Convolutional Neural Network (CNN)

You will Learn the following

1. Loading dataset
2. Preprocessing the dataset
3. Structure of Convolutional Neural Network
4. Training a Convolutional Neural Network
5. Testing a Convolutional Neural Network

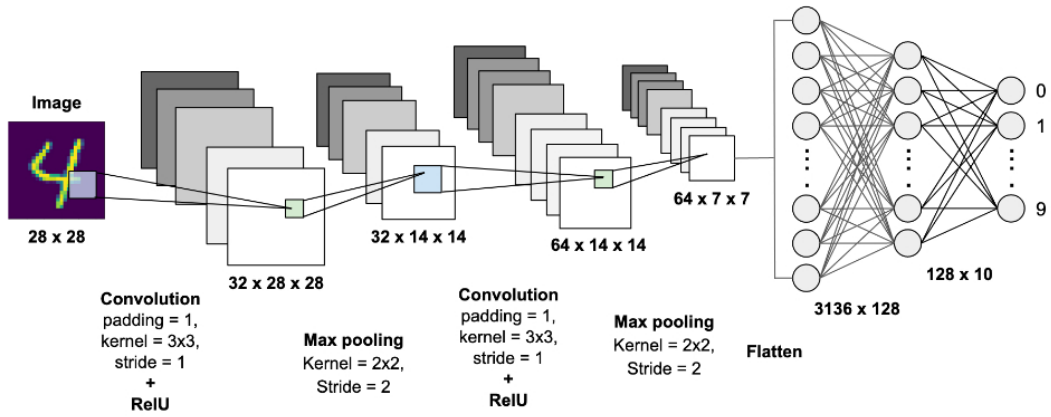
Introduction to Convolutional Networks

A convolutional neural network is a feed-forward neural network that is generally used to analyze visual images by processing data with grid-like topology. It's also known as a ConvNet. A convolutional neural network is used to detect and classify objects in an image.

In the world of computer vision, the most basic and common image recognition algorithm is the convolutional network. With the popularity of frameworks such as TensorFlow and PyTorch, it has become easier to use convolutional networks, allowing us to focus on the forward pass instead of implementing the backward pass.

This notebook assumes that we are all familiar with convolutional networks, so we won't go into much detail here. Instead, I'll just use a few GIFs to refresh your memory on the implementation of convolutional networks as we know them.

Digit Classification using Convolutional Neural Networks



1. Loading Required Packages and Dataset

Before diving into the implementation, let's start by loading the necessary packages and the dataset. This step is crucial for setting up the environment and preparing the data for our convolutional network.

Below, we'll demonstrate the code to load the required packages and fetch the dataset using popular libraries like TensorFlow.

In [12]:
```
# Import required Packages
import numpy as np
import pandas as pd
import keras
import tensorflow as tf
```

In [13]:
```
# Checking Available Devices
import tensorflow as tf
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 7175795634265292176
xla_global_id: -1
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 14626652160
locality {
  bus_id: 1
  links {
  }
}
incarnation: 14244440542709068578
physical_device_desc: "device: 0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5"
xla_global_id: 416903419
]
```

2. Loading and Data Splitting

Now that we've loaded the necessary packages and dataset, the next step is to split the data into training, validation, and testing sets. Proper data splitting is crucial for training and evaluating the performance of our convolutional network.

In [14]:
```python
# Loading Dataset
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

3. Displaying Sample Images

In this section, we visualize a subset of the test dataset by displaying the first 'n' images. This provides a glimpse into the data that the model will be working with.

In [15]:
```python
import matplotlib.pyplot as plt

# Number of digits to display
n = 10

# Create a figure to display the images
plt.figure(figsize=(20, 4))

# Loop through the first 'n' images
for i in range(n):
    # Create a subplot within the figure
    ax = plt.subplot(2, n, i + 1)

    # Display the original image
    plt.imshow(x_test[i].reshape(28, 28))

    # Set colormap to grayscale
    plt.gray()

    # Hide x-axis and y-axis labels and ticks
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

# Show the figure with the images
plt.show()

# Close the figure
plt.close()

```

4. Displaying the Shapes of the Dataset

Now, let's take a quick look at the shapes of the dataset to get a better understanding of its structure. Examining the dimensions of the data is essential for configuring the input layer of our convolutional network.

We'll explore the number of samples, features, and labels available in the training, validation, and testing sets.

In [16]:
```
#Dataset Shapes
print("x_train Shape :", x_train.shape)
print("y_train Shape :", y_train.shape)
print("x_test Shape :", x_test.shape)
print("y_test Shape :", y_test.shape)
```

```
x_train Shape : (60000, 28, 28)
y_train Shape : (60000,)
x_test Shape : (10000, 28, 28)
y_test Shape : (10000,)
```

5. Reshaping the Data

In this step, we reshape the data to ensure it is in the appropriate format for consumption by the TensorFlow backend. Reshaping is crucial to match the expected input shape of our convolutional network.

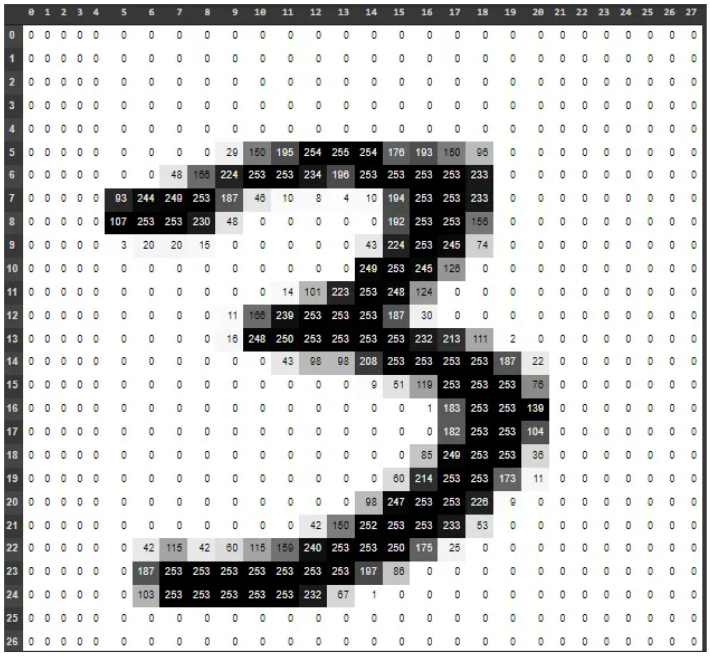
For the TensorFlow backend, the common format is "Channel Last," where the data is reshaped to have dimensions [number of samples, height, width, channels]. In this case, we reshape the input data to have a single channel.

In [17]:
```
# Reshaping Data in "Channel Last" format for consumption of Tensorflow backend
x_train = x_train.reshape(x_train.shape[0],28,28,1)
x_test = x_test.reshape(x_test.shape[0],28,28,1)
```

6. Min Max Scaling

After reshaping the data, the next step is to perform Min-Max scaling. This involves converting the pixel values to floating point format and normalizing them to a range between 0 and 1. Scaling the data helps in improving the training stability and convergence of our convolutional network.

In [18]:
```
# converting to floating point and normalizing pixel values in range [0,1]
x_train = x_train.astype("float32")
x_test = x_test.astype("float32")
x_train /= 255
x_test /= 255
```



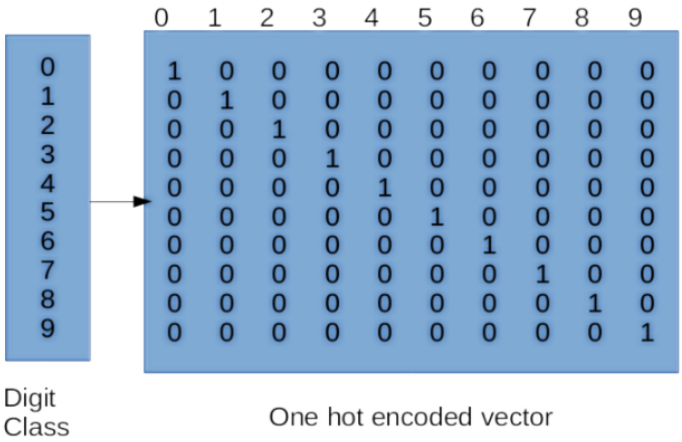
7. One-Hot Encoding

Now, we perform One-Hot encoding on the labels for multi-class classification. This step is crucial for converting categorical labels into a binary matrix format, making it suitable for training a neural network.

We utilize libraries like Keras to_categorical to perform this encoding, transforming the labels into a binary matrix representation.

In [19]:
```
# Reshaping Labels in One-hot encoding for Multi-class Classification
from keras.utils import to_categorical

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
# Seeing updated Shapes
print("x_train Shape :", x_train.shape)
print("y_train Shape :", y_train.shape)
print("x_test Shape :", x_test.shape)
print("y_test Shape :", y_test.shape)
```

```
x_train Shape : (60000, 28, 28, 1)
y_train Shape : (60000, 10)
x_test Shape : (10000, 28, 28, 1)
y_test Shape : (10000, 10)
```



8. Building the CNN

We are now ready to construct the Convolutional Neural Network (CNN) architecture. The CNN will consist of convolutional layers, max-pooling layers, and fully connected layers. These components work together to learn hierarchical features from the input data and make predictions.

The model is defined using the Keras Sequential API, and the architecture includes convolutional layers with varying numbers of filters, max-pooling layers for down-sampling, and dense layers for classification. The final model is compiled with a categorical cross-entropy loss function and the SGD optimizer.

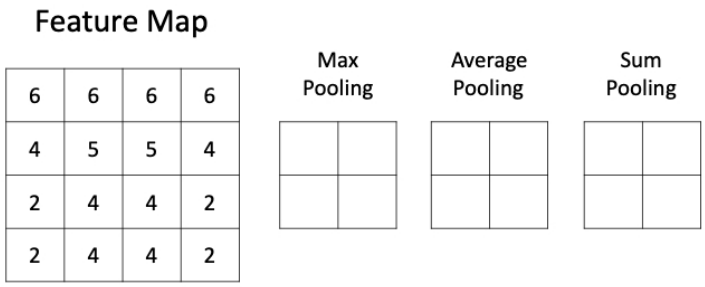
In [22]:
```
from keras.models import Sequential
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D

```

In [23]:
```

# img_rows, img_cols, channels = 28, 28, 1 # 1 for greyscale images and 3 for rgb images

# classes=10
# Define the dimensions of the input image
img_rows, img_cols, channels = 28, 28, 1  # 1 for greyscale images and 3 for rgb images

# Define the number of filters for each layer of the CNN
filters = [6, 32, 80 ,120]  # These are the number of filters in each layer of the CNN

# Define the number of classes for classification
classes = 10  # This is the number of different categories that the CNN will classify images into

```

In [24]:
```
# Creating Model

model=Sequential() #Sequential is a container to store layers
model.add(Conv2D(filters[0],(3,3),padding='same',\
                 activation='relu',input_shape=(img_rows,img_cols, channels)))
model.add(MaxPooling2D(pool_size=(2,2))) #For reducing image size
# (dim+pad-kernel)/2   (28 +3 -3)/2 = 14
model.add(Conv2D(filters[1],(2,2),padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
# (dim+pad-kernel)/2   (14 +2 -2)/2 = 7
model.add(Conv2D(filters[2],(2,2),padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
# (dim+pad-kernel)/2   (7 +2 -2)/2 = 3
model.add(Conv2D(filters[3],(2,2),padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
```

```
# (dim+pad-kernel)/2   (3 +2 -2)/2 = 1
model.add(Flatten())
model.add(Dense(64,activation='relu'))
model.add(Dense(classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

Pooling layer



In [25]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         60

 max_pooling2d (MaxPooling2  (None, 14, 14, 6)         0
 D)

 conv2d_1 (Conv2D)           (None, 14, 14, 32)        800

 max_pooling2d_1 (MaxPoolin  (None, 7, 7, 32)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 80)          10320

 max_pooling2d_2 (MaxPoolin  (None, 3, 3, 80)          0
 g2D)

 conv2d_3 (Conv2D)           (None, 3, 3, 120)         38520

 max_pooling2d_3 (MaxPoolin  (None, 1, 1, 120)         0
 g2D)

 flatten (Flatten)           (None, 120)               0

 dense (Dense)               (None, 64)                7744

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 58094 (226.93 KB)
Trainable params: 58094 (226.93 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

9. Parameters Calculations in CNN

The number of parameters in a convolutional layer is calculated using the formula:

(filter_height * filter_width * input_channels + 1) * number_of_filters

Layer 1: (3 x 3 x 1 + 1) x 6 = 60 parameters

Layer 2: (2 x 2 x 6 + 1) x 32 = 800 parameters

These parameters represent the weights and biases in the convolutional layers, and they play a crucial role in the learning process of the neural network.

10. Parameters for Artificial Neural Network (ANN) on MNIST

When working with a standard Artificial Neural Network (ANN) on the MNIST dataset, the number of parameters in each layer is calculated based on the input size, output size, and the architecture of the network.

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_19 (Dense)            (None, 400)               314000

 dense_20 (Dense)            (None, 20)                8020

 dense_21 (Dense)            (None, 10)                210

=================================================================
Total params: 322,230
Trainable params: 322,230
Non-trainable params: 0
```

11. Training the CNN

With the architecture defined, it's time to train the Convolutional Neural Network (CNN) on the provided dataset. The training process involves feeding the training data to the network, adjusting the weights and biases based on the calculated loss, and iterating through the dataset for a specified number of epochs.

We use the `fit` method to train the model, specifying parameters such as the training data, validation split, number of epochs, and batch size.

In [26]:
```
# Training Model
model.fit(x_train, y_train, validation_split= 0.2, epochs=15, batch_size=64, verbose=1)
# model.evaluate(x_test, y_test, verbose=2)
```

```
Epoch 1/15
750/750 [==============================] - 8s 5ms/step - loss: 2.0124 - accuracy: 0.3541 - val_loss: 1.0298 - val_accuracy: 0.6
811
Epoch 2/15
750/750 [==============================] - 4s 5ms/step - loss: 0.5877 - accuracy: 0.8126 - val_loss: 0.2711 - val_accuracy: 0.9
159
Epoch 3/15
750/750 [==============================] - 3s 4ms/step - loss: 0.2461 - accuracy: 0.9231 - val_loss: 0.1960 - val_accuracy: 0.9
367
Epoch 4/15
750/750 [==============================] - 5s 6ms/step - loss: 0.1625 - accuracy: 0.9496 - val_loss: 0.1314 - val_accuracy: 0.9
600
Epoch 5/15
750/750 [==============================] - 4s 5ms/step - loss: 0.1258 - accuracy: 0.9613 - val_loss: 0.0971 - val_accuracy: 0.9
711
Epoch 6/15
750/750 [==============================] - 4s 5ms/step - loss: 0.1041 - accuracy: 0.9675 - val_loss: 0.1006 - val_accuracy: 0.9
697
Epoch 7/15
750/750 [==============================] - 4s 5ms/step - loss: 0.0901 - accuracy: 0.9716 - val_loss: 0.0906 - val_accuracy: 0.9
728
Epoch 8/15
750/750 [==============================] - 4s 5ms/step - loss: 0.0804 - accuracy: 0.9743 - val_loss: 0.0809 - val_accuracy: 0.9
740
Epoch 9/15
750/750 [==============================] - 3s 4ms/step - loss: 0.0724 - accuracy: 0.9779 - val_loss: 0.0765 - val_accuracy: 0.9
775
Epoch 10/15
750/750 [==============================] - 4s 6ms/step - loss: 0.0653 - accuracy: 0.9798 - val_loss: 0.0758 - val_accuracy: 0.9
769
Epoch 11/15
750/750 [==============================] - 4s 5ms/step - loss: 0.0604 - accuracy: 0.9810 - val_loss: 0.0673 - val_accuracy: 0.9
793
Epoch 12/15
750/750 [==============================] - 3s 4ms/step - loss: 0.0559 - accuracy: 0.9827 - val_loss: 0.0634 - val_accuracy: 0.9
808
Epoch 13/15
750/750 [==============================] - 4s 5ms/step - loss: 0.0516 - accuracy: 0.9836 - val_loss: 0.0662 - val_accuracy: 0.9
797
Epoch 14/15
750/750 [==============================] - 4s 5ms/step - loss: 0.0484 - accuracy: 0.9848 - val_loss: 0.0719 - val_accuracy: 0.9
787
Epoch 15/15
750/750 [==============================] - 3s 4ms/step - loss: 0.0460 - accuracy: 0.9854 - val_loss: 0.0606 - val_accuracy: 0.9
811
```

Out[26]:
```
<keras.src.callbacks.History at 0x78e3d06bd120>
```

12. Testing the Model

Now that the Convolutional Neural Network (CNN) has been trained, it's time to evaluate its performance on the test set. We predict probabilities for the test set using the trained model and then convert these probabilities into binary predictions based on a threshold (0.5 in this case).

We calculate the test accuracy using the predicted and true labels and print the result.

In [27]:
```
# Import the necessary libraries
```

```
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt

# Predict probabilities for the test set using the trained model
y_pred_probs = model.predict(x_test, verbose=0)
y_pred = np.where(y_pred_probs > 0.5, 1, 0)

# Calculate and print the test accuracy using predicted and true labels
test_accuracy = accuracy_score(y_pred, y_test)
print("\nTest accuracy: {}".format(test_accuracy))

```

```
Test accuracy: 0.9826
```

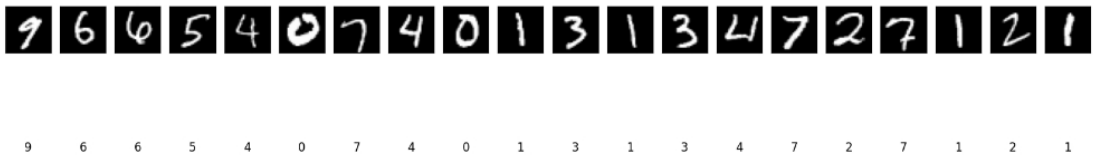
13. Visualizing Predictions for Selected Samples

In this step, we define a mask to select a range of indices (20 to 49) from the test set. We then visualize the original images along with the predicted digits for the selected validation samples.

In [28]:
```
# Define a mask for selecting a range of indices (20 to 49)
mask = range(20, 50)

# Select the first 20 samples from the test set for visualization
X_valid = x_test[20:40]
actual_labels = y_test[20:40]

# Predict probabilities for the selected validation samples
y_pred_probs_valid = model.predict(X_valid)
y_pred_valid = np.where(y_pred_probs_valid > 0.5, 1, 0)

```

```
1/1 [==============================] - 0s 165ms/step
```

In [29]:
```
# Set up a figure to display images
n = len(X_valid)
plt.figure(figsize=(20, 4))

for i in range(n):
    # Display the original image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(X_valid[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display the predicted digit
    predicted_digit = np.argmax(y_pred_probs_valid[i])
    ax = plt.subplot(2, n, i + 1 + n)
    plt.text(0.5, 0.5, str(predicted_digit), fontsize=12, ha='center', va='center')
    plt.axis('off')

# Show the plotted images
plt.show()

# Close the plot
plt.close()

```



Thanks for viewing my work. If you like it, consider sharing it to others or give feedback to improve the notebook. Have a beautiful day my friend.