## install mnist library

This Library used to load mnist dataset

```
!pip install mnist

    Collecting mnist
      Downloading mnist-0.2.2-py2.py3-none-any.whl (3.5 kB)
    Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from mnist) (1.23.5)
    Installing collected packages: mnist
    Successfully installed mnist-0.2.2
```

import mnist

```
import mnist
```

## Create training dataset and test dataset from nist dataset

- x_train: training images
- x_test: test images
- y_train: training label
- y_test: test label

```
x_train, x_test, y_train, y_test = mnist.train_images(), mnist.test_images(), mnist.train_labels(), mnist.test_labels()
```

```
x_train.shape

    (60000, 28, 28)
```

```
x_test.shape

    (10000, 28, 28)
```

## pixel normalization for training and test dataset

```
x_train, x_test = x_train / 255, x_test / 255
```

Double-click (or enter) to edit

```
from tensorflow.keras.utils import to_categorical
```

## Categorize training and test labels

```
y_train, y_test = to_categorical(y_train), to_categorical(y_test)
```

```
from tensorflow.keras import layers, models
```

## neural network model using the TensorFlow and Keras libraries.

- **models.Sequential**: This creates a sequential model, which is a linear stack of layers where we can add one layer at a time. In this context, it's used to define the architecture of the neural network.
- **layers.Flatten(input_shape=(28, 28))**: This is the input layer of the neural network. It takes an input with a shape of (28, 28), which corresponds to 28x28 pixel grayscale images. The **Flatten** layer is responsible for converting the 2D array of pixel values into a 1D array by flattening it. This is necessary because the subsequent layers in the neural network expect a 1D input.
- **layers.Dense(128, activation='relu')**: This is the first hidden layer of the neural network. It is a fully connected (Dense) layer with 128 neurons. The 'relu' activation function (Rectified Linear Unit) is applied to the output of this layer. The purpose of this layer is to learn complex patterns in the data by applying a linear transformation followed by the activation function.
- **layers.Dropout(0.2)**: This is a dropout layer. Dropout is a regularization technique that helps prevent overfitting. It randomly sets a fraction (in this case, 20%) of the input units to 0 during training, which helps to reduce the co-dependency of neurons and makes the network more robust.

- **layers.Dense(10, activation='softmax')**: This is the output layer of the neural network. It is another fully connected layer with 10 neurons, one for each possible digit (0 to 9). The 'softmax' activation function is applied to this layer, which converts the raw output values into a probability distribution over the 10 classes. This means the output values will represent the probabilities of the input image belonging to each digit class.

this neural network model takes 28x28 pixel images as input, flattens them into a 1D array, passes them through a hidden layer with 128 neurons and ReLU activation, applies dropout for regularization, and finally, produces output probabilities for each of the 10 digit classes using the softmax activation function. This architecture is suitable for the MNIST dataset, which is a collection of handwritten digit images that need to be classified into one of the 10 digits.

```
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])
```

## Configure the training process for a neural network model.

- **optimizer='adam'**:
    - The **optimizer** argument specifies the optimization algorithm used to update the model's weights during training. In this case, it is set to 'adam', which stands for Adaptive Moment Estimation. Adam is a popular optimization algorithm known for its efficiency and ability to adapt learning rates during training.
- **loss='categorical_crossentropy'**:
    - The **loss** argument defines the loss function that the model will use to measure how well it's performing during training. For classification problems with multiple classes, 'categorical_crossentropy' is a common choice. It measures the dissimilarity between the true class probabilities (one-hot encoded labels) and the predicted probabilities generated by the model. The goal during training is to minimize this loss, which essentially means making the predicted probabilities as close as possible to the true class probabilities.

3. `metrics=['accuracy']`:
    - The `metrics` argument is a list of metrics used to evaluate the model's performance during and after training. In this case, 'accuracy' is specified, which is a common metric for classification tasks. Accuracy measures the fraction of correctly classified samples over the total number of samples. During training, the model will compute and display this metric so you can monitor how well it's learning to classify the data.

**model.compile** is a critical step in setting up the neural network for training. It configures the optimization algorithm, specifies the loss function to optimize, and defines evaluation metrics to track the model's performance. Once the model is compiled, we can start training it on the dataset using the **fit** method, and it will use the specified optimizer and loss function to learn from the data while providing accuracy as a performance metric to assess its training progress.

```
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

## Train neural network model on a given dataset.

- **x_train and y_train**:
    - **x_train** represents the input data (features) that we want to use to train the model. This typically consists of a set of input samples, such as images in the case of image classification.
    - **y_train** represents the corresponding target labels or ground truth for the input data. These are the correct answers or labels associated with each input sample. In classification tasks, **y_train** usually consists of one-hot encoded labels or integer values representing the class of each input sample.
- **epochs=5**:
    - The **epochs** argument specifies the number of times the entire training dataset (i.e., **x_train and y_train**) should be passed forward and backward through the neural network. Each pass through the entire dataset is called an "epoch." In this case, the training process will go through the dataset 5 times.
- **batch_size=64**:
    - The **batch_size** argument determines how many samples are used in each update of the model's weights. Instead of updating the weights after each individual sample (which can be computationally inefficient), training is typically done in batches. Here, the batch size is set to 64, meaning that the model's weights will be updated once every 64 samples.

- **validation_split=0.2**:
  - The **validation_split** argument is used to specify the fraction of the training data that should be reserved for validation. In this case, 20% of the training data (**x_train and y_train**) will be set aside for validation purposes. This validation data is not used for training but is used to monitor the model's performance during training, helping to detect overfitting.

When we execute **model.fit**, the training process begins. During each epoch, the following steps occur:

- The training data (**x_train and y_train**) is divided into batches of size specified by **batch_size**.
- For each batch, the model computes predictions on the input data.
- The loss function specified during model compilation (in this case, 'categorical_crossentropy') is used to compute the difference between the predicted values and the true labels (i.e., the training loss).
- The optimization algorithm (Adam in this case) adjusts the model's weights to minimize this loss.
- After processing all batches in an epoch, the model's performance on the validation data (if **validation_split** is provided) is evaluated using the same loss and any specified metrics (e.g., 'accuracy').
- Training progresses through the specified number of epochs (5 in this case).

At the end of training, we will have a trained neural network model that has hopefully learned to make accurate predictions on our data. we can then use this trained model for making predictions on new, unseen data.

```
model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.2)
```

```
Epoch 1/5
750/750 [==============================] - 5s 6ms/step - loss: 0.3742 - accuracy: 0.8937 - val_loss: 0.1721 - val_accura
Epoch 2/5
750/750 [==============================] - 4s 5ms/step - loss: 0.1749 - accuracy: 0.9499 - val_loss: 0.1307 - val_accura
Epoch 3/5
750/750 [==============================] - 3s 4ms/step - loss: 0.1310 - accuracy: 0.9616 - val_loss: 0.1084 - val_accura
Epoch 4/5
750/750 [==============================] - 4s 5ms/step - loss: 0.1071 - accuracy: 0.9678 - val_loss: 0.0957 - val_accura
Epoch 5/5
750/750 [==============================] - 5s 6ms/step - loss: 0.0885 - accuracy: 0.9729 - val_loss: 0.0860 - val_accura
<keras.src.callbacks.History at 0x79191fbe50c0>
```

## Evaluate a trained neural network model on a separate test dataset to assess its performance.

- **x_test and y_test**:
  - **x_test** represents the test data, which is a set of input samples that the model has never seen during training or validation. It is used to evaluate how well the model generalizes to new, unseen data.
  - **y_test** represents the corresponding target labels or ground truth for the test data. These are the correct answers or labels associated with each input sample in **x_test**.
- **model.evaluate(x_test, y_test)**:
  - The **model.evaluate** method takes the test data **x_test and y_test** as inputs and computes two main quantities:
    - **test_loss**: This is the value of the loss function (specified during model compilation, typically 'categorical_crossentropy' for classification tasks) on the test dataset. It quantifies how well the model's predictions match the true labels on the test data. Lower values indicate better performance.
    - **test_accuracy**: This is the accuracy of the model on the test dataset. It measures the fraction of correctly classified samples in the test data.
- **Assignment**:
  - The results of **model.evaluate** are assigned to the variables **test_loss and test_accuracy** for further analysis or reporting.

This allows us to assess how well our trained neural network model performs on unseen data (the test dataset). It calculates the loss and accuracy of the model on this test data, providing a quantitative measure of its generalization and predictive performance. These metrics are crucial for evaluating the model's suitability for real-world applications and for comparing different models or configurations.

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0827 - accuracy: 0.9742
```

## print the accuracy

```
test_accuracy
```

```
0.9742000102996826
```