DL Practical

Assignment 1

Linear regression by using Deep Neural network: Implement Boston housing price prediction problem by Linear regression using Deep Neural network. Use Boston House price prediction dataset.

```
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(x_train.shape[1],)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(1)  # Output layer (1 neuron, no activation for
regression)
])
```

`model = keras.Sequential([`

- `model` — the **variable name** where we store our neural network.
- `keras.Sequential([...])` —
  - `keras` — **Keras** is a library (part of TensorFlow) that helps you easily build neural networks.
  - `.Sequential()` — a type of model where **layers are stacked one after another**, **in order**.
    ("Sequential" means one after another, step by step.)

`keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),`

- `keras.layers.Dense` —
  - "Dense" means **fully connected layer** (every neuron connects to every neuron in the next layer).
- `64` — the number of **neurons** in this layer.
  (So, this layer has 64 neurons.)
- `activation='relu'` —
  - Each neuron uses the **ReLU activation function**:
    ReLU(x) = max(0, x) — it sets negative numbers to zero.
  - This makes the network **nonlinear**, allowing it to learn complex patterns.
- `input_shape=(X_train.shape[1],)` —
  - This tells the model what **shape** of data it expects.
  - `X_train.shape[1]` means:
    - `X_train` is the training data (features only).
    - `.shape[1]` gives the **number of columns/features**.

- We write it as a **tuple**: `(number of features,)`
- Example: if `X_train` has shape (404, 13), then `X_train.shape[1] = 13`.

So this first layer connects the **input** (the 13 features) to **64 neurons**, each with ReLU.

`keras.layers.Dense(64, activation='relu'),`

- Another Dense layer with:
  - **64 neurons** again
  - **ReLU activation** again
- No `input_shape` here, because Keras already knows from the previous layer.

This adds another hidden layer to make the network deeper (more learning power).

1. Why 64 Neurons?

The **number of neurons** in each layer is a hyperparameter, which means **you can choose** it, but it's not automatically decided. Let's discuss some reasons why **64 neurons** might be a reasonable starting point:

- **More neurons = More capacity to learn**: The more neurons a layer has, the more complex patterns it can learn. However, **too many neurons** can cause overfitting (when the model learns the training data *too well* but doesn't generalize to new data).
- **64 is a good balance**: 64 neurons is a commonly used choice for the number of neurons in hidden layers in many tasks. It's usually a good balance between **learning capacity** and **computational efficiency**. You can always experiment with more or fewer neurons to see what works best for your specific problem.
- **Empirical evidence**: Many people in machine learning choose **powers of 2** (like 32, 64, 128, etc.) because these sizes tend to work well and are computationally efficient in hardware (like GPUs).
- **Tuning is important**: It's often useful to **experiment** with different sizes to find the best performing model. A smaller number might not have enough capacity to capture patterns, while a larger number could lead to overfitting. If you want to fine-tune, try other values like 32, 128, or 256, and see which works best.

2. Why the ReLU Activation Function?

ReLU (Rectified Linear Unit) is one of the most commonly used activation functions in deep neural networks. Let's dive into why it's often the default choice:

Why ReLU?

- **Avoiding vanishing gradients**:
  One of the issues with earlier activation functions (like **sigmoid** or **tanh**) is that they can cause **vanishing gradients** during training. This means when the model is learning, the gradients (or updates to the weights) can become very small and slow down learning,

especially in deep networks. **ReLU** helps avoid this by keeping gradients large and active during the learning process.

- **Simple and fast to compute**:
  ReLU is very computationally simple: it just outputs the input if it's positive, and outputs zero if it's negative. This makes it **efficient** for training.
    - Mathematically, ReLU(x) = max(0, x).
- **Non-linearity**:
  Neural networks need **nonlinear activation functions** (like ReLU) because they allow the model to learn complex, nonlinear relationships in the data. If we only used linear functions, the model would only be able to learn **linear patterns**. ReLU introduces a nonlinearity, which makes it more powerful for tasks like regression and classification.
- **Sparsity**:
  Since ReLU outputs **zero for negative values**, it introduces **sparsity**. That means many neurons will be inactive (output zero) for a lot of inputs, making the model more efficient and focused on only important features.

Are there other activation functions we can use?

Yes! You can definitely experiment with other activation functions. Some popular alternatives are:

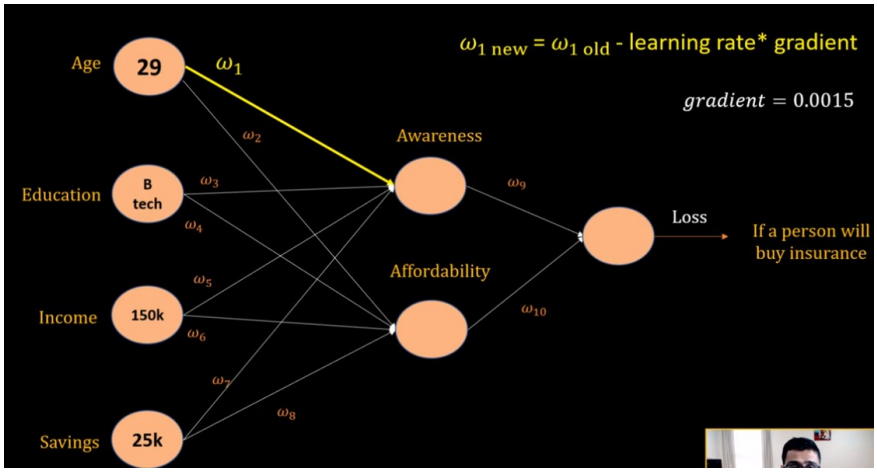
- **Sigmoid**: Outputs values between 0 and 1. It’s often used for binary classification tasks, but it can suffer from the **vanishing gradient problem**, making it harder to train deep networks.
    - Sigmoid(x) = 1 / (1 + exp(-x))
- **Tanh (Hyperbolic Tangent)**: Similar to sigmoid but outputs values between -1 and 1. It also suffers from the vanishing gradient problem and is less commonly used now in deep networks.

Can we use a different activation function or number of neurons?

- **Yes!** The choice of **activation function** and the **number of neurons** are both **hyperparameters** that can be tuned to improve the model’s performance.
    - If you choose a different activation function (like **tanh**, **sigmoid**, or **leaky ReLU**), you might find that it performs better for your particular dataset. The same goes for **changing the number of neurons**.
    - You can try out several options using **hyperparameter tuning** (e.g., grid search or random search) to find the best combination.

Summary:

- **64 neurons** is a reasonable starting point, balancing learning capacity and efficiency. You can experiment with different numbers, but starting with 64 is often good.
- **ReLU** is popular because it’s computationally efficient, helps with training deep networks by avoiding vanishing gradients, and introduces non-linearity. However, there are alternatives like **Leaky ReLU**, **Tanh**, and **Swish**, which you can try to see if they work better for your task.



What is the Vanishing Gradient Problem?

In neural networks, when we train a model using **backpropagation**, we update the weights of the network to minimize the error (loss). This happens through a process that involves calculating **gradients** — basically, how much each weight should be adjusted to reduce the error.

The **vanishing gradient problem** happens when these gradients become **very small** (close to zero) as they move backward through the network. When gradients become too small, the weights stop updating effectively, and the model stops learning, especially in deeper layers.

This is a **big issue** in deep neural networks because it prevents them from learning complex patterns in the data.

How does it happen?

The problem usually happens with **certain activation functions**, like **sigmoid** or **tanh**, because of the way they behave with large inputs or outputs. Let's see why this happens:

1. Sigmoid Activation Function:

The **sigmoid function** squashes inputs to the range between **0 and 1**. Its formula is:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- When the input value **x** is **very large** (positive or negative), the sigmoid output becomes **very close to 1 or 0**, and **the derivative of sigmoid** (how much the output changes with respect to the input) becomes **very small**.
- This means that when we calculate the gradients during backpropagation, they become **tiny** (close to zero). So, when the gradients are backpropagated to earlier layers of the network, they're **almost zero**, making it hard for those layers to learn.

Example:

Let's say we have an input value **x = 1000**:

$$\text{Sigmoid}(1000) \approx 1 \quad \text{(very close to 1)}$$

Now, the gradient of sigmoid is:

$$\frac{d}{dx}\text{Sigmoid}(x) = \text{Sigmoid}(x) \times (1 - \text{Sigmoid}(x)) = 1 \times (1 - 1) = 0$$

- So, the gradient here is **0**, which means no weight updates are made in that layer. This is a problem because the weights are not updated, and the model can't learn from that layer.

In Deep Networks:

If you have many layers, this problem becomes even worse because the gradient keeps **shrinking** as it propagates backward through each layer. By the time it reaches the earlier layers, the gradient is so small that the weights are barely updated, and the learning process stops.

Example with Multiple Layers:

Let's say you have a network with **3 layers** using the sigmoid activation function. During backpropagation, the gradients calculated in each layer might be something like this:

- **Layer 3 gradient**: 0.1
- **Layer 2 gradient**: 0.05
- **Layer 1 gradient**: 0.0001

As you can see, the gradients get **smaller and smaller** as you move backward through the layers. By the time you reach the earlier layers, the gradient is so small that the weights don't update properly, and the model **stops learning**.

How do we fix the Vanishing Gradient Problem?

There are several ways to address this issue:

1. **Use ReLU (Rectified Linear Unit)**:
    - Unlike sigmoid, **ReLU** doesn't squash values into a small range (0,1). Instead, it simply outputs the input if it's positive, or 0 if it's negative.
    - This keeps the gradients from becoming too small, helping the network learn better.
2. **Use Leaky ReLU** or **ELU** (Exponential Linear Units):
    - These are variations of ReLU that allow a small gradient even for negative inputs (leaky ReLU) or a smoother, continuous gradient (ELU).
3. **Weight Initialization**:
    - Properly initializing weights before training can help prevent the gradients from getting too small in the first place. Methods like **Xavier initialization** or **He initialization** are commonly used to prevent this.
4. **Batch Normalization**:
    - This technique normalizes the inputs of each layer, helping to keep the gradients more stable and reducing the vanishing gradient problem.

```
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

Summary of What Each Part Does:

Purpose: This is a method that configures the model for training. It specifies how the model should be trained, what optimization algorithm to use, how to measure performance, and what kind of loss function to minimize during training.

- **`optimizer='adam'`**: Specifies the use of the Adam optimizer, which adjusts the learning rate automatically for better training performance. In deep learning, optimizers play a critical role in adjusting the weights of the model during training to minimize the loss function.

  Adam is particularly effective because it combines the benefits of momentum and adaptive learning rates, making it ideal for training models like **deep neural networks** (DNNs) but also a good fit for simpler models like **linear regression**

- **`loss='mse'`**: Specifies that we want to minimize **Mean Squared Error** (MSE), a common loss function for regression tasks.

  For Regression: MSE, MAE, and Huber are the most common loss functions.

- **`metrics=['mae']`**: Specifies that we want to track **Mean Absolute Error (MAE)** during training to understand how well the model is predicting.

Summary Table of Optimizers:

Optimizer	Key Features	Pros	Cons
SGD	Basic, no momentum	Simple and easy to implement	Slow, can be sensitive to learning rate
Momentum	Adds momentum to gradients	Accelerates convergence	Can overshoot if learning rate is too high
Nesterov	Momentum with "lookahead"	Faster convergence than regular momentum	More computation required
Adagrad	Adapts learning rate based on past gradients	Good for sparse data	Learning rate may become too small
RMSprop	Adaptive learning rate with moving average	Solves Adagrad's problem	Needs tuning of decay factor
Adam	Combines momentum and RMSprop	Works well in practice, less tuning needed	Can be computationally intensive
AdaMax	Adam variant using infinity norm	More stable in certain cases	More computationally expensive
Nadam	Adam with Nesterov momentum	Good convergence, effective for deep networks	More computation than Adam

Mean Squared Error (MSE):

- Formula:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **Explanation**: This is the most commonly used loss function for regression tasks. It calculates the squared difference between the predicted value ($\hat{y}_i$) and the actual value (y_i), then averages this squared error over all data points.
- **Use case**: Predicting continuous values (e.g., house prices, temperature, etc.).

Mean Absolute Error (MAE):

- Formula:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

- **Explanation**: MAE calculates the average of the absolute differences between the actual and predicted values. Unlike MSE, which penalizes larger errors more due to squaring the differences, MAE treats all errors equally.
- **Use case**: Used when you want to treat all errors equally, regardless of size, and when you want the model to be less sensitive to outliers.

```
history = model.fit(x_train_scaled, y_train, epochs=100, validation_split=0.2,
verbose=1)
```

1. `history`:

- **What it means**: `history` is the variable that will store the result of the `fit` method. It contains information about how the model performed during training, including metrics such as loss and accuracy at each epoch.
- **Why it's used**: After training, you can use the `history` object to analyze the model's performance (for example, to plot the loss or accuracy over time). This allows you to visualize how well the model is learning during training.

2. `model.fit`:

- **What it means**: The `fit` method in Keras is used to train the model on the provided data.
- **Why it's used**: It takes the training data and trains the model on it for a set number of iterations (epochs). The model will learn from the input data (`x_train_scaled`) and try to minimize the loss function during the training process.

3. `x_train_scaled`:

- **What it means**: `x_train_scaled` is the input training data that has already been processed or scaled. "Scaled" means that the features in `x_train` have been normalized

(usually scaled to a certain range or distribution, like between 0 and 1) to help the model train more effectively.

- **Why it's used**: Scaling the input features ensures that all features contribute equally to the learning process. Without scaling, some features with larger values could dominate the learning process, leading to poor performance.

4. `y_train`:

- **What it means**: `y_train` is the corresponding target data (labels) for the training set.
- **Why it's used**: It represents the actual values that the model is trying to predict. In a regression task, for example, this would be the actual house prices the model is trying to predict based on the features in `x_train_scaled`.

5. `epochs=100`:

- **What it means**: `epochs` refers to the number of times the model will cycle through the entire training dataset.
- **Why it's used**: An epoch is one complete pass over the training data. By setting `epochs=100`, you are telling the model to go through the training data 100 times. More epochs generally lead to better learning (up to a point), but also increases the risk of overfitting if the model sees the data too many times.

6. `validation_split=0.2`:

- **What it means**: The `validation_split` parameter defines the fraction of the training data to be set aside for validating the model's performance during training.
- **Why it's used**: By setting `validation_split=0.2`, you're telling Keras to use 20% of the training data as validation data. This helps track the model's performance on data it hasn't seen before, allowing us to detect overfitting and assess generalization. The remaining 80% of the data will be used to train the model.

7. `verbose=1`:

- **What it means**: `verbose` controls the level of output that is printed during training.
- **Why it's used**:
  - `verbose=0`: No output (silent mode).
  - `verbose=1`: Progress bar for each epoch (default setting).
  - `verbose=2`: One line per epoch (less detailed but still gives progress information).
- Setting `verbose=1` will print a progress bar showing how the model is doing during each epoch of training, along with the loss and any metrics you've specified (like accuracy or MAE).

```
loss, mae = model.evaluate(x_test_scaled, y_test, verbose=0)
print(f"\nMean Absolute Error on test data: ${mae * 1000:.2f}")
```

`loss, mae = model.evaluate(x_test_scaled, y_test, verbose=2):`

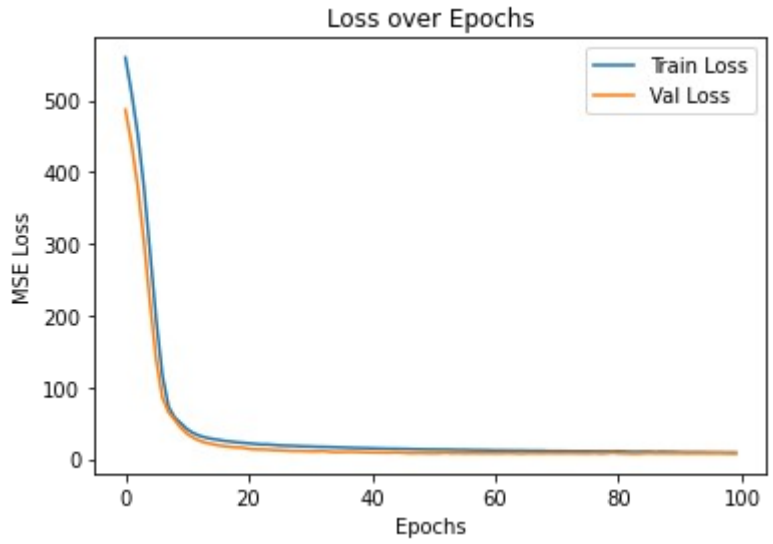
- **What it means**: The `evaluate` function returns two values:
  - **`loss`**: The computed loss value (for regression tasks, this is typically the Mean Squared Error or MSE, depending on what you set when compiling the model).
  - **`mae`**: The computed Mean Absolute Error (MAE), which is a metric specified during model compilation.
- The results are stored in the variables `loss` and `mae`. `loss` represents the overall error (e.g., MSE), and `mae` represents the Mean Absolute Error (which tells us the average absolute difference between predicted and true values).

In a typical training process, you have two things:

1. **Training Loss**: The error (e.g., how far off the predictions are from the actual values) when the model is trained on the training data.
2. **Validation Loss**: The error when the model is tested on a separate set of data that it has **never seen before** (this is the validation set).

What to Look For:

- **Decreasing Loss**: If both the training loss and validation loss are decreasing steadily, it suggests that the model is learning and generalizing well.
- **Overfitting**: If the training loss keeps decreasing but the validation loss starts increasing after a certain point, this may indicate overfitting. The model is doing well on the training data but is not generalizing well to the validation data.



Assignment 2

Multiclass classification using Deep Neural Networks: Example: Use the OCR letter recognition dataset https://archive.ics.uci.edu/ml/datasets/letter+recognition

What is Multiclass Classification?

Multiclass classification is a type of machine learning problem where the goal is to classify inputs into more than two categories (or classes).

In Simple Terms:

• Binary classification: Only two possible outcomes (e.g., Spam or Not Spam).

• Multiclass classification: More than two possible outcomes.

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.datasets import mnist
```

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

Example:

- **`x_train`**: This will be a 4D NumPy array with the shape `(60000, 28, 28)`. This means there are 60,000 images, each of size 28x28 pixels.
- **`y_train`**: This will be a 1D NumPy array with the shape `(60000,)`. Each element is an integer between 0 and 9, representing the label (or class) of the corresponding image in `x_train`.
- **`x_test`**: This will be a 4D NumPy array with the shape `(10000, 28, 28)`. This means there are 10,000 test images, each of size 28x28 pixels.
- **`y_test`**: This will be a 1D NumPy array with the shape `(10000,)`. Each element is an integer between 0 and 9, representing the label (or class) of the corresponding image in `x_test`.

```
# Normalize the image data to values between 0 and 1
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255
```

1. Reshaping the data (`reshape`):

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

- **`reshape`** is used to change the shape of the data. The original dataset (`x_train` and `x_test`) contains images in the MNIST dataset is 28x28 pixels, so each image has the shape `(28, 28)`.
- **`reshape(60000, 784)`** and **`reshape(10000, 784)`** are transforming the 28x28 pixel images into 1D arrays (flat vectors) of length 784 (since 28 * 28 = 784). This is necessary because neural networks typically expect input data to be in a flat format (1D vectors) rather than 2D image matrices.
    - `x_train` has 60,000 images, so after reshaping, it will be a 2D array with dimensions `(60000, 784)`.
    - `x_test` has 10,000 images, so after reshaping, it will be a 2D array with dimensions `(10000, 784)`.

2. Converting to `float32` type (`astype('float32')`):

```
.astype('float32')
```

- **`astype('float32')`** converts the image pixel values into 32-bit floating point numbers. This is done because neural networks typically work with floating point numbers rather than integers. Converting the image data to `float32` ensures that the calculations during the training process (such as gradient descent) are done with floating-point precision.

3. Normalizing the pixel values:

```
/ 255
```

- **Dividing by 255**: The pixel values in the MNIST dataset range from 0 to 255 (since each pixel in an image is an 8-bit value). To make the data more suitable for the neural network, we **normalize** the pixel values so they lie between 0 and 1.
- By dividing by 255, each pixel value will be a floating-point number between 0 (for black) and 1 (for white), instead of being an integer between 0 and 255.

Example:

- If a pixel has a value of 128, after dividing by 255, the new value will be approximately `0.50196` (which is closer to the middle value between black and white).
- If a pixel has a value of 255 (white), after dividing by 255, the new value will be `1.0`.
- If a pixel has a value of 0 (black), after dividing by 255, the new value will be `0.0`.

Why is Normalization Important?

Normalization helps the training process in several ways:

- **Faster convergence:** Neural networks train faster when the input data is on a similar scale, typically between 0 and 1. Large values (like 0-255) could slow down training.
- **Stable gradients:** When the inputs are normalized, the gradients during backpropagation (the process of updating weights) are more stable, preventing very large or very small gradients that could cause the network to train inefficiently or fail to converge.

```
# One-hot encoding for labels
y_train = np.eye(10)[y_train]
y_test = np.eye(10)[y_test]
```

What is One-Hot Encoding?

One-hot encoding is a way of representing labels (like categories or classes) as a series of 1s and 0s, where only one position (corresponding to the class) is marked as `1` and the rest are `0`.

For example:

- If the label is `3`, it will be represented as `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`.
- If the label is `5`, it will be represented as `[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]`.

This makes it easier for the machine learning model to recognize which class the data belongs to.

Let's break down this code:

```
# One-hot encoding for labels
y_train = np.eye(10)[y_train]
y_test = np.eye(10)[y_test]
```

1. What does `np.eye(10)` do?

- `np.eye(10)` creates a **10x10 identity matrix**, which looks like this:

  ```
  [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
  ```

  This identity matrix represents **one-hot vectors** for the digits `0` to `9`. Each row corresponds to a digit, and the 1 in each row represents that digit.

  For example:

  - The first row `[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]` is the one-hot encoding for `0`.

    - The second row `[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]` is the one-hot encoding for `1`.
    - And so on...

2. What does `[y_train]` do?

- `y_train` contains the labels of the training data (numbers between `0` and `9`).
- When you do `np.eye(10)[y_train]`, you're using the values in `y_train` as **indexes** to pick the rows from the identity matrix (`np.eye(10)`). For example:
    - If `y_train = [3, 1, 5]`, `np.eye(10)[y_train]` picks the 4th row (for label `3`), the 2nd row (for label `1`), and the 6th row (for label `5`).

3. The result:

After this operation, `y_train` will be a **list of one-hot encoded labels**. For example:

- If the first 3 values of `y_train` were `[3, 1, 5]`, then `y_train` will now look like this:

```
y_train = [[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  # One-hot for label '3'
           [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  # One-hot for label '1'
           [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]]  # One-hot for label '5'
```

The same thing happens for `y_test` as well.

Why do we do this?

- **Machine Learning Models** need the labels in a format where each class is treated separately. The model should know that class `0` is different from class `1`, and so on.
- One-hot encoding helps the model by giving it a clear and distinct representation for each possible label.

```
model = Sequential([
    Dense(512, activation='relu', input_shape=(784,)),
    Dropout(0.2),
    Dense(512, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])
```

1. `model = Sequential([...])`

- **`Sequential`**: This is the model type being used here. A **Sequential** model is a linear stack of layers where each layer has exactly one input and one output.
- Inside the `Sequential()` function, we define the layers of the model. The layers are stacked on top of each other, and the output of one layer becomes the input for the next.

2. `Dense(512, activation='relu', input_shape=(784,))`

- **`Dense`**: This defines a **fully connected layer**. In a Dense layer, each neuron is connected to every neuron in the previous layer.
    - **`512`**: The number `512` represents the number of **neurons** (or units) in this layer. So, there will be 512 neurons in this layer.
    - **`activation='relu'`**: This is the **activation function** used in the layer. `relu` stands for **Rectified Linear Unit**, and it's a function that replaces any negative values with `0` and leaves positive values unchanged. It's commonly used in hidden layers because it helps the model learn quickly and prevents issues like vanishing gradients.
    - **`input_shape=(784,)`**: This specifies the shape of the input data that the layer expects. The input is a **1D array of size 784** (which is typical for images in the MNIST dataset). Each image in MNIST is 28x28 pixels, and when flattened, it becomes a 784-dimensional vector (28 * 28 = 784). This tells the first layer of the network how many features it should expect from the input.

3. `Dropout(0.2)`

- **`Dropout`**: This is a regularization technique used to prevent **overfitting** during training.
    - **`0.2`**: The argument `0.2` means that 20% of the neurons in this layer will be randomly "dropped" (set to zero) during each training step. This helps the model generalize better and prevents it from becoming too reliant on any particular neuron.
    - Dropout helps prevent the model from memorizing the training data too well (overfitting) and encourages it to learn more general patterns.

4. `Dense(512, activation='relu')`

- Another **Dense layer** with **512 neurons** and **ReLU activation**.
    - This layer doesn't need to specify an input shape, because it automatically takes input from the previous layer.
    - Again, we are using **ReLU** activation for this hidden layer, which helps in the learning process by introducing non-linearity into the model.

5. `Dropout(0.2)`

- Another **Dropout layer** with the same dropout rate of **20%**.
    - This layer randomly drops 20% of neurons again to help with generalization and prevent overfitting.

6. `Dense(10, activation='softmax')`

- **`Dense(10)`**: This is the final **output layer**. It has **10 neurons**, one for each possible digit (0-9) in the MNIST dataset.
    - This layer will output a vector of 10 values representing the model's predictions for each class (digit). Each value will correspond to the probability of the input belonging to one of the 10 digits.
- **`activation='softmax'`**: The **softmax activation function** is used in the output layer for multi-class classification problems. It converts the raw output scores (logits) into probabilities by squashing them between 0 and 1, such that all the probabilities sum up to 1. It's commonly used when the task involves classifying data into more than two categories (multi-class classification).

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

1. `model.compile(...)`

- **`compile()`** is a method in Keras used to configure the model for training.
    - It sets up how the model should be trained by specifying three main things:
        - **Loss function**: A function that measures how well the model is performing.
        - **Optimizer**: The algorithm that adjusts the model's weights based on the loss function.
        - **Metrics**: A list of metrics that will be used to evaluate the model's performance during training.

2. `loss='categorical_crossentropy'`

- **Loss Function**: The loss function defines how the model's errors will be computed. It measures the difference between the model's predictions and the true values, and the goal of training is to minimize this loss.
    - **`'categorical_crossentropy'`**: This is the loss function used for **multi-class classification problems**. In this case, since we are working with the MNIST dataset, where each image corresponds to one of 10 classes (digits 0-9), this loss function is appropriate. It computes the error between the predicted probabilities (from the `softmax` activation in the output layer) and the actual labels (which are one-hot encoded).
        - **Why categorical cross-entropy?**: In a multi-class classification problem like MNIST, the model's output is a probability distribution (after applying softmax) over the 10 possible classes (digits). Categorical cross-entropy is used to compare these predicted probabilities with the true one-hot encoded labels. The model will try to minimize this loss during training to make its predictions as accurate as possible.

3. `optimizer=RMSprop()`

- **Optimizer**: The optimizer is responsible for updating the weights of the model during training in an attempt to minimize the loss function.
    - **`RMSprop()`**: This is a specific type of optimizer used here. **RMSprop** stands for **Root Mean Square Propagation**, and it is an adaptive learning rate optimization algorithm. It adjusts the learning rate during training based on the average of recent gradients.
        - **Why RMSprop?**: RMSprop is commonly used for training deep learning models. It helps the model converge faster and prevents issues like oscillating gradients or overshooting, especially when training on datasets with noisy or sparse gradients.
        - **How RMSprop works**: It computes the running average of the squared gradients and uses this to scale the learning rate for each parameter. This makes RMSprop adapt the learning rate dynamically and helps with faster and more stable convergence.

4. `metrics=['accuracy']`

- **Metrics**: These are the evaluation criteria used to assess the model's performance during training and testing. The model will track these metrics during each training epoch.
    - **`'accuracy'`**: This metric measures the percentage of correct predictions the model makes. It’s the most common metric used for classification tasks, and it tells us how often the model's predictions match the true labels.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

```
model.fit(x_train, y_train, batch_size=128, epochs=20, validation_data=(x_test, y_test))
```

1. `model.fit(...)`

- **`fit()`** is a method in Keras used to train the model.
    - It takes the training data (`x_train` and `y_train`), runs the model on this data for a specified number of iterations (epochs), and updates the weights of the model based on the optimizer and loss function.
    - This method performs the learning process, adjusting the model's parameters to minimize the loss function (as we previously defined when compiling the model).

2. `x_train`

- **`x_train`** is the input data that will be fed to the model during training.
    - In the case of MNIST, `x_train` is a set of images (the pixel values of 60,000 images of handwritten digits).
    - The model will process these images and try to predict which digit (from 0 to 9) each image represents.

3. `y_train`

- **`y_train`** is the target or ground truth labels corresponding to each image in the training data (`x_train`).
    - In MNIST, each image corresponds to a digit (0-9). So, `y_train` contains the actual labels, such as 0, 1, 2, ..., 9 for each image.
    - These labels will be compared to the model's predictions during training, and the difference will be used to update the model's weights.

4. `batch_size=128`

- **`batch_size`** specifies the number of samples that will be processed before the model's internal parameters (weights) are updated.
    - **What is a batch?**: Instead of feeding all 60,000 images at once, we divide them into smaller groups called **batches**. For each batch, the model computes the gradients (the derivative of the loss function with respect to the model's parameters) and updates the weights.
    - **Why 128?**: Here, the batch size is set to 128. This means that the model will use 128 images to perform one step of the training process before updating the weights.
    - **Why use batches?**:
        - Training on the entire dataset at once (called "batch training") is computationally expensive and can be slow.

- Using smaller batches allows for more frequent updates, which can make training faster and more efficient.
- It also helps avoid getting stuck in bad local minima in the loss function.

5. `epochs=20`

- **`epochs`** specifies the number of times the entire training dataset will be passed through the model.
    - **What is an epoch?**: An epoch refers to one complete pass through all the training samples. If there are 60,000 images in the training set and a batch size of 128, during one epoch, the model will see and process all 60,000 images.
    - **Why 20 epochs?**: Here, the model will be trained for 20 epochs. The number of epochs determines how many times the model will learn from the training data. More epochs usually lead to better learning, but there is a point where the model starts to overfit, meaning it learns the training data too well and doesn't generalize well to new data.

6. `validation_data=(x_test, y_test)`

- **`validation_data`** is used to specify the validation dataset, which the model will use to evaluate its performance after each epoch.
    - **`x_test`**: This is the set of input data for the validation (in this case, 10,000 test images from the MNIST dataset).
    - **`y_test`**: These are the true labels corresponding to the test images (`x_test`).
    - The model will make predictions on `x_test` after each epoch, and the loss and accuracy will be calculated for the validation data separately.
    - The validation set is **not used** to update the model's weights; it's only used to evaluate how well the model is performing on data it hasn't seen before. This helps you track if the model is overfitting (getting too good at predicting the training data but not generalizing well to new data).

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

1. `score = model.evaluate(x_test, y_test, verbose=0)`

- **`model.evaluate()`**: This is a Keras method used to evaluate the performance of a trained model on a test dataset.
    - The **model** uses the test data (`x_test` and `y_test`) to make predictions and compare them with the true labels (`y_test`).
    - The method returns the **loss** and any other metrics defined during the `compile()` stage (like **accuracy**) on the test dataset.

- **`x_test`**: This is the input data used for testing. In the MNIST dataset, it consists of 10,000 images that were not used during training.
- **`y_test`**: This is the true labels for the `x_test` data, which the model will compare its predictions against.
- **`verbose=0`**: This controls the level of output shown during the evaluation.
    - **`verbose=0`**: No output will be shown during the evaluation process.
    - If you use **`verbose=1`**, Keras will display a progress bar.
    - If you use **`verbose=2`**, it will print out the evaluation process in a more detailed format, including the batch-wise progress.

Assignment 3

```
# Reshape the data (28x28 images to a flat 1D vector of 784 values)
x_train = x_train.reshape(-1, 28, 28, 1) / 255
x_test = x_test.reshape(-1, 28, 28, 1) / 255
```

`x_train = x_train.reshape(-1, 28, 28, 1) / 255`:

- **`x_train`**: This is the variable containing the training dataset, typically a 2D array where each row corresponds to an image (in this case, each image is 28x28 pixels).
- **`reshape(-1, 28, 28, 1)`**:
    - **`reshape`** is a method from the NumPy library used to change the shape of an array.
    - **`-1`**: This is a special value that allows NumPy to automatically infer the size of this dimension based on the size of the array and other given dimensions. It means "keep whatever number of images there are" (the first dimension will be the number of images).
    - **`28, 28`**: These represent the height and width of each image, which are 28 pixels by 28 pixels.
    - **`1`**: This represents the number of channels. Since the images are grayscale, there is only one channel (hence the `1`). If the images were RGB (color), there would be 3 channels (Red, Green, Blue).

  So, `x_train.reshape(-1, 28, 28, 1)` reshapes the data so that each image in `x_train` will have the shape `(28, 28, 1)` — a 28x28 pixel image with 1 channel (grayscale). The `-1` tells NumPy to infer the number of images (examples) in the dataset.
- **`/ 255`**:
    - This divides each pixel value by 255.

- Pixel values in an image are typically between `0` and `255`, where `0` represents black and `255` represents white in grayscale. Dividing by `255` normalizes the pixel values to be between `0.0` and `1.0`.
- **Why normalize?**: Neural networks work better when the input values are normalized, typically in the range of `[0, 1]`. This helps with faster convergence and more stable training.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # 10 classes (fashion categories)
])
```

First Layer: Conv2D Layer

`First Layer: Conv2D Layer`

```
Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1))
```

- **`Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1))`**:
  - **`32`**: This is the number of filters (or kernels) in the convolutional layer. Each filter learns different features of the input data (such as edges, textures, etc.).
  - **`kernel_size=(3, 3)`**: The size of the filter (3x3 pixels). This means each filter will look at a 3x3 section of the input image at a time.
  - **`activation='relu'`**: ReLU (Rectified Linear Unit) is the activation function applied to the output of each neuron. ReLU replaces all negative values with zero and keeps positive values unchanged. It's commonly used in CNNs due to its simplicity and efficiency.
  - **`input_shape=(28, 28, 1)`**: This specifies the shape of the input data. For image data:
    - `28x28` is the size of the image (28 pixels wide and 28 pixels tall).
    - `1` indicates that the image is grayscale (one channel). If it were a colored image (RGB), the value would be `3` instead.

Second Layer: MaxPooling2D Layer

```
MaxPooling2D(pool_size=(2, 2))
```

- **`MaxPooling2D(pool_size=(2, 2))`**:
    - This performs max-pooling, which downsamples the image by taking the maximum value from each 2x2 section of the image. This reduces the image's width and height while keeping important features intact.
    - **`pool_size=(2, 2)`**: The size of the pooling window (2x2). The image will be downsampled by a factor of 2 along both axes (height and width).

Third Layer: Conv2D Layer

```
Conv2D(64, kernel_size=(3, 3), activation='relu')
```

- **`Conv2D(64, kernel_size=(3, 3), activation='relu')`**:
    - **`64`**: The number of filters (64 filters in this layer, each learning a different set of features).
    - **`kernel_size=(3, 3)`**: The size of each filter (3x3 pixels).
    - **`activation='relu'`**: The ReLU activation function is applied to this layer's output.

Fourth Layer: MaxPooling2D Layer

```
MaxPooling2D(pool_size=(2, 2))
```

- This is similar to the previous MaxPooling2D layer, performing downsampling on the output of the previous Conv2D layer with a 2x2 pooling window.

Fifth Layer: Flatten Layer

```
Flatten()
```

- **`Flatten()`**: This layer flattens the 2D output (which has height and width) into a 1D vector. After convolution and pooling, the output is usually in the form of a 2D matrix (e.g., height x width x channels), but fully connected layers (like `Dense`) require a 1D vector as input. Flatten transforms the multi-dimensional data into a 1D vector, which can then be passed to the Dense layers.

Sixth Layer: Dense Layer

```
Dense(128, activation='relu')
```

- **`Dense(128, activation='relu')`**:
    - **`128`**: This specifies that there are 128 neurons in this fully connected layer. Each of these neurons is connected to every neuron in the previous layer (the flattened output).

- **`activation='relu'`**: The ReLU activation function is applied to the output of the layer. Each neuron outputs the maximum of zero and the input value (i.e., `max(0, x)`).

Seventh Layer: Dense Layer (Output Layer)

```
Dense(10, activation='softmax')  # 10 classes (fashion categories)
```

- **`Dense(10, activation='softmax')`**:
  - **`10`**: The number of neurons in this layer represents the number of classes in the classification problem. In this case, it's 10 classes (such as the 10 fashion categories in the Fashion MNIST dataset).
  - **`activation='softmax'`**: The `softmax` activation function is used in multi-class classification problems. It converts the output of the layer into a probability distribution over all classes, where the sum of the probabilities equals 1. Each output neuron corresponds to the probability of the input image belonging to one of the 10 classes.

4. Model Compilation (not shown here, but typically follows this):

After defining the model, we typically compile it using an optimizer, a loss function, and metrics to track. Example:

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

- **`optimizer='adam'`**: Adam is an adaptive optimizer that adjusts learning rates during training to improve convergence.
- **`loss='sparse_categorical_crossentropy'`**: This is the loss function for multi-class classification problems. It measures how well the predicted probabilities match the actual class labels.
- **`metrics=['accuracy']`**: This specifies that accuracy should be tracked during training and testing.