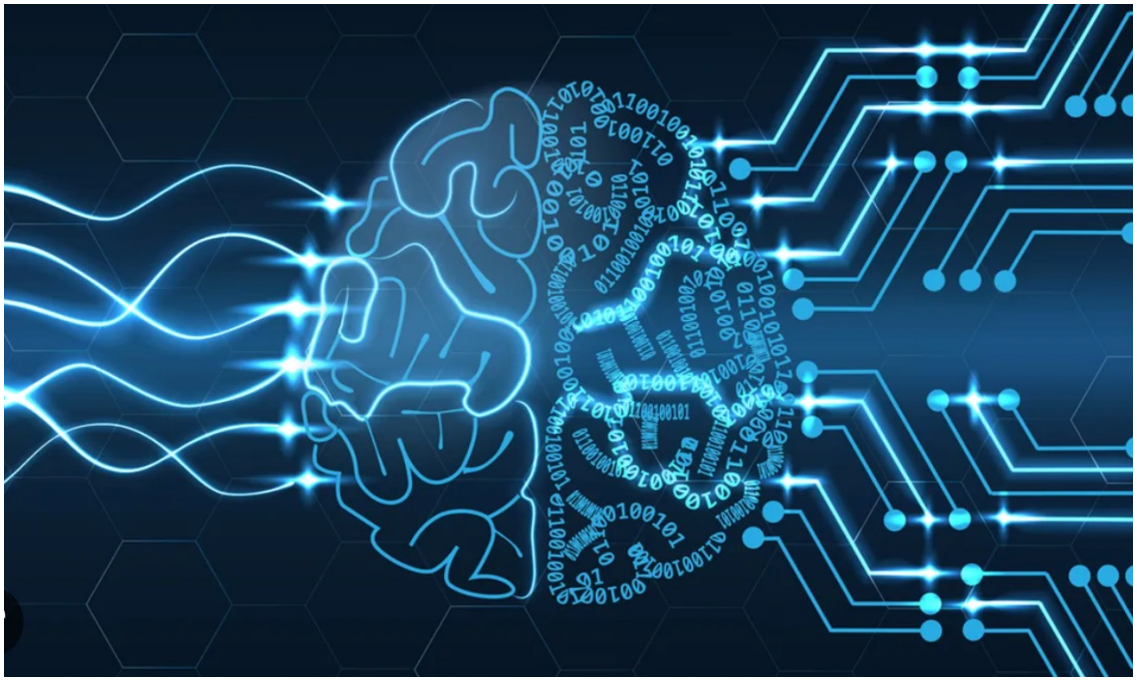
CS 4100: Introduction to AI

Wayne Snyder
Northeastern University

Lecture 21: Deep Learning – Generalization; Convolutional Networks



Generalization

Plan for this Lecture:

- Image Classification: MNIST Digit Classification
- Generalization
- Convolutional Networks for Image Processing

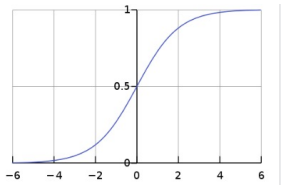
Monday's Lecture:

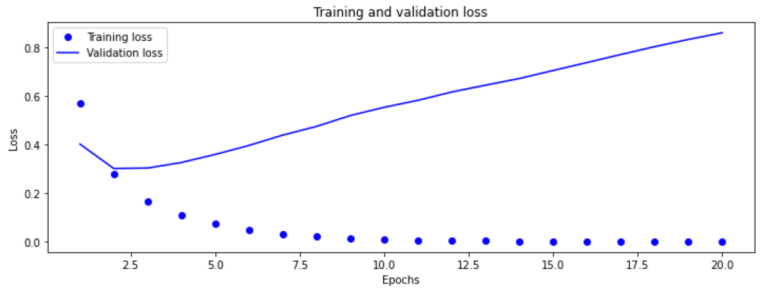
- Sequence Models
- Recurrent Networks

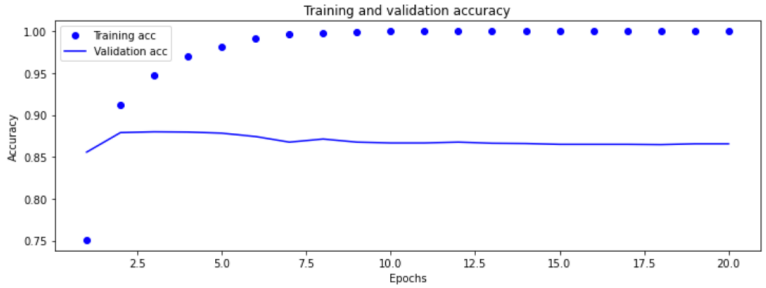
Binary Classification

Recall: In Binary Classification, our goal is to label the data (e.g., text, images, sounds) with one of 2 labels.

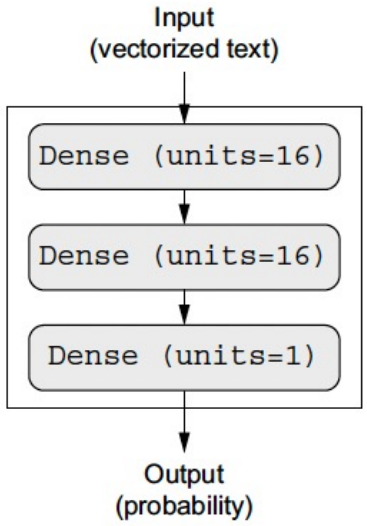
```
model = keras.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    verbose=0,
                    batch_size=512,
                    validation_split=0.2)

display_graphs(history)

model.evaluate(x_test, y_test)
```







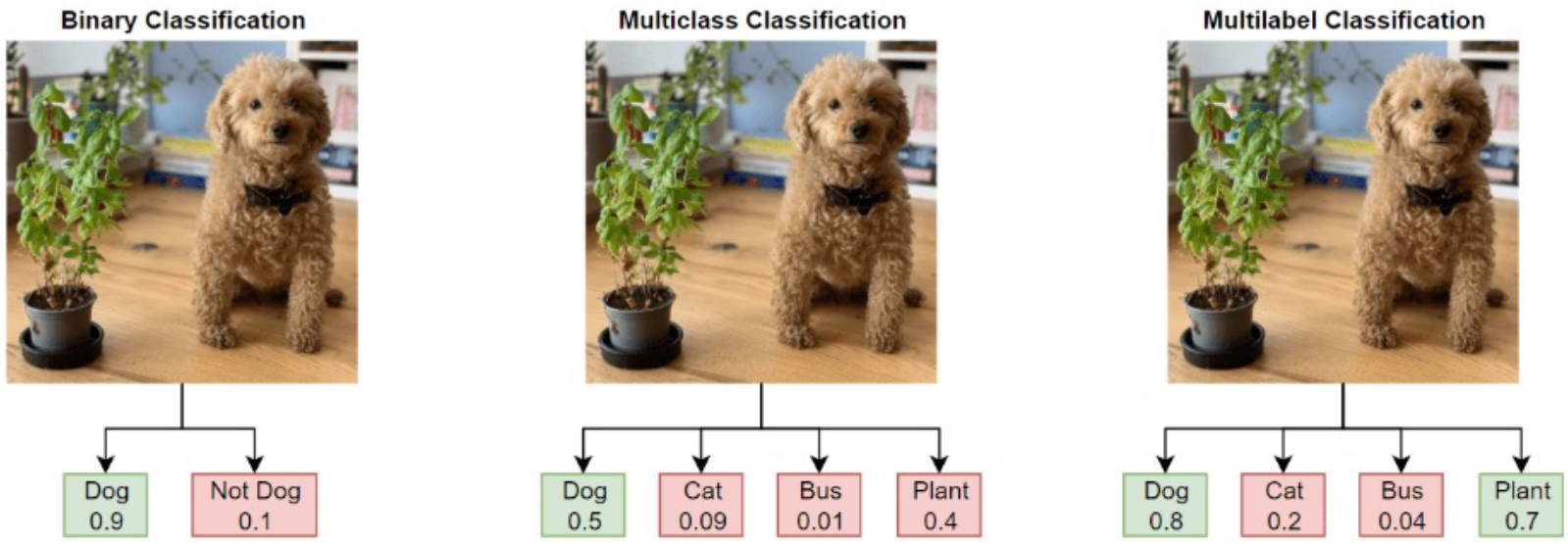

```
782/782 [==============================] - 1s 814us/step - loss: 0.8836 - accuracy: 0.8555
[0.8836396336555481, 0.8555200099945068]
```



Multiclass and Multilabel Classification

In Multiclass Classification, we have more than 2 labels, and our task is to assign a single label to each sample:

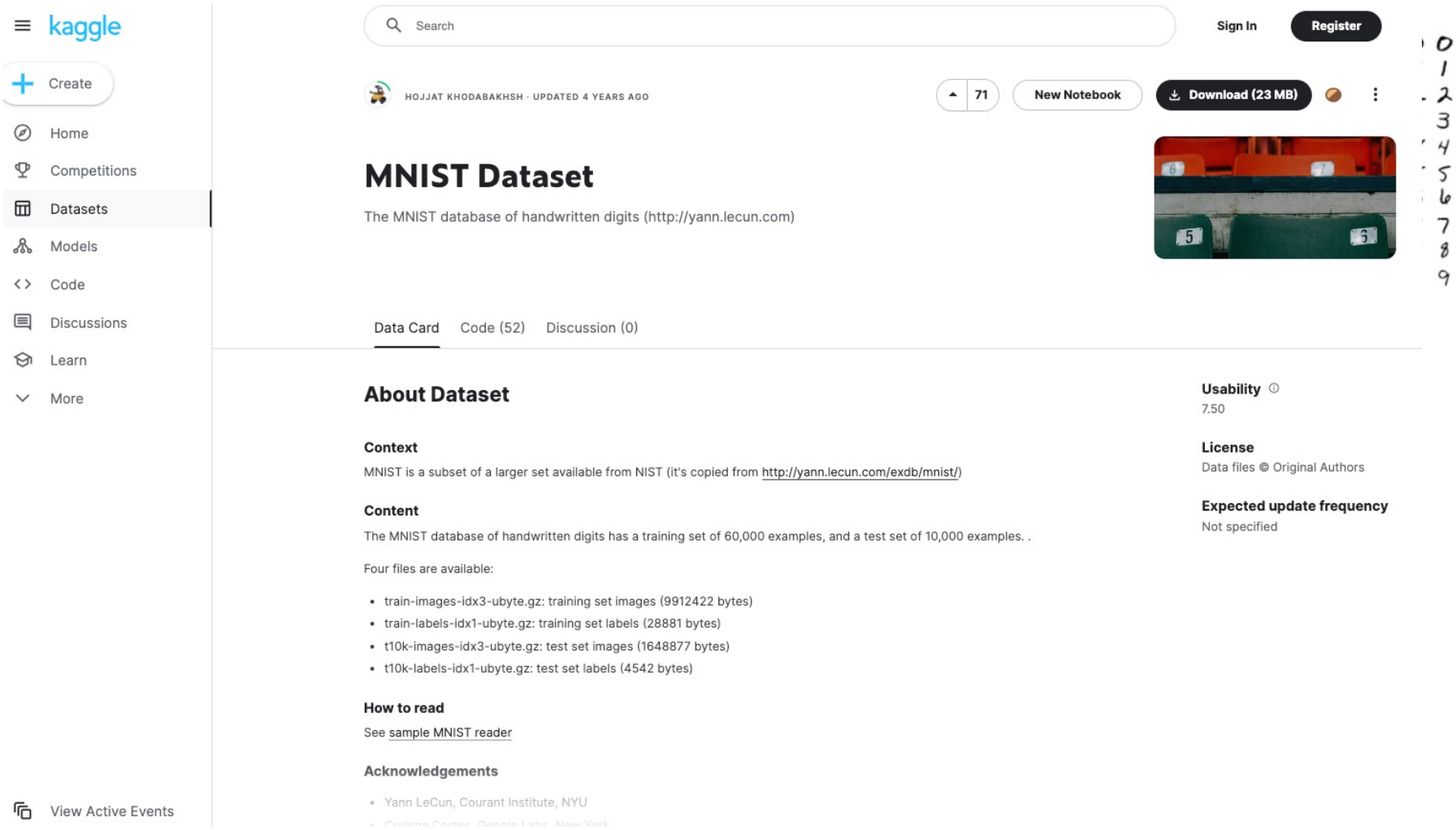
In Multilabel Classification, we have more than 2 labels, and our task is to assign any appropriate labels (not just one):



Let's look at an example of multiclass classification, using the MNIST handwritten digits database....

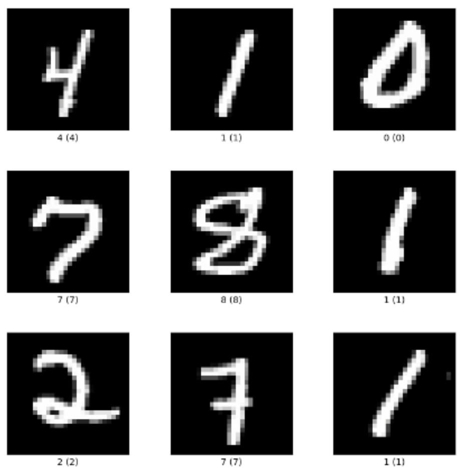
Multiclass Classification

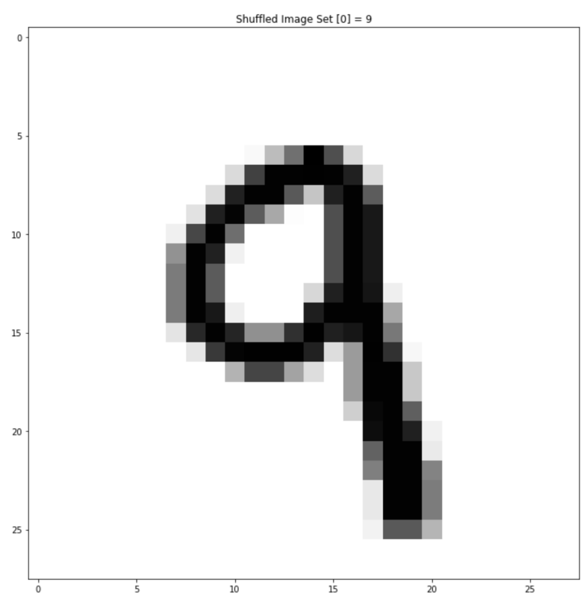
We will consider the MNIST database of handwritten digits:



Multiclass Classification

The MNIST digit database consists of 70,000 28x28 BW pixel images, stored as 28*28=784 floating-point numbers in the range [0..1]; labels are integers 0..9:






```
In [33]:    1 print(training_images[0])

[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.05882353 0.08235294 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.05882353
 0.54901963 0.49019608 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.16862746 0.98039216 0.49019608
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.7176471  0.98039216 0.49019608 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.69411767 0.9882353
 0.98039216 0.49019608 0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.08627451 0.7176471  0.9882353  0.99607843 0.93333334 0.4117647
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.11372549 0.65882355 0.98039216
 0.98039216 0.93333334 0.43529412 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.08627451 0.65882355 0.98039216 0.98039216 0.98039216 0.4117647
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.7176471  0.98039216
 0.98039216 0.98039216 0.98039216 0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.9882353  0.98039216 0.98039216 0.98039216
 0.98039216 0.         0.         0.         0.         0.
 0.         0.         0.41960785 0.27450982 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
```

Etc.

Note that the array is sparse, it is mostly 0's.

```
In [36]:    1 print(train_labels[:100])

[5 0 4 1 9 2 1 3 1 4 3 5 3 6 1 7 2 8 6 9 4 0 9 1 1 2 4 3 2 7 3 8 6 9 0 5 6
 0 7 6 1 8 7 9 3 9 8 5 9 3 3 0 7 4 9 8 0 9 4 1 4 4 6 0 4 5 6 1 0 0 1 7 1 6
 3 0 2 1 1 7 9 0 2 6 7 8 3 9 0 4 6 7 4 6 8 0 7 8 3 1]
```

Multiclass Classification

As usual, the database is stored in four parts, split into training and testing sets already:

In [1]:
```
import tensorflow as tf
from tensorflow import keras
from keras import layers
from keras.datasets import mnist

import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
(train_images,train_labels),(test_images,test_labels) = mnist.load_data()
```

In [3]:
```
train_images.shape
```

Out[3]:
```
(60000, 28, 28)
```

In [4]:
```
train_labels.shape
```

Out[4]:
```
(60000,)
```

In [5]:
```
test_images.shape
```

Out[5]:
```
(10000, 28, 28)
```

In [6]:
```
test_labels.shape
```

Out[6]:
```
(10000,)
```

Multiclass Classification

However, just to show the complete process, let's combine training and test sets, and go through the process of creating training, validation, and test sets for data and labels:

In [25]:
```
# Just to show, let's put all the data together, shuffle, and separate
# into training, validation, and test sets.

images = np.concatenate([train_images,test_images])
labels = np.concatenate([train_labels,test_labels])

num_data_instances = len(images)

shuffled_indices = np.random.permutation(num_data_instances)

images = images[shuffled_indices]
labels = labels[shuffled_indices]
```

Important: randomize the ordering of your data!

In [11]:
```
validation_percent = 0.1
testing_percent = 0.2

training_end = int( len(images) * (1 - validation_percent - testing_percent ) )
validation_end = int( len(images) * (1 - testing_percent ) )

training_images = images[:training_end]
training_labels = labels[:training_end]

validation_images = images[training_end:validation_end]
validation_labels = labels[training_end:validation_end]

testing_images = images[validation_end:]
testing_labels = labels[validation_end:]
```

In [34]:
```
print(training_images.shape)
print(training_labels.shape)
print(validation_images.shape)
print(validation_labels.shape)
print(testing_images.shape)
print(testing_labels.shape)
```

```
(49000, 784)
(49000,)
(7000, 784)
(7000,)
(14000, 784)
(14000,)
```

Multiclass Classification

In [17]:
```
model = keras.Sequential([
    layers.Dense(512,activation="relu"),
    layers.Dense(10,activation='softmax')
])

model.compile( optimizer = 'rmsprop',
               loss = 'sparse_categorical_crossentropy',
               metrics = ['accuracy'] )

history = model.fit(training_images,
                    training_labels,
                    verbose= 0,
                    epochs=100,
                    batch_size=64,
                    validation_data=(validation_images,validation_labels))

model.summary()
```

```
Epoch 1/100
Epoch 2/100
Epoch 3/100
Epoch 4/100
Epoch 5/100
Epoch 6/100
Epoch 7/100
Epoch 8/100
Epoch 9/100
Epoch 10/100
Epoch 11/100
Epoch 12/100
Epoch 13/100
Epoch 14/100
Epoch 15/100
Epoch 16/100
Epoch 17/100
Epoch 18/100
Epoch 19/100
```

In [27]:
```
model.summary()
```

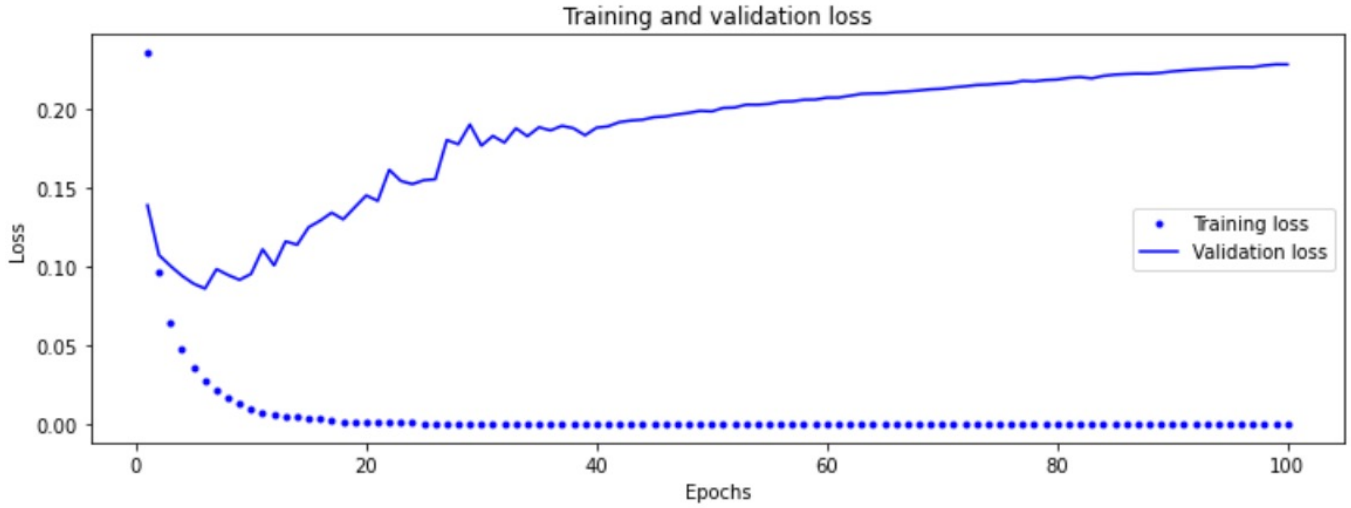
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

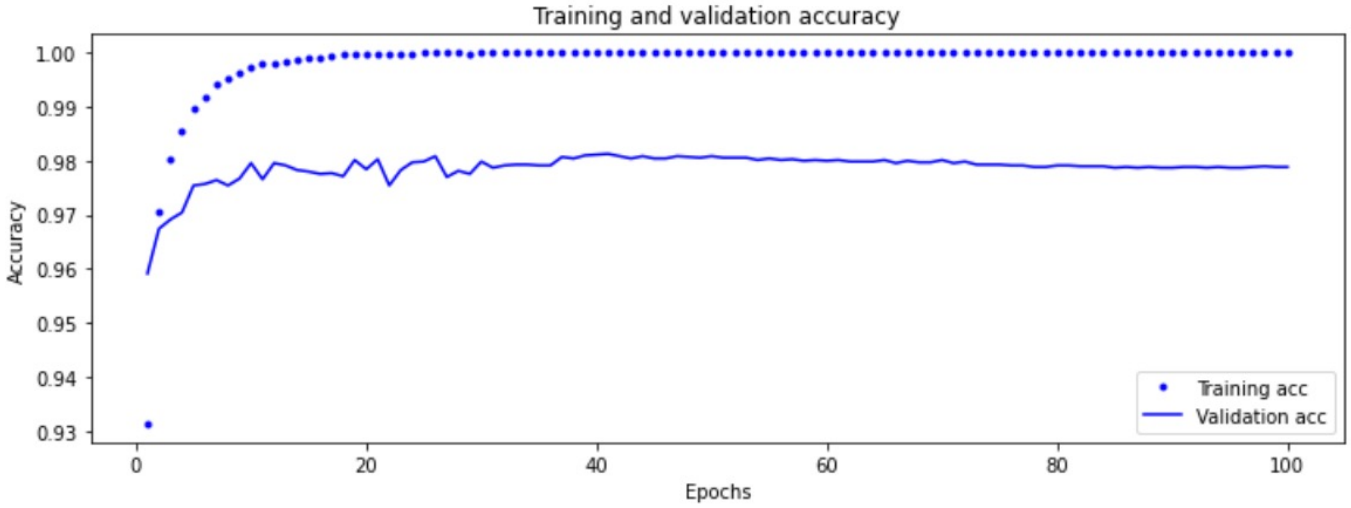
In [18]:
```
model.evaluate(testing_images, testing_labels,batch_size = 128)
```

```
110/110 [==============================] - 0s 2ms/step - loss: 0.2465 - accuracy: 0.9795
```

Out[18]:
```
[0.24646691977977753, 0.9794999957084656]
```

Multiclass Classification






```
Best Validation Accuracy: 0.9813 at epoch  40
```

Multiclass Classification

Remember: Always consider your baseline!

```
In [32]:
1 for d in range(10):
2     print("Count of",d,":",list(train_labels).count(d))
```

```
Count of 0 : 5923
Count of 1 : 6742
Count of 2 : 5958
Count of 3 : 6131
Count of 4 : 5842
Count of 5 : 5421
Count of 6 : 5918
Count of 7 : 6265
Count of 8 : 5851
Count of 9 : 5949
```

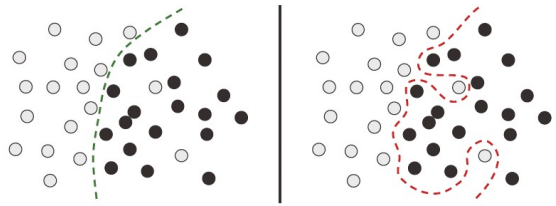
```
In [37]:
1 model.evaluate(testing_images, randint(10,size=len(testing_labels)), batch_size = 128)
```

```
110/110 [==============================] - 0s 2ms/step - loss: 4.1812 - accuracy: 0.1021
Out[37]: [4.181239128112793, 0.10207142680883408]
```

So 98% looks really great compared with random labels!

Generalization and Overfitting

Generalization – the ability of a NN to learn the patterns in a data set so as to perform well on data it has never seen – is the most important issue in deep learning.
The problem is overfitting – the NN is starting to "memorize" the training set without learning the most important patterns which characterize the essential information present in the data.



Overfitting can be seen when the training loss goes down, but the validation loss goes up. In general, you will see the validation accuracy peak at some epoch and then goes down (generally not as noticably as the rise in the validation loss):

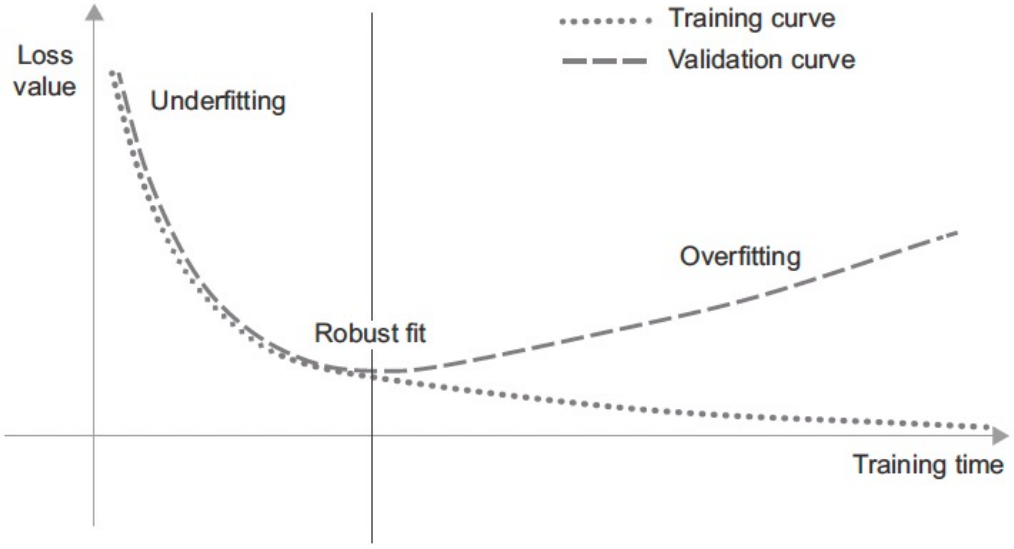
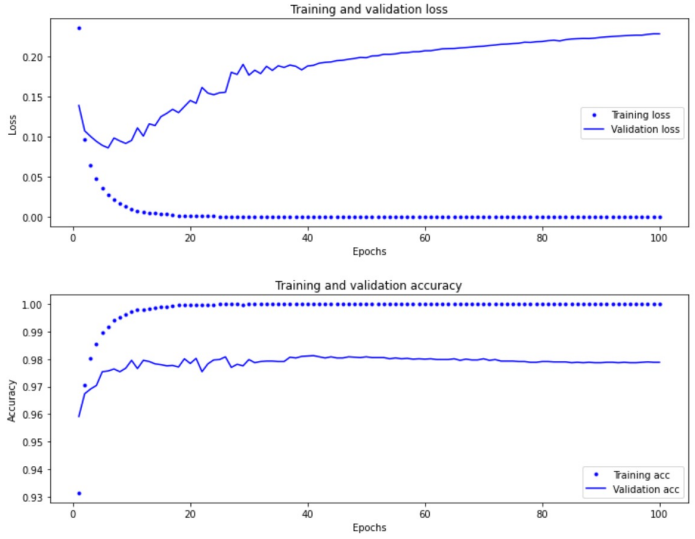


Figure 5.1 Canonical overfitting behavior

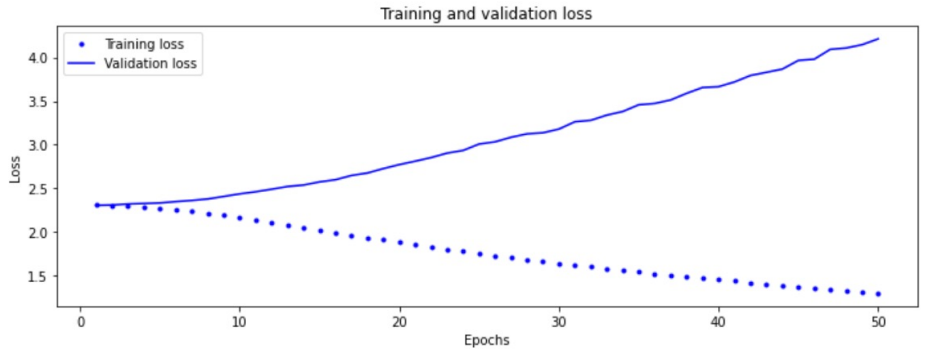


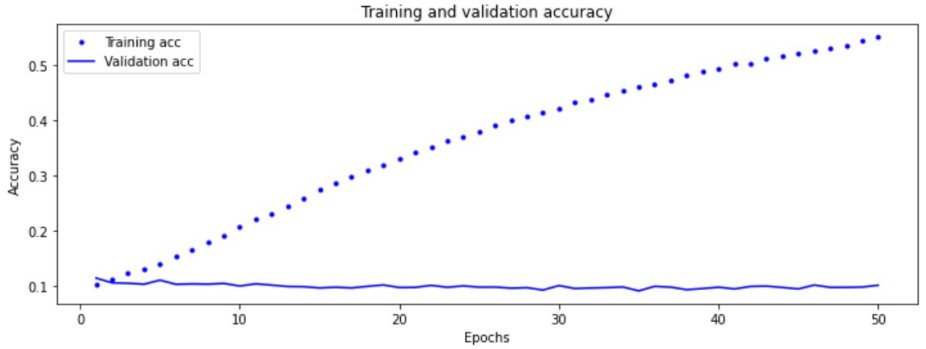

```
Best Validation Accuracy: 0.9813 at epoch  40
```

Generalization: Overfitting

The problem is that a NN can learn ANY data set you give it, essentially by memorizing the exact training set. Here is a dramatic example: we randomly permute the labels, so that there is no correspondence between data and labels. The model continues to "learn" the training set, but the validation accuracy remains around the baseline of 10%.

```
shuffled_indices = np.random.permutation(num_data_instances)
shuffled_indices2 = np.random.permutation(num_data_instances)

images = images[shuffled_indices]
labels = labels[shuffled_indices2]
```





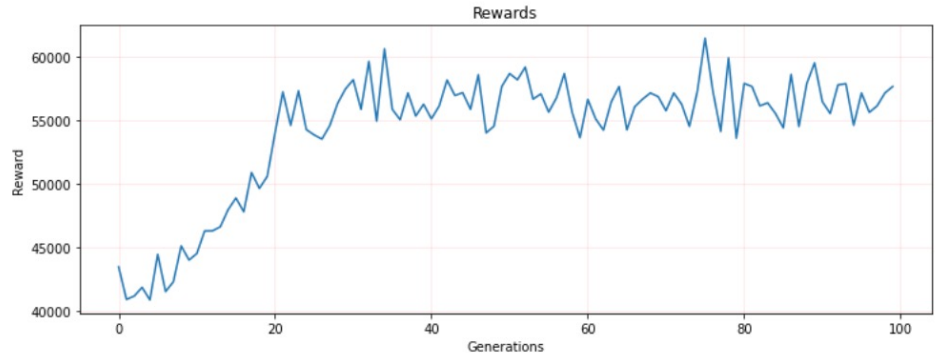

```
In [37]: model.evaluate(testing_images, randint(10,size=len(testing_labels)), batch_size = 128)
110/110 [==============================] - 0s 2ms/step - loss: 4.1812 - accuracy: 0.1021
Out[37]: [4.181239128112793, 0.10207142680883408]
```

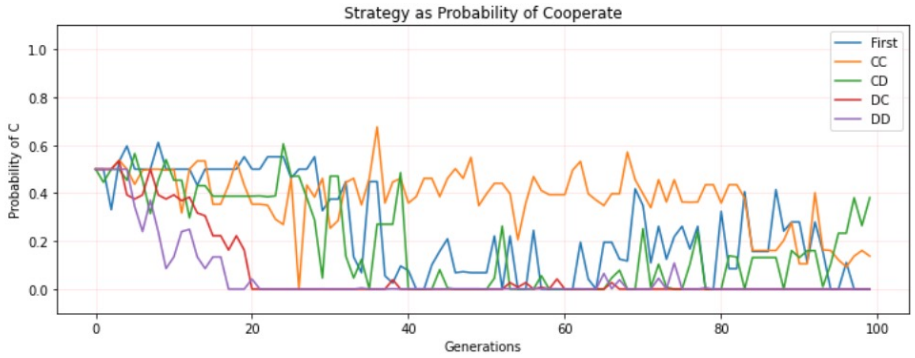
```
Best Validation Accuracy: 0.1139 at epoch  0
```

Generalization

Digression: This is an issue throughout machine learning. In the IPD experiment in HW 05, agents can learn how to do pretty well in an environment of random agents:

```
Random_Env
```



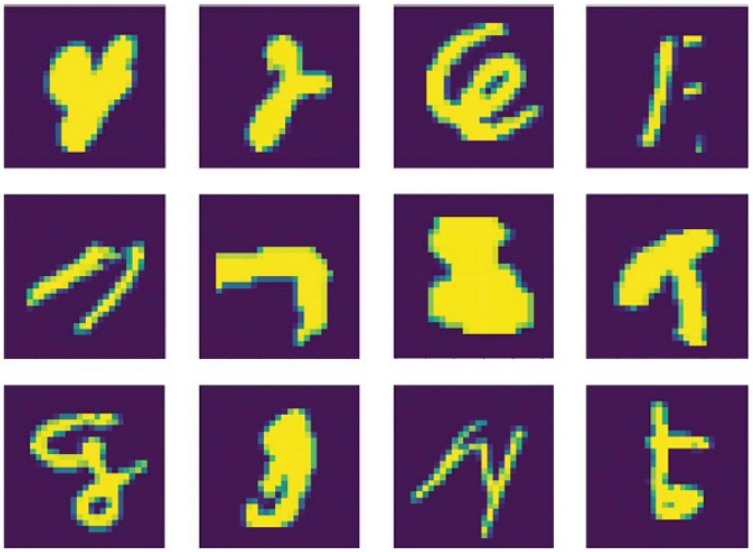


Generalization: Overfitting

Overfitting is often due to data which is

- Noisy (non-data, ambigious, or outliers)
- Mislabeled
- Or has rare features or spurious correlations.

Noisy Data in MNIST:



Mislabeled data in MNIST:

Figure 5.3 Mislabeled MNIST training samples

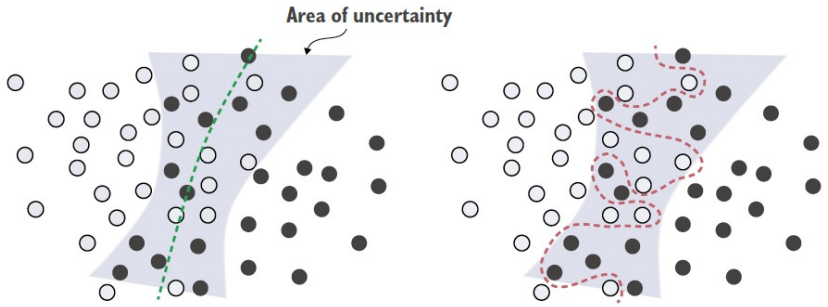


Figure 5.5 Robust fit vs. overfitting giving an ambiguous area of the feature space

Generalization: Overfitting

Overfitting is often due to data which is

- Noisy (non-data, ambigious, or outliers)
- Mislabeled
- Or has rare features or spurious correlations.

Rare features

If your data contains "one-off" features (e.g., a "Getty Images" logo in one image, or a unique or misspelled word in an email), the NN will learn to associate that feature with its label – it is overfitting!

Spurious Correlations

This is actually worse—and more common—than rare features. A word may occur 100s of time in movie reviews, but by a statistical fluke, it occurs in 58% of the positive reviews, and 42% of the negative reviews. The NN will give this word undue weight in learning the data set.

Generalization: Overfitting

Overfitting is often due to data which is

- Noisy (non-data, ambigious, or outliers)
- Mislabeled
- Or has rare features or spurious correlations.

Rare features

If your data contains "one-off" features (e.g., a "Getty Images" logo in one image, or a unique or misspelled word in an email), the NN will learn to associate that feature with its label – it is overfitting!

Spurious Correlations

This is actually worse—and more common—than rare features. A word may occur 100s of time in movie reviews, but by a statistical fluke, it occurs in 58% of the positive reviews, and 42% of the negative reviews. The NN will give this word undue weight in learning the data set.

Generalization: A Deep Dive into the Math

The Manifold Hypothesis

A manifold in an N dimensional space is a set of points which is isomophic to a lower-dimensional space that is Euclidean, i.e., is continuous and has a notion of "distance."

Ex 1: A curved line is literally in 2 D, but can be mapped 1-to-1 (isomophic) to a 1 D line:

Ex 2: A crumpled piece of paper is 3 D, but is isomophic to a 2D (flat) piece of paper:

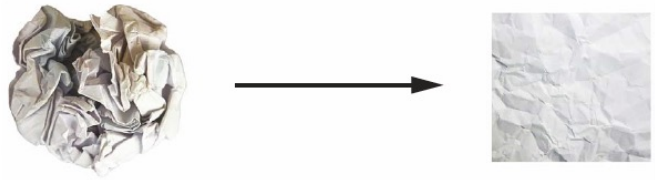


Figure 5.9 Uncrumpling a complicated manifold of data

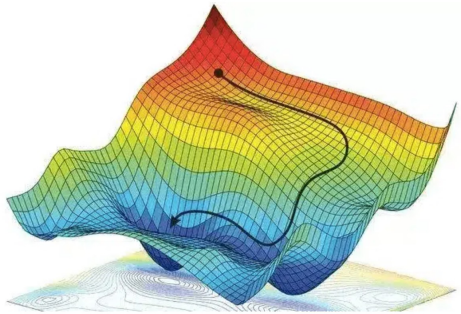
Ex 3:

Möbius strip

Generalization: A Deep Dive into the Math

The Manifold Hypothesis is "that many high-dimensional data sets that occur in the real world actually lie along low-dimensional latent manifolds inside that high-dimensional space. As a consequence of the manifold hypothesis, many data sets that appear to initially require many variables to describe, can actually be described by a comparatively small number of variables, likened to the local coordinate system of the underlying manifold. It is suggested that this principle underpins the effectiveness of machine learning algorithms in describing high-dimensional data sets by considering a few common features." -Wikipedia

Your model is searching in a high-dimensional space (= number of parameters attached as weights to neurons) for a representation of the data (lower dimensional manifold). The spaces are continuous and have a notion of distance, which are intrinsic to the gradient descent algorithm:



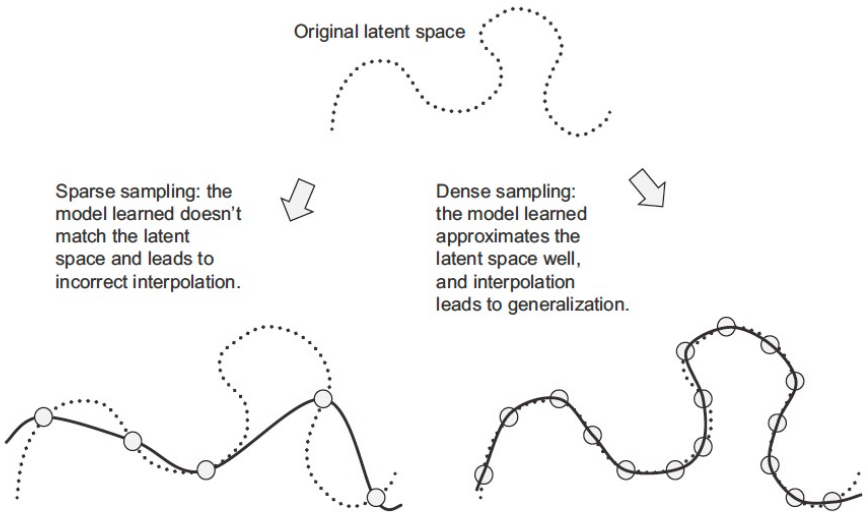


Figure 5.11 A dense sampling of the input space is necessary in order to learn a model capable of accurate generalization.

Generalization: A Deep Dive into the Math

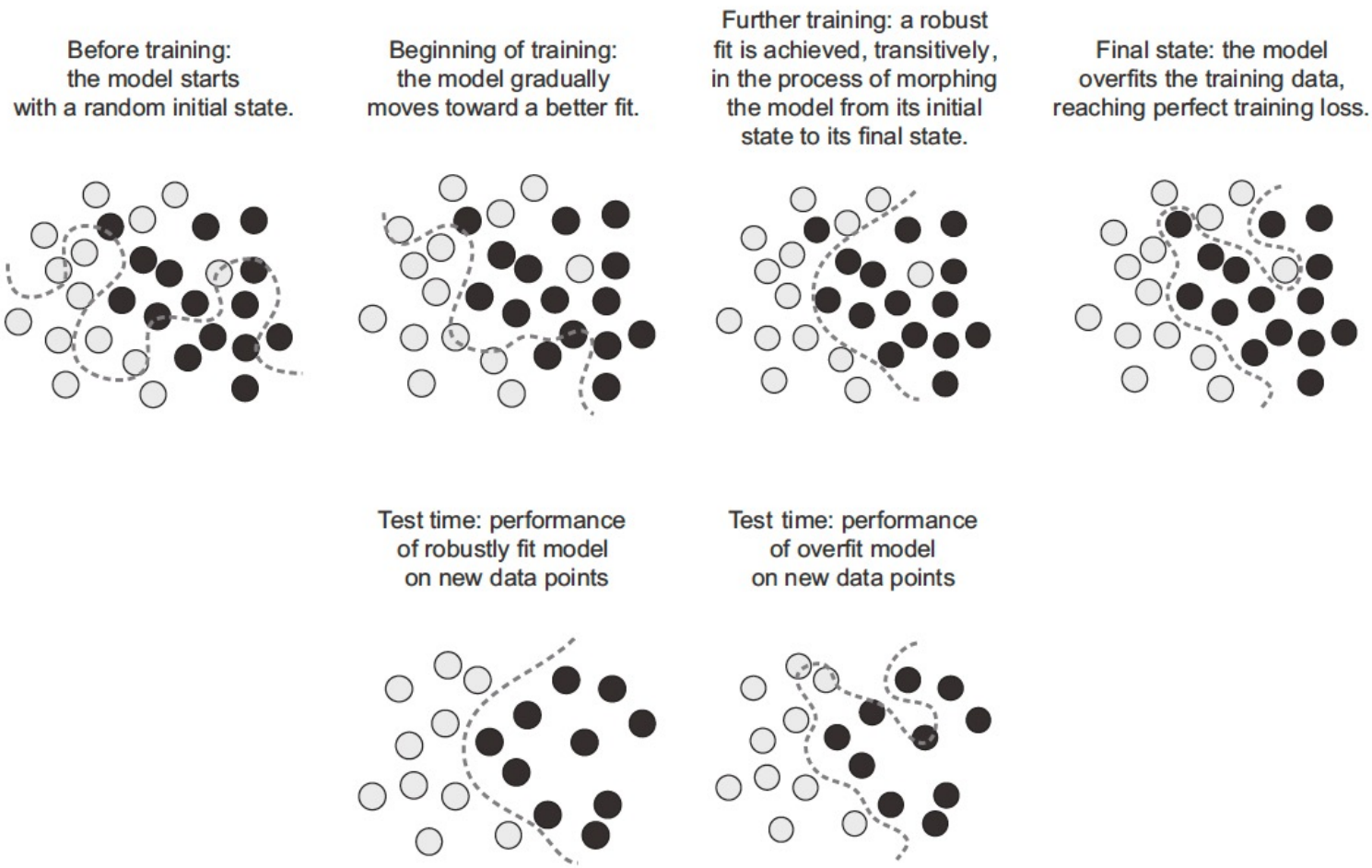


Figure 5.10 Going from a random model to an overfit model, and achieving a robust fit as an intermediate state

Generalization: Underfitting and Overfitting

Overfitting is not a sign that something is wrong with your model, in fact, it shows that your model has sufficient power to represent the patterns that characterize the true "meaning" of the data. You just have to find ways to control this power.

Chollet, p.138: "The first big milestone of a machine learning project: getting a model that has some generalization power (it can beat a trivial baseline) and that is able to overfit." p.141: "Remember that it should always be possible to overfit."

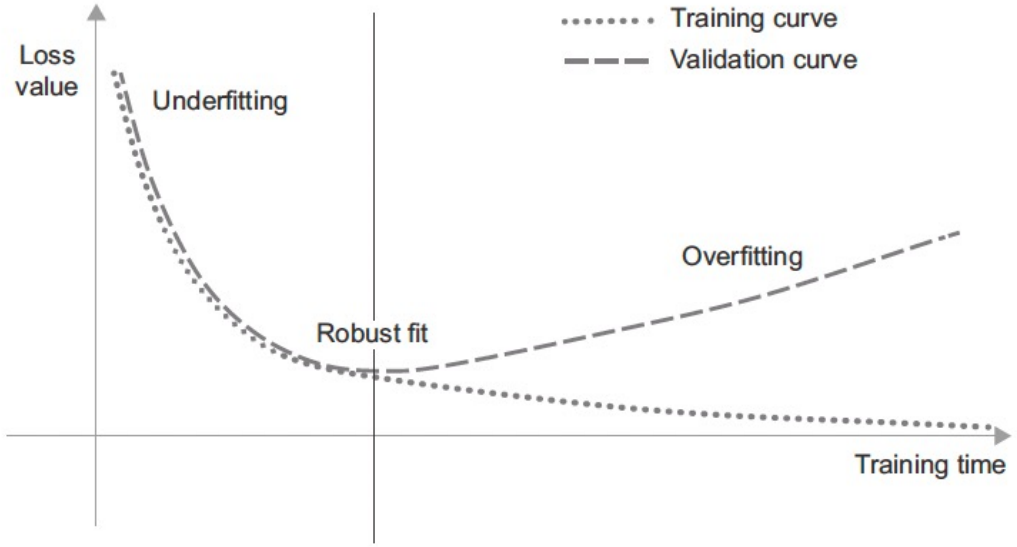


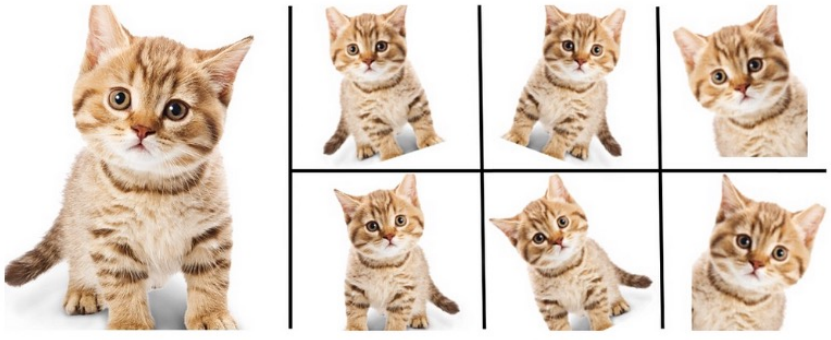
Figure 5.1 Canonical overfitting behavior

Generalization

Improving generalization can be accomplished by various techniques.

Getting more data, improving your data: more data is almost always better; make sure there are minimal labeling errors, reconsider your data normalization.

If you can not get more data, consider data augmentation: manipulating your existing data in ways that produce different samples with the same essential information.



Generalization

Improving generalization can be accomplished by various techniques.

Tuning hyperparameters: Play with the hyperparameters, including learning rate and batch size.

Better feature engineering: Use domain knowledge about the data, and experience with the model you are using to better represent the data. Tools can help with feature selection (find out which features are making the most difference).

Example: How to represent a clock face: pixels, clock hands' coordinates, angles:

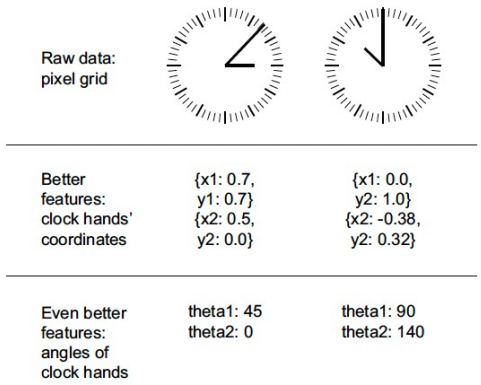


Figure 5.16 Feature engineering for reading the time on a clock

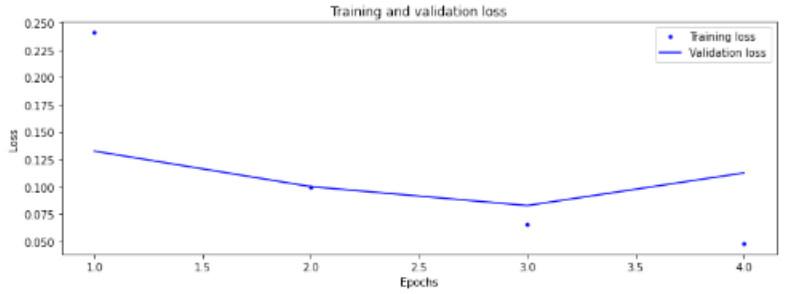
Generalization

Improving generalization can be accomplished by various techniques.

Early Stopping: Stop training when a robust fit is achieved. This can often be done automatically by setting a parameter in your model.

Here is a naive example of early stopping, which does not do so well:

```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping()
custom_early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=8,
    min_delta=0.001,
    mode='max'
)


model = keras.Sequential([
    layers.Dense(512,activation="relu"),
    layers.Dense(10,activation='softmax')
])

model.compile( optimizer = 'rmsprop',
               loss = 'sparse_categorical_crossentropy',
               metrics = ['accuracy'] )

history = model.fit(training_images,
                    training_labels,
                    verbose= 0,
                    epochs=50,
                    batch_size=64,
                    callbacks=[early_stopping],
                    validation_data=(validation_images,validation_labels))

display_results(history)
model.evaluate(testing_images, testing_labels,batch_size = 128)
```




```
Best Validation Accuracy: 0.9749 at epoch  2
110/110 [==============================] - 0s 1ms/step - loss: 0.1168 - accuracy: 0.9689

[0.11679539084434509, 0.9688571691513062]
```

Generalization

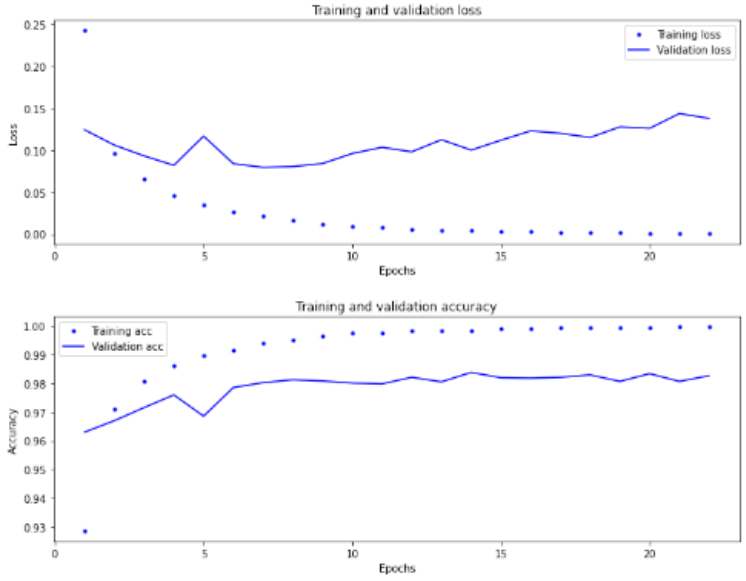
Improving generalization can be accomplished by various techniques.

Early Stopping: Stop training when a robust fit is achieved. This can often be done automatically by setting a parameter in your model.

Tuning the early stopping callback results in better results:

In [37]:

```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping()
custom_early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=8,
    min_delta=0.001,
    mode='max'
)


model = keras.Sequential([
    layers.Dense(512,activation="relu"),
    layers.Dense(10,activation='softmax')
])

model.compile( optimizer = 'rmsprop',
               loss = 'sparse_categorical_crossentropy',
               metrics = ['accuracy'] )

history = model.fit(training_images,
                    training_labels,
                    verbose= 0,
                    epochs=50,
                    batch_size=64,
                    callbacks=[custom_early_stopping],
                    validation_data=(validation_images,validation_la

display_results(history)
model.evaluate(testing_images, testing_labels,batch_size = 128)
```




```
Best Validation Accuracy: 0.9839 at epoch  13
```

In [36]:

```
model.evaluate(testing_images, testing_labels,batch_size = 128)
```

```
110/110 [==============================] - 0s 1ms/step - loss: 0.1475 - accuracy: 0.9807
```

Out[36]:

```
[0.14754530787467957, 0.9807142615318298]
```

Generalization

Regularization: Various techniques which "actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation." The model is simpler, more "regular."

Reduce model size (but not too small):

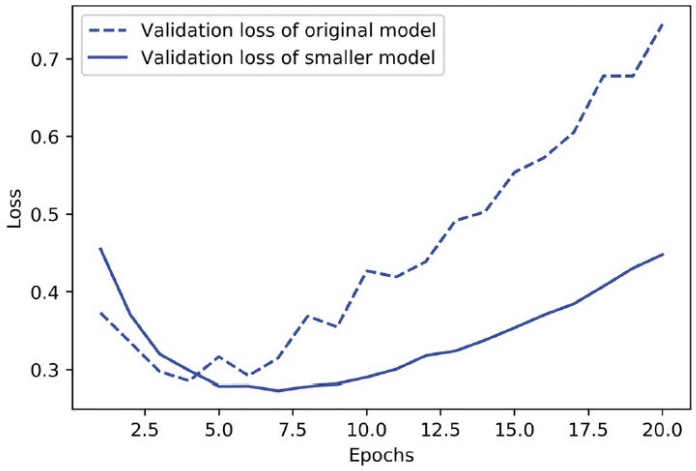


Figure 5.17 Original model vs. smaller model on IMDB review classification

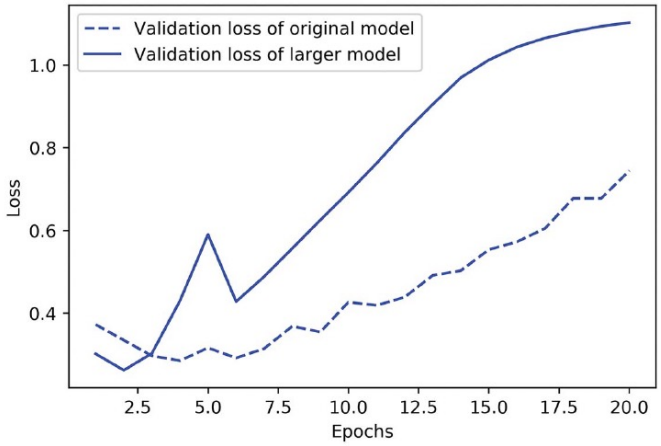


Figure 5.18 Original model vs. much larger model on IMDB review classification

Generalization

Regularization: Various techniques which "actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation." The model is simpler, more "regular."

Weight Regularization: Place limits on how large the weights in the model can become, so that the model is forced to be simpler (having fewer possibilities of weights).
There are two flavors:

- *L1 regularization*—The cost added is proportional to the *absolute value of the weight coefficients* (the *L1 norm* of the weights).
- *L2 regularization*—The cost added is proportional to the *square of the value of the weight coefficients* (the *L2 norm* of the weights). L2 regularization is also called *weight decay* in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the same as L2 regularization.

Listing 5.13 Adding L2 weight regularization to the model

```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

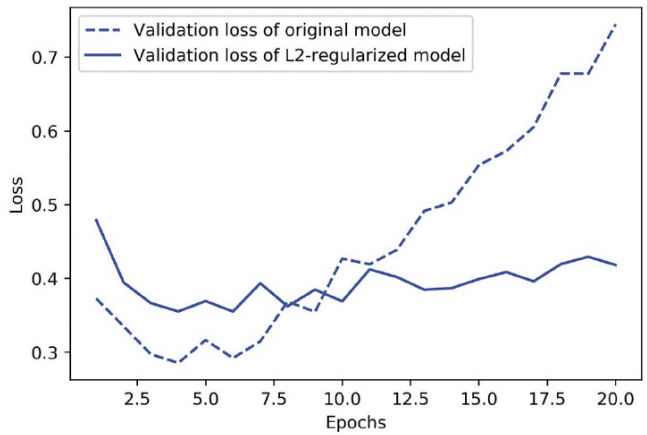


Figure 5.19 Effect of L2 weight regularization on validation loss

Generalization

Regularization: Various techniques which "actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation." The model is simpler, more "regular."

Adding Dropout: Dropout is applied to a layer, and is very simple: with some probability p, drop out (as in, setting to 0.0) the outputs from the layer.

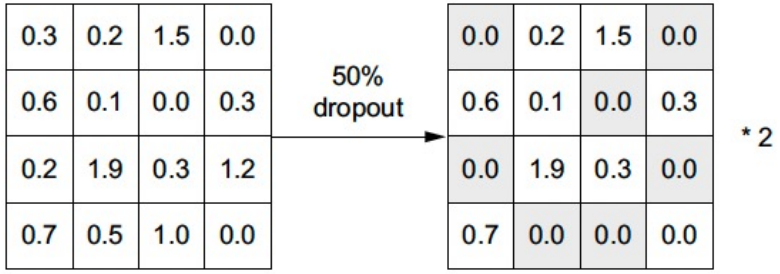


Figure 5.20 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time the activation matrix is unchanged.

This is one of the weirdest great ideas in Deep Learning: it seems like it can't possibly help, but it is one of the most effective and most common ways to regularize your model.

Listing 5.15 Adding dropout to the IMDB model

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_dropout = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

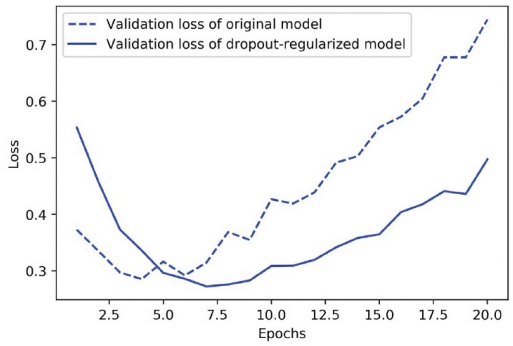


Figure 5.21 Effect of dropout on validation loss

Generalization

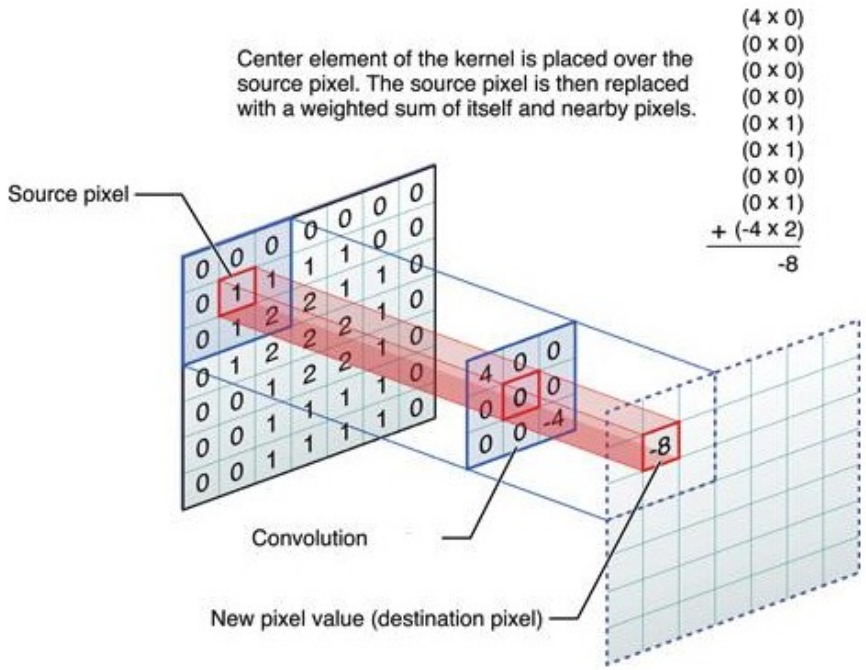
Regularization: Various techniques which "actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation." The model is simpler, more "regular."

Reconsidering your choice of architecture: Add more layers, or fewer, or of different widths.
Consider starting with wider layers, and getting narrower as you go deeper.
Consider different kinds of layers better suited to your data. Google around to see what others have done successfully with similar data.

For example, complex image data is almost always processed using a completely different kind of layer.....

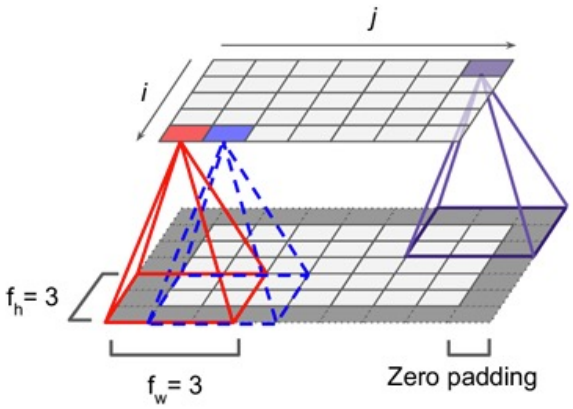
Convolutional Neural Networks

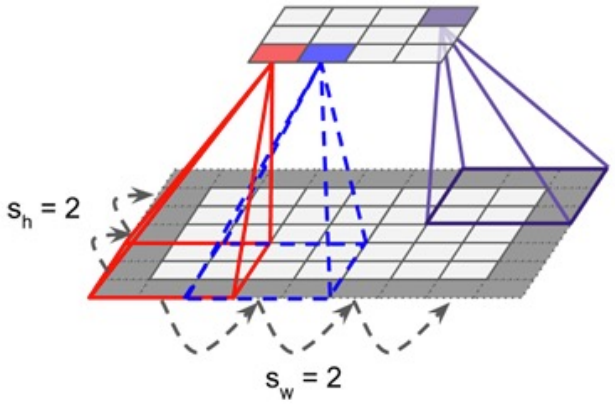
CNNs add convolution and pooling layers to focus on small regions of the data (here, images). Each input to the next layer is calculated as the dot product of the convolution kernel with a small region of the image.



Convolutional Neural Networks

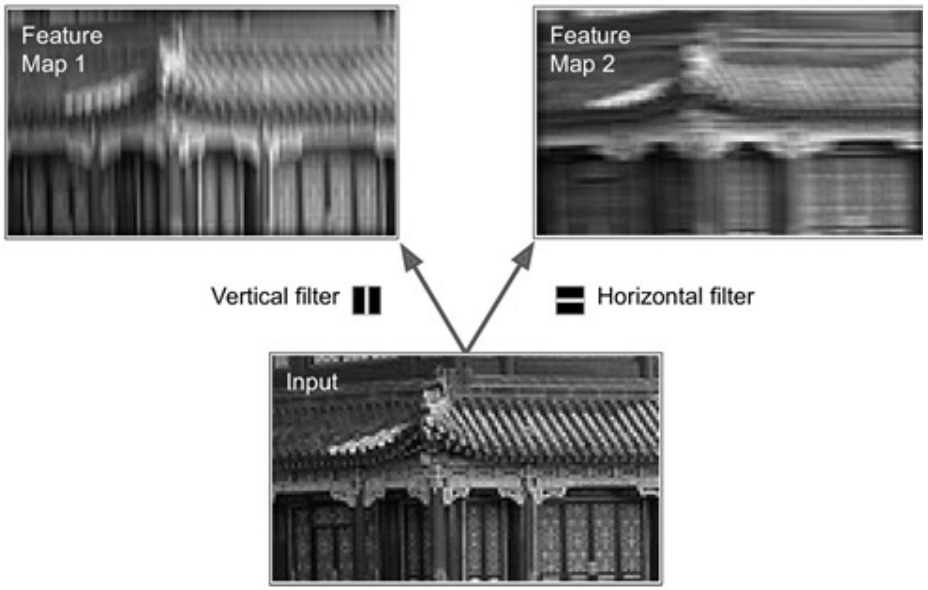
The convolution kernels are moved around the image, perhaps by some skip....





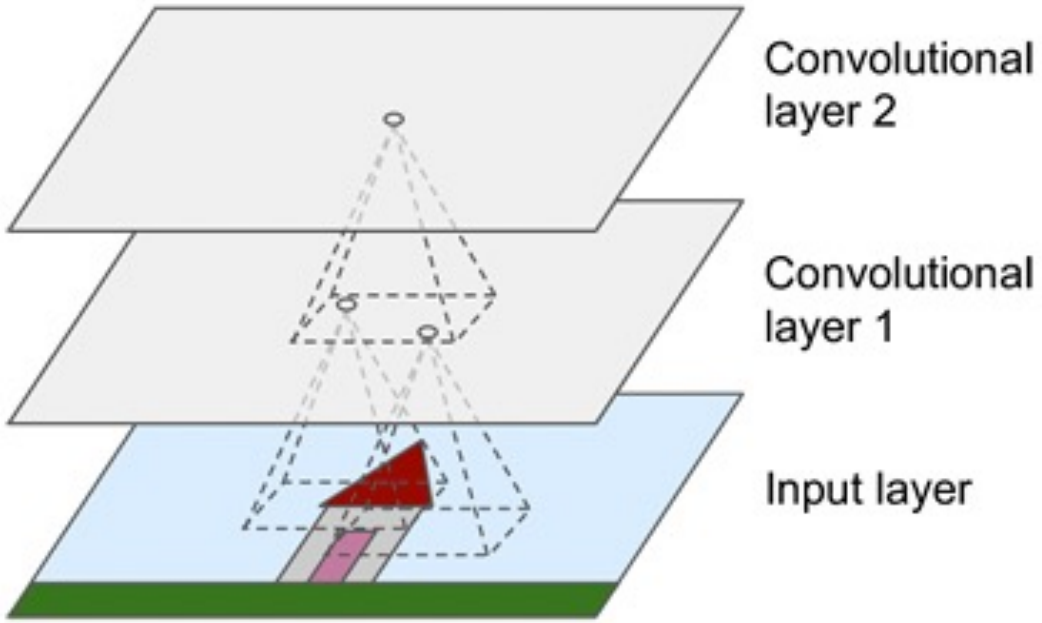
Convolutional Neural Networks

Since data is shared in the region covered by the kernel, various “features” of the image can be recognized in multiple places around the image:



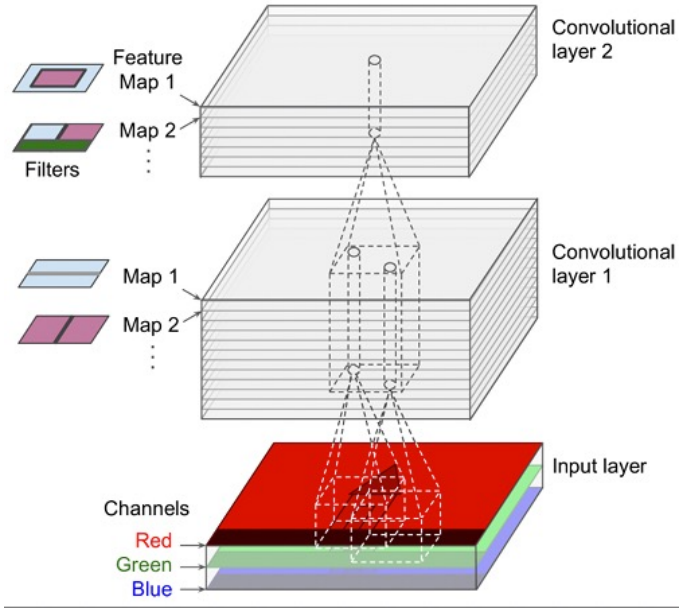
Convolutional Neural Networks

And layers can be stacked.....



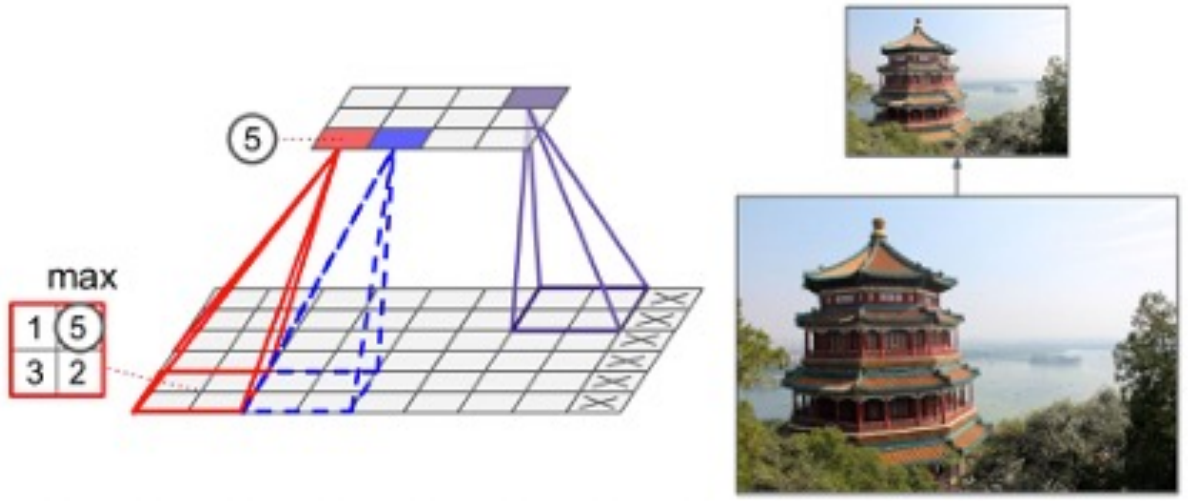
Convolutional Neural Networks

And layers can be stacked.....



Convolutional Neural Networks

A pooling layer reduces the dimensionality by averaging (or taking max) of small regions in the previous layer:



Convolutional Neural Networks

It is typical to alternate convolution and pooling layers and end with fully connected layers before output:

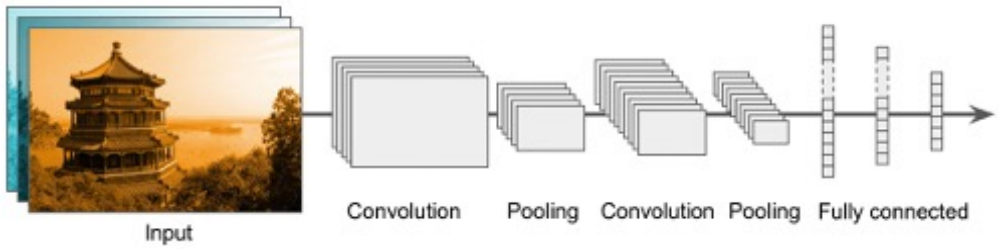


Table 14-1. LeNet-5 architecture

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	–	–	–

Convolutional Neural Networks

Convolutional networks are built similarly to FF networks, but with different components; you can specify all relevant hyperparameters:

Listing 8.1 Instantiating a small convnet

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```