CSCI E-25

Transfer Learning and Data Augmentation

Steve Elston

Introduction

Transfer learning and **data augmentation** are important approaches to paractical use of deep neural network models for computer vision. You will work with examples of each of these methods.

Transfer learning

Transfer learning employes deep neural network models which have been previously trained on large datasets. This pretraining can require enourmous computing resources, as well as massive datasets. These resources are often not available in practice. The dataset available for a particular prblem may be too small, or the computing resources may not be available for the project.

Most deep learning platforms have a module of pre-trained models. You can find an extensice list of optipns for Keras.

Instead, we can use a model with weights trained previously on other datasets. We stay that we **transfer** the learning from one task **learned** with some traning data to another task with different image characteristics. This process is known as **transfer learning**.

In most cases of transfer learning there are two major components of the model, a **backbone network** and a **head**. The pre-trained **backbone** network produces a feature map. A head, placed on the backbone, performs the task-specific learning. Examples of task-specific learning include:

- **Object classification:** Our goal for the exercises in this lesson.
- **Object detection:** Fine the objects in an image.
- **Semantic segmentation:** Detect and label the types of 'things' in an image.
- **Object tracking:** Track how the bjects in an series of images (a video) are moving.

There are several approaches to task-specific traning with transfer learning:

Frozen backbone network: The weights of the backbone network are frozen and the resulting feature map is used directly. The weights of the task-specific head are learned using case-specific data. This approach minimizes the amound of traning data requried, since only the weights of the head need be trained. This approach leads to methods known as **few shot training** for a specific case of a task. Accuracy may be sacrificed since the feature map is not optimized for the task.

Fine tune traning of the backbone network: Weights of the task specific head are learned using the case-speific data. At the same time, the weights of the backbone network are **fine-tuned** for the task. In many cases, only a few epochs are required for fine tuning. This second approaceh often provides better performance, since the feature map produced by the backbone is more likely to have features specific for the task. However, fine tuning of the backbone network may fail if there is insufficient data to effectively train the additional weights.

Data augmentation

Even with good pre-training, sufficient task-specific data may not be available even to learn the required weights using transfer learning. In such cases one can apply **data augmentation**. The process of data augmenation creats new traning samples from existing training images. The new image samples are created by **randomly** appling one or more **transformations** to the original image. The label of the transformed image is the same as the original image. Yet, given the randomly chosen transformations applied, the new image will have different characteristics. Further, since the transformations are random in nature, several new samples can be created from the same original image. Thus, augmented data will help tranin models to have better generalization.

Deep learning platforms include packages for standard random data augmentation. For example, in the Keras documentation you can find examples of applying random transformations to augment image data.

Examples of random transformations which can be used to augment image traning data include:

1. Random rotation.
2. Random translation along either or both axes.
3. Random cropping of the image followed by resizing to the original size.
4. Flipping the transformed image to create a mirror image.
5. Randomly adjusting the contrast of the image.
6. Adding Gaussian or other noise to the image.
7. Independently applying random brightness adjsutments to the histograms of the color channels.
8. Randomly down sampling followed by resizing.

Example Notebook

For these exercises you will use the Image classification via fine-tuning with Efficientnet Keras example notebook. This notebook uses the pre-trained **EfficientnetB0** model, the smallest model in the EfficientNet family introduced by Tan and Le, 2019. This version of the model is faster to train and faster at inference, but at a sacrifice in accuracy.

To run the notebook you will need a Google Colabratory account if you do not already have one. Log into your google account. Start on the Image classification via fine-tuning with Efficientnet page. Then, click the `View in Colab` tab to start the notebook in Colab (but do not execute yet!). Alternatively, you may want to run this notebook locally, if you have the resources.

The Dataset

The Stanford Dogs dataset contains over 20,000 images of 120 dog breeds. The goal for a classifier is to learn to classify dog breeds correctly.

Modifications to the Notebook

Before you attempt to run this notebook in Colab you must make two changes.

1. **Work around bugs in augmentation.** The example in the notebook uses TensorFlow data augmentation. A bug in TensorFlow versions > 2.8.x makes augmentation run extremely slowly and with many warnings. Therefore, you must roll-back the current version of TensorFlow in colab to version 2.8.3. To do so, add a code cell before any other import with the following code.

``` !pip uninstall tensorflow -y !pip install -U "tensorflow==2.8.3" import tensorflow print('TensorFlow version = ' + str(tensorflow.__version__)) ```

When you execute the notebook the above code will generate a significant amount of printed output. Check that the last two lines read as follows:

``` Successfully installed keras-2.8.0 keras-preprocessing-1.1.2 tensorboard-2.8.0 tensorflow-2.8.3 tensorflow-estimator-2.8.0 TensorFlow version = 2.8.3 ```

2. To obtain a better understanding of the learning of the model substitute the code shown below for the code in the cell following the `Training the Model from Scratch` code cell.

``` import matplotlib.pyplot as plt def plot_hist(hist): _,ax = plt.subplots(1,2, figsize = (12,6)) ax[0].plot(hist.history["accuracy"]) ax[0].plot(hist.history["val_accuracy"]) ax[0].set_title("model accuracy") ax[0].set_ylabel("accuracy") ax[0].set_xlabel("epoch") ax[0].legend(["train", "validation"], loc="upper left") ax[1].plot(hist.history["loss"]) ax[1].plot(hist.history["val_loss"]) ax[1].set_title("model loss") ax[1].set_ylabel("loss") ax[1].set_xlabel("epoch") ax[1].legend(["train", "validation"], loc="upper right") plt.show() plot_hist(hist) ```

Steps to submit notebook

Once you have successfully exectued your notebook in colab you must use `Save a copy in Drive` to save any of your work in your GoogleDrive account. You can then download the notebook from your GoogleDrive account and upload it into Canvas for submission.

Understanding the Code

Before proceeding, it is worthwhile to understand some key aspects of the code.

Importing the pre-trained model

Under the heading, *Keras implementation of EfficientNet*, code to import and configure EfficientNetB0 is described. Examine the code:

``` from tensorflow.keras.applications import EfficientNetB0 model = EfficientNetB0(include_top=False, weights='imagenet') ```

In transfer learning the feature map generating convolutional layers are pre-trained. These pre-trained layers constitued a **convolutional backbone**. A new head is added on top of the backbone and trained for the specific task. The argument `include_top=False` creates a model object without the head.

The `weights='imagenet'` argument creates a model object with weights learned by supervised learning with the large ImageNet dataset. These weights for the convolutional backbone layers for the creation of a rich feature maps without additional traning.

Load the dataset and resize the images.

The next several code cells import the Stanford Dogs dataset, and resizes the images to 253×253 pixels, as required by EfficientNetB0. The Stanford Dogs dataset comes with independently sampled train and test sets.

The code in the cell of the *Visualizing Data* subsection displays a sample of the dog images. Notice about the following about these images:

1. The are different crops and angles for the dog in each image.
2. The dog in the images have a variety of scales.
3. Some images have other objects.

Data augmentation

Very often only limited task-specific training data is available. In such cases, **augmentting** the training data can be an effictive preprocessing step. The code in the *Data augmentation* section performs a series of random data augmenation steps. The code in the second cell of this section displays a sample of agmentated verisons of one image.Examine these results.

> **Exercise 7-1:** Answer these questions:
>
> 1. Which augmentation methods are applied to the images?
> 2. Describe some of the ways you can observed the multiple-transformation agumentation manifest in the displayed samples?

> **Answers:**

> 1. There were applied four augmentation methods:

```
 - Rotation
 - Translation
 - Image flipping
 - Contrast adjustment
```

> 2. These are the transformations I perceive:

```
 1. Rotation 180°, Translation up
 2. Horizontal flipping, Translation down
 3. Rotation 180°
 4. Horizontal flipping, Rotation ~350°, Translation to the
right (although it could be a consequence of the rotation)
 5. Rotation 315°
 6. Vertical flipping, Rotation ~20°
 7. Rotation ~40°
 8. Horizontal flipping, Rotation ~340°
 9. Rotation ~40°, Translation up
```

> > *I do not see changes in contrast, although the code includes this transformation.*

Training the model

The remainder of the notebook contains code for traning the model by three different approaches. Examine the code and notice the three different traning approaches:

1. **Training from scratch:** In this case no pre-trained weights are used. The model weights are learned from scratch using the training data.
2. **Trainng the head with frozen backbone weight:** In this case, the weights of the classification head of the model are trained, with the backbone weights frozen. In other words, the feature map is created from the pretrained weights. The feature

map is used by the classifier head. The classifier uses weights learned from the training data. Notice in the code for the *Transfer learning from pre-trained weights* section of the notebook* the model object is created with the following line of code, excluding the head.

model = EfficientNetB0(include_top=False, input_tensor=x, weights="imagenet")

3. **Fine tuning weights:** In some caes, it is possible to improve model performance by **fine tuning** the weights of the backbone. The fine tuning can result in a feature map better suited to the feature specific to images used for the particular case. Once the model in the notebook is trained with the fronzen backbone weights, the weights of the trainable layers are made `trainable`, or unfrozen.

``` for layer in model.layers[-20:]: if not isinstance(layer, layers.BatchNormalization): layer.trainable = True ```

Executing the Code and Examining the Results

You are now ready to execute the code in the notebook. Do so, and allow the code to run to completion.

Warning!! Expect slow execution! It appears that the model is intended to be run using **Tensor Processing Units (TPUs)** rather than GPUs. However, configuring the environment to work correctly with TPUs is quite challenging. Further, a dedicated Google cloud storeage account (not GoogleDrive) appears to be required. The notebook will execute with the above modifications, but slowly. Over 2 minute per epoch is required for the untrained model. Models using transfer learning require about a minute or less per epoch.

The entire notebook can be executed from the menu at the top from `Runtime -> Run all`. As noted above, execution will take some time.

> **Exercise 7-2:** Examine the results of traning from scratch and answer these quesitons:
>
> 1. Based on the training loss and error rate, is the model continuing to learn over the epochs.
> 2. Based on the training and test losses and error rates, does the model show generalizatation as the the training epochs progress.

> **Answers:**

> 1. The model shows overfitting. This is evident when accuracy grows and loss decreases rapidly for the train dataset, while for the validation dataset is erratic and gets worst after a few epochs.

> 2. No, the model does not do a good job generalizing. In addition to my previous answer, test accuracy is erratic but stabilizes at a low level, without improving enough over epochs. Validation loss Is also erratic, but after a few epochs it increases. That means that the model does not generalize correctly.

> **Exercise 7-3:** Next, examine the results from the pretraned backbone (fronzen weight) model trained in the *Transfer learning from pre-trained weights* section of the notebook, then answer the following questions.
>
> 1. In terms of traning and test loss and accuracy, describe the progress of the training of the head?
> 2. What evidence is there that the model will generalize?

> **Answers:**

> 1. In this case the model is underfitting on the training data. The model does not have enough capacity to learn the training data, so it does not accurately generalize to unseen data. In this case, the validation accuracy is higher than the training accuracy because the model is not learning from the training data as well as it should be. However, the lower model loss on the validation dataset indicates that the model is still able to learn from the validation data, even if it cannot learn from the training data.

> 2. The evidence we can see that the model will generalize comes from the validation accuracy being higher than the training accuracy, while the model loss is considerably lower for the validation dataset than for the train dataset. This indicates that the model is not overfitting on the training data, so it should be able to generalize to unseen data.

> **Exercise 7-4:** Examine the results of the fine tuning withe the unfrozen backbone weights and answer these questions:
>
> 1. Comparing the validation loss and accuracy of the last epochs of the model with frozen backbone weights and the fine tuning with the unfrozen weights. Has the fine tuning improved the overall model performance and why?
> 2. Is there any additional fine-tune learning after the first epoch and why?

> **Answers:**

> 1. No. The fine tuning model is overfitting on the training data. That model is able to learn from the training data and is constantly improving (better accuracy and lower loss), but it is not generalizing well to unseen data. For that reason, the accuracy and loss remain constant on the validation dataset.

> 2. The accuracy and loss behavior indicate that the system learns from training data but does not generalizes well to unseen data.