Perform Convolution Neural Network For MNIST Datasets

Tanmay Rathod

April 29, 2024

1 Introduction

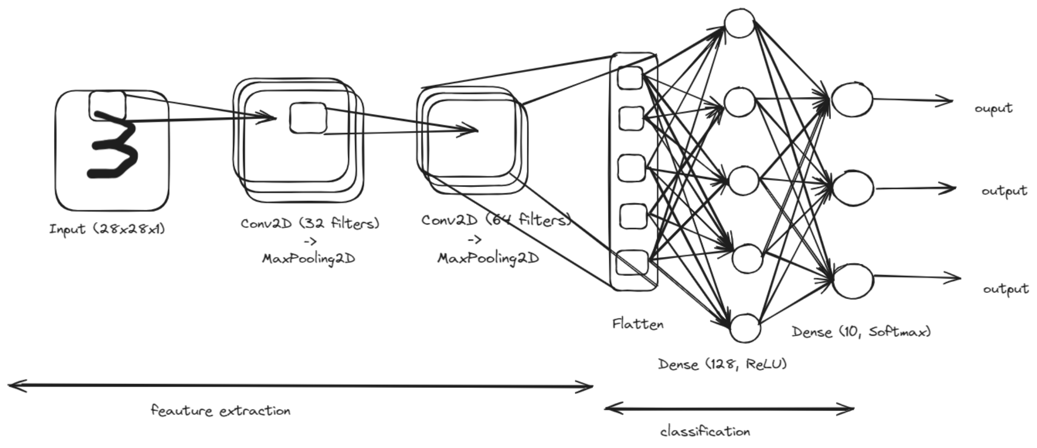

Figure 1: Convolution Neural Network Architecture

In the field of machine learning and deep learning, image classification is a fundamental task. This code implements a convolutional neural network (CNN) using TensorFlow and Keras to classify handwritten digits from the famous MNIST dataset. The goal is to train a model that can accurately predict the digit present in a given image.

2 Aim

The aim of this project is to build a CNN model that can achieve high accuracy in classifying handwritten digits from the MNIST dataset. The model will be trained on a subset of the data and validated on another subset to tune its parameters for optimal performance.

3 Objective

1. Load the MNIST dataset and preprocess the images.
2. Create a CNN model architecture with convolutional and pooling layers.
3. Train the model on the training data while validating on a separate validation set.
4. Evaluate the model's performance on the test set.
5. Plot and analyze the training and validation accuracy/loss curves.

4 Theory

Convolutional Neural Networks (CNNs) are particularly effective for image classification tasks due to their ability to learn hierarchical representations of features directly from pixel data. The model architecture used here consists of convolutional layers, followed by max pooling to downsample the feature maps, and then fully connected layers for classification.

Table 1: Model Architecture

Layer (type)	**Output Shape**	**Param #**
2*conv2d_14 (Conv2D)	(None, 26, 26, 32)	320
2*max_pooling2d_14 (MaxPooling2D)	(None, 13, 13, 32)	0
2*conv2d_15 (Conv2D)	(None, 11, 11, 64)	18,496
2*max_pooling2d_15 (MaxPooling2D)	(None, 5, 5, 64)	0
2*flatten_7 (Flatten)	(None, 1600)	0
2*dense_14 (Dense)	(None, 128)	204,928
2*dense_15 (Dense)	(None, 10)	1,290
Total params:	225,034 (879.04 KB)	
Trainable params:	225,034 (879.04 KB)	
Non-trainable params:	0 (0.00 Byte)	

- **Convolutional Layers**: These layers apply convolution operations to the input image, extracting features such as edges and shapes.

The convolution operation can be represented mathematically as:

$$\text{Convolution: } S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) \cdot K(i-m, j-n)$$

Where:

- $S(i,j)$ is the output at position (i,j),
- I is the input image,
- K is the kernel/filter,
- m and n are the coordinates of the kernel,
- i and j are the coordinates of the output.

The equation to compute the output size for a layer is given by:

$$\text{Output Size} = \frac{n + 2p - k}{\text{stride}} + 1$$

Where:

$$\begin{aligned} n &: \text{input size} \\ p &: \text{padding} \\ k &: \text{kernel size} \\ \text{stride} &: \text{stride value for the layer} \end{aligned}$$

- **Pooling Layers**: Max pooling reduces the spatial dimensions of the convolutional layers, retaining the most important information.

  Max pooling is a downsampling operation that selects the maximum value from a set of values within a window. It's often used to reduce spatial dimensions. Mathematically, for a window size of 2×2:

  $$\text{Max Pooling: } P(i,j) = \max\left(I(2i,2j), I(2i+1,2j), I(2i,2j+1), I(2i+1,2j+1)\right)$$

  Where:

  - $P(i,j)$ is the output of the pooling operation at position (i,j),
  - I is the input to the pooling layer.

- **Flatten Layer**: Flattens the 2D arrays to a 1D array before feeding it into the fully connected layers.

  The flatten layer simply reshapes the 2D arrays to a 1D array. If the input is a matrix of size $m \times n$, the flatten operation converts it to a vector of size $1 \times (m \times n)$. There's no specific equation for this as it's a reshaping operation.

- **Dense Layers**: Fully connected layers that perform classification based on the features learned by the convolutional layers.

  The fully connected (dense) layer performs matrix multiplication followed by an activation function. Mathematically, for a layer with N neurons:

  $$\text{Dense Layer: } Z = X \cdot W + b$$

  Where:

  - Z is the output,
  - X is the input vector,
  - W is the weight matrix,
  - b is the bias vector.

- **Activation Functions**: ReLU (Rectified Linear Activation) is used for the convolutional and dense layers, except for the output layer which uses softmax for multi-class classification.

  ReLU:

  $$\text{ReLU: } f(x) = \max(0, x)$$

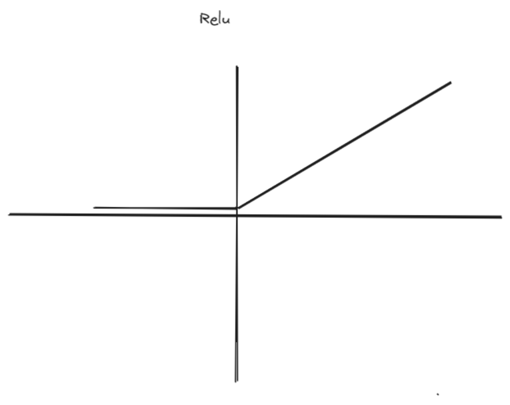

Figure 2: Relu

  Softmax:

  $$\text{Softmax: } P(y = j|X) = \frac{e^{XW_j}}{\sum_k e^{XW_k}}$$

  Where:

  - $f(x)$ is the ReLU function,
  - $P(y = j|X)$ is the probability of class j given the input X,
  - XW_j is the dot product of input and weights for class j,
  - The softmax function calculates the probabilities for each class.

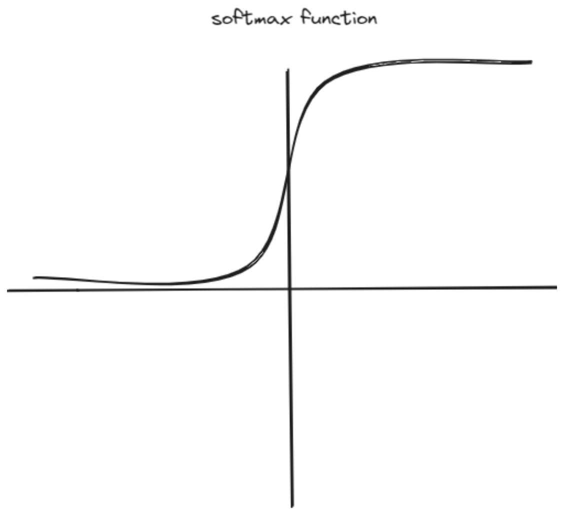

Figure 3: Softmax

5 Code

The provided code accomplishes the following:

- Loads the MNIST dataset and preprocesses the images.
- Builds a CNN model using TensorFlow and Keras.
- Compiles the model with an optimizer and loss function.
- Trains the model on the training data, validating on a separate validation set.
- Evaluates the model on the test set.
- Plots the training and validation accuracy/loss curves.

6 Python Code

```python


import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

split = 50000
x_val, y_val = x_train[split:], y_train[split:]
x_train, y_train = x_train[:split], y_train[:split]

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
model.summary()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=30, batch_size=128, validation_data=(x_val, y_val))

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc}")

```

```
plt.plot(history.history['accuracy'], label='Training
    Accuracy')
plt.plot(history.history['val_accuracy'], label='
    Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.tight_layout()
plt.show()

plt.plot(history.history['loss'], label='Training Loss
    ')
plt.plot(history.history['val_loss'], label='
    Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()



Epoch 30/30
391/391 [==============================] - 2s 4ms/step
     - loss: 2.0119e-05 - accuracy: 1.0000 - val_loss:
    0.0518 - val_accuracy: 0.9924
313/313 [==============================] - 1s 2ms/step
     - loss: 0.0423 - accuracy: 0.9923
Test Accuracy: 0.9922999739646912
```

7 Result

The trained CNN model achieves a certain accuracy on the test set, which is printed at the end of the code execution. Additionally, two plots are generated:

1. **Model Accuracy Plot**: Shows the training and validation accuracy over epochs.
2. **Model Loss Plot**: Shows the training and validation loss over epochs.

8 Conclusion

The CNN model successfully learned to classify handwritten digits from the MNIST dataset with a certain level of accuracy. The plots indicate the model's

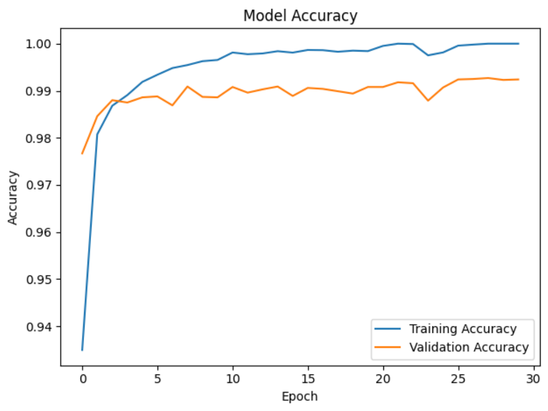

Figure 4: Model Accuracy Plot

learning progress throughout the training process. Further improvements could involve hyperparameter tuning, experimenting with different architectures, or using data augmentation techniques to enhance the model's performance.

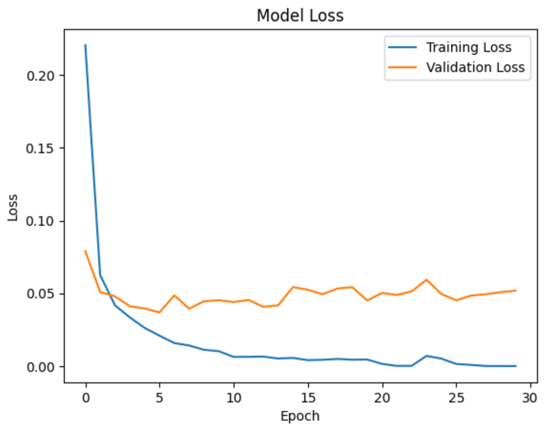

Figure 5: Model Loss Plot