Advances in CNN - Part B

Instructor: Santosh Chapaneri

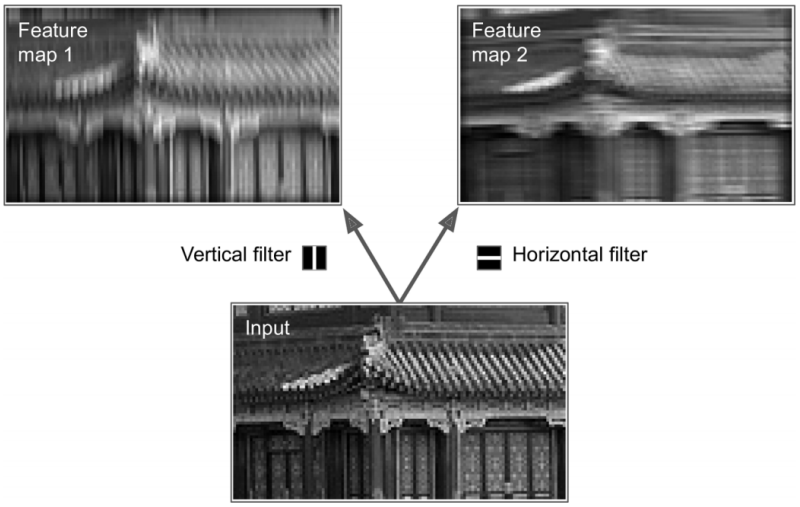
RK University, Rajkot

June 2021

In [1]:
```
import sklearn
import tensorflow as tf
from tensorflow import keras
import numpy as np
import os
np.random.seed(45)
tf.random.set_seed(45)
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

In [2]:
```
def plot_image(image):
    plt.imshow(image, cmap="gray", interpolation="nearest")
    plt.axis("off")

def plot_color_image(image):
    plt.imshow(image, interpolation="nearest")
    plt.axis("off")
```

What is Convolution?

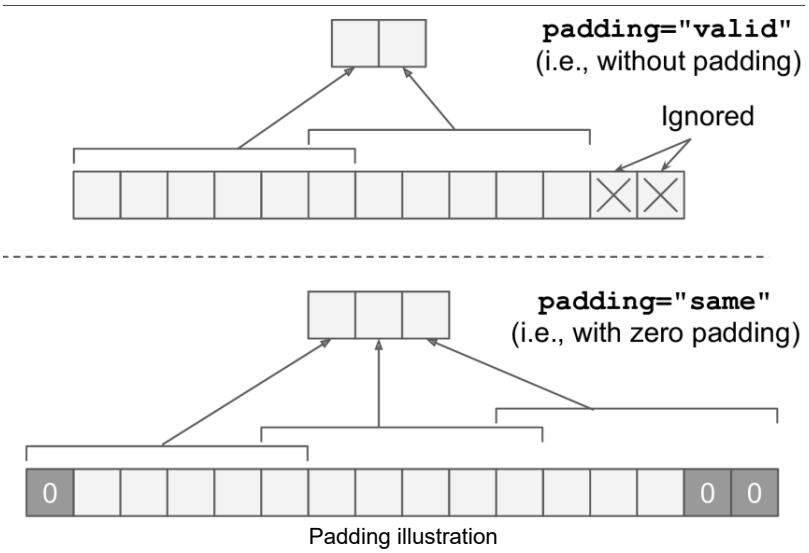


Convolution illustration

In [3]:
```
from sklearn.datasets import load_sample_image
import numpy as np

# Load sample images
china = load_sample_image("china.jpg") / 255.0
flower = load_sample_image("flower.jpg") / 255.0

images = np.array([china, flower])

batch_size, height, width, channels = images.shape
```

In [4]:
```
print(batch_size)
print(height, width, channels)
```

```
2
427 640 3
```



Padding illustration

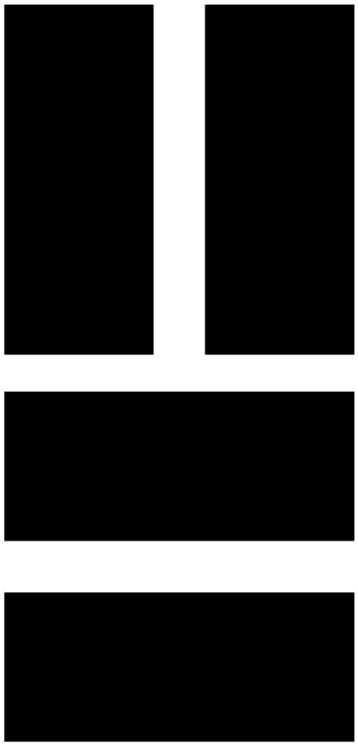
In [5]:
```
# Create two 7 × 7 filters (one with a vertical white line in the middle, and
# the other with a horizontal white line in the middle).
filters = np.zeros(shape = (7, 7, channels, 2), dtype = np.float32)

filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

outputs = tf.nn.conv2d(images, filters, strides = 1, padding = "SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.axis("off")
plt.show()
```

In [6]:
```
for image_index in (0, 1):
    for feature_map_index in (0, 1):
        plt.subplot(2, 2, image_index * 2 + feature_map_index + 1)
        plot_image(outputs[image_index, :, :, feature_map_index])

plt.show()
```

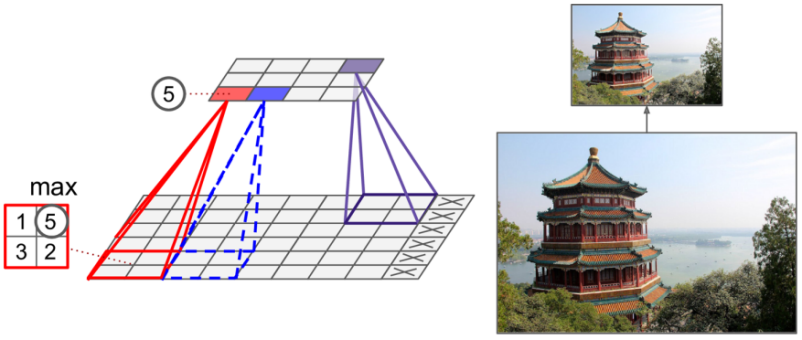
In [7]:
```
def crop(images):
    return images[150:220, 130:250]
```

In [8]:
```
plot_image(crop(images[0, :, :, 0]))
plt.show()

for feature_map_index, filename in enumerate(["china_vertical", "china_horizontal"]):
    plot_image(crop(outputs[0, :, :, feature_map_index]))
    plt.show()
```

In [9]:
```
plot_image(filters[:, :, 0, 0])
plt.show()

plot_image(filters[:, :, 0, 1])
plt.show()
```



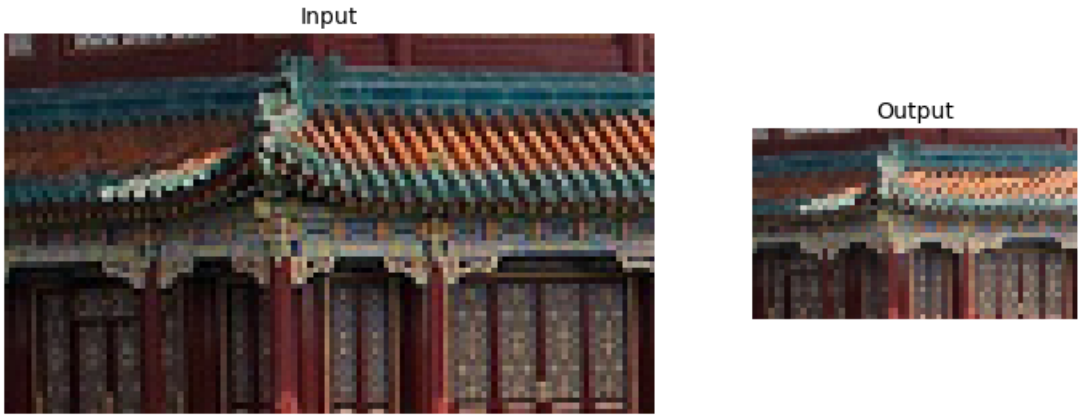
Pooling



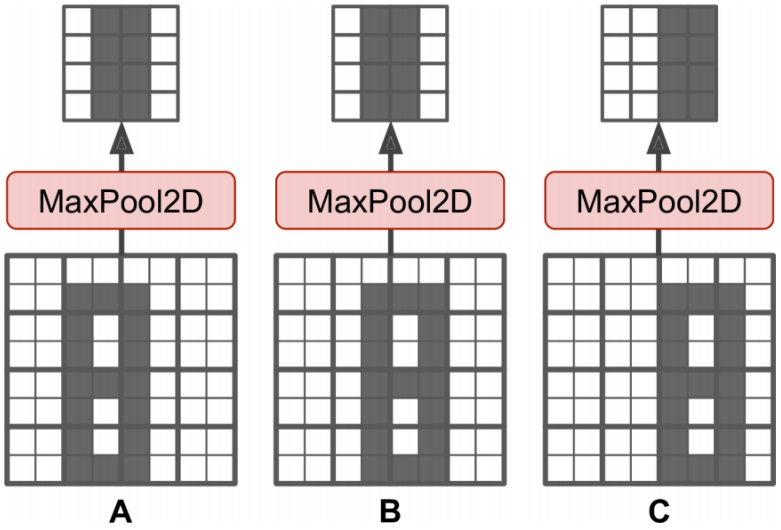
Pooling illustration

In [10]:
```
max_pool = keras.layers.MaxPool2D(pool_size=2)

cropped_images = np.array([crop(image) for image in images], dtype=np.float32)

output = max_pool(cropped_images)
```

In [11]:
```
fig = plt.figure(figsize=(12, 8))
gs = mpl.gridspec.GridSpec(nrows=1, ncols=2, width_ratios=[2, 1])

ax1 = fig.add_subplot(gs[0, 0])
ax1.set_title("Input", fontsize=14)

ax1.imshow(cropped_images[0])  # plot the 1st image
ax1.axis("off")
ax2 = fig.add_subplot(gs[0, 1])
ax2.set_title("Output", fontsize=14)

ax2.imshow(output[0])  # plot the output for the 1st image
ax2.axis("off")
plt.show()
```



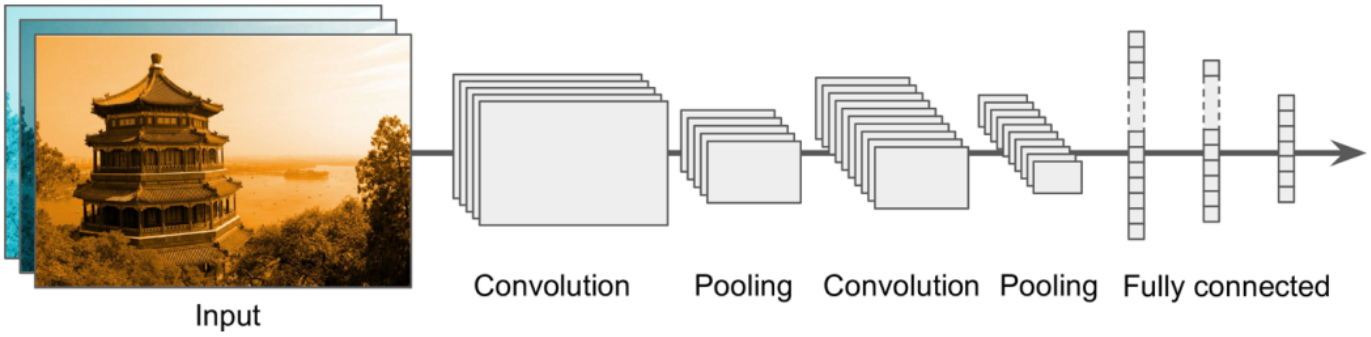
- Max pooling layer introduces some level of invariance to small translations:



Pooling invariance to small translations

- Asssume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2 × 2 kernel and stride 2.
- Images B and C are the same as image A, but shifted by one and two pixels to the right. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means.
- For image C, the output is different: it is shifted one pixel to the right (but there is still 75% invariance).
- By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale.

CNN Architectures



Typical CNN

Typical CNN

In [12]:
```
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()

X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]

X_mean = X_train.mean(axis=0, keepdims=True)
X_std = X_train.std(axis=0, keepdims=True) + 1e-7
X_train = (X_train - X_mean) / X_std
X_valid = (X_valid - X_mean) / X_std
X_test = (X_test - X_mean) / X_std

X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx
1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx
3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1
-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3
-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

In [13]:
```
from functools import partial

DefaultConv2D = partial(keras.layers.Conv2D,
                        kernel_size = 3, activation = 'relu', padding = "SAME")

model = keras.models.Sequential([
    DefaultConv2D(filters = 64, kernel_size = 7, input_shape = [28, 28, 1]),
    keras.layers.MaxPooling2D(pool_size = 2),
    DefaultConv2D(filters = 128),
    DefaultConv2D(filters = 128),
    keras.layers.MaxPooling2D(pool_size = 2),
    DefaultConv2D(filters = 256),
    DefaultConv2D(filters = 256),
    keras.layers.MaxPooling2D(pool_size = 2),
    keras.layers.Flatten(),
    keras.layers.Dense(units = 128, activation = 'relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units = 64, activation = 'relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units = 10, activation = 'softmax'),
])
```

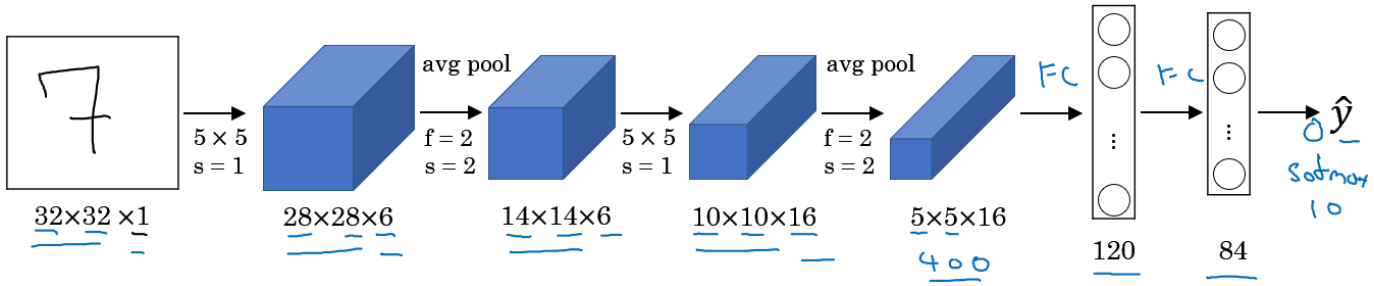
In [14]:
```
model.compile(loss      = "sparse_categorical_crossentropy",
              optimizer = "adam",
              metrics   = ["accuracy"])

history = model.fit(X_train, y_train, epochs = 10,
                    validation_data = (X_valid, y_valid))

score = model.evaluate(X_test, y_test)
```

```
Epoch 1/10
1719/1719 [==============================] - 15s 7ms/step - loss: 0.7676 - accuracy: 0.7271 - val_
loss: 0.4330 - val_accuracy: 0.8520
Epoch 2/10
1719/1719 [==============================] - 12s 7ms/step - loss: 0.4548 - accuracy: 0.8467 - val_
loss: 0.3304 - val_accuracy: 0.8824
Epoch 3/10
1719/1719 [==============================] - 12s 7ms/step - loss: 0.3809 - accuracy: 0.8725 - val_
loss: 0.3182 - val_accuracy: 0.8886
Epoch 4/10
1719/1719 [==============================] - 12s 7ms/step - loss: 0.3524 - accuracy: 0.8788 - val_
loss: 0.2878 - val_accuracy: 0.8952
Epoch 5/10
1719/1719 [==============================] - 12s 7ms/step - loss: 0.3274 - accuracy: 0.8894 - val_
loss: 0.2824 - val_accuracy: 0.8972
Epoch 6/10
1719/1719 [==============================] - 12s 7ms/step - loss: 0.3129 - accuracy: 0.8946 - val_
loss: 0.3019 - val_accuracy: 0.8966
Epoch 7/10
1719/1719 [==============================] - 11s 7ms/step - loss: 0.2929 - accuracy: 0.9000 - val_
loss: 0.2871 - val_accuracy: 0.8942
Epoch 8/10
1719/1719 [==============================] - 11s 7ms/step - loss: 0.2767 - accuracy: 0.9067 - val_
loss: 0.2895 - val_accuracy: 0.8932
Epoch 9/10
1719/1719 [==============================] - 12s 7ms/step - loss: 0.2654 - accuracy: 0.9111 - val_
loss: 0.2863 - val_accuracy: 0.9008
Epoch 10/10
1719/1719 [==============================] - 11s 7ms/step - loss: 0.2592 - accuracy: 0.9119 - val_
loss: 0.2904 - val_accuracy: 0.9030
313/313 [==============================] - 1s 4ms/step - loss: 0.2997 - accuracy: 0.9059
```

LeNet

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

LeNet Layers



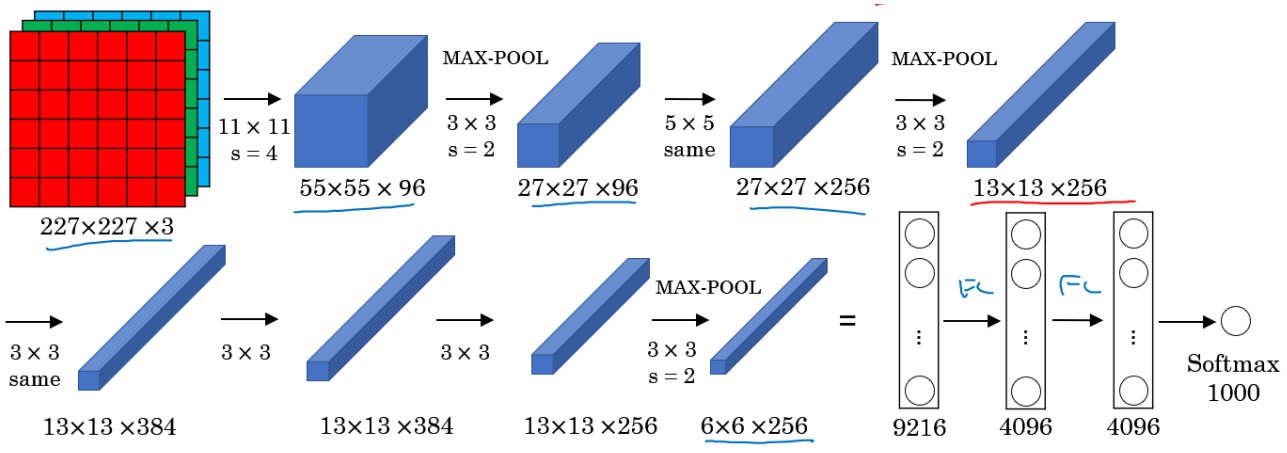
LeNet Architecture

AlexNet

- The AlexNet CNN architecture won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best achieved only 26%!
- It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton.
- It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer.

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6 × 6	3 × 3	2	valid	–
C7	Convolution	256	13 × 13	3 × 3	1	same	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	same	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	same	ReLU
S4	Max pooling	256	13 × 13	3 × 3	2	valid	–
C3	Convolution	256	27 × 27	5 × 5	1	same	ReLU
S2	Max pooling	96	27 × 27	3 × 3	2	valid	–
C1	Convolution	96	55 × 55	11 × 11	4	valid	ReLU
In	Input	3 (RGB)	227 × 227	–	–	–	–

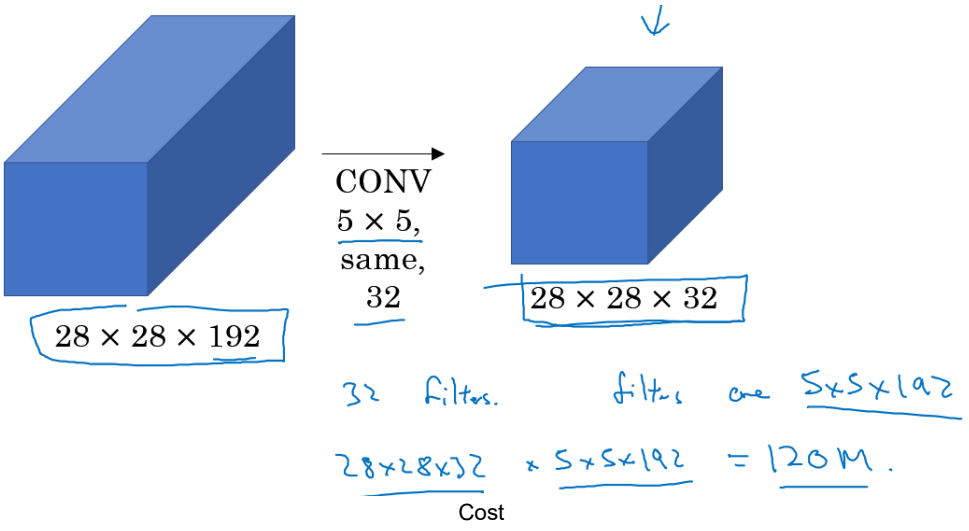
AlexNet Layers



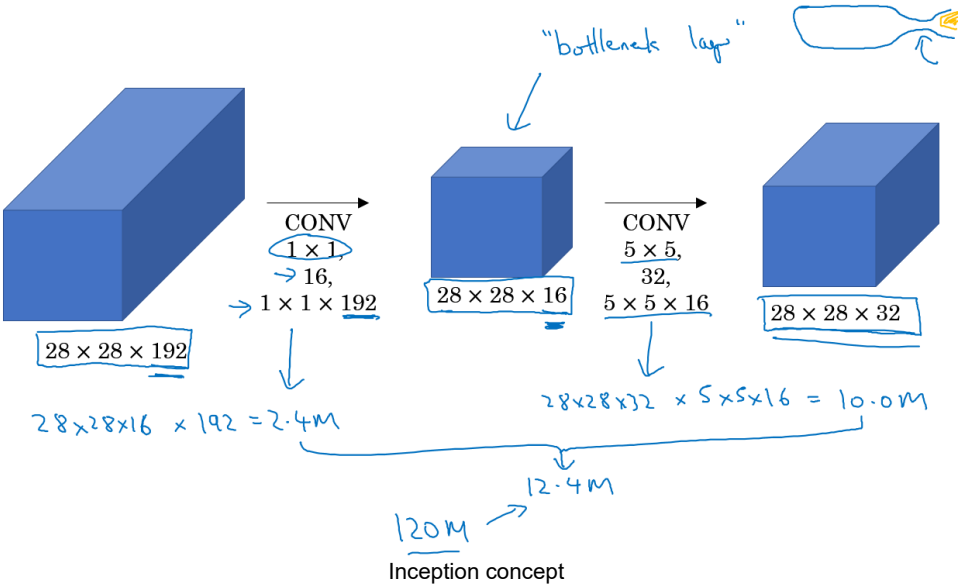
AlexNet Architecture

GoogLeNet

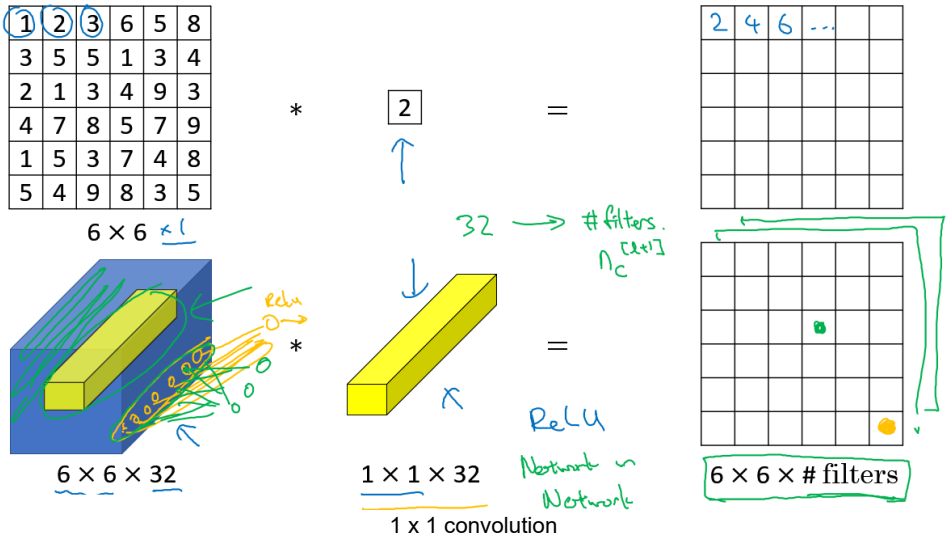
- The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research and it won the ILSVRC 2014 challenge by pushing the top-five error rate below 7%.
- This great performance came in large part from the fact that the network was much deeper than previous CNNs.
- This was made possible by subnetworks called ***inception modules***, which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

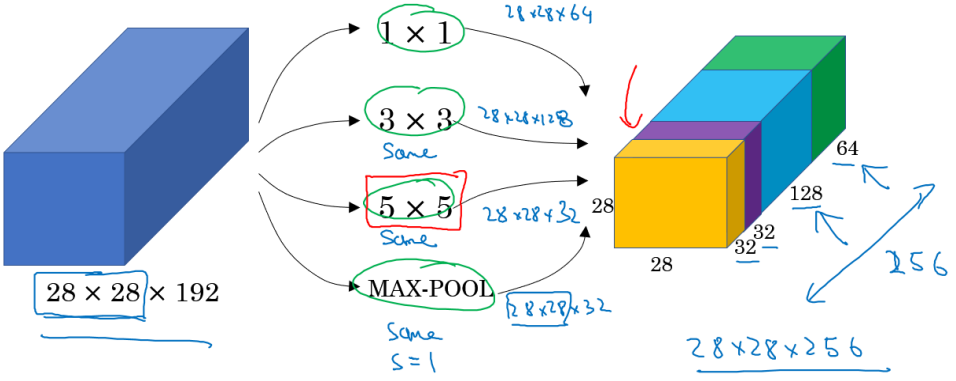


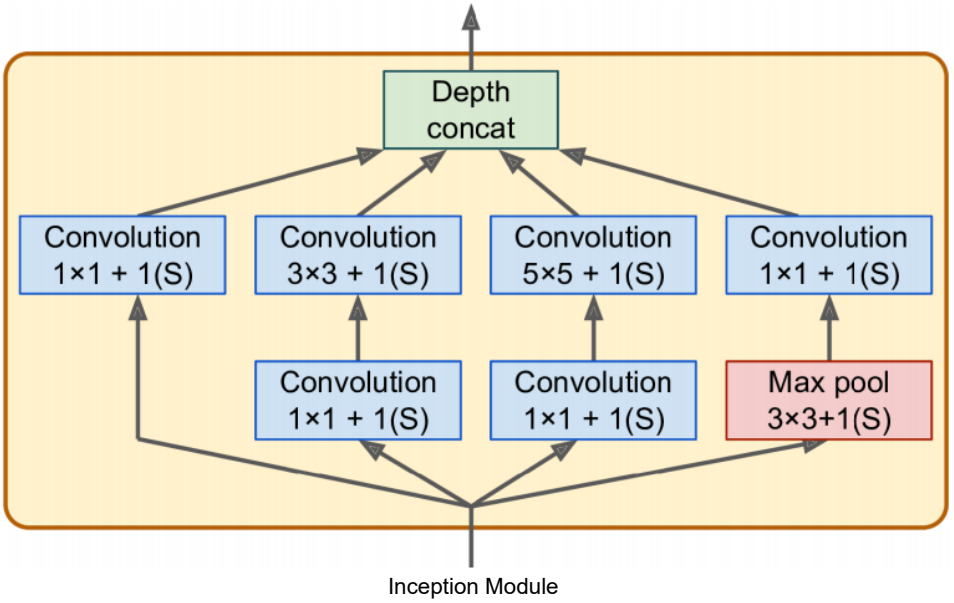
Cost



Inception concept

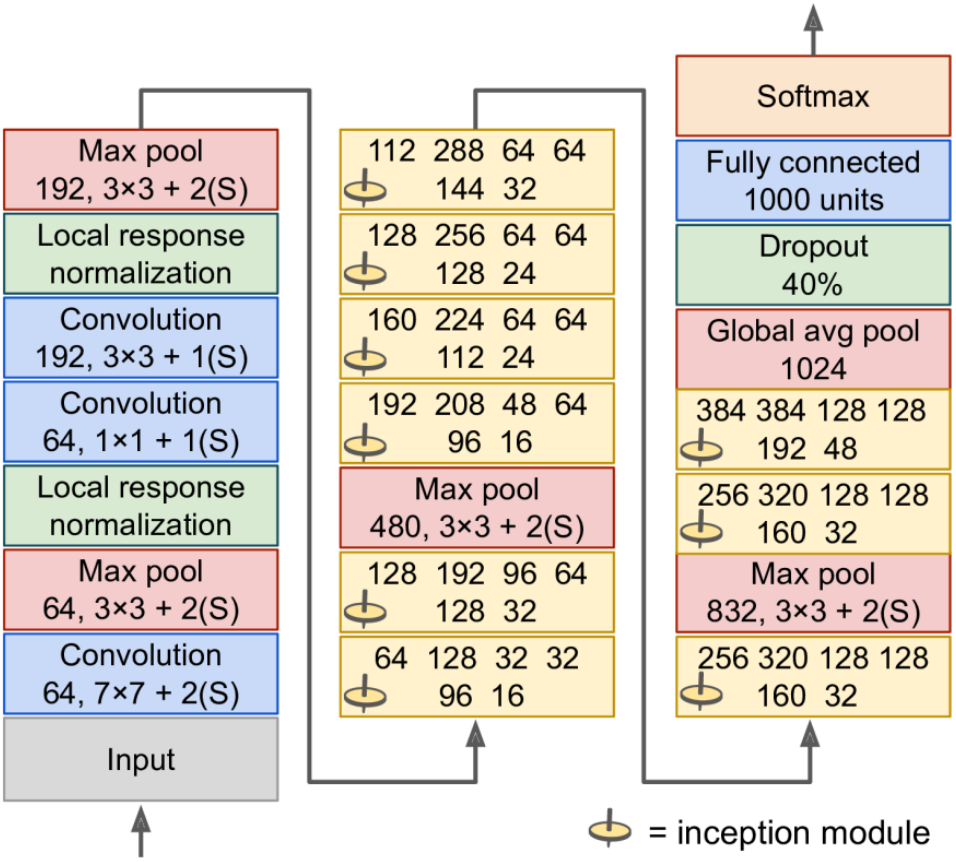

6 × 6 × 1
6 × 6 × 32
1 × 1 × 32
6 × 6 × # filters
ReLU
1 x 1 convolution


28 × 28 × 192
1 × 1
3 × 3
5 × 5
MAX-POOL
Same
28 × 28 × 256
Inception concept


Depth concat
Convolution 1×1 + 1(S)
Convolution 3×3 + 1(S)
Convolution 5×5 + 1(S)
Convolution 1×1 + 1(S)
Convolution 1×1 + 1(S)
Convolution 1×1 + 1(S)
Max pool 3×3+1(S)
Inception Module

- The notation “3 × 3 + 1(S)” means that the layer uses a 3 × 3 kernel, stride 1, and "same" padding.
- The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function.

- The second set of convolutional layers uses different kernel sizes (1 × 1, 3 × 3, and 5 × 5), allowing them to capture patterns at different scales.
- Also note that every single layer uses a stride of 1 and "same" padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final depth concatenation layer (i.e., stack the feature maps from all four top convolutional layers).
- This concatenation layer can be implemented in TensorFlow using the tf.concat() operation, with axis=3 (the axis is the depth).
- Purpose of 1 x 1 kernels: Although they cannot capture spatial patterns, they can capture patterns along the depth dimension.



GoogLeNet

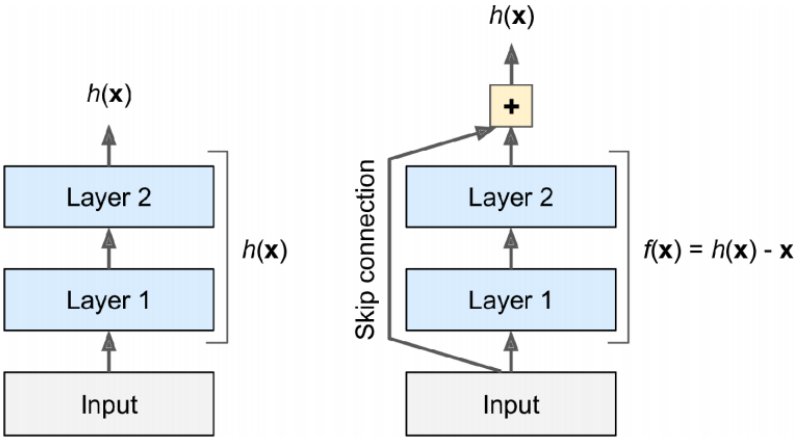
- The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size.
- The architecture is so deep that it has to be represented in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops).
- The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module. Note that all the convolutional layers use the ReLU activation function.

VGGNet

- The runner-up in the ILSVRC 2014 challenge was VGGNet, developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University.
- It had a very simple and classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of just 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used only 3 × 3 filters, but many filters.

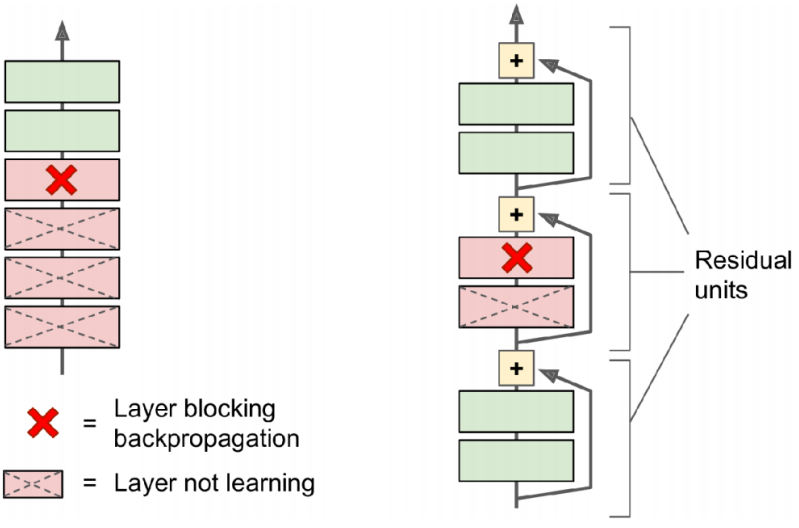
ResNet

- Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (or ResNet), that delivered an astounding top-five error rate under 3.6%.
- The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers).
- It confirmed the general trend: models are getting deeper and deeper, with fewer and fewer parameters.
- The key to being able to train such a deep network is to use ***skip connections*** (also called shortcut connections): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack.



Residual learning

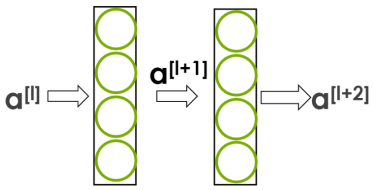
- When training a neural network, the goal is to make it model a target function h(x).
- If you add the input x to the output of the network (i.e., you add a skip connection), then the network will be forced to model f(x) = h(x) – x rather than h(x). This is called ***residual learning***.
- When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero.
- If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function.
- If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.



Regular deep neural network (left) and deep residual network (right)

- If you add many skip connections, the network can start making progress even if several layers have not started learning yet.
- Thanks to skip connections, the signal can easily make its way across the whole network.

Sequential Block

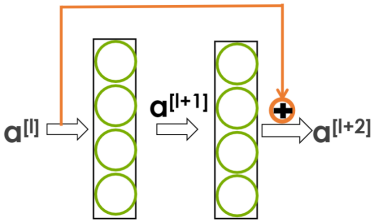


$$z^{[l+1]} = W^{[l+1]} * a^{[l]} + b^{[l+1]}$$
$$a^{[l+1]} = g(z^{[l+1]})$$
$$z^{[l+2]} = W^{[l+2]} * a^{[l+1]} + b^{[l+2]}$$
$$a^{[l+2]} = g(z^{[l+2]})$$

$a^{[l]}$ -> Linear -> ReLu -> Linear -> ReLu

Sequential Block

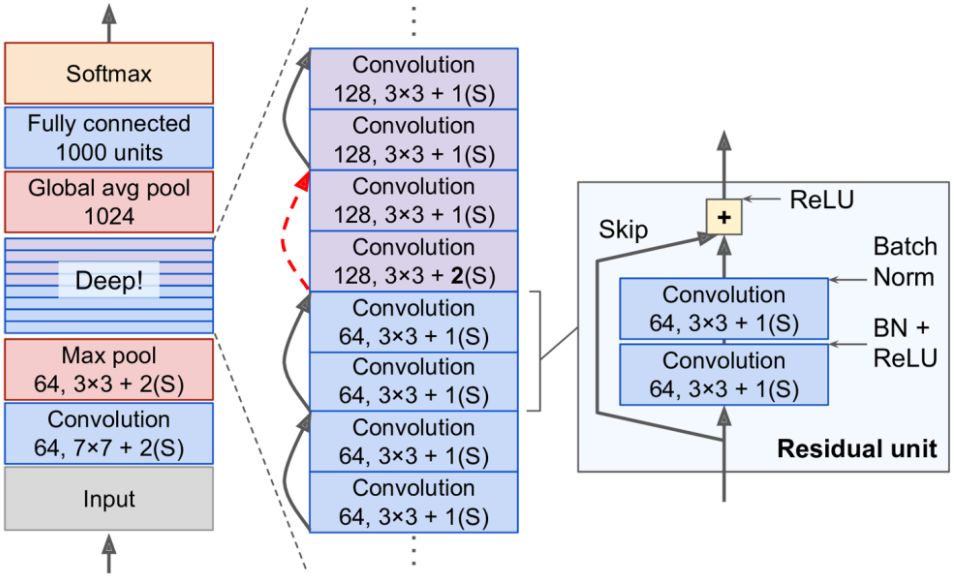
Residual Block ("shortcut" or "skip connection")



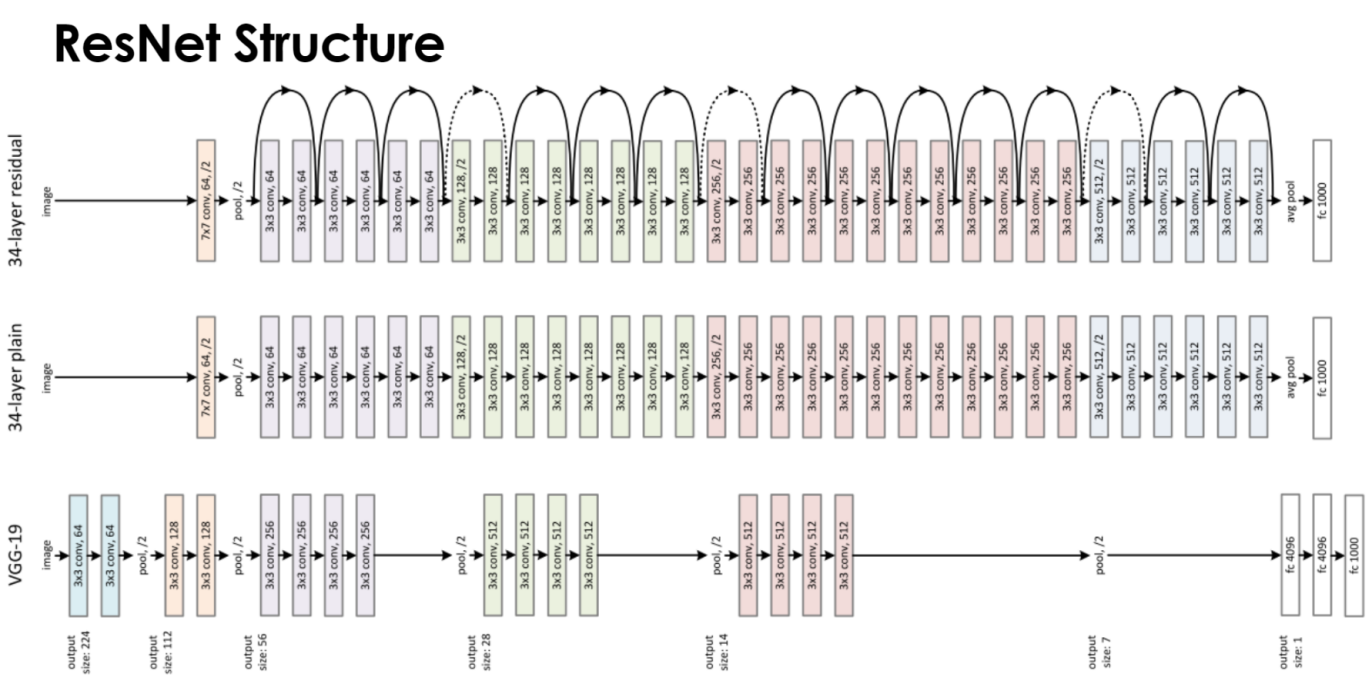
$$z^{[l+1]} = W^{[l+1]} * a^{[l]} + b^{[l+1]}$$
$$a^{[l+1]} = g(z^{[l+1]})$$
$$z^{[l+2]} = W^{[l+2]} * a^{[l+1]} + b^{[l+2]}$$
$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

$a^{[l]}$ -> Linear -> ReLu -> Linear -> ReLu

Residual Block



ResNet



ResNet architecture

- The ResNet architecture starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of simple residual units.
- Each residual unit is composed of two convolutional layers (and no pooling layer!), with Batch Normalization (BN) and ReLU activation, using 3 × 3 kernels and preserving spatial dimensions (stride 1, "same" padding).
- ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer) containing
  - 3 residual units that output 64 feature maps,
  - 4 RUs with 128 maps,
  - 6 RUs with 256 maps, and
  - 3 RUs with 512 maps.

Xception

- Xception (which stands for Extreme Inception) was proposed in 2016 by François Chollet (the author of Keras, and it significantly outperformed Inception-v3 on a huge vision task - 350 million images and 17,000 classes.
- Just like Inception-v4, it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a depthwise separable convolution layer (or separable convolution layer for short).

ResNet-34

- We will implement a ResNet-34 from scratch using Keras.
- First, let's create a ResidualUnit layer:

In [15]:
```
DefaultConv2D = partial(keras.layers.Conv2D,
                        kernel_size = 3, strides = 1,
                        padding = "SAME", use_bias = False)

class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides = strides),
            keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            keras.layers.BatchNormalization()]

        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size = 1, strides = strides),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)

        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)

        return self.activation(Z + skip_Z)
```

- In the constructor, we create all the layers we will need: the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left (only needed if the stride is greater than 1).
- Then in the call() method, we make the inputs go through the main layers and the skip layers (if any), then we add both outputs and apply the activation function.
- Next, we can build the ResNet-34 using a Sequential model, since it's really just a long sequence of layers (we can treat each residual unit as a single layer now that we have the ResidualUnit class):

In [16]:
```
model = keras.models.Sequential()

model.add(DefaultConv2D(64, kernel_size = 7, strides = 2,
                        input_shape = [224, 224, 3]))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size = 3, strides = 2, padding = "SAME"))

prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters

model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation = "softmax"))
```

In [17]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 112, 112, 64)      9408
_________________________________________________________________
batch_normalization (BatchNo (None, 112, 112, 64)      256
_________________________________________________________________
activation (Activation)      (None, 112, 112, 64)      0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
residual_unit (ResidualUnit) (None, 56, 56, 64)        74240
_________________________________________________________________
residual_unit_1 (ResidualUni (None, 56, 56, 64)        74240
_________________________________________________________________
residual_unit_2 (ResidualUni (None, 56, 56, 64)        74240
_________________________________________________________________
residual_unit_3 (ResidualUni (None, 28, 28, 128)       230912
_________________________________________________________________
residual_unit_4 (ResidualUni (None, 28, 28, 128)       295936
_________________________________________________________________
residual_unit_5 (ResidualUni (None, 28, 28, 128)       295936
_________________________________________________________________
residual_unit_6 (ResidualUni (None, 28, 28, 128)       295936
_________________________________________________________________
residual_unit_7 (ResidualUni (None, 14, 14, 256)       920576
_________________________________________________________________
residual_unit_8 (ResidualUni (None, 14, 14, 256)       1181696
_________________________________________________________________
residual_unit_9 (ResidualUni (None, 14, 14, 256)       1181696
_________________________________________________________________
residual_unit_10 (ResidualUn (None, 14, 14, 256)       1181696
_________________________________________________________________
residual_unit_11 (ResidualUn (None, 14, 14, 256)       1181696
_________________________________________________________________
residual_unit_12 (ResidualUn (None, 14, 14, 256)       1181696
_________________________________________________________________
residual_unit_13 (ResidualUn (None, 7, 7, 512)         3676160
_________________________________________________________________
residual_unit_14 (ResidualUn (None, 7, 7, 512)         4722688
_________________________________________________________________
residual_unit_15 (ResidualUn (None, 7, 7, 512)         4722688
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 21,306,826
Trainable params: 21,289,802
Non-trainable params: 17,024
_________________________________________________________________
```

- The first 3 RUs have 64 filters, then the next 4 RUs have 128 filters, and so on.
- We then set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2.
- Then we add the ResidualUnit, and finally we update prev_filters.