For more information on NVIDIA cuDNN, check out the [**official page**](https://developer.nvidia.com/cudnn).

Register for the NVIDIA Developer Program to access the latest NVIDIA SDK tools and be the first to hear about NVIDIA product announcements. Learn more at [***developer.nvidia.com/developer-program***](https://developer.nvidia.com/developer-program).

**GPU vs CPU Performance Increase Over Time**

The chart below gives a general idea of how, over time, GPU performance has increased faster than CPU performance. The data is based on a comparison of NVIDIA GPUs and Intel CPUs.

Chart, line chart

Description automatically generated

### The Four Main Cases When Using 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

Graphical user interface, application

Description automatically generated

Four Cases When Using Transfer Learning

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:

Graphical user interface, application

Description automatically generated

General Overview of a 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

Graphical user interface, application

Description automatically generated

Case 1: Small Data Set with Similar Data

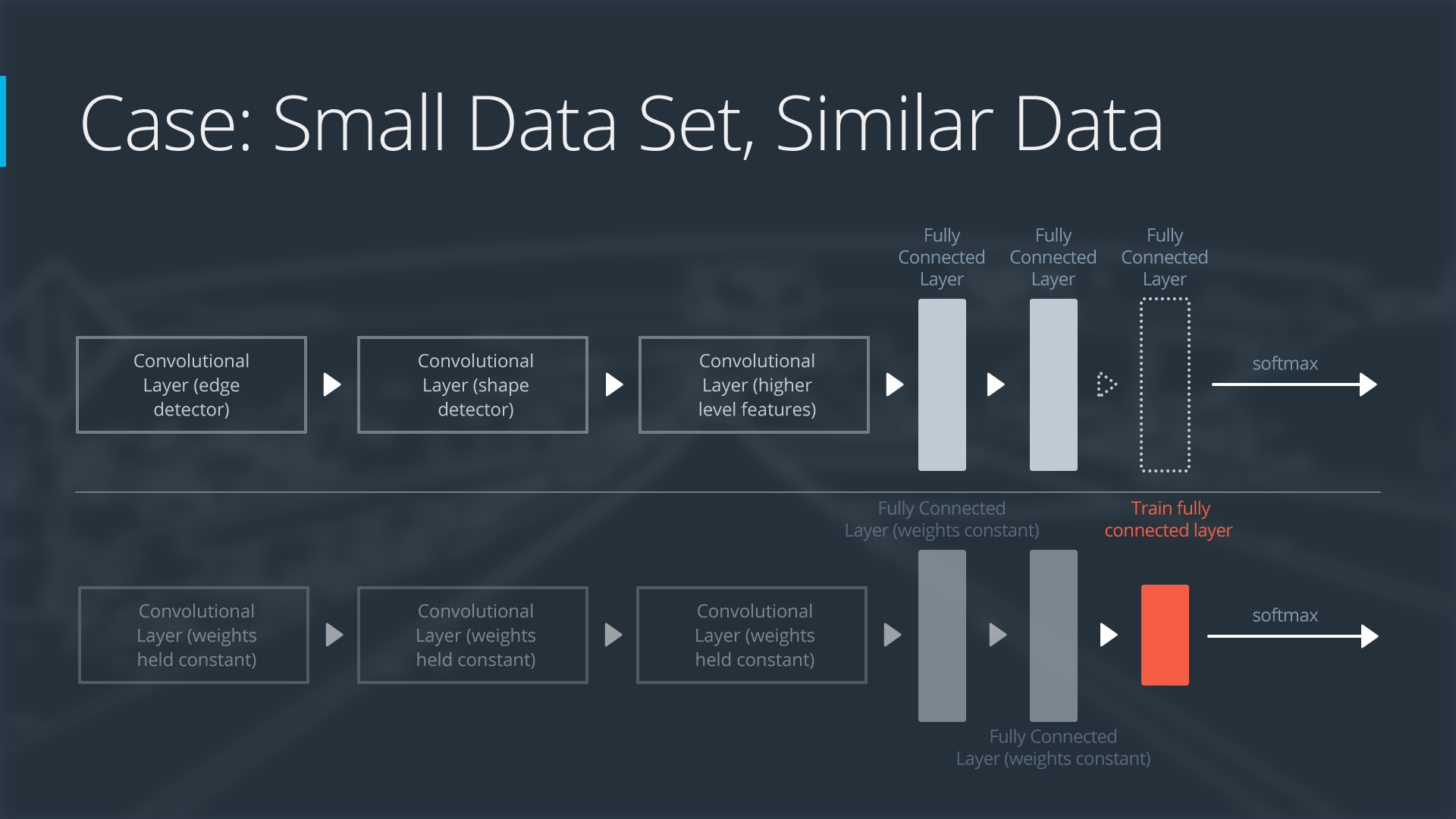
If the new data set is small and similar to the original training data:

* 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:



Neural Network with Small Data Set, Similar Data

### Case 2: Small Data Set, Different Data

Graphical user interface, application

Description automatically generated

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:

Graphical user interface, diagram

Description automatically generated

Neural Network with Small Data Set, Different Data

### Case 3: Large Data Set, Similar Data

Graphical user interface, application

Description automatically generated

Case 3: Large Data Set, Similar Data

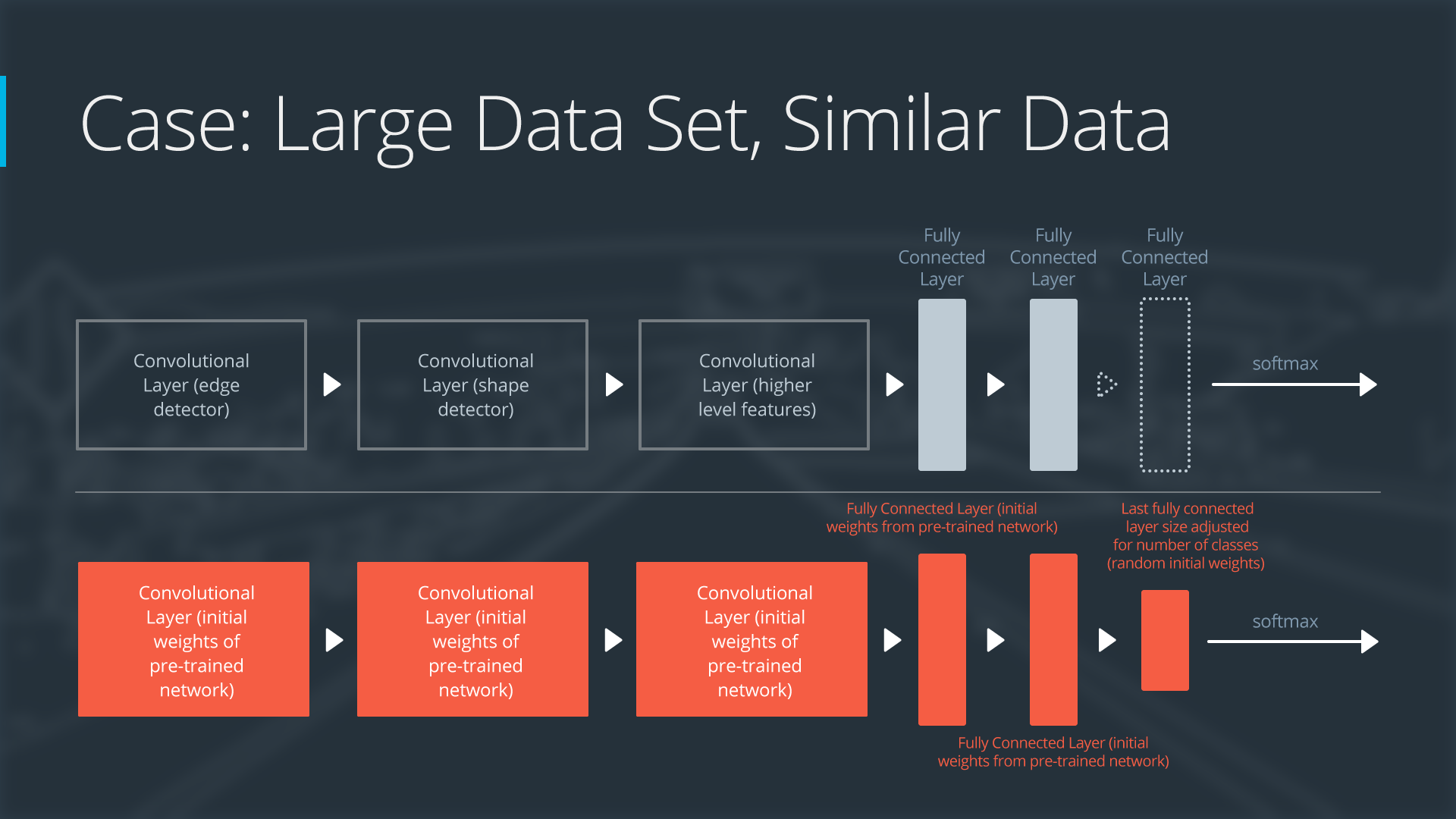
If the new data set is large and similar to the original training data:

* 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:



Neural Network with Large Data Set, Similar Data

### Case 4: Large Data Set, Different Data

Graphical user interface, application

Description automatically generated

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:

Graphical user interface, website

Description automatically generated

Neural Network with Large Data Set, Different Data

Interested in reading the original LeNet paper? You can find it [**here**](http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf).

The terms "aughts" or "oughts" refers to the first decade of a century. In this context the "late aughts" refers to the latter part of the first decade of the 2000's.

ImageNet was its own competition from 2012-2017, but now it's hosted on [**Kaggle**](https://www.kaggle.com/c/imagenet-object-localization-challenge)! There are 1,000 different image categories between over 14 million images, so it's a great way to get involved with large datasets.

Pre-training a network with the ImageNet dataset is a very common way to get a strong neural network that can be used for transfer learning. With recent versions of Keras, you can easily import a pre-trained network by using the [**Keras Applications**](https://keras.io/applications/) models. We'll come back to this soon.

Diagram

Description automatically generated

You can find the original VGG paper [**here**](https://arxiv.org/pdf/1409.1556.pdf).

## VGG in Keras

As we mentioned earlier, you can fairly quickly utilize a pre-trained model with [**Keras Applications**](https://keras.io/applications/). VGG16 is one of the built-in models supported. There are actually two versions of VGG, VGG16 and VGG19 (where the numbers denote the number of layers included in each respective model), and you can utilize either with Keras, but we'll work with VGG16 here.

**from** keras.applications.vgg16 **import** VGG16

model = VGG16(weights='imagenet', include\_top=**False**)

There are two arguments to VGG16 in this example, although there could be more or less (check out the linked documentation to see other possible arguments). The first, weights='imagenet', loads the pre-trained ImageNet weights. This is actually the default argument per the documentation, so if you don't include it, you should still be loading the ImageNet weights. However, you can also specify None here to get random weights if you just want the architecture of VGG16; this is not suggested here since you won't get the benefit of transfer learning.

The argument include\_top is for whether you want to include the fully-connected layer at the top of the network; unless you are actually trying to classify ImageNet's 1,000 classes, you likely want to set this to False and add your own additional layer for the output you desire.

### Pre-processing for ImageNet weights

There is another item to consider before jumping into using an ImageNet pre-trained model. These networks are typically pre-trained with a specific type of pre-processing, so you need to make sure to use the same pre-processing steps, or your network's outputs will likely be erratic.

VGG uses 224x224 images as input, so that's another thing to consider.

**from** keras.preprocessing **import** image

**from** keras.applications.vgg16 **import** preprocess\_input

**import** numpy **as** np

img\_path = 'your\_image.jpg'

img = image.load\_img(img\_path, target\_size=(224, 224))

x = image.img\_to\_array(img)

x = np.expand\_dims(x, axis=0)

x = preprocess\_input(x)

You can find the original GoogLeNet/Inception paper [**here**](https://arxiv.org/pdf/1409.4842.pdf).

## GoogLeNet/Inception in Keras

Inception is also one of the models included in [**Keras Applications**](https://keras.io/applications/). Utilizing this model follows pretty much the same steps as using VGG, although this time you'll use the InceptionV3 architecture.

**from** keras.applications.inception\_v3 **import** InceptionV3

model = InceptionV3(weights='imagenet', include\_top=**False**)

Don't forget to perform the necessary pre-processing steps to any inputs you include! While the original Inception model used a 224x224 input like VGG, InceptionV3 actually uses a 299x299 input.

[**Here**](https://arxiv.org/pdf/1512.03385.pdf) is the original ResNet paper, for those interested.

## ResNet in Keras

As you may have guessed, ResNet is also a model included in [**Keras Applications**](https://keras.io/applications/), under ResNet50.

**from** keras.applications.resnet50 **import** ResNet50

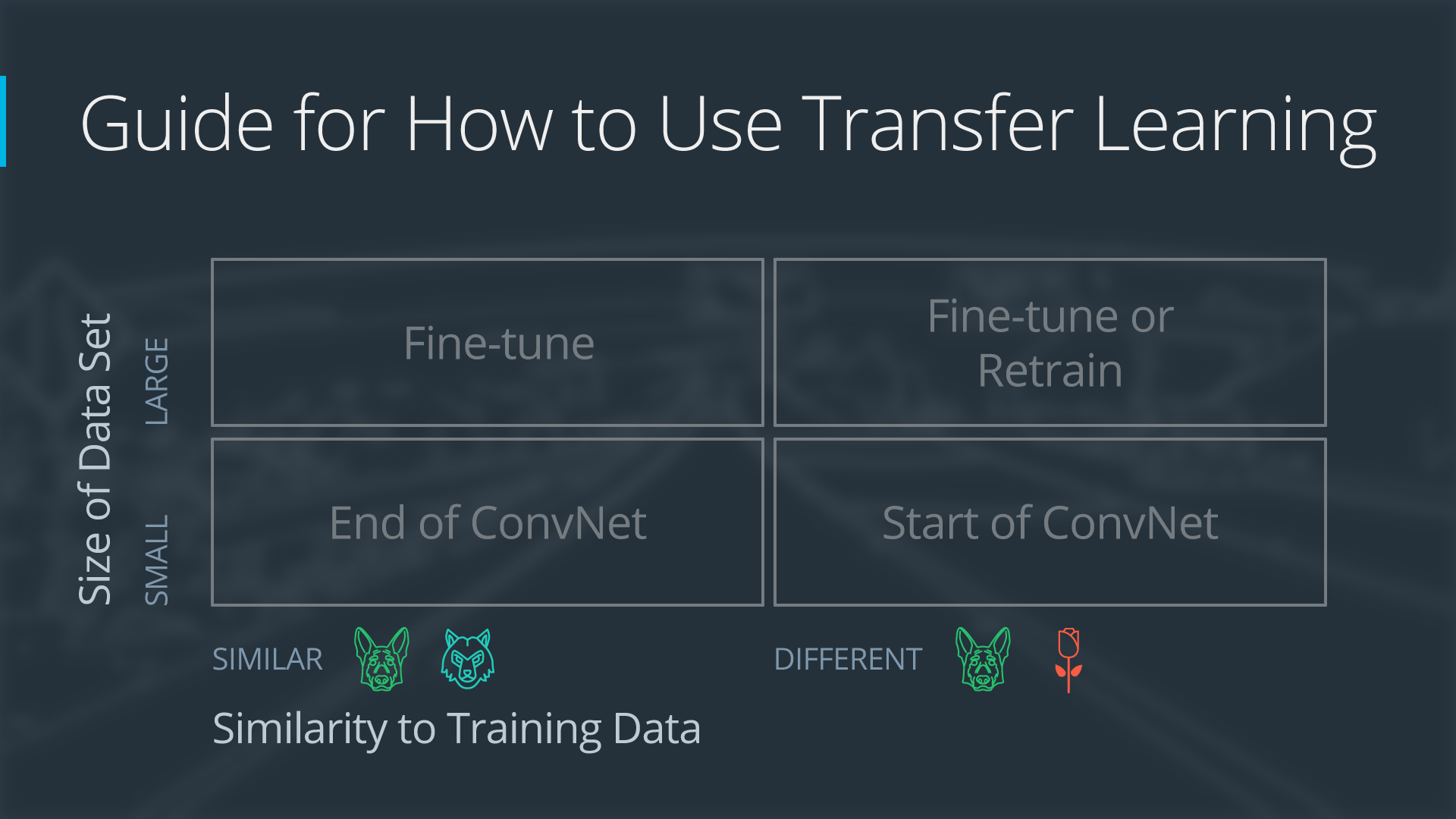
model = ResNet50(weights='imagenet', include\_top=**False**)

Again, you'll need to do ImageNet-related pre-processing if you want to use the pre-trained weights for it. ResNet50 has a 224x224 input size.

## Using Keras Applications models without pre-trained weights

So far, you've seen the effectiveness of models pre-trained on ImageNet weights, but what if we specify weights=None when we load a model? Well, you'll instead be randomly initializing the weights, as if you had built a model on your own and were starting from scratch.

From our chart before, there are few situations where this might even be a potential use case - basically, when you have data that is very different from the original data. However, given the large size of the ImageNet dataset (remember, it's over 14 million images from 1,000 classes!), it's highly unlikely this is really the case - it will almost always make the most sense to start with ImageNet pre-trained weights, and only fine-tune from there



Four Use Cases of Transfer Learning

Below, let's check out what happens when we try to use a pre-made model but set the weights to None - this means no training has occurred yet!

## Lab: Transfer Learning

In the below lab, you'll get a chance to try out a few instances of transfer learning, including both frozen and non-frozen pre-trained weights.

#### Frozen Weights

Frozen weights are often used when only fine-tuning the model, as backpropagation and weight updates will not be applied to any frozen layers during training. If you have an ImageNet pre-trained model, most of the network is likely applicable to your situation, so you may only need to cut off the top fully-connected layer, freeze all other layers, and just add one or more layers at the end that are not frozen to perform some fine-tuning.

There is also the option of not freezing the weights, which will start your model on the ImageNet pre-trained weights (if applicable) and then perform further training from there.

An additional benefit of freezing the weights also comes in the form of memory usage and training speed - for the larger networks such as VGG, there is a substantially larger memory usage and slower speed when it needs to perform backpropagation and weight updates across all layers instead of just on a small portion of (likely smaller) layers.

Note: There is a solution notebook that can be found by clicking on the Jupyter logo in the upper left of the workspace if you get stuck.

## Additional Resources on Deep Learning

Nice work reaching the end of the deep learning content! While you still have the project left to do here, we're also providing some additional resources and recent research on the topic that you can come back to if you have time later on.

Reading research papers is a great way to get exposure to the latest and greatest in the field, as well as expand your learning. However, just like the project ahead, it's often best to learn by doing - if you find a paper that really excites you, try to implement it (or even something better) yourself!

##### Optional Reading

All of these are completely optional reading - you could spend hours reading through the entirety of these! We suggest moving onto the project first so you have what you’ve learned fresh on your mind, before coming back to check these out.

We've categorized these papers to hopefully help you narrow down which ones might be of interest, as well as highlighted a couple key reads by category by including their Abstract section, which summarizes the paper.

### Behavioral Cloning

The below paper shows one of the techniques Waymo has researched using imitation learning (aka behavioral cloning) to drive a car.

[**ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst**](https://arxiv.org/abs/1812.03079) by M. Bansal, A. Krizhevsky and A. Ogale

**Abstract:** Our goal is to train a policy for autonomous driving via imitation learning that is robust enough to drive a real vehicle. We find that standard behavior cloning is insufficient for handling complex driving scenarios, even when we leverage a perception system for preprocessing the input and a controller for executing the output on the car: 30 million examples are still not enough. We propose exposing the learner to synthesized data in the form of perturbations to the expert's driving, which creates interesting situations such as collisions and/or going off the road. Rather than purely imitating all data, we augment the imitation loss with additional losses that penalize undesirable events and encourage progress -- the perturbations then provide an important signal for these losses and lead to robustness of the learned model. We show that the ChauffeurNet model can handle complex situations in simulation, and present ablation experiments that emphasize the importance of each of our proposed changes and show that the model is responding to the appropriate causal factors. Finally, we demonstrate the model driving a car in the real world.

### Object Detection and Tracking

The below papers include various deep learning-based approaches to 2D and 3D object detection and tracking.

[**SSD: Single Shot MultiBox Detector**](https://arxiv.org/abs/1512.02325) by W. Liu, et. al.

**Abstract:** We present a method for detecting objects in images using a single deep neural network. Our approach, named SSD, discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. At prediction time, the network generates scores for the presence of each object category in each default box and produces adjustments to the box to better match the object shape. Additionally, the network combines predictions from multiple feature maps with different resolutions to naturally handle objects of various sizes. Our SSD model is simple relative to methods that require object proposals because it completely eliminates proposal generation and subsequent pixel or feature resampling stage and encapsulates all computation in a single network. [...] Experimental results [...] confirm that SSD has comparable accuracy to methods that utilize an additional object proposal step and is much faster, while providing a unified framework for both training and inference. Compared to other single stage methods, SSD has much better accuracy, even with a smaller input image size. [...]

[**VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection**](https://arxiv.org/abs/1711.06396) by Y. Zhou and O. Tuzel

**Abstract:** Accurate detection of objects in 3D point clouds is a central problem in many applications, such as autonomous navigation, housekeeping robots, and augmented/virtual reality. To interface a highly sparse LiDAR point cloud with a region proposal network (RPN), most existing efforts have focused on hand-crafted feature representations, for example, a bird's eye view projection. In this work, we remove the need of manual feature engineering for 3D point clouds and propose VoxelNet, a generic 3D detection network that unifies feature extraction and bounding box prediction into a single stage, end-to-end trainable deep network. [...] Experiments on the KITTI car detection benchmark show that VoxelNet outperforms the state-of-the-art LiDAR based 3D detection methods by a large margin. Furthermore, our network learns an effective discriminative representation of objects with various geometries, leading to encouraging results in 3D detection of pedestrians and cyclists, based on only LiDAR.

[**Fast and Furious: Real Time End-to-End 3D Detection, Tracking and Motion Forecasting with a Single Convolutional Net**](http://openaccess.thecvf.com/content_cvpr_2018/papers/Luo_Fast_and_Furious_CVPR_2018_paper.pdf) by W. Luo, et. al.

**Abstract:** In this paper we propose a novel deep neural network that is able to jointly reason about 3D detection, tracking and motion forecasting given data captured by a 3D sensor. By jointly reasoning about these tasks, our holistic approach is more robust to occlusion as well as sparse data at range. Our approach performs 3D convolutions across space and time over a bird’s eye view representation of the 3D world, which is very efficient in terms of both memory and computation. Our experiments on a new very large scale dataset captured in several north american cities, show that we can outperform the state-of-the-art by a large margin. Importantly, by sharing computation we can perform all tasks in as little as 30 ms.

### Semantic Segmentation

The below paper concerns a technique called semantic segmentation, where each pixel of an image gets classified individually!

[**SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation**](https://arxiv.org/abs/1511.00561) by V. Badrinarayanan, A. Kendall and R. Cipolla

**Abstract:** We present a novel and practical deep fully convolutional neural network architecture for semantic pixel-wise segmentation termed SegNet. [...] The novelty of SegNet lies in the manner in which the decoder upsamples its lower resolution input feature map(s). Specifically, the decoder uses pooling indices computed in the max-pooling step of the corresponding encoder to perform non-linear upsampling. This eliminates the need for learning to upsample. The upsampled maps are sparse and are then convolved with trainable filters to produce dense feature maps. We compare our proposed architecture with the widely adopted FCN and also with the well known DeepLab-LargeFOV, DeconvNet architectures. This comparison reveals the memory versus accuracy trade-off involved in achieving good segmentation performance. [...] We show that SegNet provides good performance with competitive inference time and more efficient inference memory-wise as compared to other architectures. [...]