Image Classification of an American Sign Language Dataset

In this section, we will perform the data preparation, model creation, and model training steps we observed in the last section using a different dataset: images of hands making letters in American Sign Language (http://www.asl.gs/).

Objectives

- Prepare image data for training
- Create and compile a simple model for image classification
- Train an image classification model and observe the results

American Sign Language Dataset

The American Sign Language alphabet (http://www.asl.gs/) contains 26 letters. Two of those letters (j and z) require movement, so they are not included in the training dataset.

Kaggle

This dataset is available from the website Kaggle (http://www.kaggle.com), which is a fantastic place to find datasets and other deep learning resources. In addition to providing resources like datasets and "kernels" that are like these notebooks, Kaggle hosts competitions that you can take part in, competing with others in training highly accurate models.

If you're looking to practice or see examples of many deep learning projects, Kaggle is a great site to visit.

Loading the Data

This dataset is not available via Keras in the same way that MNIST is, so let's learn how to load custom data. By the end of this section we will have `x_train`, `y_train`, `x_valid`, and `y_valid` variables as before.

Reading in the Data

The sign language dataset is in CSV (https://en.wikipedia.org/wiki/Comma-separated_values) (Comma Separated Values) format, the same data structure behind Microsoft Excel and Google Sheets. It is a grid of rows and columns with labels at the top, as seen in the train (data/asl_data/sign_mnist_train.csv) and valid (data/asl_data/sign_mnist_valid.csv) datasets (they may take a moment to load).

To load and work with the data, we'll be using a library called Pandas (https://pandas.pydata.org/), which is a highly performant tool for loading and manipulating data. We'll read the CSV files into a format called a DataFrame (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html).

In [ ]:

```
import pandas as pd
```

Pandas has a read_csv (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html) method that expects a csv file, and returns a DataFrame:

In [ ]:

```
train_df = pd.read_csv("data/asl_data/sign_mnist_train.csv")
valid_df = pd.read_csv("data/asl_data/sign_mnist_valid.csv")
```

Exploring the Data

Let's take a look at our data. We can use the head (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.head.html) method to print the first few rows of the DataFrame. Each row is an image which has a `label` column, and also, 784 values representing each pixel value in the image, just like with the MNIST dataset. Note that the labels currently are numerical values, not letters of the alphabet:

In [ ]:

```
train_df.head()
```

Extracting the Labels

As with MNIST, we would like to store our training and validation labels in `y_train` and `y_valid` variables. Here we create those variables and then delete the labels from our original dataframes, where they are no longer needed:

In [ ]:

```
y_train = train_df['label']
y_valid = valid_df['label']
del train_df['label']
del valid_df['label']
```

Extracting the Images

As with MNIST, we would like to store our training and validation images in `x_train` and `x_valid` variables. Here we create those variables:

In [ ]:

```
x_train = train_df.values
x_valid = valid_df.values
```

Summarizing the Training and Validation Data

We now have 27,455 images with 784 pixels each for training...

In [ ]:

```
x_train.shape
```

...as well as their corresponding labels:

In [ ]:

```
y_train.shape
```

For validation, we have 7,172 images...

In [ ]:

```
x_valid.shape
```

...and their corresponding labels:

In [ ]:

```
y_valid.shape
```

Visualizing the Data

To visualize the images, we will again use the matplotlib library. We don't need to worry about the details of this visualization, but if interested, you can learn more about matplotlib (https://matplotlib.org/) at a later time.

Note that we'll have to reshape the data from its current 1D shape of 784 pixels, to a 2D shape of 28x28 pixels to make sense of the image:

In [ ]:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(40,40))

num_images = 20
for i in range(num_images):
    row = x_train[i]
    label = y_train[i]

    image = row.reshape(28,28)
    plt.subplot(1, num_images, i+1)
    plt.title(label, fontdict={'fontsize': 30})
    plt.axis('off')
    plt.imshow(image, cmap='gray')
```

Exercise: Normalize the Image Data

As we did with the MNIST dataset, we are going to normalize the image data, meaning that their pixel values, instead of being between 0 and 255 as they are currently:

In [ ]:

```
x_train.min()
```

In [ ]:

```
x_train.max()
```

...should be floating point values between 0 and 1. Use the following cell to work. If you get stuck, look at the solution below.

In [ ]:

```
# TODO: Normalize x_train and x_valid.
```

Solution

Click on the '...' below to show the solution.

```
x_train = x_train / 255
x_valid = x_valid / 255
```

Exercise: Categorize the Labels

As we did with the MNIST dataset, we are going to categorically encode the labels. Recall that we can use the keras.utils.to_categorical (https://www.tensorflow.org/api_docs/python/tf/keras/utils/to_categorical) method to accomplish this by passing it the values to encode, and, the number of categories to encode it into. Do your work in the cell below. We have imported `keras` and set the number of categories (24) for you.

In [ ]:

```
import tensorflow.keras as keras
num_classes = 24
```

In [ ]:

```
# TODO: Categorically encode y_train and y_valid.
```

Solution

Click on the '...' below to show the solution.

```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_valid = keras.utils.to_categorical(y_valid, num_classes)
```

Exercise: Build the Model

The data is all prepared, we have normalized images for training and validation, as well as categorically encoded labels for training and validation.

For this exercise we are going to build a sequential model. Just like last time, build a model that:

- Has a dense input layer. This layer should contain 512 neurons, use the `relu` activation function, and expect input images with a shape of `(784,)`
- Has a second dense layer with 512 neurons which uses the `relu` activation function
- Has a dense output layer with neurons equal to the number of classes, using the `softmax` activation function

Do your work in the cell below, creating a `model` variable to store the model. We've imported the Keras Sequental (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) model class and Dense (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense) layer class to get you started. Reveal the solution below for a hint:

In [ ]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

In [ ]:

```
# TODO: build a model following the guidelines above.
```

Solution

Click on the '...' below to show the solution.

```
model = Sequential()
model.add(Dense(units = 512, activation='relu', input_shape=(784,)))
model.add(Dense(units = 512, activation='relu'))
model.add(Dense(units = num_classes, activation='softmax'))
```

Summarizing the Model

Run the cell below to summarize the model you just created:

In [ ]:

```
model.summary()
```

Compiling the Model

We'll compile (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential#compile) our model with the same options as before, using categorical crossentropy (https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy) to reflect the fact that we want to fit into one of many categories, and measuring the accuracy of our model:

In [ ]:

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

Exercise: Train the Model

Use the model's `fit` method to train it for 20 epochs using the training and validation images and labels created above:

In [ ]:

```
# TODO: Train the model for 20 epochs.
```

Solution

Click on the '...' below to show the solution.

```
model.fit(x_train, y_train, epochs=20, verbose=1, validation_data=(x_valid, y_valid
))
```

Discussion: What happened?

We can see that the training accuracy got to a fairly high level, but the validation accuracy was not as high. What happened here?

Think about it for a bit before clicking on the '...' below to reveal the answer.

This is an example of the model learning to categorize the training data, but performing poorly against new data that it has not been trained on. Essentially, it is memorizing the dataset, but not gaining a robust and general understanding of the problem. This is a common issue called *overfitting*. We will discuss overfitting in the next two lectures, as well as some ways to address it.

Summary

In this section you built your own neural network to perform image classification that is quite accurate. Congrats!

At this point we should be getting somewhat familiar with the process of loading data (incuding labels), preparing it, creating a model, and then training the model with prepared data.

Clear the Memory

Before moving on, please execute the following cell to clear up the GPU memory. This is required to move on to the next notebook.

In [ ]:

```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Next

Now that you have built some very basic, somewhat effective models, we will begin to learn about more sophisticated models, including *Convolutional Neural Networks*.