TRANSFER LEARNING

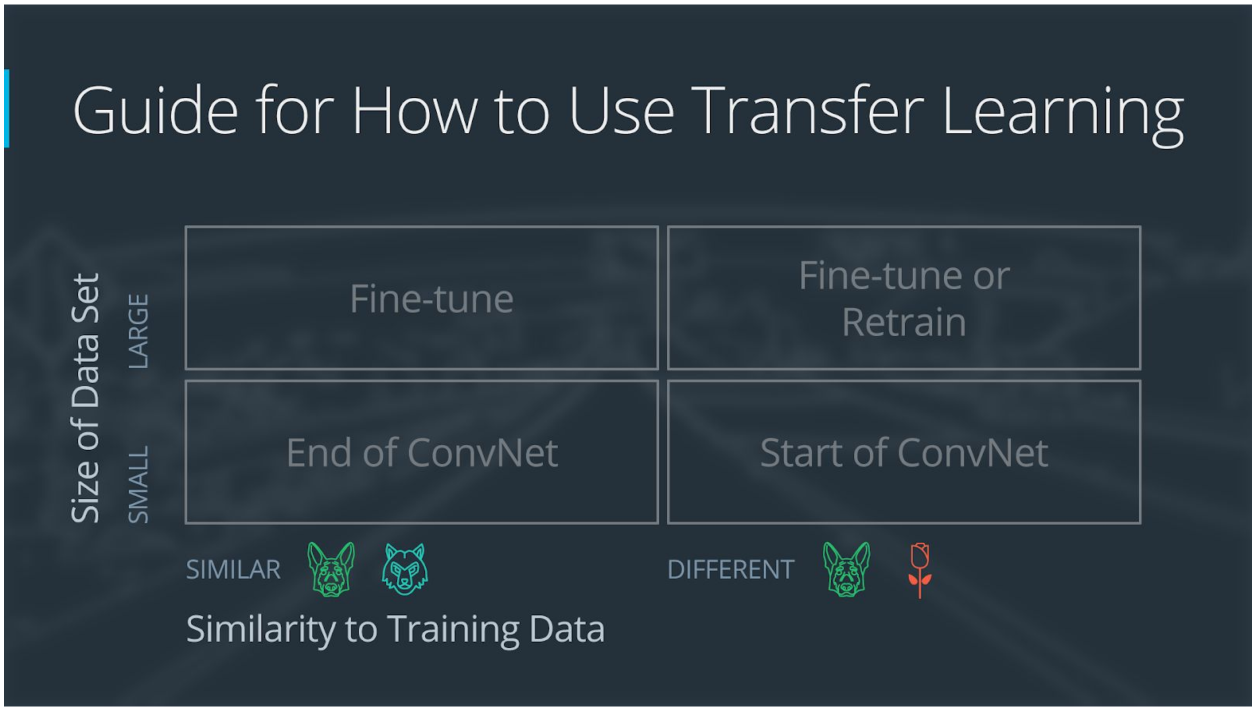
Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

Depending on both:

- the size of the new data set, and
- the similarity of the new data set to the original data set

the approach for using transfer learning will be different. There are four main cases:

1. new data set is small, new data is similar to original training data
2. new data set is small, new data is different from original training data
3. new data set is large, new data is similar to original training data
4. new data set is large, new data is different from original training data



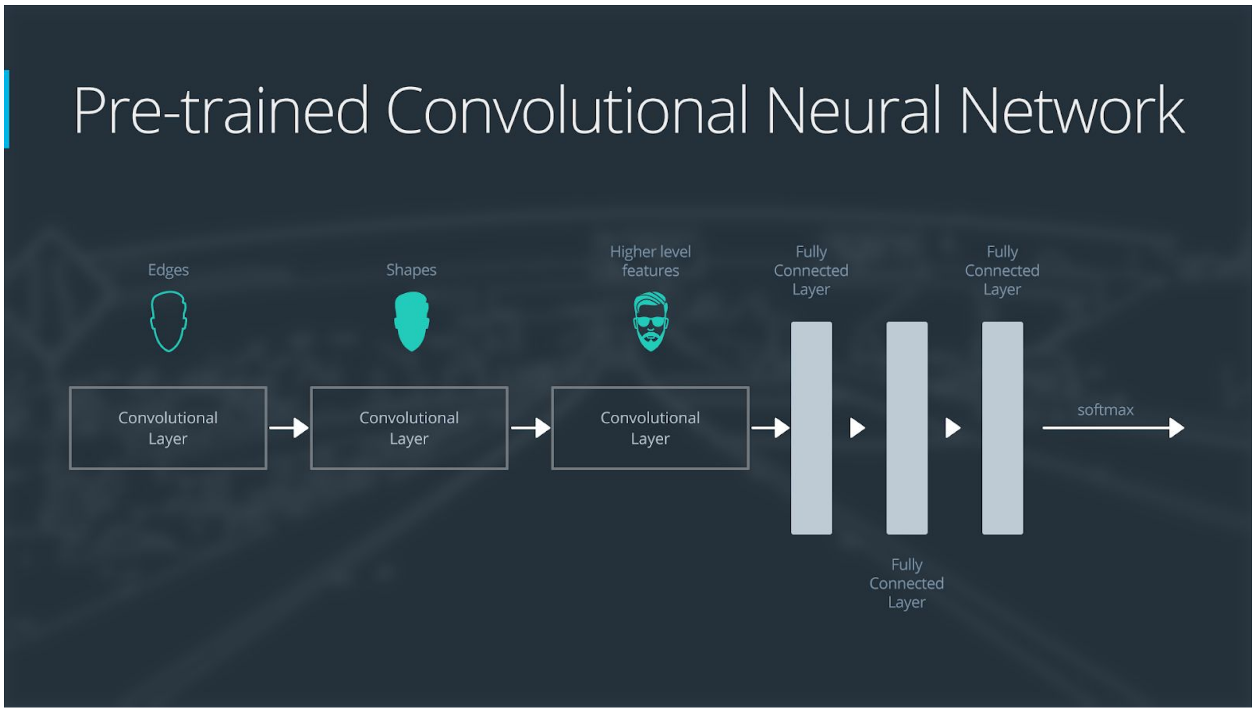
A large data set might have one million images. A small data could have two-thousand images. The dividing line between a large data set and small data set is somewhat subjective. Overfitting is a concern when using transfer learning with a small data set.

Images of dogs and images of wolves would be considered similar; the images would share common characteristics. A data set of flower images would be different from a data set of dog images.

Each of the four transfer learning cases has its own approach. In the following sections, we will look at each case one by one.

Demonstration Network

To explain how each situation works, we will start with a generic pre-trained convolutional neural network and explain how to adjust the network for each case. Our example network contains three convolutional layers and three fully connected layers:



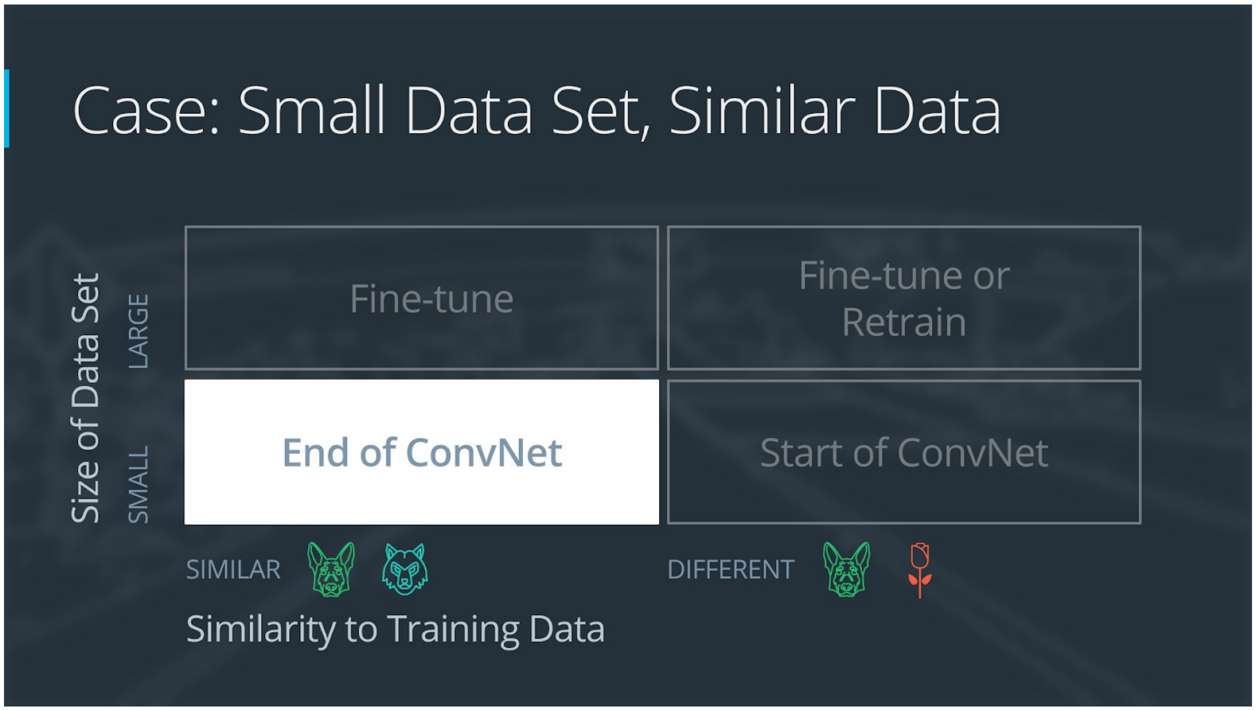
Pre-trained Convolutional Neural Network

Here is an generalized overview of what the convolutional neural network does:

- the first layer will detect edges in the image
- the second layer will detect shapes
- the third convolutional layer detects higher level features

Each transfer learning case will use the pre-trained convolutional neural network in a different way.

Case 1: Small Data Set, Similar Data



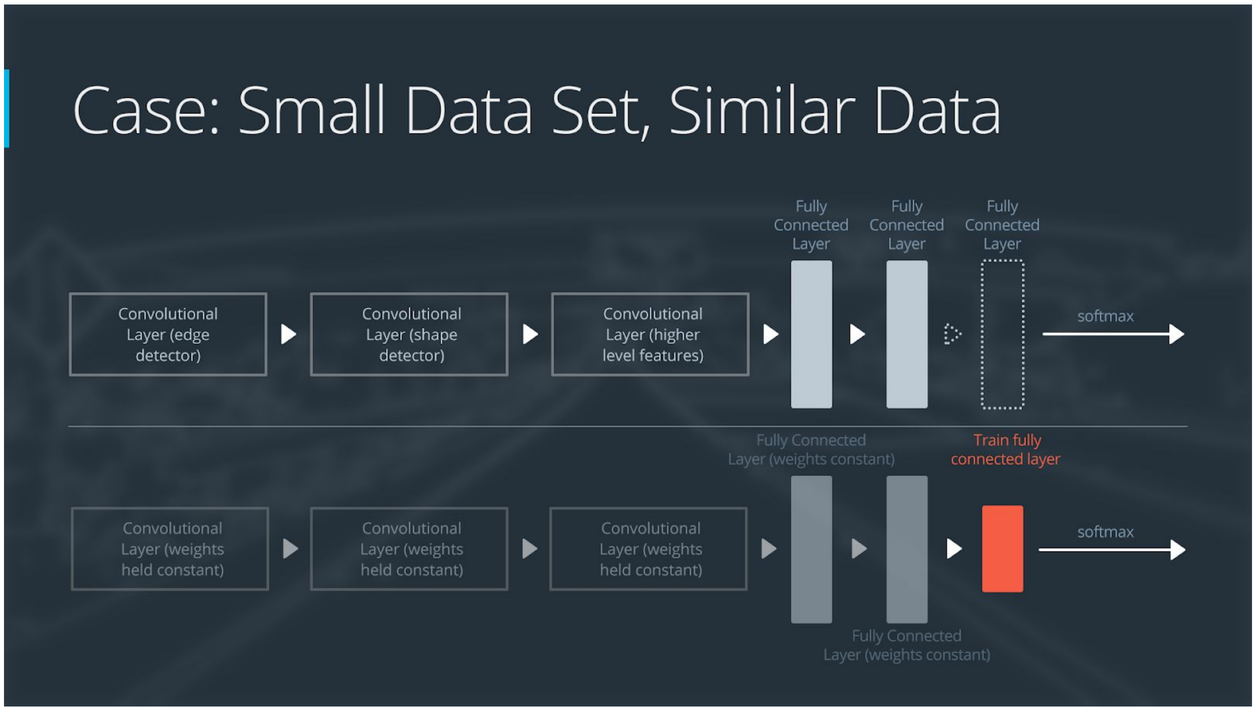
If the new data set is small and similar to the original training data:

- slice off the end of the neural network
- add a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

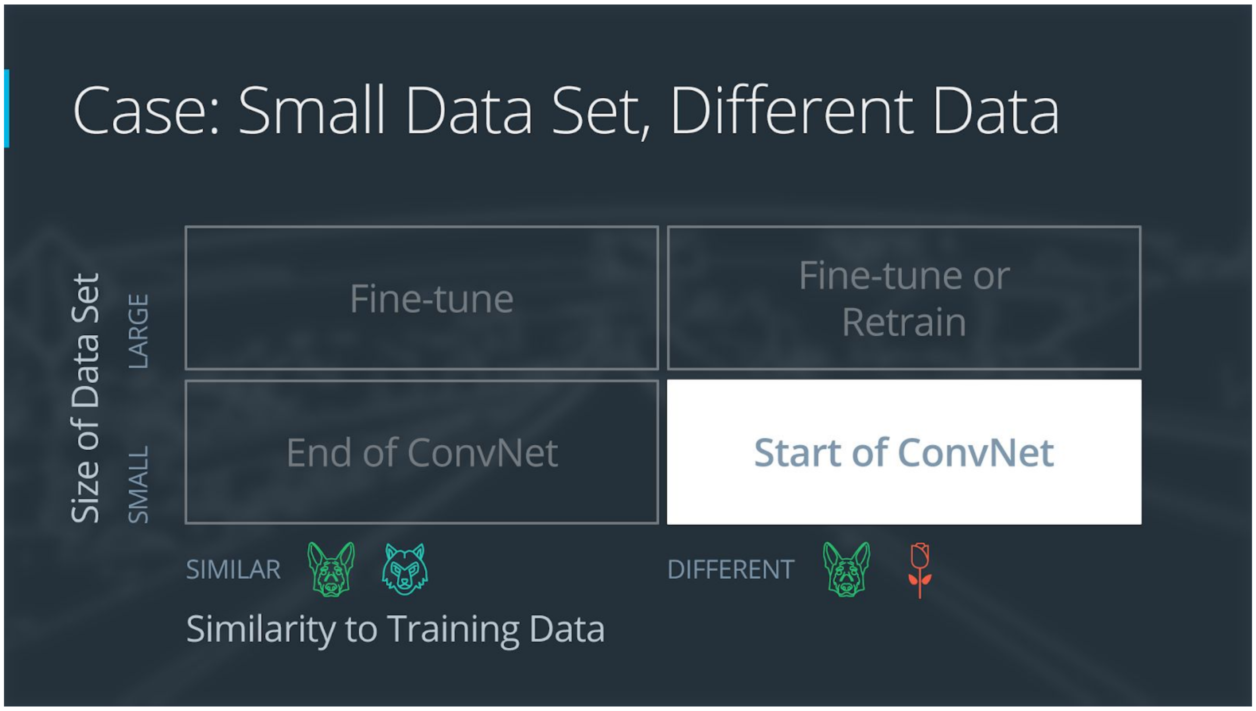
To avoid overfitting on the small data set, the weights of the original network will be held constant rather than re-training the weights.

Since the data sets are similar, images from each data set will have similar higher level features. Therefore most or all of the pre-trained neural network layers already contain relevant information about the new data set and should be kept.

Here's how to visualize this approach:


Case: Small Data Set, Similar Data
Convolutional Layer (edge detector)
Convolutional Layer (shape detector)
Convolutional Layer (higher level features)
Fully Connected Layer
Fully Connected Layer
Fully Connected Layer
softmax
Fully Connected Layer (weights constant)
Train fully connected layer
Convolutional Layer (weights held constant)
Convolutional Layer (weights held constant)
Convolutional Layer (weights held constant)
softmax
Fully Connected Layer (weights constant)

Case 2: Small Data Set, Different Data



Case 2: Small Data Set, Different Data

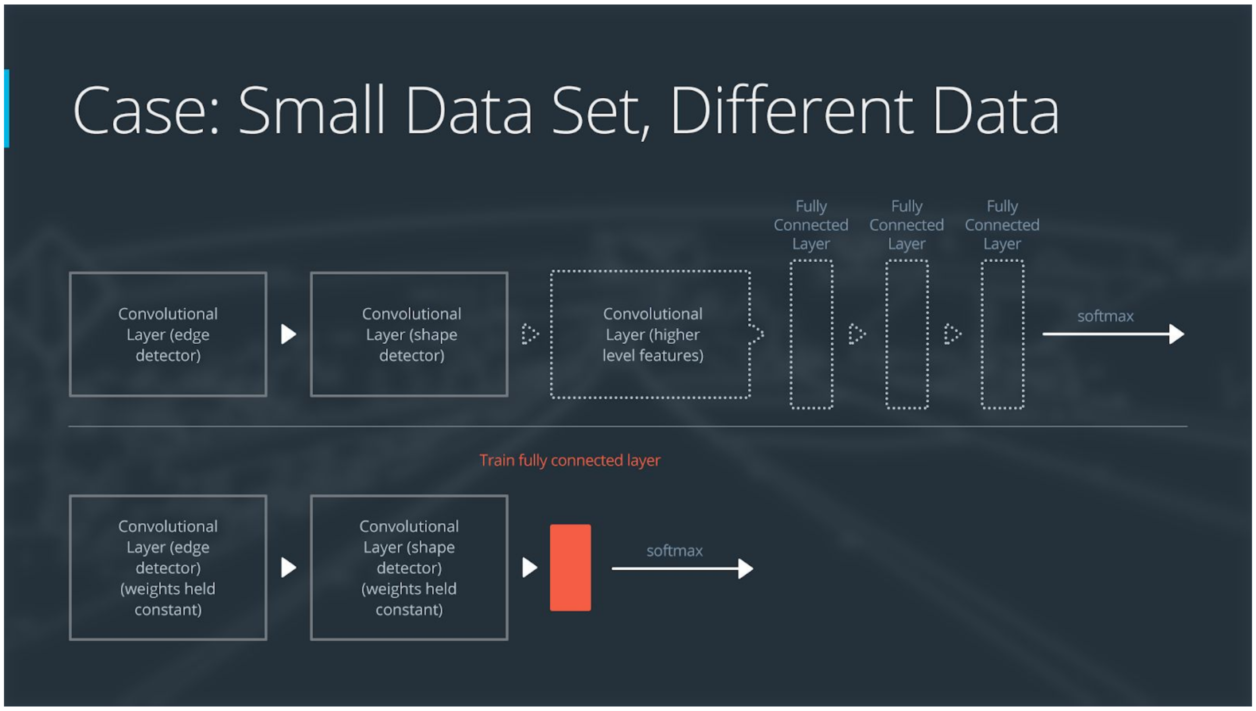
If the new data set is small and different from the original training data:

- slice off most of the pre-trained layers near the beginning of the network
- add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

Because the data set is small, overfitting is still a concern. To combat overfitting, the weights of the original neural network will be held constant, like in the first case.

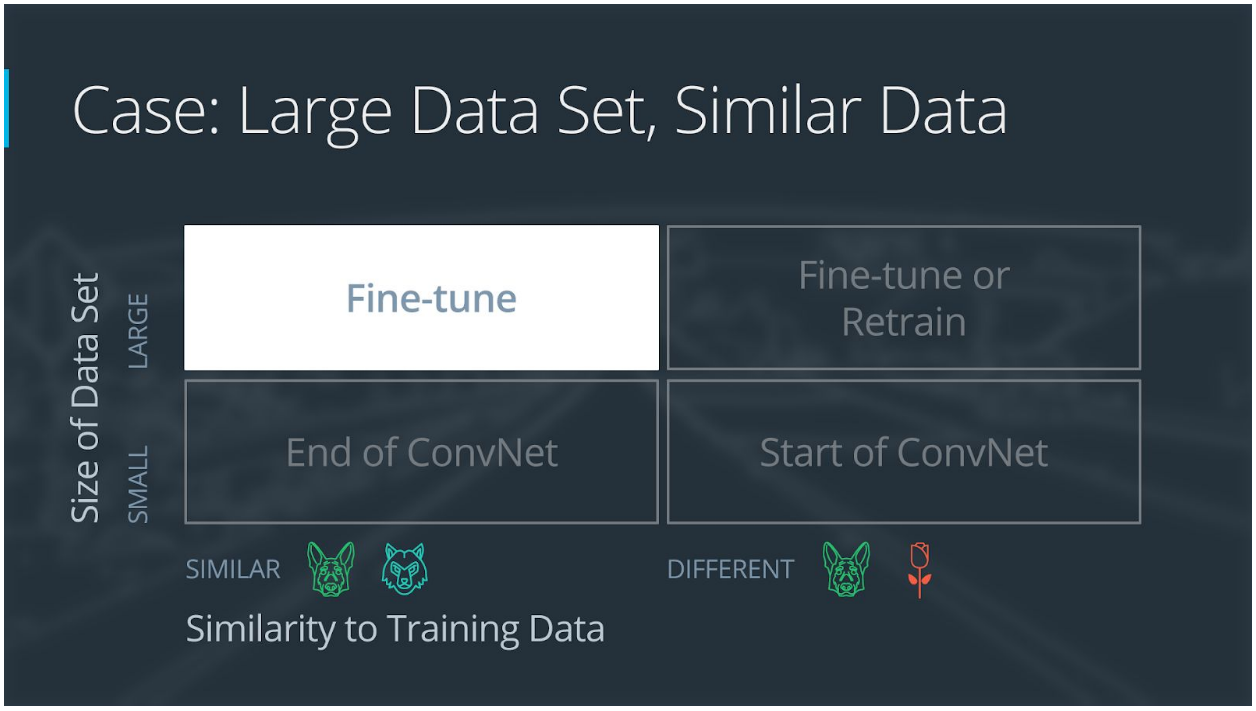
But the original training set and the new data set do not share higher level features. In this case, the new network will only use the layers containing lower level features.

Here is how to visualize this approach:


Case: Small Data Set, Different Data
Fully Connected Layer
Fully Connected Layer
Fully Connected Layer
Convolutional Layer (edge detector)
Convolutional Layer (shape detector)
Convolutional Layer (higher level features)
softmax
Train fully connected layer
Convolutional Layer (edge detector) (weights held constant)
Convolutional Layer (shape detector) (weights held constant)
softmax

Neural Network with Small Data Set, Different Data

Case 3: Large Data Set, Similar Data



Case 3: Large Data Set, Similar Data

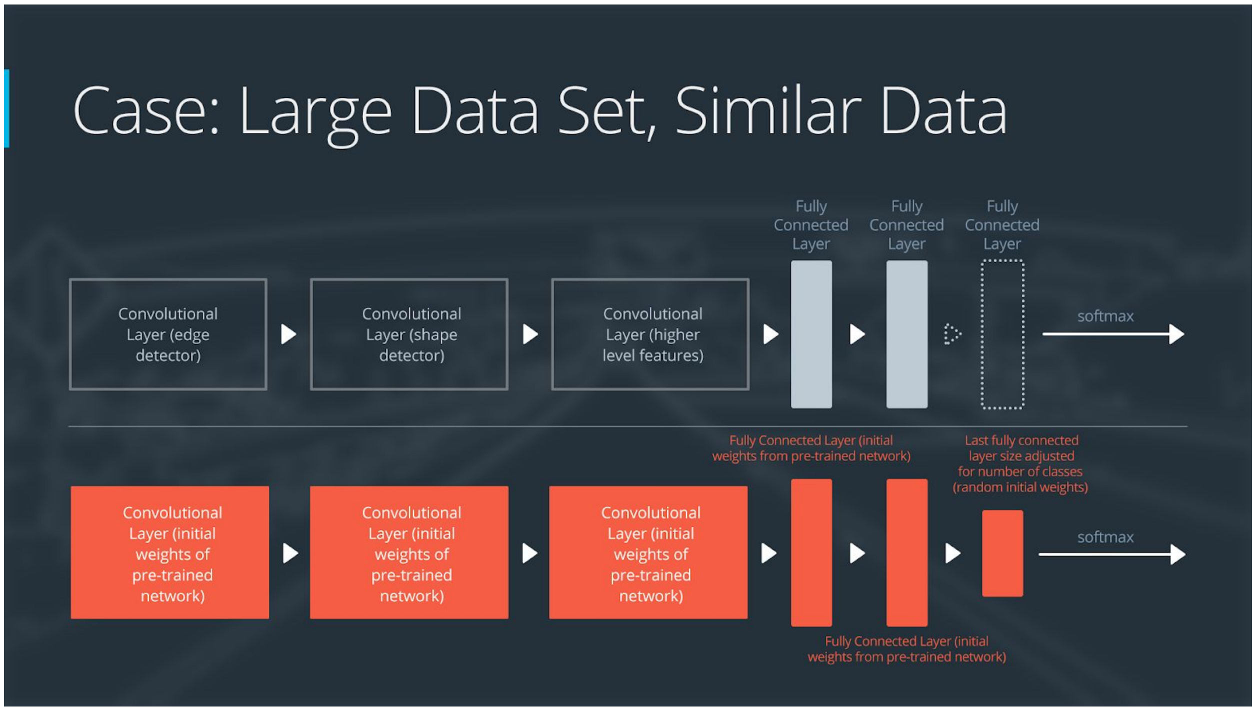
If the new data set is large and similar to the original training data:

- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- randomly initialize the weights in the new fully connected layer
- initialize the rest of the weights using the pre-trained weights
- re-train the entire neural network

Overfitting is not as much of a concern when training on a large data set; therefore, you can re-train all of the weights.

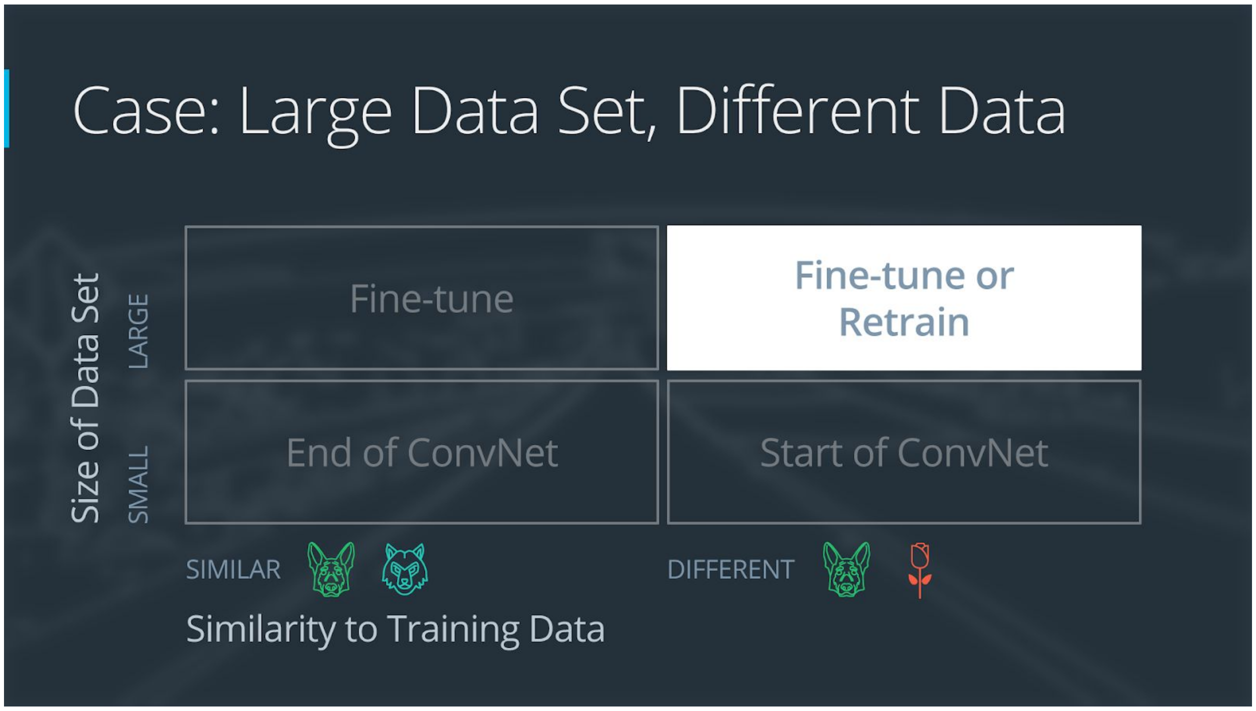
Because the original training set and the new data set share higher level features, the entire neural network is used as well.

Here is how to visualize this approach:


Case: Large Data Set, Similar Data
Convolutional Layer (edge detector)
Convolutional Layer (shape detector)
Convolutional Layer (higher level features)
Fully Connected Layer
Fully Connected Layer
Fully Connected Layer
softmax
Fully Connected Layer (initial weights from pre-trained network)
Last fully connected layer size adjusted for number of classes (random initial weights)
Convolutional Layer (initial weights of pre-trained network)
Convolutional Layer (initial weights of pre-trained network)
Convolutional Layer (initial weights of pre-trained network)
softmax
Fully Connected Layer (initial weights from pre-trained network)

Neural Network with Large Data Set, Similar Data

Case 4: Large Data Set, Different Data



Case 4: Large Data Set, Different Data

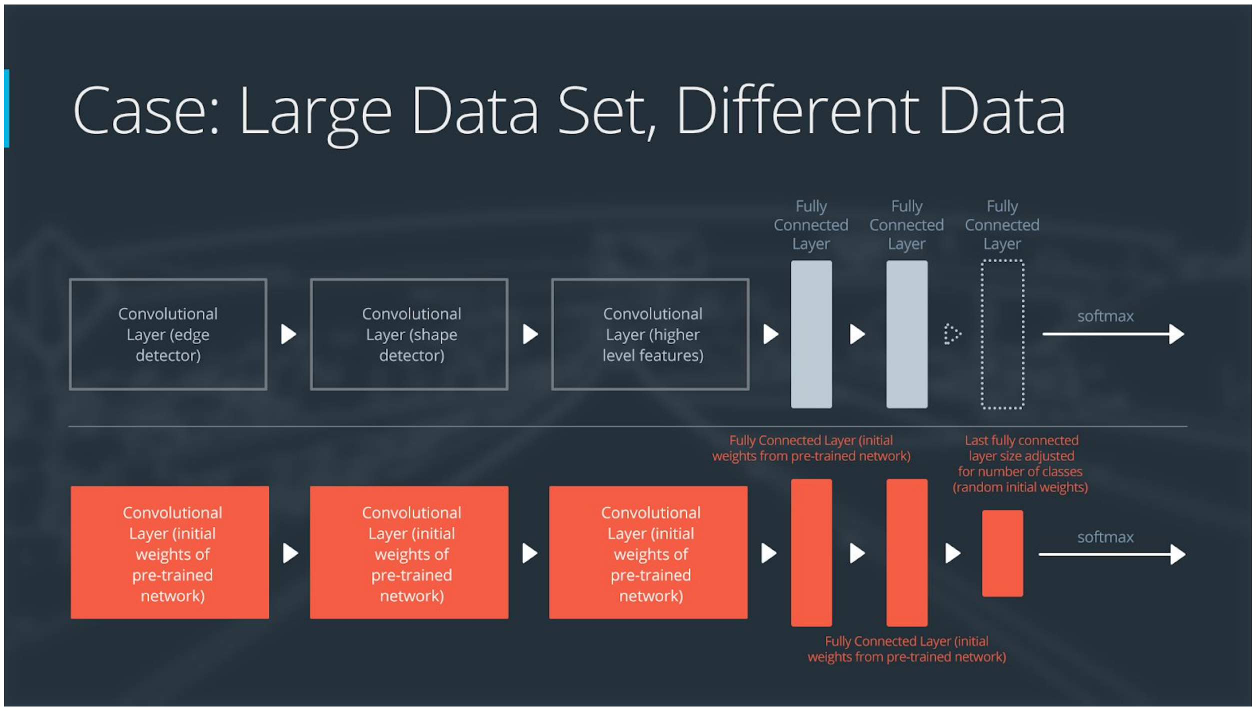
If the new data set is large and different from the original training data:

- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- retrain the network from scratch with randomly initialized weights
- alternatively, you could just use the same strategy as the "large and similar" data case

Even though the data set is different from the training data, initializing the weights from the pre-trained network might make training faster. So this case is exactly the same as the case with a large, similar data set.

If using the pre-trained network as a starting point does not produce a successful model, another option is to randomly initialize the convolutional neural network weights and train the network from scratch.

Here is how to visualize this approach:


Case: Large Data Set, Different Data
Fully Connected Layer
Fully Connected Layer
Fully Connected Layer
Convolutional Layer (edge detector)
Convolutional Layer (shape detector)
Convolutional Layer (higher level features)
softmax
Fully Connected Layer (initial weights from pre-trained network)
Last fully connected layer size adjusted for number of classes (random initial weights)
Convolutional Layer (initial weights of pre-trained network)
Convolutional Layer (initial weights of pre-trained network)
Convolutional Layer (initial weights of pre-trained network)
softmax
Fully Connected Layer (initial weights from pre-trained network)

Neural Network with Large Data Set, Different Data

FINAL IMP POINTS:

Watch those shapes

In general, you'll want to check that the tensors going through your model and other code are the correct shapes. Make use of the `.shape` method during debugging and development.

A few things to check if your network isn't training appropriately

Make sure you're clearing the gradients in the training loop with `optimizer.zero_grad()`. If you're doing a validation loop, be sure to set the network to evaluation mode with `model.eval()`, then back to training mode with `model.train()`.

CUDA errors

Sometimes you'll see this error:

```
RuntimeError: Expected object of type torch.FloatTensor but found
type torch.cuda.FloatTensor for argument #1 ‘mat1’
```

You'll notice the second type is `torch.cuda.FloatTensor`, this means it's a tensor that has been moved to the GPU. It's expecting a tensor with type `torch.FloatTensor`, no `.cuda` there, which means the tensor should be on the CPU. PyTorch can only perform operations on tensors that are on the same device, so either both CPU or both GPU. If you're trying to run your network on the GPU, check to make sure you've moved the model and all necessary tensors to the GPU with `.to(device)` where `device` is either `"cuda"` or `"cpu"`.