

- Week 4: Multi-class Classification

Welcome to this assignment! In this exercise, you will get a chance to work on a multi-class classification problem. You will be using the [Sign Language MNIST](#) dataset, which contains 28x28 images of hands depicting the 26 letters of the English alphabet.

You will need to pre-process the data so that it can be fed into your convolutional neural network to correctly classify each image as the letter it represents.

Let's get started!

NOTE: To prevent errors from the autograder, please avoid editing or deleting non-graded cells in this notebook . Please only put your solutions in between the `### START CODE HERE` and `### END CODE HERE` code comments, and refrain from adding any new cells.

```
# grader-required-cell

import csv
import string
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator, array_to_img
```

Download the training and test sets (the test set will actually be used as a validation set):

```
# sign_mnist_train.csv
!gdown --id 1z0DkA9Byt1Lx01C0BAWzknLyQmZAp0HR
# sign_mnist_test.csv
!gdown --id 1z1BIj4qmri59GWBG4ivMNFtpZ4AXIbzg

/usr/local/lib/python3.10/dist-packages/gdown/cli.py:121: FutureWarning: Option `--id` was deprecated in version 4.3.1 and will be removed in a future version.
  warnings.warn(
Downloading...
From: https://drive.google.com/uc?id=1z0DkA9Byt1Lx01C0BAWzknLyQmZAp0HR
To: /content/sign_mnist_train.csv
100% 83.3M/83.3M [00:02<00:00, 37.6MB/s]
/usr/local/lib/python3.10/dist-packages/gdown/cli.py:121: FutureWarning: Option `--id` was deprecated in version 4.3.1 and will be removed in a future version.
  warnings.warn(
Downloading...
From: https://drive.google.com/uc?id=1z1BIj4qmri59GWBG4ivMNFtpZ4AXIbzg
To: /content/sign_mnist_test.csv
100% 21.8M/21.8M [00:00<00:00, 52.3MB/s]
```

Define some globals with the path to both files you just downloaded:

```
# grader-required-cell

TRAINING_FILE = './sign_mnist_train.csv'
VALIDATION_FILE = './sign_mnist_test.csv'
```

Unlike previous assignments, you will not have the actual images provided, instead you will have the data serialized as `csv` files.

Take a look at how the data looks like within the `csv` file:

```
# grader-required-cell

with open(TRAINING_FILE) as training_file:
    line = training_file.readline()
    print(f"First line (header) looks like this:\n{line}")
    line = training_file.readline()
    print(f"Each subsequent line (data points) look like this:\n{line}")

    First line (header) looks like this:
    label,pixel1,pixel2,pixel3,pixel4,pixel5,pixel6,pixel7,pixel8,pixel9,pixel10,pixel11,pixel12,pixel13,pixel14,pixel15,pixel16,pixel17

    Each subsequent line (data points) look like this:
    3,107,118,127,134,139,143,146,150,153,156,158,160,163,165,159,166,168,170,170,171,171,171,172,171,171,170,170,169,111,121,129,135,14
```

As you can see, each file includes a header (the first line) and each subsequent data point is represented as a line that contains 785 values.

The first value is the label (the numeric representation of each letter) and the other 784 values are the value of each pixel of the image. Remember that the original images have a resolution of 28x28, which sums up to 784 pixels.

✓ Parsing the dataset

Now complete the `parse_data_from_input` below.

This function should be able to read a file passed as input and return 2 numpy arrays, one containing the labels and one containing the 28x28 representation of each image within the file. These numpy arrays should have type `float64`.

A couple of things to keep in mind:

- The first line contains the column headers, so you should ignore it.
- Each successive line contains 785 comma-separated values between 0 and 255
 - The first value is the label
 - The rest are the pixel values for that picture

Hint:

You have two options to solve this function.

- 1. One is to use `csv.reader` and create a for loop that reads from it, if you take this approach take this into consideration:
 - `csv.reader` returns an iterable that returns a row of the csv file in each iteration. Following this convention, `row[0]` has the label and `row[1:]` has the 784 pixel values.
 - To reshape the arrays (going from 784 to 28x28), you can use functions such as [np.array_split](#) or [np.reshape](#).
 - For type conversion of the numpy arrays, use the method [np.ndarray.astype](#).
- 2. The other one is to use `np.loadtxt`. You can find the documentation [here](#).

Regardless of the method you chose, your function should finish its execution in under 1 minute. If you see that your function is taking a long time to run, try changing your implementation.

```
# grader-required-cell

# GRADED FUNCTION: parse_data_from_input
def parse_data_from_input(filename):
    """
    Parses the images and labels from a CSV file

    Args:
        filename (string): path to the CSV file

    Returns:
        images, labels: tuple of numpy arrays containing the images and labels
    """
    with open(filename) as file:
        ### START CODE HERE

        # Use csv.reader, passing in the appropriate delimiter
        # Remember that csv.reader can be iterated and returns one line in each iteration
        csv_reader = csv.reader(file, delimiter=' ')

        labels = []
        images = []
        count = 0
        for row in csv_reader:
            if (count > 0):
                row = row[0].split(',')
                row = list(map(float, row))
                # import pdb; pdb.set_trace()
                images.append(row[1:])
                labels.append(row[0])
                count += 1

        # reshape the images
        images, labels = np.array(images), np.array(labels)
        to_size = int(images.shape[1]**0.5)
        images = np.reshape(images, (images.shape[0], to_size, to_size))

        ### END CODE HERE

    return images, labels
```

```
# grader-required-cell
```

```
# Test your function
training_images, training_labels = parse_data_from_input(TRAINING_FILE)
validation_images, validation_labels = parse_data_from_input(VALIDATION_FILE)

print(f"Training images has shape: {training_images.shape} and dtype: {training_images.dtype}")
print(f"Training labels has shape: {training_labels.shape} and dtype: {training_labels.dtype}")
print(f"Validation images has shape: {validation_images.shape} and dtype: {validation_images.dtype}")
print(f"Validation labels has shape: {validation_labels.shape} and dtype: {validation_labels.dtype}")

Training images has shape: (27455, 28, 28) and dtype: float64
Training labels has shape: (27455,) and dtype: float64
Validation images has shape: (7172, 28, 28) and dtype: float64
Validation labels has shape: (7172,) and dtype: float64
```

Expected Output:

```
Training images has shape: (27455, 28, 28) and dtype: float64
Training labels has shape: (27455,) and dtype: float64
Validation images has shape: (7172, 28, 28) and dtype: float64
Validation labels has shape: (7172,) and dtype: float64
```

✓ Visualizing the numpy arrays

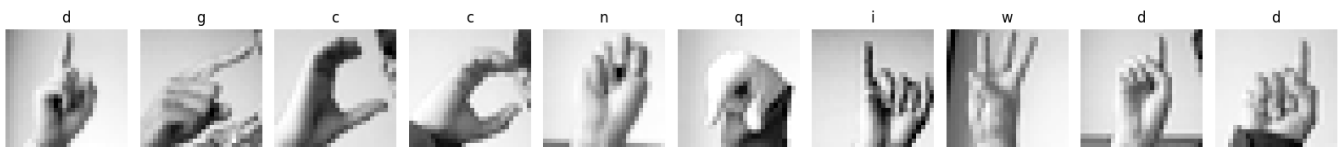
Now that you have converted the initial csv data into a format that is compatible with computer vision tasks, take a moment to actually see how the images of the dataset look like:

```
# Plot a sample of 10 images from the training set
def plot_categories(training_images, training_labels):
    fig, axes = plt.subplots(1, 10, figsize=(16, 15))
    axes = axes.flatten()
    letters = list(string.ascii_lowercase)

    for k in range(10):
        img = training_images[k]
        img = np.expand_dims(img, axis=-1)
        img = array_to_img(img)
        ax = axes[k]
        ax.imshow(img, cmap="Greys_r")
        ax.set_title(f"{letters[int(training_labels[k])]}")
        ax.set_axis_off()

    plt.tight_layout()
    plt.show()

plot_categories(training_images, training_labels)
```



✓ Creating the generators for the CNN

Now that you have successfully organized the data in a way that can be easily fed to Keras' `ImageDataGenerator`, it is time for you to code the generators that will yield batches of images, both for training and validation. For this complete the `train_val_generators` function below.

Some important notes:

- The images in this dataset come in the same resolution so you don't need to set a custom `target_size` in this case. In fact, you can't even do so because this time you will not be using the `flow_from_directory` method (as in previous assignments). Instead you will use the [flow](#) method.
- You need to add the "color" dimension to the numpy arrays that encode the images. These are black and white images, so this new dimension should have a size of 1 (instead of 3, which is used when dealing with colored images). Take a look at the function [np.expand_dims](#) for this.

```
# grader-required-cell
```

```

# GRADED FUNCTION: train_val_generators
def train_val_generators(training_images, training_labels, validation_images, validation_labels):
    """
    Creates the training and validation data generators

    Args:
        training_images (array): parsed images from the train CSV file
        training_labels (array): parsed labels from the train CSV file
        validation_images (array): parsed images from the test CSV file
        validation_labels (array): parsed labels from the test CSV file

    Returns:
        train_generator, validation_generator - tuple containing the generators
    """
    ### START CODE HERE

    # In this section you will have to add another dimension to the data
    # So, for example, if your array is (10000, 28, 28)
    # You will need to make it (10000, 28, 28, 1)
    # Hint: np.expand_dims
    training_images = np.expand_dims(training_images, axis=3)
    validation_images = np.expand_dims(validation_images, axis=3)

    # Instantiate the ImageDataGenerator class
    # Don't forget to normalize pixel values
    # and set arguments to augment the images (if desired)
    train_datagen = ImageDataGenerator(
        rescale = 1./255,
        rotation_range=10,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.1,
        zoom_range=0.3,
        horizontal_flip=False,
        fill_mode='nearest')

    # Pass in the appropriate arguments to the flow method
    train_generator = train_datagen.flow(x=training_images,
                                         y=training_labels,
                                         batch_size=32)

    # Instantiate the ImageDataGenerator class (don't forget to set the rescale argument)
    # Remember that validation data should not be augmented
    validation_datagen = ImageDataGenerator(rescale = 1./255)

    # Pass in the appropriate arguments to the flow method
    validation_generator = validation_datagen.flow(x=validation_images,
                                                  y=validation_labels,
                                                  batch_size=32)

    ### END CODE HERE

    return train_generator, validation_generator

# grader-required-cell

# Test your generators
train_generator, validation_generator = train_val_generators(training_images, training_labels, validation_images, validation_labels)

print(f"Images of training generator have shape: {train_generator.x.shape}")
print(f"Labels of training generator have shape: {train_generator.y.shape}")
print(f"Images of validation generator have shape: {validation_generator.x.shape}")
print(f"Labels of validation generator have shape: {validation_generator.y.shape}")

    Images of training generator have shape: (27455, 28, 28, 1)
    Labels of training generator have shape: (27455,)
    Images of validation generator have shape: (7172, 28, 28, 1)
    Labels of validation generator have shape: (7172,)

```

Expected Output:

```

Images of training generator have shape: (27455, 28, 28, 1)
Labels of training generator have shape: (27455,)
Images of validation generator have shape: (7172, 28, 28, 1)
Labels of validation generator have shape: (7172,)

```

✓ Coding the CNN

One last step before training is to define the architecture of the model.

Complete the `create_model` function below. This function should return a Keras' model that uses the `Sequential` or the `Functional` API.

The last layer of your model should have a number of units equal to the number of letters in the English alphabet. It should also use an activation function that will output the probabilities per letter.

Note: The [documentation](#) of the dataset mentions that there are actually no cases for the last letter, Z, and this will allow you to reduce the recommended number of output units above by one. If you're not yet convinced, you can safely ignore this fact for now and study it later. You will pass the assignment even without this slight optimization.

Aside from defining the architecture of the model, you should also compile it so make sure to use a `loss` function that is suitable for multi-class classification.

Note that you should use no more than 2 Conv2D and 2 MaxPooling2D layers to achieve the desired performance.

```
# grader-required-cell

def create_model():

    ### START CODE HERE

    # Define the model
    # Use no more than 2 Conv2D and 2 MaxPooling2D
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(2, 2),

        tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
        tf.keras.layers.MaxPooling2D(2,2),

        tf.keras.layers.Flatten(),
        # tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(512, activation='relu'),
        # tf.keras.layers.Dense(1)
        tf.keras.layers.Dense(26, activation='softmax')
    ])

    model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
                  loss = 'sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    ### END CODE HERE

    return model

# Save your model
model = create_model()

# Train your model
history = model.fit(train_generator,
                    epochs=15,
                    validation_data=validation_generator)

Epoch 1/15
858/858 [=====] - 26s 19ms/step - loss: 2.0851 - accuracy: 0.3565 - val_loss: 0.8918 - val_accuracy: 0.7141
Epoch 2/15
858/858 [=====] - 16s 18ms/step - loss: 1.0006 - accuracy: 0.6715 - val_loss: 0.3865 - val_accuracy: 0.8721
Epoch 3/15
858/858 [=====] - 16s 19ms/step - loss: 0.6513 - accuracy: 0.7820 - val_loss: 0.2478 - val_accuracy: 0.9071
Epoch 4/15
858/858 [=====] - 16s 19ms/step - loss: 0.4905 - accuracy: 0.8366 - val_loss: 0.1755 - val_accuracy: 0.9381
Epoch 5/15
858/858 [=====] - 16s 19ms/step - loss: 0.3768 - accuracy: 0.8779 - val_loss: 0.1016 - val_accuracy: 0.9641
Epoch 6/15
858/858 [=====] - 17s 19ms/step - loss: 0.3189 - accuracy: 0.8924 - val_loss: 0.0814 - val_accuracy: 0.9811
Epoch 7/15
858/858 [=====] - 16s 19ms/step - loss: 0.2673 - accuracy: 0.9128 - val_loss: 0.0548 - val_accuracy: 0.9801
Epoch 8/15
858/858 [=====] - 18s 21ms/step - loss: 0.2417 - accuracy: 0.9194 - val_loss: 0.0889 - val_accuracy: 0.9741
Epoch 9/15
858/858 [=====] - 16s 19ms/step - loss: 0.2133 - accuracy: 0.9283 - val_loss: 0.0594 - val_accuracy: 0.9791
Epoch 10/15
858/858 [=====] - 16s 18ms/step - loss: 0.1894 - accuracy: 0.9363 - val_loss: 0.0538 - val_accuracy: 0.9781
Epoch 11/15
858/858 [=====] - 16s 18ms/step - loss: 0.1724 - accuracy: 0.9430 - val_loss: 0.0322 - val_accuracy: 0.9901
Epoch 12/15
858/858 [=====] - 16s 19ms/step - loss: 0.1592 - accuracy: 0.9480 - val_loss: 0.0325 - val_accuracy: 0.9901
Epoch 13/15
```

```

858/858 [=====] - 16s 18ms/step - loss: 0.1549 - accuracy: 0.9461 - val_loss: 0.0446 - val_accuracy: 0.9845
Epoch 14/15
858/858 [=====] - 16s 19ms/step - loss: 0.1425 - accuracy: 0.9535 - val_loss: 0.0294 - val_accuracy: 0.9925
Epoch 15/15
858/858 [=====] - 16s 19ms/step - loss: 0.1346 - accuracy: 0.9568 - val_loss: 0.0127 - val_accuracy: 0.9975

```

Now take a look at your training history:

```

# Plot the chart for accuracy and loss on both training and validation
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

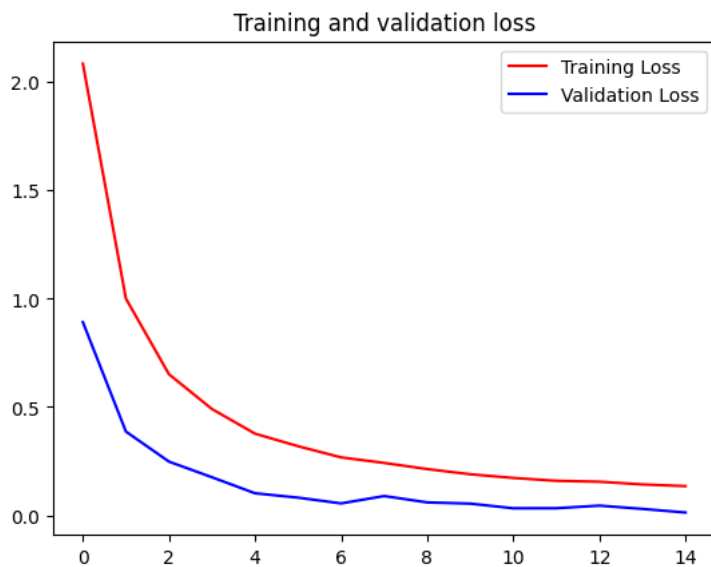
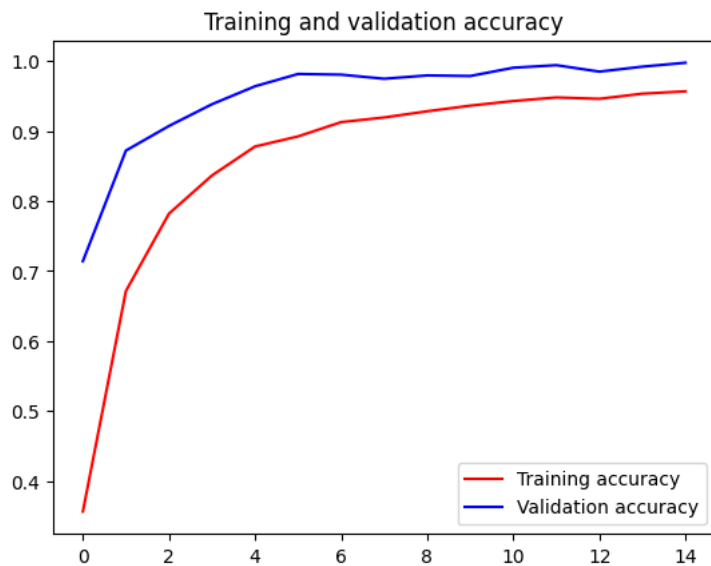
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'r', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()

plt.plot(epochs, loss, 'r', label='Training Loss')
plt.plot(epochs, val_loss, 'b', label='Validation Loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()

```



You will not be graded based on the accuracy of your model but try making it as high as possible for both training and validation, as an optional exercise, **after submitting your notebook for grading.**

A reasonable benchmark is to achieve over 99% accuracy for training and over 95% accuracy for validation within 15 epochs. Try tweaking your model's architecture or the augmentation techniques to see if you can achieve these levels of accuracy.

✓ Download your notebook for grading

You will need to submit your solution notebook for grading. The following code cells will check if this notebook's grader metadata (i.e. hidden data in the notebook needed for grading) is not modified by your workspace. This will ensure that the autograder can evaluate your code properly. Depending on its output, you will either:

- *if the metadata is intact*: Download the current notebook. Click on the File tab on the upper left corner of the screen then click on Download -> Download .ipynb. You can name it anything you want as long as it is a valid .ipynb (jupyter notebook) file.
- *if the metadata is missing*: A new notebook with your solutions will be created on this Colab workspace. It should be downloaded automatically and you can submit that to the grader.

```
# Download metadata checker
!wget -nc https://storage.googleapis.com/tensorflow-1-public/colab_metadata_checker.py

--2023-11-18 06:20:19-- https://storage.googleapis.com/tensorflow-1-public/colab_metadata_checker.py
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.130.207, 74.125.68.207, 64.233.170.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.130.207|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1997 (2.0K) [text/x-python-script]
Saving to: 'colab_metadata_checker.py'

colab_metadata_chec 100%[=====] 1.95K --.-KB/s in 0s

2023-11-18 06:20:20 (27.8 MB/s) - 'colab_metadata_checker.py' saved [1997/1997]

import colab_metadata_checker

# Please see the output of this cell to see which file you need to submit to the grader
colab_metadata_checker.run('C2W4_Assignment_fixed.ipynb')

Grader metadata detected! You can download this notebook by clicking `File > Download > Download as .ipynb` and submit it to the grader
```

Please disregard the following note if the notebook metadata is detected

Note: Just in case the download fails for the second point above, you can also do these steps:

- Click the Folder icon on the left side of this screen to open the File Manager.
- Click the Folder Refresh icon in the File Manager to see the latest files in the workspace. You should see a file ending with a `_fixed.ipynb`.
- Right-click on that file to save locally and submit it to the grader.

Congratulations on finishing this week's assignment!

You have successfully implemented a convolutional neural network that is able to perform multi-class classification tasks! Nice job!

Keep it up!