University of Cincinnati

Convolutional Neural Network

Deep learning: Homework 2

Benjamin PHAN

Contents

INTRODUCTION

Deep learning models have become pivotal in tasks such as image classification, object detection, and more. In this report, we delve into the world of CNNs, exploring their architecture, training, and performance in the context of image classification.

The goal of this homework is to create and experiment with various Convolutional Neural Network (CNN) models to predict the numbers depicted in images from the MNIST dataset. MNIST is a dataset consisting of handwritten digit images, each labeled with the corresponding digit it represents.

During this project we will compare different models of prediction such as DNN model, ConvNet model and lastly ResNet model.

I. DATA

A. Data preview

MNIST dataset aims to solve the problem of handwritten digit recognition. It serves as a benchmark for testing the performance of image classification models. The pixel values in the dataset range from 0 (black) to 255 (white), representing grayscale intensities. Each data sample consists of a 28x28 pixel image.

The dataset contains labels for each image, indicating the digit (0-9) that the handwritten digit represents. The MNIST dataset is well-preprocessed, and there are no missing features or data points.

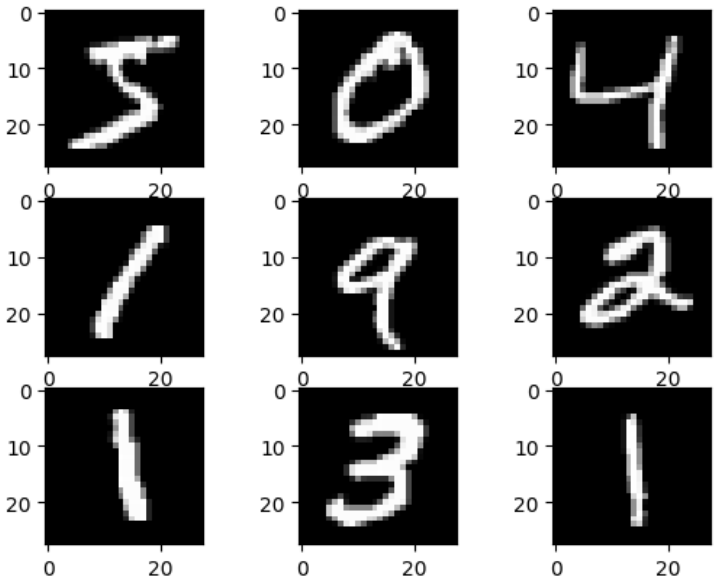


Figure 1 First images

B. Data split

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

In Keras, the MNIST dataset is already divided into a total of 70,000 samples. Approximately 60,000 of these samples are designated for training, while the remaining 10,000 are reserved for testing.

C. Data normalization

```
train_images, test_images = train_images / 255.0, test_images / 255.0
```

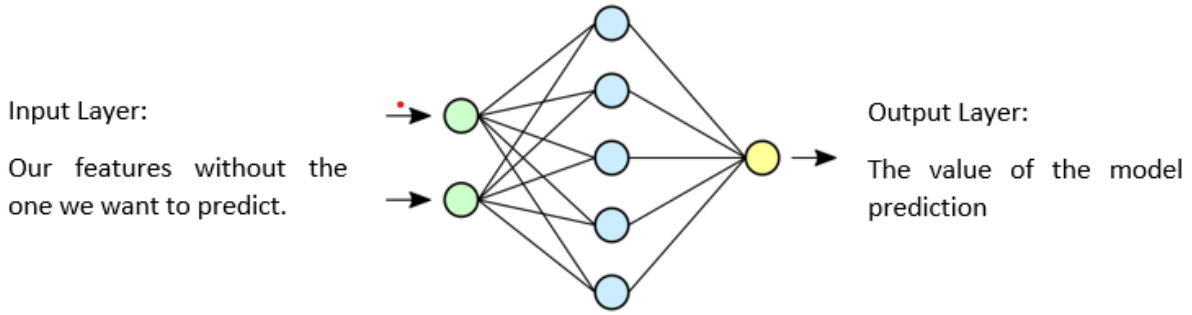
Initially images have pixel values ranging from 0 to 255, representing grayscale intensity. To ensure uniformity and facilitate model convergence, we apply a normalization. Through this process, we rescale the pixel values to a standardized range of 0 to 1.

II. Models

A. DNN

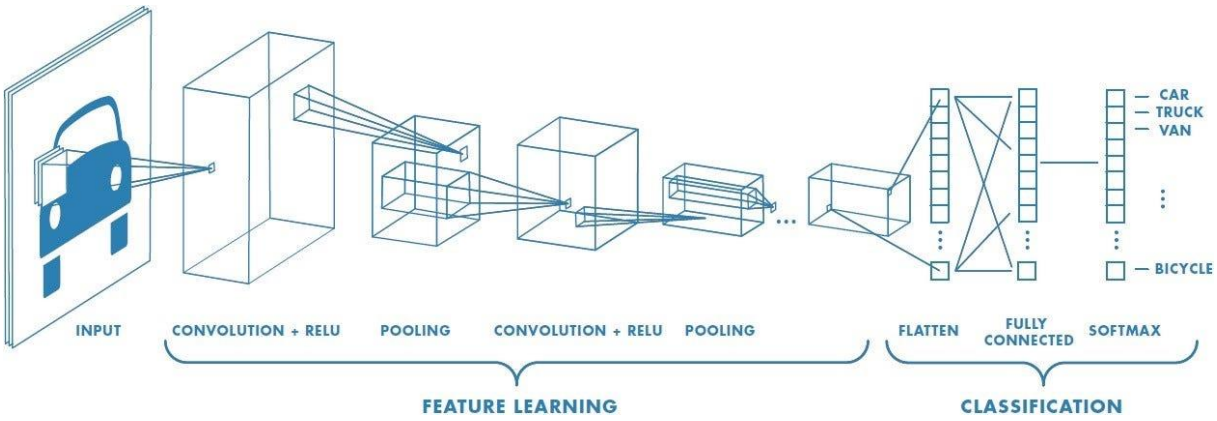
Deep Neural Network (DNN) model, which is a standard feedforward neural network composed of multiple fully connected layers.

```
def model_DNN(input_shape,num_classes, LR, *layers):
      model = tf.keras.Sequential()
      tf.keras.Input(input_shape)

      for elt in layers[0:len(layers)]:
            model.add(tf.keras.layers.Dense(elt, activation='relu'))

      model.add(tf.keras.layers.Flatten())

      model.add(tf.keras.layers.Dense(num_classes,activation='softmax'))

      model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=LR),
            loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
            metrics=['accuracy'])

      return model
```



The code iteratively adds dense layers with ReLU activation, followed by a flatten layer to prepare the data for classification. The output layer employs softmax activation to produce class probabilities.

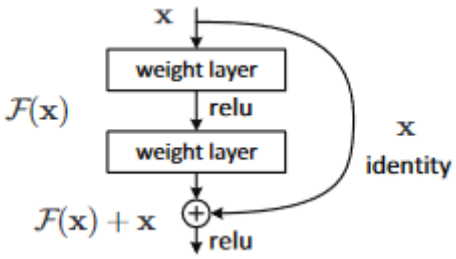
B. ConvNet




```python
def model_ConvNet(input_shape,num_classes, LR, *layers):
      model = tf.keras.Sequential()

      tf.keras.Input(input_shape)

      for elt in layers[0:len(layers)-1]:
            model.add(tf.keras.layers.Conv2D(elt,(3,3), activation='relu'))
            model.add(tf.keras.layers.MaxPooling2D((2, 2)))

      model.add(tf.keras.layers.Conv2D(layers[-1],(3,3), activation='relu'))

      model.add(tf.keras.layers.Flatten())

      model.add(tf.keras.layers.Dense(64, activation='relu'))

      model.add(tf.keras.layers.Dense(num_classes,activation='softmax'))

      model.build((None,28,28,1))
      model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=LR),
            loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
            metrics=['accuracy'])

      return model
```

The ConvNet model (Convolutional Neural Network), is a deep learning architecture specialized for image-related tasks. It is designed to automatically learn and extract relevant features from images, making it highly effective for image classification tasks. This model typically consists of convolutional layers, pooling layers, and fully connected layers.

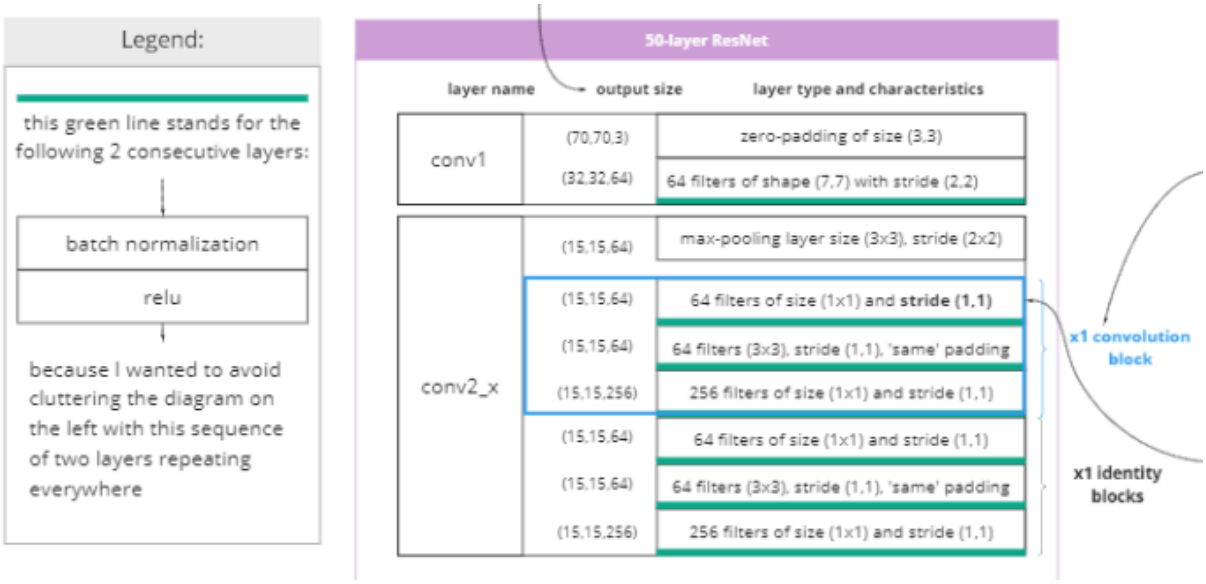
C. ResNet

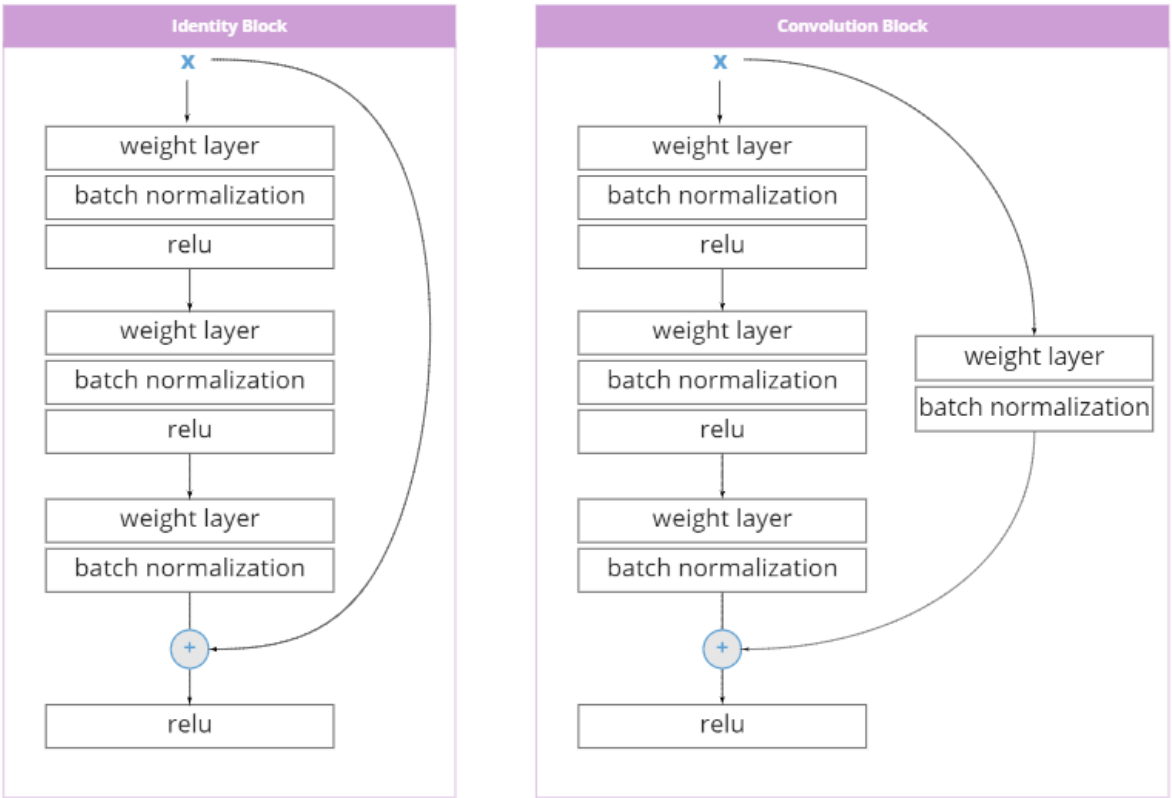



```
def model_ResNet(input_shape,num_classes, nbOfBlock, LR):
      model = tf.keras.Sequential()
      input_layer = tf.keras.layers.Input(input_shape)

      x =tf.keras.layers.Conv2D(64,(3,3))(input_layer)
      x = tf.keras.layers.BatchNormalization()(x)
      x = tf.keras.layers.Activation('relu')(x)

      x = tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2))(x)
      layers = [64,64,256]
      for i in range(nbOfBlock):
            if i == 0:
                  x = ResNetBlockConv(x, layers)
                  x = ResNetBlockId(x, layers)
            else:
                  l = list(np.array(layers) * (i+1))
                  x = ResNetBlockConv(x, l, 2)
                  x = ResNetBlockId(x, l)


      x = tf.keras.layers.GlobalAveragePooling2D()(x)

      x = tf.keras.layers.Flatten()(x)

      output_layer = tf.keras.layers.Dense(num_classes,activation='softmax')(x)
      model = tf.keras.models.Model(inputs=input_layer, outputs=output_layer)

      model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=LR),
            loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
            metrics=['accuracy'])

      return model
```

The ResNet model (Residual Neural Network), is a deep learning architecture that addresses the vanishing gradient problem in very deep networks. It introduces the concept of residual blocks, which allow the network to learn residual functions. These blocks include skip connections that skip one or more layers, making it easier for gradients to flow during training.

For my RestNet model I tried to build something like the first block of a ResNet50:





(source : https://www.kaggle.com/code/mishki/resnet-keras-code-from-scratch-train-on-gpu)

III. Objective

Cross-entropy is the loss function used to train the different models.

$$CE = -\sum_{i=l}^{n} y_i \log(\hat{y}_i)$$

IV. Optimization

The Adam optimizer, short for Adaptive Moment Estimation, is a widely adopted optimization algorithm for training deep neural networks. It combines the advantages of two other popular optimizers, namely RMSprop and Momentum. Adam adapts the learning rates of individual model parameters based on their historical gradients and updates, making it highly effective in optimizing models with complex and non-stationary loss surfaces.

The key reasons for choosing Adam as an optimization algorithm is its ability to accelerate convergence during training, typically resulting in faster model training and convergence to a good solution.

V. Model selection

By comparing accuracy scores between the training and testing datasets, we can effectively gauge whether a model is encountering issues of underfitting or overfitting. For instance, if a model exhibits strong performance on the training data but significantly lower accuracy on the test set, it's indicative of overfitting. On contrary if the model shows low performance, it might be underfitting.

During the training process, even as the accuracy on the training data appears to improve, a drop in accuracy on the validation dataset can be observed. This is often a consequence of overfitting. To avoid this, early stopping can be employed to halt training and preserve the model's best parameters, ensuring better generalization.

VI. Model performance

Best Model	ACCURACY | F1
DNN	0.965 | 0.965
ConvNet	0.991 | 0.991
ResNet	0.992 | 0.992

Table 1

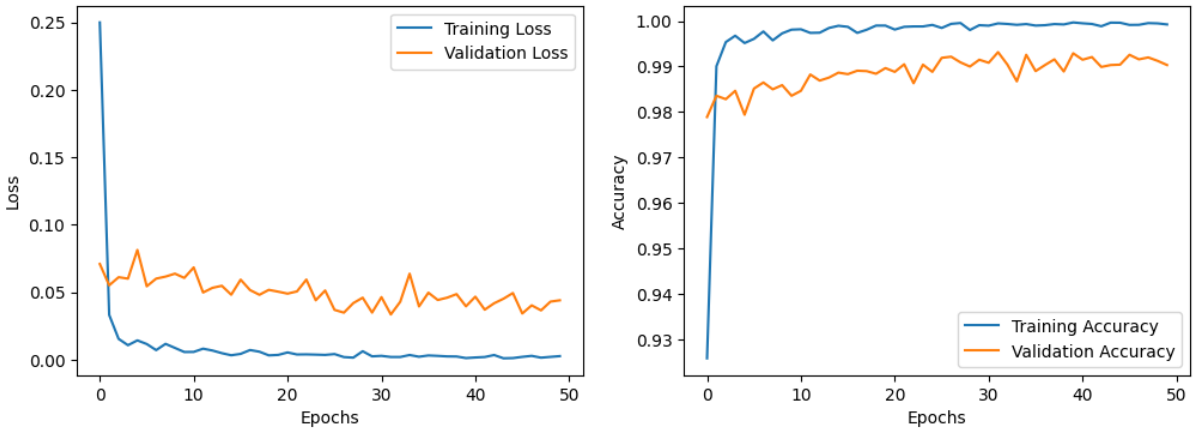
Model (ACCURACY | F1)	LR: 0.1	LR: 0.01	LR: 0.001	LR: 0.0001
DNN	0.78 | 0.77	0.95 | 0.95	0.964 | 0.964	0.965 | 0.995
ConvNet	0.11 | 0.02	0.98 | 0.98	0.987 | 0.987	0.991 | 0.991
ResNet	0.97 | 0.98	0.99 | 0.98	0.982 | 0.982	0.989 | 0.989

Table 2

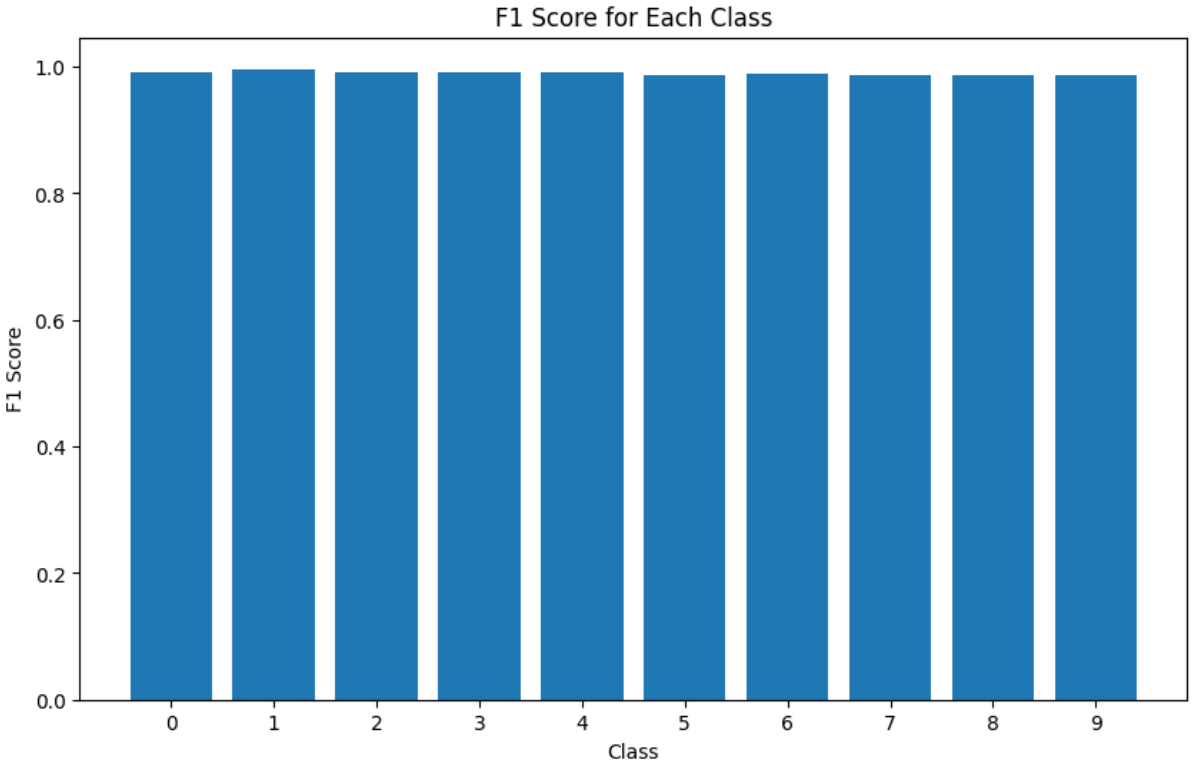
As presented in Table 1, the ResNet model stands out as the top-performing model, demonstrating its best performance when trained with a learning rate of 0.001. Table 2 provides an overview of the performance metrics achieved across various learning rates, showcasing the model's sensitivity to different learning rate values. Moreover, Table 2 shows that lower is the learning rate, lower are the performance. Nonetheless, the ResNet model seems to have good performance over all levels of learning rate.

A high learning rate typically results in a shorter training time, requiring fewer epochs to converge to a satisfactory model performance. Nonetheless, the model may overshoot and oscillate around the minimum of the loss function.

Below are examples of plots and values obtained for a ResNet model with a learning rate of 0.0001 (others screenshots can be find in the “screenshot” folder) :




```
313/313 [==============================] - 2s 6ms/step - loss: 0.0466 - accuracy: 0.9899
Test accuracy: 0.9898999929428101
```




```
Macro-Average F1 Score: 0.98977694008127
```