Convolutional neural networks

Convolutional neural networks (CNNs)

- We have seen before that neural networks are quite good at dealing with images, and even one-layer perceptron is able to recognize handwritten digits from MNIST dataset with reasonable accuracy.
- However, MNIST dataset is very special, and all digits are centered inside the image, which makes the task simpler.

- In real life, we want to be able to recognize objects on the picture regardless of their exact location in the image.
- When we are trying to find a certain object in the picture, we are scanning the image looking for some specific **patterns** and their combinations.

- For example, when looking for a cat, we first may look for horizontal lines, which can form whiskers, and then certain combination of whiskers can tell us that it is actually a picture of a cat.
- *Relative position and presence of certain patterns* is important, and not their exact position on the image.

Convolutional neural networks (CNNs)

- To extract patterns, we will use the notion of **convolutional filters**.

In [1]:
```
import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter(action='ignore', category=FutureWarning)
```

In [2]:
```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
from tfcv import *
```

In [3]:
```
(x_train,y_train),(x_test,y_test) = keras.datasets.mnist.load_data()
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
```

Convolutional filters

- Convolutional filters are small windows that run over each pixel of the image and compute weighted average of the neighboring pixels.
- They are defined by matrices of weight coefficients.

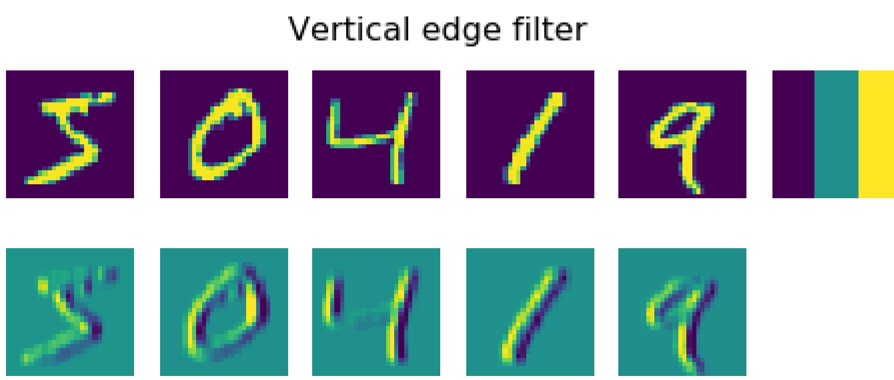
- For example, a **vertical edge filter** is defined by the following matrix:

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

- When this filter goes over relatively uniform pixel field, all values add up to 0.
- However, when it encounters a vertical edge in the image, high spike values are generated.

Convolutional filters on MNIST digits

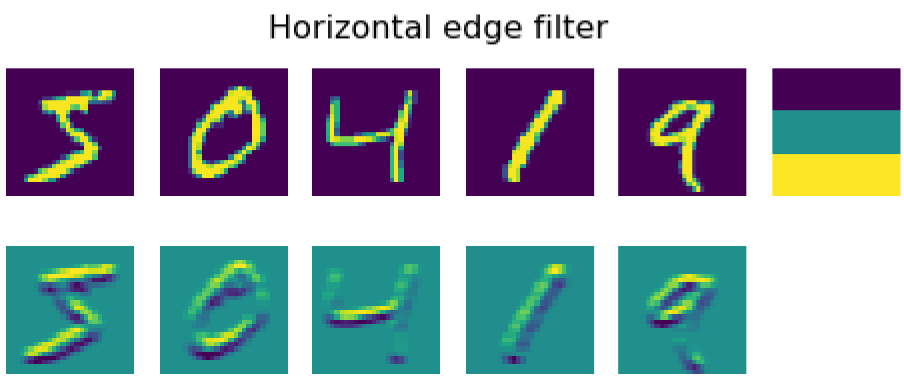
In [4]:
```
plot_convolution(x_train[:5],[[-1.,0.,1.],[-1.,0.,1.],[-1.,0.,1.]],'Vertical edge f
```



In the images above, vertical edges are represented by high and low values, while horizontal edges are average out.

Convolutional filters on MNIST digits

In [5]:
```
plot_convolution(x_train[:5],[[-1.,-1.,-1.],[0.,0.,0.],[1.,1.,1.]],'Horizontal edge
```

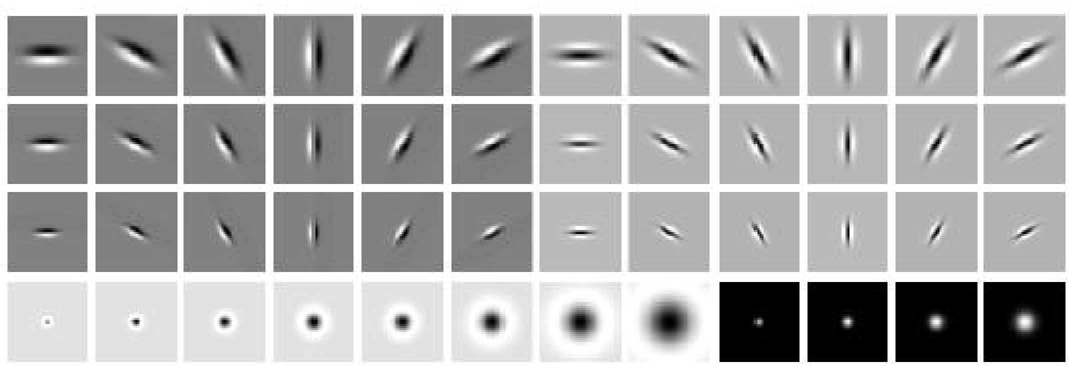


When we apply horizontal edge filter, horizontal lines are amplified, and vertical lines are averaged out.

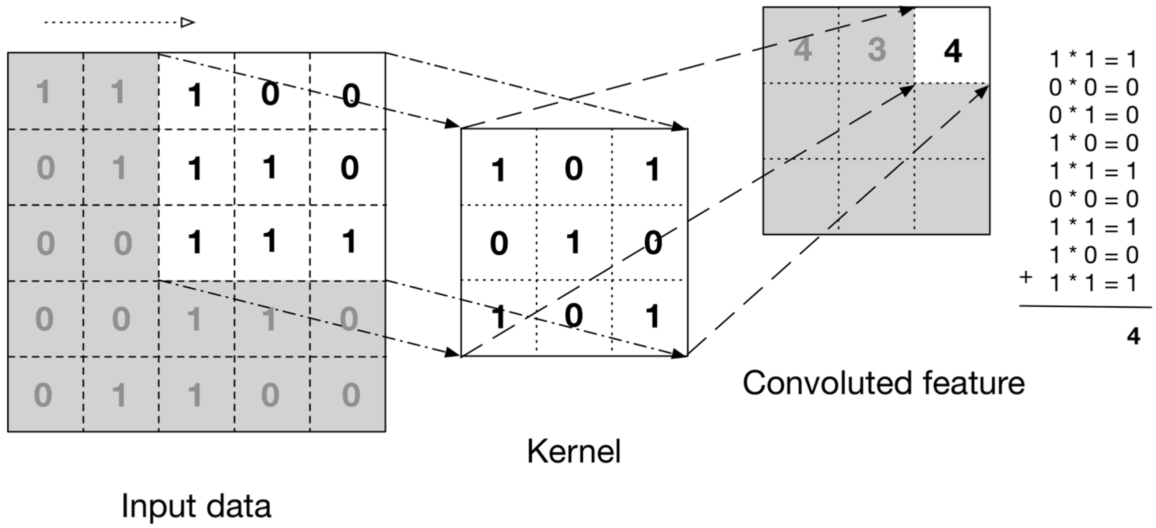
Convolutional filters

- In classical computer vision, multiple filters were applied to the image to generate **features**, which then were used by machine learning algorithm to build a classifier.

- Those filters are in fact similar to neural structures that are available in the vision system of some animals.



- However, in **deep learning** we construct networks that **learn bestconvolutional filters** to solve classification problem. We introduce *convolutional layers*.



Convolutional layers

- To make the weights of convolutional layer trainable, we need somehow to reduce the process of applying convolutional filter window to the image to the matrix operations, which can then be subject to backward propagation training.

- We use a matrix transformation, called **im2col**.

- Suppose we have a small image $\mathbf{x}$ with the following pixels:

$$\mathbf{x} = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix}$$

- We want to apply two convolutional filters with the following weights:

$$W^{(i)} = \begin{pmatrix} w^{(i)}_{00} & w^{(i)}_{01} & w^{(i)}_{02} \\ w^{(i)}_{10} & w^{(i)}_{11} & w^{(i)}_{12} \\ w^{(i)}_{20} & w^{(i)}_{21} & w^{(i)}_{22} \end{pmatrix}$$

- Suppose we have a small image $\mathbf{x}$ with the following pixels:

$$\mathbf{x} = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \end{pmatrix}$$

- We want to apply two convolutional filters with the following weights:

$$W^{(i)} = \begin{pmatrix} w_{00}^{(i)} & w_{01}^{(i)} & w_{02}^{(i)} \\ w_{10}^{(i)} & w_{11}^{(i)} & w_{12}^{(i)} \\ w_{20}^{(i)} & w_{21}^{(i)} & w_{22}^{(i)} \end{pmatrix}$$

- When applying the convolution, the first pixel of the result is obtained by element-wise multiplication of $\begin{pmatrix} a & b & c \\ f & g & h \\ k & l & m \end{pmatrix}$ and $W^{(i)}$.

- The second pixel of the result is obtained by element-wise multiplying $\begin{pmatrix} b & c & d \\ g & h & i \\ l & m & n \end{pmatrix}$ by $W^{(i)}$.

- Let's extract all 3×3 fragments of the original image $\mathbf{x}$ into the following matrix:

$$\mathrm{im2col}(x) = \begin{bmatrix} a & b & \dots & g & \dots & m \\ b & c & \dots & h & \dots & n \\ c & d & \dots & i & \dots & o \\ f & g & \dots & l & \dots & r \\ g & h & \dots & m & \dots & s \\ h & i & \dots & n & \dots & t \\ k & l & \dots & q & \dots & w \\ l & m & \dots & r & \dots & x \\ m & n & \dots & s & \dots & y \end{bmatrix}$$

- To get the result of the convolution, we multiply this matrix by the weights:

$$\mathbf{W} = \begin{bmatrix} w_{00}^{(0)} & w_{01}^{(0)} & w_{02}^{(0)} & w_{10}^{(0)} & w_{11}^{(0)} & \dots & w_{21}^{(0)} & w_{22}^{(0)} \\ w_{00}^{(1)} & w_{01}^{(1)} & w_{02}^{(1)} & w_{10}^{(1)} & w_{11}^{(1)} & \dots & w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix}$$

- Each row of this matrix contains weights of i-th filter, flattened into one row.

- The application of a filter to the original image is replaced by matrix multiplication:

$$C(x) = W \times \mathbf{im2col}(x)$$

Convolutional layers

For example, if we use `Keras`, convolutional layers are defined using `Conv2D` class. We specify the following:

- `filters`: number of filters to use.
- `kernel_size`: the size of the sliding window. Usually 3x3 or 5x5 filters are used.

In [14]:
```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=9, kernel_size=(5,5), input_shape=(28,28,1), activa
    keras.layers.Flatten(),
    keras.layers.Dense(10)
])
model.compile(loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True), me
```

- Simple CNN contains one convolutional layer. We use 9 different filters, giving the network multiple opportunities to explore which filters work best for our scenario.
- After convolution, we flat the result tensor into one vector, then add a linear layer to produce 10 classes.

In [15]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_4 (Conv2D)            (None, 24, 24, 9)         234       
_________________________________________________________________
flatten_3 (Flatten)          (None, 5184)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 10)                51850     
=================================================================
Total params: 52,084
Trainable params: 52,084
Non-trainable params: 0
_________________________________________________________________
```

Convolutional layers

In most cases, we apply convolutional layers to color images. `Conv2D` layer expects the input to be of the shape $W \times H \times C$ where W and H are width and height of the image, and C is the number of color channels. For grayscale images, we need the same shape with $C = 1$.

In [16]:
```
x_train.shape
```

Out[16]:
```
(60000, 28, 28)
```

In [17]:
```
x_train_c = np.expand_dims(x_train, 3)
x_test_c = np.expand_dims(x_test, 3)
```

In [18]:
```
x_train_c.shape
```

Out[18]:
```
(60000, 28, 28, 1)
```

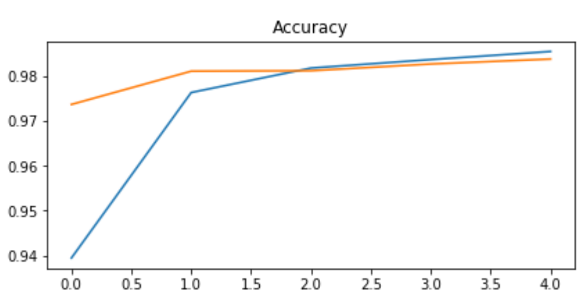
Convolutional neural networks (CNNs)

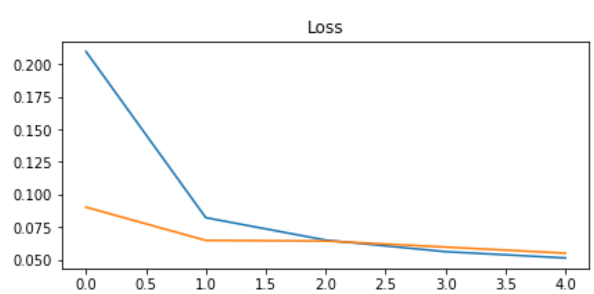
In [19]:
```
hist = model.fit(x_train_c, y_train, validation_data=(x_test_c,y_test),epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1985 - acc: 0.942
6 - val_loss: 0.0842 - val_acc: 0.9759
Epoch 2/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0834 - acc: 0.975
6 - val_loss: 0.0673 - val_acc: 0.9782
Epoch 3/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0659 - acc: 0.980
3 - val_loss: 0.0656 - val_acc: 0.9795
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0576 - acc: 0.983
2 - val_loss: 0.0542 - val_acc: 0.9825
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0522 - acc: 0.985
1 - val_loss: 0.0535 - val_acc: 0.9821
```

Convolutional neural networks (CNNs)

In [9]:
```
plot_results(hist)
```





Visualizing convolutional layers

In [51]:
```
l = model.layers[0].weights[0]
fig, ax = plt.subplots(1, 9)
for i in range(9):
    ax[i].imshow(l[...,0,i])
    ax[i].axis('off')
```

Train the same network with 3x3 filters and visualize them.

Multi-layered CNNs and pooling layers

- First convolutional layers looks for primitive patterns, such as horizontal or vertical lines.

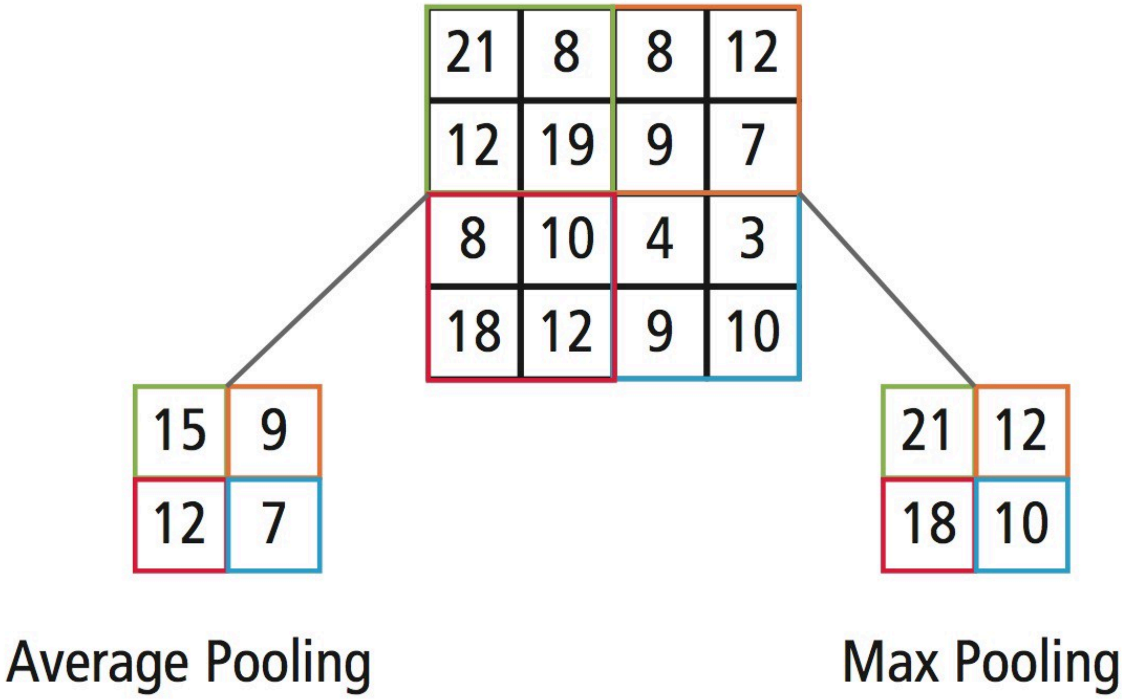
- We can apply further convolutional layers on top of them to look for higher-level patterns, such as primitive shapes.
- Then more convolutional layers can combine those shapes into some parts of the picture, up to the final object that we are trying to classify.

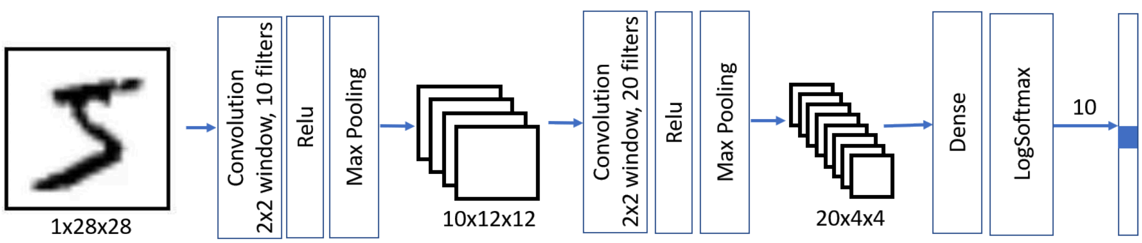
- We may also apply one trick: *reducing the spatial size of the image*. Once we have detected there is a horizontal stoke within sliding 3x3 window, it is not so important at which exact pixel it occurred.

- We can "scale down" the size of the image, which is done using **pooling layers**.

Multi-layered CNNs and pooling layers

- **Average Pooling** takes a sliding window (for example, 2x2 pixels) and computes an average of values within the window
- **Max Pooling** replaces the window with the maximum value. The idea behind max pooling is to detect a presence of a certain pattern within the sliding window.

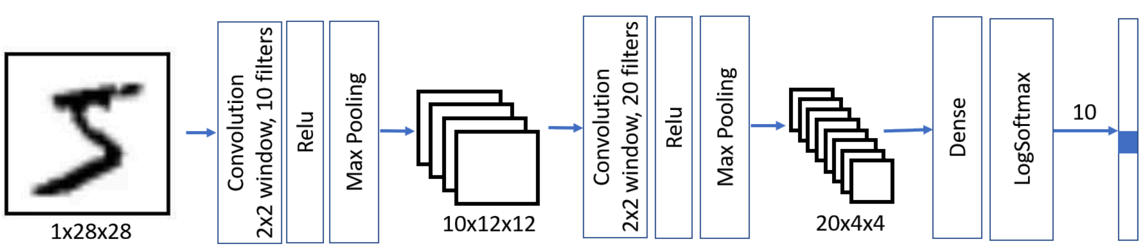




- In a typical CNN, there are several convolutional layers, with pooling layers in between them to decrease the dimensions of the image.

- We also increase the number of filters because as patterns become more advanced, there are more possible interesting combinations.
- Because of decreasing spatial dimensions and increasing feature/filters dimensions, this architecture is called **pyramid architecture**.

Multi-layered CNNs and pooling layers



```
In [52]: model = keras.models.Sequential([
    keras.layers.Conv2D(filters=10, kernel_size=(5,5), input_shape=(28,28,1), activ
    keras.layers.MaxPooling2D(),
    keras.layers.Conv2D(filters=20, kernel_size=(5,5), activation='relu'),
    keras.layers.MaxPooling2D(),
    keras.layers.Flatten(),
    keras.layers.Dense(10)
])
model.compile(loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True), me
```

The number of trainable parameters of this example model is 8490, much smaller than in previous cases. Convolutional layers in general have few parameters, and the dimensionality of the image before applying final `Dense` layer is considerably reduced.

```
In [53]: model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 24, 24, 10)        260
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 10)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 20)          5020
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 20)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 320)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                3210
=================================================================
Total params: 8,490
Trainable params: 8,490
Non-trainable params: 0
_________________________________________________________________
```

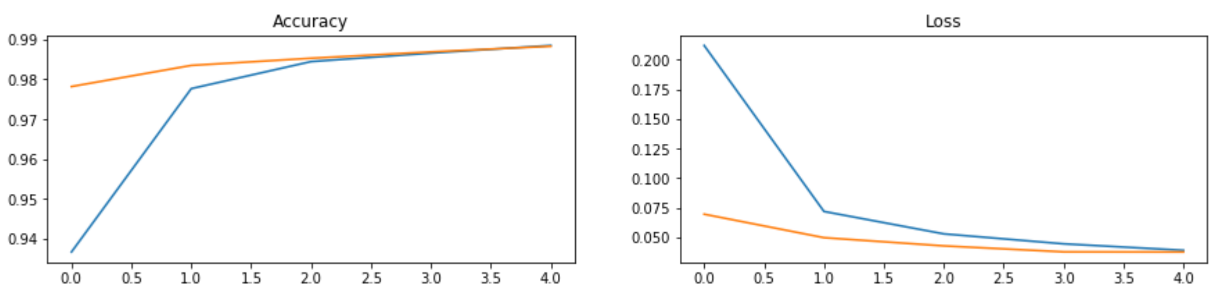
Multi-layered CNNs and pooling layers

```
In [54]: hist = model.fit(x_train_c, y_train, validation_data=(x_test_c, y_test), epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 13s 7ms/step - loss: 0.2121 - acc: 0.93
67 - val_loss: 0.0697 - val_acc: 0.9782
Epoch 2/5
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0720 - acc: 0.97
77 - val_loss: 0.0498 - val_acc: 0.9835
Epoch 3/5
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0530 - acc: 0.98
45 - val_loss: 0.0429 - val_acc: 0.9853
Epoch 4/5
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0446 - acc: 0.98
66 - val_loss: 0.0378 - val_acc: 0.9869
Epoch 5/5
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0392 - acc: 0.98
85 - val_loss: 0.0378 - val_acc: 0.9883
```

Multi-layered CNNs and pooling layers

In [55]:
```
plot_results(hist)
```



In [60]:
```
fig, ax = plt.subplots(1,10)
l = model.layers[2].weights[0]
for i in range(10):
    ax[i].imshow(l[...,0,i])
    ax[i].axis('off')
```

Example: CIFAR-10 dataset

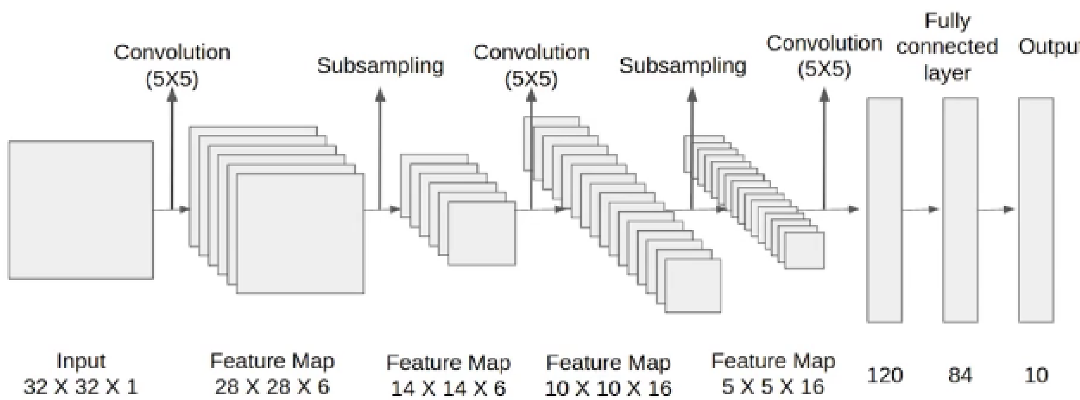
In [61]:
```
(x_train,y_train),(x_test,y_test) = keras.datasets.cifar10.load_data()
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 60s 0us/step
170508288/170498071 [==============================] - 60s 0us/step
```

In [62]:
```
display_dataset(x_train,y_train,classes=classes)
```

Example: LeNet architecture

A well-known architecture for CIFAR-10 is `LeNet` which was proposed by *Yann LeCun*

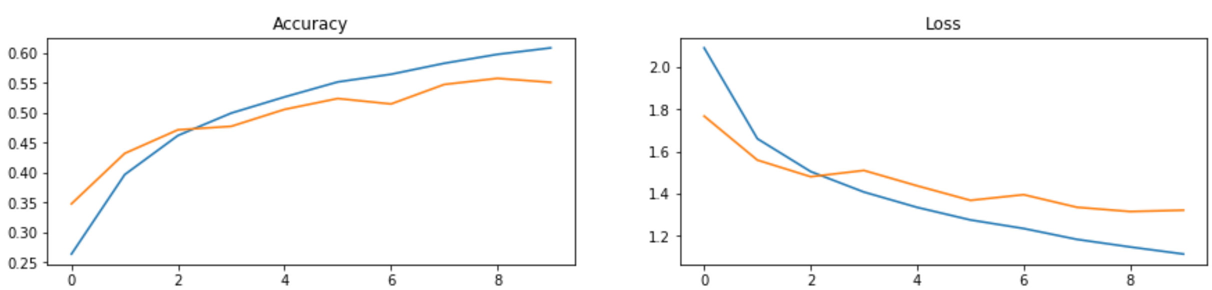


In [63]:
```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters = 6, kernel_size = 5, strides = 1, activation = 're
    keras.layers.MaxPooling2D(pool_size = 2, strides = 2),
    keras.layers.Conv2D(filters = 16, kernel_size = 5, strides = 1, activation = 'r
    keras.layers.MaxPooling2D(pool_size = 2, strides = 2),
    keras.layers.Flatten(),
    keras.layers.Dense(120, activation = 'relu'),
    keras.layers.Dense(84, activation = 'relu'),
    keras.layers.Dense(10, activation = 'softmax')
])
```

In [64]:
```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics
hist = model.fit(x_train, y_train, validation_data=(x_test,y_test), epochs=10)
```

```
Epoch 1/10
1563/1563 [==============================] - 15s 9ms/step - loss: 2.0915 - acc: 0.26
34 - val_loss: 1.7676 - val_acc: 0.3473
Epoch 2/10
1563/1563 [==============================] - 14s 9ms/step - loss: 1.6602 - acc: 0.39
62 - val_loss: 1.5587 - val_acc: 0.4316
Epoch 3/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.5044 - acc: 0.46
15 - val_loss: 1.4798 - val_acc: 0.4710
Epoch 4/10
1563/1563 [==============================] - 14s 9ms/step - loss: 1.4071 - acc: 0.49
89 - val_loss: 1.5096 - val_acc: 0.4768
Epoch 5/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.3342 - acc: 0.52
59 - val_loss: 1.4366 - val_acc: 0.5051
Epoch 6/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.2745 - acc: 0.55
10 - val_loss: 1.3676 - val_acc: 0.5233
Epoch 7/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.2339 - acc: 0.56
38 - val_loss: 1.3945 - val_acc: 0.5142
Epoch 8/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.1822 - acc: 0.58
22 - val_loss: 1.3347 - val_acc: 0.5468
Epoch 9/10
1563/1563 [==============================] - 14s 9ms/step - loss: 1.1462 - acc: 0.59
71 - val_loss: 1.3145 - val_acc: 0.5570
Epoch 10/10
1563/1563 [==============================] - 14s 9ms/step - loss: 1.1126 - acc: 0.60
79 - val_loss: 1.3209 - val_acc: 0.5503
```

In [65]:
```
plot_results(hist)
```



In [66]:
```
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
```

In [67]:
```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters = 6, kernel_size = 5, strides = 1, activation = 're
    keras.layers.MaxPooling2D(pool_size = 2, strides = 2),
    keras.layers.Conv2D(filters = 16, kernel_size = 5, strides = 1, activation = 'r
    keras.layers.MaxPooling2D(pool_size = 2, strides = 2),
    keras.layers.Flatten(),
    keras.layers.Dense(120, activation = 'relu'),
    keras.layers.Dense(84, activation = 'relu'),
    keras.layers.Dense(10, activation = 'softmax')
```

```
])
model.summary()
```

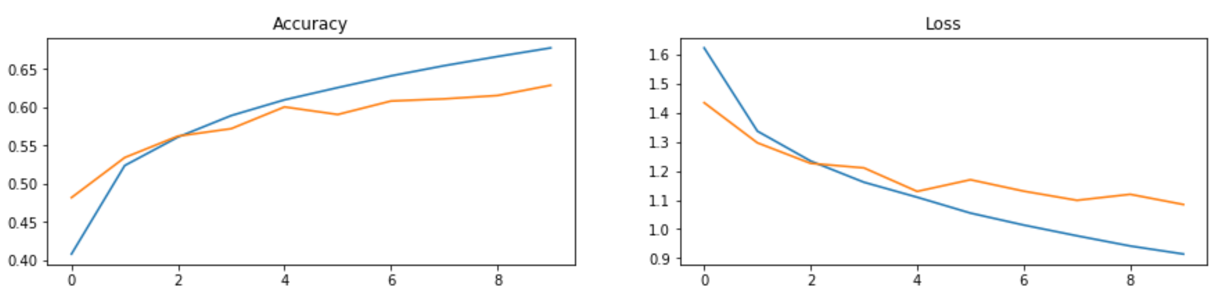
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_9 (Conv2D)            (None, 28, 28, 6)         456       
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 14, 14, 6)         0         
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 10, 10, 16)        2416      
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 5, 5, 16)          0         
_________________________________________________________________
flatten_6 (Flatten)          (None, 400)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 120)               48120     
_________________________________________________________________
dense_9 (Dense)              (None, 84)                10164     
_________________________________________________________________
dense_10 (Dense)             (None, 10)                850       
=================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
_________________________________________________________________
```

In [68]:

```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics
hist = model.fit(x_train, y_train, validation_data=(x_test,y_test), epochs=10)
```

```
Epoch 1/10
1563/1563 [==============================] - 15s 10ms/step - loss: 1.6230 - acc: 0.4
082 - val_loss: 1.4345 - val_acc: 0.4819
Epoch 2/10
1563/1563 [==============================] - 15s 10ms/step - loss: 1.3367 - acc: 0.5
238 - val_loss: 1.2965 - val_acc: 0.5342
Epoch 3/10
1563/1563 [==============================] - 15s 10ms/step - loss: 1.2340 - acc: 0.5
609 - val_loss: 1.2259 - val_acc: 0.5622
Epoch 4/10
1563/1563 [==============================] - 15s 10ms/step - loss: 1.1610 - acc: 0.5
890 - val_loss: 1.2103 - val_acc: 0.5719
Epoch 5/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.1094 - acc: 0.60
96 - val_loss: 1.1295 - val_acc: 0.6003
Epoch 6/10
1563/1563 [==============================] - 15s 9ms/step - loss: 1.0549 - acc: 0.62
55 - val_loss: 1.1696 - val_acc: 0.5904
Epoch 7/10
1563/1563 [==============================] - 15s 10ms/step - loss: 1.0138 - acc: 0.6
408 - val_loss: 1.1303 - val_acc: 0.6080
Epoch 8/10
1563/1563 [==============================] - 15s 10ms/step - loss: 0.9768 - acc: 0.6
541 - val_loss: 1.0988 - val_acc: 0.6108
Epoch 9/10
1563/1563 [==============================] - 15s 9ms/step - loss: 0.9417 - acc: 0.66
61 - val_loss: 1.1195 - val_acc: 0.6152
Epoch 10/10
1563/1563 [==============================] - 15s 9ms/step - loss: 0.9141 - acc: 0.67
74 - val_loss: 1.0844 - val_acc: 0.6286
```

In [69]:
```python
plot_results(hist)
```



Real-life architectures for image classification, object detection, and even image generation networks are based on CNNs with more layers, more sophisticated designs and model training techniques.