CNN

February 20, 2025

[ ]:
```
'''
    Convolutional Neural Networks -->

    Introduction -->

    Convolutional Neural Networks (CNNs) are specialized deep learning
    models designed to process grid-like data, particularly images.
    They mimic the way the human brain processes visual information,
    recognizing patterns, edges, textures, and hierarchical structures
    automatically.
'''
```

[ ]:
```
'''
    Intuition Behind CNNs -->

    Why Not Traditional Neural Networks ?

    Traditional fully connected neural networks (FCNNs) struggle with
    high-dimensional image data due to:

    Parameter Explosion: Fully connected layers require enormous numbers
    of weights when handling high-resolution images.
    Spatial Redundancy: Pixels in an image have local dependencies;
    FCNNs fail to leverage these relationships efficiently.
    Translation Variance: A simple shift in an image can affect FCNN
    predictions drastically.

    CNNs address these issues by:

    Using local receptive fields instead of fully connected layers.
    Sharing weights across the image using kernels.
    Applying downsampling (pooling) to reduce dimensions while preserving
    essential features.
'''
```

[ ]:
```
'''
    Core Components of CNNs -->
```

```
    1. Convolutional Layer

    The core idea of CNNs lies in convolution operations, which apply a
    small filter (kernel) over the input image to extract essential
    features such as edges, corners, and textures.

    Filters (Kernels): Small matrices that slide over the image.
    Stride: Defines the step size for moving the filter.
    Padding: Controls how borders of the image are handled
    (zero-padding or valid-padding).
    Feature Maps: The output of convolution layers after applying
    activation functions.

    2. Activation Function (ReLU)

    ReLU introduces non-linearity, allowing CNNs to learn complex patterns.
    ReLu(x) = max(0,x)

    3. Pooling Layer

    Pooling reduces spatial dimensions while retaining significant
    information.

    Max Pooling: Retains the highest value in a region.
    Average Pooling: Takes the average of pixel values in a region.

    Pooling helps with:

    Reducing computation.
    Enhancing feature robustness.
    Mitigating overfitting.

    4. Fully Connected Layer

    After extracting meaningful features, CNNs flatten the feature maps
    and pass them through dense layers to make predictions.

    5. Softmax (Output Layer)

    Converts logits into probability distributions for classification.
'''
```

[ ]:

```
'''
    Applications of CNNs -->

    Image Classification (e.g., CIFAR-10, ImageNet)
    Object Detection (e.g., YOLO, SSD)
```

```
    Facial Recognition
    Medical Image Analysis
    Autonomous Vehicles
'''
```

[39]:
```
#   Importing Libraries -->

import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten,␣
 ↪MaxPooling2D, Dropout

print("Libraries Imported !")
```

```
Libraries Imported !
```

[40]:
```
#   Image Augmentation [Training Set] -->

train_datagen = ImageDataGenerator(
    rescale = 1./255,
    shear_range = 0.2,
    zoom_range = 0.2,
    horizontal_flip = True
)

training_set = train_datagen.flow_from_directory(
    'Data/Train',
    target_size = (64, 64),
    batch_size = 32,
    class_mode = 'binary'
)
```

```
Found 8000 images belonging to 2 classes.
```

[41]:
```
#   Image Augmentation [Test Set] -->

test_datagen = ImageDataGenerator(
    rescale = 1./255
)

test_set = test_datagen.flow_from_directory(
    'Data/Test',
    target_size = (64, 64),
    batch_size = 32,
    class_mode = 'binary'
```

```
)
```

Found 2000 images belonging to 2 classes.

[42]:
```python
#   Building The CNN -->

model = Sequential([
    Input(shape=(64,64,3)),
    Conv2D(filters=32, kernel_size=3, activation='relu'),
    MaxPooling2D(pool_size=2, strides=2),
    Flatten(),
    Dense(units=128, activation='relu'),
    Dense(units=1, activation='sigmoid')
])

print("Model Built !")
```

Model Built !

[43]:
```python
#   Summarizing The Model -->

model.summary()
```

Model: "sequential_2"

```
 Layer (type)                         Output Shape                        ␣
↪Param #

 conv2d_2 (Conv2D)                    (None, 62, 62, 32)                    ␣
↪896

 max_pooling2d_2 (MaxPooling2D)       (None, 31, 31, 32)                    ␣
↪  0

 flatten_2 (Flatten)                  (None, 30752)                         ␣
↪  0

 dense_4 (Dense)                      (None, 128)                       ␣
↪3,936,384

 dense_5 (Dense)                      (None, 1)                             ␣
↪129
```

Total params: 3,937,409 (15.02 MB)

Trainable params: 3,937,409 (15.02 MB)

Non-trainable params: 0 (0.00 B)

[44]:
```
#   Compiling The Model -->

model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

print("Model Compiled !")
```

```
Model Compiled !
```

[46]:
```
#   Training The Model -->

model.fit(
    x = training_set,
    validation_data = test_set,
    epochs = 25
)
```

```
Epoch 1/25
250/250              147s 588ms/step -
accuracy: 0.5650 - loss: 0.7286 - val_accuracy: 0.6850 - val_loss: 0.6020
Epoch 2/25
250/250              59s 234ms/step -
accuracy: 0.6760 - loss: 0.6050 - val_accuracy: 0.6840 - val_loss: 0.5908
Epoch 3/25
250/250              57s 227ms/step -
accuracy: 0.7007 - loss: 0.5746 - val_accuracy: 0.7210 - val_loss: 0.5473
Epoch 4/25
250/250              81s 225ms/step -
accuracy: 0.7228 - loss: 0.5492 - val_accuracy: 0.7090 - val_loss: 0.5640
Epoch 5/25
250/250              83s 227ms/step -
accuracy: 0.7326 - loss: 0.5304 - val_accuracy: 0.7125 - val_loss: 0.5655
Epoch 6/25
250/250              57s 227ms/step -
accuracy: 0.7414 - loss: 0.5197 - val_accuracy: 0.6980 - val_loss: 0.6099
Epoch 7/25
250/250              61s 245ms/step -
accuracy: 0.7482 - loss: 0.5013 - val_accuracy: 0.7430 - val_loss: 0.5394
Epoch 8/25
250/250              78s 228ms/step -
accuracy: 0.7658 - loss: 0.4814 - val_accuracy: 0.7605 - val_loss: 0.5189
Epoch 9/25
250/250              61s 244ms/step -
```

```
accuracy: 0.7706 - loss: 0.4734 - val_accuracy: 0.7365 - val_loss: 0.5420
Epoch 10/25
250/250            56s 225ms/step -
accuracy: 0.7676 - loss: 0.4699 - val_accuracy: 0.7635 - val_loss: 0.5221
Epoch 11/25
250/250            82s 225ms/step -
accuracy: 0.7865 - loss: 0.4480 - val_accuracy: 0.7610 - val_loss: 0.5144
Epoch 12/25
250/250            83s 230ms/step -
accuracy: 0.7832 - loss: 0.4500 - val_accuracy: 0.7710 - val_loss: 0.5115
Epoch 13/25
250/250            82s 229ms/step -
accuracy: 0.7959 - loss: 0.4357 - val_accuracy: 0.7720 - val_loss: 0.5070
Epoch 14/25
250/250            81s 226ms/step -
accuracy: 0.7976 - loss: 0.4240 - val_accuracy: 0.7710 - val_loss: 0.5227
Epoch 15/25
250/250            81s 223ms/step -
accuracy: 0.8122 - loss: 0.4058 - val_accuracy: 0.7750 - val_loss: 0.5124
Epoch 16/25
250/250            58s 230ms/step -
accuracy: 0.8294 - loss: 0.3862 - val_accuracy: 0.7520 - val_loss: 0.5687
Epoch 17/25
250/250            59s 235ms/step -
accuracy: 0.8313 - loss: 0.3797 - val_accuracy: 0.7500 - val_loss: 0.5987
Epoch 18/25
250/250            62s 248ms/step -
accuracy: 0.8371 - loss: 0.3684 - val_accuracy: 0.7720 - val_loss: 0.5321
Epoch 19/25
250/250            61s 244ms/step -
accuracy: 0.8525 - loss: 0.3290 - val_accuracy: 0.7545 - val_loss: 0.5417
Epoch 20/25
250/250            61s 243ms/step -
accuracy: 0.8589 - loss: 0.3265 - val_accuracy: 0.7630 - val_loss: 0.5772
Epoch 21/25
250/250            57s 228ms/step -
accuracy: 0.8538 - loss: 0.3291 - val_accuracy: 0.7745 - val_loss: 0.5574
Epoch 22/25
250/250            58s 233ms/step -
accuracy: 0.8527 - loss: 0.3285 - val_accuracy: 0.7565 - val_loss: 0.6065
Epoch 23/25
250/250            80s 227ms/step -
accuracy: 0.8745 - loss: 0.2939 - val_accuracy: 0.7560 - val_loss: 0.6208
Epoch 24/25
250/250            82s 228ms/step -
accuracy: 0.8656 - loss: 0.3037 - val_accuracy: 0.7530 - val_loss: 0.6938
Epoch 25/25
250/250            57s 229ms/step -
```

```
accuracy: 0.8811 - loss: 0.2796 - val_accuracy: 0.7630 - val_loss: 0.6182
```

[46]:
```
<keras.src.callbacks.history.History at 0x2c0ab61dc60>
```

[51]:
```python
#   Accuracy -->

test_loss, test_acc = model.evaluate(test_set)
print(f"Test Accuracy: {test_acc * 100:.2f}%")
```

```
63/63              6s 99ms/step -
accuracy: 0.7739 - loss: 0.6049
Test Accuracy: 76.30%
```

[50]:
```python
#   Predicting Results For Single Image -->

test_image = image.load_img('Data/Sample/cat_or_dog_1.jpg', target_size=(64,64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis=0)

result = model.predict(test_image)
training_set.class_indices

if (result[0][0] == 1) :
    print("It is a Dog !")
else :
    print("It is a Cat !")
```

```
1/1              0s 140ms/step
It is a Dog !
```

[52]:
```python
#   Predicting Results For Test Set -->

true_labels = test_set.classes

predictions = model.predict(test_set)  # This returns probabilities
predicted_labels = np.argmax(predictions, axis=1)  # Convert to class indices

correct_predictions = np.sum(predicted_labels == true_labels)
total_images = len(true_labels)

accuracy = (correct_predictions / total_images) * 100

print(f"Correct Predictions: {correct_predictions}/{total_images}")
print(f"\nTest Accuracy: {accuracy:.2f}%")
```

```
63/63              6s 89ms/step
Correct Predictions: 1000/2000

Test Accuracy: 50.00%
```