Backpropagation

For a deep neural network, the loss function that should be minimized during training is a complex composite function of the weights and biases parameterizing the layers of the network. This requires computing the gradient of the function, i.e. the partial derivatives with respect to all its variables – in a deep network there can easily be millions of these variables.

The gradient can be calculated separately for each successive layer, and by making use of the chain rule of differentiation. The partial derivatives within any single layer are relatively easy to calculate, since they only involve a linear weighting and the activation function – and both of these are easily differentiated. The total gradient can then be calculated by multiplying the derivatives of all the layers, evaluated at their respective input values (these are available, since they are computed and stored during the feed-forward phase).

The computation of the gradient proceeds in the reverse order: starting from the output layer, and ending up with the input layer – this is referred to as *backpropagation*. During the training phase, the information flows alternately forward and backward through the neural network.

ConvNet data preprocessing

The input data for ConvNets are image files, which typically are situated on a drive as JPEG files. However, the data must be fed into the network in suitable-sized batches of 3D NumPy tensors. The necessary conversion is done with specially constructed Python generators, found in `keras.preprocessing.image` . In classification problems, the image files must first be divided in separate subdirectories (two in binary classification problems) depending on which class they belong to.

Example of setting up data batch generators for training and validation.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir, target_size=(150,150),
        batch_size=16, class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir, target_size=(150,150),
        batch_size=16, class_mode='binary')
```

Training with generators

The model can be trained by using the the `fit` method – it accepts generators as arguments. The user needs to supply both the number of epochs and the number of batches that are to be drawn during each epoch (`steps_per_epoch`).

It is also possible to pass a validation generator as a `validation_data` argument. Here also the number of drawn batches must be given separately.

Fitting the model to training data, with validation included.

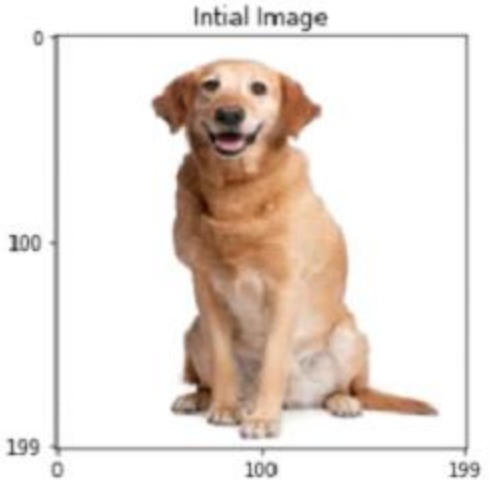
Saving models after training is a sensible thing to do.

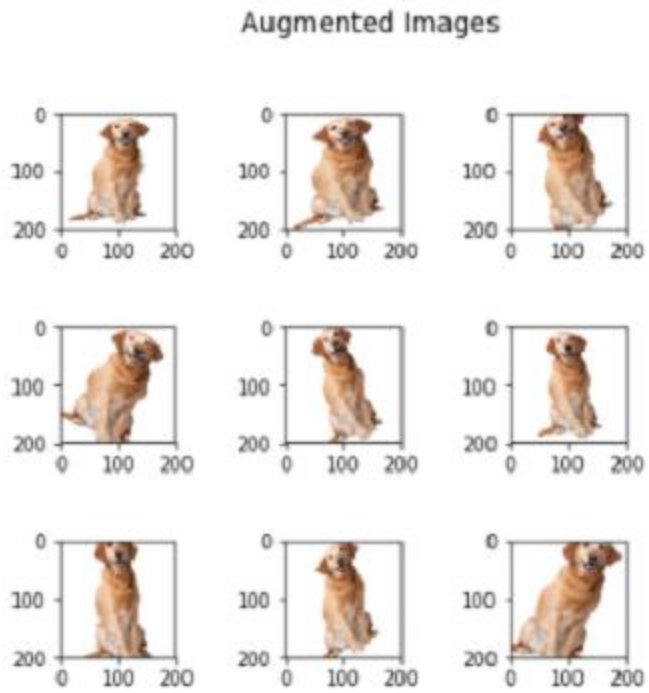
```
history = model.fit(train_generator,
        steps_per_epoch=100,
        epochs=30,
        validation_data=validation_generator,
        validation_steps=50)

model.save('my_model.h5')
```

Data augmentation

With small datasets, overfitting usually becomes an important problem. **Data augmentation** addresses this problem by artificially enlargening the amount of available data.

In data augmentation, the sample images are subjected to a set of various possible transformations (translation, rotation, zooming, etc.) before feeding them to the network. Because of the added variation, the specific features of single images become less prominent. The model is then forced to concentrate on more general features.





Implementing data augmentation

In Keras, data augmentation can be implemented easily when instantiating the training-data generator. Note that *only the training data should be augmented* – never the validation data.

Example showing how to use some of the many available transformations for data augmentation. See Keras documentation for more options and details.

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

Pretrained ConvNets

Rather than training an entirely new ConvNet from scratch, it is sometimes useful to use a pretrained network. Pretrained networks and their parameters have been saved after training with huge datasets (such as ImageNet). Especially the bottom layers in such networks contain low-level features that are encountered in many kinds of different images, and thus useful in many different contexts.

The *convolutional base* (without the top classifier) of a pretrained ConvNet can be used to produce new representations of the input data, by running the input samples through it – using the `predict` method of the base. The resulting NumPy tensors are then flattened and used for training a new dense (fully-connected) classifier.

The problem with the above approach is that it doesn't allow for data augmentation. To include that, it is possible to train the entire model (conv-base + classifier) together, with the parameters of the convolutional base freezed. This, however, is *very* time-consuming.

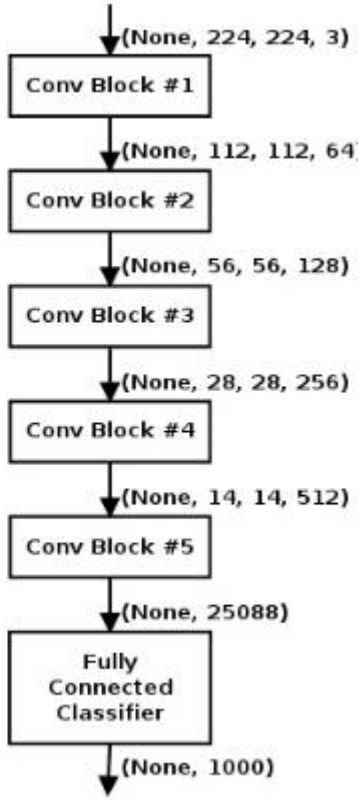
Pretraining example: VGG16

One possible choice for a pretrained ConvNet is the VGG16 model, which can be accessed directly from Keras. Below is an example of instantiating its convolutional base (without classifier).

```
from tensorflow.keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
        include_top=False,
        input_shape=(150,150,3))
```

Keras VGG-16 Model

```
( 0, 'input_6',      (None, 224, 224, 3))
( 1, 'block1_conv1', (None, 224, 224, 64))
( 2, 'block1_conv2', (None, 224, 224, 64))
( 3, 'block1_pool',  (None, 112, 112, 64))
( 4, 'block2_conv1', (None, 112, 112, 128))
( 5, 'block2_conv2', (None, 112, 112, 128))
( 6, 'block2_pool',  (None, 56, 56, 128))
( 7, 'block3_conv1', (None, 56, 56, 256))
( 8, 'block3_conv2', (None, 56, 56, 256))
( 9, 'block3_conv3', (None, 56, 56, 256))
(10, 'block3_pool',  (None, 28, 28, 256))
(11, 'block4_conv1', (None, 28, 28, 512))
(12, 'block4_conv2', (None, 28, 28, 512))
(13, 'block4_conv3', (None, 28, 28, 512))
(14, 'block4_pool',  (None, 14, 14, 512))
(15, 'block5_conv1', (None, 14, 14, 512))
(16, 'block5_conv2', (None, 14, 14, 512))
(17, 'block5_conv3', (None, 14, 14, 512))
(18, 'block5_pool',  (None, 7, 7, 512))
(19, 'flatten',      (None, 25088))
(20, 'fc1',          (None, 4096))
(21, 'fc2',          (None, 4096))
(22, 'predictions',  (None, 1000))
```



Feature extraction with pretrained base

```
datagen = ImageDataGenerator(rescale=1./255)
batch_size = 16

def extract_features(directory, sample_count):
        features = np.zeros(shape=(sample_count, 4, 4, 512))
        labels = np.zeros(shape=(sample_count))
        generator = datagen.flow_from_directory(
                directory,
                target_size=(150,150),
                batch_size=batch_size,
                class_mode='binary')
        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, 1000)
validation_features, validation_labels = extract_features(validation_dir, 500)
```