

Transfer Learning for Computer Vision

Topics Covered

1. Mobilenet
2. Retinanet
3. Optimizing Transfer Learning-Based Models
4. Tensorflow Hub for Computer Vision Applications
5. Video Processing Using Neural Networks
6. Best Practices with Transfer Learning for Computer Vision
7. Summary

MobileNet

As the name applied, the MobileNet model is designed to be used in mobile applications, and it is TensorFlow's first mobile computer vision model.

MobileNet uses **depthwise separable convolutions**. It significantly **reduces the number of parameters** when compared to the network with regular convolutions with the same depth in the nets. This results in lightweight deep neural networks.

A depthwise separable convolution is made from two operations.

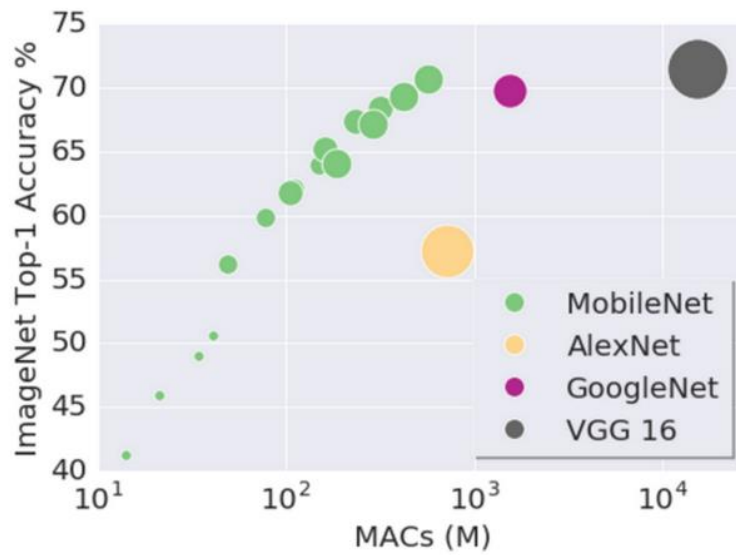
1. **Depthwise convolution.**
2. **Pointwise convolution.**

MobileNet is a class of CNN that was open-sourced by Google, and therefore, this gives us an excellent starting point for training our classifiers that are insanely small and insanely fast.



When MobileNets Applied to Real Life

The speed and power consumption of the network is proportional to the number of MACs (Multiply-Accumulates) which is a measure of the number of fused Multiplication and Addition operations.



RetinaNet

RetinaNet is one of the best one-stage object detection models that has proven to work well with dense and small scale objects. For this reason, it has become a popular object detection model to be used with aerial and satellite imagery.



Figure 1. Swimming Pools detection using RetinaNet

RetinaNet has been formed by making two improvements over existing single stage object detection models - Feature Pyramid Networks (FPN) and Focal Loss. Before diving into RetinaNet's architecture, let's first understand FPN.

What's a Feature Pyramid Network?

Traditionally, in computer vision, featurized image pyramids have been used to detect objects with varying scales in an image. Featurized image pyramids are feature pyramids built upon image pyramids. This means one would take an image and subsample it into lower resolution and smaller size images (thus, forming a pyramid). Hand-engineered features are then extracted from each layer in the pyramid to detect the objects. This makes the pyramid scale-invariant. But, this process is compute and memory intensive.

With the advent of deep learning, these hand-engineered features were replaced by CNNs. Later, the pyramid itself was derived from the inherent pyramidal hierarchical structure of the CNNs. In a CNN architecture, the output size of feature maps decreases after each successive block of convolutional operations, and forms a pyramidal structure.

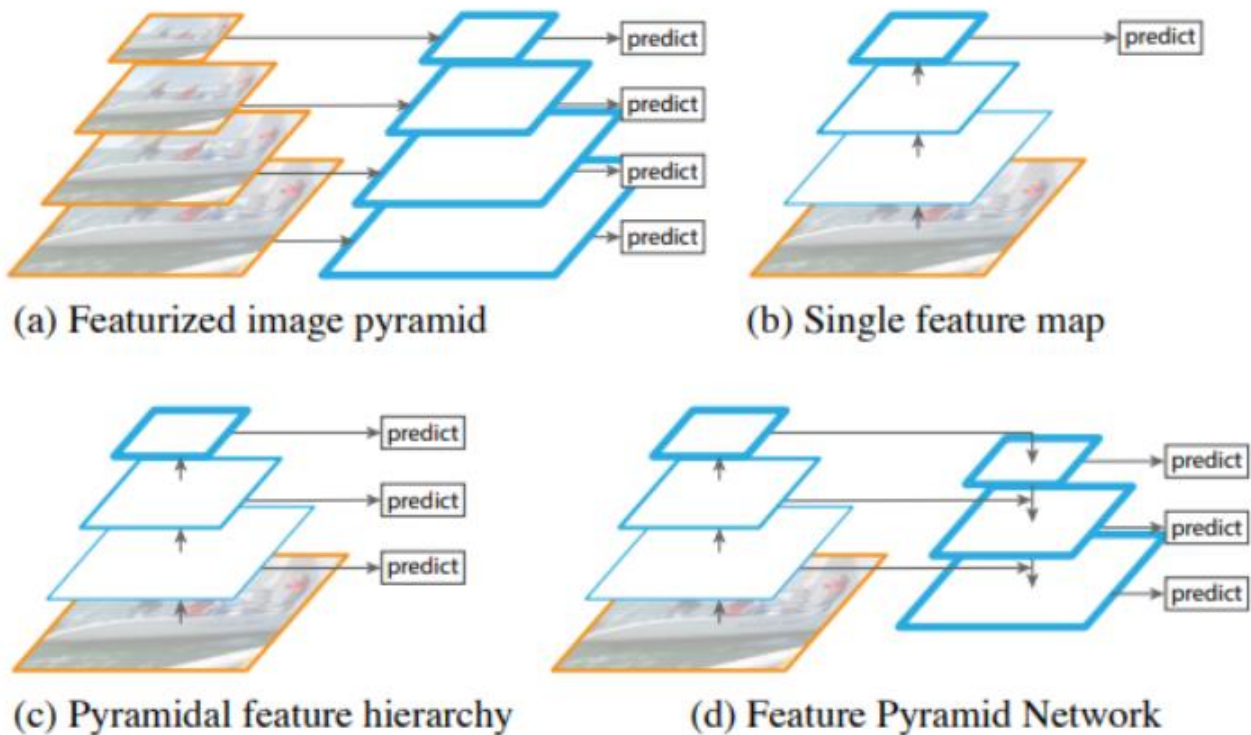


Figure 2. Different types of pyramid architectures

There have been various architectures that utilize the pyramid structure (Figure 2).

The (a) Featurized image pyramid, as we have discussed, is compute intensive.

(b) Single (scale) feature maps have been used for faster detections. Even though they are robust and fast, pyramids are still needed to get the most accurate results.

(c) Pyramidal feature hierarchy has been utilized by models such as Single Shot detector, but it doesn't reuse the multi-scale feature maps from different layers.

(d) Feature Pyramid Network (FPN) makes up for the shortcomings in these variations.

FPN creates an architecture with rich semantics at all levels as it combines low-resolution semantically strong features with high-resolution semantically weak features. This is achieved by creating a top-down pathway with lateral connections to bottom-up convolutional layers.

Top-down pathway, bottom-up pathway and lateral connections will be better understood in the next section when we take a look at the RetinaNet architecture. RetinaNet incorporates FPN and adds classification and regression subnetworks to create an object detection model.

RetinaNet architecture

There are four major components of a RetinaNet model architecture (Figure 3):

- Bottom-up Pathway - The backbone network (e.g. ResNet) which calculates the feature maps at different scales, irrespective of the input image size or the backbone.
- Top-down pathway and Lateral connections - The top-down pathway up samples the spatially coarser feature maps from higher pyramid levels, and the lateral connections merge the top-down layers and the bottom-up layers with the same spatial size.
- Classification subnetwork - It predicts the probability of an object being present at each spatial location for each anchor box and object class.
- Regression subnetwork - It's regressing the offset for the bounding boxes from the anchor boxes for each ground-truth object.

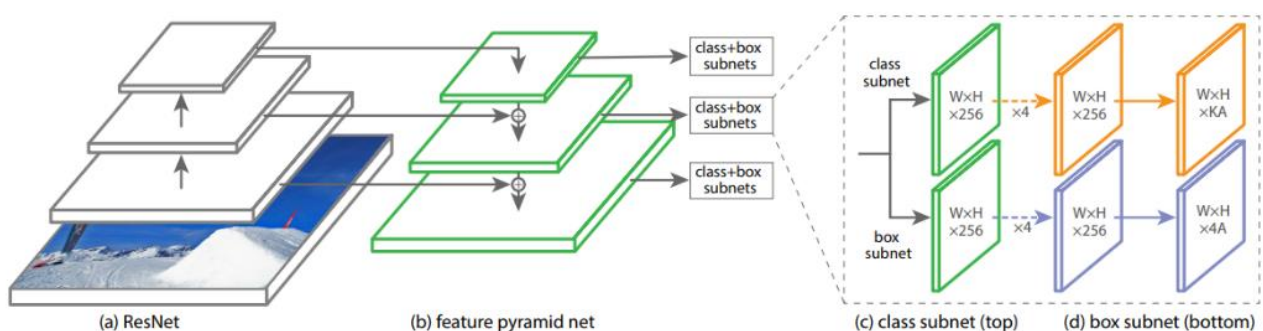


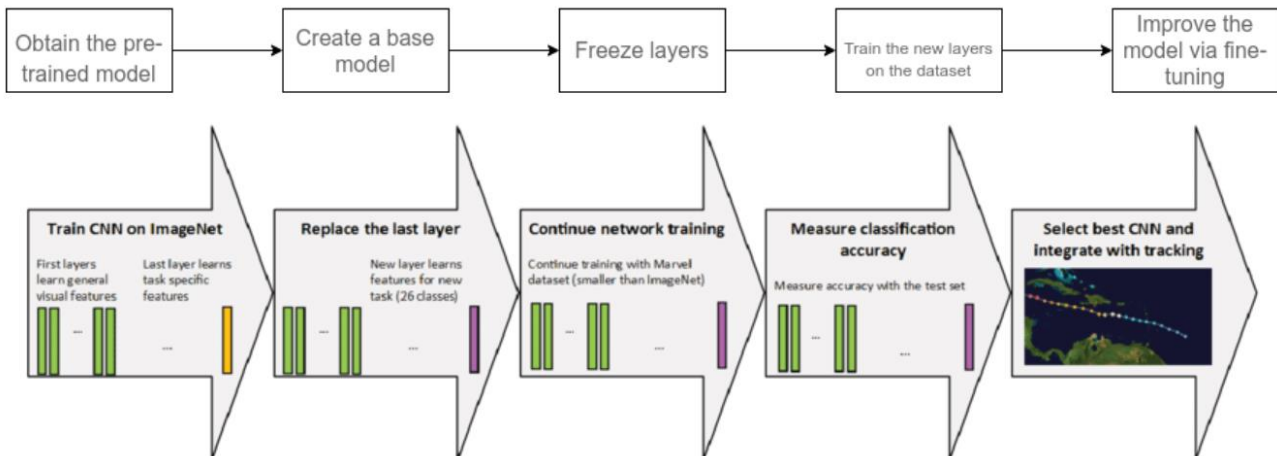
Figure 3. RetinaNet model architecture

Focal Loss

Focal Loss (FL) is an enhancement over Cross-Entropy Loss (CE) and is introduced to handle the class imbalance problem with single-stage object detection models. Single Stage models suffer from a extreme foreground-background class imbalance problem due to dense sampling of anchor boxes (possible object locations). In RetinaNet, at each pyramid layer there can be thousands of anchor boxes. Only a few will be

assigned to a ground-truth object while the vast majority will be background class. These easy examples (detections with high probabilities) although resulting in small loss values can collectively overwhelm the model. Focal Loss reduces the loss contribution from easy examples and increases the importance of correcting misclassified examples.

Optimizing Transfer Learning-based models

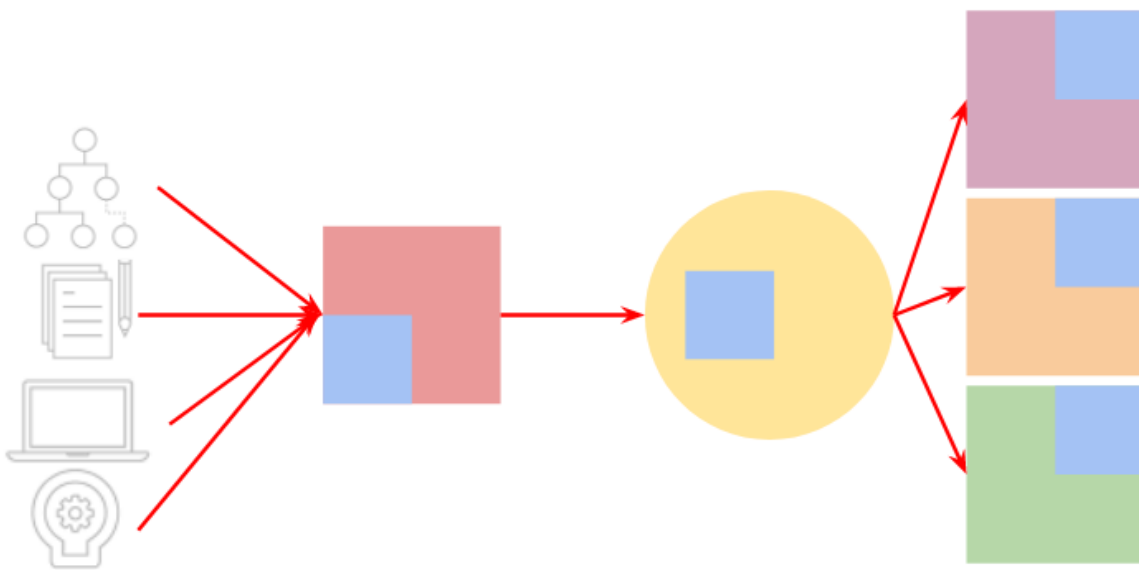


- Transfer learning models focus on storing knowledge gained while solving one problem and applying it to a different but related problem.
- Instead of training a neural network from scratch, many pre-trained models can serve as the starting point for training. These pre-trained models give a more reliable architecture and save time and resources.
- Transfer learning is used in scenarios where there is not enough data for training or when we want better results in a short amount of time.
- Transfer learning involves selecting a source model similar to the target domain, adapting the source model to the target model before transferring the knowledge, and training the source model to achieve the target model.
- It is common to fine-tune the higher-level layers of the model while freezing the lower levels as the basic knowledge is the same that is transferred from the source task to the target task of the same domain.
- In tasks with a small amount of data, if the source model is too similar to the target model, there might be an issue of overfitting. To prevent the transfer learning model from overfitting, it is essential to tune the learning rate, freeze some layers from the source model, or add linear classifiers while training the target model can help avoid this issue.

Tensorflow hub for Computer Vision applications

TensorFlow Hub is a repository of trained machine learning models ready for fine-tuning and deployable anywhere. Reuse trained models like BERT and Faster R-CNN with just a few lines of code.

Making any model from scratch requires a good algorithm selection and architecture, lot of data, compute and of course domain expertise. That seems like quite a lot.



The idea behind using TF Hub

So what you do with TF Hub is that someone takes all this what is required to make a model from scratch makes what is called a module and then takes out some part of the model which is reusable.

So in the case of text, we use word embeddings which give it a lot of power and often require quite some compute to train. In the case of images, it could be features. So you have this reusable part from a module in the TF Hub repo. With transfer learning this reusable piece could be used in your different models and it might even serve a different purpose and this is essentially how you would be using TF Hub. So when I say different purpose it could maybe be a classifier trained on a thousand labels but you just use it to predict or do a distinction between 3 classes. That's just one example but you now maybe understand it.

Transfer Learning is useful because of

- Generalization

Transfer Learning allows you to achieve generalization on your data, this is particularly quite helpful when you want to run your model on real-life data.

- Less data

As you have your embeddings or weights or convolutions learned pre-hand you can train high-quality models with very little data. This is very helpful when you can just not get any more data or getting more data is too costly.

- Training time

We already discussed earlier that Transfer Learning would take less time to train.

TF Hub in Practice

Install TF Hub

We will not be focussing on installing TF Hub here it is pretty straightforward, find installation steps [here](#).

The `tensorflow_hub` library can be installed alongside TensorFlow 1 and TensorFlow 2. We recommend that new users start with TensorFlow 2 right away, and current users upgrade to it.

Use `pip` to install TensorFlow 2 as usual. Then install a current version of `tensorflow-hub` next to it (must be 0.5.0 or newer).

```
$ pip install "tensorflow>=2.0.0"
$ pip install --upgrade tensorflow-hub
```

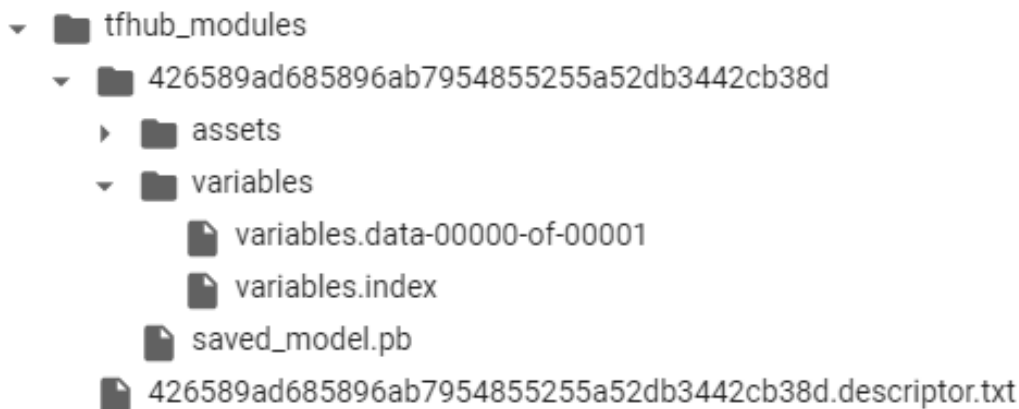
The TF1-style API of TensorFlow Hub works with the v1 compatibility mode of TensorFlow 2.

Loading models

You can easily load your model with this piece of code

```
MODULE_HANDLE =  
'https://tfhub.dev/google/imagenet/inception_v3/classification/4'module =  
hub.load(MODULE_URL)
```

You can, of course, change the MODULE_HANDLE URL for the model you need.



```
tfhub_modules  
├── 426589ad685896ab7954855255a52db3442cb38d  
│   ├── assets  
│   └── variables  
│       ├── variables.data-00000-of-00001  
│       ├── variables.index  
│       └── saved_model.pb  
└── 426589ad685896ab7954855255a52db3442cb38d.descriptor.txt
```

So, when you call the model. load method TF Hub downloads this for you, and you will notice that it is a saved model directory and you have your model as a protobuf file which does the graph definition for you.

- Saved Model CLI

There is another great interface called the saved model CLI which I find pretty useful. This gives you a lot of useful information about your saved model like operation signatures and input-output shapes.

```
!saved_model_cli show --dir [DIR] --all
```

Here is sample output showing the information this tool provides-

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['__saved_model_init_op']:
  The given SavedModel SignatureDef contains the following input(s):
  The given SavedModel SignatureDef contains the following output(s):
    outputs['__saved_model_init_op'] tensor_info:
      dtype: DT_INVALID
      shape: unknown_rank
      name: NoOp
  Method name is:

Defined Functions:
  Function Name: '__call__'
    Option #1
      Callable with:
        Argument #1
          inputs: TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=u'inputs')
        Argument #2
          DType: bool
          Value: False
        Argument #3
          DType: bool
          Value: False
        Argument #4
          batch_norm_momentum: TensorSpec(shape=(), dtype=tf.float32,
name=u'batch_norm_momentum')
      ...
      ...
```

Saved Model CLI Output

- Inference

You can now directly perform inference with your loaded model.

```
tf.nn.softmax(module([img]))[0]
```

But this is a bad approach as you are not generalizing your model.

Performing Inference for Images

A lot of times it is better to use image feature vectors. They remove the final classification layer from the network and allow you to train a network on top of that allowing for greater generalization and in turn better performance on real-life data.

You could do it simply in this manner-

```
model = tf.keras.Sequential([
    hub.KerasLayer(MODULE_HANDLE,
input_shape=IMG_SIZE + (3,),
output_shape=[FV_size], trainable=True),
tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')])
```

So you can see we now use `hub.KerasLayer` to create my model as a Keras layer and I also set trainable to be True as we want to perform transfer learning.

So we can then have our model add a Dense layer after it so you are taking the model adding your own layers and then retraining it with the data you have, of course, you could have multiple layers or even convolutional layers on top of this. And then you could fit it like any other Keras model after compiling it. The fitting and compiling process remains the same which shows us the level of integration of TF Hub. let's refer example of simple model

Neural Style Transfer

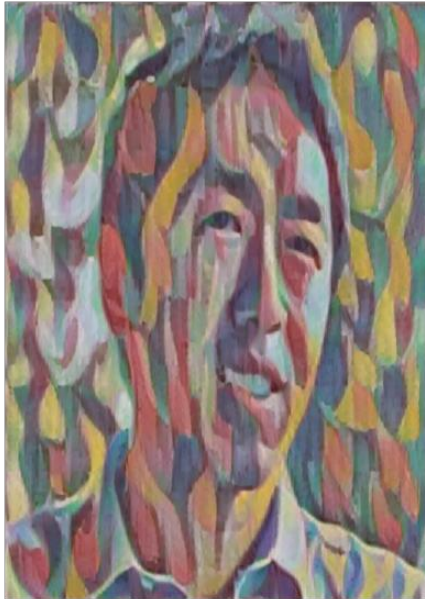
A neural style transfer algorithm would ideally require quite some amount of compute and time, we can use TensorFlow Hub to do this easily. We will start by defining some helper functions to convert images to tensors and vice-versa.

You can then convert the images to tensors, create an arbitrary image stylization model and we can start building images right away!

```
hub_module = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-
stylization-v1-256/2')

combined_result = hub_module(tf.constant(content_image_tensor[0]),
tf.constant(style_image_tensor[1]))[0] tensor_to_image(combined_result)
```

With this, you can now create an image like this



The output of Neural Style Transfer

Video Processing using Neural Networks

Video data is a common asset which is used everyday, whether it is a live stream in a personal blog or security camera in a manufacturing building. A machine learning appliance is becoming a normal tool for processing video in a variety of tasks.

Video is simply a sequence of images referred to as *frames*. Precision of video therefore is a combination of image precision and also the number of frames captured per second (the frame rate).

It's possible to analyze video simply by considering each image in isolation, which may be perfectly adequate for many scenarios. For example, facial detection and recognition can be performed on a frame-by-frame basis by feeding each frame into the neural network one by one. However, the additional dimension of time opens opportunities for understanding movement. This allows video understanding to extend to more complex semantic understanding, such as:

Entity tracking

Tracking the paths of specific objects (such as people or vehicles) over time. This might include inferring positional information when the entity is obscured or leaves the scene.

Activity recognition

Extending the idea of object recognition to detect activities within a scene using additional information pertaining to movement. For example, this could be understanding gestures used to control a device or detecting behavior (such as aggression) within a scene. The ability to recognize activities within videos enables other higher-level processing, such as video description.

As with image and audio, there are more classical approaches to video processing that do not use neural networks. But, once again, DNNs remove the requirement to hand-code the rules for extracting features.

Unsurprisingly, the element of time will add complexity to processing frames. There are, however, approaches to dealing with this additional dimension using the architectural principles described previously for CNNs and RNNs. For example, one option is to exploit 3D Convolutions. This extension of the

convolutional principles used in image CNNs includes a third temporal dimension across frames which is processed in the same way as spatial dimensions within each frame. Alternatively, it's possible to combine spatial learning of a CNN with sequential learning borrowed from an RNN architecture. This can be done by exploiting the CNN to extract features on a per-frame basis and then using these features as sequential inputs to an RNN.

Whereas color images are passed to a neural network as a 4D tensor, video inputs are represented by a whopping 5D tensor. For example, if we have one minute's worth of video sampled at 15 frames per second, that gives us 900 frames. Let's assume low-definition video (224 x 224 pixels) and color using RGB channels. We have a 4D tensor with the following shape:

shape = (900, 224, 224, 3)

If there are 10 videos in the input batch, the shape becomes 5D:

shape = (10, 900, 224, 224, 3)

Best practices with Transfer learning for computer vision

Fixed features, fine-tuning and trained from scratch

There are two types of transfer learning: use fixed features and fine-tuning. Both involve restoring weights from a pre-trained ImageNet model, and re-training the network for the new classes of interests. The difference is, the fixed features approach freezes early layers and only trains the last layer. In contrast, the fine-tuning approach trains all layers. Intuitively, the fixed features approach is less prone to over-fitting while the fine-tuning approach is better at handling new classes that are irrelevant to ImageNet.

The Google researchers set themselves the goal of systematically studying the pros and cons of these two approaches, together with the case that training the network from scratch (no transfer learning).

Figure 1 summarizes their findings: First, **Dataset similar to ImageNet benefit more from transfer learning than dataset un-similar to ImageNet**. The first column of Figure 1 shows an example: the t-SNE embeddings of the penultimate layer of the same Inception-V4 network have very different characters on these two datasets: the embedding of FGVC aircraft dataset has no structure while the embedding of CIFAR-10 (similar to ImageNet) has clear separation between different classes.

Second, **both fine-tuning and trained from scratch led to significantly better features** (column 2&3, Figure 1, where the classes are clearly separated). This behavior also generalizes to many datasets. So the message is do not fix the early layers in transfer learning. However there is one caveat: Using fixed features can be better if there is very little training data. We will come back to this in details later.

Third, although training from scratch achieved parity with fine-tune, it is at the cost of significantly more training data and longer training time. So **one should prefer fine-tuning over trained from scratch**. Again there is an exception: training from scratch can give better accuracy if the new dataset has little similarity with ImageNet. However this does not happen very often considering the scale of ImageNet.

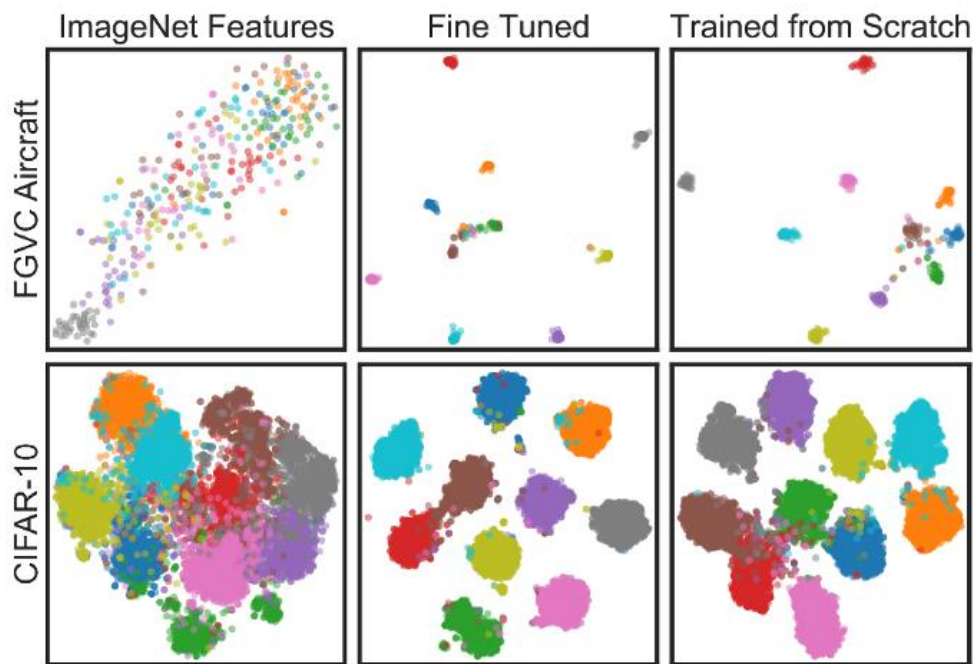


Figure 1. 2D embedding (t-SNE) of features from the penultimate layer of Inception for different approaches.

In summary,

Always prepare yourself at least a few dozen of training images for each class and use fine-tune.

ResNets: the best fixed feature extractor

If for whatever reason, you can't get a few dozen examples for each class. Then fixed features is a better choice. Why? Because it is safer to make decisions based on the past experience if you don't have enough observation in a new situation.

An interesting discovery of the study is that ResNets seems to be the best fixed feature extractor. However this is completely empirical based and there is no theory behind it. Moreover, ImageNet accuracy does not correlate well with the transfer accuracy in the case of fixed feature extractor: the Pearson correlation coefficient between the two is only 0.24. This meaning in practice there is no way to choose the best fixed feature extractor based on their ImageNet performance.

Our experience is that Inception-V4 and NASNet-A-Large are also very good fixed feature extractors and they sometimes perform better than ResNets. In conclusion, if you want to use the fixed feature approach, at least try ResNets, Inception-V4 and NASNet-A-Large.

ImageNet: the oracle of fine-tuning

There is a world where ImageNet accuracy correlate well with transfer accuracy — fine-tuning. The Pearson correlation coefficient between the two is 0.86. Remember it was 0.24 in the fixed feature case. This gives us an other reason to use fine-tune: Use it as the an oracle of fine-tuning. One can expect the best fine-tuning performance to come from the model that has the best ImageNet performance.

The best model on ImageNet is NASNet-A-Large. It has won 9 of the 12 transfer tasks in the study (Figure 2). So choose NASNet-A-Large for fine-tuning

Dataset	Acc.	Best network
Food-101	90.1	NASNet-A Large, fine-tuned
CIFAR-10	98.4^a	NASNet-A Large, fine-tuned
CIFAR-100	88.2^a	NASNet-A Large, fine-tuned
Birdsnap	78.5	NASNet-A Large, fine-tuned
SUN397	66.5	NASNet-A Large, fine-tuned
Stanford Cars	93.0	Inception v4, random init
FGVC Aircraft	89.4	Inception v3, fine-tuned
VOC 2007 Cls.	88.4	NASNet-A Large, fine-tuned
DTD	76.7	Inception-ResNet v2, fine-tuned
Oxford-IIIT Pets	94.3	NASNet-A Large, fine-tuned
Caltech-101	95.0	NASNet-A Large, fine-tuned
Oxford 102 Flowers	97.7	NASNet-A Large, fine-tuned

Figure 2. Twelve transfer tasks and the best network for each of them.

Don't train from scratch

There is really no reason to prefer train from scratch over fine-tuning. Just consider the difference between the two: how weights are initialized. Fine-tuning uses weights from a pre-trained model and trained from scratch uses random weights. As long as there is similarity between the original task and the transferred task, weights from a pre-trained model should do better than random numbers.

In consequence, the direct benefit of fine-tuning is **accelerated training**. The study compared the number of epochs required to reach 90% of the maximum accuracy, and found an on average 17-fold speedup in fine-tuning (Figure 3)

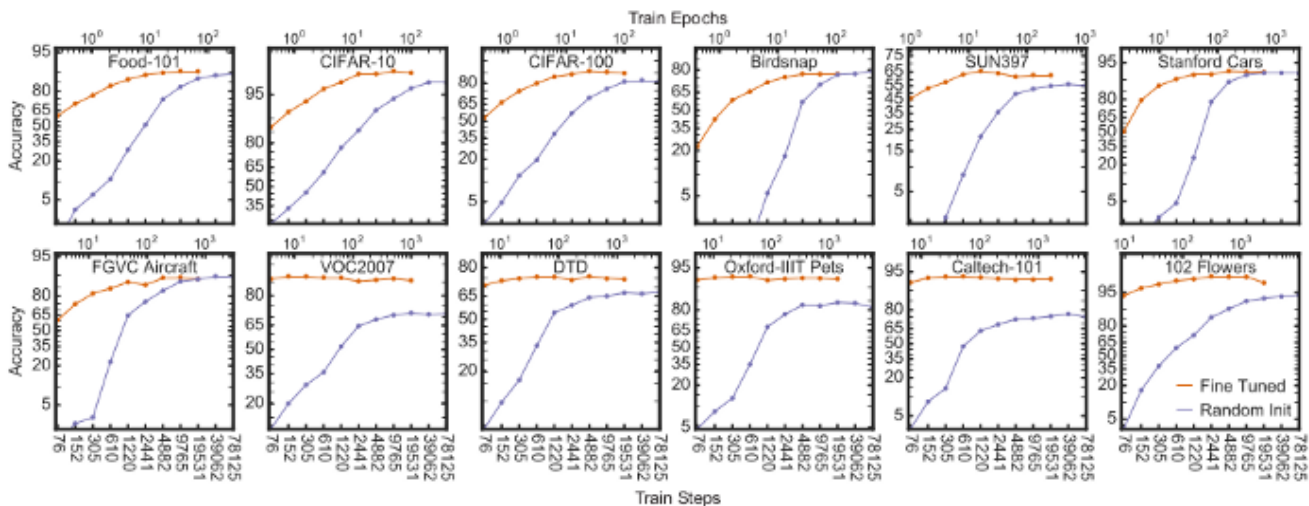


Figure 3. Fine-tuning (orange) needs a lot fewer epochs to reach high accuracy than training from scratch (blue).

Another reason to not train from scratch is it has poor performance when presented with insufficient amount of training samples (Figure 4). This suggests transfer learning is less prone to over-fitting if it can leverage knowledge learnt from a large, relevant task.

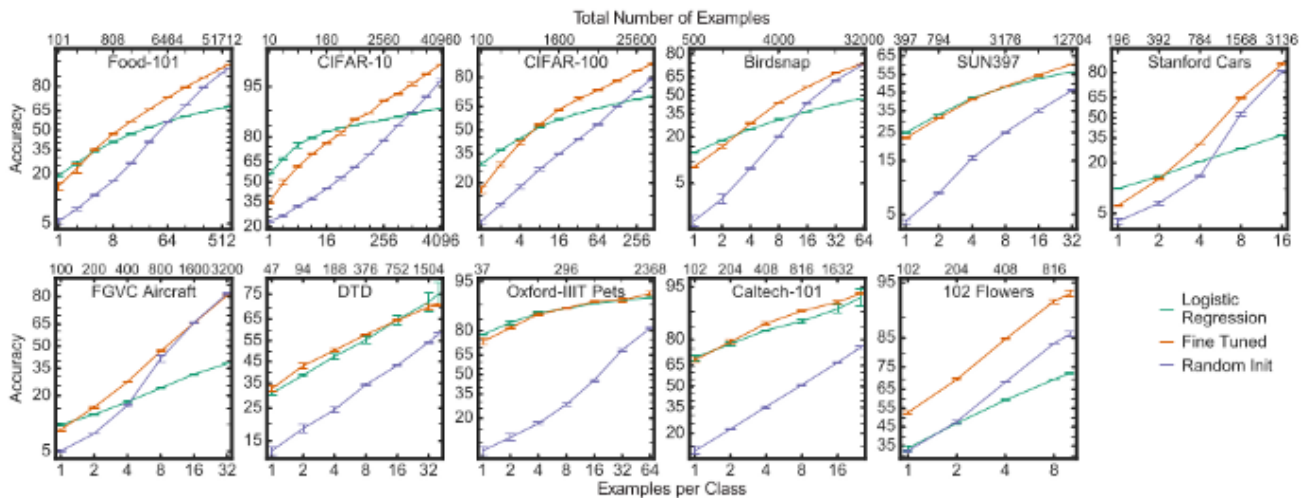


Figure 4. Training from scratch (blue) needs a lot more data to catch up with the performance of fine-tuning (orange). Also notice that fixed features (green) has certain advantage when only extremely small amount of data is available (far left side of the plots)

Summary

- Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch.
- TensorFlow Hub is a repository of trained machine learning models ready for fine-tuning and deployable anywhere.
- Video is simply a sequence of images referred to as *frames*. Precision of video therefore is a combination of image precision and also the number of frames captured per second (the frame rate).
- Whereas color images are passed to a neural network as a 4D tensor, video inputs are represented by a whopping 5D tensor.
- There are two types of transfer learning: use fixed features and fine-tuning. Both involve restoring weights from a pre-trained ImageNet model, and re-training the network for the new classes of interests.
- There is really no reason to prefer train from scratch over fine-tuning. Just consider the difference between the two: how weights are initialized. Fine-tuning uses weights from a pre-trained model and trained from scratch uses random weights.