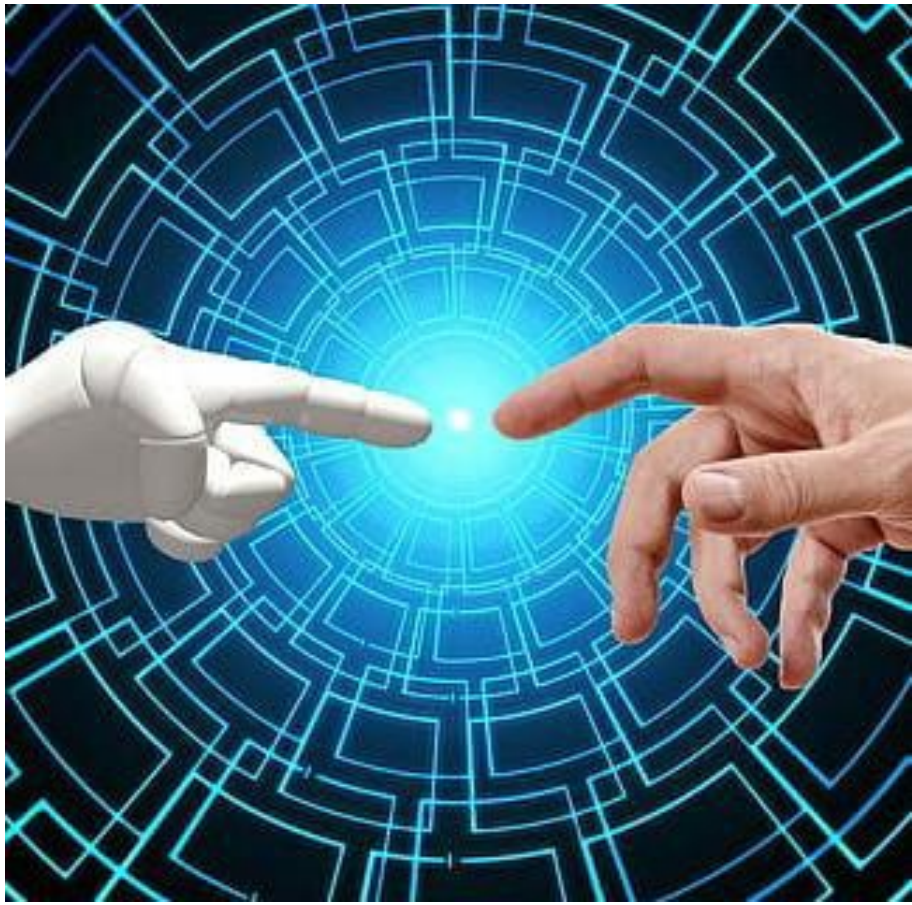
Artificial Intelligence (AI) for Engineering

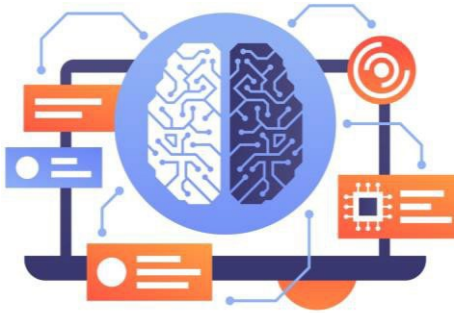


COS40007

Dr. Afzal Azeem Chowdhary

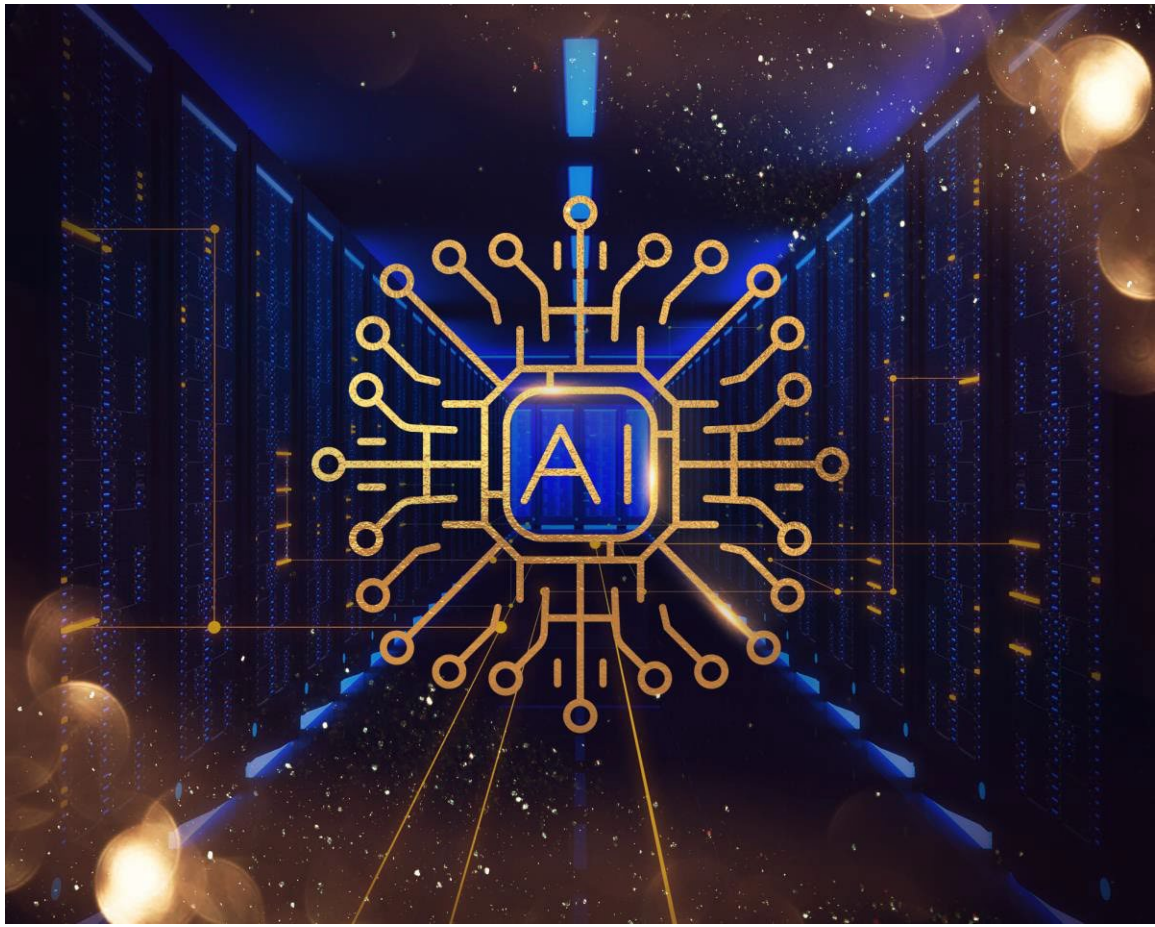
Lecturer, SoCET, Swinburne University of Technology

Seminar 5: 1st April 2025



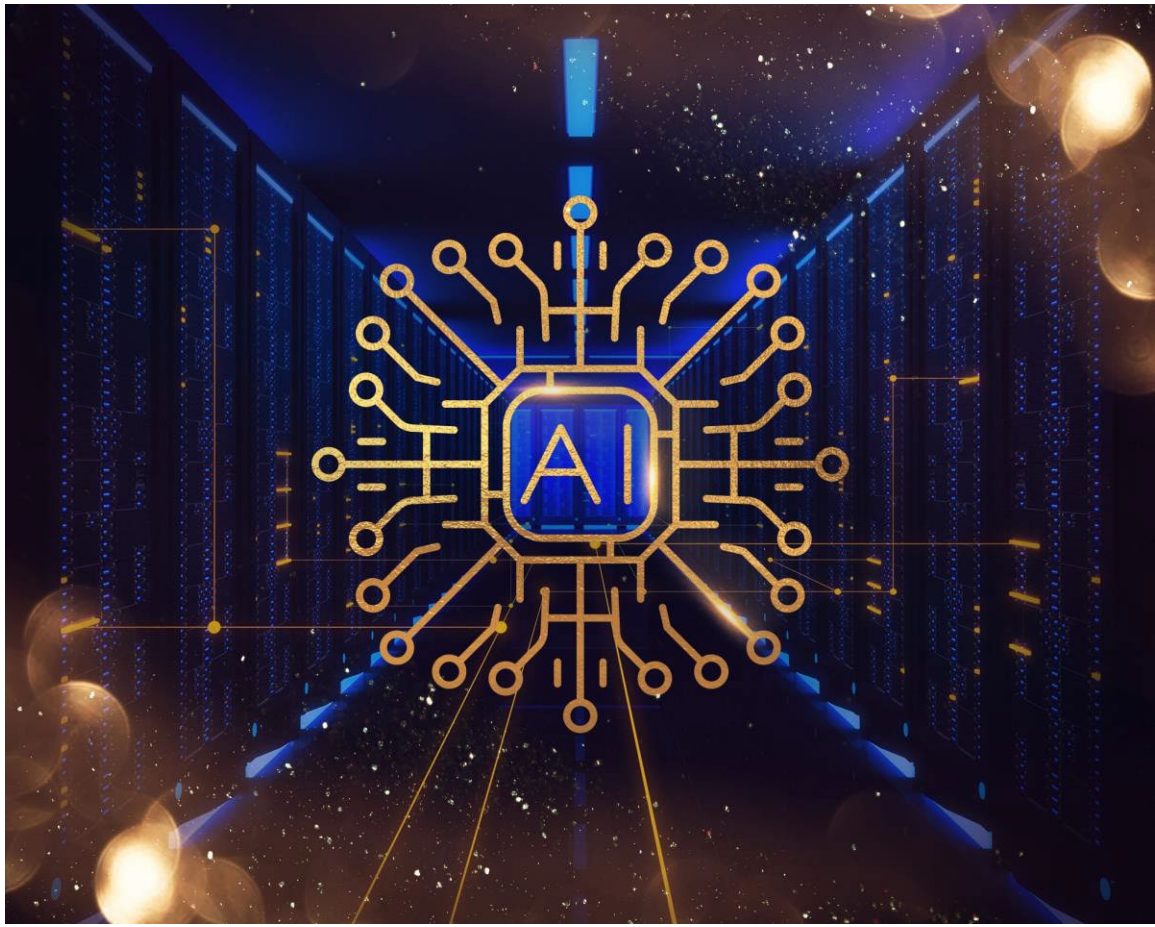
Overview

- Deep Learning
- Basics of CNN
- Transfer Learning
- R CNN
- Examples of deep learning with keras and tensorflow



Required Reading

- Chapter 8, 10, 12 of “Applied Machine Learning and AI for Engineers”
- Chapter 14 of “Machine Learning with Pytorch and Scikit-Learn”



At the end of this you should be able to

- Understand what is Deep Learning
- Understand what is CNN and relevant functionalities
- Understand how to create CNN, train it and use it for image classification
- Understand how to use transfer learning using RestNet
- Understand how to do object recognition using Mask RCNN

Deep Learning

Deep learning use cases

- Computer Vision
- Speech Recognition
- Image Processing
- Bioinformatics
- Social Network Filtering
- Drug Design
- Recommendation Systems
- Bioinformatics
- Mobile Advertising
- Many Others

Image Classification

CNN (Convolutional Nueral Network)

It is a class of deep learning, is one of the main ML model to do images recognition, images classifications, objects detections, recognising faces etc.

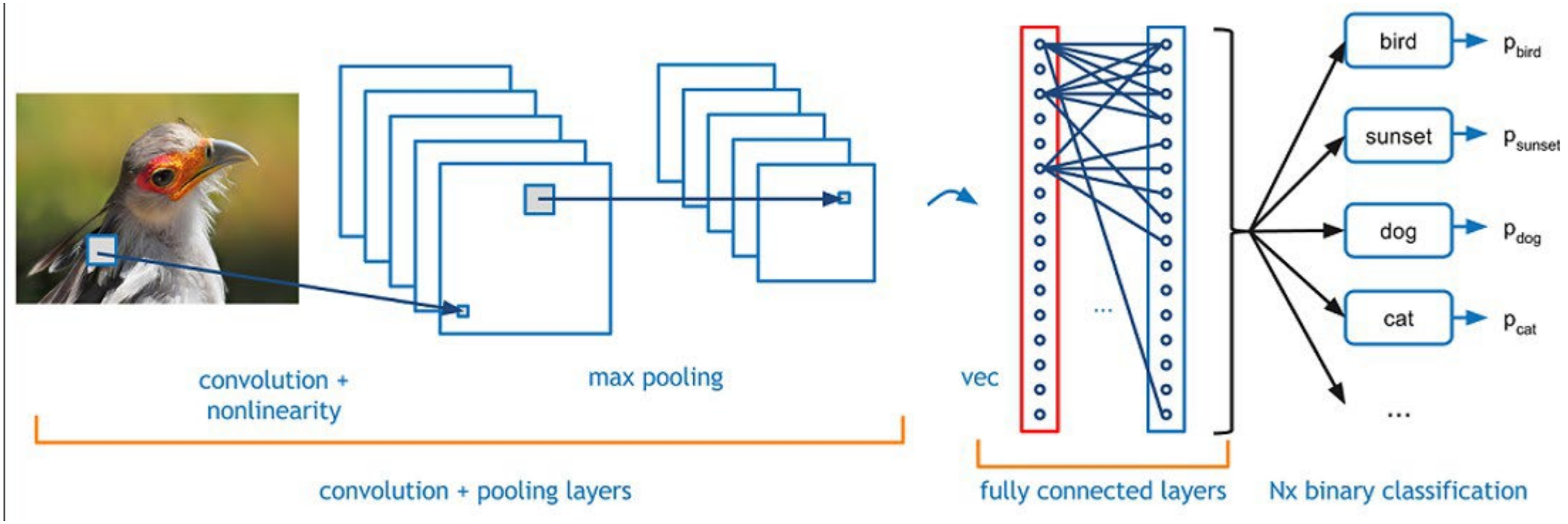
It is similar to the basic neural network.

CNN have learnable parameter like neural network i.e., weights, biases etc.

The three things to define CNN are:

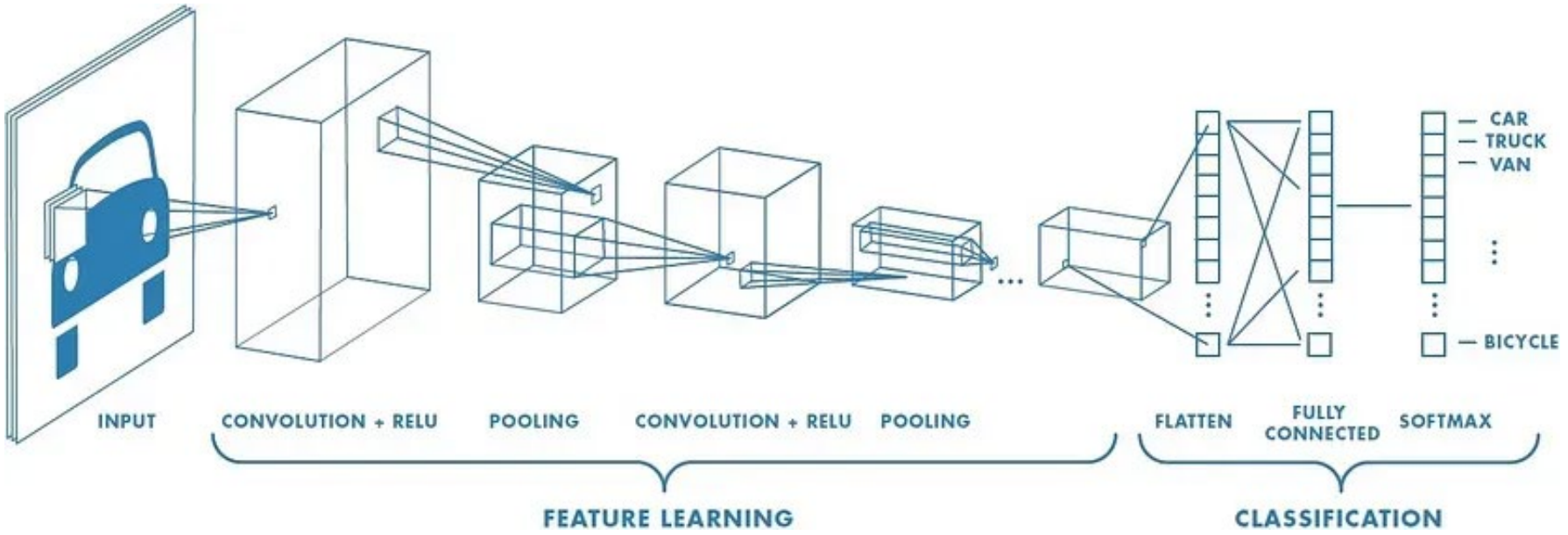
1. The Convolution Layer
2. The Pooling Layer
3. The Output Layer (or Fully Connected Layer

CNN Architecture



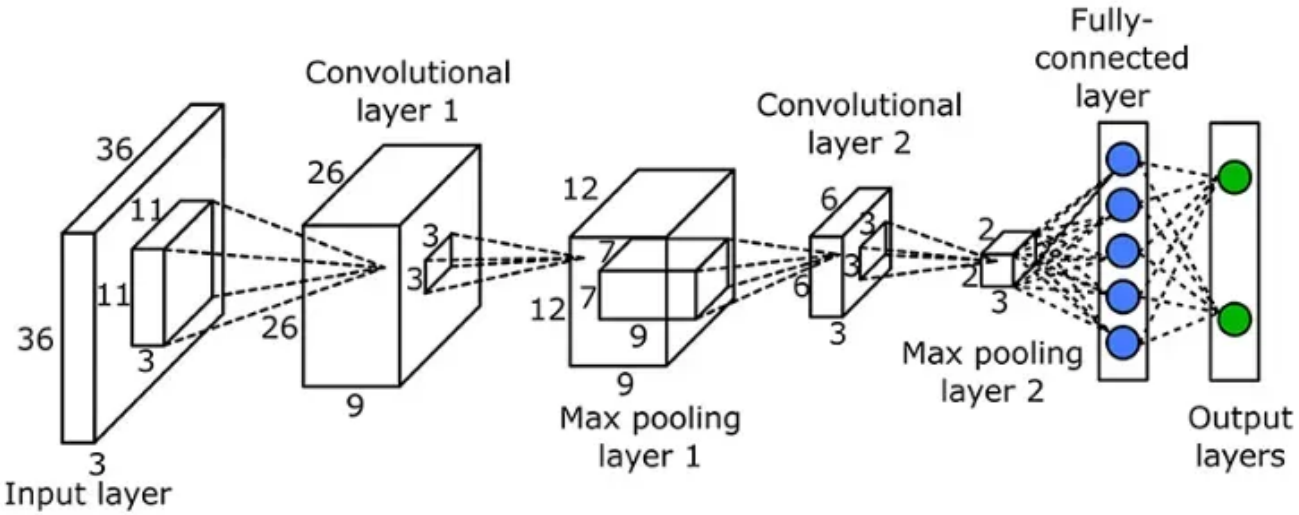
Layers

- CNNs apply filters (small rectangles) to an input image to detect features like edges or shapes. The filters slide over the width and height of the input image, computing dot products between the filter and the input to produce an activation map.
- Activation maps are fed into pooling layers, which downsample the maps to reduce their dimensionality. This makes the model more efficient and robust. The final layer is a fully connected layer that classifies the input image into categories like “dog” or “cat”.

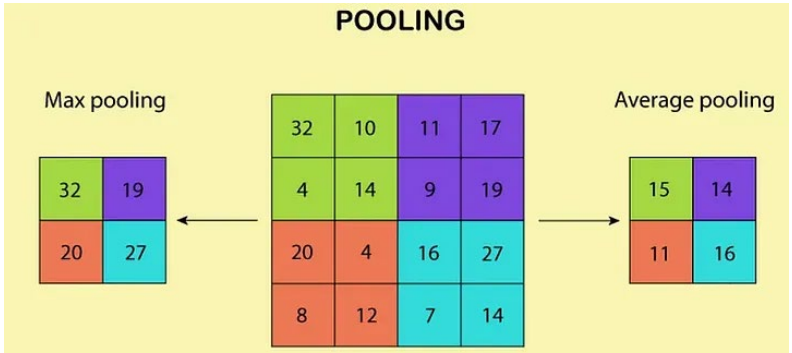


First Layers

- Convolution - Convolutional layers apply a convolution operation to the input, passing a filter over the entire image. This filter detects features like edges or curves in the picture. Multiple filters can detect different features.

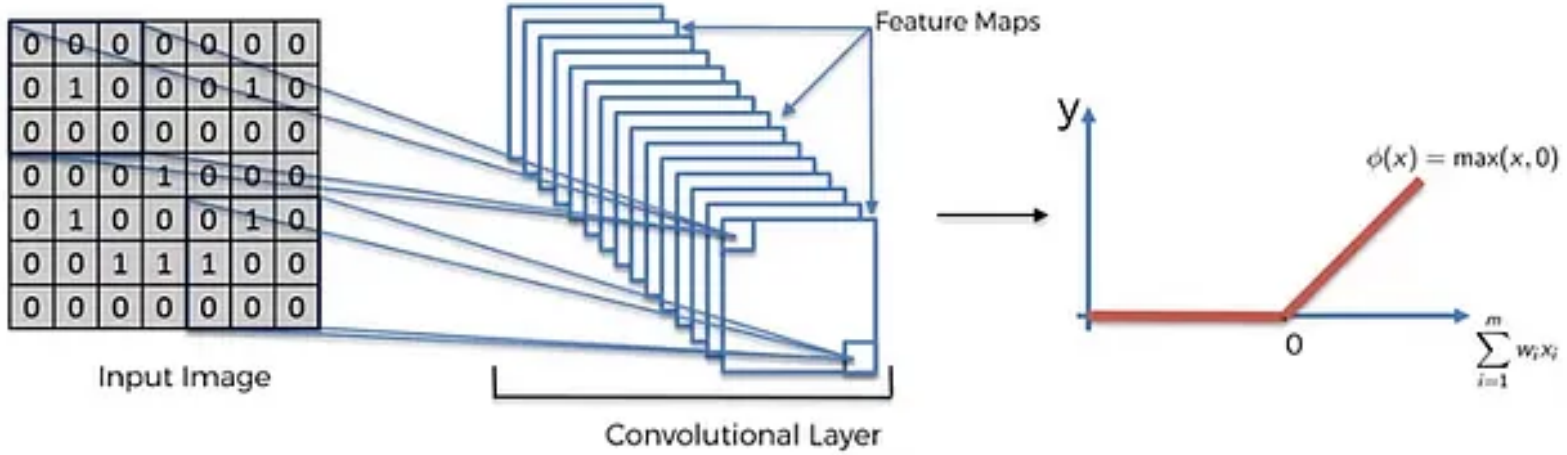


- Pooling - If images are large in size, we need to reduce the number of trainable parameters. For this, we need to use pooling layers between convolution layers. Pooling layers are inserted between convolutional layers. They downsample the feature maps to reduce the number of parameters, control overfitting, and make the network invariant to small translations. There are three types: Max Pooling, Average Pooling, and Sum Pooling.

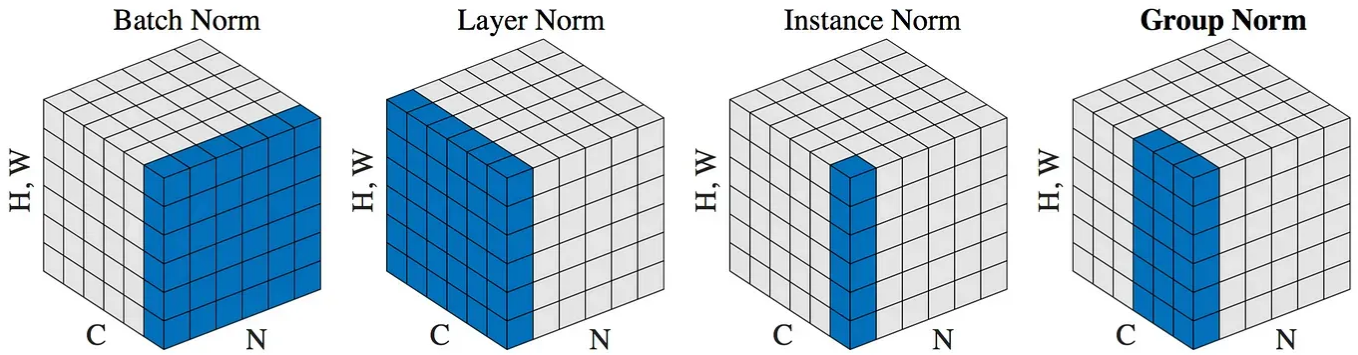


Sub Layers

- Activation - The activation layer applies a non-linear activation function, such as the ReLU function, to the output of the pooling layer. This function helps to introduce non-linearity into the model, allowing it to learn more complex representations of the input data.

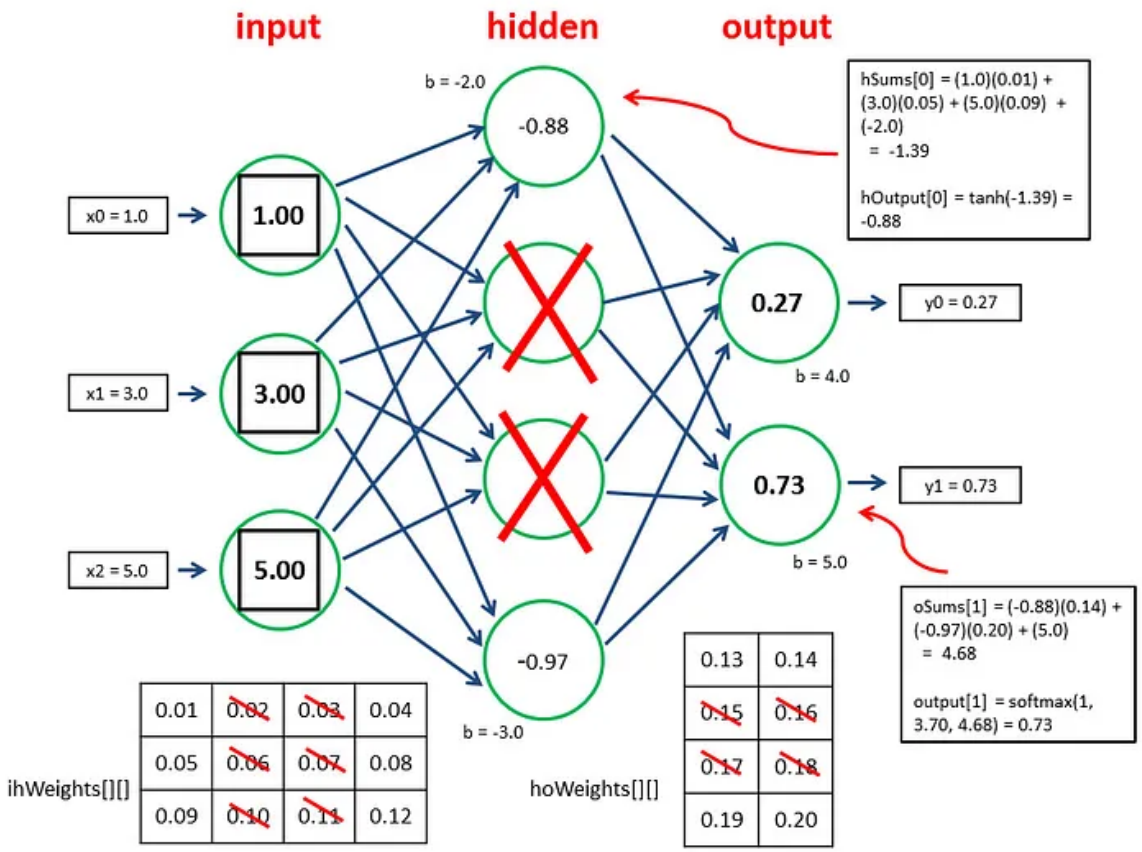


- Normalization- The normalization layer performs normalisation operations, such as batch normalisation or layer normalisation, to ensure that the activations of each layer are well-conditioned and prevent overfitting.



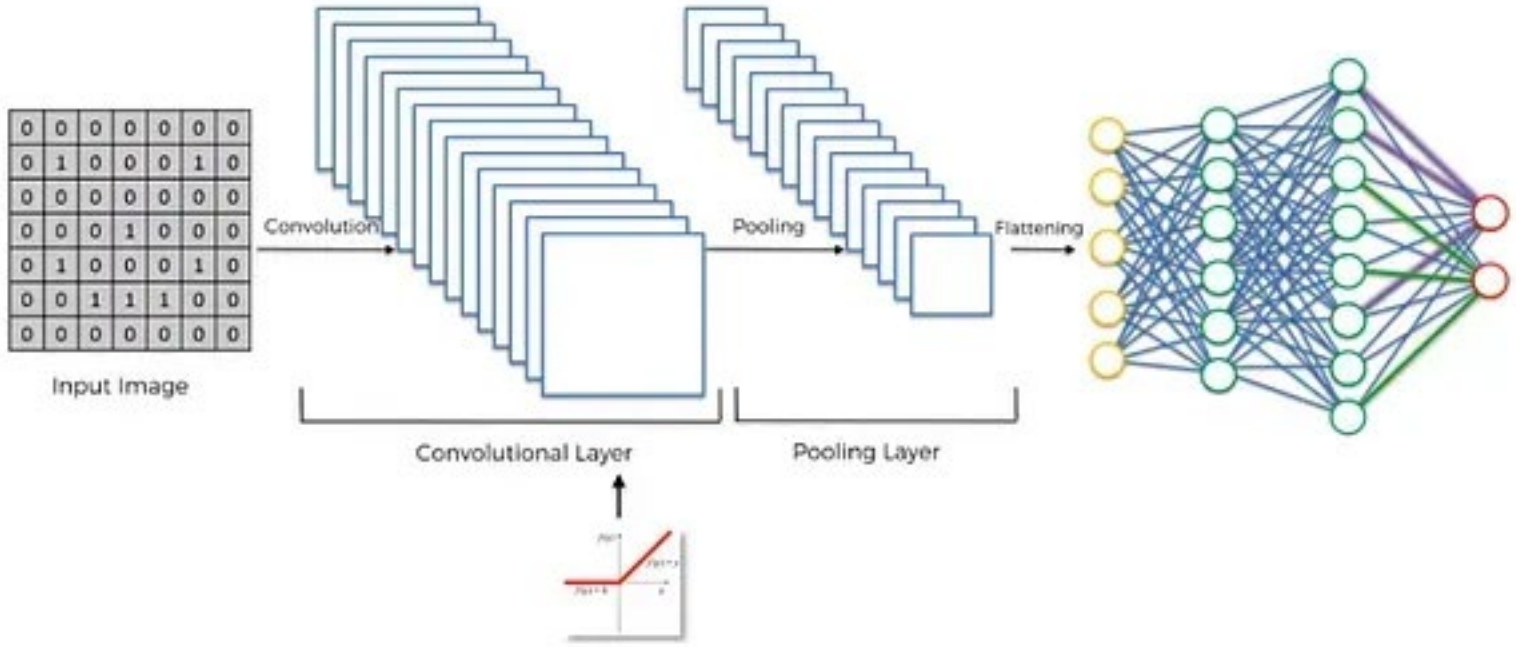
Sub Layers

- Dropout - The dropout layer is used to prevent overfitting by randomly dropping out neurons during training. This helps to ensure that the model does not memorise the training data but instead generalises to new, unseen data.



Final Layer

- Output- After the convolutional and pooling layers have extracted features from the input image, the dense layer can then be used to combine those features and make a final prediction. In CNNs, the dense layer is usually the final layer and is used to produce the output predictions. The activations from the previous layers are flattened and passed as inputs to the dense layer, which performs a weighted sum of the inputs and applies an activation function to produce the final output.



Number of Layers

- Deeper networks is always better, at the cost of more data and increased complexity of learning.
- You should initially use fewer filters and gradually increase and monitor the error rate to see how it is varying.
- Very small filter sizes will capture very fine details of the image. On the other hand, having a bigger filter size will leave out minute details in the image.

Types of CNN

The five major areas which can be addressed using CNN.

- Image Classification
- Object Detection
- Object Tracking
- Semantic Segmentation
- Instance Segmentation

CNN Types: Image Classification

In an image classification we can use the traditional CNN models or there also many architectures designed by developers to decrease the error rate and increasing the trainable parameters.

- LeNet (1998)
- AlexNet (2012)
- ZFNet (2013)
- GoogLeNet19 (2014)
- VGGNet 16 (2014)
- ResNet(2015)

CNN Types: Object Detection

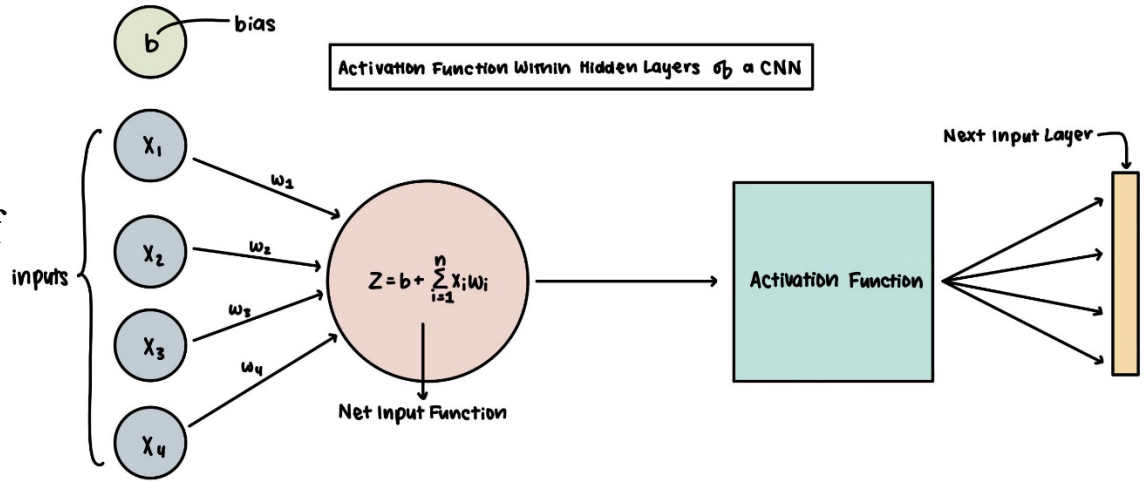
Object Detection:

- Here the implementation of CNN is different compared to the previous image classification.
- Here the task is to identify the objects present in the image. Therefore, traditional implementation of CNN may not help.
  - R CNN
  - Fast R CNN
  - Faster R CNN
  - Mask R CNN
  - YOLO

Activation Functions

- An activation function is a function that takes the input of a neuron and produces an output that is used as the input for the next layer of neurons in a neural network. Activation functions are essential because they introduce nonlinearity into the neural network, allowing it to learn complex patterns and perform nonlinear computations.
- Without activation functions, neural networks would be equivalent to a single-layer linear model, which can only learn linear relationships between the input and output. Activation functions enable neural networks to learn nonlinear mappings.

1. Softmax: Outputs a probability distribution over multiple classes, commonly used in the output layer for multi-class classification.
2. Sigmoid: The sigmoid function maps input values to the range of (0, 1), making it suitable for binary classification tasks. However, it suffers from the vanishing gradient problem, which can slow down training.

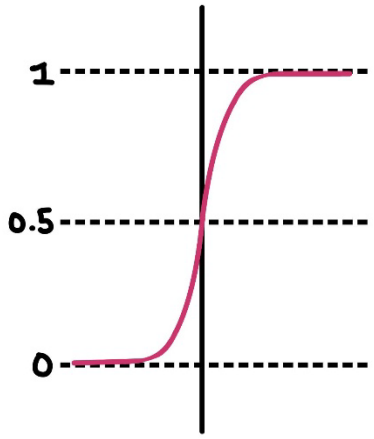


Softmax Function

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

$K \rightarrow$ number of classes

$j \rightarrow$ indicates variable output neuron

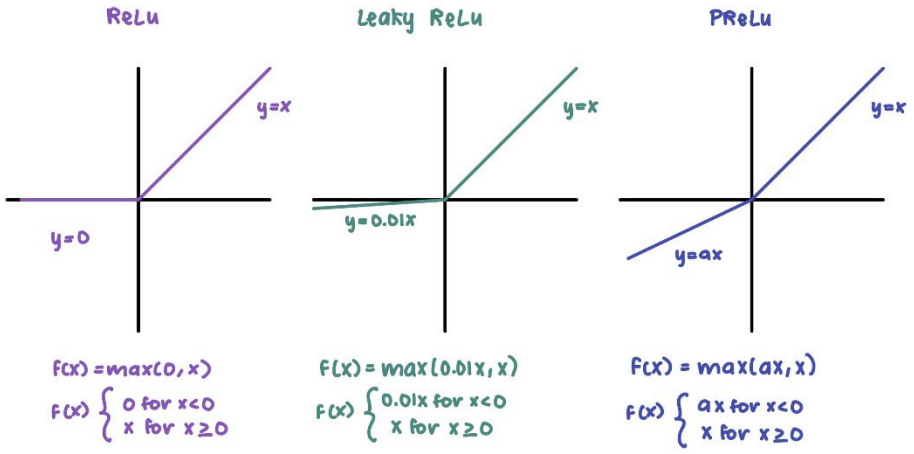


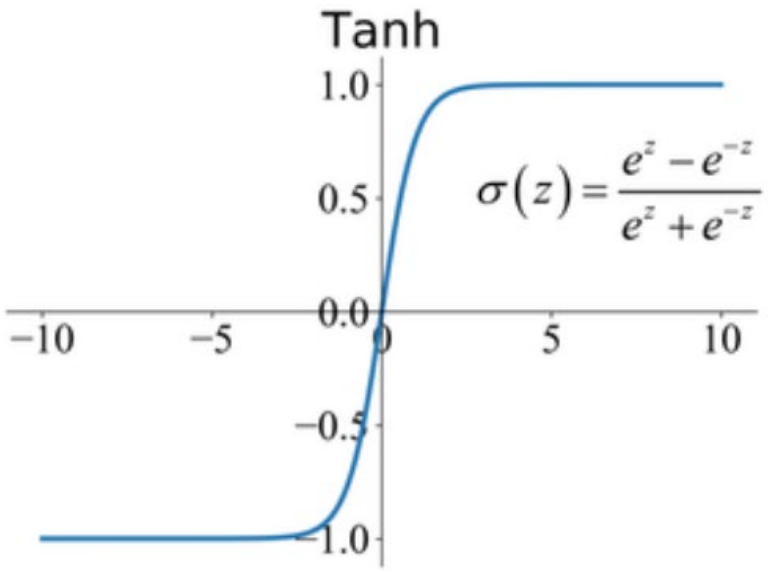
Sigmoid Function

$$z = \sum w_i x_i + b$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Activation Functions





1. ReLU (Rectified Linear Unit): The ReLU is one of the most widely used activation functions. It is computationally efficient and overcomes the vanishing gradient problem. However, it is susceptible to the dying ReLU problem, where neurons can become inactive during training.
2. Leaky ReLU: Leaky ReLU addresses the dying ReLU problem by allowing a slight gradient for negative input values, thereby preventing neurons from becoming entirely inactive.
3. prelu or parametric ReLU: Instead of having a fixed coefficient like 0.01 in Leaky ReLU, this coefficient is learned during training.
4. tanh (Hyperbolic Tangent): The tanh function s similar to the sigmoid, but it applies input values to the range of (-1, 1), providing better symmetry around zero. It also suffers from the vanishing gradient problem.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Activation function in Python

```
import numpy as np

Python sigmoid example:
z = 1/(1 + np.exp(-np.dot(W, x)))

Python tanh example:
z = np.tanh(np.dot(W,x));

Python ReLU example:
z = np.maximum(0, np.dot(W, x))
```

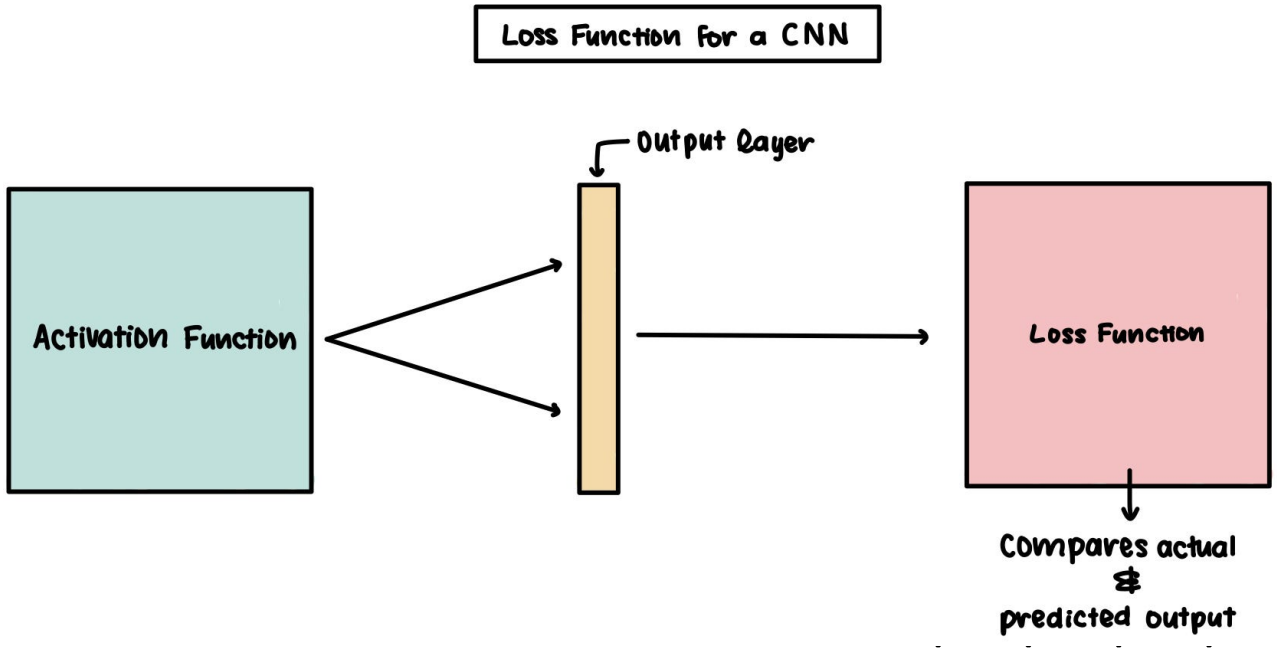
- **W** =(shape: output_neurons × input_features), is typically a weight matrix (for a neural network layer)
- **x** = Input vector (shape: input_features × 1) or matrix (shape: input_features × batch_size), is an input vector (or matrix, if processing multiple inputs at once).
- The activation functions **(sigmoid, tanh, ReLU)** introduce non-linearity after the linear transformation (Wx).

"Best" Activation Function

- Initially, sigmoid was popular, then tanh became popular
- Now: RELU is preferred (better results)
- Softmax: for FC (fully connected) layers
- Sigmoid and tanh are used in LSTMs

Loss Function

Loss functions are synonymous with "cost functions" as they calculate the function's loss to determine its viability.
For example, let's say a Convolutional Neural Network predicts that an image has a 30% chance of being a cat, a 10% chance of being a frog, and a 60% chance of being a horse. If the actual label for the image were a cat, then there would be a very high loss.



Mean Absolute Error

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

n → number of samples; y_j → actual value; $\hat{y}_j$ → predicted value

Mean Squared Error

$$MSE = \frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2$$

MSE imposes a more significant penalty compared to MAE as the difference between the predicted and actual values increases.

Binary Cross Entropy

$$BCE = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(p_i) + (1-y_i)\log(1-p_i)$$

p_i → predicted probability; y_i → actual value (0 or 1)

if $y_i = 0 \rightarrow BCE = -\frac{1}{n}\sum_{i=1}^{n}(1-y_i)\log(1-p_i)$

if $y_i = 1 \rightarrow BCE = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(p_i)$

Log(x) function, so values with higher probabilities will have a lower loss relative to 1. This approach would be helpful for binary classification problems.

Cross-Entropy

$$CE = -\frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c} y_i \log(\hat{y}_i)$$

- i = class number
- c = number of classes
- y_i = actual value (0 or 1)
- $\hat{y}_i$ = predicted value (probability)

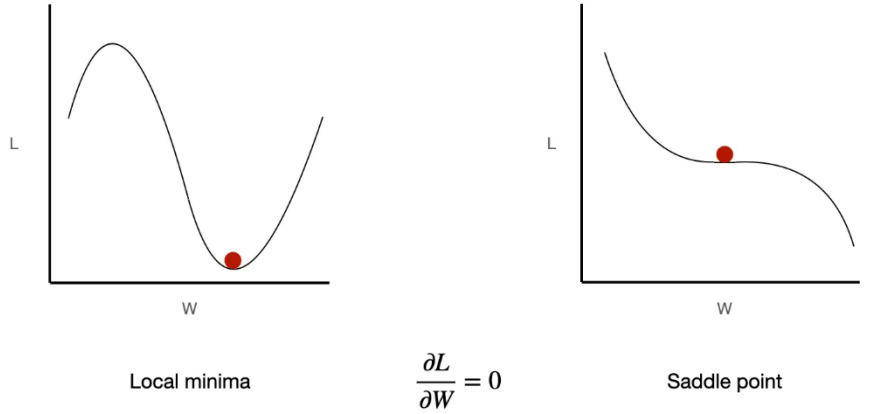
CE generalises this function to networks with multi-class outputs.

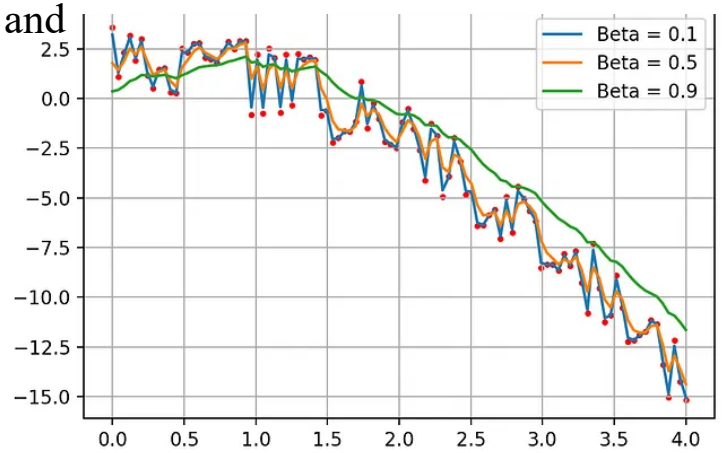
SWINBURNE UNIVERSITY OF TECHNOLOGY

Optimisers

The ultimate goal of the ML model is to minimise the loss function. After we pass the input, we calculate the error and update the weights accordingly. This is where the optimiser comes into play. It defines how to adjust the parameters to approach the minimum.

- SGD was the most popular and first figure on the right.
- Exponential Moving Average (EMA) - $e_t = \beta e_{t-1} + (1-\beta)x_t$, below figure
- RMSProp - $g_t = \beta g_{t-1} + (1-\beta)dW^2$ $W = W - \frac{\alpha}{\sqrt{g_t + \epsilon}} * dW$
- Momentum - $v_t = \beta v_{t-1} + (1-\beta)dW$ $W = W - \alpha * v_t$ alpha is the learning rate.
- AdaGrad - $g_t = g_{t-1} + dW^2$ $W = W - \frac{\alpha}{\sqrt{g_t + \epsilon}} * dW$, if the learning rate is too high for a large gradient, we overshoot and bounce around. If the learning rate is too low, the learning is slow and might never converge.
- Adam- combination of Momentum and RMSProp





https://medium.com/nerd-for-tech/optimizers-in-machine-learning-f1a9c549f8b4

SWINBURNE UNIVERSITY OF TECHNOLOGY

Deep Learning Summary

- input layer, multiple hidden layers, and output layer
- nonlinear processing via activation functions
- perform transformation and feature extraction
- gradient descent algorithm with back propagation
- Each layer receives the output from the previous layer
- Results are comparable/superior to human experts

Keras and Tensorflow

CNN in Keras

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Conv2D,
MaxPooling2D
from keras.optimizers import Adadelta

# Define input shape
input_shape = (32, 32, 3)
nb_classes = 10

# Initialize model
model = Sequential()

# First convolutional layer
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=input_shape))
model.add(Activation('relu'))

# Second convolutional layer
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))

# MaxPooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))

# Dropout layer
model.add(Dropout(0.25))

# Print model summary
model.summary()
```

Tensorflow

- An open-source framework for ML and DL
- Created by Google (released 11/2015)
- Evolved from Google Brain
- Visualization via TensorBoard
- TF tensors are n-dimensional arrays
- TF tensors are very similar to numpy ndarrays

CNN: Training in Keras

```
rom keras.preprocessing.image import
ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout,
Flatten, Dense, GlobalAveragePooling2D
from keras import backend as K
from keras.callbacks import CSVLogger

# Dimensions of our images
img_width, img_height = 330, 247

# Paths to training and validation data
train_data_dir = '../images/data/train'
validation_data_dir = '../images/data/validation'

# Training parameters
epochs = 100
batch_size = 64

# Define input shape based on image data format
if K.image_data_format() == 'channels_first':
    input_shape = (3, img_width, img_height)
else:
    input_shape = (img_width, img_height, 3)

# Build the CNN model
model = Sequential()
```

```
# First Conv Layer
model.add(Conv2D(32, (3, 3),
input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Second Conv Layer
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Third Conv Layer
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Fully Connected Layers
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])

# Print model summary
model.summary()
```

CNN: Training in Keras

```
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import CSVLogger

# Data Augmentation for training
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Only rescaling for validation data
validation_datagen = ImageDataGenerator(rescale=1.0 / 255)

# Load training images
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    shuffle=True,
    class_mode='binary'
)

# Load validation images
validation_generator =
validation_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    shuffle=True,
    class_mode='binary'
)
```

```
# Train the model
history = model.fit(
    train_generator,
    epochs=epochs,
    steps_per_epoch=len(train_generator),
    validation_data=validation_generator,
    validation_steps=len(validation_generator),
    callbacks=[CSVLogger("training.log", append=False,
separator=";")]
)

# Saving model architecture
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)

# Save model weights
model.save_weights('model_weights.h5')
model.save("model.h5")

# Extract training accuracy
training_accuracy = history.history['accuracy']
validation_accuracy = history.history['val_accuracy']
```

Transfer Learning: Training

A technique in deep learning where a pre-trained model, trained on a large dataset such as ImageNet, is fine-tuned on a smaller dataset. This helps achieve high accuracy even with limited data.

```
import keras
from keras.layers import Dense, GlobalAveragePooling2D
from keras.utils import multi_gpu_model
from keras.applications.resnet50 import ResNet50,
preprocess_input
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model
from keras import backend as K

# Load ResNet50 model without the top layers
base_model = ResNet50(weights=None, include_top=False)

# Load pretrained weights
base_model.load_weights('resnet50_weights_tf_dim_ordering_tf
_kernels_notop.h5')

# Build custom classification layers
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# Output layer (binary classification)
preds = Dense(1, activation='sigmoid')(x)
```

```
# Create model
model = Model(inputs=base_model.input, outputs=preds)

# Freeze base model layers to prevent training
for layer in base_model.layers:
    layer.trainable = False

# Enable multi-GPU training
parallel_model = multi_gpu_model(model, gpus=2)

# Compile the model
parallel_model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Classification

```
from keras.preprocessing.image import ImageDataGenerator

# Create a test data generator
test_data_generator = ImageDataGenerator(rescale=1.0 / 255)

# Load test images
test_generator = test_data_generator.flow_from_directory(
    dir_name,
    target_size=(img_width, img_height),
    batch_size=1,
    class_mode=None,  # No labels since this is a test set
    shuffle=False
)

# Predict using the trained model
probabilities = model.predict(test_generator,
steps=len(test_generator))

# Print predictions
print(probabilities)
```

Mask RCNN

- Mask R-CNN is framework to solve the instance segmentation problem.
- Instance segmentation is a task of detecting and delineating each object in an image in a fine-grained pixel level
- Instance segmentation can estimate object position given an image, so tasks such as robot manipulation can perform grasp planning
- It is an extension of the Faster R-CNN framework.

Learn, Practice and Enjoy the AI journey