Tutorial: Implementing CNN, RNN, and LSTM in Machine Learning

Student Id: 23082925

Github link: https://github.com/GVDRamKumar/ML-NN-ASSIGNMENT.git

---

In this tutorial, we will implement and compare three fundamental types of neural networks: **Convolutional Neural Networks (CNN)**, **Recurrent Neural Networks (RNN)**, and **Long Short-Term Memory Networks (LSTM)**, using **TensorFlow** and **Keras**. These three models will be applied to two distinct datasets: **MNIST** for image classification and **IMDB** for sentiment analysis.

The goal of this tutorial is to give a clear understanding of how these different architectures work, and how they can be applied to solve real-world machine learning problems.

---

1. Overview of the Models: CNN, RNN, and LSTM

Convolutional Neural Networks (CNN)

CNNs are primarily used for image data due to their ability to extract spatial features from images. They are designed to automatically detect patterns such as edges, textures, and shapes, making them excellent for tasks like image classification and object detection.

Mathematical Function: The convolution operation in a CNN is defined as:

$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

where:

- I is the input image
- K is the kernel (filter)
- (i, j) are the spatial coordinates of the output feature map
- (m, n) are the spatial coordinates of the kernel

How CNN Works:

- The core idea of CNNs is the **convolution operation**. This operation applies filters (or kernels) across the image, extracting spatial features at different levels of abstraction.
- CNNs typically have **Convolutional layers**, **Pooling layers** (to reduce the image size), and **Fully Connected layers** (for classification).

Recurrent Neural Networks (RNN)

RNNs are designed for **sequential data**, such as time series, text, or speech. Unlike CNNs, which process independent data points, RNNs maintain an internal state that gets updated with each new input, making them suitable for tasks where the order of the data matters (e.g., language translation or sentiment analysis).

$$\boldsymbol{h_t = \sigma\left(W_h h_{\{t-1\}} + W_x x_t + b\right)}$$

where:

- $\boldsymbol{h_t}$ is the hidden state at time $\boldsymbol{t}$
- $\boldsymbol{h_{\{t-1\}}}$ is the hidden state at time $\boldsymbol{t-1}$ (the previous time step)
- $\boldsymbol{x_t}$is the input at time $\boldsymbol{t}$
- $\boldsymbol{W_h}$ and $\boldsymbol{W_x}$ are weight matrices for the hidden state and the input respectively
- $\boldsymbol{\sigma}$ is an activation function (usually tanh or ReLU)
- $\boldsymbol{b}$ is the bias term

How RNN Works:

- RNNs process input data in a **sequential manner**, passing information from previous steps to the current step through their internal state.
- RNNs have the potential to remember information across time steps, which is useful for applications like text classification or speech recognition.

Long Short-Term Memory Networks (LSTM)

LSTMs are a type of RNN that are designed to overcome the **vanishing gradient problem**, which makes traditional RNNs struggle with long-term dependencies. LSTMs maintain a more complex internal state, allowing them to remember information over longer sequences.

Forget Gate: Decides what information should be discarded from the cell state.

$$f_t = \sigma\left(W_f \cdot \left[h_{\{t-1\}}, x_t\right] + b_f\right)$$

Input Gate: Decides what new information should be stored in the cell state.

$$i_t = \sigma\left(W_i \cdot \left[h_{\{t-1\}}, x_t\right] + b_i\right)$$

The input gate creates candidate values $\{C\}_t$that could be added to the cell state:

$$\{C\}_t = \backslash tanh\left(W_C \cdot \left[h_{\{t-1\}}, x_t\right] + b_C\right)$$

Update Cell State: The cell state C_t is updated using the forget and input gates:

$$C_t = f_t \cdot C_{\{t-1\}} + i_t \cdot {}^{\sim}\{C\}_t$$

Output Gate: Decides the output based on the current cell state and the previous hidden state.

$$o_t = \sigma\left(W_o \cdot \left[h_{\{t-1\}}, x_t\right] + b_o\right)$$

The final output h_t is:

$$h_t = o_t \cdot \backslash tanh(C_t)$$

How LSTM Works:

- LSTMs have **gates** (input, forget, and output gates) that regulate the flow of information into, out of, and within the network, which helps them remember important data over longer sequences.

---

2. Dataset Overview

MNIST Dataset (for CNN and LSTM)

- The **MNIST dataset** is a collection of 28x28 pixel grayscale images of handwritten digits from 0 to 9.
- The goal is to classify each image into one of 10 classes (0-9).

IMDB Dataset (for RNN)

- The **IMDB dataset** consists of movie reviews, labeled as either positive or negative.
- The task is to classify each review based on sentiment.

---

3. Preprocessing the Data

For CNN Model (MNIST)

- **Reshaping**: The MNIST dataset consists of 28x28 grayscale images, which are reshaped to have dimensions (28, 28, 1) to match the input requirements of CNN layers.

- **Normalization**: The pixel values are scaled from the range [0, 255] to [0, 1] by dividing by 255.
- **One-hot Encoding**: The labels (digits) are one-hot encoded, converting the numeric labels (0-9) into binary vectors.

For RNN Model (IMDB)

- **Padding Sequences**: The IMDB dataset consists of reviews with varying lengths. To ensure that all reviews are the same length, we use **padding** to make each sequence 500 words long.
- **Tokenization**: The words are converted into integer indices, representing each word in the vocabulary.

For LSTM Model (MNIST)

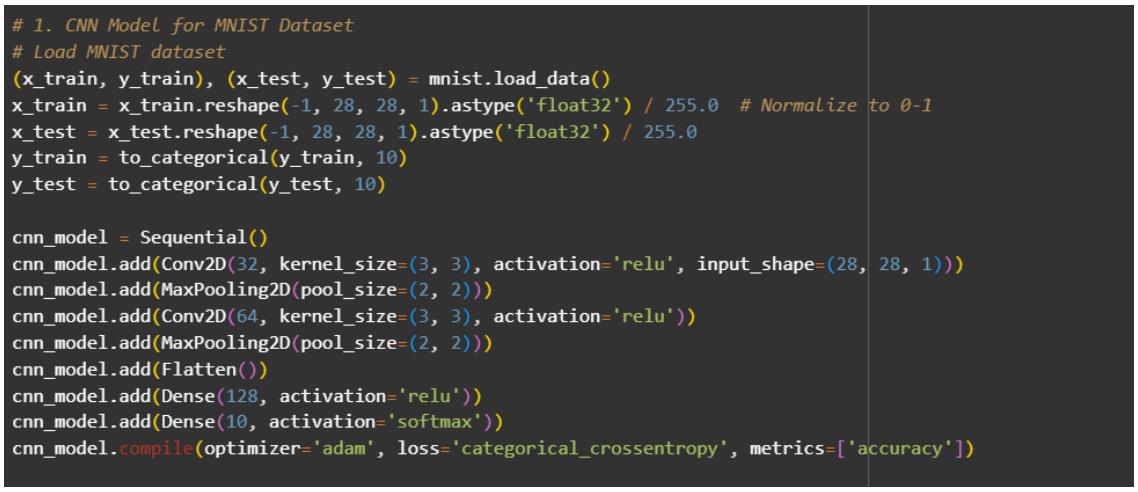
- **Reshaping for Sequence Processing**: The MNIST dataset is reshaped into sequences, where each row of the image is treated as a timestep, with each pixel in the row being a feature.
- **Normalization**: Like the CNN model, the pixel values are normalized.

---

4. Model Building

CNN Model

- **Convolutional Layers**: The CNN uses two convolutional layers (Conv2D), each followed by a max-pooling layer (MaxPooling2D), which helps to reduce the image dimensions while retaining essential features.
- **Flattening**: The output from the convolutional layers is flattened into a 1D array to pass into the fully connected (Dense) layers.
- **Fully Connected Layer**: A dense layer with 128 units followed by an output layer with 10 units (for the 10 classes of digits).

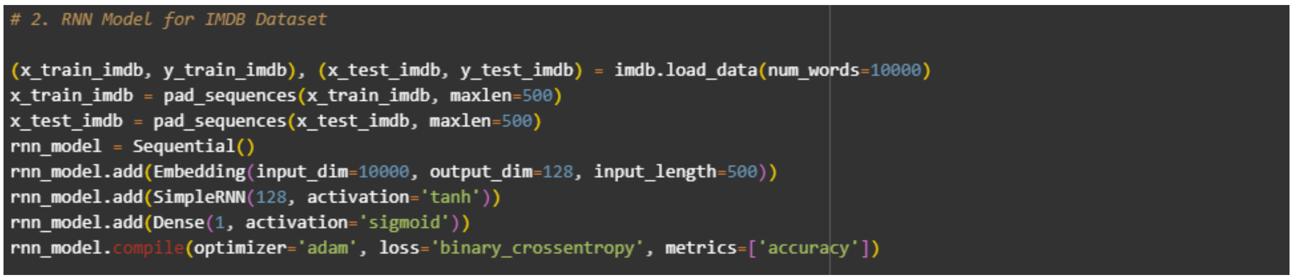


```python
# 1. CNN Model for MNIST Dataset
# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0  # Normalize to 0-1
x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

cnn_model = Sequential()
cnn_model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model.add(Flatten())
cnn_model.add(Dense(128, activation='relu'))
cnn_model.add(Dense(10, activation='softmax'))
cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

RNN Model

- **Embedding Layer**: The first layer is an embedding layer that converts words into dense vectors.
- **Simple RNN Layer**: The second layer is a **SimpleRNN** layer, which processes the sequence of word vectors.
- **Output Layer**: The output layer consists of a dense layer with a **sigmoid activation** for binary classification.

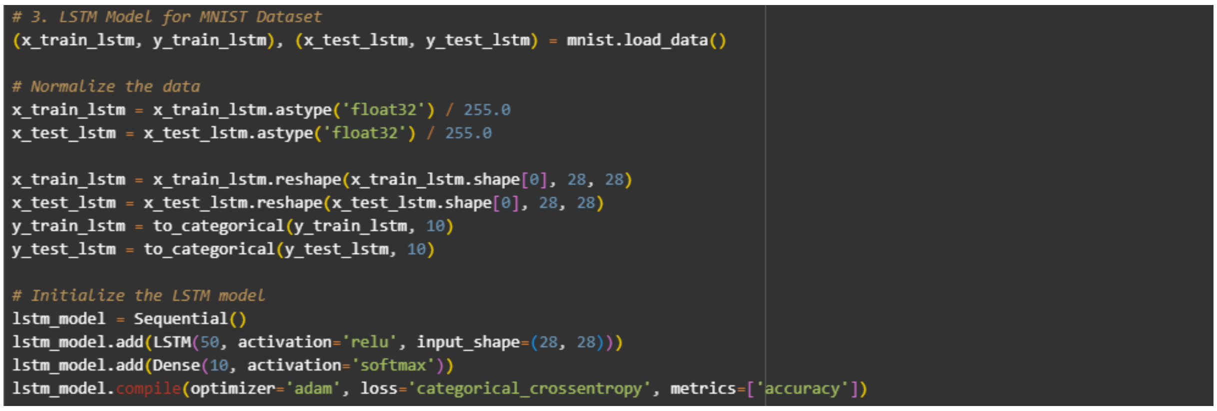


```python
# 2. RNN Model for IMDB Dataset

(x_train_imdb, y_train_imdb), (x_test_imdb, y_test_imdb) = imdb.load_data(num_words=10000)
x_train_imdb = pad_sequences(x_train_imdb, maxlen=500)
x_test_imdb = pad_sequences(x_test_imdb, maxlen=500)
rnn_model = Sequential()
rnn_model.add(Embedding(input_dim=10000, output_dim=128, input_length=500))
rnn_model.add(SimpleRNN(128, activation='tanh'))
rnn_model.add(Dense(1, activation='sigmoid'))
rnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

LSTM Model

- **LSTM Layer**: The LSTM model uses an LSTM layer with 50 units to process sequences of pixel rows from the MNIST dataset.
- **Output Layer**: Like the CNN model, the LSTM has a **Dense output layer** with 10 units for the classification of digits.



```python
# 3. LSTM Model for MNIST Dataset
(x_train_lstm, y_train_lstm), (x_test_lstm, y_test_lstm) = mnist.load_data()

# Normalize the data
x_train_lstm = x_train_lstm.astype('float32') / 255.0
x_test_lstm = x_test_lstm.astype('float32') / 255.0

x_train_lstm = x_train_lstm.reshape(x_train_lstm.shape[0], 28, 28)
x_test_lstm = x_test_lstm.reshape(x_test_lstm.shape[0], 28, 28)
y_train_lstm = to_categorical(y_train_lstm, 10)
y_test_lstm = to_categorical(y_test_lstm, 10)

# Initialize the LSTM model
lstm_model = Sequential()
lstm_model.add(LSTM(50, activation='relu', input_shape=(28, 28)))
lstm_model.add(Dense(10, activation='softmax'))
lstm_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

---

5. Compiling the Models

All three models are compiled with:

- **Adam Optimizer**: A popular choice for optimization due to its adaptive learning rate.
- **Loss Function**: We use **categorical crossentropy** for multi-class classification (CNN and LSTM on MNIST) and **binary crossentropy** for binary classification (RNN on IMDB).

- **Accuracy Metric**: The **accuracy** is used to evaluate the model's performance during training and testing.

---

6. Training the Models

Each model is trained on its respective dataset:

- **CNN** and **LSTM** are trained on the MNIST dataset for digit classification.
- **RNN** is trained on the IMDB dataset for sentiment analysis.

```python
# Train the CNN model
cnn_history = cnn_model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test, y_test))
```

```python
# Train the RNN model
rnn_history = rnn_model.fit(x_train_imdb, y_train_imdb, epochs=5, batch_size=64, validation_data=(x_test_imdb, y_test_imdb))
```

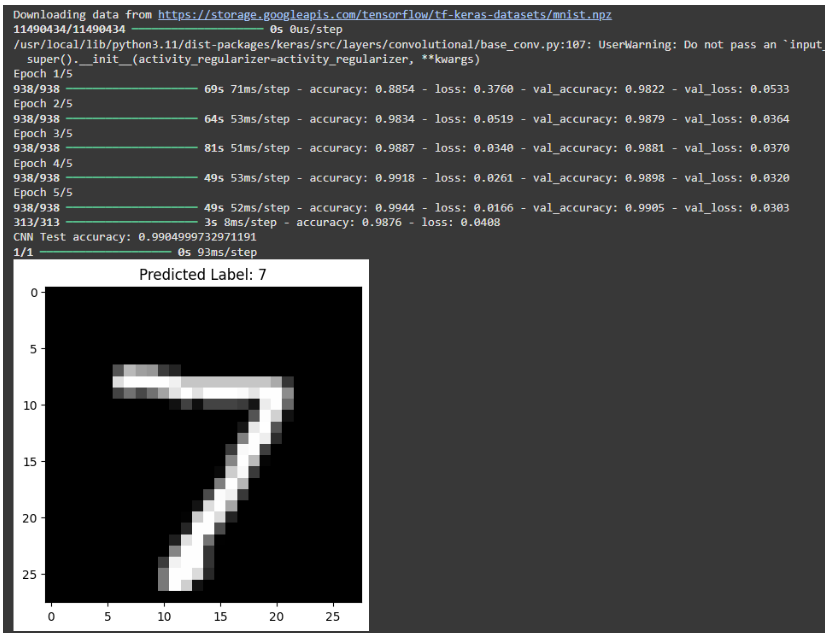
```python
# Train the LSTM model
lstm_history = lstm_model.fit(x_train_lstm, y_train_lstm, epochs=5, batch_size=64, validation_data=(x_test_lstm, y_test_lstm))
lstm_test_loss, lstm_test_acc = lstm_model.evaluate(x_test_lstm, y_test_lstm)
print(f"LSTM Test accuracy: {lstm_test_acc}")
```

---

7. Visualizing the Results

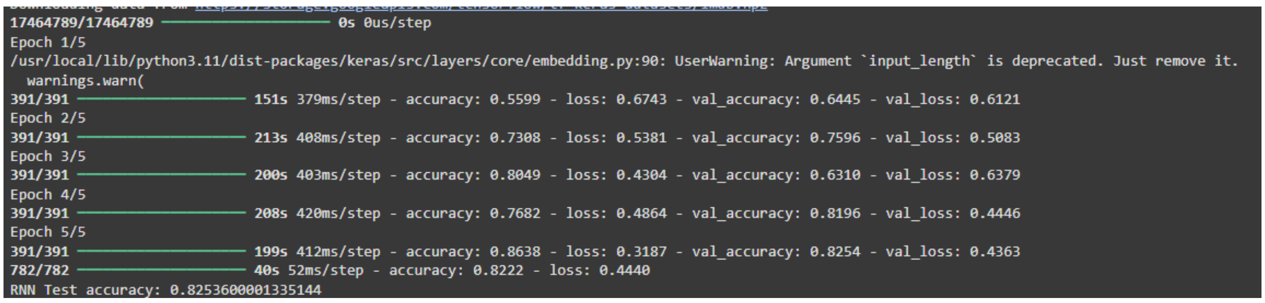
Finally, we visualize the accuracy of all three models over the training epochs to compare their performance.

To see the result by generating the image, this is the output received for CNN model:



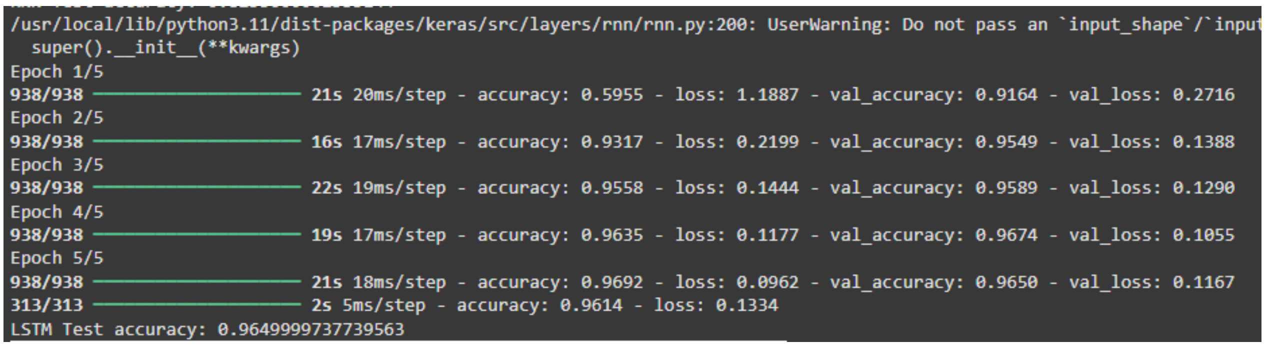
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 69s 71ms/step - accuracy: 0.8854 - loss: 0.3760 - val_accuracy: 0.9822 - val_loss: 0.0533
Epoch 2/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 64s 53ms/step - accuracy: 0.9834 - loss: 0.0519 - val_accuracy: 0.9879 - val_loss: 0.0364
Epoch 3/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 81s 51ms/step - accuracy: 0.9887 - loss: 0.0340 - val_accuracy: 0.9881 - val_loss: 0.0370
Epoch 4/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 49s 53ms/step - accuracy: 0.9918 - loss: 0.0261 - val_accuracy: 0.9898 - val_loss: 0.0320
Epoch 5/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 49s 52ms/step - accuracy: 0.9944 - loss: 0.0166 - val_accuracy: 0.9905 - val_loss: 0.0303
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9876 - loss: 0.0408
CNN Test accuracy: 0.9904999732971191
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step
```

As a result, the output for RNN is:



```
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/5
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
391/391 ━━━━━━━━━━━━━━━━━━━━ 151s 379ms/step - accuracy: 0.5599 - loss: 0.6743 - val_accuracy: 0.6445 - val_loss: 0.6121
Epoch 2/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 213s 408ms/step - accuracy: 0.7308 - loss: 0.5381 - val_accuracy: 0.7596 - val_loss: 0.5083
Epoch 3/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 200s 403ms/step - accuracy: 0.8049 - loss: 0.4304 - val_accuracy: 0.6310 - val_loss: 0.6379
Epoch 4/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 208s 420ms/step - accuracy: 0.7682 - loss: 0.4864 - val_accuracy: 0.8196 - val_loss: 0.4446
Epoch 5/5
391/391 ━━━━━━━━━━━━━━━━━━━━ 199s 412ms/step - accuracy: 0.8638 - loss: 0.3187 - val_accuracy: 0.8254 - val_loss: 0.4363
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 52ms/step - accuracy: 0.8222 - loss: 0.4440
RNN Test accuracy: 0.8253600001335144
```

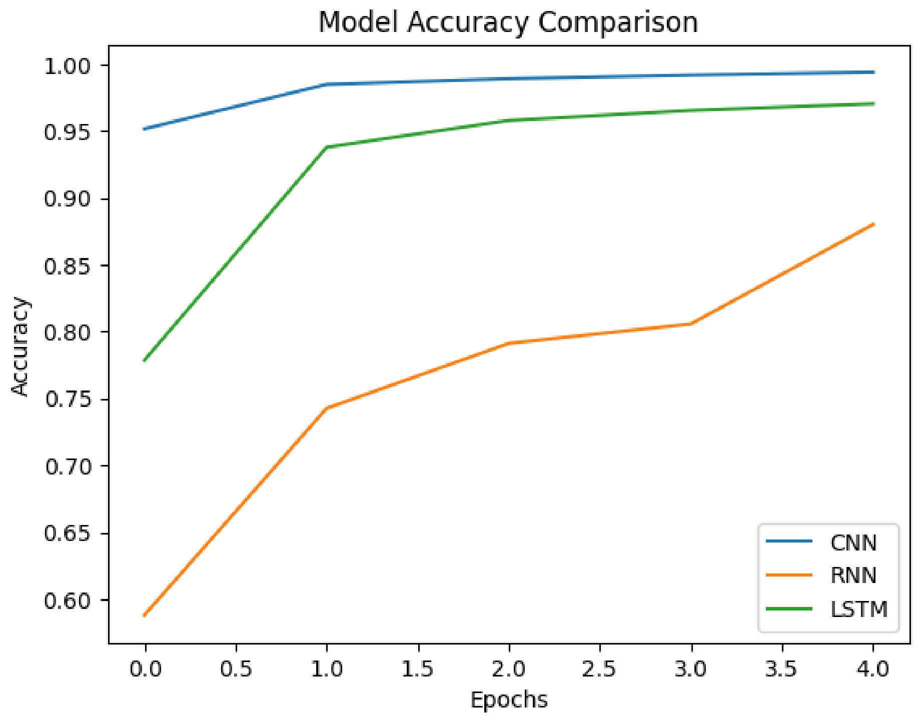
As a result, the output for LSTM is:



```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`input
  super().__init__(**kwargs)
Epoch 1/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 21s 20ms/step - accuracy: 0.5955 - loss: 1.1887 - val_accuracy: 0.9164 - val_loss: 0.2716
Epoch 2/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 16s 17ms/step - accuracy: 0.9317 - loss: 0.2199 - val_accuracy: 0.9549 - val_loss: 0.1388
Epoch 3/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 22s 19ms/step - accuracy: 0.9558 - loss: 0.1444 - val_accuracy: 0.9589 - val_loss: 0.1290
Epoch 4/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 19s 17ms/step - accuracy: 0.9635 - loss: 0.1177 - val_accuracy: 0.9674 - val_loss: 0.1055
Epoch 5/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 21s 18ms/step - accuracy: 0.9692 - loss: 0.0962 - val_accuracy: 0.9650 - val_loss: 0.1167
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9614 - loss: 0.1334
LSTM Test accuracy: 0.9649999737739563
```

8. Conclusion

This tutorial demonstrated how to implement and train **CNN**, **RNN**, and **LSTM** models for two different tasks: **image classification** with MNIST and **sentiment analysis** with IMDB.



- **CNNs** are highly effective for tasks that involve images because of their ability to automatically detect important spatial features in the data.
- **RNNs** are great for sequential data such as text, where the order of data points is crucial.
- **LSTMs**, a type of RNN, are particularly useful when handling long-term dependencies in sequences.

Each of these models is applied to the appropriate dataset, showcasing their strengths and how they can be tailored to solve different machine learning problems.

References

1. TensorFlow and Keras Documentation
2. MNIST Dataset: LeCun, Y., et al., “Gradient-Based Learning Applied to Document Recognition,” Proceedings of the IEEE, 1998.
3. IMDB Dataset: Maas, A. L., et al., “Learning Word Vectors for Sentiment Analysis,” ACL, 2011.