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 # a standard PyTorch import that lets you conveniently access neural network building blocks
import pandas as pd
import torch
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
#Dataset: Represents your dataset (images, signals, text, etc.).
#DataLoader: Takes a Dataset and manages batching, shuffling, and parallel loading.
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	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel78
0	3	107	118	127	134	139	143	146	150	153	...	207	207	207	207	206	2
1	6	155	157	156	156	156	157	156	158	158	...	69	149	128	87	94	1
2	2	187	188	188	187	187	186	187	188	187	...	202	201	200	199	198	1
3	2	211	211	212	212	211	210	211	210	210	...	235	234	233	231	230	2
4	12	164	167	170	172	176	179	180	184	185	...	92	105	105	108	133	1

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

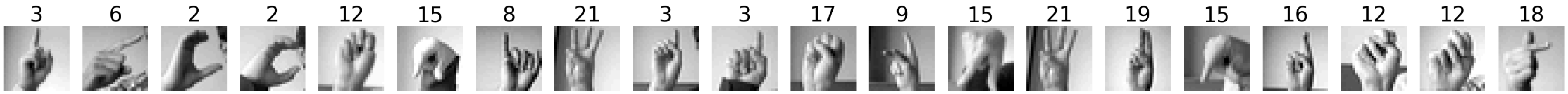
To visualize the images, we will again use the matplotlib library. We don't need to worry about the details of this visualization, but if interested, you can learn more about matplotlib at a later time.

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 [15]:

```
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 [16]:

```
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 [17]:

```
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 [18]:

```
train_loader
```

Out[18]:

```
<torch.utils.data.dataloader.DataLoader at 0x7f665f832ef0>
```

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

In [23]:

```
batch = next(iter(train_loader))
batch
```

```
Out[23]: [tensor([[0.7451, 0.7608, 0.7765,  ..., 0.4431, 0.4549, 0.4510],
                  [0.6314, 0.6353, 0.6353,  ..., 0.0000, 0.0000, 0.0000],
                  [0.6510, 0.6588, 0.6667,  ..., 0.7333, 0.7294, 0.7216],
                  ...,
                  [0.4196, 0.4549, 0.4863,  ..., 0.3843, 0.3882, 0.3882],
                  [0.4078, 0.4235, 0.4431,  ..., 0.7922, 0.7961, 0.7961],
                  [0.4000, 0.4118, 0.4196,  ..., 0.6000, 0.6000, 0.5922]],
                 device='cuda:0'),
          tensor([ 4, 21, 13, 22, 17, 13, 17, 16, 15,  8, 13, 14,  3,  9,  0,  8,  9, 18,
                  10, 12, 16, 10,  0,  5, 23, 14, 14, 22, 23, 15, 16, 20],
                 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 [21]: batch[0].shape
```

```
Out[21]: torch.Size([32, 784])
```

```
In [22]: batch[1].shape
```

```
Out[22]: 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 [24]:
```
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 Sequental model class and Linear layer class to get you started. Reveal the solution below for a hint:

In [26]:
```
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
)
```

Solution

In [27]:
```
# 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 [28]:
```
model = torch.compile(model.to(device))
model
```

Out[28]:

```
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). `nn.CrossEntropyLoss` includes the softmax function, and is computationally faster when passing class indices as opposed to predicted probabilities.

In [29]:

```
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 [30]:

```
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 [31]:

```
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 [32]:

```
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
```

Solution

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

In [33]:

```
# 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 [34]:

```
epochs = 20
```

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

```
Epoch: 0
Train - Loss: 1585.0111 Accuracy: 0.3997
Valid - Loss: 328.6787 Accuracy: 0.5070
Epoch: 1
Train - Loss: 741.2053 Accuracy: 0.7083
Valid - Loss: 230.9441 Accuracy: 0.6573
Epoch: 2
Train - Loss: 395.8987 Accuracy: 0.8463
Valid - Loss: 224.4831 Accuracy: 0.6627
Epoch: 3
Train - Loss: 217.4490 Accuracy: 0.9216
Valid - Loss: 216.0749 Accuracy: 0.7327
Epoch: 4
Train - Loss: 133.3726 Accuracy: 0.9547
Valid - Loss: 236.7112 Accuracy: 0.7368
Epoch: 5
Train - Loss: 86.0935 Accuracy: 0.9724
Valid - Loss: 229.0477 Accuracy: 0.7584
Epoch: 6
Train - Loss: 69.1449 Accuracy: 0.9770
Valid - Loss: 230.1872 Accuracy: 0.7794
Epoch: 7
Train - Loss: 51.8858 Accuracy: 0.9822
Valid - Loss: 242.1255 Accuracy: 0.7805
Epoch: 8
Train - Loss: 64.6953 Accuracy: 0.9768
Valid - Loss: 238.8041 Accuracy: 0.7927
Epoch: 9
Train - Loss: 62.7457 Accuracy: 0.9778
Valid - Loss: 256.1916 Accuracy: 0.7731
Epoch: 10
Train - Loss: 6.2813 Accuracy: 0.9993
Valid - Loss: 240.3939 Accuracy: 0.8012
Epoch: 11
Train - Loss: 61.5650 Accuracy: 0.9788
Valid - Loss: 241.8659 Accuracy: 0.7950
Epoch: 12
Train - Loss: 37.3037 Accuracy: 0.9886
Valid - Loss: 238.9540 Accuracy: 0.7977
Epoch: 13
Train - Loss: 27.5844 Accuracy: 0.9903
```

```
Valid - Loss: 259.9572 Accuracy: 0.7854
Epoch: 14
Train - Loss: 1.3289 Accuracy: 1.0000
Valid - Loss: 258.9286 Accuracy: 0.7948
Epoch: 15
Train - Loss: 48.4883 Accuracy: 0.9850
Valid - Loss: 456.0155 Accuracy: 0.5938
Epoch: 16
Train - Loss: 40.6741 Accuracy: 0.9858
Valid - Loss: 247.5474 Accuracy: 0.8086
Epoch: 17
Train - Loss: 52.8402 Accuracy: 0.9818
Valid - Loss: 294.3217 Accuracy: 0.6926
Epoch: 18
Train - Loss: 14.2855 Accuracy: 0.9959
Valid - Loss: 236.0706 Accuracy: 0.8038
Epoch: 19
Train - Loss: 1.3985 Accuracy: 1.0000
Valid - Loss: 255.3629 Accuracy: 0.8105
```

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

```
Out[35]: {'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