Sign Language Recognition using CNN

By Rachit Bhalla 21BAI1869 ✦ Natanya Modi 21BAI1405

Background

Sign language recognition with CNN involves teaching a deep learning model to recognise sign language hand gestures. The steps in the process include gathering and preprocessing data, designing and training the model, assessing its performance, and optimising it for use in real-time. Convolutional layers are typically used to extract features from input data, followed by fully connected layers to classify the gesture. Once trained and optimised, the model can be used for real-time sign language recognition, potentially improving communication and accessibility for people who are deaf.

Introduction

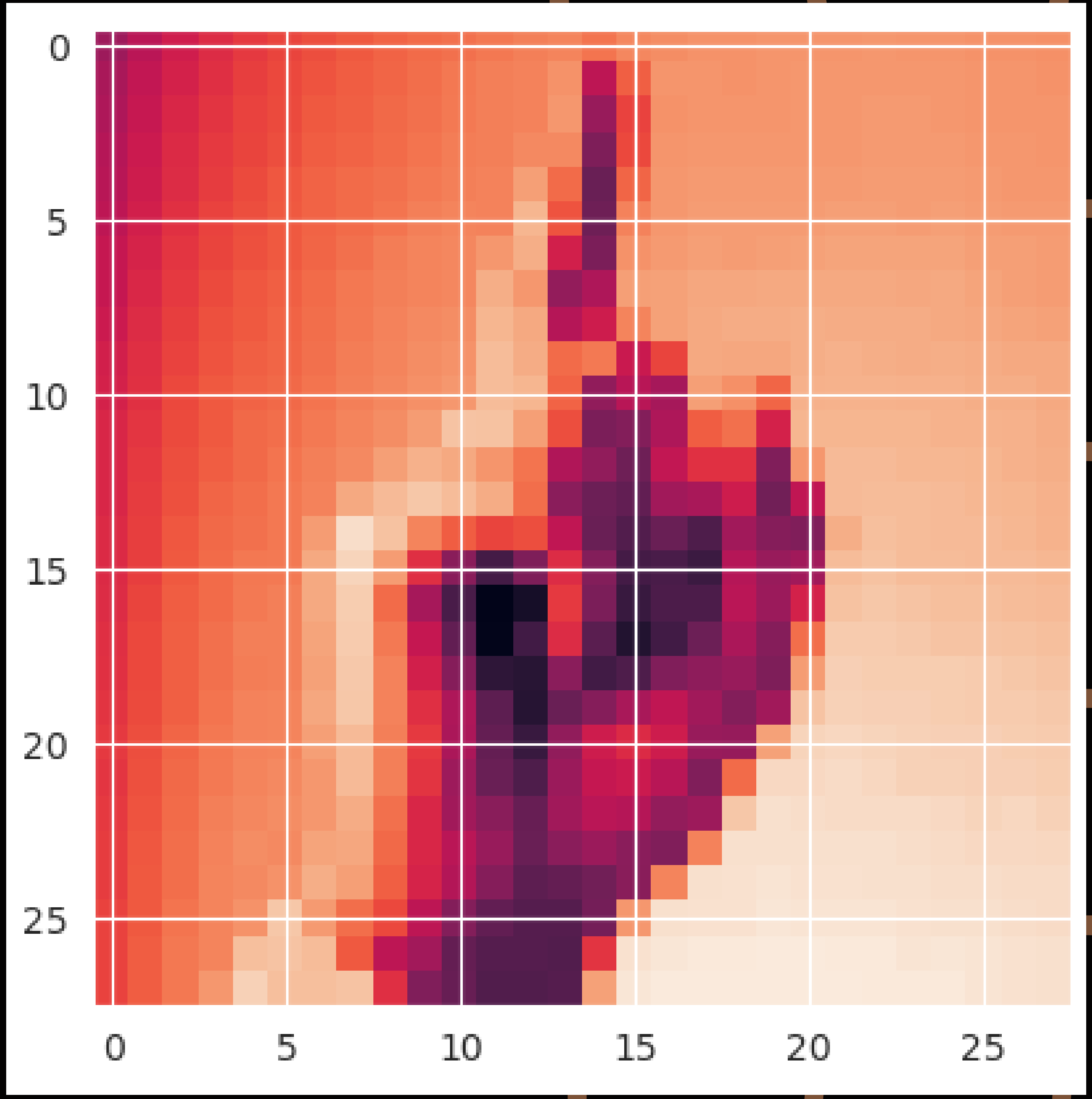
CNN is a network architecture for deep learning that learns and increases its knowledge base from data to recognize and interpret hand gestures in sign language. It takes in image data, extracts features using convolutional and pooling layers, and then classifies the gesture using fully connected layers. The CNN model is trained by feeding it a large dataset of sign language gestures and their corresponding labels, and it adjusts its weights to minimize the difference between predicted and actual labels. Once trained, the CNN model can be used to identify signs and make learning and teaching sign language more convenient for those who are deaf or hard of hearing.

Dataset

Dataset Used:

https://www.kaggle.com/datasets/datamunge/sign-language-mnist

There are no cases for 9=J or 25=Z due to gesture motions; each training and test case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z. The About half the size of the normal MNIST, training data (27,455 cases) and test data (7172 cases) each have a header row of labels that read "pixel1, pixel2,... pixel784" and represent a single 28x28 pixel picture with grayscale values between 0-255.



Algorithm

01 Import the necessary libraries and modules from Keras, including numpy, pandas, matplotlib, tensorflow

02 Load the Kaggle MNIST Sign Language Dataset into two dataframes: train data and test data.

03 Reshape the image data into the appropriate format and separate the label and data columns and convert them to numpy arrays.

04 Divide the image data by 255.0 to normalzse the pixel values. Using the Keras Sequential model, define two different models with the same architecture, which includes Conv2D, MaxPooling2D, Flatten, Dense, and Dropout layers.

05 Compile both models using two distinct optimizers: Adam and RMSprop.
Both models should be trained on training data and validated on test data using the fit() method.

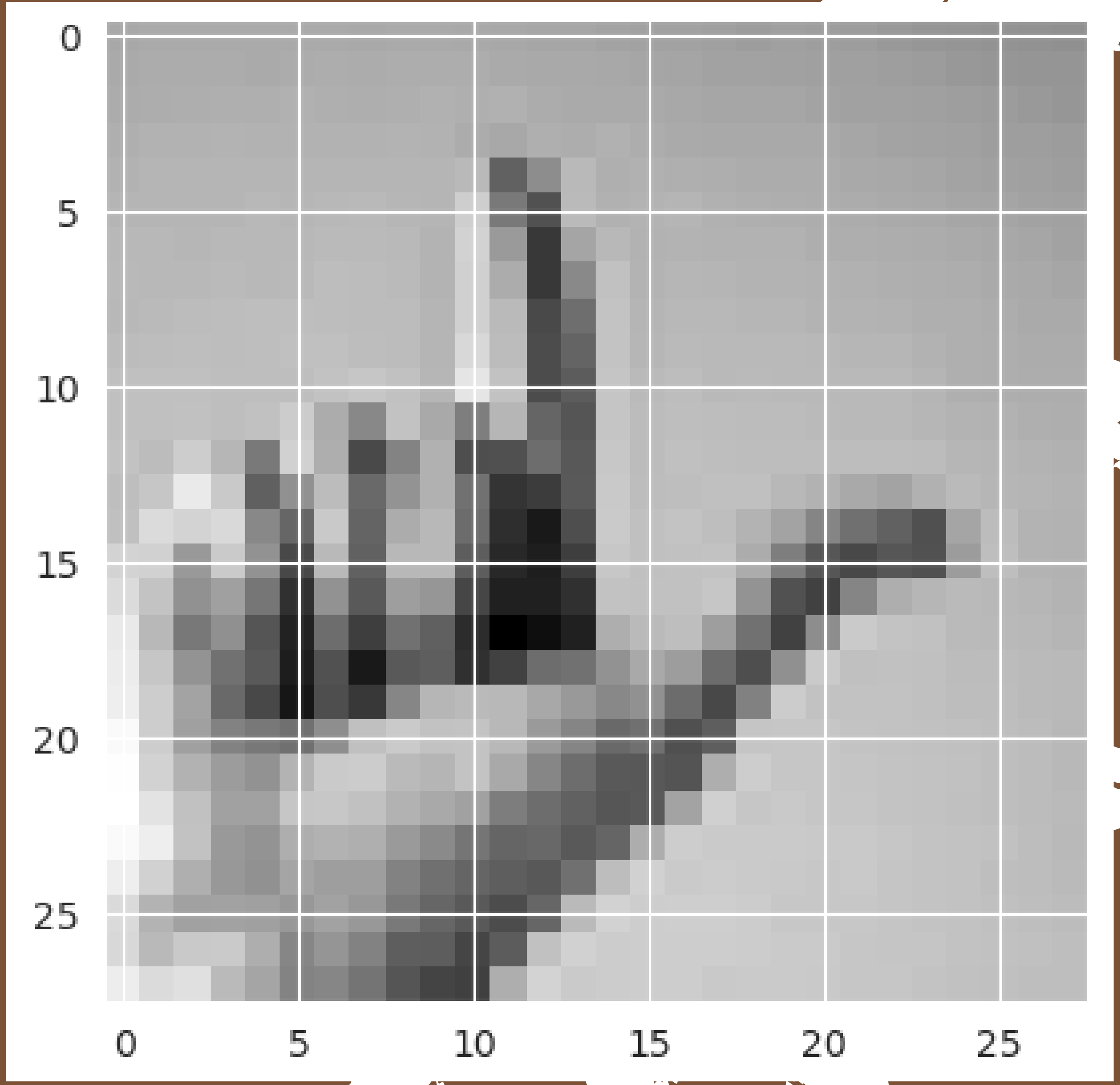
06 Using matplotlib, plot the accuracy and loss curves for both models.

Optimizers

Adam and RMSprop are the two optimizers used in the code.

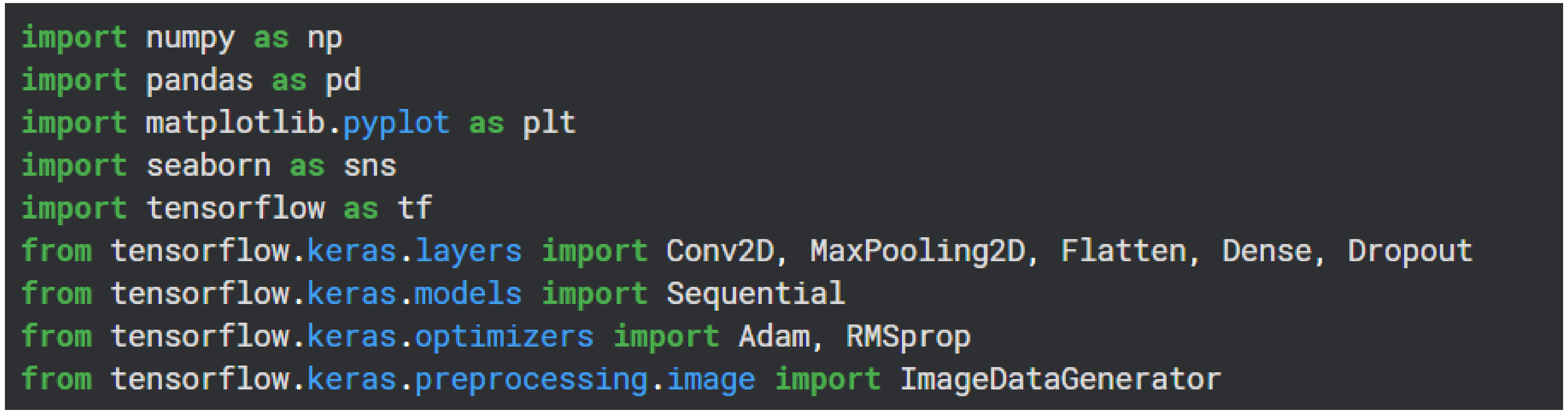
The Adam optimizer is an SGD extension that keeps a moving average of the gradient and the squared gradient. This enables it to adjust the learning rate of each weight adaptively based on the historical gradient information. In practice, this can result in faster convergence and higher performance than traditional SGD.

The RMSprop optimizer, on the other hand, is an extension of SGD that updates the learning rate for each weight based on the average of the weight's squared gradients. This means that weights with more stable gradients will learn faster than weights with more volatile gradients. This can aid in the prevention of oscillations during the training process and improve convergence.



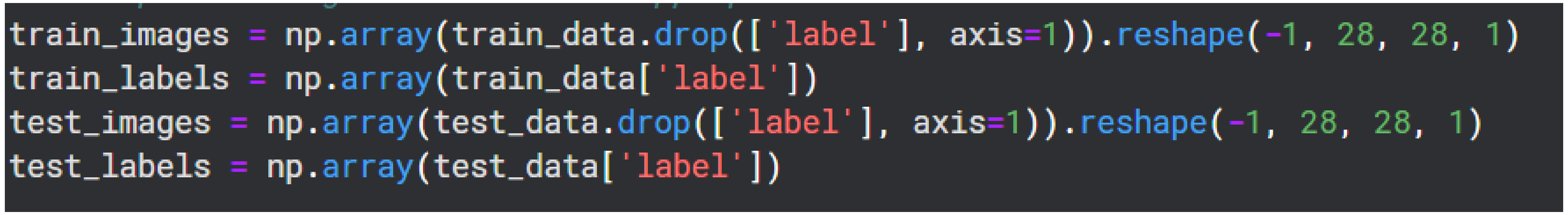
Code

Libraries :

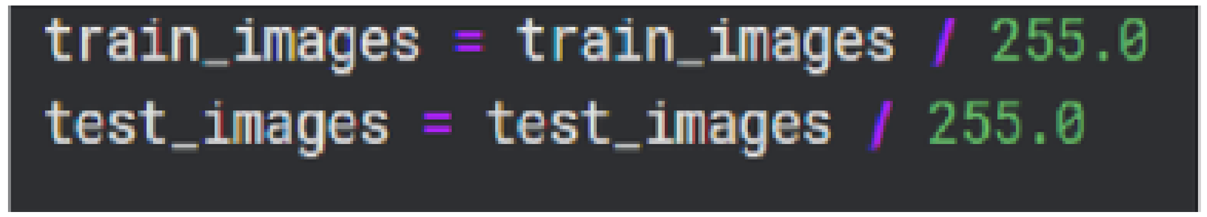


```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Reshaping and training image data :



```python
train_images = np.array(train_data.drop(['label'], axis=1)).reshape(-1, 28, 28, 1)
train_labels = np.array(train_data['label'])
test_images = np.array(test_data.drop(['label'], axis=1)).reshape(-1, 28, 28, 1)
test_labels = np.array(test_data['label'])
```



```python
train_images = train_images / 255.0
test_images = test_images / 255.0
```

```
model1 = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(26, activation='softmax')
])
```

Defining a basic CNN architecture

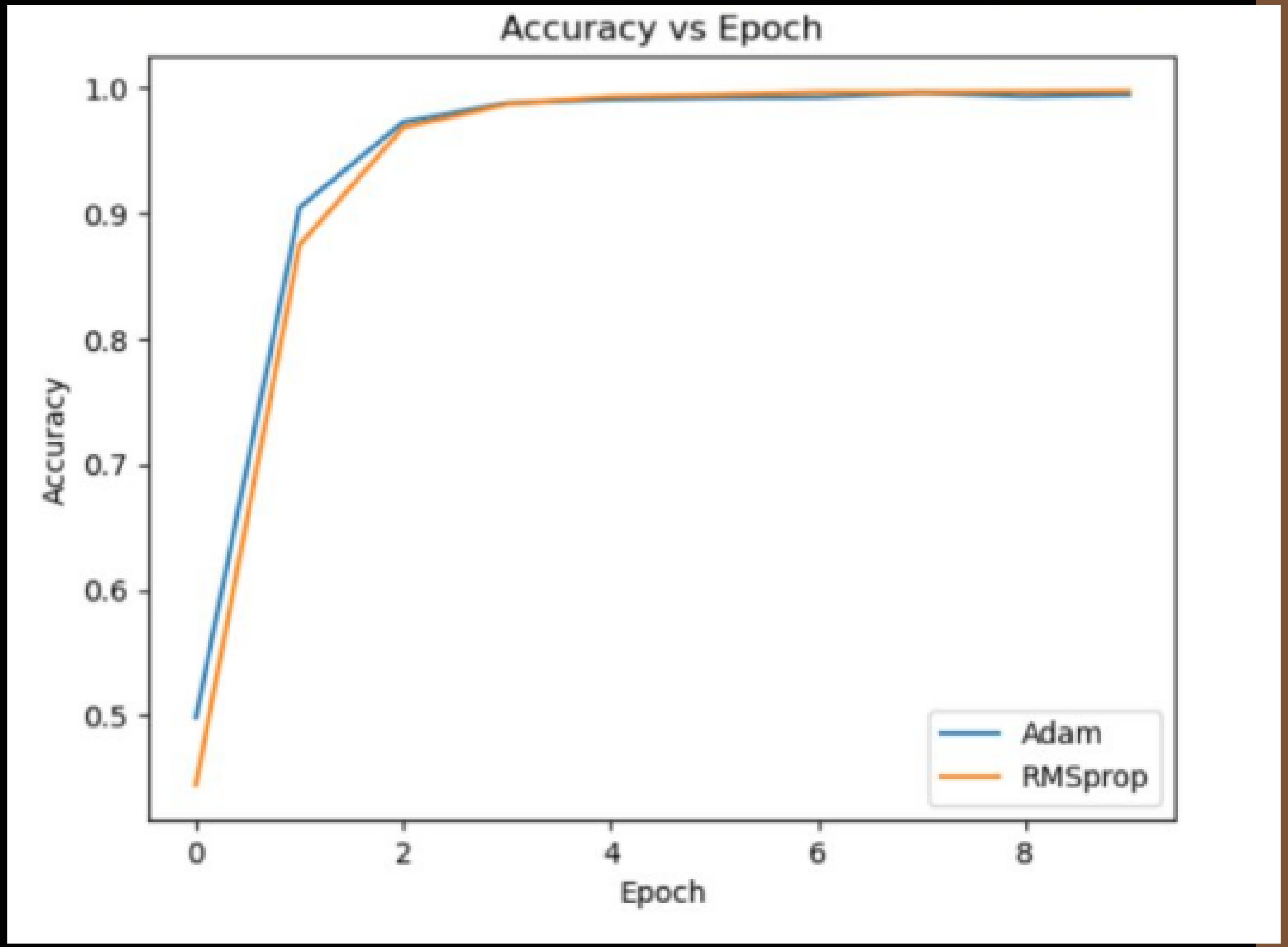
```
model1.compile(optimizer=Adam(learning_rate=0.001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history_adam = model1.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))

model2.compile(optimizer=RMSprop(learning_rate=0.001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history_rmsprop = model2.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))

# Plot the accuracy and loss curves for all three optimizers
plt.plot(history_adam.history['accuracy'], label='Adam')
plt.plot(history_rmsprop.history['accuracy'], label='RMSprop')
plt.title('Accuracy vs Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.plot(history_adam.history['loss'], label='Adam')
plt.plot(history_rmsprop.history['loss'], label='RMSprop')
plt.title('Loss vs Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

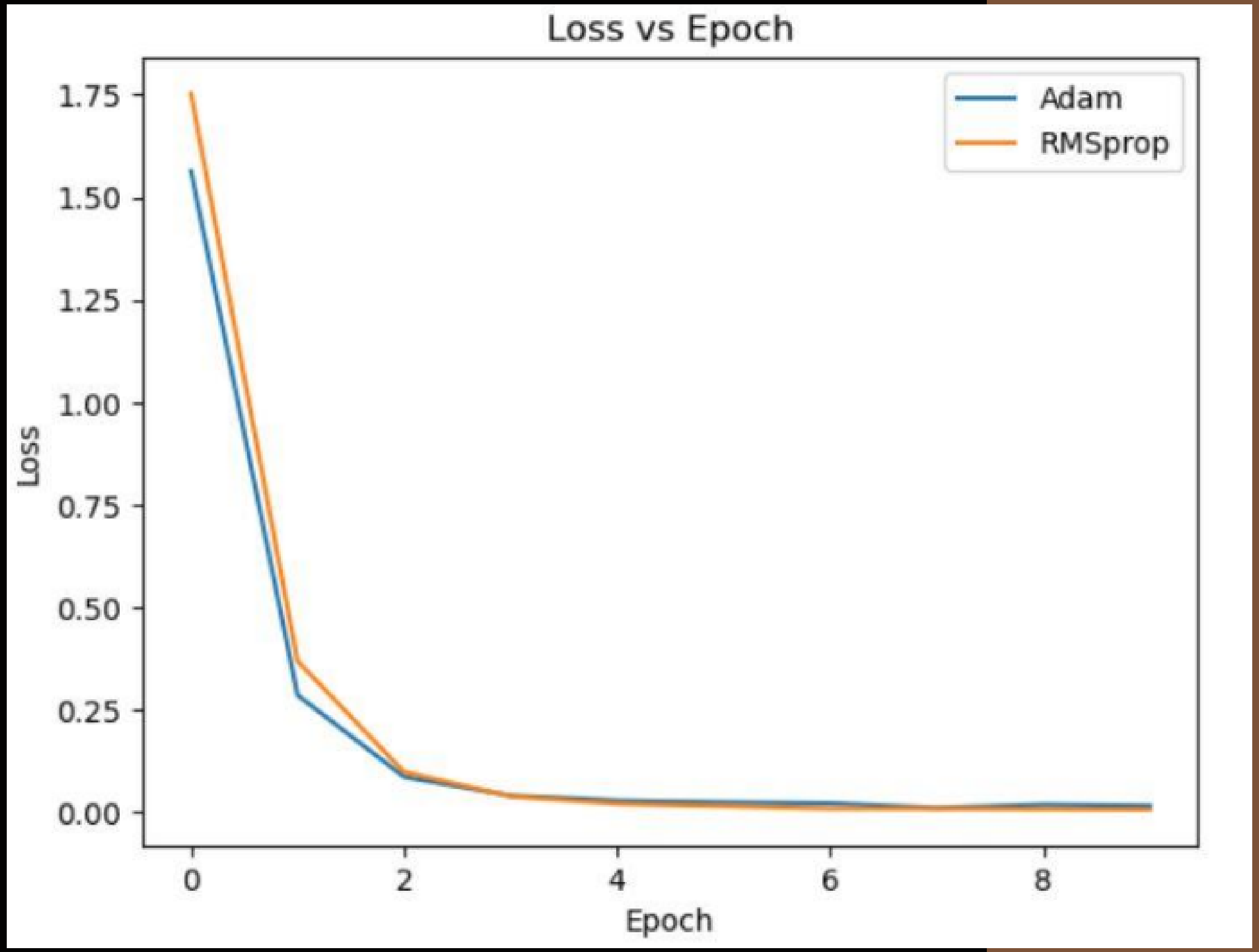
Compiling with optimizers and plots

Results Comparison

From the table, we can see that both models achieve high accuracy on the training data, with the Adam model slightly outperforming the RMSprop model. However, there is a negligible difference in the accuracy of both models on the testing data, with the Adam model having a slightly higher accuracy of 0.9364 compared to the RMSprop model's 0.9309.

Metric	Adam Model	RMSprop Model
Training Accuracy	0.9984	0.9982
Testing Accuracy	0.9364	0.9309
Training Loss	0.0042	0.0044
Testing Loss	0.1942	0.2243





Outcome

The result is the model that can accurately classify sign language images. CNNs are particularly effective for image recognition tasks because they can identify important image features using convolutional and pooling layers, and they can learn to distinguish between different classes of images using fully connected layers.

The model provided an accuracy of approx 97% on the MNIST sign language dataset. This means that the model correctly classified 97% of the images it was given. This demonstrates that CNNs can be an extremely effective tool for recognizing sign language and other types of images.

Learning

CNNs are effective at recognising hand gestures based on specific features such as finger position and hand shape. Because the model requires a significant amount of data to learn the complex patterns and variations in hand gestures, large datasets are critical for improved accuracy. Pre-processing and data augmentation can improve outcomes. Transfer learning can shorten training time and increase accuracy, while optimised hardware and software allows for real-time performance. CNNs have the potential to significantly improve accessibility and communication for the deaf and hard-of-hearing community.

By Rachit Bhalla 21BAI1869 ✦ Natanya Modi 21BAI1405

Thank You So Much!