Handwritten Digit Recognition using Convolutional Neural Networks in Keras with the EMNIST Dataset

Author: Ali Al Bataineh, Ph.D.

Introduction

In this project, we implemented a system for recognizing handwritten digits using a Convolutional Neural Network (CNN). The system is trained on the EMNIST Digits dataset, which consists of images of digits across 10 distinct classes.

Dataset and Preprocessing

The EMNIST Digits dataset is an extension of the original MNIST dataset, containing only digits, with a total of 280,000 images. We loaded the dataset and split it into training and testing sets. Then, we normalized the pixel values by dividing them by 255 to obtain a range of 0 to 1. We also one-hot encoded the labels to prepare them for use with the CNN.

CNN Model Architecture and Training

We designed a CNN architecture that includes convolutional layers, pooling layers, dropout layers for regularization, and fully connected layers. The model is compiled with the categorical_crossentropy loss function, the Adam optimizer, and the accuracy metric. We used early stopping to prevent overfitting and monitored the training time.

Model Evaluation

After training, we evaluated the model's performance on the testing set. We plotted the training and validation accuracy and loss values to visualize the model's performance throughout training.

Prediction

We tested the model on new instances from the test set, displaying the images and making predictions using the saved model. The model correctly predicted the digit classes for the selected images.

Conclusion

This project demonstrates the effectiveness of CNNs for handwritten digit recognition. By training a CNN on the EMNIST Digits dataset, we achieved an accuracy of approximately 99.4% on the test set, enabling accurate recognition of digits from new instances.

Project Code Overview

1. Download and extract the EMNIST dataset
2. Load the dataset and split it into training and testing sets
3. Normalize the pixel values and one-hot encode the labels
4. Define the CNN architecture and compile the model
5. Train the model with early stopping
6. Evaluate the model's performance on the testing set
7. Save the trained model
8. Test the model on new instances from the test set and display the images
9. Make predictions using the saved model and print the results

By following these steps, we built a functional system for recognizing handwritten digits using a Convolutional Neural Network and the EMNIST Digits dataset.

Adapting the Model to Different EMNIST Datasets

The EMNIST dataset is an extension of the original MNIST dataset and consists of six subsets: ByClass, ByMerge, Balanced, Letters, Digits, and MNIST. Each subset has a different number of classes, ranging from 10 (Digits) to 62 (ByClass). To adapt the model for different subsets, the only change needed in the CNN architecture is to adjust the number of units in the final dense layer to match the number of classes in the chosen dataset. For example, if you want to use the EMNIST ByClass dataset with 62 classes, you would need to change the final dense layer to have 62 units, like this: model.add(Dense(62, activation="softmax"))

Keep exploring, experimenting, and refining your skills. Happy learning!

References Cohen, G., Afshar, S., Tapson, J., & van Schaik, A. (2017). EMNIST: an extension of MNIST to handwritten letters. Retrieved from https://arxiv.org/abs/1702.05373

```
import os
import requests
import zipfile

import numpy as np
import matplotlib.pyplot as plt

from scipy.io import loadmat
from sklearn.model_selection import train_test_split

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D,
MaxPooling2D
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.utils import to_categorical

# Download and extract the dataset
url = "https://www.itl.nist.gov/iaui/vip/cs_links/EMNIST/matlab.zip"
response = requests.get(url)

with open("matlab.zip", "wb") as f:
    f.write(response.content)

with zipfile.ZipFile("matlab.zip", "r") as zip_ref:
    zip_ref.extractall(".")

os.remove("matlab.zip")

# Function to load the EMNIST dataset
def load_emnist(file_path):
    data = loadmat(file_path)
    X = data['dataset'][0][0][0][0][0][0]
```

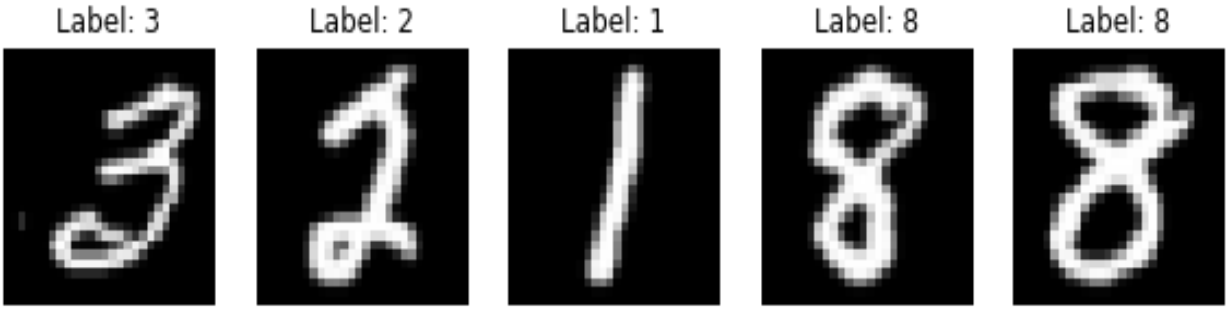
```
    y = data['dataset'][0][0][0][0][0][1]
    X = X.reshape(X.shape[0], 28, 28, 1).transpose([0, 2, 1, 3])
    y = y.reshape(-1)
    return X, y

# Load the dataset and split into training and testing sets
X, y = load_emnist("matlab/emnist-digits.mat")
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Display some sample images
num_samples = 5
fig, axes = plt.subplots(1, num_samples, figsize=(10, 2))

for i in range(num_samples):
    axes[i].imshow(X_train[i].reshape(28, 28), cmap="gray")
    axes[i].set_title(f"Label: {y_train[i]}")
    axes[i].axis("off")

plt.show()
```




```
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
X_train shape: (192000, 28, 28, 1)
y_train shape: (192000,)
X_test shape: (48000, 28, 28, 1)
y_test shape: (48000,)

# Normalize the pixel values
X_train = X_train.astype("float32") / 255
X_test = X_test.astype("float32") / 255

# One-hot encode the labels
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
# Define the CNN architecture
model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation="relu", input_shape=(28,
28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(10, activation="softmax"))

#Compile the model
model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])

model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 dropout (Dropout)           (None, 5, 5, 64)          0

 flatten (Flatten)           (None, 1600)              0

 dense (Dense)               (None, 128)               204928

 dropout_1 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
```

```
Total params: 225,034
Trainable params: 225,034
Non-trainable params: 0
_________________________________________________________________

#Train the model with early stopping
from keras.callbacks import EarlyStopping
import time

early_stopping = EarlyStopping(monitor="val_loss", patience=1, verbose=1)

start_time = time.time()

history = model.fit(X_train, y_train, batch_size=128, epochs=20, verbose=1,
validation_split=0.1, callbacks=[early_stopping])

elapsed_time = time.time() - start_time

print("Training time: {:.2f} seconds".format(elapsed_time))
Epoch 1/20
1350/1350 [==============================] - 65s 48ms/step - loss: 0.1600 -
accuracy: 0.9514 - val_loss: 0.0372 - val_accuracy: 0.9895
Epoch 2/20
1350/1350 [==============================] - 65s 48ms/step - loss: 0.0564 -
accuracy: 0.9842 - val_loss: 0.0314 - val_accuracy: 0.9918
Epoch 3/20
1350/1350 [==============================] - 63s 46ms/step - loss: 0.0436 -
accuracy: 0.9875 - val_loss: 0.0283 - val_accuracy: 0.9922
Epoch 4/20
1350/1350 [==============================] - 64s 48ms/step - loss: 0.0370 -
accuracy: 0.9895 - val_loss: 0.0278 - val_accuracy: 0.9932
Epoch 5/20
1350/1350 [==============================] - 70s 52ms/step - loss: 0.0321 -
accuracy: 0.9911 - val_loss: 0.0256 - val_accuracy: 0.9944
Epoch 6/20
1350/1350 [==============================] - 69s 51ms/step - loss: 0.0292 -
accuracy: 0.9915 - val_loss: 0.0227 - val_accuracy: 0.9954
Epoch 7/20
1350/1350 [==============================] - 71s 53ms/step - loss: 0.0272 -
accuracy: 0.9921 - val_loss: 0.0236 - val_accuracy: 0.9945
Epoch 00007: early stopping
Training time: 467.52 seconds

test_loss, test_acc = model.evaluate(X_test, y_test, verbose=1)
print(f"Test loss: {test_loss}, Test accuracy: {test_acc}")
```

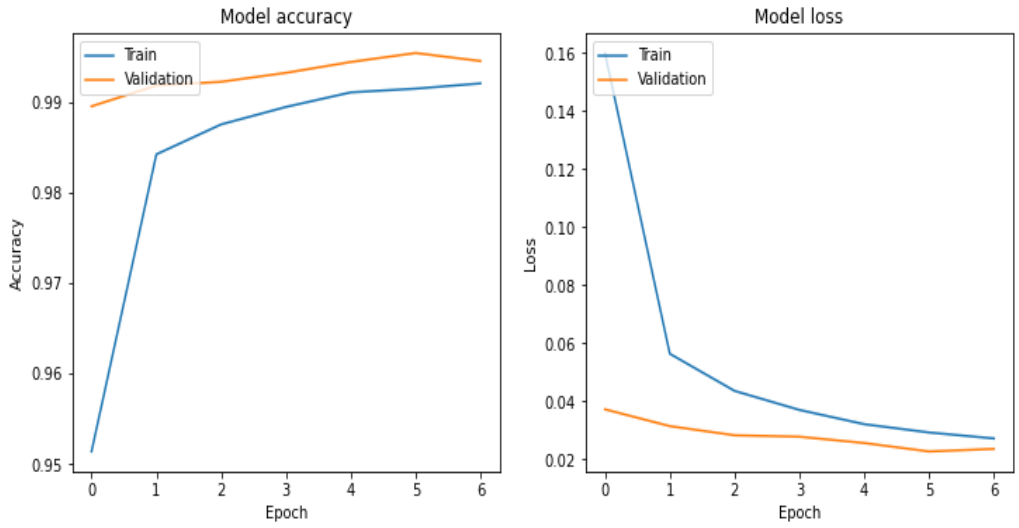
```
1500/1500 [==============================] - 10s 6ms/step - loss: 0.0240 -
accuracy: 0.9938
Test loss: 0.02398458868265152, Test accuracy: 0.9938333630561829

import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.show()
```



This code will generate two plots: one for the training and validation accuracy, and another for the training and validation loss. The x-axis represents the epoch number, while the y-axis represents the accuracy or loss value. The training and validation data are plotted separately, allowing you to see how well the model is generalizing to the validation data.

```
#Save the trained model
model.save("emnist_digits_model.h5")
```

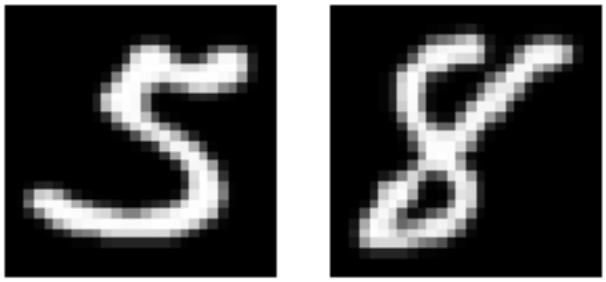
Let's now test the model on new instances from the test set, we first display the images and then make predictions using the saved model. Here's a step-by-step guide on how to do this

```
#Load the saved model
from keras.models import load_model

model = load_model("emnist_digits_model.h5")

#Select 2 random instances from the test set and display them
num_samples = 2
random_indices = np.random.choice(X_test.shape[0], size=num_samples,
replace=False)

# Plot the images
fig, axes = plt.subplots(1, num_samples, figsize=(4, 2))
for i, idx in enumerate(random_indices):
    axes[i].imshow(X_test[idx].reshape(28, 28), cmap='gray')
    axes[i].axis('off')
plt.show()
```



Make predictions on the selected instances and print the results

```
selected_images = X_test[random_indices]

# Make predictions on preprocessed test images
predictions = model.predict(selected_images)

# Get the predicted class for each instance
predicted_classes = np.argmax(predictions, axis=1)

print("Predicted classes:", predicted_classes)
```

```
Predicted classes: [5 8]
```

Fantastic! It appears that our model is making correct predictions for the two images.

Keep exploring, experimenting, and refining your skills. Happy learning!