Transfer Learning

Samuel Navarro

June 19, 2019

Contents

1 Transfer Learning

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.

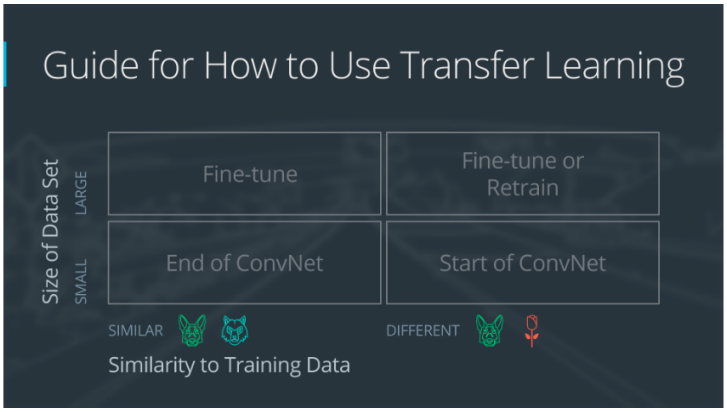

Figure 1: Transfer Learning Guide

4. New data set is large, new data is different from original training data.

Overfitting is a concern when using transfer learning with a small data set.

1.1 Demonstration Network

Our example network contains three convolutional layers and three fully connected layers:

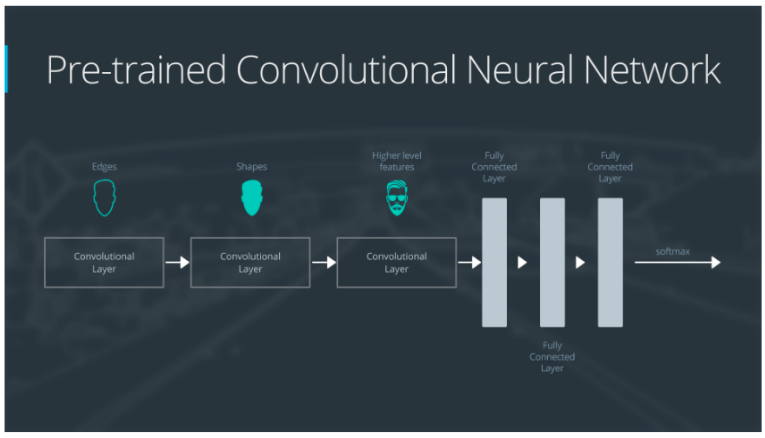

Figure 2: Pretrained ConvNet

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

1.1.1 Case1: Small Data Set, Similar Data

If the 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 pretrained 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 form 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.

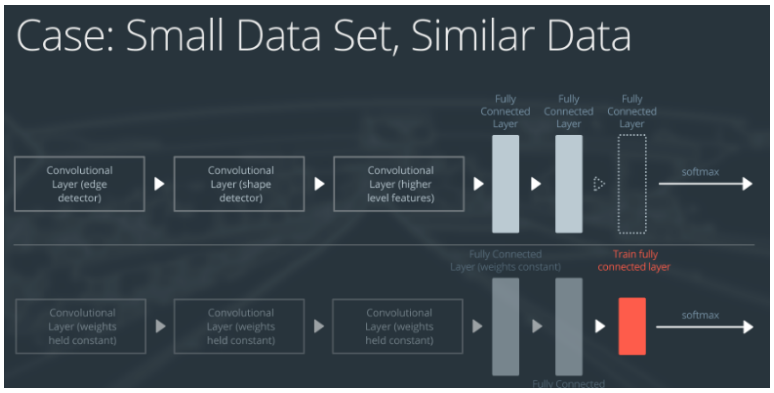

Figure 3: Small Dataset Similar Data

1.1.2 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.

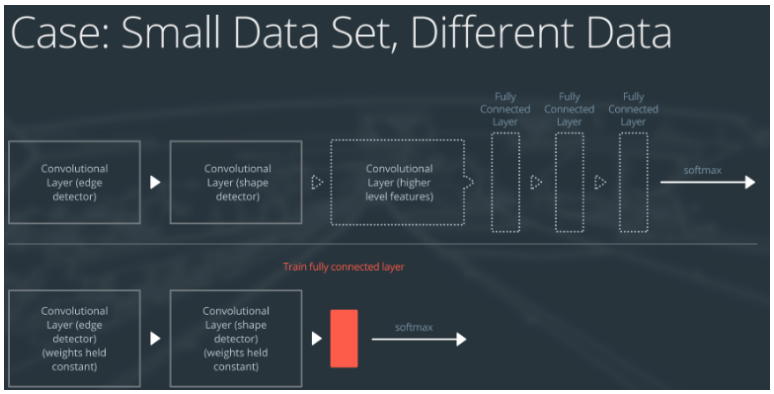

Figure 4: Small Dataset, Different Data

1.1.3 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 dta set share higher level features, the entire neural network is used.

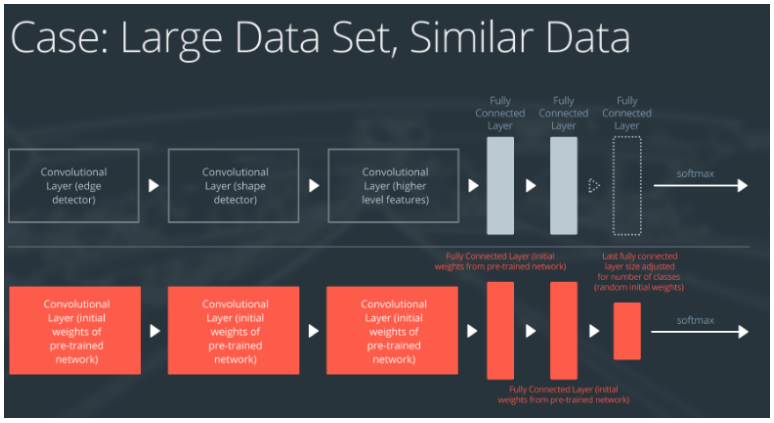

Figure 5: Large Dataset Similar Data

1.1.4 Case 4: Large Data Set, Different Data

If the new data set is large and different from the original 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.

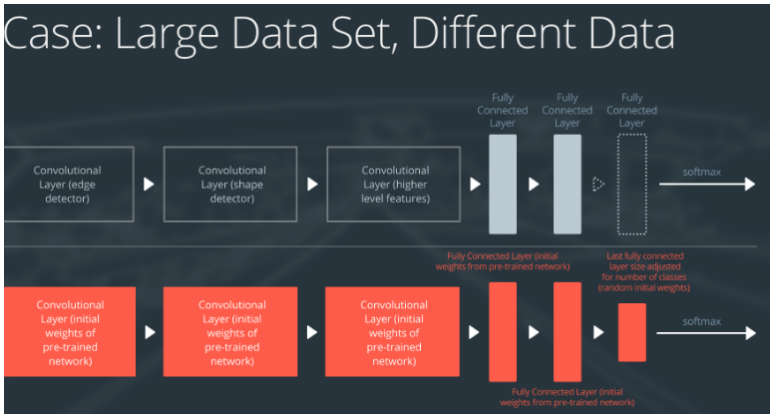

Figure 6: Large Dataset, Different Data

1.2 Popular Architectures

1.2.1 AlexNet

AlexNet puts the network on two GPUs. Although most of the calculations are done in parallel, the GPUs communicate with each other in certain layers. The original paper on AlexNet said that parallelizing the network decreased the classification error rate by 1.7% when compared to a neural network that used half as may neurons on one GPU.

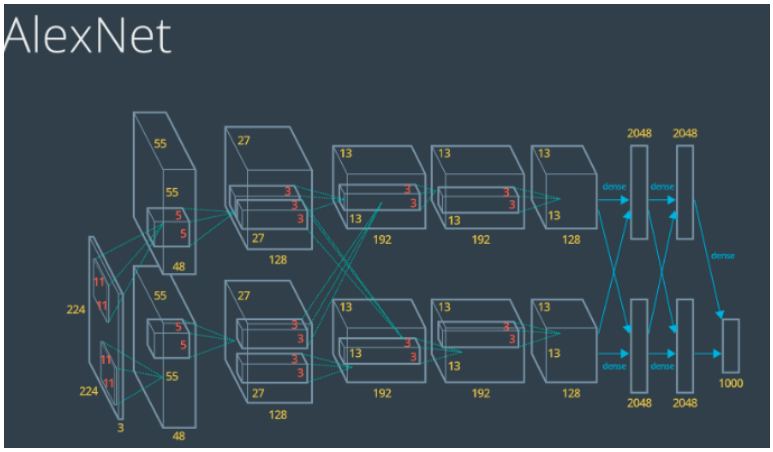

Figure 7: AlexNet Architecture

AlexNet is still used today as a starting point for building and training neural networks. Actually, sometimes the researchers uses a small version of AlexNet because they discovered they some features of the AlexNet wasn't necessary.

1.2.2 VGG

Here is the VGG Paper

Is a large and elegant Architecture which makes it great for transfer learning. It is just a long sequence of three-by-three convolutions, broken up by two-by-two pooling layers and finished by a trio of fully connected layers at the end.

Lot's of engineers uses VGG as a starting point for other image classification tasks.

There are actually two versions of VGG, VGG16 and VGG19 (where the numbers denote the number of layers included in each respective model).

The argument `weights='imagenet'` loads the pre-trained ImageNet weights. You can also specify `None` to get random weights if you just want the architecture of **VGG16**. The argument `include_top` is for whether you want to include the fully-connected layer at the top of the network; you likely want

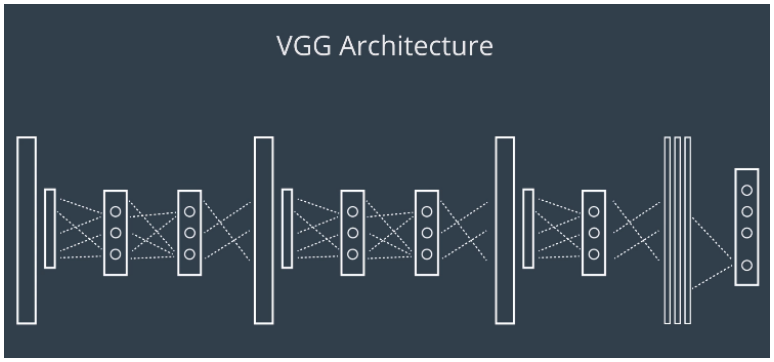

Figure 8: VGG

```
from keras.applications.vgg16 import VGG16

model = VGG16(weights='imagenet', include_top=False)
```

Listing 1: caption name

to set this to `False` unless you are actually trying to classify ImageNet's 1,000 classes.

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

1.2.3 GoogLeNet

The paper of GoogleNet/Inception

GoogLeNet great advantage is that it runs really fast. The team at google develop a concept called the Inception module, which trains really well and is efficiently deployable.

The inception module creates a situation which the total number of parameters is very small and this is why **GoogLeNet** runs almost as fast as **AlexNet**. **GoogLeNet** is a great choice if you want to run your network in real time, like in a SelfDriving car.

In a normal **Inception Network**, you would see form the model summary that the last two layers were a global average pooling layers, and a fully-connected "Dense" layer.

Notes on Keras v.2.0.9:

1. How many layers you drop is up to yo. We fropped the final two already by setting `include_top` to False in the original loading of the model, but you could instead just run `pop()` twice to achieve similar results. *Keras requires us to set include_top to False in order to change*

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input, VGG16, decode_predictions
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)

model = VGG16(weights='imagenet')
predictions = model.predict(x)

print("Predicted:', decode_predictions(predictions, top=3)[0])
```

Listing 2: caption name

the input_shape. Additional layers could be dropped by additional calls to `pop()`

2. If you make a mistake with `pop()` you'll want to reload the model. If you use it multiple times, the model will continue to drop more and more layers, so you may need to check `model.summary()` again.

The Model API instead of using `model.add()` you explicitly tell the model which previous layer to attach to the current layer. This is useful is you want to use more advanced concepts like skip layers

For example:

`x = Dropout(0.2)(inp)` is how you would attach a dropout with it's input coming from `inp`.

Keras Callbacks

Keras callbacks allow you to gather and store additional information during training, such as the best model, or even stop training early if the validation accuracy has stopped improving. These methods can help to avoid overfitting, or avoid other issues.

There's two key callbacks to mention here, 'ModelCheckpoint' and 'EarlyStopping'. As the names may suggest, model checkpoint saves down the best model so far based on a given metric, while early stopping will end training before the specified number of epochs if the chosen metric no longer improves after a given amount of time.

To set these callbacks, you could do the following:

`checkpoint = ModelCheckpoint(filepath=save_path, monitor='val_loss', save_best_only=True)`

```
from keras.applications.resnet50 import ResNet50
model = ResNet50(weights='imagenet', include\_top=False)
```

Listing 3: caption name

This would save a model to a specified `save_path`, based on validation loss, and only save down the best models. If you set `save_best_only` to 'False', every single epoch will save down another version of the model.

`stopper = EarlyStopping(monitor='val_acc', min_delta=0.0003, patience=5)`

This will monitor validation accuracy, and if it has not decreased by more than 0.0003 from the previous best validation accuracy for 5 epochs, training will end early.

You still need to actually feed these callbacks into 'fit()' when you train the model (along with all other relevant data to feed into 'fit'):

`model.fit(callbacks=[checkpoint, stopper])`

1.2.4 ResNet

The 2015 ImageNet winner was the **ResNet**. Here's the paper. It has 152 layers (for contrast, **AlexNet** has 8 layers. **VGG** has 19 layers and **GoogLeNet** has 22 layers.

2 Further

2.1 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 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.

2.2 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 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 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 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.

2.3 Semantic Segmentation

SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation 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. [...]