Header

4a. Data Augmentation

So far, we've selected a model architecture that vastly improves the model's performance, as it is designed to recognize important features in the images. The validation accuracy is still lagging behind the training accuracy, which is a sign of overfitting: the model is getting confused by things it has not seen before when it tests against the validation dataset.

In order to teach our model to be more robust when looking at new data, we're going to programmatically increase the size and variance in our dataset. This is known as *data augmentation*, a useful technique for many deep learning applications.

The increase in size gives the model more images to learn from while training. The increase in variance helps the model ignore unimportant features and select only the features that are truly important in classification, allowing it to generalize better.

4a.1 Objectives

- Augment the ASL dataset
- Use the augmented data to train an improved model
- Save the well-trained model to disk for use in deployment

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
import torchvision.transforms.v2 as transforms
import torchvision.transforms.functional as F
import matplotlib.pyplot as plt

import utils

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

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

4a.2 Preparing the Data

As we're in a new notebook, we will load and process our data again. To do this, execute the following cell:

In [2]:
```
IMG_HEIGHT = 28
IMG_WIDTH = 28
```

```
IMG_CHS = 1
N_CLASSES = 24

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

class MyDataset(Dataset):
    def __init__(self, base_df):
        x_df = base_df.copy()
        y_df = x_df.pop('label')
        x_df = x_df.values / 255  # Normalize values from 0 to 1
        x_df = x_df.reshape(-1, IMG_CHS, IMG_WIDTH, IMG_HEIGHT)
        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)

n = 32
train_data = MyDataset(train_df)
train_loader = DataLoader(train_data, batch_size=n, shuffle=True)
train_N = len(train_loader.dataset)

valid_data = MyDataset(valid_df)
valid_loader = DataLoader(valid_data, batch_size=n)
valid_N = len(valid_loader.dataset)
```

4a.3 Model Creation

We will also need to create our model again. As we learned in the last lesson, convolutional neural networks use a repeated sequence of layers. Let's take advantage of this pattern to make our own custom module. We can then use this module like a layer in our Sequential model.

To do this, we will extend the Module class. Then we will define two methods:

- `__init__` : defines any properties we want our module to have, including our neural network layers. We will effectively be using a model within a model.
- `forward` : defines how we want the module to process any incoming data from the previous layer it is connected to. Since we are using a `Sequential` model, we can pass the input data into it like we are making a prediction.

In [3]:
```
class MyConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, dropout_p):
        kernel_size = 3
        super().__init__()

        self.model = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size, stride=1, padding=1),
            nn.BatchNorm2d(out_ch),
```

```
            nn.ReLU(),
            nn.Dropout(dropout_p),
            nn.MaxPool2d(2, stride=2)
        )

    def forward(self, x):
        return self.model(x)
```

Now that we've define our custom module, let's see it in action. The below model ia archecturially the same as in the previous lesson. Can you see the connection?

In [4]:
```
flattened_img_size = 75 * 3 * 3

# Input 1 x 28 x 28
base_model = nn.Sequential(
    MyConvBlock(IMG_CHS, 25, 0), # 25 x 14 x 14
    MyConvBlock(25, 50, 0.2), # 50 x 7 x 7
    MyConvBlock(50, 75, 0),  # 75 x 3 x 3
    # Flatten to Dense Layers
    nn.Flatten(),
    nn.Linear(flattened_img_size, 512),
    nn.Dropout(.3),
    nn.ReLU(),
    nn.Linear(512, N_CLASSES)
)
```

When we print the model, not only will it now show the use of our custom module, it will also show the layers within our custom module:

In [5]:
```
loss_function = nn.CrossEntropyLoss()
optimizer = Adam(base_model.parameters())

model = torch.compile(base_model.to(device))
model
```

Out[5]:
```
OptimizedModule(
  (_orig_mod): Sequential(
    (0): MyConvBlock(
      (model): Sequential(
        (0): Conv2d(1, 25, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
        (2): ReLU()
        (3): Dropout(p=0, inplace=False)
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
      )
    )
    (1): MyConvBlock(
      (model): Sequential(
        (0): Conv2d(25, 50, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
        (2): ReLU()
        (3): Dropout(p=0.2, inplace=False)
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
      )
    )
    (2): MyConvBlock(
      (model): Sequential(
        (0): Conv2d(50, 75, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(75, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
        (2): ReLU()
        (3): Dropout(p=0, inplace=False)
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
      )
    )
    (3): Flatten(start_dim=1, end_dim=-1)
    (4): Linear(in_features=675, out_features=512, bias=True)
    (5): Dropout(p=0.3, inplace=False)
    (6): ReLU()
    (7): Linear(in_features=512, out_features=24, bias=True)
  )
)
```

Custom modules are flexible, and we can define any other methods or properties we wish to have. This makes them powerful when data scientists are trying to solve complex problems.

4a.4 Data Augmentation

Before defining our training loop, it's time to set up our data augmentation.

We've seen TorchVision's Transforms before, but in this lesson, we will further explore its data augmentation tools. First, let's get a sample image to test with:

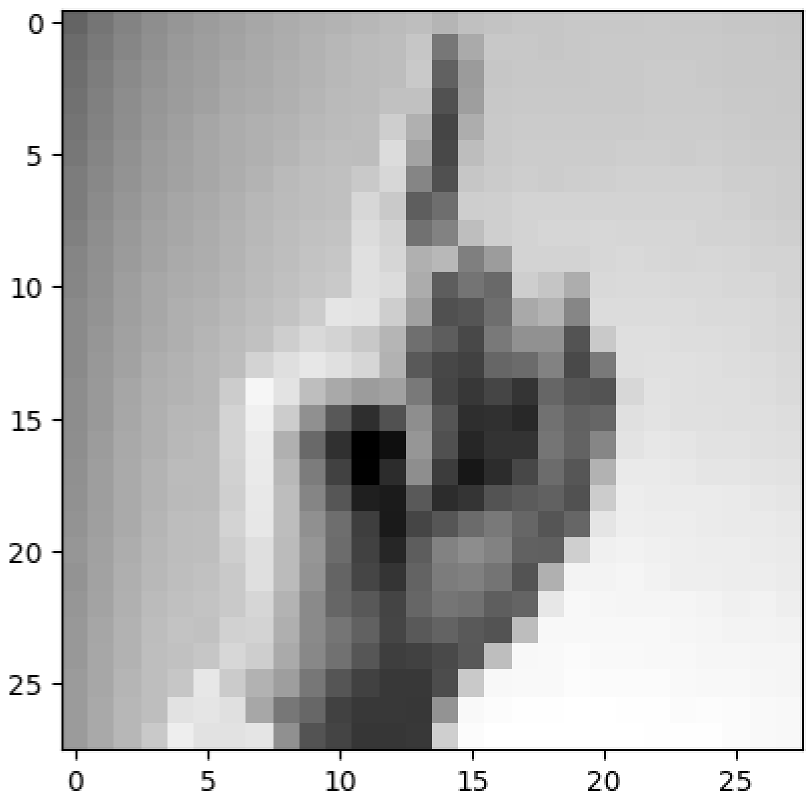
In [6]:
```python
row_0 = train_df.head(1)
y_0 = row_0.pop('label')
```

```
x_0 = row_0.values / 255
x_0 = x_0.reshape(IMG_CHS, IMG_WIDTH, IMG_HEIGHT)
x_0 = torch.tensor(x_0)
x_0.shape
```

Out[6]: `torch.Size([1, 28, 28])`

In [7]:
```
image = F.to_pil_image(x_0)
plt.imshow(image, cmap='gray')
```

Out[7]: `<matplotlib.image.AxesImage at 0x7f21e29ad420>`



4a.4.1 RandomResizeCrop

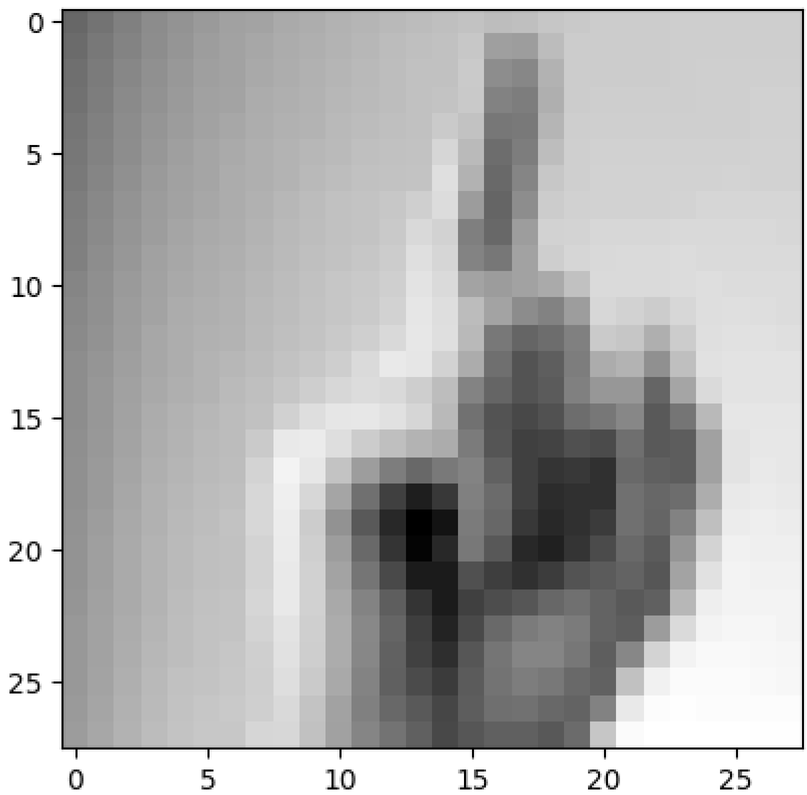
This transform will randomly resize the input image based on `scale`, and then crop) it to a size we specify. In this case, we will crop it to the original image dimensions. To do this, TorchVision needs to know the aspect ratio) of the image it is scaling. Since our height is the same as our width, our aspect `ratio` is 1:1.

In [8]:
```
trans = transforms.Compose([
    transforms.RandomResizedCrop((IMG_WIDTH, IMG_HEIGHT), scale=(.7, 1), ratio=(
])
```

Try running the below cell a few times. It should be different each time.

In [9]:
```
new_x_0 = trans(x_0)
image = F.to_pil_image(new_x_0)
plt.imshow(image, cmap='gray')
```

Out[9]: `<matplotlib.image.AxesImage at 0x7f21e1edc550>`



In [10]:
```
new_x_0.shape
```

Out[10]:
```
torch.Size([1, 28, 28])
```

4a.4.2 RandomHorizontalFlip

We can also randomly flip our images Horizontally or Vertically. However, for these images, we will only flip them horizontally.

Take a moment to think about why we would want to flip images horizontally, but not vertically. When you have an idea, reveal the text below.

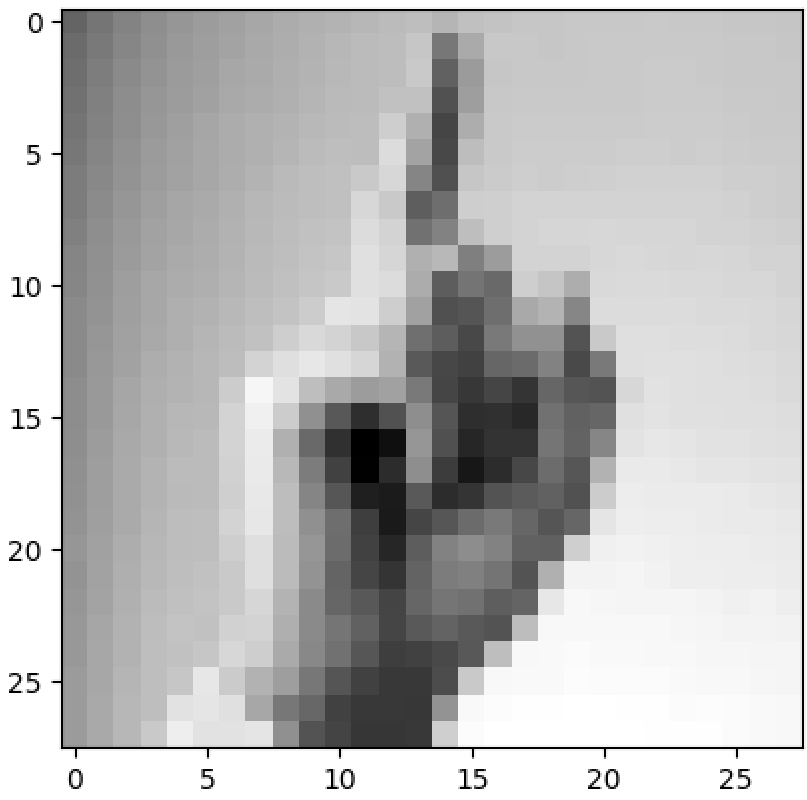
`# SOLUTION` Fun fact: American Sign Language can be done with either the left or right hand being dominant. However, it is unlikely to see sign language from upside down. This kind of domain-specific reasoning can help make good decisions for your own deep learning applications.

In [11]:
```
trans = transforms.Compose([
    transforms.RandomHorizontalFlip()
])
```

Try running the below cell a few times. Does the image flip about half the time?

In [12]:
```
new_x_0 = trans(x_0)
image = F.to_pil_image(new_x_0)
plt.imshow(image, cmap='gray')
```

Out[12]:
```
<matplotlib.image.AxesImage at 0x7f21e287c3a0>
```



4a.4.3 RandomRotation

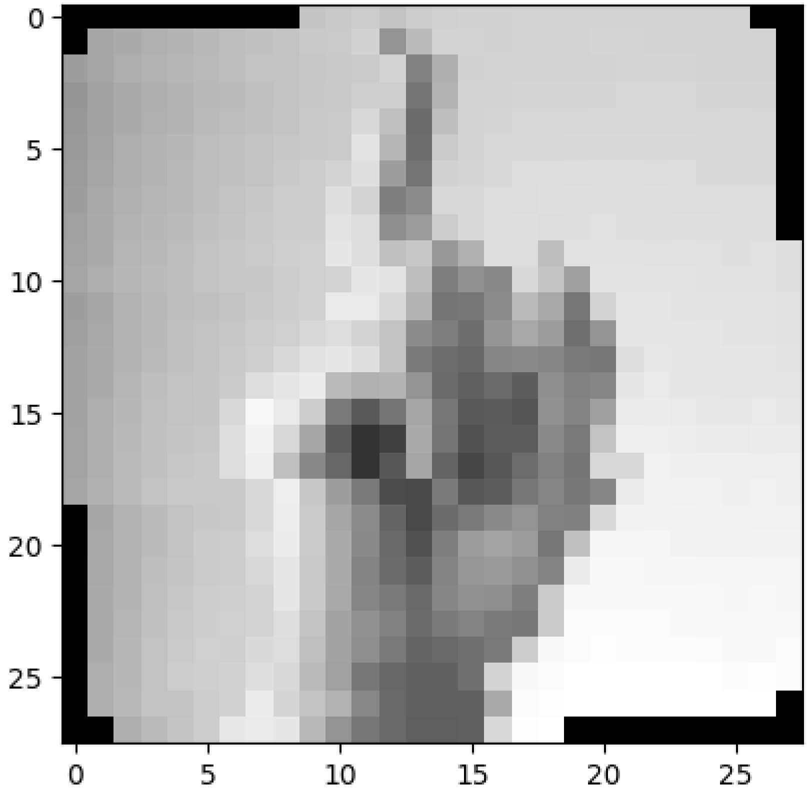
We can also randomly rotate the image to add more variability. Just like with with other augmentation techniques, it's easy to accidentally go too far. With ASL, if we rotate too much, our `D` s might look like `G` s and visa versa. Because of this, let's limit it to `30` degrees.

In [13]:
```
trans = transforms.Compose([
    transforms.RandomRotation(10)
])
```

When we run the cell block below, some black pixels may appear. The corners or our image disappear when we rotate, and for almost every pixel we lose, we gain an empty pixel.

In [14]:
```
new_x_0 = trans(x_0)
image = F.to_pil_image(new_x_0)
plt.imshow(image, cmap='gray')
```

Out[14]: `<matplotlib.image.AxesImage at 0x7f21e1fa7bb0>`



4a.4.3 ColorJitter

The `ColorJitter` transform has 4 arguments:

- brightness
- contrast)
- saturation
- hue

The latter 2 apply to color images, so we will only use the first 2 for now.

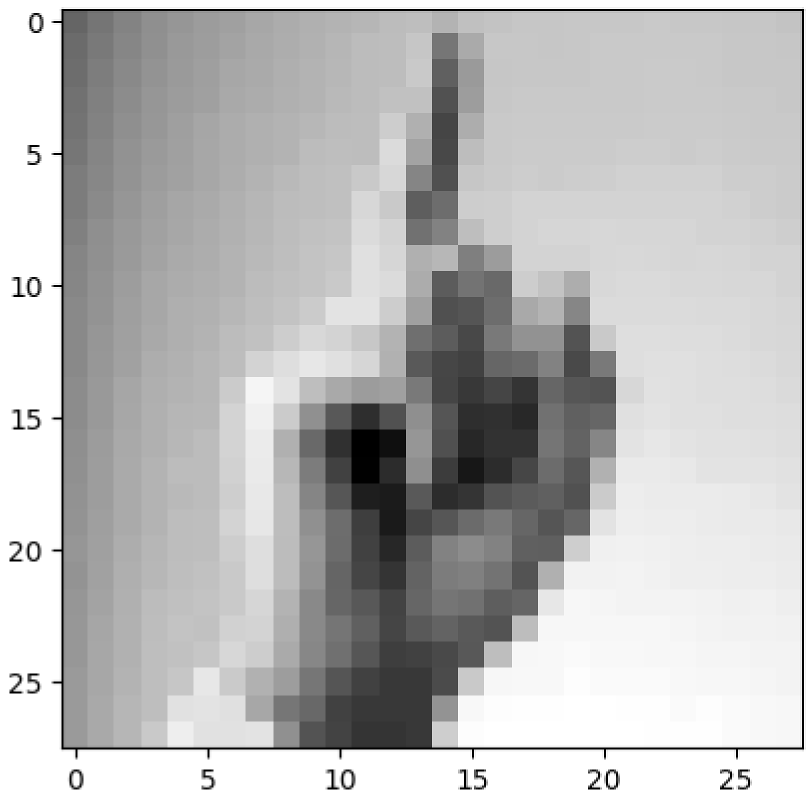
In [15]:
```
brightness = .2  # Change to be from 0 to 1
contrast = .5  # Change to be from 0 to 1

trans = transforms.Compose([
    transforms.ColorJitter(brightness=brightness, contrast=contrast)
])
```

Try running the below a few times, but also try changing either `brightness` or `contrast` to `1`. Get any intersting results?

In [16]:
```
new_x_0 = trans(x_0)
image = F.to_pil_image(new_x_0)
plt.imshow(image, cmap='gray')
```

Out[16]: `<matplotlib.image.AxesImage at 0x7f21e1e35030>`



4a.3.4 Compose

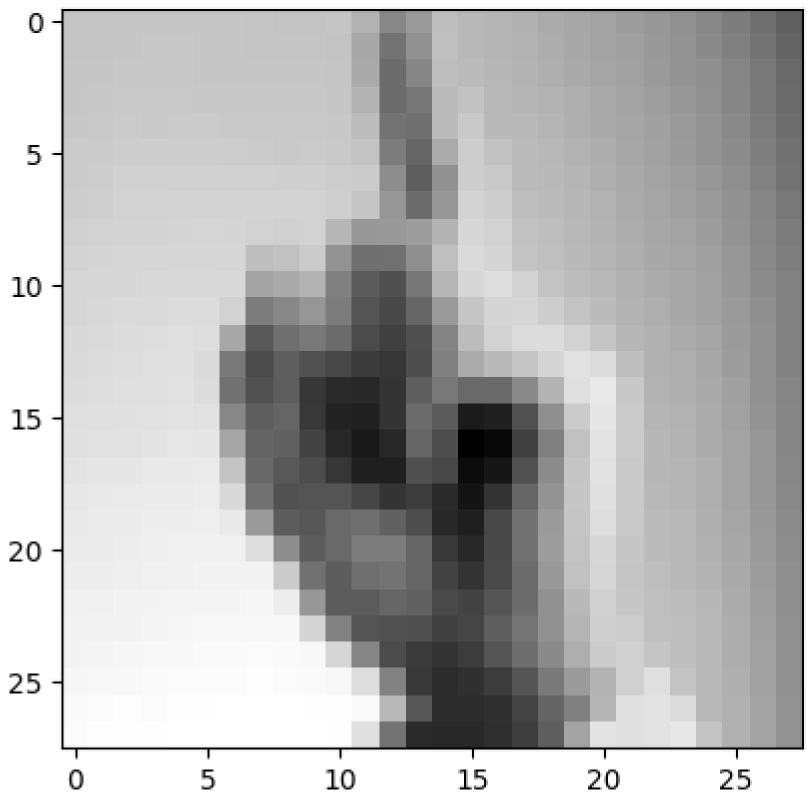
Time to bring it all together. We can create a sequence of these random transformations with `Compose`.

In [17]:
```
random_transforms = transforms.Compose([
    transforms.RandomRotation(5),
    transforms.RandomResizedCrop((IMG_WIDTH, IMG_HEIGHT), scale=(.9, 1), ratio=(
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=.2, contrast=.5)
])
```

Let's test it out. With all the different combinations how many varations are there of this one image? Infinite?

In [18]:
```
new_x_0 = random_transforms(x_0)
image = F.to_pil_image(new_x_0)
plt.imshow(image, cmap='gray')
```

Out[18]:
```
<matplotlib.image.AxesImage at 0x7f21e1e9a410>
```



4a.4 Training with Augmentation

Our training is mostly the same, but there is one line of change. Before passing our images to our model, we will apply our `random_transforms`. For conveneince, we moved `get_batch_accuracy` to a utils file.

In [19]:
```
def train():
    loss = 0
    accuracy = 0

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

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

On the other hamd, validation remains the same. There are no random transformations.

In [20]:
```
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 += utils.get_batch_accuracy(output, y, valid_N)
    print('Valid - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

Let's put data augmentation to the test.

In [21]:
```
epochs = 20

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

```
Epoch: 0
Train - Loss: 613.5756 Accuracy: 0.7663
Valid - Loss: 73.3947 Accuracy: 0.8822
Epoch: 1
Train - Loss: 105.6962 Accuracy: 0.9602
Valid - Loss: 17.5984 Accuracy: 0.9725
Epoch: 2
Train - Loss: 50.5980 Accuracy: 0.9803
Valid - Loss: 18.5039 Accuracy: 0.9697
Epoch: 3
Train - Loss: 41.6222 Accuracy: 0.9844
Valid - Loss: 21.3439 Accuracy: 0.9735
Epoch: 4
Train - Loss: 32.9101 Accuracy: 0.9878
Valid - Loss: 26.9878 Accuracy: 0.9639
Epoch: 5
Train - Loss: 30.2374 Accuracy: 0.9884
Valid - Loss: 22.8965 Accuracy: 0.9690
Epoch: 6
Train - Loss: 23.6386 Accuracy: 0.9908
Valid - Loss: 19.9227 Accuracy: 0.9734
Epoch: 7
Train - Loss: 22.9253 Accuracy: 0.9913
Valid - Loss: 25.6728 Accuracy: 0.9741
Epoch: 8
Train - Loss: 20.0298 Accuracy: 0.9925
Valid - Loss: 15.8704 Accuracy: 0.9838
Epoch: 9
Train - Loss: 20.2951 Accuracy: 0.9926
Valid - Loss: 6.8531 Accuracy: 0.9887
Epoch: 10
Train - Loss: 20.7779 Accuracy: 0.9927
Valid - Loss: 10.4751 Accuracy: 0.9868
Epoch: 11
Train - Loss: 15.1522 Accuracy: 0.9941
Valid - Loss: 14.0117 Accuracy: 0.9771
Epoch: 12
Train - Loss: 12.4576 Accuracy: 0.9954
Valid - Loss: 36.5509 Accuracy: 0.9516
Epoch: 13
Train - Loss: 16.5814 Accuracy: 0.9939
Valid - Loss: 11.5821 Accuracy: 0.9833
Epoch: 14
Train - Loss: 10.5703 Accuracy: 0.9961
Valid - Loss: 7.7762 Accuracy: 0.9858
Epoch: 15
Train - Loss: 12.7173 Accuracy: 0.9948
Valid - Loss: 13.9180 Accuracy: 0.9855
Epoch: 16
Train - Loss: 18.8320 Accuracy: 0.9939
Valid - Loss: 10.5035 Accuracy: 0.9911
Epoch: 17
Train - Loss: 13.1409 Accuracy: 0.9952
Valid - Loss: 15.0482 Accuracy: 0.9782
Epoch: 18
Train - Loss: 8.8914 Accuracy: 0.9969
Valid - Loss: 8.2252 Accuracy: 0.9894
Epoch: 19
Train - Loss: 14.9841 Accuracy: 0.9946
Valid - Loss: 20.0923 Accuracy: 0.9824
```

Discussion of Results

You will notice that the validation accuracy is higher, and more consistent. This means that our model is no longer overfitting in the way it was; it generalizes better, making better predictions on new data.

The training accuracy may be lower, and that's ok. Compared to before, the model is being exposed to a much larger variety of data.

Saving the Model

Now that we have a well-trained model, we will want to deploy it to perform inference on new images.

It is common, once we have a trained model that we are happy with to save it to disk. PyTorch has multiple ways to do this, but for now, we will use `torch.save`. We will also need to save the code for our `MyConvBlock` custom module, which we did in utils.py. In the next notebook, we'll load the model and use it to read new sign language pictures.

PyTorch cannot save a compiled model (see this post), so we will instead

In [22]:
```
torch.save(base_model, 'model.pth')
```

Summary

In this section, you used TorchVision to augment a dataset. This resulted in a trained model with less overfitting and excellent validation image results.

Clear the Memory

Before moving on, please execute the following cell to clear up the GPU memory.

In [23]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[23]:
```
{'status': 'ok', 'restart': True}
```

Next

Now that you have a well-trained model saved to disk, you will, in the next section, deploy it to make predictions on not-yet-seen images.

Please continue to the next notebook: *Model Predictions*.

Header