolder for each class and each subfolder has the corresponding images that belong to that class
  - Test folder: Same organization as training
- Then import the needed libraries

```python
import torch
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import numpy as np
import os
import torch
import torch.nn as nn
import torch.optim as optim
```

Cont.

- **Step 2: Create datasets and dataloaders for training and testing**

```python
class MNISTDataset(Dataset):
    def __init__(self, root_dir):

        self.root_dir = root_dir
        self.image_paths = []  # image paths
        self.labels = []       # labels

        for label in os.listdir(root_dir):
            label_dir = os.path.join(root_dir, label)
            if os.path.isdir(label_dir):
                for img_file in os.listdir(label_dir):
                    img_path = os.path.join(label_dir, img_file)
                    self.image_paths.append(img_path)
                    self.labels.append(int(label))
    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):

        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert("L")
        label = self.labels[idx]
        image = torch.tensor(np.array(image), dtype=torch.float32) / 255.0
        image = image.unsqueeze(0)
        label = torch.tensor(label, dtype=torch.long)
        return image, label
```

```python
train_dataset = MNISTDataset(root_dir='mnist_images/train')
test_dataset = MNISTDataset(root_dir='mnist_images/test')

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

Cont.

- **Step 2: Create datasets and dataloaders for training and testing**

- **ALTERNATIVE APPROACH**
- **Creating the dataset from ImageFolder, directly**

```
transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Step 2: Create the training dataset using ImageFolder
train_dataset = datasets.ImageFolder(root='mnist_images/train', transform=transforn
test_dataset = datasets.ImageFolder(root='mnist_images/test', transform=transform)


train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

Cont.

- **Step 3: Define the model**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Cont.

- **Step 4: Create the model, loss and optimizer**

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = CNN().to(device)

criterion = nn.CrossEntropyLoss()  # For classification tasks
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- **It is better to use GPU if available as it speeds the training process.**
- **The model and data should be moved to the GPU during the training**

Cont.

- **Step 5: Define the model**

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Cont.

- **Step 6: Define the training function**

```
def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()  # Set model to training mode
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device) # Move the data to GPU, if available

        optimizer.zero_grad()

        output = model(data)
        loss = criterion(output, target)

        loss.backward()

        optimizer.step()

        if batch_idx % 100 == 0:
            print(f'Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)}] Loss: {loss.item():.6f}')
```

Cont.

- **Step 7: Define the testing function**

```
def test(model, device, test_loader, criterion):
    model.eval() # Evaluation mode
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f'\nTest: Average loss: {test_loss:.4f}, Accuracy: {correct}/{len(test_loader.dataset)}
({accuracy:.2f}%)\n')
```

Cont.

- **Step 8: Final training loop**

```
num_epochs = 5
for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader, optimizer, criterion, epoch)
    test(model, device, test_loader, criterion)
```

Final note

- Instead of applying the normalization and conversion to tensors manually, one can use transforms
- Transforms contains a lot of functionality that applies to the input image
  - Used in the augmentation process
  - it can be defined as follows, and passed to the dataset constructor

```
data_transform = transforms.Compose([
                        transforms.RandomSizedCrop(224),
                        transforms.RandomHorizontalFlip(),
                        transforms.ToTensor(),
                        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                             std=[0.229, 0.224, 0.225])
                    ])
```

  - And before returning the image we pass it to the transform we defined

```
def __init__(self, root_dir, transform=None):
            self.root_dir = root_dir
            self.transform = transform

          ...
def __getitem__(self, idx):
          ...
     if self.transform:
         sample = self.transform(sample)
```

Cont.

- Deep learning is powerful in performing several AI and machine learning-based applications
- Using deep learning requires tuning large number of hyperparameters to fit the problem in hand
  - Number of layers, type of activation functions, learning rate, optimizer, number of filters, etc.
  - Selecting the best parameters is largely dependent upon experience and trial-error process
- Deep learning is more vulnerable to overfitting as the model size grows.
  - Needs large amount of data for good fitting
  - Augmentation can be used to enrich the dataset