In [1]:
```
import torch
import torch.nn as nn
import copy
import torchvision
import torch.optim as optim
import torchvision.transforms as transforms
import sklearn
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
```

Exploring Deep Learning Through the Lense of Example Difficulty

Much of this homework is inspired by the following paper: [https://arxiv.org/abs/2106.09647 (https://arxiv.org/abs/2106.09647)](https://arxiv.org/abs/2106.09647)

Deep Learning Practioners have recognized that within the same task, particular examples in the test set can actually be harder to perform predictions on that others. Why is that? What kinds of things are easier to learn? We explore the notion of example difficulty, proposed by Baldock et. al. that will allow us to perform deeper investigations on the topic.

Defining of Prediction Depth

Consider a N-Layer neural network, with KNN Classifiers after each layer.

$K_L(x)$ is the classification of the KNN after layer L

We will say that a prediction is made at depth L if L is the minimum value such that $m > L$ implies $K_m(x) = K_N(x)$

Essentially, we make a prediction at depth L if after that layer, the classifications stay consistent.

Why Prediction Depth Matters

Prediction depth can be viewed as a proxy for how hard a particular training example is. In this notebook we will explore the relation to what appears to be qualitatively difficult and prediction depth.

Network Setup

We will first train a ResNet-18. Once trained, we will pass in all the training data once more to get the intermediate representations after each layer. We will use these representations to train a KNN at each layer to classify data. We will then use the trained KNN classifiers on the evaluation/test data to determine prediction depth and accuracy.

First Glance at the Data

Let's take a look at the data

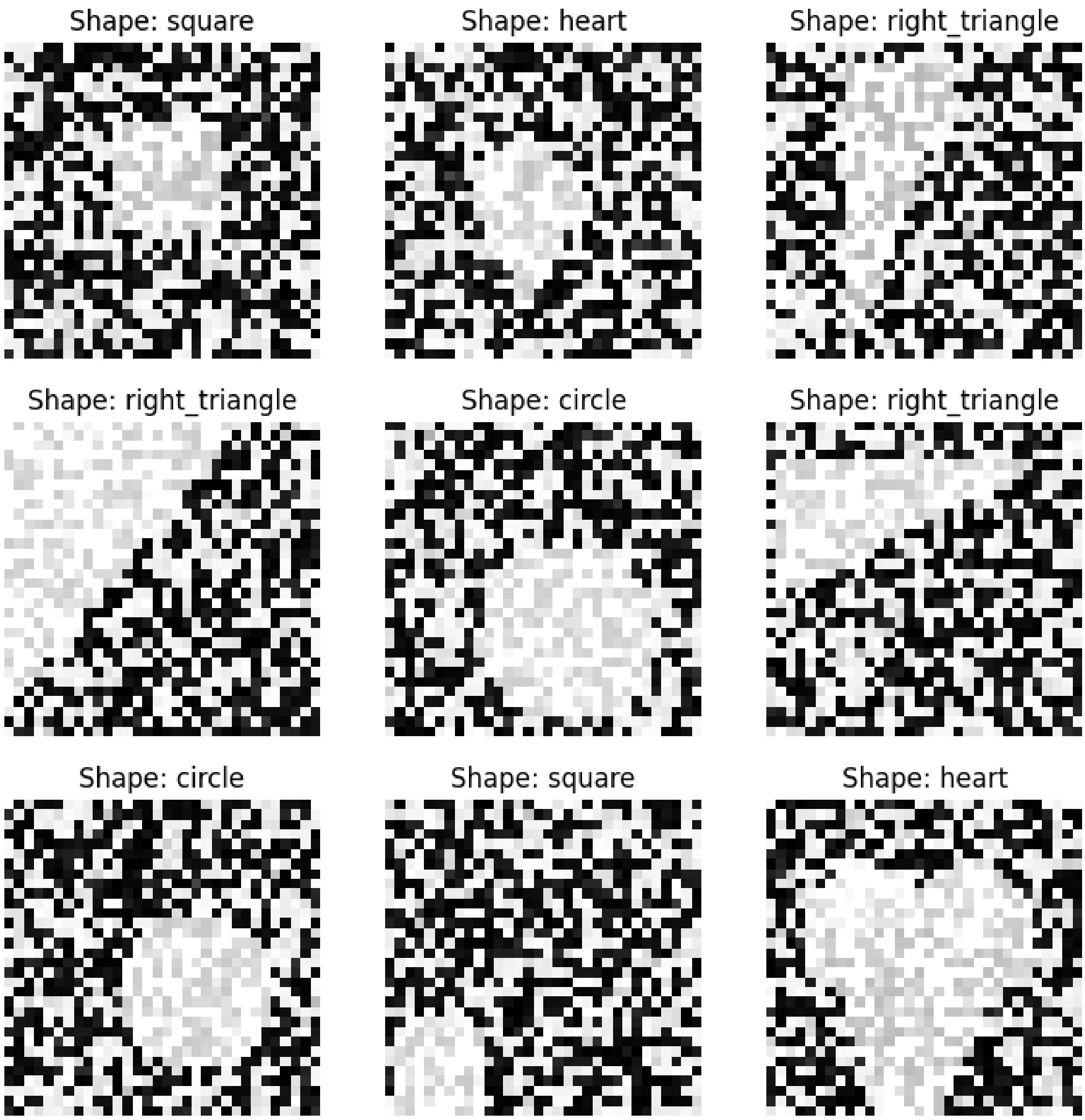
In [2]:
```
batch_size = 256
shapes = ['circle', 'square', 'rectangle', 'right_triangle', 'heart', 'ellipse']
```

Please download the data from the website and drag it into this folder.

We start with the standard dataloading pytorch definitions

In [3]:
```
data = np.load('data.npy', allow_pickle=True).item()
x_tensor = torch.FloatTensor(data['x'])
y_tensor = torch.LongTensor(data['y'])
dataset = torch.utils.data.TensorDataset(x_tensor, y_tensor)
trainloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, num_worker

test_data = np.load('test_data.npy', allow_pickle=True).item()
x_tensor = torch.FloatTensor(test_data['x'])
y_tensor = torch.LongTensor(test_data['y'])
test_dataset = torch.utils.data.TensorDataset(x_tensor, y_tensor)
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=1, num_workers=2,
```

In [4]:
```
random_indices = np.random.choice([i for i in range(6000)], 9, replace=False)

plt.figure(figsize=(9, 9))

for i, index in enumerate(random_indices, 1):
    x, y = test_data['x'][index], test_data['y'][index]

    plt.subplot(3, 3, i)  # 2 rows and 5 columns of subplots
    plt.imshow(x.reshape((32, 32, 1)), cmap='gray')
    plt.axis('off')  # Turn off axis numbers and ticks
    plt.title(f'Shape: {shapes[y]}')
```



Difficulty

What kind of properties do you think will make an example from this dataset difficult?

Training a ResNet

We begin by training a standard ResNet-18 to classify each example by its shape

In [5]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [6]:
```
transform = transforms.Compose([
    transforms.ToPILImage(),                          # Convert arrays to PIL images
    transforms.Grayscale(num_output_channels=3),      # Convert grayscale to RGB
    transforms.Resize((224, 224)),                    # Resize all images to 224x224
    transforms.ToTensor(),                         # Convert the images to PyTorch tens
])
from copy import deepcopy
resnet_dataset = deepcopy(dataset)
resnet_dataset.transform = transform

resnet_trainloader = torch.utils.data.DataLoader(resnet_dataset, batch_size=batch_si
x_tensor = torch.FloatTensor(test_data['x'])
y_tensor = torch.LongTensor(test_data['y'])
test_dataset = torch.utils.data.TensorDataset(x_tensor, y_tensor)
resnet_test_dataset = deepcopy(test_dataset)
resnet_test_dataset.transform = transform
resnet_testloader = torch.utils.data.DataLoader(resnet_test_dataset, batch_size=batc
```

In [7]:
```
resnet = torchvision.models.resnet18()
num_ftrs = resnet.fc.in_features
resnet.fc = torch.nn.Linear(num_ftrs, 6)
resnet = resnet.to(device)
```

In [8]:
```
criterion = nn.CrossEntropyLoss()
resnet_optimizer = optim.Adam(resnet.parameters(), lr=0.0001)
```

Processing math: 100%

In [9]:
```python
step = 0
resnet_losses = []
for epoch in tqdm(range(10)):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(resnet_trainloader, 0):
        step += 1
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = inputs, labels = data[0].to(device), data[1].to(device)

        inputs = inputs.unsqueeze(1)
        inputs = inputs.repeat(1, 3, 1, 1)
        inputs = inputs.to(device)

        # zero the parameter gradients
        resnet_optimizer.zero_grad()


        # forward + backward + optimize
        outputs = resnet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        resnet_optimizer.step()
        resnet_losses.append(loss.item())
        # print statistics
        running_loss += loss.item()
        if i % 50 == 49:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1}] loss: {running_loss / 20:.3f}')
            running_loss = 0.0



print('Finished Training')
```

```
  0%|          | 0/10 [00:00<?, ?it/s]

[1, 50] loss: 3.739
[1, 100] loss: 3.017

 10%|█         | 1/10 [00:02<00:21,  2.38s/it]

[2, 50] loss: 1.952
[2, 100] loss: 1.850

 20%|██        | 2/10 [00:03<00:14,  1.81s/it]

[3, 50] loss: 1.199
[3, 100] loss: 1.169

 30%|███       | 3/10 [00:05<00:11,  1.65s/it]

[4, 50] loss: 0.656
[4, 100] loss: 0.668

 40%|████      | 4/10 [00:06<00:09,  1.58s/it]

[5, 50] loss: 0.326
[5, 100] loss: 0.280

 50%|█████     | 5/10 [00:08<00:07,  1.55s/it]

[6, 50] loss: 0.133
[6, 100] loss: 0.108
```

```
 60%|██████    | 6/10 [00:09<00:06,  1.54s/it]

[7, 50] loss: 0.057
[7, 100] loss: 0.043

 70%|███████   | 7/10 [00:11<00:04,  1.55s/it]

[8, 50] loss: 0.025
[8, 100] loss: 0.019

 80%|████████  | 8/10 [00:12<00:03,  1.52s/it]

[9, 50] loss: 0.008
[9, 100] loss: 0.007

 90%|█████████ | 9/10 [00:14<00:01,  1.51s/it]

[10, 50] loss: 0.004
[10, 100] loss: 0.004

100%|██████████| 10/10 [00:15<00:00,  1.58s/it]

Finished Training
```

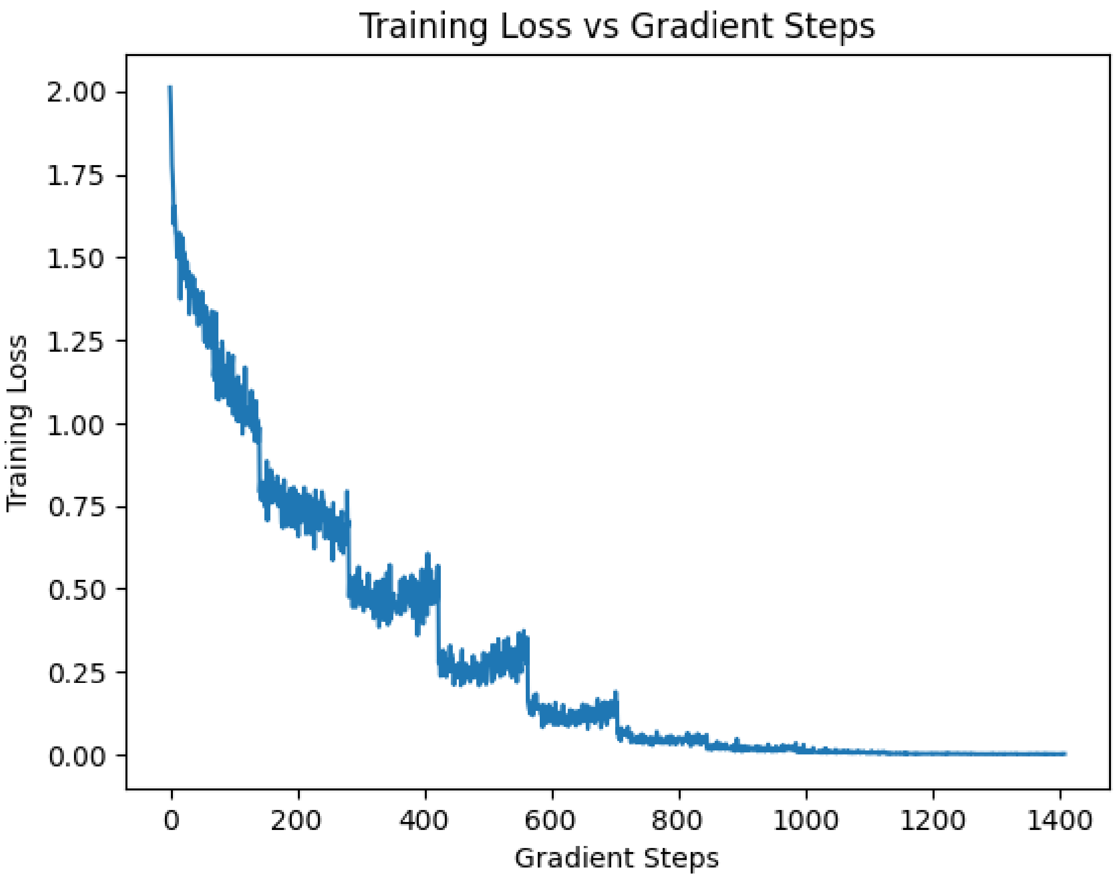
Finished Training

Now that we've finished training our ResNet, let's visualize the training curve to make sure we've trained to convergence. 10 epochs should be enough

Note: Analyzing example difficulty without training to convergence would be faulty.

In [10]:
```
plt.plot([i for i in range(len(resnet_losses))], resnet_losses)
plt.xlabel('Gradient Steps')
plt.ylabel('Training Loss')
plt.title('Training Loss vs Gradient Steps')
```

Out[10]: Text(0.5, 1.0, 'Training Loss vs Gradient Steps')



Evaluation Set

But did it actually learn? What is the evaluation accuracy?

Processing math: 100%

In [11]:
```
resnet.eval()

for epoch in range(1):  # loop over the dataset multiple times

    total_correct = 0
    with torch.no_grad():
        for i, data in tqdm(enumerate(resnet_test_dataset, 0)):
            # get the inputs; data is a list of [inputs, labels]

            inputs, labels = inputs, labels = data[0].to(device), data[1].to(device)

            inputs = inputs.unsqueeze(0).unsqueeze(0)
            inputs = inputs.repeat(1, 3, 1, 1)
            inputs = inputs.to(device)


            # forward + backward + optimize
            outputs = resnet(inputs)

            indices = torch.argmax(outputs, dim=1)

            total_correct += torch.sum(labels == indices)

print(total_correct)


print('Finished Training')
print(f'Accuracy: {total_correct/6000 * 100} %')
```

```
6000it [00:09, 619.13it/s]

tensor(4286, device='cuda:0')
Finished Training
Accuracy: 71.43333435058594 %
```

Capturing Activations

We will need to capture the activations to run KNN. We can do this in pytorch by attaching forward hooks. We need to this since we can't directly edit the model, as the code is abstracted away.

In [12]:
```
activations = dict()
resnet_labels = []
```

Processing math: 100%

In [13]:
```
def forward_hook(layer_num, activations):
    def hook(module, input, output):
        if layer_num + 1 not in activations:
            if layer_num == 0:
                activations[layer_num] = [input[0]]
            activations[layer_num + 1] = [output]
        else:
            if layer_num == 0:
                activations[layer_num].append(input[0])
            activations[layer_num + 1].append(output)
    return hook
```

In [14]:
```
layer_num = 0
handles = []
for layer in resnet.children():
    handles.append(layer.register_forward_hook(forward_hook(layer_num, activations))
    layer_num += 1
```

In [15]:
```

for epoch in tqdm(range(1)):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(resnet_trainloader, 0):
        step += 1
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = inputs, labels = data[0].to(device), data[1].to(device)

        inputs = inputs.unsqueeze(1)
        inputs = inputs.repeat(1, 3, 1, 1)
        inputs = inputs.to(device)
        resnet_labels.append(labels)

        # zero the parameter gradients
        with torch.no_grad():
        # forward + backward + optimize
            outputs = resnet(inputs)
```

```
100%|██████████| 1/1 [00:00<00:00,  1.87it/s]
```

Training KNN Classifiers and Removing Hooks

Let's train the classifiers with the activations that we've collected

In [16]:
```
for layer in activations:

    activations[layer] = torch.cat(activations[layer], dim=0)
    activations[layer] = torch.flatten(activations[layer], start_dim=1)

resnet_labels = torch.cat(resnet_labels, dim=0)

resnet_classifiers = [KNeighborsClassifier(n_neighbors=30) for _ in range(len(activ

for i, neigh in enumerate(resnet_classifiers):
    neigh.fit(activations[i].cpu().numpy(), resnet_labels.cpu().numpy())

for handle in handles:
    handle.remove()
```

Collecting Test Set Activations

Now we want to check the predictions of the test set examples. Using the activations and trained KNN's we can predict the output at each layer in the ResNet to determine things like prediction depth

In [17]:
```
test_activations = dict()
test_resnet_labels = []
layer_num = 0
for layer in resnet.children():
    layer.register_forward_hook(forward_hook(layer_num, test_activations))
    layer_num += 1


    test_resnet_labels = []
for epoch in tqdm(range(1)):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(resnet_testloader, 0):
        step += 1
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = inputs, labels = data[0].to(device), data[1].to(device)

        inputs = inputs.unsqueeze(1)
        inputs = inputs.repeat(1, 3, 1, 1)
        inputs = inputs.to(device)
        test_resnet_labels.append(labels)

        # zero the parameter gradients
        with torch.no_grad():
        # forward + backward + optimize
            outputs = resnet(inputs)

        if i == 0:
            correct = torch.argmax(outputs, dim=1) == labels
        else:
            correct = torch.cat((correct, torch.argmax(outputs, dim=1) == labels))
```

```
100%|█████████████| 1/1 [00:00<00:00,  3.37it/s]
```

In [18]:
```
for layer in test_activations:

    test_activations[layer] = torch.cat(test_activations[layer], dim=0)
    test_activations[layer] = torch.flatten(test_activations[layer], start_dim=1)

test_resnet_labels = torch.cat(test_resnet_labels, dim=0)
knn_outputs = [knn.predict(test_activations[i].cpu().numpy()) for i, knn in tqdm(en
```

```
1it [00:04,  4.83s/it]OpenBLAS Warning : Detect OpenMP Loop and this applicati
on may hang. Please rebuild the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
build the library with USE_OPENMP=1 option.
OpenBLAS Warning : Detect OpenMP Loop and this application may hang. Please re
```

Finding Prediction depths

We will need a function to find the prediction depth.

In [19]:
```
def find_constant_index(row):

    """
    Input: [Knn(L) for L in 1...N]
    Output: Prediction  depth
    """
    # Start from the end of the row
    value = row[-1]
    for i in range(len(row)-2, -1, -1): # iterate backwards
        if row[i] != value:
            return i+1
    return 0
```

Preparations for Analysis

We need a few things before we conduct some analysis

Predictions[i][j] = a numpy array containing the knn outputs of data point i at layer j

indices[i] = prediction depth of data point i

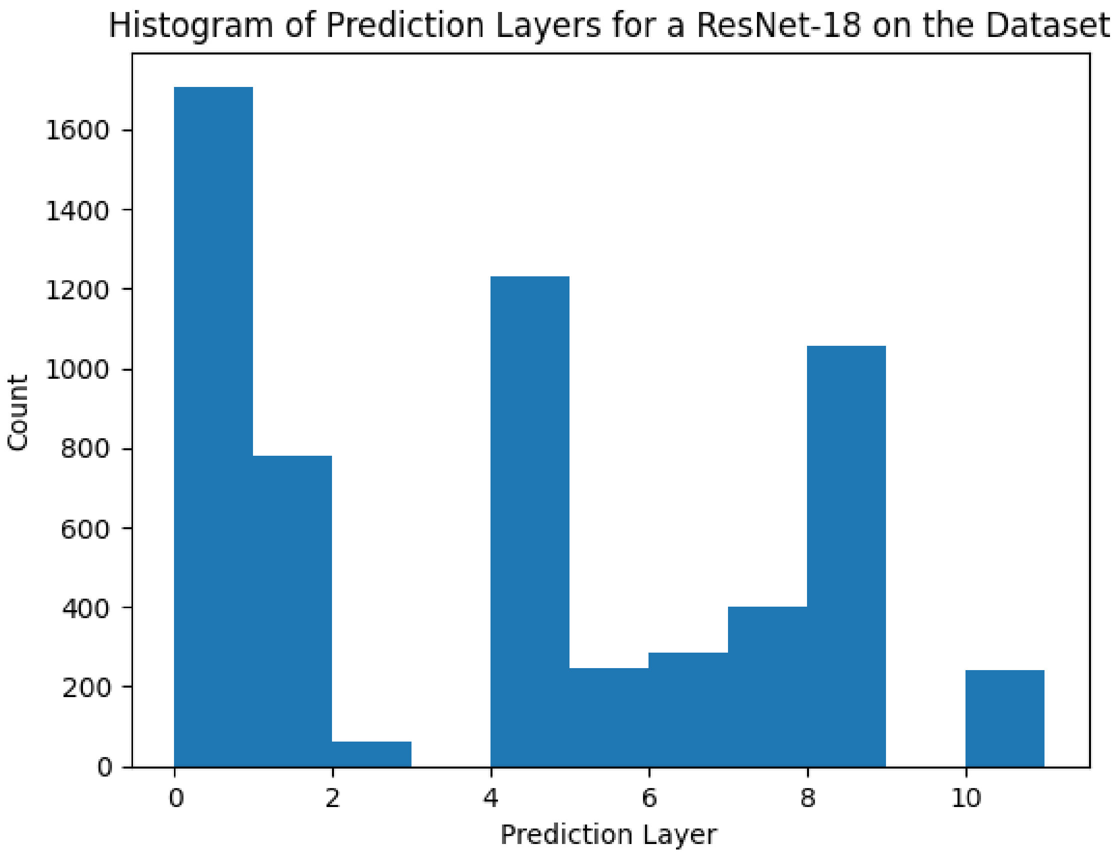
Processing math: 100%

```python
In [20]: predictions = np.array(knn_outputs)
indices = np.apply_along_axis(find_constant_index, axis=0, arr=predictions)
correct = correct.cpu().numpy()
prediction_layer_list = []
for num in range(11):  # Numbers 0-9
    temp_indices = np.where(indices == num)[0]
    prediction_layer_list.append(temp_indices.tolist())
total_accuracy_list = {}
for i, layer in enumerate(prediction_layer_list):
    if layer != []:
        total_accuracy_list[i] = (np.sum(correct[layer])/len(layer), len(layer)/600
    else:
        total_accuracy_list[i] = None
```

Visualizing the Histogram of Prediction Layers

The below visualization shows how many of each data point had prediction layer 0, for instance. If there were 500 examples that had prediction layer 1, this means that the KNN outputs do not change after layer 1 for 500 images. This could be interpreted as there was enough information at layer 1 to determine the class of the image with high confidence, and that the extra computation of the resnet was not necessary

```python
In [24]: plt.hist(indices, bins=[i for i in range(12)], weights=[1 for _ in range(6000)])
plt.xlabel('Prediction Layer')
plt.ylabel('Count')
plt.title('Histogram of Prediction Layers for a ResNet-18 on the Dataset')
plt.show()
```

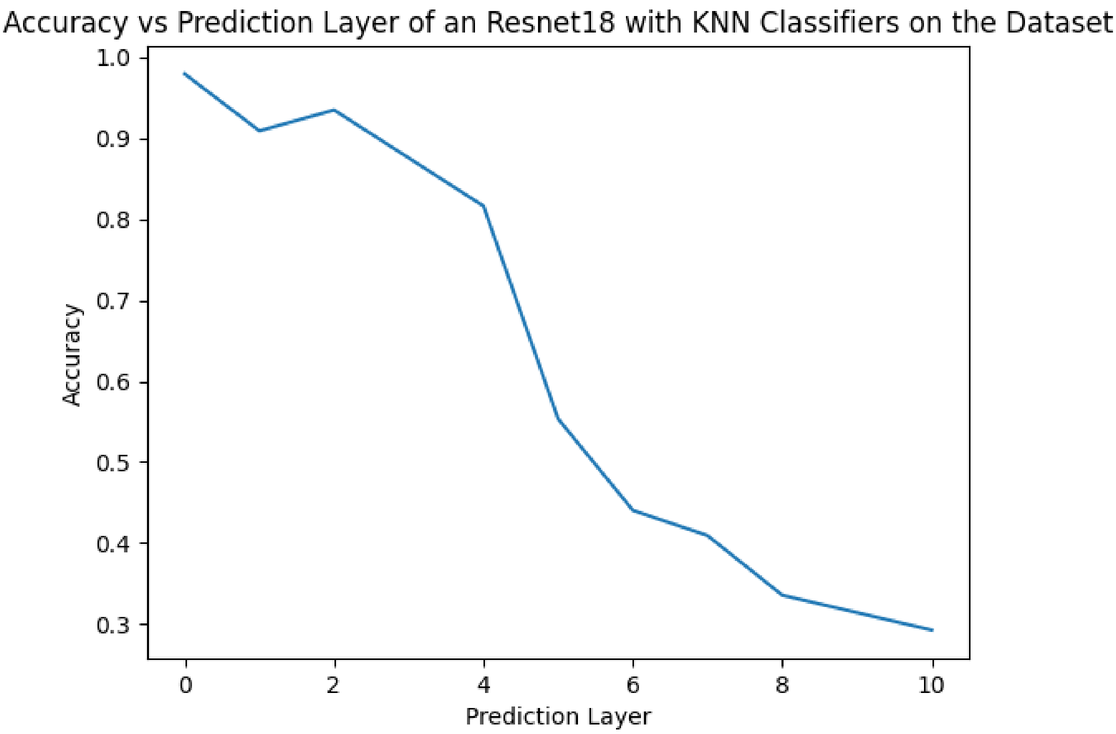


Processing math: 100%

Visualizing the Output Accuracy vs Prediction Layer

The below visualization shows the average accuracy of ResNet classification on points that exited at layer L. Notice that test examples that had lower prediction layer generally had higher accuracy from the ResNet. Note that the accuracy is from the predictions at the end of the ResNet, not the KNN classifiers. Prediction layer is still determined by the outputs of the KNN classifiers

In [25]:
```
plt.plot([i for i in range(11) if total_accuracy_list[i] is not None], [ total_acc
plt.xlabel('Prediction Layer')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Prediction Layer of an Resnet18 with KNN Classifiers on the D
plt.show()
```



Visualizing Easy and Hard Examples

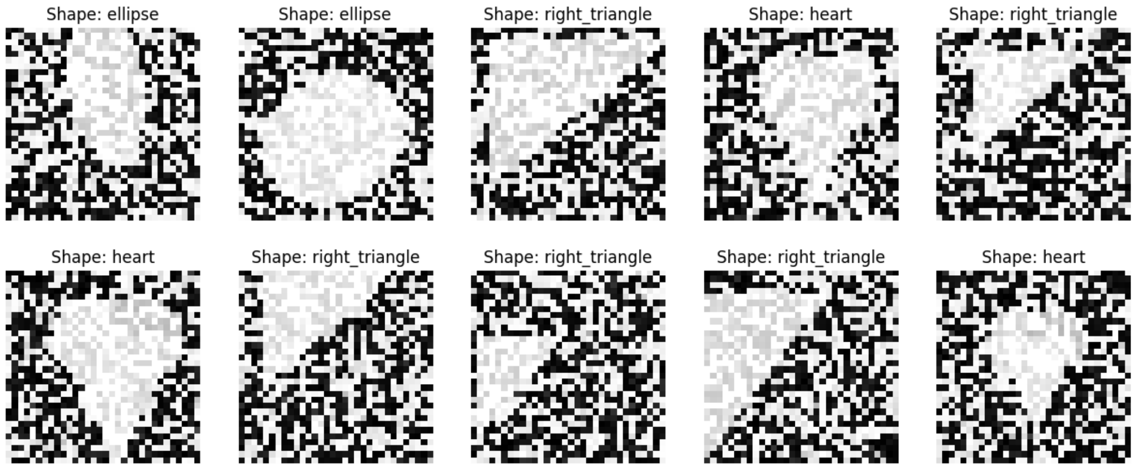
Let's try to find some patterns in what might make an easy example different than a hard example

In [26]:
```
easiest_examples = prediction_layer_list[0]
hardest_examples = prediction_layer_list[10]
```

Processing math: 100%

Easy Examples

In [27]:
```
from itertools import islice

random_indices = np.random.choice(easiest_examples, 10, replace=False)

plt.figure(figsize=(15, 6))

for i, index in enumerate(random_indices, 1):
    x, y = test_data['x'][index], test_data['y'][index]

    plt.subplot(2, 5, i)  # 2 rows and 5 columns of subplots
    plt.imshow(x.reshape((32, 32, 1)), cmap='gray')
    plt.axis('off')  # Turn off axis numbers and ticks
    plt.title(f'Shape: {shapes[y]}')
```



Processing math: 100%

Hard Examples

In [28]:
```
random_indices = np.random.choice(hardest_examples, 10, replace=False)

plt.figure(figsize=(15, 6))

for i, index in enumerate(random_indices, 1):
    x, y = test_data['x'][index], test_data['y'][index]

    plt.subplot(2, 5, i)  # 2 rows and 5 columns of subplots
    plt.imshow(x.reshape((32, 32, 1)), cmap='gray')
    plt.axis('off')  # Turn off axis numbers and ticks
    plt.title(f'Shape: {shapes[y]}')
```

Can you spot a difference

What would make these hard vs easy?

In [29]:
```
prediction_shapes = []
for layer in prediction_layer_list:
    prediction_shapes.append([])
    for index in layer:
        prediction_shapes[-1].append(test_data['y'][index])
```

What kinds of Shapes Exit at each Layer?

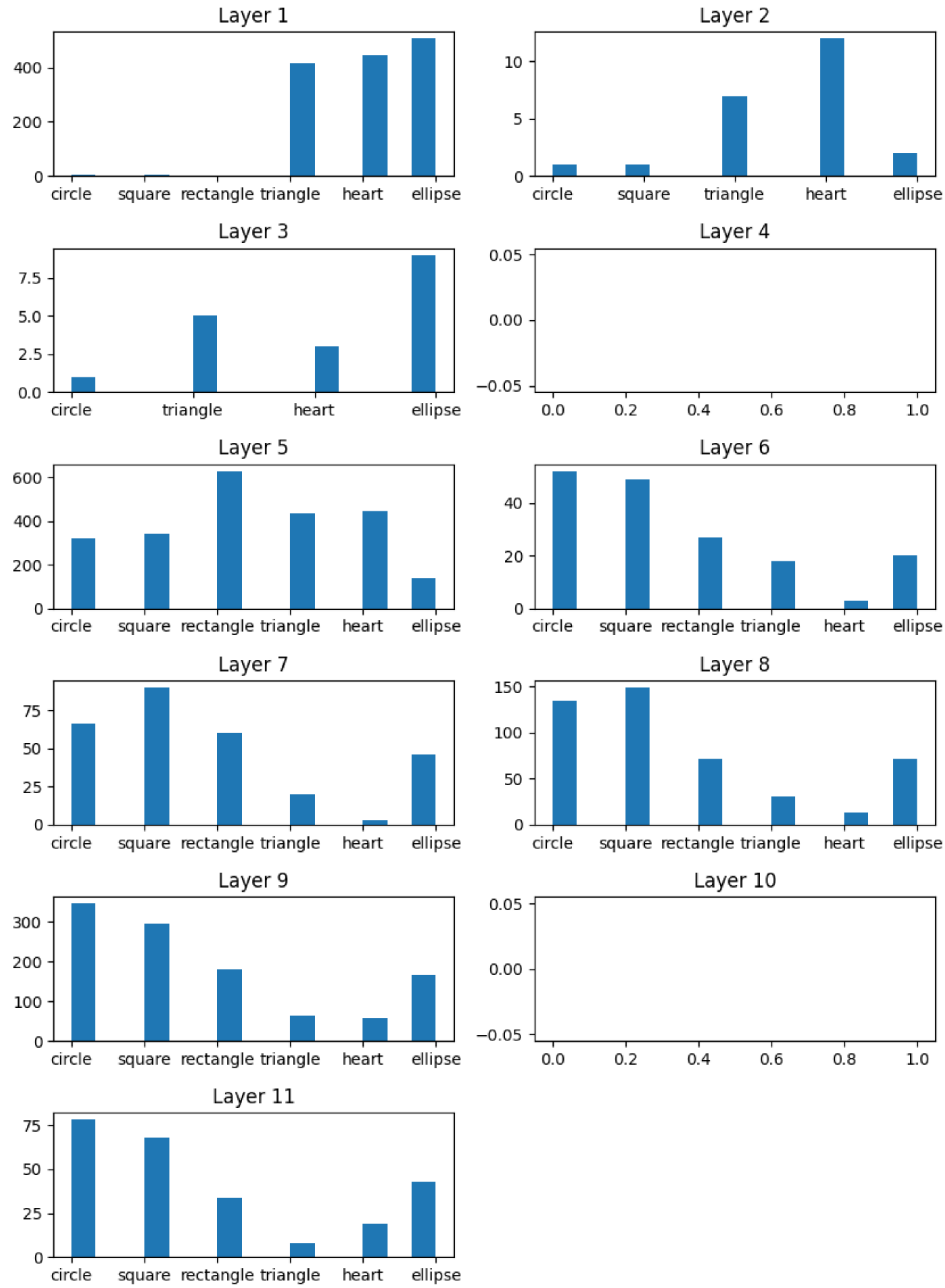
At each prediction layer, there may be different classes of shapes that are more common to appear. The following visualization shows at each layer, what is the distribution and count of the classes of shapes that will have prediction layer L

Processing math: 100%

In [102]:

```
import matplotlib.pyplot as plt

data = prediction_shapes

# Adjust to 6 rows and 2 columns for 11 plots + 1 empty
fig, axs = plt.subplots(6, 2, figsize=(10, 14))

for i, ax in enumerate(axs.flatten()):
    if i < len(data):  # Check if current index is within data's length
        hist = [shapes[j] for j in data[i]]
        for j in range(len(hist)):
            if hist[j] == 'right_triangle':
                hist[j] = 'triangle'
        ax.hist(hist, bins=15)
        ax.set_title(f'Layer {i}')
    else:
        ax.axis('off')  # Turn off the axis for the last empty plot

plt.subplots_adjust(hspace=0.5)
plt.show()
```

Processing math: 100%

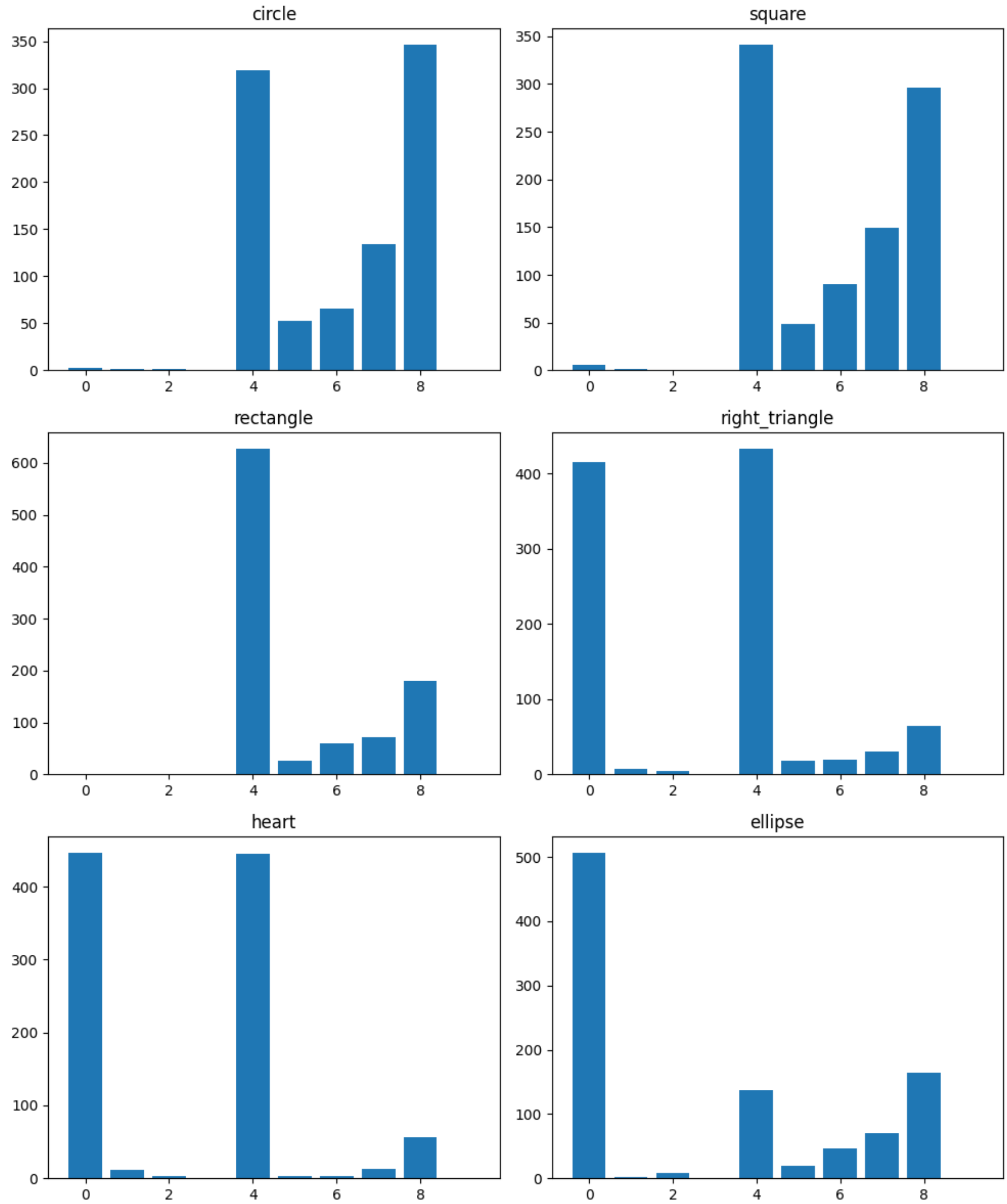


What is the empirical prediction layer distribution for each Shape?

Each shape may have a different distribution of layers that they exit on. For instance, one might think that triangles are harder to classify, and therefore more of the distribution mass would be towards the later layers. We aim to show, for each shape, the distribution of what prediction layers the shape generally tended to

In [103]:
```
frequency = []
data = [np.array(d) for d in data]
for i in range(6):
    frequency.append(dict())
    for j in range(10):
        frequency[-1][j] = np.sum(data[j] == i)
```

In [104]:
```
fig, axs = plt.subplots(3, 2, figsize=(10, 12))

for i, ax in enumerate(axs.flatten()):
    categories, counts = zip(*frequency[i].items())
    ax.bar(categories, counts)
    ax.set_title(shapes[i])

plt.tight_layout()
plt.show()
```



Processing math: 100%

Examining Layers

In [17]:
```
print("Input Layer")
print('________________________--')
for child in resnet.children():
    print(child)
    print('________________________--')
```

Processing math: 100%

```
Input Layer
--------------------------------------------------------------
Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
--------------------------------------------------------------
BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
--------------------------------------------------------------
ReLU(inplace=True)
--------------------------------------------------------------
MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
--------------------------------------------------------------
Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bi
as=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bi
as=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bi
as=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bi
as=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_st
ats=True)
  )
)
--------------------------------------------------------------
Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), b
ias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_s
```

```
tats=True)
  )
)
________________________--
Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
  )
)
________________________--
Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
```

```
  )
)
__________________________________--
AdaptiveAvgPool2d(output_size=(1, 1))
__________________________________--
Linear(in_features=512, out_features=6, bias=True)
__________________________________--
```

Patterns

What kinds of patterns do you notice? Based on the composition of the layers, does it make sense?

Concluding Thoughts

From what you witnessed in this homework, what can you say about example difficulty? How can we come up with better metrics of example difficulty? Why does it even matter? What are some possible applications of this line of work? In the next section of the homework, we will answer some of these questions.