Description about Import

1. os: The os module in Python provides a way of using operating system-dependent functionality. In this context, it might be used for operations related to file and directory manipulation, such as checking file existence or listing directory contents.

2. cv2 (OpenCV): OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision. It's commonly used for image processing tasks such as reading, writing, and manipulating images, as well as various computer vision algorithms.

3. numpy (alias np): NumPy is a powerful Python library for numerical computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. In this context, it might be used for numerical operations on image data.

4. matplotlib.pyplot (alias plt): Matplotlib is a plotting library for Python that provides a MATLAB-like interface. The pyplot module provides a convenient way to create plots and visualizations. It can be used here to display images or visualize data.

5. tensorflow (alias tf): TensorFlow is an open-source machine learning library developed by Google. It provides a comprehensive ecosystem of tools and libraries for building and deploying machine learning models, including neural networks. In this context, it might be used for building and training neural networks for tasks such as image classification or object detection.

In [ ]:
```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

Load the Data Set

In [ ]:
```
mnist = tf.keras.datasets.mnist
```

In [ ]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
```

Preprocessing

Normalization

Normalization is a common preprocessing technique used in machine learning to scale the input features to a similar range. This ensures that each feature contributes approximately proportionately to the learning process and prevents any particular feature from dominating due to its larger magnitude.

'tf.keras.utils.normalize' : This function normalizes the input data along the specified axis. In this case, axis=1 indicates that normalization is performed along the feature axis, i.e., each feature vector is normalized independently. The normalization is typically done by dividing each feature vector by its Euclidean norm (L2-norm), resulting in a unit norm for each sample.

Reasons for Normalization:

- Improved Convergence: Normalizing the input data can help the optimization algorithm converge faster during training. When features are on different scales, the learning rate may need to be adjusted to accommodate the varying magnitudes, leading to slower convergence.
- Stability: Normalization can improve the numerical stability of the training process, especially for algorithms sensitive to the scale of input features. It prevents issues such as vanishing or exploding gradients.
- Generalization: Normalization can aid in better generalization of the model by preventing overfitting. It helps the model focus on learning meaningful patterns rather than being influenced by the scale of the input features.
- Interpretability: Normalized features are often easier to interpret, as they are on a similar scale, making comparisons between features more straightforward.

In [ ]:
```
X_train = tf.keras.utils.normalize(X_train, axis = 1)
X_test = tf.keras.utils.normalize(X_test, axis = 1)
```

Building the Model

This code defines a neural network model using TensorFlow's Keras API.

tf.keras.models.Sequential(): This creates a sequential model, which is a linear stack of layers. In this type of model, each layer has exactly one input tensor and one output tensor. The sequential model is suitable for most deep learning tasks, where the data flows sequentially through the layers.

model.add(tf.keras.layers.Flatten(input_shape=(28,28))): The Flatten layer is the first layer of the model. It transforms the input data, which in this case is 2D (28x28) images, into a 1D array (or vector) of size 784 (28*28). This flattening step is necessary because the subsequent layers, such as densely connected layers, require a 1D input.

model.add(tf.keras.layers.Dense(128, activation='relu')): This adds a fully connected (dense) layer with 128 neurons and ReLU (Rectified Linear Unit) activation function. Dense

layers are used to capture complex patterns in the data by connecting each neuron to every neuron in the previous layer. The ReLU activation function introduces non-linearity to the model, allowing it to learn and represent non-linear relationships in the data.

model.add(tf.keras.layers.Dense(128, activation='relu')): Another fully connected layer with 128 neurons and ReLU activation function is added. This additional layer can help the model learn more complex representations from the data and potentially improve its performance.

model.add(tf.keras.layers.Dense(10, activation='softmax')): The final layer is a fully connected layer with 10 neurons, corresponding to the number of classes in the classification task (assuming this is a classification problem with 10 classes). The softmax activation function is used in this layer to output probabilities for each class. Softmax ensures that the output values sum up to 1, making them interpretable as probabilities. The predicted class is typically the one with the highest probability.

Reasons for this architecture:

- Input Layer: The Flatten layer converts the 2D image data into a format suitable for the subsequent dense layers.
- Hidden Layers: The two Dense layers with ReLU activation introduce non-linearity and enable the model to learn complex patterns in the data.
- Output Layer: The final Dense layer with softmax activation produces the probability distribution over the output classes, allowing the model to make predictions.

In [ ]:
```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(128, activation = 'relu'))
model.add(tf.keras.layers.Dense(128, activation = 'relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

Evaluate the Model

The compile method configures the model for training.

optimizer='adam': This specifies the optimizer to be used during training. Adam (short for Adaptive Moment Estimation) is a popular optimization algorithm that combines ideas from RMSProp and Momentum. It's well-suited for a wide range of deep learning tasks due to its adaptive learning rate and momentum properties. Adam adapts the learning rate for each parameter individually, which can lead to faster convergence and better performance compared to traditional optimization algorithms.

loss='sparse_categorical_crossentropy': This specifies the loss function to be used during training. Cross-entropy loss is commonly used in classification tasks to measure the difference between the predicted probability distribution and the true distribution of the labels. In this case, sparse_categorical_crossentropy is used because the labels are integers (sparse representation) and not one-hot encoded. This loss function calculates the cross-entropy between the true labels and the predicted probabilities.

metrics='accuracy': This specifies the evaluation metric to be used during training and testing. Accuracy is a common metric used in classification tasks to measure the proportion of correctly classified samples. It calculates the percentage of samples for which the predicted class matches the true class. Monitoring accuracy during training provides insights into how well the model is learning to classify the data correctly.

Reasons for these choices:

Adam Optimizer: Adam is a popular choice for optimization due to its adaptive learning rate, which helps to adjust the learning rates for different parameters individually. This adaptiveness can lead to faster convergence and better generalization.

Sparse Categorical Crossentropy Loss: Since the labels are integers (not one-hot encoded), sparse_categorical_crossentropy is appropriate for this classification task. It computes the cross-entropy loss between the true labels and the predicted probabilities, providing a measure of how well the model's predictions match the true distribution of the labels.

Accuracy Metric: Accuracy is a straightforward and intuitive metric for classification tasks. It provides a clear measure of the model's performance in terms of correctly classified samples. Monitoring accuracy during training helps to assess the progress of the model and its ability to classify the data correctly.

In [ ]:
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
```

An epoch is one complete pass through the entire training dataset. The epochs parameter specifies the number of times the training algorithm will iterate over the entire dataset during training. In this case, epochs = 3 indicates that the training process will iterate over the dataset three times.

Reasons for these choices:

Training Data: The fit method requires both input features (X_train) and corresponding target labels (y_train) to train the model. This allows the model to learn the mapping between the input data and the target labels, enabling it to make predictions on unseen data.

Number of Epochs: Training a neural network involves adjusting the model's parameters (weights and biases) iteratively to minimize the loss function on the training data. Increasing the number of epochs allows the model to see the training data multiple times, potentially improving its ability to learn complex patterns in the data and converge to a better solution. However, using too many epochs can lead to overfitting, where the model memorizes the training data instead of learning generalizable patterns. Choosing an appropriate number of epochs involves balancing the trade-off between training time and model performance on unseen data.

In [ ]:
```
model.fit(X_train, y_train, epochs = 3)
```

```
Epoch 1/3
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1062 - accurac
y: 0.9673
Epoch 2/3
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0711 - accurac
y: 0.9773
Epoch 3/3
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0534 - accurac
y: 0.9830
```

Out[ ]: <keras.src.callbacks.History at 0x7f19e6177190>

In [ ]:
```
model.save('handwritten.model')
```

In [ ]:
```
model = tf.keras.models.load_model('handwritten.model')
```

In [ ]:
```
loss, accuracy = model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0883 - accuracy:
0.9716
```

In [ ]:
```
print(loss)
print(accuracy)
```

```
0.08826439827680588
0.9715999960899353
```

This function can be used to evaluate reall life hand written images to evaluate the model.

In [ ]:
```
image_number = 1
while os.path.isfile(f"digits/digit{image_number}.png"):
  try:
    img = cv2.imread(f"digits/digit{image_number}.png")[:,:,0]
    img - np.invert(np.array([img]))
    prediction = model.predict(img)
    print("The digit is probably a {np.argmax(prediciton)}")
    plt.imshow(img[0], cmap=plt.cm.binary)
  except:
    print("Error")
  finally:
    image_number+=1
```

Extra

Layers in Neural Networks

1. Input Layer:

- The input layer is the first layer in the neural network.
- It receives the input data and passes it on to the next layer.
- It doesn't perform any computations and simply serves as the entry point for the data.
- In TensorFlow/Keras, the input layer is often implicitly defined when specifying the input shape in the first hidden layer.

2. Hidden Layers:

-Hidden layers are the intermediate layers between the input and output layers.

- Each hidden layer consists of multiple neurons (nodes) that perform computations on the input data.
- The number of hidden layers and neurons in each hidden layer can vary based on the complexity of the problem and the desired model architecture. Each neuron in a hidden layer applies a weighted sum of inputs, followed by an activation function to introduce non-linearity.
- Common activation functions used in hidden layers include ReLU (Rectified Linear Unit), sigmoid, tanh (Hyperbolic Tangent), etc.

3. Output Layer:

- The output layer is the final layer in the neural network.
- It produces the output predictions of the model based on the computations performed in the hidden layers.
- The number of neurons in the output layer depends on the nature of the problem: For binary classification problems, a single neuron with a sigmoid activation function is often used to output a probability score between 0 and 1. For multi-class classification problems, the number of neurons in the output layer corresponds to the number of classes, with each neuron representing the probability of belonging to a particular class (usually softmax activation is used). For regression problems, the output layer typically has a single neuron without any activation function, or a specific activation function based on the range of target values.

Optimizers

1. Stochastic Gradient Descent (SGD): This is the most basic optimization algorithm. It updates the weights in the opposite direction of the gradient of the loss function with respect to the weights. It can have trouble navigating complex loss surfaces and finding the global minimum.

2. Adam (Adaptive Moment Estimation): Adam combines the ideas of Momentum and RMSProp. It adapts the learning rates for each parameter based on estimates of the first and second moments of the gradients. It is computationally efficient and widely used in practice.

3. RMSProp (Root Mean Square Propagation): RMSProp adjusts the learning rates for each parameter based on the average of the recent magnitudes of the gradients for that parameter. It divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight. It helps to alleviate the diminishing learning rates problem in Adagrad.

4. Adagrad (Adaptive Gradient Algorithm): Adagrad adapts the learning rates for each parameter based on the historical gradients for that parameter. It scales the learning

rate of each weight by the inverse of the square root of the sum of the squares of the historical gradients for that weight. It performs well for sparse data.

5. Adadelta: Adadelta is an extension of Adagrad that tries to address its diminishing learning rates problem. It uses a more sophisticated update rule that eliminates the need for an explicit learning rate. It adapts the learning rates based on a moving window of gradient updates.

6. Adamax: Adamax is a variant of Adam that uses the infinity norm of the gradients instead of the L2 norm. It is more robust to the choice of initial learning rate and less affected by vanishing gradients compared to Adam.

7. Nadam (Nesterov-accelerated Adaptive Moment Estimation): Nadam is a variant of Adam that incorporates the Nesterov accelerated gradient (NAG) method. It combines the benefits of Adam and NAG to improve convergence rates and stability.

Loses

1. Mean Squared Error (MSE): MSE is used for regression tasks. It calculates the average of the squared differences between the predicted and actual values. MSE penalizes larger errors more heavily than smaller errors.

2. Binary Cross-Entropy Loss (Binary Crossentropy): Binary cross-entropy loss is used for binary classification tasks. It measures the difference between the probability distributions of the true labels and the predicted probabilities. It is commonly used with sigmoid activation in the output layer.

3. Categorical Cross-Entropy Loss (Categorical Crossentropy): Categorical cross-entropy loss is used for multi-class classification tasks. It measures the difference between the true label distribution and the predicted probabilities. It is commonly used with softmax activation in the output layer.

4. Sparse Categorical Cross-Entropy Loss (Sparse Categorical Crossentropy): Sparse categorical cross-entropy loss is similar to categorical cross-entropy but is used when the true labels are integers (not one-hot encoded). It is commonly used for multi-class classification tasks where the labels are represented as integers.

5. Binary Hinge Loss (Hinge): Hinge loss is used for binary classification tasks, especially in support vector machines (SVMs). It penalizes misclassifications linearly and encourages correct classification with a margin. It is commonly used in SVMs and in neural networks with linear activation in the output layer.

6. Kullback-Leibler Divergence (KLD): KLD is a measure of how one probability distribution diverges from a second, expected probability distribution. It is commonly used in variational autoencoders (VAEs) as a regularization term to encourage the learned distribution to match a predefined distribution.

In [ ]: