6CS012 - Artificial Intelligence and Machine Learning. Building Fully Connected Neural Networks for Devnagari Handwritten Digit Classification.

Prepared By: Siman Giri {Module Leader - 6CS012}

March 16, 2025

Worksheet - 4.

1 Instructions

This worksheet attempts to teach you how to build a Fully Connected Neural Network for performinng Multiclass Classification of Devnagari Handwritten Digits, with keras.

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook {Preferred}.

Learning Objectives:

By the end of this tutorial, learners should be able to effectively train, evaluate, save, and reuse deep learning models, providing a solid foundation for real-world machine learning projects.

- **Understand Model Compilation and Training:**
- **Evaluate and Test Model Performance:**
- **Make Predictions and Interpret Results:**

2 Introduction to Keras.

Keras is a high-level API for building and training deep learning models. It runs on top of other deep learning frameworks such as TensorFlow, Theano, and CNTK, and provides a simple interface for designing complex neural networks. Here, we'll use Keras with TensorFlow as the backend. To begin, install Keras and TensorFlow:

Installing Keras.

```
pip install keras tensorflow
```

Already comes in built with Colab.

Identifying the version of Keras.

```
import tensorflow as tf
print(tf.keras.__version__)
```

2.1 Why not Just Use Numpy?

1. NumPy Lacks Automatic Differentiation:

Neural Networks rely on backpropagation to compute gradients for optimization. Numpy does not support automatic differentiation.

Manual Gradient Calculation in Numpy

```
import numpy as np
# Simple function f(x) = x^2
def f(x):
    return x ** 2
# Manual derivative (f'(x) = 2x)
def gradient(x):
    return 2 * x
# Update rule: x = x - learning_rate * gradient
x = 5.0
learning_rate = 0.1
for _ in range(10): # Manually optimize for 10 steps
    x -= learning_rate * gradient(x)
    print(f"x: {x}, f(x): {f(x)}")
```

- ✓Works for Simple cases
- ✗ Hard to scale for deeper networks

Gradient Computations with Keras.

```
import tensorflow as tf
x = tf.Variable(5.0) # Trainable variable
with tf.GradientTape() as tape:
    y = x ** 2 # y = x^2
grad = tape.gradient(y, x) # Computes dy/dx automatically
print(grad.numpy()) # Output: 10.0
```

- ✓With keras, we don't need to compute gradients manually.{Thus No Manual gradient calculations needed!} ✓Scales to large networks.

2. Numpy is Slow for Large Models:

- Numpy runs only on CPU and does not support GPU acceleration.
- Deep Learning requires millions of matrix operations, which Numpy alone can not optimize efficiently.

Example: Matrix Multiplication Speed (Numpy vs. Tensorflow on GPU).

```
import numpy as np
import tensorflow as tf
import time
# Create large random matrices
size = (1000, 1000)
A = np.random.rand(*size)
B = np.random.rand(*size)
# NumPy Multiplication
start = time.time()
C_numpy = np.dot(A, B)
print("NumPy Time:", time.time() - start)
# TensorFlow Multiplication (for colab uses GPU Runtime if available)
A_tf = tf.constant(A)
B_tf = tf.constant(B)

start = time.time()
C_tf = tf.matmul(A_tf, B_tf)
print("TensorFlow Time:", time.time() - start)
```

3. No Pre-built Activation Functions and Layers in NumPy.

In deep learning, we use functions like:

- **sigmoid:**
- **Softmax:**

With Numpy we'd have to manually implement every function. In keras it already has built it.

Implementation of Activation Function with Keras.

```
from tensorflow.keras.layers import Dense, sigmoid
layer = Dense(64, activation='sigmoid')
```

4. Keras supports Model Training, Numpy Does not:

Once your model is built, training it in Numpy is complex:

- Compute forward pass.
- Compute loss.
- Compute backward pass manually.
- Update weights using gradients.

Manually Training Network in Numpy.

```
for epoch in range(10):
    # Forward pass
    y_pred = np.dot(x_train, weights)
    # Compute loss
    loss = np.mean((y_pred - y_train) ** 2)
    # Compute gradients manually
    gradients = 2 * np.dot(x_train.T, (y_pred - y_train)) / len(x_train)
    # Update weights
    weights -= learning_rate * gradients
```

- ✗ Difficult to debug.
- ✗ Does not scale well.
- ✗ No automatic gradient computation.

keras training is One Line.

```
    model.fit(x_train, y_train, epochs=10, batch_size=32)
```

✓Easy and Efficient.

5. No Prebuilt Layers in Numpy:

In keras, we can easily define layers like:

Layers in Keras.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(10, activation='softmax')
])
```

✓With Numpy, we'd need to implement each layer manually!

6. No prebuilt Optimizers in NumPy:

Deep learning requires optimization algorithms like:

- SGD (Stochastic Gradient Descent) or Mini - Batch Gradient Descent.
- Adam (Adaptive Momentum Estimation){ we will discuss this in upcoming weeks.}

Implementing SGD in Numpy.

```
learning_rate = 0.01
weights = np.random.randn(3, 3)
for _ in range(100): # Training loop
    gradient = np.random.randn(3, 3) # Fake gradient for illustration
    weights -= learning_rate * gradient
```

✓Works, but hard to scale for complex models.

In Keras.

```
from tensorflow.keras.optimizers import SGD
optimizer = SGD(learning_rate=0.01)
```

Lesson Learned:

- Can you build a neural network with NumPy? Yes!
- Should you?
  - Not if you want efficiency, scalability, and easy debugging!

3 Understanding Fully Connected Layers.

A fully connected (dense) layer means that every input node is connected to every node in the next layer. This is a fundamental building block of neural networks. In Keras, this is implemented using the Dense class.

1. Dense Class in Keras:

The Dense lass in keras is used to create a **fully connected layers also called dense layers** in a neural network. Each neuron in a Dense layer is connected to every neuron in the previous layer.

Suntax of Dense Layer.

```
from tensorflow.keras.layers import Dense
layer = Dense(units, activation=None, use_bias=True, kernel_initializer="glorot_uniform")
```

Main Arguments:

Argument	Description
`units`	Number of neurons in the layer.
`activation`	Activation function applied to the neurons. Default: `None` (linear activation).
`use_bias`	Whether to include a bias term. Default: `True`.
`kernel_initializer`	Method to initialize the weights. Default: `"glorot_uniform"`.
`bias_initializer`	Method to initialize the bias. Default: `"zeros"`.
`kernel_regularizer`	Regularization applied to the weights (e.g., L1, L2).
`bias_regularizer`	Regularization applied to the bias.

Table 1: Arguments of the `Dense` Layer in Keras

2. How Does Dense Work ?

Each neuron in a Dense layer computes:

$$y = \text{activation}(Wx + b) \tag{1}$$

where:

- x = Input data (vector or matrix).

- W = Weights matrix (learnable parameters).
- b = Bias term (optional).
- activation = Activation function applied to the output.

Some Common Activation Function:

Activation	**Function**
ReLU (`relu`)	$f(x) = \max(0, x)$ (Good for hidden layers)
Sigmoid (`sigmoid`)	$f(x) = \frac{1}{1+e^{-x}}$ (Used for binary classification)
Softmax (`softmax`)	Converts output into probabilities (Used for multi-class classification)
Tanh (`tanh`)	$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (Outputs values between -1 and 1)

Table 2: Common Activation Functions for Dense Layers.

{`We will discuss all of the above as we progress further.`}

Example: A Dense Layer with 64 Neurons and sigmoid Activation.

```
from tensorflow.keras.layers import Dense
layer = Dense(64, activation="sigmoid") # 64 neurons with sigmoid activation
```

4 Building a Simple Fully Connected Neural Network in Keras.

In this section, we will build a general pipeline for solving an image classification problem using a fully connected neural network in Keras. This pipeline will largely remain the same for future implementations of more complex models for image classification.

1. Load and Preprocess the Data:

The following section provides a sample code snippet on how to manually load images from your storage, convert them into a feature matrix, and create a label vector. Before implementing the code above, please ensure that your data is stored in the following directory structure:

```
dataset/
|— Train/
|   ├— digit_0/
|   ├— digit_1/
|   ├— digit_2/
|   ├— digit_3/
|   ├— digit_4/
|   ├— digit_5/
|   ├— digit_6/
|   ├— digit_7/
|   ├— digit_8/
|   ├— digit_9/
|— Test/
|   ├— digit_0/
|   ├— digit_1/
|   ├— digit_2/
|   ├— digit_3/
|   ├— digit_4/
|   ├— digit_5/
|   ├— digit_6/
|   ├— digit_7/
|   ├— digit_8/
|   ├— digit_9/
```

Figure 1: Expected Directory Structure.

Loading and Preprocessing Data with PIL.

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from PIL import Image # Import Pillow

# Define dataset paths
train_dir = "dataset/Train/"
test_dir = "dataset/Test/"
# Define image size
img_height, img_width = 28, 28
# Function to load images and labels using PIL
def load_images_from_folder(folder):
    images = []
    labels = []
    class_names = sorted(os.listdir(folder)) # Sorted class names (digit_0, digit_1, ...)
    class_map = {name: i for i, name in enumerate(class_names)} # Map class names to labels
    for class_name in class_names:
        class_path = os.path.join(folder, class_name)
        label = class_map[class_name]
        for filename in os.listdir(class_path):
            img_path = os.path.join(class_path, filename)

            # Load image using PIL
            img = Image.open(img_path).convert("L") # Convert to grayscale
            img = img.resize((img_width, img_height)) # Resize to (28,28)
            img = np.array(img) / 255.0 # Normalize pixel values to [0,1]

            images.append(img)
            labels.append(label)
    return np.array(images), np.array(labels)
# Load training and testing datasets
x_train, y_train = load_images_from_folder(train_dir)
x_test, y_test = load_images_from_folder(test_dir)
# Reshape images for Keras input
x_train = x_train.reshape(-1, img_height, img_width, 1) # Shape (num_samples, 28, 28, 1)
x_test = x_test.reshape(-1, img_height, img_width, 1)
# One-hot encode labels
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
# Print dataset shape
print(f"Training set: {x_train.shape}, Labels: {y_train.shape}")
print(f"Testing set: {x_test.shape}, Labels: {y_test.shape}")
# Visualize some images
plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_train[i].reshape(28, 28), cmap='gray')
    plt.title(f"Label: {np.argmax(y_train[i])}")
    plt.axis("off")
plt.show()
```

The above code is expected to do following:

1. Loads the image from the specified directory structure using `PIL.Image` Library. ✓.
2. Converts images to grayscale (.convert("L")). ✓
3. Resizes images to 28×28 pixels. ✓.
4. Normalizes pixel values to range $[0, 1]$ ✓.
5. Creates corresponding labels based on folder names. ✓.
6. Reshapes images to be compatible with keras (num_samples, 28, 28, 1)) ✓.
7. One - hot encodes labels for training a neural network.
8. Prints dataset shapes for verification. ✓.
9. Displays sample images to visually confirm correctness. ✓.

The Images Compatible with Keras:

In keras, the input to the model must follow a specific shape. For image data, the expected input shape typically looks like:

- **Grayscale Image:** `(num_samples, height, width, 1)`
- **RGB Image:** `(num_samples, height, width, 3)`

In our case, we are working with grayscale images and resizing them to

$$(\texttt{num\_samples}, 28 \times 28, 1)$$

This shape means that:

- `num_samples`: Number of images in the dataset.
- **28, 28:** Height and Width of the image.
- **1:** Number of channels (since grayscale images have only one channel, representing gray scale intensity.)

Reshaping for Keras compatibility:

In your code, you can rehape the images for compatibility as following:

Compatibility check for Grayscale Image:

```
x_train = x_train.reshape(-1, img_height, img_width, 1)
# Use with Cautions.
```

Compatibility check for RGB Image:

```
x_train = x_train.reshape(-1, img_height, img_width, 3)
# Use with Cautions.
```

For the purposes of this demonstration, I will be using the MNIST dataset loaded directly from the Keras library. However, this is not the recommended approach, and you are expected to follow the aforementioned method for your workshop tasks

Loading and Preprocessing MNIST Handwritten Digit Dataset:

Loading and Preprocessing MNIST Handwritten Digit Dataset:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Normalize the images to values between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0
# Flatten the 28x28 images into 784-dimensional vectors
x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)
# One-hot encode the labels (0-9) for classification
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

2. Build the Model:

Keras provides two ways to build models: **Sequential API** and **Functional API**. Both approaches allow you to create neural networks, but the choice between them depends on the complexity and flexibility of the model you want to create. Here's a description of both, along with simple examples:

1. Sequential API:

The Sequential API is the simplest and most straightforward way to define a model in Keras. It is used when your model consists of a linear stack of layers. Each layer has one input tensor and one output tensor, making it ideal for simple, layer-by-layer models.

Example using Sequential API.

```
# Model parameters
import tensorflow as tf
from tensorflow import keras
num_classes = 10
input_shape = (28, 28, 1)
model = keras.Sequential(
    [
        keras.layers.Input(shape=input_shape),
        keras.layers.Flatten(), # Flatten the 28x28 image to a 784-dimensional vector
        keras.layers.Dense(64, activation="sigmoid"),
        keras.layers.Dense(128, activation="sigmoid"),
        keras.layers.Dense(256, activation="sigmoid"),
        keras.layers.Dense(num_classes, activation="softmax"),
    ]
)
```

The above code is expected to do the following:

1. **Input Layer:** Specifies the input shape as $(28, 28, 1)$.
2. **Flatten Layer:** Converts the 2D image $28 \times 28 \times 1$ into a **1D** vector of **784** values.

3. **Hidden Layers:**
   - A dense layer with 64 neurons and sigmoid activation.
   - A dense layer with 128 neurons and sigmoid activation.
   - A dense layer with 256 neurons and sigmoid activation.
4. **Output Layer:** A dense layer with 10 neurons (one per class) and softmax activation for multi-class classification.
5. **Next steps:** To use this model we must first compile and train it.

Model Summary:

`model.summary()` provides a structured ovverview of your neural network model. It prints a table containg:

- **Layer Type and Name:** The name and type of each layer (e.g. Dense, Flatten).
- **Output Shape:** The shape of the output tensor after each layer.
- **Number of Parameters:** The total trainable and non - trainable parameters in each layer.

Syntax Model Summary

```
model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_2 (Dense)	(None, 64)	50,240
dense_3 (Dense)	(None, 128)	8,320
dense_4 (Dense)	(None, 256)	33,024
dense_5 (Dense)	(None, 10)	2,570

```
Total params: 94,154 (367.79 KB)
Trainable params: 94,154 (367.79 KB)
Non-trainable params: 0 (0.00 B)
```

Figure 2: Expected Output of model.summary()

2. Functional API:

The Functional API is a more flexible way to define models. It allows you to define complex models with multiple inputs, outputs, shared layers, or even non-linear topologies (such as multi-branch networks). You have more control over the architecture and can define layers that may not be directly connected.

Example using Functional API.

```
# Model parameters
import tensorflow as tf
from tensorflow import keras
num_classes = 10
input_shape = (28, 28, 1)
def build_functional_model():
    # Input layer
    inputs = keras.Input(shape=input_shape)
    # Flatten layer
    x = keras.layers.Flatten()(inputs)
    # Hidden layers
    x = keras.layers.Dense(64, activation="sigmoid")(x)
    x = keras.layers.Dense(128, activation="sigmoid")(x)
    x = keras.layers.Dense(256, activation="sigmoid")(x)
    # Output layer
    outputs = keras.layers.Dense(num_classes, activation="softmax")(x)
    # Create model
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
# Build the model
functional_model = build_functional_model()
functional_model.summary()
```

The above code is expected to do the following:

1. **Input Layer:** Defines the input shape as $(28, 28, 1)$ using `keras.Input`.
2. **Flatten Layer:** Converts the 2D image $28 \times 28 \times 1$ into a **1D** vector of **784** values.
3. **Hidden Layers:**
   - A dense layer with 64 neurons and sigmoid activation.
   - A dense layer with 128 neurons and sigmoid activation.
   - A dense layer with 256 neurons and sigmoid activation.
4. **Output Layer:** A dense layer with 10 neurons (one per class) and softmax activation for multi-class classification.
5. **Model Definition:** Uses `keras.Model(inputs, outputs)` to define the network explicitly.
6. **Next steps:** To use this model, we must first compile and train it.

Model: "functional_3"

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 28, 28, 1)	0
flatten_3 (Flatten)	(None, 784)	0
dense_10 (Dense)	(None, 64)	50,240
dense_11 (Dense)	(None, 128)	8,320
dense_12 (Dense)	(None, 256)	33,024
dense_13 (Dense)	(None, 10)	2,570

Total params: 94,154 (367.79 KB)
Trainable params: 94,154 (367.79 KB)
Non-trainable params: 0 (0.00 B)

Figure 3: Expected Output of model.summary() of Functional API

You can follow either approach you prefer; for this demonstration, we will continue with the Sequential approach.

3. Compiling and Training the Model:

After defining a model using either the Sequential API or Functional API, we need to compile and train it before making predictions.

1. Compilation:

Compilation prepares the model for training by specifying:

- **Loss function**: Measures how well the model performs (e.g., categorical_crossentropy for multi-class classification).
- **Optimizer**: Updates the model parameters to minimize loss (e.g., adam).
- **Metrics**: Monitors performance (e.g., accuracy).

The `model.compile()` function in keras is used to configure the model training by specifying the optimizer, loss function and evaluation metrics.

Syntax of model.compile():

```
model.compile(
    optimizer=<optimizer>,
    loss=<loss_function>,
    metrics=[<metric1>, <metric2>, ...]
)
```

Parameter	Description	Example Values
`optimizer`	Specifies the optimization algorithm used to update weights.	`"sgd"`, `"adam"`, `etc`
`loss`	Defines the loss function that measures model performance.	`"categorical_crossentropy"`, `"sparse_categorical_crossentropy"`, `"binary_crossentropy"`
`metrics`	Specifies evaluation metrics to monitor during training.	`["accuracy"]`, `["precision", "recall"]`, `["mae"]`

Table 3: Explanation of model.compile() parameters in Keras

Example: Compiling the Model

```
model.compile(
    optimizer="sgd", # Stochastic Gradient Descent
    loss="categorical_crossentropy", # Loss function for multi-class classification
    metrics=["accuracy"] # Track accuracy during training
)
```

Some Common Loss function used for Classification Task in Keras:

Loss Function	Description	When to Use?
`categorical_crossentropy`	Computes the cross-entropy loss for multi-class classification with one-hot encoded labels.	Multi-class classification (e.g., MNIST with 10 digits).
`sparse_categorical_crossentropy`	Similar to categorical cross-entropy but expects labels as integers instead of one-hot encoded vectors.	When labels are integers (e.g., 0,1,2,...,9) instead of one-hot encoding.
`binary_crossentropy`	Computes the loss for binary classification problems.	When there are only two possible classes (e.g., cat vs. dog).
`mean_squared_error (MSE)`	Measures the squared difference between actual and predicted values.	Rarely used in classification; mainly for regression tasks.

Table 4: Common Loss Functions for Image Classification Task

Some Common Optimizers used for training of FCN:

Optimizer	Description	When to Use?
`adam`	Combines the advantages of Adagrad and RMSprop. It adjusts the learning rate dynamically and works well for a wide range of tasks.	General purpose optimizer; works well for most image classification tasks.
`sgd`	Stochastic Gradient Descent. It updates parameters by calculating the gradient on each training example (or a mini-batch).	When you need fine control over learning rate or when training large datasets.
`rmsprop`	Divides the learning rate by an exponentially decaying average of squared gradients. It works well for non-stationary objectives.	Suitable for problems with noisy or sparse gradients, like RNNs.
`adagrad`	Adapts the learning rate based on how frequently each parameter is updated. It works well for sparse data.	Works well with sparse data, such as natural language processing or computer vision tasks with sparse features.
`adadelta`	An extension of Adagrad that seeks to solve its diminishing learning rate problem by limiting the accumulated past gradients.	When you want to address the decreasing learning rate problem in Adagrad.

Table 5: Common Optimizers for Image Classification

2. Training of the Model:

Once the model are compiled, keras uses `fit()` function to train the model. It takes the training data and trains the model for a fixed number of epochs (iterations over the entire dataset.)

Syntax of fit() function.

```
model.fit(
    x=<input_data>,
    y=<target_labels>,
    batch_size=<batch_size>,
    epochs=<epochs>,
    validation_data=(<x_val>, <y_val>),
    validation_split=<validation_split>,
    callbacks=[<callback1>, <callback2>, ...],
    verbose=<verbose_level>
)
```

Parameter	Description	Example Values
`x`	The input data (e.g., training features). It could be a Numpy array, list, or a Tensor.	`x_train`, Numpy array of shape (num_samples, input_size)
`y`	The target labels corresponding to the input data. It can be a Numpy array or a list.	`y_train`, Numpy array of shape (num_samples, num_classes)
`batch_size`	The number of samples to process before updating the model's weights. A higher value leads to faster training but can reduce model generalization.	32, 64, 128
`epochs`	The number of times to iterate over the entire training dataset.	10, 20, 50
`validation_data`	Data on which the model will be evaluated after each epoch, used for early stopping or monitoring. It is a tuple (`x_val`, `y_val`).	(`x_val`, `y_val`)
`validation_split`	Fraction of the training data to be used as validation data. This is an alternative to providing explicit validation data.	0.2 (20% of the training data will be used for validation)
`callbacks`	A list of Keras callbacks to apply during training. Callbacks are functions that are applied at certain points during training (e.g., ModelCheckpoint, EarlyStopping).	[`early_stopping`], [`checkpoint`]
`verbose`	The level of verbosity during training. Controls the display of logs and progress bars.	0 (no output), 1 (progress bar), 2 (one line per epoch)
`steps_per_epoch`	The number of batches to process per epoch if you are using a generator instead of a Numpy array for training data.	100, `len(x_train) // batch_size`
`validation_steps`	The number of batches to process for validation data (if a generator is used for validation).	100, `len(x_val) // batch_size`

Table 6: Parameters used in the `fit()` function for training a model

The following code snippets show case how we implement fit function in practice:

Example Code for fit()

```
batch_size = 128
epochs = 2000
# Callbacks
callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="model_at_epoch_{epoch}.keras"),
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=4 ),
]
# Train the model with callbacks and validation split
history = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.15,
    callbacks=callbacks,
)
```

This code demonstrates how to train a model using the fit() function in Keras, with additional features like callbacks, validation_split, and specific training parameters. Let's break it down step by step:

1. Parameters and Setup:

Parameters and Setup

```
batch_size = 128
epochs = 2000
```

- **batch_size = 128**: This parameter defines the number of training samples to process before updating the model's weights. In this case, the model will process 128 samples in each batch during training.
- **epochs = 2000**: This defines the number of times the entire training dataset will be passed through the model. Here, the model will be trained for 2000 epochs.

2. Callbacks

Callbacks

```
callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="model_at_epoch_{epoch}.keras"),
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=4),
]
```

ModelCheckpoint

- **ModelCheckpoint**: This callback saves the model at each epoch with a filename that includes the epoch number (e.g., `model_at_epoch_5.keras`). This is useful for saving models during training, especially if you want to preserve the best model or checkpoints in case of interruptions.
- `filepath="model_at_epoch_epoch.keras"`: Specifies the path where the model will be saved, including the epoch number. The filename will be automatically updated with the current epoch number (e.g., `model_at_epoch_5.keras`).

EarlyStopping

- **EarlyStopping**: This callback stops training if the model's performance (on the validation set) doesn't improve for a certain number of epochs.
- `monitor="val_loss"`: The metric to monitor, in this case, validation loss (`val_loss`). This specifies that training will be stopped if the validation loss does not improve.
- `patience=4`: Specifies that if there is no improvement in validation loss for 4 consecutive epochs, training will stop early. This helps prevent overfitting by halting the training process when the model stops improving.

3. Training the Model

Training the Model

```
history = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.15,
    callbacks=callbacks,
)
```

- **x_train**: The training input data (features).
- **y_train**: The corresponding target labels for the input data.
- **batch_size = 128**: Specifies that the model will process 128 samples per batch during each training step.
- **epochs = 2000**: Specifies the number of times the model will iterate over the entire training dataset.
- **validation_split = 0.15**: This splits 15% of the training data as validation data.
- **callbacks = callbacks**: This passes the callbacks (ModelCheckpoint and EarlyStopping) to the training process.

4. History Object

- **history**: When you call `model.fit()`, Keras returns a `History` object that contains the training and validation loss and metrics for each epoch. The history object stores the training progress and can be used for further analysis.
- `history.history`: This attribute of the `History` object is a dictionary that holds the values of the metrics (such as loss, accuracy) for each epoch. It has keys like `loss`, `val_loss`, `accuracy`, `val_accuracy`, etc., depending on the metrics defined during model compilation.
- **Use of History Object**:

- `history.history` can be used to visualize the model's performance over time. For example, you can plot the training and validation loss curves to see if the model is overfitting or underfitting.
- It helps to analyze the training process and decide on adjustments to the model or training parameters (e.g., reducing learning rate, adjusting patience for early stopping).

Sample Code for visualizing Model's Training Progress.

```
import matplotlib.pyplot as plt
# Assuming 'history' is the object returned by model.fit()
# Extracting training and validation loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']
# Extracting training and validation accuracy (if metrics were specified)
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
# Plotting training and validation loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(range(1, len(train_loss) + 1), train_loss, label='Training Loss', color='blue')
plt.plot(range(1, len(val_loss) + 1), val_loss, label='Validation Loss', color='orange')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
# Plotting training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, len(train_acc) + 1), train_acc, label='Training Accuracy', color='blue')
plt.plot(range(1, len(val_acc) + 1), val_acc, label='Validation Accuracy', color='orange')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.tight_layout()
plt.show()
```

The above code:

The `history.history` attribute contains the loss and accuracy values for both the training and validation sets.

- **train_loss, val_loss, train_acc, and val_acc:** These variables are extracted from the `history` object, which stores the progress of training.

The code creates two plots:

- **Training and Validation Loss:** Shows how the loss decreases (or increases) over epochs.
- **Training and Validation Accuracy:** Shows how the accuracy improves over epochs.

Both plots are displayed side by side using subplots. This visualization helps in understanding the model's learning curve, including whether it is overfitting, underfitting, or generalizing well.

4. Evaluate the Model:

The model.evaluate() function in Keras is used to evaluate the model's performance on a test dataset (or validation dataset). It returns the loss value and the metric(s) specified during the model.compile() phase, which can help assess how well the model generalizes to new, unseen data.

Syntax for model.evaluate()

```
model.evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None, callbacks
    =None, max_queue_size=10, workers=1, use_multiprocessing=False)
```

Parameter	Description	Example Values
`x`	The input data on which the model will be evaluated. Can be a Numpy array, Tensor, or a dataset object.	`x_test`
`y`	The true labels for the input data. Can be one-hot encoded or integer labels, depending on the loss function.	`y_test`
`batch_size`	Number of samples per evaluation batch. If None, the default batch size of 32 is used.	64, 128
`verbose`	Controls the verbosity of the evaluation process.	0 (Silent), 1 (Progress bar), 2 (One line per epoch)
`sample_weight`	Weights for each sample during evaluation. If None, each sample is given equal weight.	`weights_array`
`steps`	The number of steps (batches) to run for evaluation when using a generator. If None, the entire dataset is used.	100, 200
`callbacks`	List of Keras callbacks to apply during evaluation.	`[callback_1, callback_2]`
`max_queue_size`	Maximum size of the generator queue for evaluation.	10 (default)
`workers`	Number of workers for loading data in the background.	1, 4
`use_multiprocessing`	Whether to use the multiprocessing start method for parallel data loading.	False (default)

Table 7: Parameters for the `model.evaluate()` Function

The **model.evaluate Returns** following:

- **loss:** The loss value, as defined by the loss function.
- **metrics:** The value(s) of the metrics specified during model.compile().

Example code for Evaluation.

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"Test accuracy: {test_acc:.4f}")
```

Explanation of `model.evaluate()` and Output

- **x_test and y_test**: These represent the input features and corresponding true labels of the test dataset. `x_test` contains the test data, while `y_test` contains the true labels for the evaluation.
- **verbose=2**: This argument ensures that the evaluation output will be printed as a summary for each epoch (or evaluation step), giving a detailed overview of the evaluation process.

The `model.evaluate()` function returns the following values:

- **test_loss**: The loss value computed based on the test dataset. This gives an indication of how well the model is performing on unseen data.
- **test_acc**: The accuracy value computed based on the test dataset. It shows the proportion of correct predictions made by the model.

5. Making Predictions with Keras:

Once the model is trained, you can use it to make predictions on new data using model.predict() function of keras:

Syntax for model.predict()

```
model.predict(
    x,
    batch_size=None,
    verbose=0,
    steps=None,
    callbacks=None,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False
)
```

Parameter	Description	Example Values
`x`	The input data to predict on. It can be a Numpy array, Tensor, or dataset (e.g., tf.data.Dataset).	`x_test`, Numpy array of shape (num_samples, input_size)
`batch_size`	The number of samples per evaluation batch. Default is None, and the batch size is determined by the model.	32, 64, 128
`verbose`	Integer value to control the display of progress:	0 (silent), 1 (progress bar), 2 (one line per batch)
`steps`	The number of steps to run for prediction. If using a generator, this defines the number of batches to run.	100 (if using a generator)
`callbacks`	A list of callbacks to apply during prediction.	None, [callback]
`max_queue_size`	The maximum size of the queue for data loading.	10, 20
`workers`	The number of worker threads to use for data loading in parallel.	1, 4
`use_multiprocessing`	Whether to use the multiprocessing start method for parallel data loading.	False, True

Table 8: Parameters for model.predict()

Example code for model.predict()

```
# Predict on test data
predictions = model.predict(x_test)
# Convert predictions from probabilities to digit labels
predicted_labels = np.argmax(predictions, axis=1)
# Check the first prediction
print(f"Predicted label for first image: {predicted_labels[0]}")
print(f"True label for first image: {np.argmax(y_test[0])}")
```

Explanation of `model.predict()` and Predicted Labels

- **predictions = model.predict(x_test)**: This line uses the trained model to make predictions on the test data (`x_test`). The `predict()` function returns a Numpy array of predicted probabilities for each class. In a multi-class classification task, this will return a probability distribution over the classes for each test sample.

Next, we convert these probabilities into class labels:

- **predicted_labels = np.argmax(predictions, axis=1)**: This line extracts the predicted class labels by selecting the class with the highest probability. The `np.argmax()` function returns the index of the maximum value along the specified axis (axis=1 refers to the class dimension).

The following `print` statements display the predicted and true labels for the first image in the test set:

- **print(f"Predicted label for first image: predicted_labels[0]")**: This prints the predicted label for the first image in the test set.
- **print(f"True label for first image: np.argmax(y_test[0])")**: This prints the true label for the first image, where `np.argmax(y_test[0])` extracts the index of the correct class from the one-hot encoded vector in `y_test`.

The predicted label corresponds to the class with the highest predicted probability, while the true label corresponds to the actual class. Comparing these values allows us to check how well the model is performing on individual test samples.

6. Saving and Loading the Model:

1. Saving the Model:

Saving the Model

```
model.save('mnist_fully_connected_model.h5')
```

- `model.save()`: This function saves the entire model (including architecture, weights, and optimizer state) to a specified file. In this case, the model is saved in the HDF5 format (`.h5` file).
- `'mnist_fully_connected_model.h5'`: This is the filename where the model will be saved. The model will be stored in the current working directory.

2. Loading the Model:

Loading the Model

```
loaded_model = tf.keras.models.load_model('mnist_fully_connected_model.h5')
```

- `tf.keras.models.load_model()`: This function loads a previously saved Keras model from the specified file.
- `'mnist_fully_connected_model.h5'`: This is the filename where the model was saved earlier.
- `loaded_model`: This variable holds the loaded model, which can now be used for further evaluation or inference.

5 Conclusion:

In this tutorial we explored:

- **Training, Evaluating, and Saving Keras Models:**
- **Callbacks:** Highlighted the importance of callbacks like ModelCheckpoint for saving intermediate models and EarlyStopping for preventing overfitting.
- **Model Evaluation and Prediction:**
  - Learned how to evaluate model performance on test data using evaluate().
  - Explored making predictions on unseen samples using predict().
- **Saving and Loading Models:**
  - Explained how to persist trained models using the save() function.
  - Learned how to reload models with load_model(), making it easy to continue training or use the model for inference in different contexts.
- **Visualization:** Discussed visualizing training metrics (e.g., loss and accuracy) to diagnose how well the model is learning and generalizing.

6 Exercise: Building a Fully Connected Network (FCN) for Devnagari Digit Classification.

Objective

In this exercise, you will build and train a Fully Connected Network (FCN) to classify Devnagari digits using TensorFlow and Keras. You will manually load and process the dataset using the Python Imaging Library (PIL) and then train the model with three hidden layers.

Task 1: Data Preparation

Loading the Data

- Download the provided folder that contains the Devnagari digits dataset.
- Use the Python Imaging Library (PIL) to load and read the image files from the dataset.
- Convert the images into Numpy arrays and normalize them to a range of 0-1.
- use train folder for training and test for testing.
- Extract the corresponding labels for each image.

Hints:

- Ensure that the images are resized to a consistent shape (e.g., 28x28).
- Convert labels to one-hot encoded format for multi-class classification.

Task 2: Build the FCN Model

Model Architecture

- Create a Sequential model using Keras.
- Add 3 hidden layers with the following number of neurons:
  - 1st hidden layer: 64 neurons
  - 2nd hidden layer: 128 neurons
  - 3rd hidden layer: 256 neurons
- Use sigmoid activation functions for all hidden layers.
- Add an output layer with 10 units with softmax (since Devnagari digits have 10 classes) and a softmax activation function.

Task 3: Compile the Model

Model Compilation

- Choose an appropriate optimizer (e.g., Adam), loss function (e.g., sparse categorical crossentropy), and evaluation metric (e.g., accuracy).

Task 4: Train the Model

Model Training

- Use the model.fit() function to train the model. Set the batch size to 128 and the number of epochs to 20.
- Use validation split (`validation_split=0.2`) to monitor the model's performance on validation data.
- Optionally, use callbacks such as `ModelCheckpoint` and `EarlyStopping` for saving the best model and avoiding overfitting.

Task 5: Evaluate the Model

Model Evaluation

- After training, evaluate the model using `model.evaluate()` on the test set to check the test accuracy and loss.

Task 6: Save and Load the Model

Model Saving and Loading

- Save the trained model to an `.h5` file using `model.save()`.
- Load the saved model and re-evaluate its performance on the test set.

Task 7: Predictions

Making Predictions

- Use `model.predict()` to make predictions on test images.
- Convert the model's predicted probabilities to digit labels using `np.argmax()`.

Expected Deliverables

- **Code Implementation:** Complete code for building, training, evaluating, saving, and loading the model.
- **Visualization:** Graphs showing the training and validation loss and accuracy.
- **Test Accuracy:** Display the final test accuracy.
- **Saved Model:** Submit the saved `.h5` model file.

Good Luck.