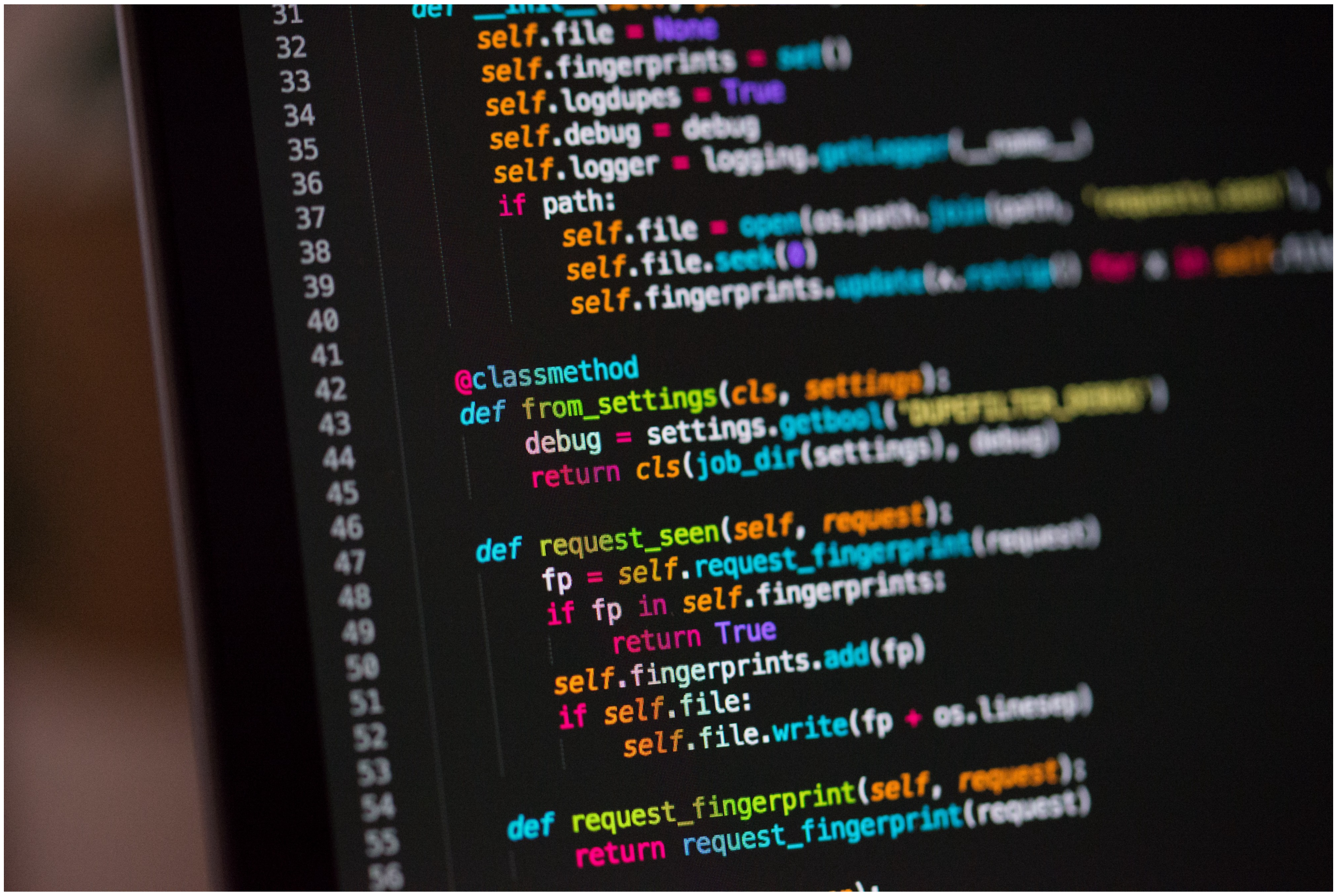
Transfer learning from pre-trained models

Pedro Marcelino
Oct 23, 2018 · 14 min read

How to solve any image classification problem quickly and easily

This article teaches you how to use transfer learning to solve image classification problems. A practical example using Keras and its pre-trained models is given for demonstration purposes.



The beauty of code by Chris Ried on Unsplash

. . .

Deep learning is fast becoming a key instrument in artificial intelligence applications (LeCun et al. 2015). For example, in areas such as computer vision, natural language processing, and speech recognition, deep learning has been producing remarkable results. Therefore, there is a growing interest in deep learning.

One of the problems where deep learning excels is **image classification** (Rawat & Wang 2017). The goal in image classification is to classify a specific picture according to a set of possible categories. A classic example of image classification is the identification of cats and dogs in a set of pictures (e.g. Dogs vs. Cats Kaggle Competition).

From a deep learning perspective, the image classification problem can be solved through **transfer learning**. Actually, several state-of-the-art results in image classification are based on transfer learning solutions (Krizhevsky et al. 2012, Simonyan & Zisserman 2014, He et al. 2016). A comprehensive review on transfer learning is provided by Pan & Yang (2010).

This article shows how to implement a transfer learning solution for image classification problems. The implementation proposed in this article is based on Keras (Chollet 2015), which uses the programming language Python. Following this implementation, you will be able to solve any image classification problem quickly and easily.

The article has been organised in the following way:

1. Transfer learning

2. Convolutional neural networks

. . .

1. Transfer learning

Transfer learning is a popular method in computer vision because it allows us to **build accurate models in a timesaving way** (Rawat & Wang 2017). With transfer learning, instead of starting the learning process from scratch, you start from patterns that have been learned when solving a different problem. This way you leverage previous learnings and avoid starting from scratch. Take it as the deep learning version of Chartres' expression 'standing on the shoulder of giants'.

In computer vision, transfer learning is usually expressed through the use of **pre-trained models**. A pre-trained model is a model that was trained on a large benchmark dataset to solve a problem similar to the one that we want to solve. Accordingly, due to the computational cost of training such models, it is common practice to import and use models from published literature (e.g. VGG, Inception, MobileNet). A comprehensive review of pre-trained models' performance on computer vision problems using data from the ImageNet (Deng et al. 2009) challenge is presented by Canziani et al. (2016).

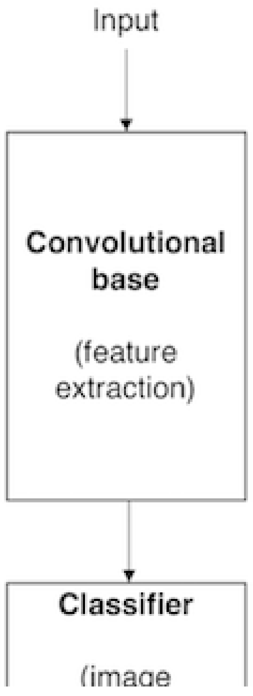
2. Convolutional neural networks

Several pre-trained models used in transfer learning are based on large **convolutional neural networks (CNN)** (Voulodimos et al. 2018). In general, CNN was shown to excel in a wide range of computer vision tasks (Bengio 2009). Its high performance and its easiness in training are two of the main factors driving the popularity of CNN over the last years.

A typical CNN has two parts:

1. **Convolutional base**, which is composed by a stack of convolutional and pooling layers. The main goal of the convolutional base is to generate features from the image. For an intuitive explanation of convolutional and pooling layers, please refer to Chollet (2017).
2. **Classifier**, which is usually composed by fully connected layers. The main goal of the classifier is to classify the image based on the detected features. A fully connected layer is a layer whose neurons have full connections to all activation in the previous layer.

Figure 1 shows the **architecture of a model based on CNN**. Note that this is a simplified version, which fits the purposes of this text. In fact, the architecture of this type of model is more complex than what we suggest here.



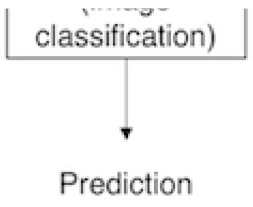


Figure 1. Architecture of a model based on CNN.

One important aspect of these deep learning models is that they can automatically learn **hierarchical feature representations**. This means that features computed by the first layer are general and can be reused in different problem domains, while features computed by the last layer are specific and depend on the chosen dataset and task. According to Yosinski et al. (2014), '*if first-layer features are general and last-layer features are specific, then there must be a transition from general to specific somewhere in the network*'. As a result, the convolutional base of our CNN — especially its lower layers (those who are closer to the inputs) — refer to general features, whereas the classifier part, and some of the higher layers of the convolutional base, refer to specialised features.

3. Repurposing a pre-trained model

When you're repurposing a pre-trained model for your own needs, you start by removing the original classifier, then you add a new classifier that fits your purposes, and finally you have to **fine-tune your model according to one of three strategies**:

1. **Train the entire model.** In this case, you use the architecture of the pre-trained model and train it according to your dataset. You're learning the model from scratch, so you'll need a large dataset (and a lot of computational power).

2. **Train some layers and leave the others frozen.** As you remember, lower layers refer to general features (problem independent), while higher layers refer to specific features (problem dependent). Here, we play with that dichotomy by choosing how much we want to adjust the weights of the network (a frozen layer does not change during training). Usually, if you've a small dataset and a large number of parameters, you'll leave more layers frozen to avoid overfitting.

By contrast, if the dataset is large and the number of parameters is small, you can improve your model by training more layers to the new task since overfitting is not an issue.

3. **Freeze the convolutional base.** This case corresponds to an extreme situation of the train/freeze trade-off. The main idea is to keep the convolutional base in its original form and then use its outputs to feed the classifier. You're using the pre-trained model as a fixed feature extraction mechanism, which can be useful if you're short on computational power, your dataset is small, and/or pre-trained model solves a problem very similar to the one you want to solve.

Figure 2 presents these three strategies in a schematic way.

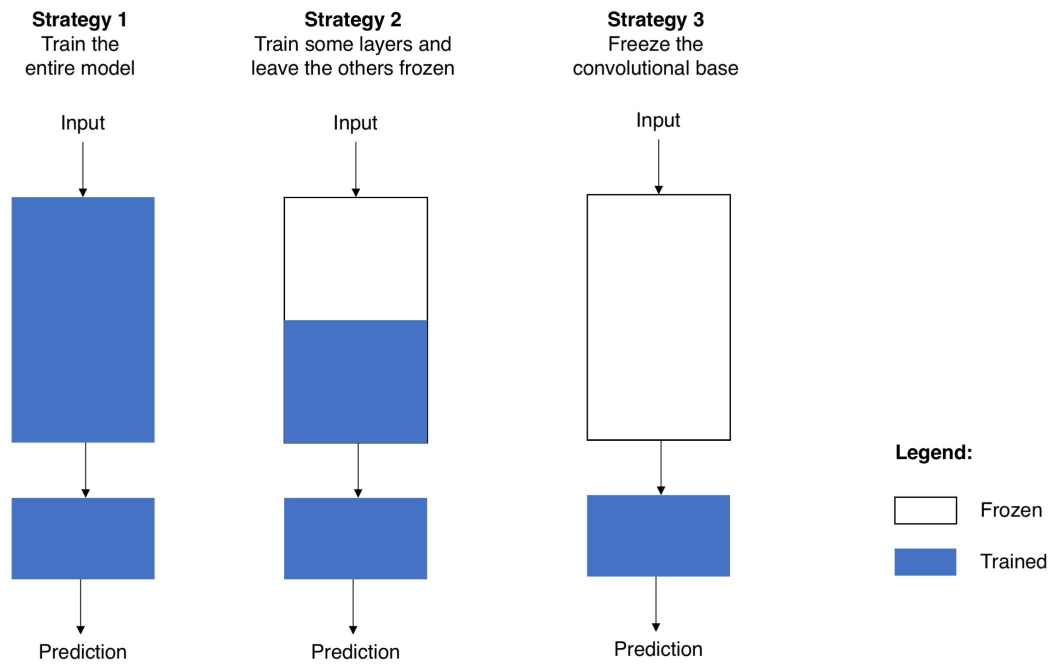


Figure 2. Fine-tuning strategies.

Unlike **Strategy 3**, whose application is **straightforward**, **Strategy 1** and **Strategy 2** require you to **be careful** with the learning rate used in the convolutional part. The learning rate is a hyper-parameter that controls how much you adjust the weights of your network. When you're using a pre-trained model based on CNN, it's smart to use a small learning rate because high learning rates increase the risk of losing previous knowledge. Assuming that the pre-trained model has been well trained, which is a fair assumption, keeping a small learning rate will ensure that you don't distort the CNN weights too soon and too much.

4. Transfer learning process

From a practical perspective, the entire transfer learning process can be summarised as follows:

1. **Select a pre-trained model**. From the wide range of pre-trained models that are available, you pick one that looks suitable for your problem. For example, if you're using Keras, you immediately have access to a set of models, such as VGG (Simonyan & Zisserman 2014), InceptionV3 (Szegedy et al. 2015), and ResNet5 (He et al. 2015). Here you can see all the models available on Keras.

2. **Classify your problem according to the Size-Similarity Matrix.** In Figure 3 you have 'The Matrix' that controls your choices. This matrix classifies your computer vision problem considering the size of your dataset and its similarity to the dataset in which your pre-trained model was trained. As a rule of thumb, consider that your dataset is small if it has less than 1000 images per class. Regarding dataset similarity, let common sense prevail. For example, if your task is to identify cats and dogs, ImageNet would be a similar dataset because it has images of cats and dogs. However, if your task is to identify cancer cells, ImageNet can't be considered a similar dataset.

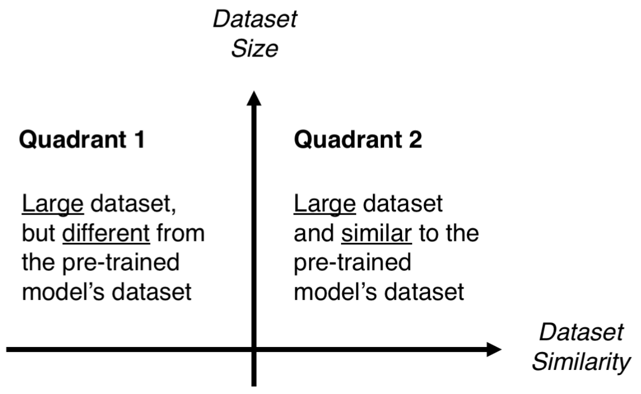
3. **Fine-tune your model.** Here you can use the Size-Similarity Matrix to guide your choice and then refer to the three options we mentioned before about repurposing a pre-trained model. Figure 4 provides a visual summary of the text that follows.

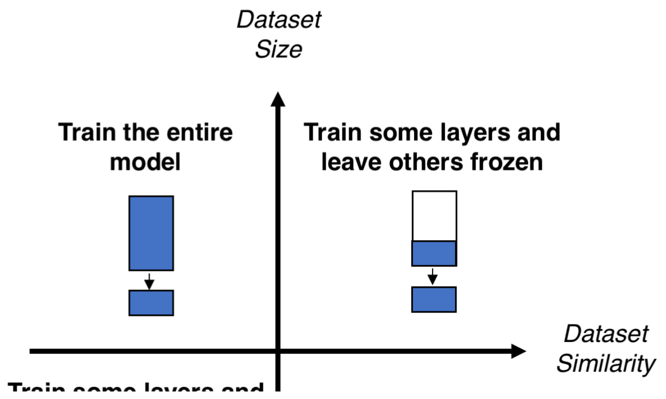
- **Quadrant 1**. Large dataset, but different from the pre-trained model's dataset. This situation will lead you to *Strategy 1*. Since you have a large dataset, you're able to train a model from scratch and do whatever you want. Despite the dataset dissimilarity, in practice, it can still be useful to initialise your model from a pre-trained model, using its architecture and weights.

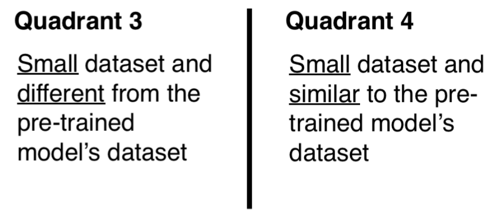
- **Quadrant 2.** Large dataset and similar to the pre-trained model's dataset. Here you're in la-la land. Any option works. Probably, the most efficient option is *Strategy 2*. Since we have a large dataset, overfitting shouldn't be an issue, so we can learn as much as we want. However, since the datasets are similar, we can save ourselves from a huge training effort by leveraging previous knowledge. Therefore, it should be enough to train the classifier and the top layers of the convolutional base.

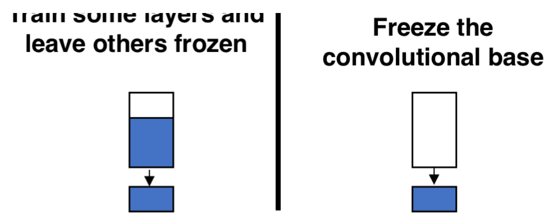
- **Quadrant 3.** Small dataset and different from the pre-trained model's dataset. This is the 2–7 off-suit hand of computer vision problems. Everything is against you. If complaining is not an option, the only hope you have is *Strategy 2*. It will be hard to find a balance between the number of layers to train and freeze. If you go to deep your model can overfit, if you stay in the shallow end of your model you won't learn anything useful. Probably, you'll need to go deeper than in Quadrant 2 and you'll need to consider data augmentation techniques (a nice summary on data augmentation techniques is provided here).

- **Quadrant 4.** Small dataset, but similar to the pre-trained model's dataset. I asked Master Yoda about this one he told me that '*be the best option, Strategy 3 should*'. I don't know about you, but I don't underestimate the Force. Accordingly, go for *Strategy 3*. You just need to remove the last fully-connected layer (output layer), run the pre-trained model as a fixed feature extractor, and then use the resulting features to train a new classifier.


Dataset Size
Quadrant 1
Large dataset, but different from the pre-trained model's dataset
Quadrant 2
Large dataset and similar to the pre-trained model's dataset
Dataset Similarity




Quadrant 3
Small dataset and different from the pre-trained model's dataset
Quadrant 4
Small dataset and similar to the pre-trained model's dataset



Figures 3 and 4. Size-Similarity matrix (left) and decision map for fine-tuning pre-trained models (right).

5. Classifiers on top of deep convolutional neural networks

As mentioned before, **models for image classification** that result from a transfer learning approach **based on pre-trained convolutional neural networks** are usually composed of **two parts**:

1. **Convolutional base**, which performs feature extraction.
2. **Classifier**, which classifies the input image based on the features extracted by the convolutional base.

Since in this section we focus on the classifier part, we must start by saying that different approaches can be followed to build the classifier. Some of the most popular are:

1. **Fully-connected layers.** For image classification problems, the standard approach is to use a stack of fully-connected layers followed by a softmax activated layer (Krizhevsky et al. 2012, Simonyan & Zisserman 2014, Zeiler & Fergus 2014). The softmax layer outputs the probability distribution over each possible class label and then we just need to classify the image according to the most probable class.
2. **Global average pooling.** A different approach, based on global average pooling, is proposed by Lin et al. (2013). In this approach, instead of adding fully connected layers on top of the convolutional base, we add a global average pooling layer and feed its output directly into the softmax activated layer. Lin et

al. (2013) provides a detailed discussion on the advantages and disadvantages of this approach.

3. **Linear support vector machines.** Linear support vector machines (SVM) is another possible approach. According to Tang (2013), we can improve classification accuracy by training a linear SVM classifier on the features extracted by the convolutional base. Further details about the advantages and disadvantages of the SVM approach can be found in the paper.

6. Example

In this example, we will see **how each of these classifiers can be implemented in a transfer learning solution for image classification**. According to Rawat and Wang (2017), '*comparing the performance of different classifiers on top of deep convolutional neural networks still requires further investigation and thus makes for an interesting research direction*'. So it will be interesting to see how each classifier performs in a standard image classification problem.

You can find the full code of this example on my GitHub page.

6.1. Prepare data

In this example, we will use a smaller version of the original dataset. This will allow us to run the models faster, which is great for people who have limited computational power (like me).

To build a smaller version of the dataset, we can adapt the code provided by Chollet (2017) as shown in Code 1.

```
# Create smaller dataset for Dogs vs. Cats
import os, shutil

original_dataset_dir = '/Users/macbook/dogs_cats_dataset/train/'

base_dir = '/Users/macbook/book/dogs_cats/data'
if not os.path.exists(base_dir):
    os.mkdir(base_dir)
```

```

# Create directories
train_dir = os.path.join(base_dir,'train')
if not os.path.exists(train_dir):
    os.mkdir(train_dir)
validation_dir = os.path.join(base_dir,'validation')
if not os.path.exists(validation_dir):
    os.mkdir(validation_dir)
test_dir = os.path.join(base_dir,'test')
if not os.path.exists(test_dir):
    os.mkdir(test_dir)

train_cats_dir = os.path.join(train_dir,'cats')
if not os.path.exists(train_cats_dir):
    os.mkdir(train_cats_dir)

train_dogs_dir = os.path.join(train_dir,'dogs')
if not os.path.exists(train_dogs_dir):
    os.mkdir(train_dogs_dir)

validation_cats_dir = os.path.join(validation_dir,'cats')
if not os.path.exists(validation_cats_dir):
    os.mkdir(validation_cats_dir)

validation_dogs_dir = os.path.join(validation_dir, 'dogs')
if not os.path.exists(validation_dogs_dir):
    os.mkdir(validation_dogs_dir)

test_cats_dir = os.path.join(test_dir, 'cats')
if not os.path.exists(test_cats_dir):
    os.mkdir(test_cats_dir)

test_dogs_dir = os.path.join(test_dir, 'dogs')
if not os.path.exists(test_dogs_dir):
    os.mkdir(test_dogs_dir)

# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(100)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)
```

```python

# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(200, 250)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(250,300)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)

# Copy first 1000 dog images to train_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(100)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to validation_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(200,250)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)

# Copy next 500 dog images to test_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(250,300)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)

```

The convolutional base will be used to extract features. These features will feed the classifiers that we want to train so that we can identify if images have dogs or cats.

Once again, the code provided by Chollet (2017) is adapted. Code 2 shows the code used.

```
# Extract features
import os, shutil
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 32

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 7, 7, 512))  # Must be equal to the output of
    labels = np.zeros(shape=(sample_count))
    # Preprocess data
    generator = datagen.flow_from_directory(directory,
                                            target_size=(img_width,img_height),
                                            batch_size = batch_size,
                                            class_mode='binary')
    # Pass data through convolutional base
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size: (i + 1) * batch_size] = features_batch
        labels[i * batch_size: (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            break
    return features, labels

train_features, train_labels = extract_features(train_dir, train_size)  # Agree with our sm
validation_features, validation_labels = extract_features(validation_dir, validation_size)
test_features, test_labels = extract_features(test_dir, test_size)
```

6.3. Classifiers

6.3.1. Fully-connected layers

The first solution that we present is based on fully-connected layers. This classifier adds a stack of fully-connected layers that is fed by the features extracted from the convolutional base.

To keep it simple (and fast), we will use the solution proposed by Chollet (2018) with slight modifications. In particular, we will use the Adam optimizer instead of the RMSProp because Stanford says so (what a beautiful *argumentum ad verecundiam*).

Code 3 shows the code used, while Figures 5 and 6 present the learning curves.

```
# Define model
from keras import models
from keras import layers
from keras import optimizers

epochs = 100

model = models.Sequential()
model.add(layers.Flatten(input_shape=(7,7,512)))
model.add(layers.Dense(256, activation='relu', input_dim=(7*7*512)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()

# Compile model
model.compile(optimizer=optimizers.Adam(),
              loss='binary_crossentropy',
              metrics=['acc'])

# Train model
history = model.fit(train_features, train_labels,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_data=(validation_features, validation_labels))
```

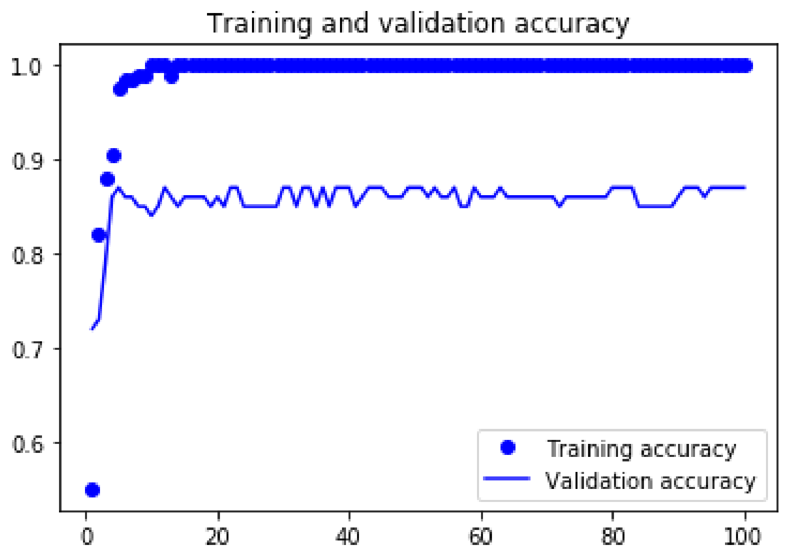


Figure 5. Accuracy of the fully connected layers solution.

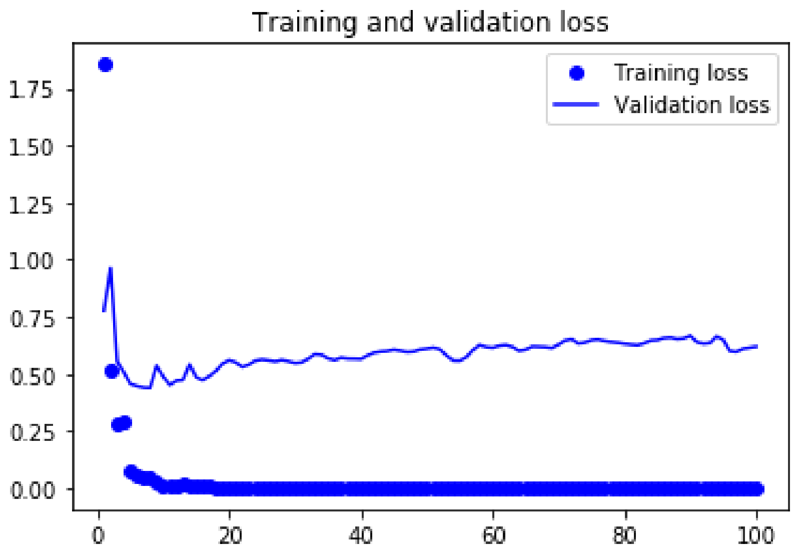


Figure 6. Loss of the fully connected layers solution.

Brief discussion of results:

1. Validation accuracy is around 0.85, which is encouraging given the size of the dataset.
2. The model strongly overfits. There's a big gap between the training and the validation curves.
3. Since we already used dropout, we should increase the size of the dataset to improve the results.

6.3.2. Global average pooling

The difference between this case and the previous one is that, instead of adding a stack of fully-connected layers, we will add a global average pooling layer and feed its output into a sigmoid activated layer.

Note that we are talking about a sigmoid activated layer instead of a softmax one, which is what is recommended by Lin et al. (2013). We are changing to the sigmoid activation because in Keras, to perform binary classification, you should use *sigmoid* activation and *binary_crossentropy* as the loss (Chollet 2017). Therefore, it was necessary to do this small modification to the original proposal of Lin et al. (2013).

Code 4 shows the code to build the classifier. Figure 7 and 8 show the resulting learning curves.

```
# Define model
from keras import models
from keras import layers
from keras import optimizers

epochs = 100

model = models.Sequential()
model.add(layers.GlobalAveragePooling2D(input_shape=(7,7,512)))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()

# Compile model
model.compile(optimizer=optimizers.Adam(),
              loss='binary_crossentropy',
              metrics=['acc'])

# Train model
history = model.fit(train_features, train_labels,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_data=(validation_features, validation_labels))
```

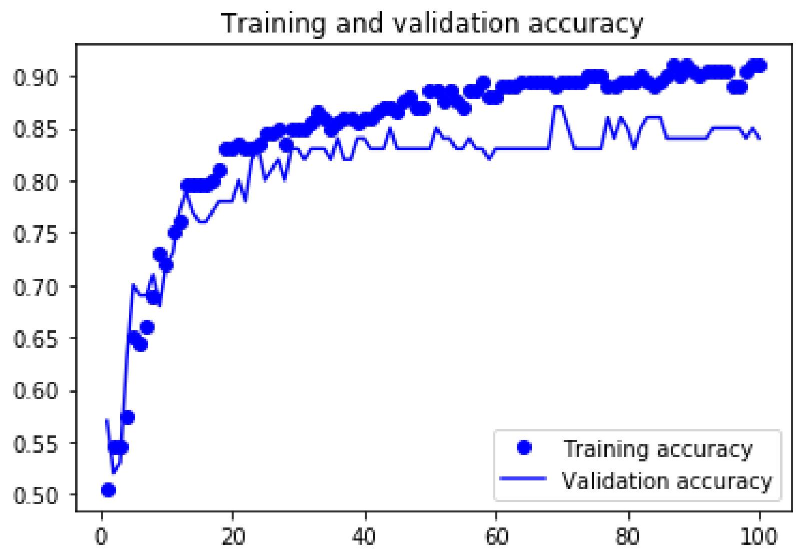


Figure 7. Accuracy of the global average pooling solution.

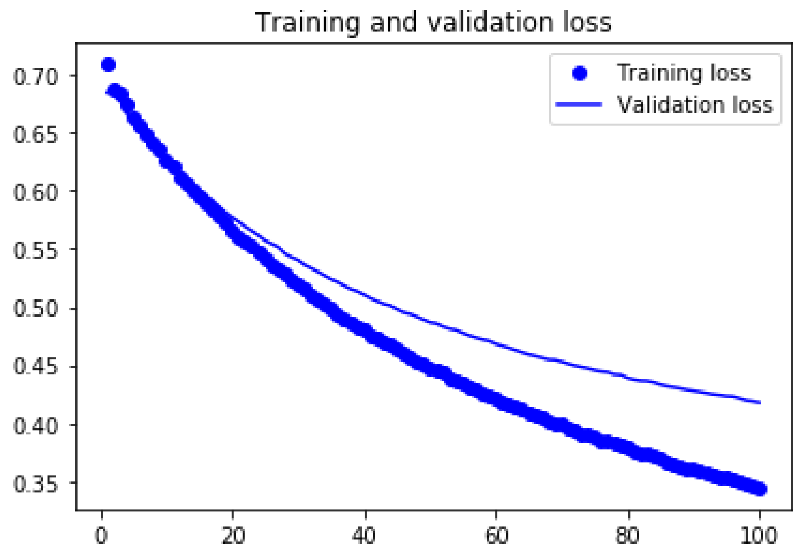


Figure 8. Loss of the global average pooling solution.

Brief discussion of results:

1. Validation accuracy is similar to the one resulting from the fully-connected layers solution.
2. The model doesn't overfit as much as in the previous case.
3. The loss function is still decreasing when the model stops training. Probably, it is possible to improve the model by increasing the number of epochs.

6.3.3 Linear support vector machines

In this case, we will train a linear support vector machines (SVM) classifier on the features extracted by the convolutional base.

To train this classifier, a traditional machine learning approach is preferable. Consequently, we will use k-fold cross-validation to estimate the error of the classifier. Since k-fold cross-validation will be used, we can concatenate the train and the validation sets to enlarge our training data (we keep the test set untouched, as we did in the previous cases). Code 5 shows how data was concatenated.

```
# Concatenate training and validation sets
svm_features = np.concatenate((train_features, validation_features))
svm_labels = np.concatenate((train_labels, validation_labels))
```

svm_data_concatenation hosted with ♡ by GitHub view raw

Code 5. Data concatenation.

Finally, we must be aware that the SVM classifier has one hyperparameter. This hyperparameter is the penalty parameter C of the error term. To optimize the choice of this hyperparameter, we will use exhaustive grid search. Code 6 presents the code used to build this classifier, while Figure 9 illustrates the learning curves.

```
# Build model
import sklearn
from sklearn.cross_validation import train_test_split
from sklearn.grid_search import GridSearchCV
from sklearn.svm import LinearSVC

X_train, y_train = svm_features.reshape(300,7*7*512), svm_labels

param = [{
          "C": [0.01, 0.1, 1, 10, 100]
         }]

svm = LinearSVC(penalty='l2', loss='squared_hinge')  # As in Tang (2013)
clf = GridSearchCV(svm, param, cv=10)
clf.fit(X_train, y_train)
```

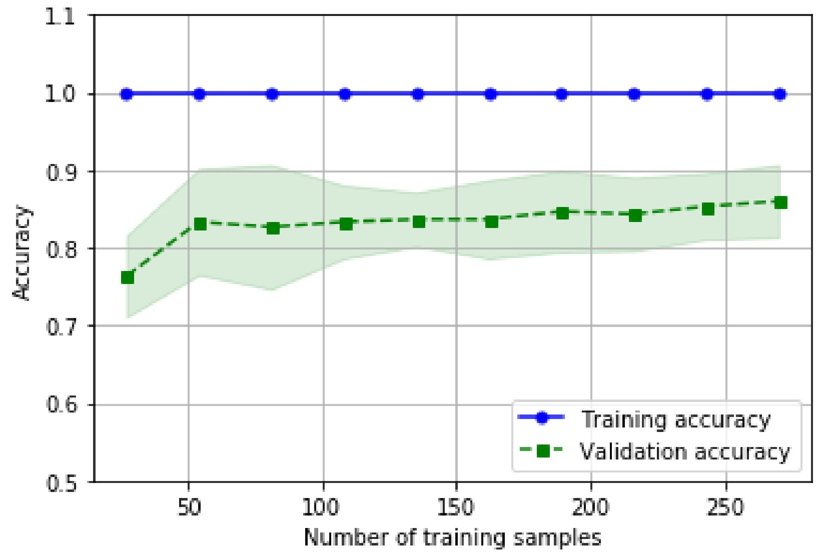


Figure 9. Accuracy of the linear SVM solution.

Brief discussion of results:

1. Model's accuracy is around 0.86, which is similar to the accuracy of the previous solutions.

2. Overfitting is around the corner. Moreover, the training accuracy is always 1.0, which is not usual and can be interpreted as a sign of overfitting.

3. The accuracy of the model should increase with the number of training samples. However, that doesn't seem to happen. This may be due to overfitting. It would be interesting to see how the model reacts when the dataset increases.

7. Summary

In this article, we:

- Presented the concepts of transfer learning, convolutional neural networks, and pre-trained models.

- Defined the basic fine-tuning strategies to repurpose a pre-trained model.

- Described a structured approach to decide which fine-tuning strategy should be used, based on the size and similarity of the dataset.
- Introduced three different classifiers that can be used on top of the features extracted from the convolutional base.
- Provided a end-to-end example on image classification for each of the three classifiers presented in this article.

I hope that you feel motivated to start developing your deep learning projects on computer vision. This is a great field of study and new exciting findings are coming out everyday.

I'd be glad to help you, so let me know if you have any questions or improvement suggestions!

8. References

1. Bengio, Y., 2009. Learning deep architectures for AI. Foundations and trends in Machine Learning, 2(1), pp.1–127.

2. Canziani, A., Paszke, A. and Culurciello, E., 2016. An analysis of deep neural network models for practical applications. arXiv preprint arXiv:1605.07678.

3. Chollet, F., 2015. Keras.

4. Chollet, F., 2017. Deep learning with python. Manning Publications Co..

5. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K. and Fei-Fei, L., 2009, June. Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on (pp. 248–255). Ieee.

6. He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770–778).

7. Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097–1105).

8. LeCun, Y., Bengio, Y. and Hinton, G., 2015. Deep learning. nature, 521(7553), p.436.

9. Lin, M., Chen, Q. and Yan, S., 2013. Network in network. arXiv preprint arXiv:1312.4400.

10. Pan, S.J. and Yang, Q., 2010. A survey on transfer learning. IEEE Transactions on knowledge and data engineering, 22(10), pp.1345–1359.

11. Rawat, W. and Wang, Z., 2017. Deep convolutional neural networks for image classification: A comprehensive review. Neural computation, 29(9), pp.2352–2449.

12. Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

13. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818–2826).

14. Tang, Y., 2013. Deep learning using linear support vector machines. arXiv preprint arXiv:1306.0239.

15. Voulodimos, A., Doulamis, N., Doulamis, A. and Protopapadakis, E., 2018. Deep learning for computer vision: A brief review. Computational intelligence and neuroscience, 2018.

16. Yosinski, J., Clune, J., Bengio, Y. and Lipson, H., 2014. How transferable are features in deep neural networks?. In Advances in neural information processing systems (pp. 3320–3328).

17. Zeiler, M.D. and Fergus, R., 2014, September. Visualizing and understanding convolutional networks. In European conference on computer vision (pp. 818–833). Springer, Cham.

Acknowledgments

Thanks to João Coelho for reading drafts of this.

. . .

You can find more about me and my projects at pmarcelino.com. Also, you can sign up for my newsletter to receive my latest updates on Humans, Machines, and Science.

Machine Learning Deep Learning Convolutional Network Data Science

Towards Data Science