Header

2. 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.

2.1 Objectives

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

In [1]:
```
import torch.nn as nn
import pandas as pd
import torch
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.is_available()
```

Out[1]:
```
True
```

2.2 American Sign Language Dataset

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

No description has been provided for this image

2.2.1 Kaggle

This dataset is available from the website Kaggle, 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.

2.3 Loading the Data

This dataset is not available via TorchVision 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.

2.3.1 Reading in the Data

The sign language dataset is in CSV (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 and valid datasets (they may take a moment to load).

To load and work with the data, we'll be using a library called Pandas, which is a highly performant tool for loading and manipulating data. We'll read the CSV files into a format called a DataFrame.

Pandas has a read_csv method that expects a csv file, and returns a DataFrame:

In [2]:
```
train_df = pd.read_csv("data/asl_data/sign_mnist_train.csv")
valid_df = pd.read_csv("data/asl_data/sign_mnist_valid.csv")
```

2.3.2 Exploring the Data

Let's take a look at our data. We can use the head 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 [3]:
```
train_df.head()
```

Out[3]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel7
0	3	107	118	127	134	139	143	146	150	153	...	2
1	6	155	157	156	156	156	157	156	158	158	...	
2	2	187	188	188	187	187	186	187	188	187	...	2
3	2	211	211	212	212	211	210	211	210	210	...	2
4	12	164	167	170	172	176	179	180	184	185	...	

5 rows × 785 columns

2.3.3 Extracting the Labels

Let's store our training and validation labels in `y_train` and `y_valid` variables. We can use the pop method to remove a column from our DataFrame and assign the removed values to a variable.

In [4]:
```
y_train = train_df.pop('label')
y_valid = valid_df.pop('label')
y_train
```

Out[4]:
```
0         3
1         6
2         2
3         2
4        12
         ..
27450    12
27451    22
27452    17
27453    16
27454    22
Name: label, Length: 27455, dtype: int64
```

2.3.4 Extracting the Images

Next, let's store our training and validation images in `x_train` and `x_valid` variables. Here we create those variables:

In [5]:
```
x_train = train_df.values
x_valid = valid_df.values
x_train
```

Out[5]:
```
array([[107, 118, 127, ..., 204, 203, 202],
       [155, 157, 156, ..., 103, 135, 149],
       [187, 188, 188, ..., 195, 194, 195],
       ...,
       [174, 174, 174, ..., 202, 200, 200],
       [177, 181, 184, ...,  64,  87,  93],
       [179, 180, 180, ..., 205, 209, 215]])
```

2.3.5 Summarizing the Training and Validation Data

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

In [6]:
```
x_train.shape
```

Out[6]:
```
(27455, 784)
```

...as well as their corresponding labels:

In [7]:
```
y_train.shape
```

Out[7]:
```
(27455,)
```

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

In [8]:
```
x_valid.shape
```

Out[8]:
```
(7172, 784)
```

...and their corresponding labels:

In [9]:
```
y_valid.shape
```

Out[9]:
```
(7172,)
```

2.4 Visualizing the Data

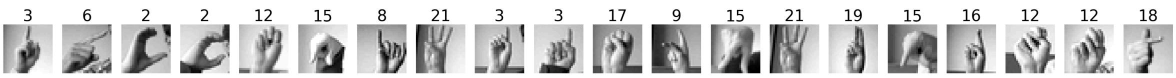
To visualize the images, we will again use the matplotlib library. We don't need to worry about the details of this visualization, but if interested, you can learn more about matplotlib at a later time.

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 [10]:
```
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')
```



2.4.1 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 [11]:
```
x_train.min()
```

Out[11]:
```
0
```

In [12]:
```
x_train.max()
```

Out[12]:
```
255
```

In the previous lab, we used ToTensor, but we can also modify our data before turning it into a tensor.

In [13]:
```
x_train = train_df.values / 255
x_valid = valid_df.values / 255
```

2.4.2 Custom Datasets

We can use PyTorch's Dataset tools in order to create our own dataset. `__init__` will run once when the class is initialized. `__getitem__` returns our images and labels.

Since our dataset is small enough, we can store it on our GPU for faster processing. In the previous lab, we sent our data to the GPU when it was drawn from each batch. Here, we will send it to the GPU in the `__init__` function.

In [14]:
```
class MyDataset(Dataset):
    def __init__(self, x_df, y_df):
        self.xs = torch.tensor(x_df).float().to(device)
        self.ys = torch.tensor(y_df).to(device)

    def __getitem__(self, idx):
        x = self.xs[idx]
        y = self.ys[idx]
        return x, y

    def __len__(self):
        return len(self.xs)
```

A custom PyTorch dataset works just like a prebuilt one. It should be passed to a DataLoader for model training.

In [15]:
```
BATCH_SIZE = 32

train_data = MyDataset(x_train, y_train)
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True)
train_N = len(train_loader.dataset)
```

In [16]:
```
valid_data = MyDataset(x_valid, y_valid)
valid_loader = DataLoader(valid_data, batch_size=BATCH_SIZE)
valid_N = len(valid_loader.dataset)
```

We can verify the DataLoader works as expected with the code below. We'll make the DataLoader iterable, and then call next to draw the first hand from the deck.

In [17]:
```
train_loader
```

Out[17]:
```
<torch.utils.data.dataloader.DataLoader at 0x7fe5911fe200>
```

Try running the below a few times. The values should change each time.

In [18]:
```
batch = next(iter(train_loader))
batch
```

Out[18]:
```
[tensor([[0.6549, 0.6667, 0.6745,  ..., 0.7765, 0.7686, 0.7608],
         [0.6941, 0.7098, 0.7137,  ..., 0.8000, 0.7882, 0.7922],
         [0.1569, 0.1686, 0.1843,  ..., 0.0000, 0.0000, 0.0353],
         ...,
         [0.5804, 0.6000, 0.6235,  ..., 0.7843, 0.7843, 0.7725],
         [0.3294, 0.3686, 0.2314,  ..., 0.6157, 0.6275, 0.6314],
         [0.6392, 0.6549, 0.6706,  ..., 0.7686, 0.7647, 0.7647]],
        device='cuda:0'),
 tensor([23, 15, 19, 10, 11,  9, 18,  2, 15, 23,  7, 15, 16, 16,  9,  5,  7, 1
6,
         20,  0,  5,  8, 21,  1, 10, 23, 20, 13, 15, 20,  0, 12],
        device='cuda:0')]
```

Notice the batch has two values. The first is our `x`, and the second is our `y`. The first dimension of each should have `32` values, which is the `batch_size`.

In [19]:
```
batch[0].shape
```

Out[19]:
```
torch.Size([32, 784])
```

In [20]:
```
batch[1].shape
```

Out[20]:
```
torch.Size([32])
```

2.5 Build the Model

We've created our DataLoaders, now it's time to build our models.

Exercise

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

- Has a flatten layer.
- Has a dense input layer. This layer should contain 512 neurons amd use the `relu` activation function
- 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

We will define a few variables to get started:

In [21]:
```
input_size = 28 * 28
n_classes = 24
```

Do your work in the cell below, creating a `model` variable to store the model. We've imported the Sequential model class and Linear layer class to get you started. Reveal the solution below for a hint:

In [22]:
```
model = nn.Sequential(

)
```

Solution

In [23]:
```
# SOLUTION
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(input_size, 512),  # Input
    nn.ReLU(),  # Activation for input
    nn.Linear(512, 512),  # Hidden
    nn.ReLU(),  # Activation for hidden
    nn.Linear(512, n_classes)  # Output
)
```

This time, we'll combine compiling the model and sending it to the GPU in one step:

In [24]:
```
model = torch.compile(model.to(device))
model
```

Out[24]:
```
OptimizedModule(
  (_orig_mod): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=512, bias=True)
    (2): ReLU()
    (3): Linear(in_features=512, out_features=512, bias=True)
    (4): ReLU()
    (5): Linear(in_features=512, out_features=24, bias=True)
  )
)
```

Since categorizing these ASL images is similar to categorizing MNIST's handwritten digits, we will use the same `loss_function` (Categorical CrossEntropy) and `optimizer` (Adam).

In [25]:
```
loss_function = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())
```

2.6 Training the Model

This time, let's look at our `train` and `validate` functions in more detail.

2.6.1 The Train Function

This code is almost the same as in the previous notebook, but we no longer send `x` and `y` to our GPU because our DataLoader already does that.

Before looping through the DataLoader, we will set the model to model.train to make sure its parameters can be updated. To make it easier for us to follow training progress, we'll keep track of the total `loss` and `accuracy`.

Then, for each batch in our `train_loader`, we will:

1. Get an `output` prediction from the model
2. Set the gradient to zero with the `optimizer`'s zero_grad function

3. Calculate the loss with our `loss_function`
4. Compute the gradient with backward
5. Update our model parameters with the `optimizer`'s step function.
6. Update the `loss` and `accuracy` totals

In [26]:
```
def train():
    loss = 0
    accuracy = 0

    model.train()
    for x, y in train_loader:
        output = model(x)
        optimizer.zero_grad()
        batch_loss = loss_function(output, y)
        batch_loss.backward()
        optimizer.step()

        loss += batch_loss.item()
        accuracy += get_batch_accuracy(output, y, train_N)
    print('Train - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

2.6.2 The Validate Function

The model does not learn during validation, so the `validate` function is simpler than the `train` function above.

One key difference is we will set the model to evaluation mode with model.evaluate, which will prevent the model from updating any parameters.

In [27]:
```
def validate():
    loss = 0
    accuracy = 0

    model.eval()
    with torch.no_grad():
        for x, y in valid_loader:
            output = model(x)

            loss += loss_function(output, y).item()
            accuracy += get_batch_accuracy(output, y, valid_N)
    print('Valid - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

2.6.3 Calculating the Accuracy

Both the `train` and `validate` functions use `get_batch_accuracy`, but we have not defined that in this notebook yet.

Exercise

The function below has three `FIXME` s. Each one corresponds to the functions input arguments. Can you replace each FIXME with the correct argument?

It may help to view the documentation for argmax, eq, and view_as.

In [28]:
```
def get_batch_accuracy(output, y, N):
    pred = FIXME.argmax(dim=1, keepdim=True)
    correct = pred.eq(FIXME.view_as(pred)).sum().item()
    return correct / FIXME
```

Solution

Click the `...` below for the solution.

In [29]:
```
# SOLUTION
def get_batch_accuracy(output, y, N):
    pred = output.argmax(dim=1, keepdim=True)
    correct = pred.eq(y.view_as(pred)).sum().item()
    return correct / N
```

2.6.3 The Training Loop

Let's bring it all together! Run the cell below to train the data for 20 `epochs`.

In [30]:
```
epochs = 20

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    train()
    validate()
```

```
Epoch: 0
Train - Loss: 1581.7891 Accuracy: 0.4029
Valid - Loss: 321.8966 Accuracy: 0.5294
Epoch: 1
Train - Loss: 742.8489 Accuracy: 0.7066
Valid - Loss: 232.1240 Accuracy: 0.6468
Epoch: 2
Train - Loss: 390.4394 Accuracy: 0.8482
Valid - Loss: 216.7112 Accuracy: 0.7132
Epoch: 3
Train - Loss: 206.8601 Accuracy: 0.9261
Valid - Loss: 196.0227 Accuracy: 0.7550
Epoch: 4
Train - Loss: 145.2274 Accuracy: 0.9476
Valid - Loss: 202.4981 Accuracy: 0.7649
Epoch: 5
Train - Loss: 79.3949 Accuracy: 0.9736
Valid - Loss: 228.8129 Accuracy: 0.7607
Epoch: 6
Train - Loss: 72.5534 Accuracy: 0.9746
Valid - Loss: 281.4385 Accuracy: 0.7288
Epoch: 7
Train - Loss: 55.8076 Accuracy: 0.9797
Valid - Loss: 215.6231 Accuracy: 0.7953
Epoch: 8
Train - Loss: 62.5188 Accuracy: 0.9782
Valid - Loss: 214.9981 Accuracy: 0.8084
Epoch: 9
Train - Loss: 34.8716 Accuracy: 0.9875
Valid - Loss: 232.7552 Accuracy: 0.7851
Epoch: 10
Train - Loss: 13.2755 Accuracy: 0.9959
Valid - Loss: 222.5701 Accuracy: 0.8193
Epoch: 11
Train - Loss: 75.2331 Accuracy: 0.9753
Valid - Loss: 208.8311 Accuracy: 0.8293
Epoch: 12
Train - Loss: 4.2657 Accuracy: 0.9998
Valid - Loss: 227.1098 Accuracy: 0.8224
Epoch: 13
Train - Loss: 67.4808 Accuracy: 0.9762
Valid - Loss: 212.2502 Accuracy: 0.8059
Epoch: 14
Train - Loss: 2.8549 Accuracy: 0.9998
Valid - Loss: 220.2399 Accuracy: 0.8309
Epoch: 15
Train - Loss: 54.7750 Accuracy: 0.9827
Valid - Loss: 213.9484 Accuracy: 0.8048
Epoch: 16
Train - Loss: 6.8517 Accuracy: 0.9984
Valid - Loss: 218.9971 Accuracy: 0.8254
Epoch: 17
Train - Loss: 1.1296 Accuracy: 1.0000
Valid - Loss: 233.6943 Accuracy: 0.8185
Epoch: 18
Train - Loss: 0.7331 Accuracy: 1.0000
Valid - Loss: 241.9852 Accuracy: 0.8243
Epoch: 19
Train - Loss: 91.5331 Accuracy: 0.9734
Valid - Loss: 224.7301 Accuracy: 0.7982
```

2.6.4 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.

`# SOLUTION` 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.

2.7 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 (including labels), preparing it, creating a model, and then training the model with prepared data.

2.7.1 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 [31]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[31]:
```
{'status': 'ok', 'restart': True}
```

2.7.2 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*.

Header