Header

4b. Deploying Your Model

Now that we have a well trained model, it's time to use it. In this exercise, we'll expose new images to our model and detect the correct letters of the sign language alphabet. Let's get started!

4b.1 Objectives

- Load an already-trained model from disk
- Reformat images for a model trained on images of a different format
- Perform inference with new images, never seen by the trained model and evaluate its performance

In [1]:
```
import pandas as pd
import torch
import torch.nn as nn
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
import torchvision.io as tv_io
import torchvision.transforms.v2 as transforms
import torchvision.transforms.functional as F
import matplotlib.pyplot as plt

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

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

4b.2 Loading the Model

Now that we're in a new notebook, let's load the saved model that we trained. Our save from the previous exercise created a folder called "asl_model". We can load the model by selecting the same folder.

Since our model uses a custom module, we will need to load the code for that class. We have saved a copy of the code in utils.py.

In [2]:
```
from utils import MyConvBlock
```

Now that we have a definition for `MyConvBlock`, we can use torch.load to load a model from a path. We can use `map_location to specify the device. When we print the model, does it look the same as in the last notebook?

In [3]:
```
model = torch.load('model.pth', map_location=device)
```

```
model
```

Out[3]:

```
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_running_
stats=True)
      (2): ReLU()
      (3): Dropout(p=0, inplace=False)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
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_running_
stats=True)
      (2): ReLU()
      (3): Dropout(p=0.2, inplace=False)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
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_running_
stats=True)
      (2): ReLU()
      (3): Dropout(p=0, inplace=False)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
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)
)
```

We can also verify if the model is on our GPU.

In [4]:

```
next(model.parameters()).device
```

Out[4]:

```
device(type='cuda', index=0)
```

4b.3 Preparing an Image for the Model

It's now time to use the model to make predictions on new images that it's never seen before. This is also called inference. We have a set of images in the `data/asl_images` folder. Try opening it using the left navigation and explore the images.

You'll notice that the images we have are much higher resolution than the images in our dataset. They are also in color. Remember that our images in the dataset were 28x28 pixels and grayscale. It's important to keep in mind that whenever we make predictions with a model, the input must match the shape of the data that the model was trained on. For this model, the training dataset was of the shape: (27455, 28, 28, 1). This corresponded to 27455 images of 28 by 28 pixels each with one color channel (grayscale).

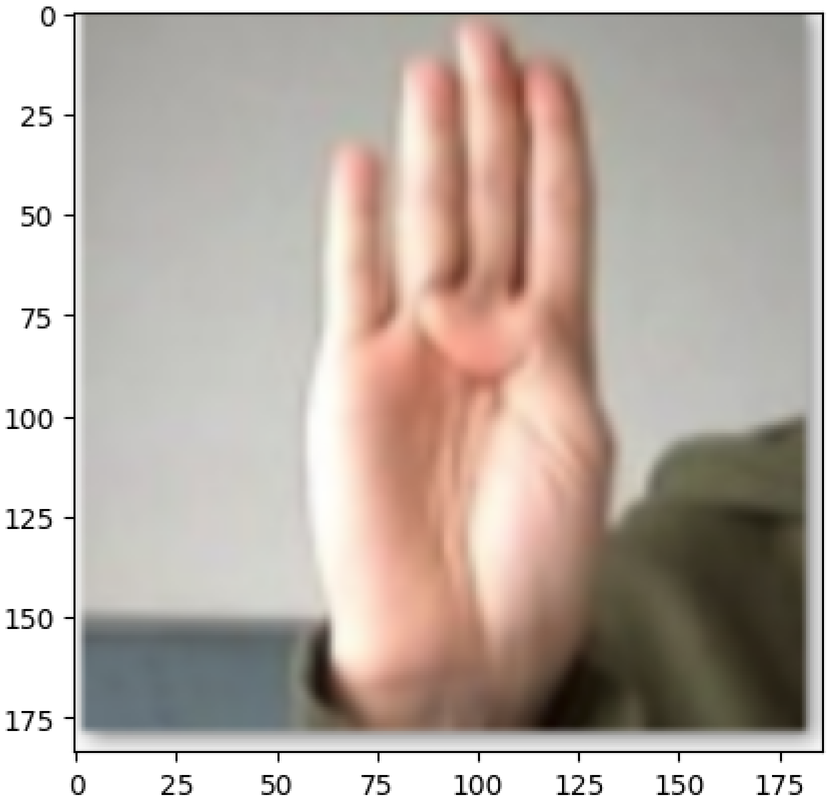
4b.3.1 Showing the Images

When we use our model to make predictions on new images, it will be useful to show the image as well. We can use the matplotlib library to do this.

In [5]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

def show_image(image_path):
    image = mpimg.imread(image_path)
    plt.imshow(image, cmap='gray')
```

In [6]:
```
show_image('data/asl_images/b.png')
```



4b.3.2 Scaling the Images

The images in our dataset were 28x28 pixels and grayscale. We need to make sure to pass the same size and grayscale images into our method for prediction. There are a few ways to edit images with Python, but TorchVision also has the read_image function. We can let it know what kind of image to read with ImageReadMode.

In [7]:
```
image = tv_io.read_image('data/asl_images/b.png', tv_io.ImageReadMode.GRAY)
```

```
image
```

Out[7]:
```
tensor([[[247, 223, 178,  ..., 223, 231, 238],
         [245, 221, 178,  ..., 213, 223, 232],
         [244, 220, 177,  ..., 204, 217, 228],
         ...,
         [247, 243, 238,  ..., 223, 231, 238],
         [249, 246, 243,  ..., 232, 238, 243],
         [250, 248, 246,  ..., 240, 243, 246]]], dtype=torch.uint8)
```

Let's look at the shape of the image.

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

Out[8]:
```
torch.Size([1, 184, 186])
```

This image is much larger than what we trained on. We can use TorchVision's Transforms again to get the data in the form our model expects.

We will:

- Convert the image to float with ToDtype
  - We will set `scale` to `True` in order to convert from [0, 255] to [0, 1]
- Resize the image to be 28 x 28 pixels
- Convert the images to Grayscale
  - This step doesn't do anything since our models are already grayscale, but we've added it here to show an alternative way to get grayscale images.

In [9]:
```
IMG_WIDTH = 28
IMG_HEIGHT = 28

preprocess_trans = transforms.Compose([
    transforms.ToDtype(torch.float32, scale=True), # Converts [0, 255] to [0, 1]
    transforms.Resize((IMG_WIDTH, IMG_HEIGHT)),
    transforms.Grayscale()  # From Color to Gray
])
```

Let's test `preprocess_trans` on an image to make sure it works correctly:

In [10]:
```
processed_image = preprocess_trans(image)
processed_image
```

```
Out[10]: tensor([[[0.6960, 0.6279, 0.6348, 0.6493, 0.6584, 0.6599, 0.6638, 0.6630,
                   0.6652, 0.6677, 0.6711, 0.6696, 0.6661, 0.6677, 0.7103, 0.6559,
                   0.6369, 0.6507, 0.6464, 0.6379, 0.6293, 0.6216, 0.6139, 0.6074,
                   0.5984, 0.5895, 0.5836, 0.6977],
                  [0.7013, 0.6388, 0.6480, 0.6578, 0.6647, 0.6687, 0.6709, 0.6734,
                   0.6746, 0.6794, 0.6814, 0.6784, 0.6902, 0.6801, 0.7648, 0.6606,
                   0.5610, 0.6348, 0.6418, 0.6509, 0.6417, 0.6308, 0.6252, 0.6197,
                   0.6095, 0.6007, 0.5923, 0.6773],
                  [0.7061, 0.6479, 0.6588, 0.6668, 0.6743, 0.6811, 0.6845, 0.6848,
                   0.6872, 0.6898, 0.6913, 0.6909, 0.7671, 0.6833, 0.6945, 0.6803,
                   0.4720, 0.6061, 0.5686, 0.6548, 0.6566, 0.6471, 0.6380, 0.6273,
                   0.6198, 0.6124, 0.6078, 0.6837],
                  [0.7165, 0.6603, 0.6699, 0.6770, 0.6821, 0.6880, 0.6959, 0.7001,
                   0.7027, 0.7014, 0.7009, 0.7041, 0.8257, 0.6885, 0.6387, 0.7202,
                   0.4938, 0.6614, 0.5266, 0.6351, 0.6649, 0.6595, 0.6516, 0.6407,
                   0.6298, 0.6250, 0.6173, 0.6882],
                  [0.7256, 0.6709, 0.6799, 0.6854, 0.6910, 0.6969, 0.7043, 0.7103,
                   0.7102, 0.7125, 0.7133, 0.7139, 0.8611, 0.7188, 0.6335, 0.7608,
                   0.5303, 0.6766, 0.5499, 0.6246, 0.6762, 0.6699, 0.6645, 0.6554,
                   0.6432, 0.6351, 0.6240, 0.6924],
                  [0.7330, 0.6805, 0.6897, 0.6953, 0.7027, 0.7094, 0.7154, 0.7206,
                   0.7205, 0.7473, 0.7414, 0.6798, 0.8693, 0.7476, 0.5998, 0.7269,
                   0.5397, 0.7189, 0.6083, 0.6100, 0.6883, 0.6822, 0.6731, 0.6665,
                   0.6588, 0.6521, 0.6393, 0.6989],
                  [0.7381, 0.6909, 0.7028, 0.7075, 0.7144, 0.7201, 0.7251, 0.7277,
                   0.7299, 0.7945, 0.7620, 0.6435, 0.8501, 0.7202, 0.5661, 0.6856,
                   0.5410, 0.7322, 0.6136, 0.5906, 0.7007, 0.6934, 0.6836, 0.6775,
                   0.6707, 0.6633, 0.6509, 0.7029],
                  [0.7454, 0.7010, 0.7137, 0.7207, 0.7251, 0.7304, 0.7372, 0.7406,
                   0.7431, 0.8278, 0.7804, 0.6528, 0.8622, 0.7322, 0.5645, 0.7076,
                   0.5461, 0.7335, 0.6120, 0.5866, 0.7123, 0.7053, 0.6987, 0.6888,
                   0.6808, 0.6742, 0.6646, 0.7099],
                  [0.7548, 0.7075, 0.7196, 0.7268, 0.7351, 0.7442, 0.7504, 0.7523,
                   0.7529, 0.8489, 0.8071, 0.6684, 0.8756, 0.7723, 0.5502, 0.6870,
                   0.5366, 0.7457, 0.6330, 0.6015, 0.7237, 0.7172, 0.7115, 0.7009,
                   0.6905, 0.6835, 0.6765, 0.7163],
                  [0.7651, 0.7217, 0.7307, 0.7373, 0.7441, 0.7520, 0.7572, 0.7596,
                   0.7575, 0.8611, 0.7932, 0.6405, 0.8541, 0.7321, 0.4840, 0.6115,
                   0.4894, 0.7340, 0.6253, 0.6107, 0.7308, 0.7246, 0.7194, 0.7067,
                   0.6986, 0.6930, 0.6829, 0.7200],
                  [0.7702, 0.7298, 0.7419, 0.7504, 0.7515, 0.7571, 0.7627, 0.7644,
                   0.7634, 0.8619, 0.8145, 0.6129, 0.7897, 0.6791, 0.5235, 0.5689,
                   0.5194, 0.6971, 0.5656, 0.6145, 0.7395, 0.7311, 0.7311, 0.7198,
                   0.7063, 0.6992, 0.6877, 0.7236],
                  [0.7731, 0.7333, 0.7501, 0.7598, 0.7619, 0.7667, 0.7735, 0.7755,
                   0.7703, 0.8683, 0.8244, 0.6060, 0.7601, 0.7788, 0.6607, 0.6570,
                   0.7361, 0.6433, 0.4766, 0.6062, 0.7470, 0.7415, 0.7397, 0.7300,
                   0.7173, 0.7084, 0.7005, 0.7292],
                  [0.7803, 0.7425, 0.7562, 0.7661, 0.7716, 0.7772, 0.7853, 0.7869,
                   0.7865, 0.8939, 0.8366, 0.7290, 0.7087, 0.6765, 0.7185, 0.7146,
                   0.7481, 0.6516, 0.4623, 0.5844, 0.7550, 0.7527, 0.7436, 0.7345,
                   0.7249, 0.7188, 0.7139, 0.7344],
                  [0.7857, 0.7519, 0.7614, 0.7720, 0.7795, 0.7852, 0.7930, 0.7927,
                   0.8062, 0.9326, 0.8685, 0.7303, 0.6154, 0.5670, 0.6339, 0.6400,
                   0.7295, 0.7404, 0.5492, 0.5713, 0.7595, 0.7567, 0.7497, 0.7437,
                   0.7353, 0.7283, 0.7236, 0.7384],
                  [0.7962, 0.7608, 0.7683, 0.7789, 0.7867, 0.7942, 0.7976, 0.7982,
                   0.8256, 0.9599, 0.8613, 0.6690, 0.5525, 0.5470, 0.5942, 0.6353,
                   0.8142, 0.8427, 0.6780, 0.5950, 0.7625, 0.7631, 0.7567, 0.7536,
                   0.7469, 0.7327, 0.7221, 0.7353],
                  [0.7973, 0.7658, 0.7763, 0.7849, 0.7925, 0.8019, 0.8045, 0.8049,
                   0.8447, 0.9687, 0.8495, 0.6453, 0.5374, 0.5607, 0.6472, 0.7540,
                   0.8591, 0.8658, 0.7467, 0.5874, 0.7299, 0.7715, 0.7537, 0.7251,
                   0.7073, 0.6471, 0.5496, 0.6368],
                  [0.7989, 0.7678, 0.7793, 0.7858, 0.7936, 0.7991, 0.8039, 0.8086,
                   0.8519, 0.9733, 0.8659, 0.6545, 0.5515, 0.5869, 0.6860, 0.8312,
                   0.8993, 0.8434, 0.7308, 0.5500, 0.6788, 0.7556, 0.5820, 0.4817,
                   0.4493, 0.3947, 0.3452, 0.5655],
                  [0.7989, 0.7647, 0.7767, 0.7855, 0.7954, 0.7989, 0.8028, 0.8095,
                   0.8439, 0.9725, 0.8849, 0.6788, 0.5622, 0.6192, 0.7184, 0.8763,
                   0.9149, 0.8298, 0.6990, 0.5179, 0.6845, 0.6413, 0.4226, 0.3916,
                   0.3717, 0.3501, 0.3414, 0.5592],
                  [0.8004, 0.7709, 0.7811, 0.7876, 0.7997, 0.8057, 0.8108, 0.8123,
                   0.8408, 0.9708, 0.9007, 0.7175, 0.5854, 0.6416, 0.7505, 0.9147,
                   0.9114, 0.8024, 0.6537, 0.5100, 0.6431, 0.4907, 0.3608, 0.3266,
                   0.3042, 0.2913, 0.3058, 0.5493],
                  [0.8028, 0.7784, 0.7892, 0.7968, 0.8067, 0.8115, 0.8168, 0.8176,
                   0.8394, 0.9602, 0.9093, 0.7519, 0.6211, 0.6364, 0.7657, 0.9326,
                   0.8834, 0.7556, 0.6154, 0.4715, 0.4537, 0.3496, 0.2947, 0.2483,
                   0.2108, 0.1780, 0.2274, 0.5359],
                  [0.8084, 0.7843, 0.7959, 0.8028, 0.8100, 0.8145, 0.8210, 0.8248,
                   0.8378, 0.9428, 0.9149, 0.7884, 0.6723, 0.6375, 0.7476, 0.9119,
                   0.8402, 0.7004, 0.5736, 0.4211, 0.3561, 0.3111, 0.2596, 0.2185,
                   0.2065, 0.1963, 0.2376, 0.5366],
                  [0.8119, 0.7835, 0.7957, 0.8038, 0.8107, 0.8182, 0.8257, 0.8293,
                   0.8348, 0.9140, 0.9279, 0.8250, 0.7242, 0.6641, 0.7005, 0.8559,
                   0.7827, 0.6306, 0.5190, 0.3836, 0.3422, 0.3344, 0.2802, 0.2049,
                   0.1894, 0.2016, 0.2194, 0.5103],
                  [0.7556, 0.7140, 0.7286, 0.7326, 0.7406, 0.7492, 0.7635, 0.7719,
                   0.7784, 0.8478, 0.9378, 0.8447, 0.7514, 0.6784, 0.6332, 0.7672,
                   0.7035, 0.5411, 0.4454, 0.3365, 0.3069, 0.3305, 0.3317, 0.2358,
                   0.1783, 0.1908, 0.2093, 0.4976],
                  [0.5617, 0.4596, 0.4660, 0.4625, 0.4623, 0.4683, 0.4773, 0.4887,
                   0.5028, 0.6108, 0.9405, 0.8575, 0.7469, 0.6421, 0.5607, 0.6235,
                   0.5803, 0.4467, 0.3607, 0.2649, 0.2650, 0.2896, 0.3480, 0.2902,
                   0.1924, 0.1703, 0.1971, 0.4916],
                  [0.5361, 0.4209, 0.4179, 0.4154, 0.4167, 0.4162, 0.4127, 0.4203,
                   0.3978, 0.4286, 0.8714, 0.8004, 0.6628, 0.5521, 0.4909, 0.5096,
                   0.4805, 0.3808, 0.2422, 0.1745, 0.2185, 0.2453, 0.3183, 0.3221,
                   0.2315, 0.1644, 0.1780, 0.4908],
                  [0.5491, 0.4269, 0.4296, 0.4264, 0.4323, 0.4326, 0.4330, 0.4380,
                   0.3963, 0.2552, 0.6048, 0.6484, 0.5450, 0.4699, 0.4480, 0.4281,
                   0.3992, 0.2762, 0.1146, 0.1069, 0.1878, 0.2146, 0.2748, 0.3332,
                   0.2809, 0.1870, 0.1570, 0.4802],
                  [0.5653, 0.4458, 0.4453, 0.4379, 0.4472, 0.4517, 0.4541, 0.4578,
                   0.4619, 0.3272, 0.4061, 0.5462, 0.5046, 0.4474, 0.4330, 0.3681,
                   0.3074, 0.1693, 0.0934, 0.1482, 0.2546, 0.2427, 0.2538, 0.3271,
                   0.3277, 0.2437, 0.1819, 0.4824],
                  [0.7822, 0.6822, 0.6749, 0.6725, 0.6757, 0.6784, 0.6790, 0.6801,
                   0.6824, 0.6579, 0.6527, 0.6838, 0.6824, 0.6756, 0.6680, 0.6401,
                   0.6141, 0.5686, 0.5584, 0.5850, 0.6220, 0.6086, 0.6050, 0.6293,
                   0.6379, 0.6134, 0.5933, 0.7271]]])
```

The numbers look correct, but how about the shape?

In [11]:

```
processed_image.shape
```

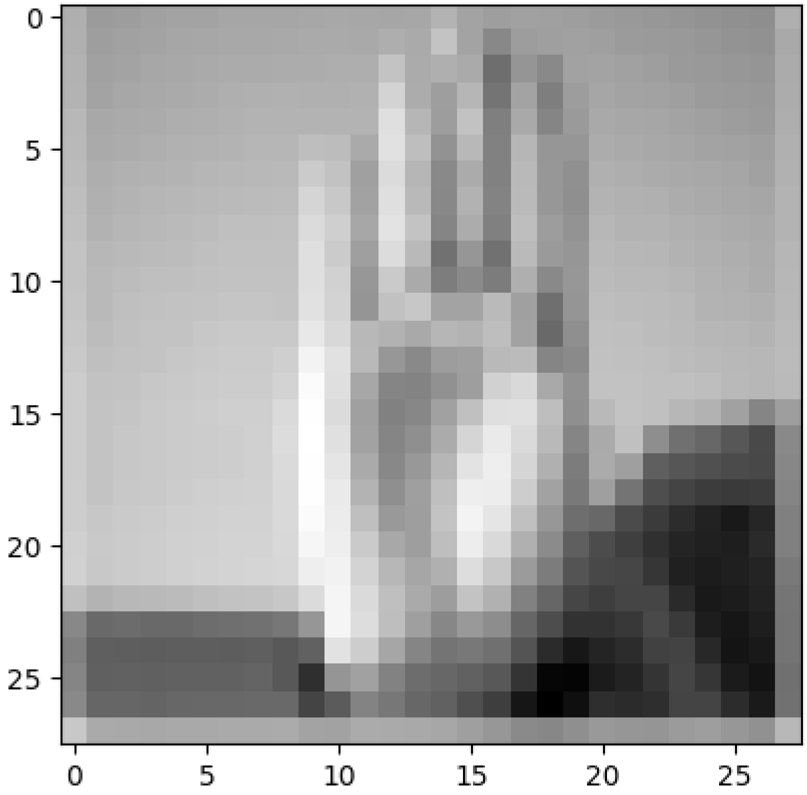
Out[11]:

```
torch.Size([1, 28, 28])
```

Next, let's plot the image to see if it looks like what we trained on.

In [12]:
```
plot_image = F.to_pil_image(processed_image)
plt.imshow(plot_image, cmap='gray')
```

Out[12]:
```
<matplotlib.image.AxesImage at 0x7fdc78729180>
```



Looking good! Let's pass it to our model.

4b.4 Making Predictions

Okay, now we're ready to predict! Our model still expects a batch of images. If the squeeze removes dimensions of 1, unsqueeze adds a dimension of 1 at the index we specify. The first dimension is usually the batch dimension, so we can say `.unsqueeze(0)`.

In [13]:
```
batched_image = processed_image.unsqueeze(0)
batched_image.shape
```

Out[13]:
```
torch.Size([1, 1, 28, 28])
```

Next, we should make sure the input tensor is on the same `device` as the model.

In [14]:
```
batched_image_gpu = batched_image.to(device)
batched_image_gpu.device
```

Out[14]:
```
device(type='cuda', index=0)
```

Now we're ready to pass it to the model!

In [15]:
```
output = model(batched_image_gpu)
output
```

Out[15]:
```
tensor([[-17.1014,  20.1510,  -7.4440, -25.4337,   0.5459,  -2.6385, -21.5080,
         -35.9991,  -6.1889, -12.9911, -18.2531, -16.9514, -14.6619, -26.2377,
         -11.8481, -17.1016, -34.8570, -30.2593, -21.5567,  -7.9934, -29.0655,
           5.0064, -10.0033, -28.8769]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

4b.4.1 Understanding the Prediction

The predictions are in the format of a 24 length array. The larger the value, the more likely the input image belongs to the corresponding class. Let's make it a little more readable. We can start by finding which element of the array represents the highest probability. This can be done easily with the numpy library and the argmax function.

In [16]:
```
prediction = output.argmax(dim=1).item()
prediction
```

Out[16]:
```
1
```

Each element of the prediction array represents a possible letter in the sign language alphabet. Remember that j and z are not options because they involve moving the hand, and we're only dealing with still photos. Let's create a mapping between the index of the predictions array, and the corresponding letter.

In [17]:
```
# Alphabet does not contain j or z because they require movement
alphabet = "abcdefghiklmnopqrstuvwxy"
```

We can now pass in our prediction index to find the corresponding letter.

In [18]:
```
alphabet[prediction]
```

Out[18]:
```
'b'
```

Exercise: Put it all Together

Let's put everything in a function so that we can make predictions just from the image file. Implement it in the function below using the functions and steps above. If you need help, you can reveal the solution by clicking the three dots below.

In [19]:
```
def predict_letter(file_path):
    # Show image
    FIXME
    # Load and grayscale image
    image = FIXME
    # Transform image
    image = FIXME
    # Batch image
    image = FIXME
    # Send image to correct device
    image = FIXME
    # Make prediction
    output = FIXME
    # Find max index
    prediction = FIXME
    # Convert prediction to letter
    predicted_letter = FIXME
    # Return prediction
    return predicted_letter
```

Solution

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

In [20]:

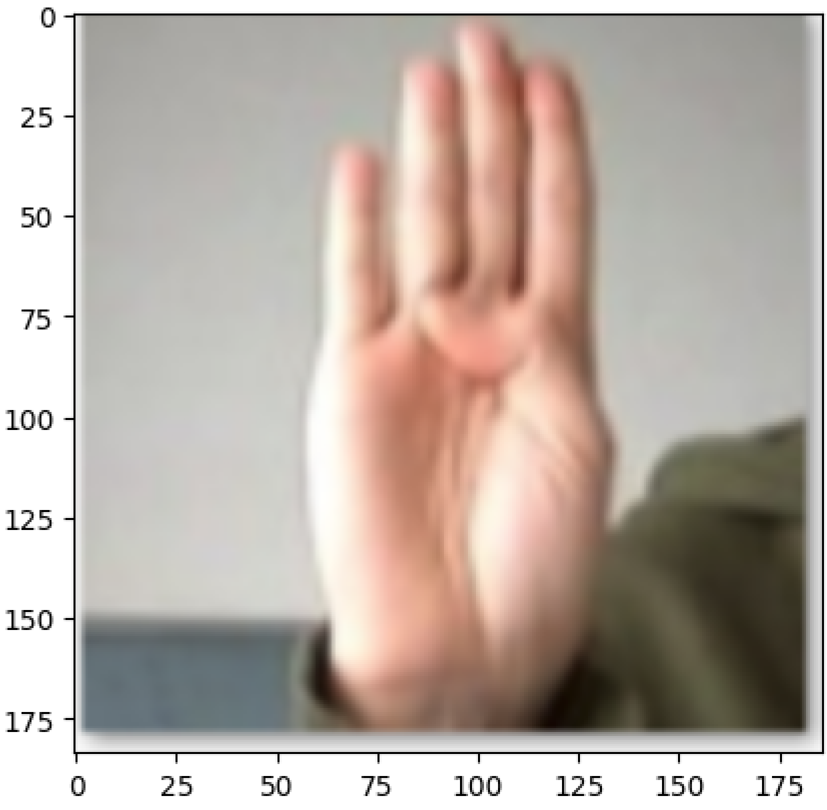
```
# SOLUTION
def predict_letter(file_path):
    show_image(file_path)
    image = tv_io.read_image(file_path, tv_io.ImageReadMode.GRAY)
    image = preprocess_trans(image)
    image = image.unsqueeze(0)
    image = image.to(device)
    output = model(image)
    prediction = output.argmax(dim=1).item()
    # convert prediction to letter
    predicted_letter = alphabet[prediction]
    return predicted_letter
```

In [21]:

```
predict_letter("data/asl_images/b.png")
```

Out[21]:

```
'b'
```

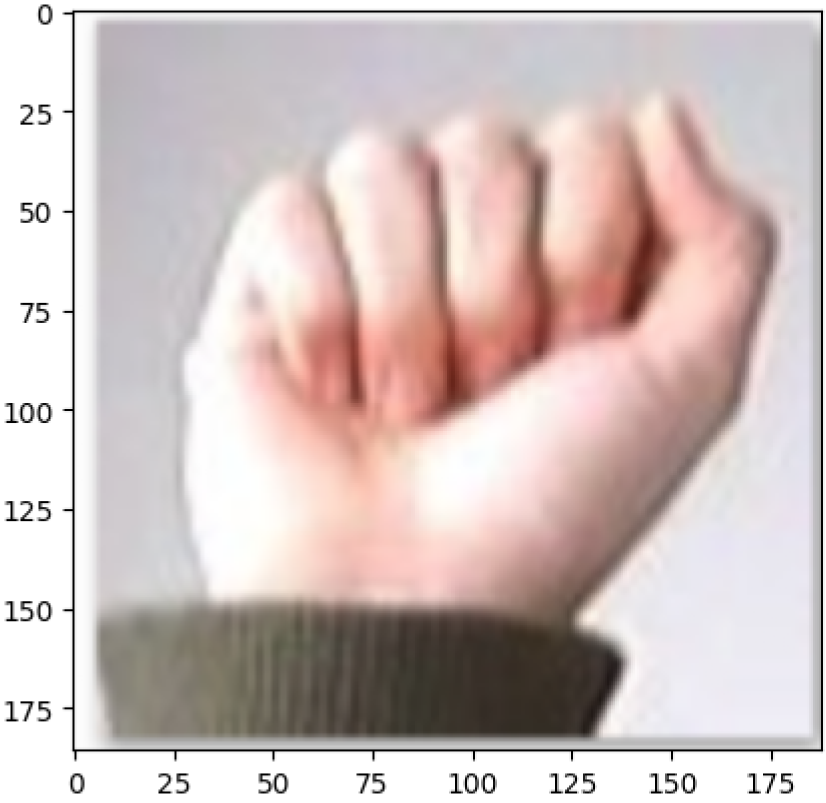


Let's also use the function with the 'a' letter in the asl_images datset:

In [22]:

```
predict_letter("data/asl_images/a.png")
```

```
Out[22]: 'a'
```



4b.5 Summary

Great work on these exercises! You've gone through the full process of training a highly accurate model from scratch, and then using the model to make new and valuable predictions. If you have some time, we encourage you to take pictures with your webcam, upload them by dropping them into the data/asl_images folder, and test out the model on them. For Mac you can use Photo Booth. For windows you can select the Camera app from your start screen. We hope you try it. It's a good opportunity to learn some sign language! For instance, try out the letters of your name.

We can imagine how this model could be used in an application to teach someone sign language, or even help someone who cannot speak interact with a computer.

4b.5.1 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}
```

4b.5.2 Next

We hope you've enjoyed these exercises. In the next sections we will learn how to take advantage of deep learning when we don't have a robust dataset available. See you there! To learn more about inference on the edge, check out this paper on the topic.

Now that we're familiar building your own models and have some understanding of how they work, we will turn our attention to the very powerful technique of using pre-trained models to expedite your work.

Header