# American Sign Language Recognition Project

By Jacob Flores & Jason Flores

First we imported all packages needed, then we installed kaggle in order to download the American Sign Language dataset. Before exploring the data we confirmed that the amount of classes are the correct (29). Also, we made sure to have latest version of PyTorch and Torchvision.

## 0 - Imports

```
import torch
from torch import nn
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import DataLoader, Subset, random_split
from torchvision import datasets, transforms, models
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
import numpy as np
import random
import os

!pip install -q kaggle

os.environ['KAGGLE_USERNAME'] = "jacobbb593"
os.environ['KAGGLE_KEY'] = "56bc160421849119bc1cfde80d32e83f"

!kaggle datasets download -d grassknoted/asl-alphabet

!unzip -q -o asl-alphabet.zip -d asl_alphabet_data


data_path = "/content/asl_alphabet_data/asl_alphabet_train/asl_alphabet_train"
print("Classes found:", os.listdir(data_path)[:5])
print("Total classes:", len(os.listdir(data_path)))

# Check Version
print(f"PyTorch version: {torch.__version__}\ntorchvision version: {torchvision.__version__}")
```

```
Dataset URL: https://www.kaggle.com/datasets/grassknoted/asl-alphabet
License(s): GPL-2.0
asl-alphabet.zip: Skipping, found more recently modified local copy (use --force to force download)
Classes found: ['R', 'V', 'H', 'T', 'Y']
Total classes: 29
PyTorch version: 2.6.0+cu124
torchvision version: 0.21.0+cu124
```

## 1 - Getting a Dataset

After having the dataset download we decided to modified it based on our needs. So we transformed it by reducing the size of images from 200x200(originally) to 64x64 so that we can process the images faster.

After having that modified, we decided to reduce our dataset (from 87,000 images to 5,000 images). This is because we are looking for efficiency in our project and to reduce the amount of time during the training.

As usual, we split into 80% train and 20% test so that we have a balance between the data.

After having that defined we create a Dataloader for training and test set. Then we print the dataset that we have defined.

After having the dataset that we are going to use, we explored the whole dataset by looking at lenghts, tensor values, classes names, classes index. In addition we checked input and output shapes of our data and then we visualize each class.

```
data_dir = "/content/asl_alphabet_data/asl_alphabet_train/asl_alphabet_train"
```

```
transform = transforms.Compose([
    transforms.Resize((64, 64)),
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

full_dataset = datasets.ImageFolder(root=data_dir,
                                    transform=transform)

```

```
# Use a smaller subset for faster training (5000 samples)
subset_indices = random.sample(range(len(full_dataset)), 5000)
subset = Subset(full_dataset, subset_indices)

# split into 80% train and 20% test
train_size = int(0.8 * len(subset))
test_size = len(subset) - train_size

# random split
train_data, test_data = random_split(subset, [train_size, test_size])

train_loader = DataLoader(train_data,
                          batch_size = 64,
                          shuffle = True)

train_dataset = DataLoader(test_data,
                           batch_size = 64,
                           shuffle = False)

# get class labels from the original dataset
classes = full_dataset.classes

print(f"Using {len(train_dataset)} samples from {len(classes)} classes")
```

```
Using 16 samples from 29 classes
```

```
len(train_data.dataset), len(test_data.dataset)
```

```
(5000, 5000)
```

```
len(train_data), len(test_data)
```

```
(4000, 1000)
```

```
image, label = train_data[0]
image, label
```

```
(tensor([[[-0.8275, -0.6314, -0.6314,  ..., -0.7569, -0.6784, -0.6706],
          [-0.6471, -0.2157, -0.1922,  ..., -0.2941, -0.0980, -0.2627],
          [-0.6941, -0.3098, -0.2941,  ..., -0.0196, -0.0275, -0.2784],
          ...,
          [-0.4275,  0.3961,  0.4039,  ...,  0.1451,  0.1765, -0.1137],
          [-0.4275,  0.3725,  0.4039,  ...,  0.1373,  0.1922, -0.1137],
          [-0.5686,  0.0353,  0.0510,  ..., -0.1451, -0.1294, -0.3569]],

         [[-0.8431, -0.6549, -0.6392,  ..., -0.7961, -0.7098, -0.7098],
          [-0.7020, -0.2863, -0.2235,  ..., -0.4039, -0.1608, -0.2941],
          [-0.7176, -0.3569, -0.3725,  ..., -0.1294, -0.0824, -0.3020],
          ...,
          [-0.3725,  0.5059,  0.5137,  ...,  0.0902,  0.1294, -0.1216],
          [-0.3647,  0.5137,  0.5059,  ...,  0.0745,  0.1373, -0.1137],
          [-0.5216,  0.1686,  0.1529,  ..., -0.1608, -0.1294, -0.3255]],

         [[ 0.6784,  0.3569,  0.3412,  ...,  0.2627,  0.3412,  0.5922],
          [ 0.3333, -0.4039, -0.4196,  ..., -0.4510, -0.1686,  0.1843],
          [ 0.3020, -0.4745, -0.5451,  ..., -0.1922, -0.0824,  0.1843],
          ...,
          [ 0.7490,  0.5686,  0.5059,  ...,  0.1059,  0.1373,  0.3961],
          [ 0.7412,  0.5608,  0.5137,  ...,  0.0745,  0.1137,  0.3725],
          [ 0.7647,  0.6078,  0.6000,  ...,  0.3098,  0.3255,  0.4902]]]),
 9)
```

```
class_names = full_dataset.classes
class_names
```

```
['A',
 'B',
 'C',
```

```
 'D',
 'E',
 'F',
 'G',
 'H',
 'I',
 'J',
 'K',
 'L',
 'M',
 'N',
 'O',
 'P',
 'Q',
 'R',
 'S',
 'T',
 'U',
 'V',
 'W',
 'X',
 'Y',
 'Z',
 'del',
 'nothing',
 'space']
```

```
class_to_idx = full_dataset.class_to_idx
class_to_idx
```

```
{'A': 0,
 'B': 1,
 'C': 2,
 'D': 3,
 'E': 4,
 'F': 5,
 'G': 6,
 'H': 7,
 'I': 8,
 'J': 9,
 'K': 10,
 'L': 11,
 'M': 12,
 'N': 13,
 'O': 14,
 'P': 15,
 'Q': 16,
 'R': 17,
 'S': 18,
 'T': 19,
 'U': 20,
 'V': 21,
 'W': 22,
 'X': 23,
 'Y': 24,
 'Z': 25,
 'del': 26,
 'nothing': 27,
 'space': 28}
```

## 1.1 - Check input and output shapes of data

```
# Check the shape of our image
print(f"Image shape: {image.shape} -> [color_channels, height, width]")
print(f"Image label: {class_names[label]}")
```

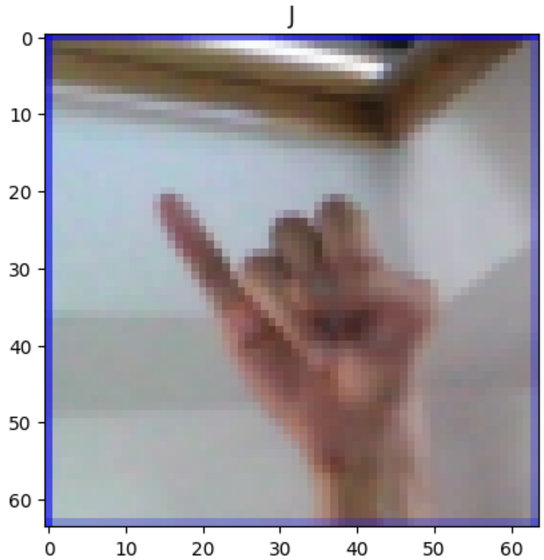
```
Image shape: torch.Size([3, 64, 64]) -> [color_channels, height, width]
Image label: J
```

## 1.2 - Visualizing our data

```
# Before we show a sample, notice how we change the image values, this is because
# we are performing reverse normalization in order to have pixel values in the range and avoid errors
image = image * 0.5 + 0.5 # You are going to see this more often along the way
print(f"Image shape: {image.shape}")
```

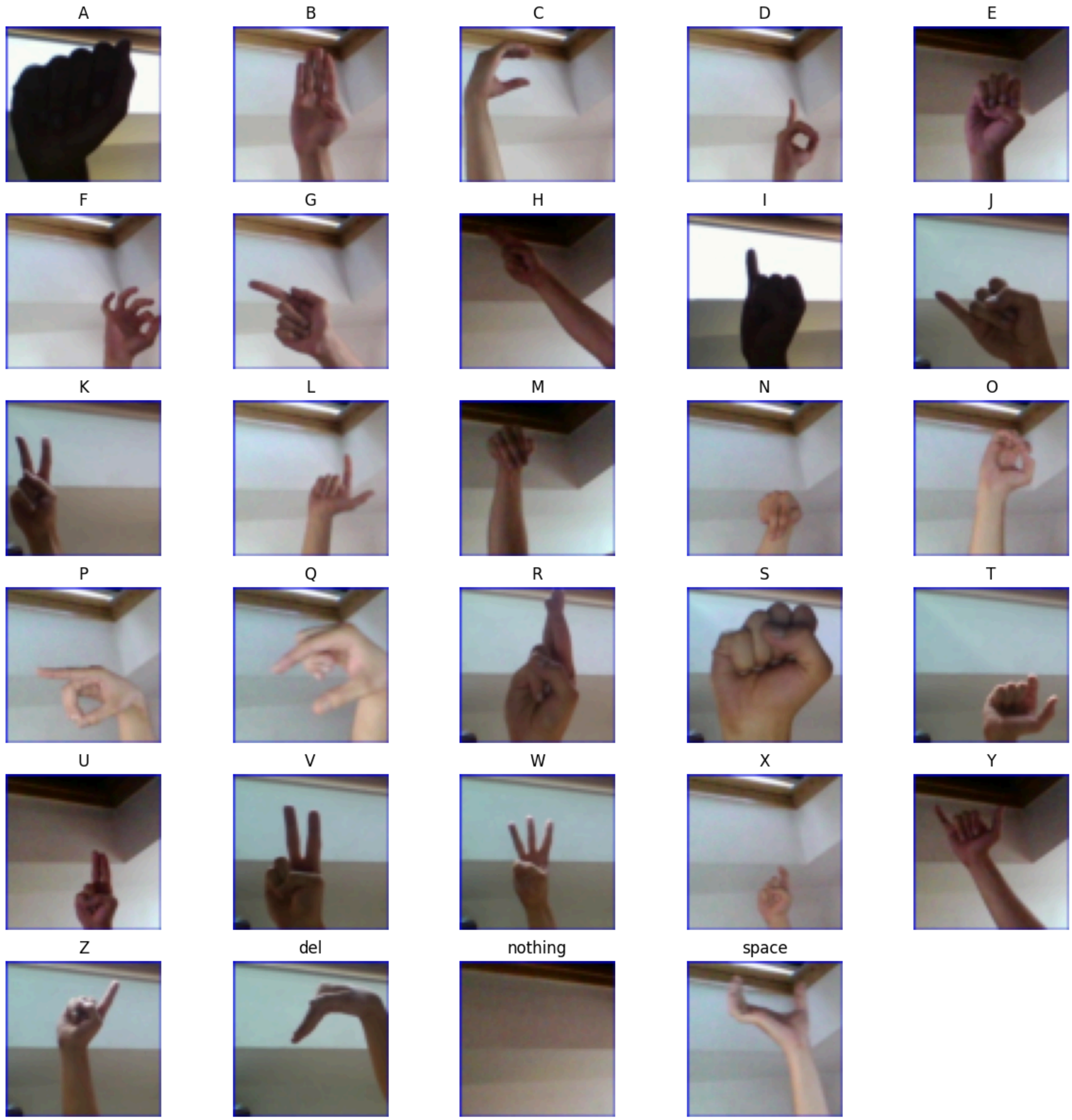
```
plt.imshow(image.permute(1, 2, 0))
plt.title(classes[label]);
```

```
Image shape: torch.Size([3, 64, 64])
```




```
# Get one image for each class
images = []
labels = []
for label in range(len(classes)):
    # Find the index of the first image with the current label
    index = next((i for i, (image, target) in enumerate(subset) if target == label), None)
    if index is not None:
        image, target = subset[index]
        images.append(image)
        labels.append(target)
```

```
# Plot images
fig = plt.figure(figsize=(15, 15))
rows, cols = 6, 5  # Adjust rows and cols to fit all classes
for i in range(len(images)):
    img, label = images[i], labels[i]

    # Reverse normalization for display as mentioned earlier
    img = img * 0.5 + 0.5

    fig.add_subplot(rows, cols, i + 1)
    plt.imshow(img.permute(1, 2, 0))
    plt.title(class_names[label])
    plt.axis(False)

```




```
train_data, test_data
```

```
(<torch.utils.data.dataset.Subset at 0x7e1fb489ec90>,
 <torch.utils.data.dataset.Subset at 0x7e1fb489e950>)
```

## 2 - Prepare DataLoader

This part prepares data batches for training and testing. First we defined our batch size. By using DataLoader, the training and testing datasets are converted into iterable objects. As we can see in the code the training dataloader shuffles the data in each epoch so that we prevent the model to learn the order of the data. But the testing Dataloader does not shuffle the data as we rather consistent evaluation during the testing.

Then we check out the output, and further inspect the data by retrieving a single batch and printing its shape to confirm it is in the format we expect.

Finally an image from the batch is shown along with its label and shape to ensure the data is loaded and labeled correctly. This process is important to make sure that we are in the right track.

```
# Setup the batch size hyperparameter
BATCH_SIZE = 32

# Turn datasets into iterables (batches)
train_dataloader = DataLoader(train_data, # dataset to turn into iterable
                              batch_size=BATCH_SIZE, # how many samples per batch?
                              shuffle=True) # shuffle data every epoch?

test_dataloader = DataLoader(test_data,
                             batch_size=BATCH_SIZE,
                             shuffle=False) # don't necessarily have to shuffle the testing data

train_dataloader, test_dataloader
```

```
(<torch.utils.data.dataloader.DataLoader at 0x7e1fb437c9d0>,
 <torch.utils.data.dataloader.DataLoader at 0x7e1fb4158c10>)
```

```
# Let's check out what we've created
print(f"Dataloaders: {train_dataloader, test_dataloader}")
print(f"Length of train dataloader: {len(train_dataloader)} batches of {BATCH_SIZE}")
print(f"Length of test dataloader: {len(test_dataloader)} batches of {BATCH_SIZE}")
```

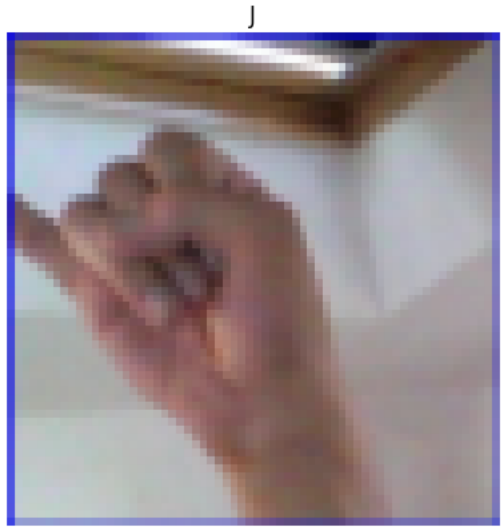
```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x7e1fb437c9d0>, <torch.utils.data.dataloader.DataLoader obj
Length of train dataloader: 125 batches of 32
Length of test dataloader: 32 batches of 32
```

```
# Check out what's inside the training dataloader
train_features_batch, train_labels_batch = next(iter(train_dataloader))
train_features_batch.shape, train_labels_batch.shape
```

```
(torch.Size([32, 3, 64, 64]), torch.Size([32]))
```

```
# Show a sample
torch.manual_seed(42)
random_idx = torch.randint(0, len(train_features_batch), size=[1]).item()
img, label = train_features_batch[random_idx], train_labels_batch[random_idx]

# Reverse normalization for display
img = img * 0.5 + 0.5

plt.imshow(img.permute(1, 2, 0))
plt.title(class_names[label])
plt.axis(False);
print(f"Image size: {img.shape}")
print(f"Label: {label}, label size: {label.shape}")
```

```
Image size: torch.Size([3, 64, 64])
Label: 9, label size: torch.Size([])
```




## 3 - BASELINE MODEL

After having worked on our dataset and dataloader, for the sake of getting to know our model better, we decided to implement a baseline model where we are using a flatten layer and where we are evaluating a simple sample. We implemented it because is the best way to know our model needs depending on how high the accuracy is or how low the loss is. Later on, we'll be creating more a non-linear model and a CNN model to see how much our model is learning in each one of them.

For making complex things we first have to start simple.

```python
# Create a flatten layer
flatten_model = nn.Flatten()

# Get a single sample
x = train_features_batch[0]

# Flatten the sample
output = flatten_model(x) # perform forward pass

# Print out what happened
print(f"Shape before flattening: {x.shape} -> [color_channels, height, width]")
print(f"Shape after flattening: {output.shape} -> [color_channels, height*width]")
```

```
Shape before flattening: torch.Size([3, 64, 64]) -> [color_channels, height, width]
Shape after flattening: torch.Size([3, 4096]) -> [color_channels, height*width]
```

```python
from torch import nn

# Baseline model definition along with its forward function
# we did it with just two linear layers to make it simple enough
class ASLBaseline(nn.Module):
  def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
    super().__init__()
    self.layer_stack = nn.Sequential(
        nn.Flatten(),
        nn.Linear(in_features=input_shape,
                  out_features=hidden_units),
        nn.Linear(in_features=hidden_units,
                  out_features=output_shape)
    )

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

```python
torch.manual_seed(42)

# Create an instance model_0
```

```
model_0 = ASLBaseline(
    input_shape = 3*64*64,
    hidden_units = 10,
    output_shape=len(class_names)
).to("cpu")

model_0
```

```
ASLBaseline(
  (layer_stack): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=12288, out_features=10, bias=True)
    (2): Linear(in_features=10, out_features=29, bias=True)
  )
)
```

```
model_0.state_dict()
```

```
OrderedDict([('layer_stack.1.weight',
              tensor([[ 6.8970e-03,  7.4876e-03, -2.1134e-03,  ...,  6.6472e-04,
                       -4.7116e-03, -5.2405e-03],
                      [-1.9907e-04,  3.1398e-03, -8.3313e-03,  ...,  4.0213e-03,
                        6.6041e-03,  5.0459e-03],
                      [ 6.0608e-03, -5.4492e-03,  7.6998e-03,  ...,  3.5302e-05,
                       -6.7376e-03,  7.3019e-03],
                      ...,
                      [-4.3874e-03,  1.9926e-04,  7.9915e-03,  ..., -5.0085e-03,
                       -3.9315e-03, -7.5393e-03],
                      [ 4.2025e-03, -1.3148e-03, -2.7806e-03,  ..., -5.4377e-03,
                       -4.8458e-03,  3.0181e-03],
                      [ 3.0575e-03,  5.3473e-03, -7.9494e-03,  ..., -8.9380e-03,
                        8.5260e-03,  2.1958e-03]])),
             ('layer_stack.1.bias',
              tensor([ 0.0036,  0.0022,  0.0069, -0.0080, -0.0077,  0.0016, -0.0080,  0.0013,
                       0.0007,  0.0080])),
             ('layer_stack.2.weight',
              tensor([[ 0.2271, -0.2909,  0.0526,  0.2420, -0.0226,  0.1177,  0.0311,  0.0199,
                        0.0727,  0.0551],
                      [-0.1879, -0.1115, -0.2149,  0.1204, -0.2628,  0.0700,  0.0217, -0.1841,
                       -0.1254, -0.0144],
                      [-0.1534,  0.1803, -0.1216, -0.2853,  0.1073,  0.1263, -0.2445,  0.1226,
                        0.1612, -0.2763],
                      [ 0.1309,  0.0899, -0.1710, -0.1101, -0.1523,  0.1366, -0.1008, -0.0388,
                       -0.1357, -0.1274],
                      [ 0.2154, -0.0421,  0.0956, -0.0118, -0.2286,  0.3034,  0.2304, -0.1224,
                        0.1647,  0.0040],
                      [ 0.1699, -0.0286,  0.1463, -0.2019, -0.1383, -0.0608, -0.0046,  0.1950,
                        0.2775, -0.1048],
                      [-0.2643, -0.2789,  0.2549,  0.1814, -0.2726,  0.0818, -0.0546, -0.2327,
                        0.1538,  0.3080],
                      [-0.2690,  0.2196,  0.2174, -0.2745,  0.1323, -0.0436, -0.0068, -0.1576,
                        0.0625,  0.0813],
                      [ 0.1939,  0.1772, -0.2130,  0.2027, -0.0021, -0.1484, -0.2197, -0.0111,
                        0.0386,  0.0530],
                      [-0.2387, -0.0817,  0.2055,  0.0559,  0.2870, -0.3095, -0.1049, -0.0042,
                       -0.1549,  0.3149],
                      [ 0.2623,  0.0288,  0.1195, -0.2930, -0.0005,  0.3093,  0.1367,  0.2194,
                        0.0865,  0.0384],
                      [ 0.0579,  0.1749, -0.1450, -0.0794,  0.2511, -0.1864, -0.1222,  0.1361,
                       -0.1253,  0.2578],
                      [-0.3055, -0.1784,  0.1948, -0.2012, -0.1424, -0.1791,  0.2238, -0.0434,
                       -0.1099,  0.2489],
                      [-0.1836, -0.1452,  0.3038,  0.0538,  0.0413,  0.2980, -0.0539,  0.2846,
                        0.1856, -0.0064],
                      [ 0.0710, -0.1581,  0.1057, -0.0919,  0.1336, -0.2215,  0.1867, -0.2535,
                       -0.0709,  0.1048],
                      [ 0.2143,  0.2477, -0.0449,  0.2926, -0.0164,  0.2143, -0.1815,  0.1759,
                        0.0603,  0.0334],
                      [ 0.0896, -0.1494,  0.2653, -0.0646, -0.2180,  0.0760, -0.0473,  0.1632,
                       -0.0911, -0.0837],
                      [-0.2410,  0.1765, -0.1889, -0.0837,  0.2526, -0.1805, -0.2914, -0.0554,
                       -0.0757,  0.1784],
                      [ 0.2267,  0.1200,  0.2232, -0.2571, -0.1645, -0.1279,  0.0547, -0.1862,
                       -0.0088,  0.0695],
                      [-0.2006,  0.1453, -0.1304, -0.0065,  0.2275,  0.1927, -0.1856,  0.0585,
                       -0.0081, -0.0397],
```

## 3.1 - Setup loss, optimizer and evaluation metrics

```
import requests
from pathlib import Path

# Download helper functions if we do not have it. This will help us having an accuracy value
if Path("helper_functions.py").is_file():
  print("helper_functions.py already exists, skipping download...")
else:
  print("Downloading helper_functions.py")
  request = requests.get("https://raw.githubusercontent.com/mrdbourke/pytorch-deep-learning/main/helper_functions.py")
  with open("helper_functions.py", "wb") as f:
    f.write(request.content)
```

```
helper_functions.py already exists, skipping download...
```

```
# Import accuracy metric
from helper_functions import accuracy_fn

# Setup loss function and optimizer
loss_fn = nn.CrossEntropyLoss() # For multi classification
optimizer = torch.optim.SGD(params=model_0.parameters(), # stochastic gradient descent as usual
                            lr=0.01)
```

## 3.2 - Creating a function to time our experiments

As ML is very experimental, two of the main things we'll often want to track are the model's performance and how fast it runs

Please find our timing function below:

```
from timeit import default_timer as timer

def print_train_time(start: float,
                     end: float,
                     device: torch.device = None):
  """Prints difference between start and end time."""
  total_time = end - start
  print(f"Train time on {device}: {total_time:.3f} seconds")
  return total_time
```

```
# Quick example to see how if it works
start_time = timer()

end_time = timer()
print_train_time(start=start_time, end=end_time, device="cpu")
```

```
Train time on cpu: 0.000 seconds
4.5781999915561755e-05
```

## 3.3 - Creating a training loop and training model on batches of data

```
# Import tqdm for progress bar
from tqdm.auto import tqdm

# Set the seed and start the timer
torch.manual_seed(42)
train_time_start_on_cpu = timer()

# Set the number of epochs (we'll keep this small for faster training time as mentioned in our first presentation)
epochs = 3

# Create training and test loop
for epoch in tqdm(range(epochs)):
  print(f"Epoch: {epoch}\n------")
  ### Training
  train_loss = 0
  # Add a loop to loop through the training batches
  for batch, (X, y) in enumerate(train_dataloader):
    model_0.train()
    # 1. Forward pass
    y_pred = model_0(X)

```

```
    # 2. Calculate loss (per batch)
    loss = loss_fn(y_pred, y)
    train_loss += loss # accumulate train loss

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Optimizer step (update the model's parameters once *per batch*)
    optimizer.step()

    # Print out what's happening
    if batch % 400 == 0:
        print(f"Looked at {batch * len(X)}/{len(train_dataloader.dataset)} samples.")

  # Divide total train loss by length of train dataloader
  train_loss /= len(train_dataloader)

  ### Testing
  test_loss, test_acc = 0, 0
  model_0.eval()
  with torch.inference_mode():
    for X_test, y_test in test_dataloader:
      # 1. Forward pass
      test_pred = model_0(X_test)

      # 2. Calculate loss (accumulatively)
      test_loss += loss_fn(test_pred, y_test)

      # 3. Calculate accuracy
      test_acc += accuracy_fn(y_true=y_test, y_pred=test_pred.argmax(dim=1))

    # Calculate the test loss average per batch
    test_loss /= len(test_dataloader)

    # Calculate the test acc average per batch
    test_acc /= len(test_dataloader)

  # Print out what's happening
  print(f"\nTrain loss: {train_loss:.4f} | Test loss: {test_loss:.4f}, Test acc: {test_acc:.4f}")

# Calculate training time
train_time_end_on_cpu = timer()
total_train_time_model_0 = print_train_time(start=train_time_start_on_cpu,
                                            end=train_time_end_on_cpu,
                                            device=str(next(model_0.parameters()).device))
```

```
100%                                   3/3 [00:21<00:00,  6.90s/it]
Epoch: 0
------
Looked at 0/4000 samples.

Train loss: 3.2142 | Test loss: 3.0513, Test acc: 16.1133
Epoch: 1
------
Looked at 0/4000 samples.

Train loss: 2.9499 | Test loss: 2.9235, Test acc: 18.3594
Epoch: 2
------
Looked at 0/4000 samples.

Train loss: 2.7644 | Test loss: 2.8075, Test acc: 19.4336
Train time on cpu: 21.860 seconds
```

## 3.4 - Make predictions and get Model 0 results

```
torch.manual_seed(42)
def eval_model(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               accuracy_fn):
  """Returns a dictionary containing the results of model predicting on data_loader."""
```

```
  loss, acc = 0, 0
  model.eval()
  with torch.inference_mode():
    for X, y in tqdm(data_loader):
      # Make predictions
      y_pred = model(X)

      # Accumulate the loss and acc values per batch
      loss += loss_fn(y_pred, y)
      acc += accuracy_fn(y_true=y,
                         y_pred=y_pred.argmax(dim=1))

    # Scale loss and acc to find the average loss/acc per batch
    loss /= len(data_loader)
    acc /= len(data_loader)

  return {"model_name": model.__class__.__name__, # only works when model was created with a class
          "model_loss": loss.item(),
          "model_acc": acc}

# Calculate model 0 results on test dataset
model_0_results = eval_model(model=model_0,
                             data_loader=test_dataloader,
                             loss_fn=loss_fn,
                             accuracy_fn=accuracy_fn)
model_0_results
```

100% 32/32 [00:01<00:00, 21.88it/s]

```
{'model_name': 'ASLBaseline',
 'model_loss': 2.8074750900268555,
 'model_acc': 19.43359375}
```

## 3.5 Setup device agnostic-code (for using a GPU if there is one)

```
!nvidia-smi
```

```
Fri Apr 25 17:10:38 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   47C    P8              9W /   70W |       2MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

```
torch.cuda.is_available()
```

```
True
```

```
# Setup device-agnostic code
import torch
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

```
'cuda'
```

## 4 - Model 1: Building a better model with non-linearity

Once built our baseline model, we've noticed the lack of accuracy as well as a high loss, reason why we've decided to implement a better model with non-linearity.

```
# Create a model with non-linear and linear layers along with its forward function
# Here we are using ReLU unlike the baseline model where we trated it as a linear model
class ASLModelV1(nn.Module):
  def __init__(self,
               input_shape: int,
               hidden_units: int,
               output_shape: int):
    super().__init__()
    self.layer_stack = nn.Sequential(
        nn.Flatten(), # flatten inputs into a single vector
        nn.Linear(in_features=input_shape,
                  out_features=hidden_units),
        nn.ReLU(),
        nn.Linear(in_features=hidden_units,
                  out_features=output_shape),
        nn.ReLU()
    )

  def forward(self, x: torch.Tensor):
    return self.layer_stack(x)
```

```
# Create an instance of model_1
torch.manual_seed(42)
model_1 = ASLModelV1(input_shape=12288, # this is the output of the flatten after our 3*64*64 image goes in
                              hidden_units=10,
                              output_shape=len(class_names)).to(device) # send to the GPU if it's available
next(model_1.parameters()).device
```

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

## 4.1 - Setup loss, optimizer and evaluation metrics

```
from helper_functions import accuracy_fn
loss_fn = nn.CrossEntropyLoss() # measure how wrong our model is
optimizer = torch.optim.SGD(params=model_1.parameters(), # tries to update our model's parameters to reduce the loss
                            lr=0.01)
```

## 4.2 - Functoinizing training and evaluation/testing loops

```
# Here we are defining train_step in order to avoid rewriting it over and over again in case we implement more models
# in that way we would be saving memory and time, and the process would run smoothly and be more efficient
def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               accuracy_fn,
               device: torch.device = device):
  """Performs a training with model trying to learn on data_loader."""
  train_loss, train_acc = 0, 0

  # Put model into training mode
  model.train()

  # Add a loop to loop through the training batches
  for batch, (X, y) in enumerate(data_loader):
    # Put data on target device
    X, y = X.to(device), y.to(device)

    # 1. Forward pass (outputs the raw logits from the model)
    y_pred = model(X)

    # 2. Calculate loss and accuracy (per batch)
    loss = loss_fn(y_pred, y)
    train_loss += loss # accumulate train loss
    train_acc += accuracy_fn(y_true=y,
                             y_pred=y_pred.argmax(dim=1)) # go from logits -> prediction labels

```

```
    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Optimizer step (update the model's parameters once *per batch*)
    optimizer.step()

  # Divide total train loss and acc by length of train dataloader
  train_loss /= len(data_loader)
  train_acc /= len(data_loader)
  print(f"Train loss: {train_loss:.5f} | Train acc: {train_acc:.2f}%")

  # return the loss value and accuracy for future linear plots
  return train_loss.item(), train_acc
```

```
# Here we are defining test_step in order to avoid rewriting it over and over again in case we implement more models
# in that way we would be saving memory and time, and the process would run smoothly and be more efficient
def test_step(model: torch.nn.Module,
              data_loader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              accuracy_fn,
              device: torch.device = device):
  """Performs a testing loop step on model going over data_loader."""
  test_loss, test_acc = 0, 0

  # Put the model in eval mode
  model.eval()

  # Turn on inference mode context manager
  with torch.inference_mode():
    for X, y in data_loader:
      # Send the data to the target device
      X, y = X.to(device), y.to(device)

      # 1. Forward pass (outputs raw logits)
      test_pred = model(X)

      # 2. Calculuate the loss/acc
      test_loss += loss_fn(test_pred, y)
      test_acc += accuracy_fn(y_true=y,
                              y_pred=test_pred.argmax(dim=1)) # go from logits -> prediction labels

    # Adjust metrics and print out
    test_loss /= len(data_loader)
    test_acc /= len(data_loader)
    print(f"Test loss: {test_loss:.5f} | Test acc: {test_acc:.2f}%\n")

    # return the loss value and accuracy for future plots
    return test_loss.item(), test_acc
```

```
# Here we are just calling the functions above along with the timer
torch.manual_seed(42)

# Measure time
from timeit import default_timer as timer
train_time_start_on_gpu = timer()

# Set epochs
epochs = 3

# Create a optimization and evaluation loop using train_step() and test_step()
for epoch in tqdm(range(epochs)):
  print(f"Epoch: {epoch}\n----------")
  train_step(model=model_1,
             data_loader=train_dataloader,
             loss_fn=loss_fn,
             optimizer=optimizer,
             accuracy_fn=accuracy_fn,
             device=device)
  test_step(model=model_1,
            data_loader=test_dataloader,
            loss_fn=loss_fn,
            accuracy_fn=accuracy_fn,
            device=device)
```

```python

train_time_end_on_gpu = timer()
total_train_time_model_1 = print_train_time(start=train_time_start_on_gpu,
                                            end=train_time_end_on_gpu,
                                            device=device)
```

```
100%                                    3/3 [00:18<00:00,  5.97s/it]
Epoch: 0
----------
Train loss: 3.31834 | Train acc: 7.97%
Test loss: 3.26682 | Test acc: 11.23%

Epoch: 1
----------
Train loss: 3.21522 | Train acc: 12.47%
Test loss: 3.17533 | Test acc: 14.45%

Epoch: 2
----------
Train loss: 3.11438 | Train acc: 16.30%
Test loss: 3.09349 | Test acc: 15.14%

Train time on cuda: 18.100 seconds
```

```python
model_0_results
```

```
{'model_name': 'ASLBaseline',
 'model_loss': 2.8074750900268555,
 'model_acc': 19.43359375}
```

```python
# Train time
total_train_time_model_0
```

```
21.860167875000116
```

```python
torch.manual_seed(42)
def eval_model(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               accuracy_fn,
               device=device):
  """Returns a dictionary containing the results of model predicting on data_loader."""
  loss, acc = 0, 0
  model.eval()
  with torch.inference_mode():
    for X, y in tqdm(data_loader):
      # Make our data device agnostic
      X, y = X.to(device), y.to(device)
      # Make predictions
      y_pred = model(X)

      # Accumulate the loss and acc values per batch
      loss += loss_fn(y_pred, y)
      acc += accuracy_fn(y_true=y,
                         y_pred=y_pred.argmax(dim=1))

    # Scale loss and acc to find the average loss/acc per batch
    loss /= len(data_loader)
    acc /= len(data_loader)

  return {"model_name": model.__class__.__name__, # only works when model was created with a class
          "model_loss": loss.item(),
          "model_acc": acc}
```

```python
# Get model_1 results dictionary
model_1_results = eval_model(model=model_1,
                             data_loader=test_dataloader,
                             loss_fn=loss_fn,
                             accuracy_fn=accuracy_fn,
                             device=device)
model_1_results
```

100% 32/32 [00:01<00:00, 23.82it/s]

```
{'model_name': 'ASLModelV1',
 'model_loss': 3.0934855937957764,
 'model_acc': 15.13671875}
```

```
model_0_results
```

```
{'model_name': 'ASLBaseline',
 'model_loss': 2.8074750900268555,
 'model_acc': 19.43359375}
```

## 5 - ASL Model 2: Building a Convolutional Neural Network (CNN)

Now after having implemented a baseline model and a better non-linear model, we've noticed that the accuracy can't even go beyond 50% which is kinda concerning, reason why we are going to implement Convolutional Neural Networks where we would be using two blocks for the data to go through along with 10 hidden units, please find it below

```
# Create a convolutional neural network
class ASLModelV2(nn.Module):
  def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
    super().__init__()
    self.block_1 = nn.Sequential(
        nn.Conv2d(in_channels = input_shape,
                  out_channels = hidden_units,
                  kernel_size = 3,
                  stride = 1,
                  padding = 1),
        nn.ReLU(),
        nn.Conv2d(in_channels = hidden_units,
                  out_channels = hidden_units,
                  kernel_size = 3,
                  stride = 1,
                  padding = 1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = 2,
                     stride = 2)
    )
    self.block_2 = nn.Sequential(
        nn.Conv2d(hidden_units, hidden_units, 3, padding = 1),
        nn.ReLU(),
        nn.Conv2d(hidden_units, hidden_units, 3, padding = 1),
        nn.ReLU(),
        nn.MaxPool2d(2)
    )
    self.classifier = nn.Sequential(
        nn.Flatten(),
        nn.Linear(in_features = hidden_units * 16 * 16, # used to be 7 * 7
                  out_features = output_shape)
    )

  def forward(self, x: torch.Tensor):
    x = self.block_1(x)
    #print(f"Output shape of conv_block_1: {x.shape}")
    x = self.block_2(x)
    #print(f"Output shape of conv_block_2: {x.shape}")
    x = self.classifier(x)
    #print(f"Output shape of classifier: {x.shape}")
    return x

```

```
torch.manual_seed(42)
model_2 = ASLModelV2(input_shape = 3,
                 hidden_units = 10,
                 output_shape = len(class_names)).to(device)
model_2
```

```
ASLModelV2(
  (block_1): Sequential(
    (0): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

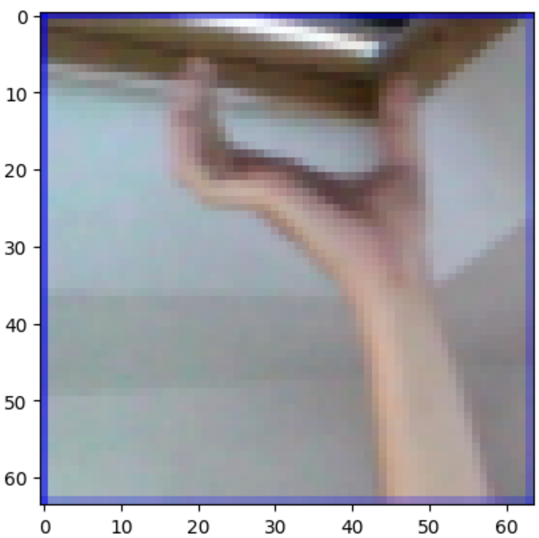
```
      (3): ReLU()
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (block_2): Sequential(
      (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU()
      (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU()
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
      (0): Flatten(start_dim=1, end_dim=-1)
      (1): Linear(in_features=2560, out_features=29, bias=True)
    )
  )
```

```
# Pass image through model
rand_image_tensor = torch.randn(size=(1, 3, 64, 64))

# Pass image through model
model_2(rand_image_tensor.to(device))
```

```
tensor([[ 0.1282,  0.0933,  0.0602, -0.0449, -0.0577,  0.0749, -0.0905,  0.1415,
          0.0035,  0.0949, -0.0194, -0.0308,  0.0029, -0.0799,  0.1232,  0.0477,
          0.0469,  0.0493, -0.0039,  0.1222, -0.0345, -0.0842,  0.0605,  0.0132,
          0.0390,  0.1593,  0.0406,  0.1068,  0.0161]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

```
image = image * 0.5 + 0.5
plt.imshow(image.permute(1, 2, 0))
```

```
<matplotlib.image.AxesImage at 0x7e1f9c058690>
```




```
model_2.state_dict()
```

```
OrderedDict([('block_1.0.weight',
              tensor([[[[ 0.1471,  0.1597, -0.0451],
                        [ 0.1768, -0.0422,  0.0388],
                        [-0.0937,  0.1130,  0.1697]],

                       [[-0.1412,  0.1673,  0.0360],
                        [ 0.1422,  0.0261,  0.0928],
                        [-0.0272,  0.1484,  0.0284]],

                       [[-0.0898,  0.0491, -0.0887],
                        [-0.0226, -0.0782,  0.1277],
                        [-0.1519, -0.0887, -0.0543]]],


                      [[[-0.1157,  0.0182, -0.1901],
                        [ 0.1738, -0.1635,  0.1486],
                        [ 0.0320, -0.0625,  0.1189]],

                       [[ 0.0300,  0.1555,  0.0210],
                        [-0.0607,  0.0517, -0.0522],
                        [ 0.0810,  0.1718,  0.1112]],
```

```
          [[-0.0841,  0.1111,  0.0344],
           [ 0.0977, -0.1173, -0.1905],
           [-0.0744, -0.1476,  0.1579]]],


         [[[ 0.0554,  0.0797,  0.0609],
           [-0.0033,  0.1506, -0.1367],
           [ 0.0121, -0.1314,  0.0593]],

          [[-0.0663,  0.0590, -0.0401],
           [ 0.1596, -0.1141, -0.1148],
           [-0.1148,  0.1731,  0.0641]],

          [[ 0.1852, -0.1588, -0.1909],
           [-0.1506, -0.1295,  0.0780],
           [ 0.0689,  0.1599, -0.0994]]],


         [[[-0.1312,  0.1021, -0.0778],
           [ 0.1168, -0.0457,  0.1101],
           [-0.1495, -0.0971,  0.0587]],

          [[ 0.0407, -0.0491,  0.1147],
           [ 0.1308, -0.1396, -0.1027],
           [ 0.1762, -0.0649, -0.0682]],

          [[-0.1862, -0.1102,  0.0481],
           [-0.0254, -0.1397,  0.0045],
           [-0.1315, -0.1633, -0.1060]]],


         [[[-0.1684, -0.1225,  0.1924],
           [ 0.0363,  0.0593, -0.1795],
           [-0.1264, -0.0641,  0.0301]],

          [[-0.1693, -0.0829, -0.1152],
```

## 5.1 - Stepping through `nn.Conv2d()`

```python
torch.manual_seed(42)

# Create sample batch of random numbers with same size as image batch
images = torch.randn(size=(32, 3, 64, 64)) # [batch_size, color_channels, height, width]
test_image = images[0] # get a single image for testing

print(f"Image batch shape: {images.shape} -> [batch_size, color_channels, height, width]")
print(f"Single image shape: {test_image.shape} -> [color_channels, height, width]")
print(f"Single image pixel values:\n{test_image}")
```

```
Image batch shape: torch.Size([32, 3, 64, 64]) -> [batch_size, color_channels, height, width]
Single image shape: torch.Size([3, 64, 64]) -> [color_channels, height, width]
Single image pixel values:
tensor([[[ 1.9269,  1.4873,  0.9007,  ...,  1.8446, -1.1845,  1.3835],
         [ 1.4451,  0.8564,  2.2181,  ...,  0.3399,  0.7200,  0.4114],
         [ 1.9312,  1.0119, -1.4364,  ..., -0.5558,  0.7043,  0.7099],
         ...,
         [-0.5610, -0.4830,  0.4770,  ..., -0.2713, -0.9537, -0.6737],
         [ 0.3076, -0.1277,  0.0366,  ..., -2.0060,  0.2824, -0.8111],
         [-1.5486,  0.0485, -0.7712,  ..., -0.1403,  0.9416, -0.0118]],

        [[-0.5197,  1.8524,  1.8365,  ...,  0.8935, -1.5114, -0.8515],
         [ 2.0818,  1.0677, -1.4277,  ...,  1.6612, -2.6223, -0.4319],
         [-0.1010, -0.4388, -1.9775,  ...,  0.2106,  0.2536, -0.7318],
         ...,
         [ 0.2779,  0.7342, -0.3736,  ..., -0.4601,  0.1815,  0.1850],
         [ 0.7205, -0.2833,  0.0937,  ..., -0.1002, -2.3609,  2.2465],
         [-1.3242, -0.1973,  0.2920,  ...,  0.5409,  0.6940,  1.8563]],

        [[-0.7978,  1.0261,  1.1465,  ...,  1.2134,  0.9354, -0.0780],
         [-1.4647, -1.9571,  0.1017,  ..., -1.9986, -0.7409,  0.7011],
         [-1.3938,  0.8466, -1.7191,  ..., -1.1867,  0.1320,  0.3407],
         ...,
         [ 0.8206, -0.3745,  1.2499,  ..., -0.0676,  0.0385,  0.6335],
         [-0.5589, -0.3393,  0.2347,  ...,  2.1181,  2.4569,  1.3083],
         [-0.4092,  1.5199,  0.2401,  ..., -0.2558,  0.7870,  0.9924]]])
```

```python
test_image.shape
```

```
torch.Size([3, 64, 64])
```

```
torch.manual_seed(42)

# Create a convolutional layer
conv_layer = nn.Conv2d(in_channels=3,
                       out_channels=10,
                       kernel_size=(3, 3),
                       stride=1,
                       padding=0)

# Pass the data through the convolutional layer
conv_output = conv_layer(test_image.unsqueeze(dim=0)) # add an extra dimension for batch
conv_output.shape
```

```
torch.Size([1, 10, 62, 62])
```

```
# Add extra dimension to test image
test_image.unsqueeze(dim=0).shape
```

```
torch.Size([1, 3, 64, 64])
```

## 5.2 - Stepping through `nn.MaxPool2d()`

```
test_image.shape
```

```
torch.Size([3, 64, 64])
```

```
# Print out original image shape without and with unsqueezed dimension
print(f"Test image original shape: {test_image.shape}")
print(f"Test image with unsqueezed dimension: {test_image.unsqueeze(dim=0).shape}")

# Create a sample nn.MaxPoo2d() layer
max_pool_layer = nn.MaxPool2d(kernel_size=2)

# Pass data through just the conv_layer
test_image_through_conv = conv_layer(test_image.unsqueeze(dim=0))
print(f"Shape after going through conv_layer(): {test_image_through_conv.shape}")

# Pass data through the max pool layer
test_image_through_conv_and_max_pool = max_pool_layer(test_image_through_conv)
print(f"Shape after going through conv_layer() and max_pool_layer(): {test_image_through_conv_and_max_pool.shape}")
```

```
Test image original shape: torch.Size([3, 64, 64])
Test image with unsqueezed dimension: torch.Size([1, 3, 64, 64])
Shape after going through conv_layer(): torch.Size([1, 10, 62, 62])
Shape after going through conv_layer() and max_pool_layer(): torch.Size([1, 10, 31, 31])
```

```
torch.manual_seed(42)
# Create a random tensor with a similar number of dimensions to our images
random_tensor = torch.randn(size=(1, 1, 2, 2))
print(f"Random tensor:\n{random_tensor}")
print(f"Random tensor shape: {random_tensor.shape}")

# Create a max pool layer
max_pool_layer = nn.MaxPool2d(kernel_size=2) # see what happens when you change the kernel_size value

# Pass the random tensor through the max pool layer
max_pool_tensor = max_pool_layer(random_tensor)
print(f"\nMax pool tensor:\n{max_pool_tensor} <- this is the maximum value from random_tensor")
print(f"Max pool tensor shape: {max_pool_tensor.shape}")
```

```
Random tensor:
tensor([[[[0.3367, 0.1288],
          [0.2345, 0.2303]]]])
Random tensor shape: torch.Size([1, 1, 2, 2])

Max pool tensor:
tensor([[[[0.3367]]]]) <- this is the maximum value from random_tensor
Max pool tensor shape: torch.Size([1, 1, 1, 1])
```

## 5.3 - Setup a loss function and optimizer for model

```
# Import accuracy metric
from helper_functions import accuracy_fn # Note: could also use torchmetrics.Accuracy(task = 'multiclass', num_classes=len(c

# Setup loss and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_2.parameters(),
                       lr = 0.001)

# Good try :(
"""loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(params=model_2.parameters(),
                            lr=0.1)"""
```

```
'loss_fn = nn.CrossEntropyLoss()\noptimizer = torch.optim.SGD(params=model_2.parameters(),\n                            lr=
0.1)'
```

## 5.4 - Training and Testing model using our training and test functions

```
torch.manual_seed(42)
torch.cuda.manual_seed(42)

# Measure time
from timeit import default_timer as timer
train_time_start_model_2 = timer()

# Train and test model
epochs = 11
for epoch in tqdm(range(epochs)):
    print(f"Epoch: {epoch}\n---------")
    train_step(data_loader=train_dataloader,
               model=model_2,
               loss_fn=loss_fn,
               optimizer=optimizer,
               accuracy_fn=accuracy_fn,
               device=device)
    test_step(data_loader=test_dataloader,
              model=model_2,
              loss_fn=loss_fn,
              accuracy_fn=accuracy_fn,
              device=device)

train_time_end_model_2 = timer()
total_train_time_model_2 = print_train_time(start=train_time_start_model_2,
                                            end=train_time_end_model_2,
                                            device=device)
```

```
100%                                    11/11 [01:10<00:00,  6.25s/it]
Epoch: 0
---------
Train loss: 3.09838 | Train acc: 12.35%
Test loss: 2.53919 | Test acc: 26.07%

Epoch: 1
---------
Train loss: 2.06714 | Train acc: 39.23%
Test loss: 1.83359 | Test acc: 46.19%

Epoch: 2
---------
Train loss: 1.35661 | Train acc: 58.58%
Test loss: 1.55622 | Test acc: 54.69%

Epoch: 3
---------
Train loss: 0.89766 | Train acc: 71.38%
Test loss: 1.30300 | Test acc: 63.57%

Epoch: 4
---------
Train loss: 0.61466 | Train acc: 80.40%
Test loss: 1.31237 | Test acc: 64.36%

Epoch: 5
---------
Train loss: 0.39621 | Train acc: 87.28%
Test loss: 1.40472 | Test acc: 68.36%

Epoch: 6
---------
Train loss: 0.27861 | Train acc: 91.12%
Test loss: 1.20559 | Test acc: 73.54%

Epoch: 7
---------
Train loss: 0.19016 | Train acc: 93.80%
Test loss: 1.43037 | Test acc: 72.85%

Epoch: 8
---------
Train loss: 0.14587 | Train acc: 95.40%
Test loss: 1.44314 | Test acc: 72.27%

Epoch: 9
---------
Train loss: 0.10936 | Train acc: 96.83%
Test loss: 1.50674 | Test acc: 73.83%

Epoch: 10
---------
Train loss: 0.07435 | Train acc: 97.53%
Test loss: 1.59951 | Test acc: 74.41%

Train time on cuda: 70.527 seconds
```

```python
# Get model_2 results
model_2_results = eval_model(
    model=model_2,
    data_loader=test_dataloader,
    loss_fn=loss_fn,
    accuracy_fn=accuracy_fn
)
model_2_results
```

```
100%                                    32/32 [00:01<00:00, 27.43it/s]
{'model_name': 'ASLModelV2',
 'model_loss': 1.599514365196228,
 'model_acc': 74.4140625}
```

## 6 - Compare model results and training time

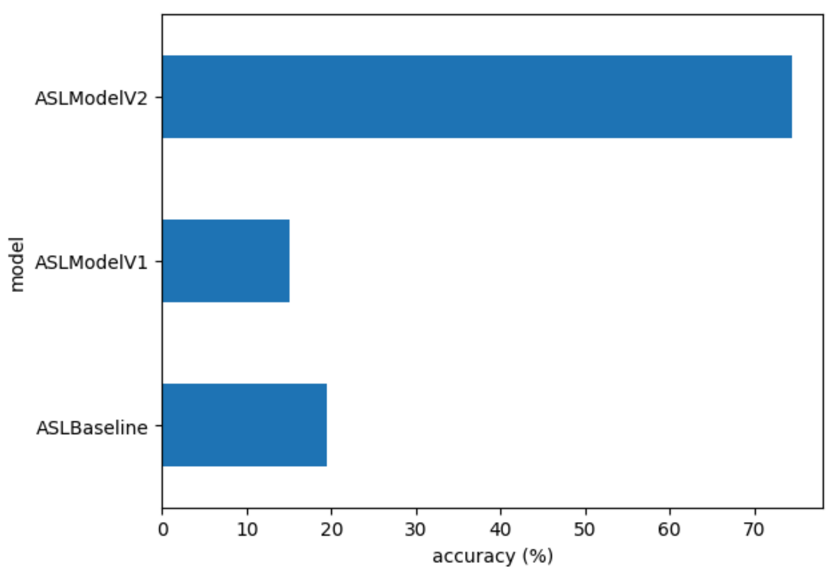
```
import pandas as pd
compare_results = pd.DataFrame([model_0_results,
                                model_1_results,
                                model_2_results])
compare_results
```

| | model_name | model_loss | model_acc |
|---|---|---|---|
| 0 | ASLBaseline | 2.807475 | 19.433594 |
| 1 | ASLModelV1 | 3.093486 | 15.136719 |
| 2 | ASLModelV2 | 1.599514 | 74.414062 |

```
# Add training time to results comparison
compare_results["training_time"] = [total_train_time_model_0,
                                    total_train_time_model_1,
                                    total_train_time_model_2]
compare_results
```

| | model_name | model_loss | model_acc | training_time |
|---|---|---|---|---|
| 0 | ASLBaseline | 2.807475 | 19.433594 | 21.860168 |
| 1 | ASLModelV1 | 3.093486 | 15.136719 | 18.099551 |
| 2 | ASLModelV2 | 1.599514 | 74.414062 | 70.526626 |

```
# Visualize our model results
compare_results.set_index("model_name")["model_acc"].plot(kind="barh")
plt.xlabel("accuracy (%)")
plt.ylabel("model");
```




## 7 - Make and evaluate random predictions

```
def make_predictions(model: torch.nn.Module,
                     data: list,
                     device: torch.device = device):
  pred_probs = []
  model.to(device)
  model.eval()
  with torch.inference_mode():
    for sample in data:
      # Prepare the sample (add a batch dimension and pass to target device)
      sample = torch.unsqueeze(sample, dim=0).to(device)

      # Forward pass (model outputs raw logits)
      pred_logit = model(sample)

      # Get prediction probability (logit -> prediction probability)
```

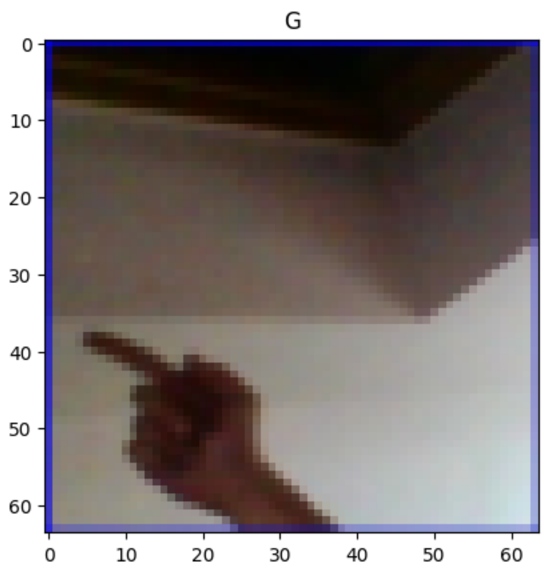
```
      pred_prob = torch.softmax(pred_logit.squeeze(), dim=0)

      # Get pred_prob off the GPU for further calculations
      pred_probs.append(pred_prob.cpu())

  # Stack the pred_probs to turn list into a tensor
  return torch.stack(pred_probs)
```

```
import random
# random.seed(42)
test_samples = []
test_labels = []
for sample, label in random.sample(list(test_data), k=9):
  test_samples.append(sample)
  test_labels.append(label)

# View the first sample shape
test_samples[0].shape
```

```
torch.Size([3, 64, 64])
```

```
normalized_test_samples = [sample * 0.5 + 0.5 for sample in test_samples] # Reverse normalization
plt.imshow(normalized_test_samples[0].permute(1, 2, 0))
plt.title(class_names[test_labels[0]])
```

```
Text(0.5, 1.0, 'G')
```




```
# Make predictions
pred_probs = make_predictions(model=model_2,
                              data=test_samples)

# View first two prediction probabilities
pred_probs[:2]
```

```
tensor([[6.8947e-10, 3.6740e-12, 1.2474e-09, 8.9101e-15, 1.2081e-14, 2.0770e-04,
         1.1964e-03, 6.6200e-09, 1.6727e-10, 7.6147e-08, 3.1653e-11, 2.9266e-14,
         4.9254e-07, 4.4872e-07, 3.1988e-12, 3.6626e-09, 5.2994e-20, 7.1557e-03,
         1.3134e-17, 3.3368e-08, 5.3310e-11, 1.6446e-02, 4.1875e-12, 1.7497e-06,
         3.8812e-13, 4.3434e-18, 4.8906e-10, 7.8114e-13, 9.7499e-01],
        [9.6114e-13, 4.5881e-17, 3.3778e-12, 1.7567e-10, 2.1434e-08, 2.3279e-10,
         3.1673e-10, 1.3308e-10, 4.2215e-07, 2.0996e-05, 6.0409e-09, 5.8643e-07,
         2.3985e-02, 2.4080e-03, 2.5756e-06, 2.0818e-06, 8.0176e-06, 8.2397e-05,
         1.5705e-05, 4.3257e-05, 1.5224e-05, 4.8218e-03, 8.1809e-02, 7.1317e-04,
         8.7580e-04, 4.4739e-05, 5.9217e-02, 3.6400e-13, 8.2593e-01]])
```

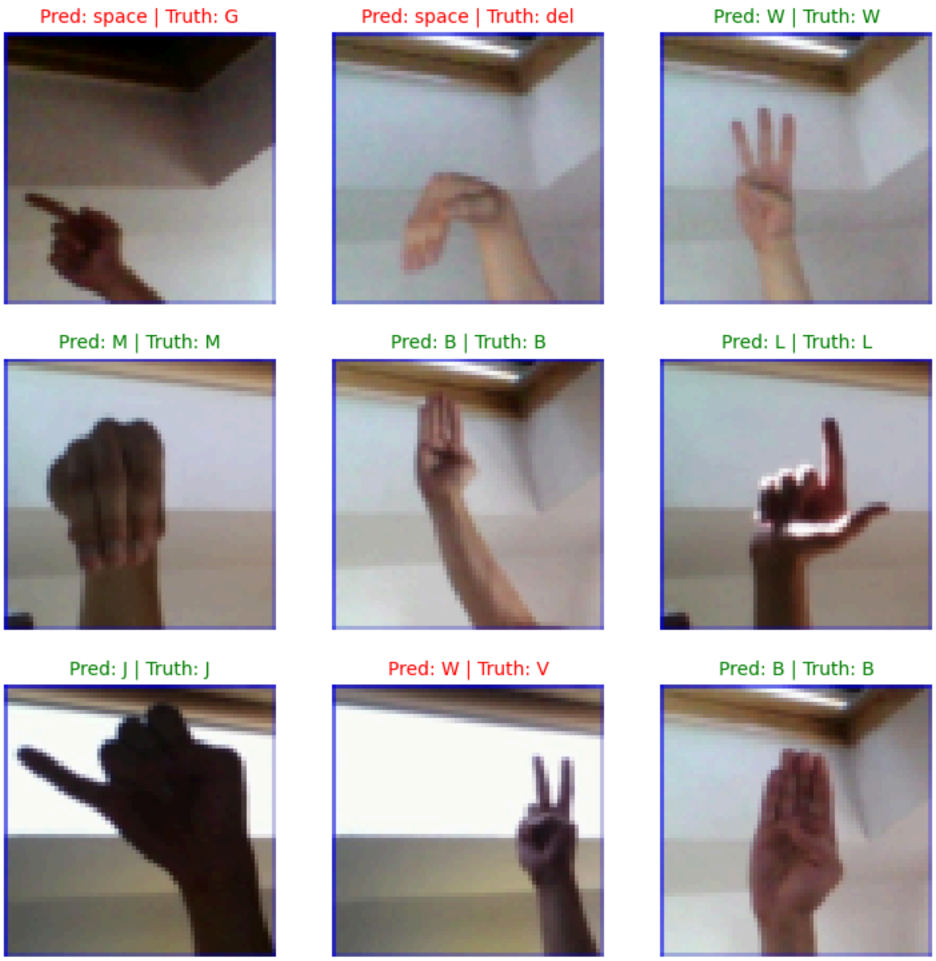
```
# Convert prediction probabilities to labels
pred_classes = pred_probs.argmax(dim=1)
pred_classes
```

```
tensor([28, 28, 22, 12,  1, 11,  9, 22,  1])
```

```
test_labels
```

```
[6, 26, 22, 12, 1, 11, 9, 21, 1]
```

```
# Plot predictions
plt.figure(figsize=(9, 9))
nrows = 3
ncols = 3
for i, sample in enumerate(test_samples):
  # Create subplot
  plt.subplot(nrows, ncols, i+1)

  normalized_sample = sample * 0.5 + 0.5

  # Plot the target image
  plt.imshow(normalized_sample.permute(1, 2, 0))

  # Find the prediction (in text form, e.g "Sandal")
  pred_label = class_names[pred_classes[i]]

  # Get the truth label (in text form)
  truth_label = class_names[test_labels[i]]

  # Create a title for the plot
  title_text = f"Pred: {pred_label} | Truth: {truth_label}"

  # Check for equality between pred and truth and change color of title text
  if pred_label == truth_label:
    plt.title(title_text, fontsize=10, c="g") # green text if prediction same as truth
  else:
    plt.title(title_text, fontsize=10, c="r")

  plt.axis(False);
```




Great! Now we can tell our model is actually learning something and not getting stuck in some accuracy value below 50% :,(

## 8 - Making a confusion matrix for further prediction evaluation

For those who doesn't know, a confustion matrix is an efficient way of evaluating our classification models by showcasing data visualizations where we can tell how accurate every class is based on the number of sample each one of them have

```
# Import tqdm.auto
from tqdm.auto import tqdm


# 1. Make predictions with trained model
y_preds = []
model_2.eval()
with torch.inference_mode():
  for X, y in tqdm(test_dataloader, desc="Making predictions..."):
    # Send the data and targets to target device
    X, y = X.to(device), y.to(device)
    # Do the forward pass
    y_logit = model_2(X)
    # Turn predictions from logits -> prediction probabilities -> prediction labels
    y_pred = torch.softmax(y_logit.squeeze(), dim=0).argmax(dim=1)
    # Put prediction on CPU for evaluation
    y_preds.append(y_pred.cpu())

# Concatenate list of predictions into a tensor
y_pred_tensor = torch.cat(y_preds)
y_pred_tensor
```

```
Making predictions...: 100%                                    32/32 [00:01<00:00, 25.45it/s]
tensor([ 5, 19,  3, 24, 10, 13,  1, 17, 26,  3, 12, 22,  4, 24, 14,  0, 11,  3,
        27, 19, 13,  2,  0,  9,  4,  7, 20, 14, 14, 17, 12,  8,  2,  7, 14,  3,
        17, 17, 23, 25, 15, 20, 13,  8,  5,  4,  6, 19, 20,  7, 22, 23, 24, 14,
        19, 12, 28, 19, 21, 16, 13, 28, 26, 21, 26,  6, 28, 25,  2,  4, 22, 21,
         8, 10, 19, 20, 13, 11,  2,  9, 23, 19,  1, 10, 16, 24, 27, 12, 12, 21,
         6, 10, 24,  8,  5, 20, 23, 27, 22, 26,  9, 12,  0,  9,  3,  2,  4,  6,
        24, 21,  9, 28, 13, 19, 22, 22,  8, 22, 16, 15,  1, 24, 17, 11, 24, 24,
         7, 11, 10,  6,  0, 14,  5,  4, 10,  2, 25, 26,  9, 23, 17, 16, 27, 26,
         3,  1,  8, 28, 13,  6, 19,  7, 12,  6,  6, 15, 20, 22,  5,  9, 20, 12,
         4,  8, 21,  0, 24, 21, 24, 12, 28, 15,  9, 15,  2, 12,  4,  9, 12, 27,
        22,  8,  2,  2, 23, 21,  7, 14, 13, 28, 24, 12, 26,  8, 23, 24,  6, 27,
        12, 20, 24, 10, 23,  4, 21,  7, 15,  6,  5, 16, 14, 22,  2, 10,  9,  5,
        25, 19, 11, 28, 17, 19,  1, 18,  0, 16,  3,  8,  7, 22, 15, 14,  2,  8,
         7,  8, 10, 25, 19, 12,  8, 16,  6, 13, 22, 17, 28,  5, 21, 27,  1, 22,
         9, 15, 12, 26,  7,  9,  1, 10, 25, 12, 23,  8,  0, 28, 13, 14, 22,  4,
        18, 16,  5, 15, 24, 14, 12, 28, 23, 15, 15, 27, 20, 11, 21,  3,  2, 28,
        20, 28, 28,  0, 19, 20,  9, 13,  8,  6,  3, 10,  2, 21,  2, 14, 14, 25,
        18, 16, 23, 11, 18, 15, 11,  3, 26, 27,  1, 11, 28, 14,  9, 13,  3,  8,
        11, 16, 21,  7, 19, 11, 22, 28,  4, 19, 10, 25, 27, 26, 28,  6, 23, 23,
         9, 15, 25, 24, 10, 17, 28, 26, 26, 23,  1, 26, 23, 11, 11, 19,  2, 17,
        23, 21, 23, 12, 19, 26, 10,  9, 17, 12, 16, 20, 18, 28, 22, 25, 12, 25,
        12,  9,  7,  4,  5, 15, 20, 21, 27, 10, 14,  1, 12, 15, 10, 28,  8,  3,
        13, 18,  6, 23, 14, 19, 24,  5, 25, 11,  6,  2,  4,  5, 20, 20,  5,  4,
        12,  2,  3, 27, 25, 21,  0, 21,  6,  5, 15, 12, 19,  9,  2, 24, 20, 26,
         9, 19, 12, 12,  9, 23, 18,  7, 10, 28, 23,  2, 17, 24, 24,  8, 11,  0,
        21, 15, 16,  1, 28, 22, 10, 16, 12, 24, 25,  8,  0, 17,  6, 28, 19,  9,
        28, 23, 12, 26, 21,  3, 18, 23, 10, 20,  5,  2, 16,  2,  8, 15, 20, 22,
        18,  2,  0, 11,  1,  7, 24,  2, 28, 12, 26, 11, 16,  5, 14,  6,  9, 21,
         4,  8,  1, 18, 11, 27, 17, 19, 13, 28, 24, 18, 15, 25, 12, 23, 13, 19,
         0,  7, 17,  8, 28, 28, 11,  2, 20, 26, 18,  3, 23, 12,  8, 23,  5, 14,
         1, 10, 21, 10, 22, 21,  2, 23, 13,  1,  0, 26,  9,  5, 17, 14, 13, 14,
        19,  2, 25, 12, 15, 19,  3, 23, 20, 27, 16, 24, 14,  5,  6, 25, 11, 26,
         7,  4,  5, 21,  4, 23, 16, 12, 19, 13, 22,  6, 11,  2, 25,  5, 12, 27,
        10, 12,  6, 11, 26, 17, 15, 11, 27,  6, 24,  5, 28,  8, 26,  3, 13, 19,
        14, 11, 26, 22,  6, 26, 24,  4, 28,  5,  5, 20, 16,  2, 12,  1, 19, 22,
        19, 21, 11, 20,  8, 17, 15, 18, 11, 22, 19,  9,  1, 12,  6, 22, 19, 24,
        14, 20, 21, 25, 12, 23,  7, 10, 13, 19,  6, 26,  3,  8, 17, 10, 19, 14,
        20, 15,  6,  5,  9, 12,  9,  1,  0, 19, 25,  4, 15,  6, 26, 17, 20,  2,
        26, 17, 14, 11, 12, 23,  5,  8,  5, 27, 10, 17, 15, 19, 21,  0, 13,  8,
        14, 16, 14, 23, 28,  6, 23,  7, 22,  9,  2, 13, 14,  1,  8, 18, 12,  4,
        17, 19, 16, 10, 15, 15, 19, 23, 20, 22, 25, 24, 14, 21, 23,  5, 14, 13,
         5, 20,  9, 15, 19,  6,  5, 11, 18, 11, 13,  1,  8, 19, 28, 10,  0,  5,
        14, 21, 17, 12,  3, 28,  4, 14,  5, 13,  5, 24, 11,  0, 13, 22,  7,  9,
        21, 28, 15, 23, 25, 20,  3,  2,  8, 28, 25, 23, 17, 10, 18, 12,  5, 28,
        18, 27,  9, 28, 26, 17,  1, 24, 25, 21, 25, 27,  2, 12, 22, 26, 23, 28,
         4, 14, 24,  1, 24, 12,  7, 20,  1, 20, 17, 24, 24, 28, 13,  7, 11, 14,
        20, 25,  3, 19, 10, 23, 20, 28,  2, 24, 21, 19, 23, 16,  8,  7, 23,  9,
        28,  1,  6, 12, 11, 24, 28, 28,  0, 18, 15, 22,  8, 12, 13, 17, 18, 26,
         9, 13,  1,  6, 17, 10,  5, 22,  9, 23, 16,  6, 26, 15, 23, 11, 27, 19,
        20, 28, 13,  0, 12, 15, 21,  0, 24, 26, 19,  5,  3,  8,  8,  9, 14, 12,
        26, 26, 14, 11,  1,  5,  0, 13, 18,  7, 23, 28,  2,  8, 15, 22, 10, 15,
        23, 21, 17, 14, 25, 12, 19, 16, 22, 20, 28, 12, 13, 13, 20,  8, 17,  4,
        21, 15, 13,  6, 10, 10,  1,  6,  5,  5, 28, 17, 14, 24, 25, 12, 25,  2,
        27,  9, 26, 19, 11, 25,  0, 25, 17, 23, 20,  6, 18, 12, 24,  1,  8, 13,
        21, 26, 25, 17, 25,  4,  9, 28,  2, 13, 19, 12,  1, 10, 25,  1, 23, 20,
        27, 14, 19,  3, 18,  8, 24, 15, 22,  1])
```

```python
len(y_pred_tensor)
```

```
1000
```

```python
# See if required packages are installed and if not, install them...
try:
  import torchmetrics, mlxtend
  print(f"mlxtend version: {mlxtend.__version__}")
  assert int(mlxtend.__version__.split(".")[1] >= 19, "mlxtend version should be 0.19.0 or higher")
except:
  !pip install torchmetrics -U mlxtend
  import torchmetrics, mlxtend
  print(f"mlxtend version: {mlxtend.__version__}")
```

```
mlxtend version: 0.23.4
Requirement already satisfied: torchmetrics in /usr/local/lib/python3.11/dist-packages (1.7.1)
Requirement already satisfied: mlxtend in /usr/local/lib/python3.11/dist-packages (0.23.4)
Requirement already satisfied: numpy>1.20.0 in /usr/local/lib/python3.11/dist-packages (from torchmetrics) (2.0.2)
Requirement already satisfied: packaging>17.1 in /usr/local/lib/python3.11/dist-packages (from torchmetrics) (24.2)
Requirement already satisfied: torch>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from torchmetrics) (2.6.0+cu124)
Requirement already satisfied: lightning-utilities>=0.8.0 in /usr/local/lib/python3.11/dist-packages (from torchmetrics) (0.
```

```
Requirement already satisfied: scipy>=1.2.1 in /usr/local/lib/python3.11/dist-packages (from mlxtend) (1.14.1)
Requirement already satisfied: pandas>=0.24.2 in /usr/local/lib/python3.11/dist-packages (from mlxtend) (2.2.2)
Requirement already satisfied: scikit-learn>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from mlxtend) (1.6.1)
Requirement already satisfied: matplotlib>=3.0.0 in /usr/local/lib/python3.11/dist-packages (from mlxtend) (3.10.0)
Requirement already satisfied: joblib>=0.13.2 in /usr/local/lib/python3.11/dist-packages (from mlxtend) (1.4.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from lightning-utilities>=0.8.0->torch
Requirement already satisfied: typing_extensions in /usr/local/lib/python3.11/dist-packages (from lightning-utilities>=0.8.0
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxtend)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxtend) (0.
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxtend
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxtend
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxtend) (11.1.
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxtend)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.0.0->mlxt
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.24.2->mlxtend) (2025.
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.24.2->mlxtend) (202
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.3.1->ml
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torchmetrics) (3.18.0
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torchmetrics) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torchmetrics) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torchmetrics) (2025.3.2
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->to
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->t
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->to
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0-
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0-
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torch
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->tor
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torchmetrics) (3
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->torchmetrics) (1
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch>=2.0
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib>=3
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch>=2.0.0->torchm
mlxtend version: 0.23.4
```

```python
import mlxtend
print(mlxtend.__version__)
```

```
0.23.4
```

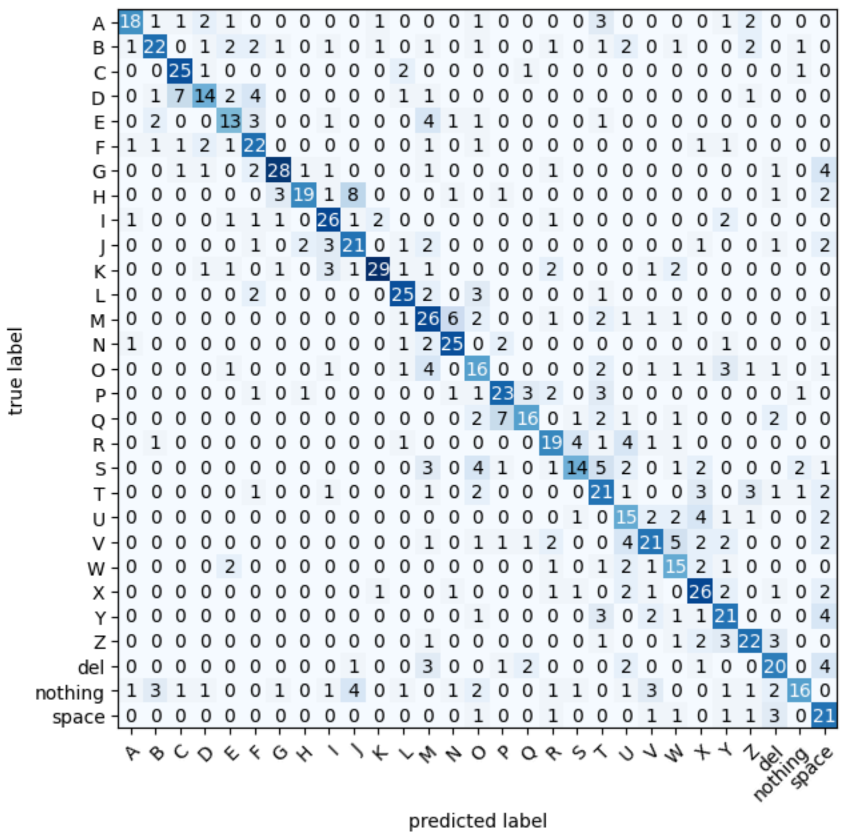
```python
class_names
```

```
['A',
 'B',
 'C',
 'D',
 'E',
 'F',
 'G',
 'H',
 'I',
 'J',
 'K',
 'L',
 'M',
 'N',
 'O',
 'P',
 'Q',
 'R',
 'S',
 'T',
 'U',
 'V',
 'W',
 'X',
 'Y',
 'Z',
 'del',
 'nothing',
 'space']
```

```python
y_pred_tensor[:10]
```

```
tensor([ 5, 19,  3, 24, 10, 13,  1, 17, 26,  3])
```

```
from torchmetrics import ConfusionMatrix
from mlxtend.plotting import plot_confusion_matrix

# 2. Setup confusion instance and compare predictions to targets
confmat = ConfusionMatrix(task="multiclass", num_classes=len(class_names))

# Here we are using a list comprehension that extracts the target labels from our test data,
# in other words, this is just iterating through our test dataset taking out only the target labels
# and putting them into a new list called targets in this case
targets = [label for _, label in test_data]

confmat_tensor = confmat(preds=y_pred_tensor,
                         target=torch.tensor(targets))
                         #target=[label for _, label in test_data])
                         #target=test_data.targets)

# 3. Plot the confusion matrix
fig, ax = plot_confusion_matrix(
    conf_mat=confmat_tensor.numpy(), # matplotlib likes working with numpy
    class_names=class_names,
    figsize=(10, 7)
)
```




## 9 - Save and Load Model

```
from pathlib import Path

# Create model dictory path
MODEL_PATH = Path("models")
MODEL_PATH.mkdir(parents=True,
                 exist_ok=True)

# Create model save
MODEL_NAME = "ASL_Model.pth"
MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

# Save the model state dict
```

```python
print(f"Saving model to: {MODEL_SAVE_PATH}")
torch.save(obj=model_2.state_dict(),
           f=MODEL_SAVE_PATH)
```

```
Saving model to: models/ASL_Model.pth
```

```python
image_shape = [3, 64, 64]
```

```python
# Create a new instance
torch.manual_seed(42)

loaded_model_2 = ASLModelV2(input_shape=3,
                        hidden_units=10,
                        output_shape=len(class_names))

# Load in the save state_dict()
loaded_model_2.load_state_dict(torch.load(f=MODEL_SAVE_PATH))

# Send the model to the target device
loaded_model_2.to(device)
```

```
ASLModelV2(
  (block_1): Sequential(
    (0): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (block_2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=2560, out_features=29, bias=True)
  )
)
```

```python
# Evaluate loaded model
torch.manual_seed(42)

loaded_model_2_results = eval_model(
    model=loaded_model_2,
    data_loader=test_dataloader,
    loss_fn=loss_fn,
    accuracy_fn=accuracy_fn
)

loaded_model_2_results
```

```
100%                                        32/32 [00:01<00:00, 27.12it/s]
{'model_name': 'ASLModelV2',
 'model_loss': 1.599514365196228,
 'model_acc': 74.4140625}
```

```python
model_2_results
```

```
{'model_name': 'ASLModelV2',
 'model_loss': 1.599514365196228,
 'model_acc': 74.4140625}
```

```python
# Check if model results are close to each other
torch.isclose(torch.tensor(model_2_results["model_loss"]),
              torch.tensor(loaded_model_2_results["model_loss"]),
              atol=1e-02)
```

```
tensor(True)
```

## 10 - Transfer Learning

After having started a baseline model, then builiding a better model with non-linearity, and then a CNN, we decided to perform transfer learning using a pre-trained ResNet18 model to find the most feasible model among the ones we have so far. We need a model that has the highest efficiency and accuracy and the lowest running time and loss.

We resized images to 200x200(same as originals) to hold the quality during training, converted them to tensors, and normalized pixel values to a common scale.

We loaded the dataset using ImageFolder, and increment the subset by using 10,000 samples (we need to find a balance between image size and number of samples to get the best performance). We decided to increment samples and size because ResNet18 allows us to perform faster computations and is used to process high amount of data.

We split the subset into 80% training and 20% testing like before.

Then we loaded the ResNet18 model and we decided to froze its convolutional layers so that it can retain the visual features it has already learned from ImageNet.

Just the final fully-connected (fc) layer is replaced and trained. We believe this reduced the training time and the risk of overfitting (that we have before) specially with this new dataset.

We used helper function like train_step and test_step to to track their results after each epoch.

After the training, we visualize the training and test loss curves, to see how well the model is learning. Then we printed the predicted and actual labels to see how the model performs after the training and to check if everything is okay.

To conclude we find transfer learning to be the most efficient among the models that we built, demonstraiting less run time and loss, and high accuracy.

Resnet18 documentation: https://debuggercafe.com/implementing-resnet18-in-pytorch-from-scratch/

## 10.1 - Create DataLoader for New Dataset

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
#from torchinfo import summary

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

torch.manual_seed(42)
random.seed(42)

# Path to your new dataset
new_data_dir = "/content/asl_alphabet_data/asl_alphabet_train/asl_alphabet_train"

"""transforms.Normalize([0.485, 0.456, 0.406],
                     [0.229, 0.224, 0.225])"""

# Define transforms
new_transform = transforms.Compose([
    transforms.Resize((200, 200)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])

# Create the dataset
new_dataset = datasets.ImageFolder(root=new_data_dir, transform=new_transform)

# Use a smaller subset (10000 samples)
subset_indices = random.sample(range(len(new_dataset)), 10000)
subset = Subset(new_dataset, subset_indices)

train_size = int(0.8 * len(subset))
test_size = len(subset) - train_size
train_dataset, test_dataset = random_split(subset, [train_size, test_size])


# Create the DataLoader
train_dataloader = DataLoader(train_dataset,
                              batch_size=32,
                              shuffle=True)
test_dataloader = DataLoader(test_dataset,
                             batch_size=32,
                             shuffle=False)
```

```
num_new_classes = len(new_dataset.classes)
```

## 10.2 - Freeze Layers and changing the output layer to suit our needs

```
base_model = models.resnet18(pretrained=True)

for param in base_model.parameters():
  param.requires_grad = False

base_model.fc = nn.Linear(base_model.fc.in_features, num_new_classes)
base_model = base_model.to(device)
```

```
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is depreca
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or
  warnings.warn(msg)
```

```
len(class_names)
```

```
29
```

## 10.3 - Train Model

```
# Define loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(base_model.fc.parameters(), lr=0.001)
```

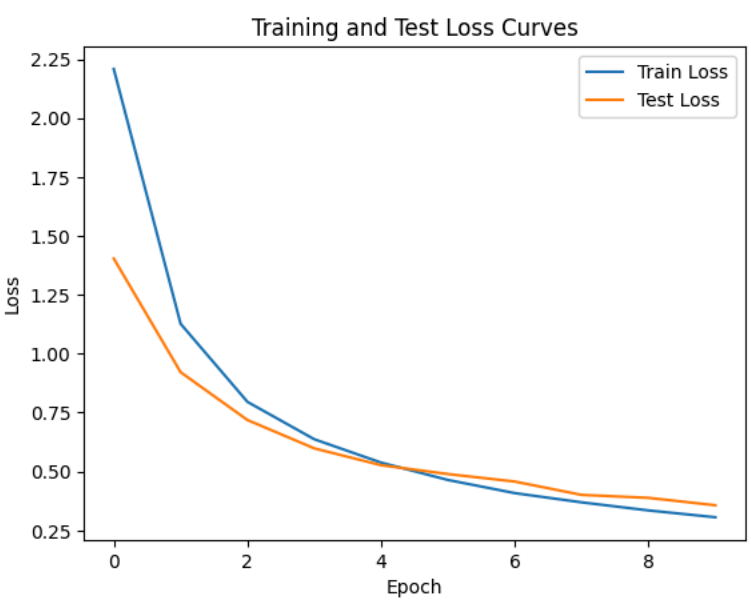
```
# Lists to store losses for plotting
train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []

# Training loop
epochs = 10
for epoch in range(epochs):
    print(f"Epoch: {epoch}\n---------")

    # Train on the new dataset and get train loss
    train_loss, train_acc = train_step(data_loader=train_dataloader,
                           model= base_model,
                           loss_fn=loss_fn,
                           optimizer=optimizer,
                           accuracy_fn=accuracy_fn,
                           device=device)
    train_losses.append(train_loss)  # Append train loss and accuracy to list
    train_accuracies.append(train_acc)

    # Evaluate on the test dataset and get test loss
    test_loss, test_acc = test_step(data_loader=test_dataloader,
                          model= base_model,
                          loss_fn=loss_fn,
                          accuracy_fn=accuracy_fn,
                          device=device)
    test_losses.append(test_loss)  # Append test loss and accuracy to list
    test_accuracies.append(test_acc)
```

```
Epoch: 0
---------
Train loss: 2.20810 | Train acc: 47.80%
Test loss: 1.40413 | Test acc: 73.71%

Epoch: 1
---------
Train loss: 1.12713 | Train acc: 78.04%
Test loss: 0.92178 | Test acc: 80.70%

Epoch: 2
---------
Train loss: 0.79570 | Train acc: 84.03%
```

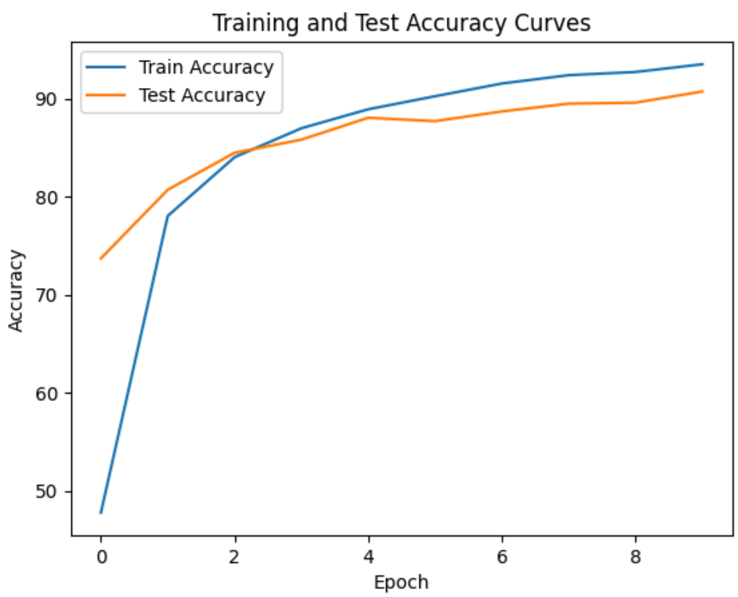
```
Test loss: 0.71868 | Test acc: 84.47%

Epoch: 3
---------
Train loss: 0.63709 | Train acc: 86.97%
Test loss: 0.59842 | Test acc: 85.81%

Epoch: 4
---------
Train loss: 0.53795 | Train acc: 88.91%
Test loss: 0.52690 | Test acc: 88.05%

Epoch: 5
---------
Train loss: 0.46431 | Train acc: 90.24%
Test loss: 0.48939 | Test acc: 87.70%

Epoch: 6
---------
Train loss: 0.40846 | Train acc: 91.54%
Test loss: 0.45753 | Test acc: 88.69%

Epoch: 7
---------
Train loss: 0.36935 | Train acc: 92.39%
Test loss: 0.40096 | Test acc: 89.48%

Epoch: 8
---------
Train loss: 0.33506 | Train acc: 92.71%
Test loss: 0.38843 | Test acc: 89.58%

Epoch: 9
---------
Train loss: 0.30606 | Train acc: 93.50%
Test loss: 0.35697 | Test acc: 90.72%
```

## 10.4 - Evaluate model by plotting loss curves

```python
import matplotlib.pyplot as plt

# Plot the losses
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.title('Training and Test Loss Curves')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



```
# Plot the accuracies
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.title('Training and Test Accuracy Curves')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```




## 10.5 - Transfer learning confusion matrix

```
# Import tqdm.auto
from tqdm.auto import tqdm

# 1. Make predictions with trained model
y_preds = []
model_2.eval()
with torch.inference_mode():
  for X, y in tqdm(test_dataloader, desc="Making predictions..."):
    # Send the data and targets to target device
    X, y = X.to(device), y.to(device)
    # Do the forward pass
    y_logit = base_model(X)
    # Turn predictions from logits -> prediction probabilities -> prediction labels
    y_pred = torch.softmax(y_logit.squeeze(), dim=0).argmax(dim=1)
    # Put prediction on CPU for evaluation
    y_preds.append(y_pred.cpu())

# Concatenate list of predictions into a tensor
y_pred_tensor = torch.cat(y_preds)
y_pred_tensor


from torchmetrics import ConfusionMatrix
from mlxtend.plotting import plot_confusion_matrix

# 2. Setup confusion instance and compare predictions to targets
confmat = ConfusionMatrix(task="multiclass", num_classes=len(class_names))

# Here we are using a list comprehension that extracts the target labels from our test data,
# in other words, this is just iterating through our test dataset taking out only the target label
# and putting them into a new list called targets in this case
targets = []
for _, label in test_dataloader: # iterate through the test_dataloader to get all target labels
  targets.extend(label.tolist()) # add the labels from the current batch to the targets list

confmat_tensor = confmat(preds=y_pred_tensor,
                         target=torch.tensor(targets))
                         #target=[label for _, label in test_data])
                         #target=test_data.targets)
```

```
# 3. Plot the confusion matrix
fig, ax = plot_confusion_matrix(
    conf_mat=confmat_tensor.numpy(), # matplotlib likes working with numpy
    class_names=class_names,
```