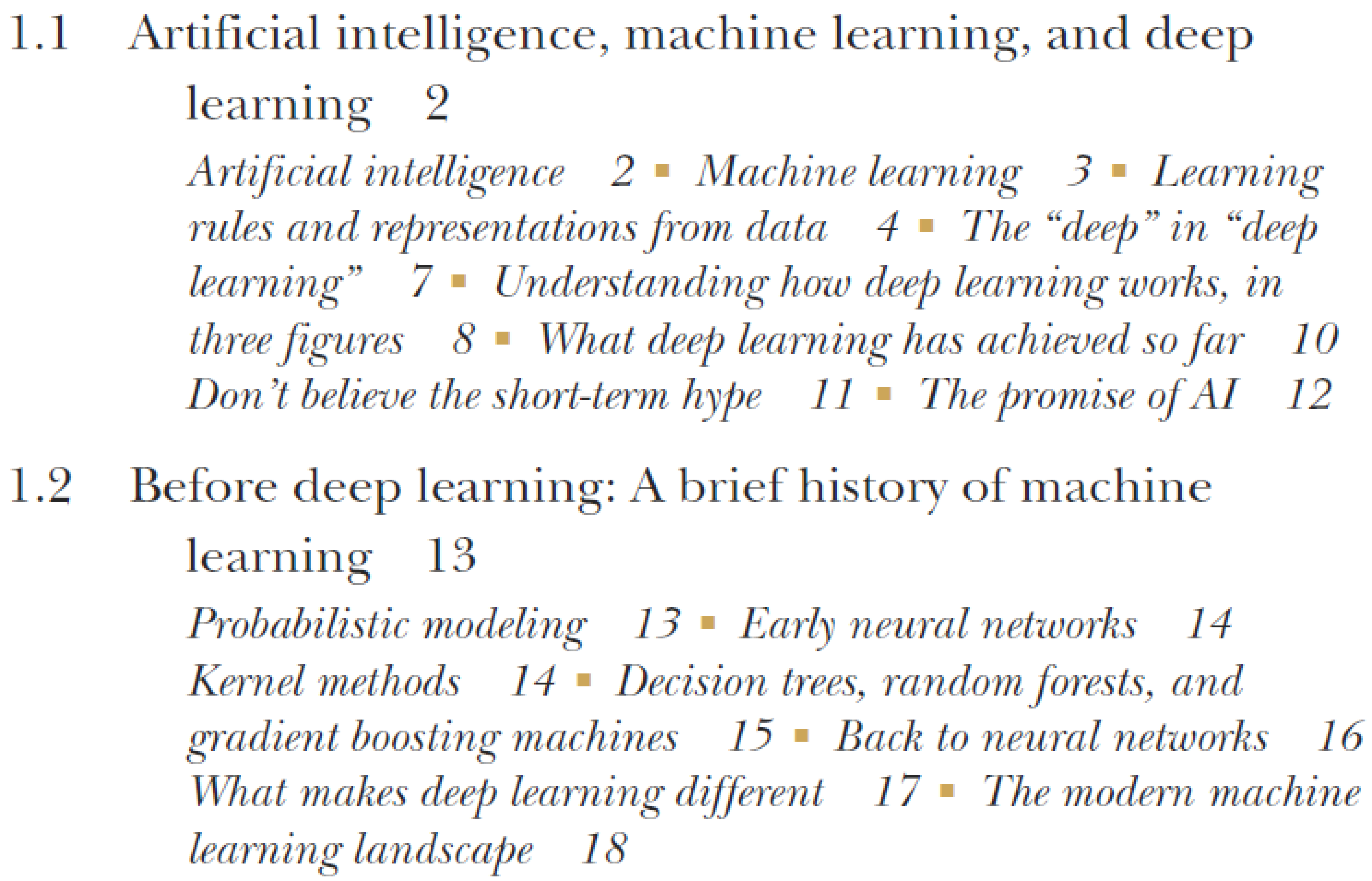
[DLP] Introduction

Oct 6, 2022

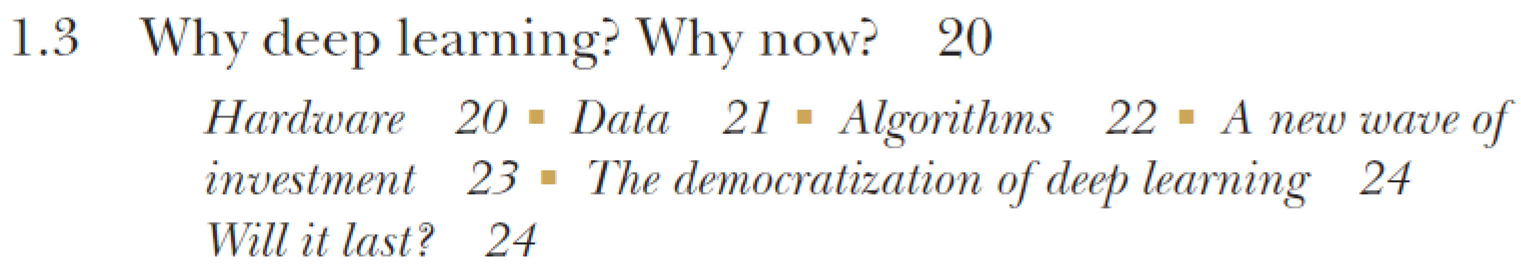
ddebarr@uw.edu

https://cross-entropy.net/ML530/Deep_Learning_1.pdf

What is Deep Learning?



What is Deep Learning?



Notebooks

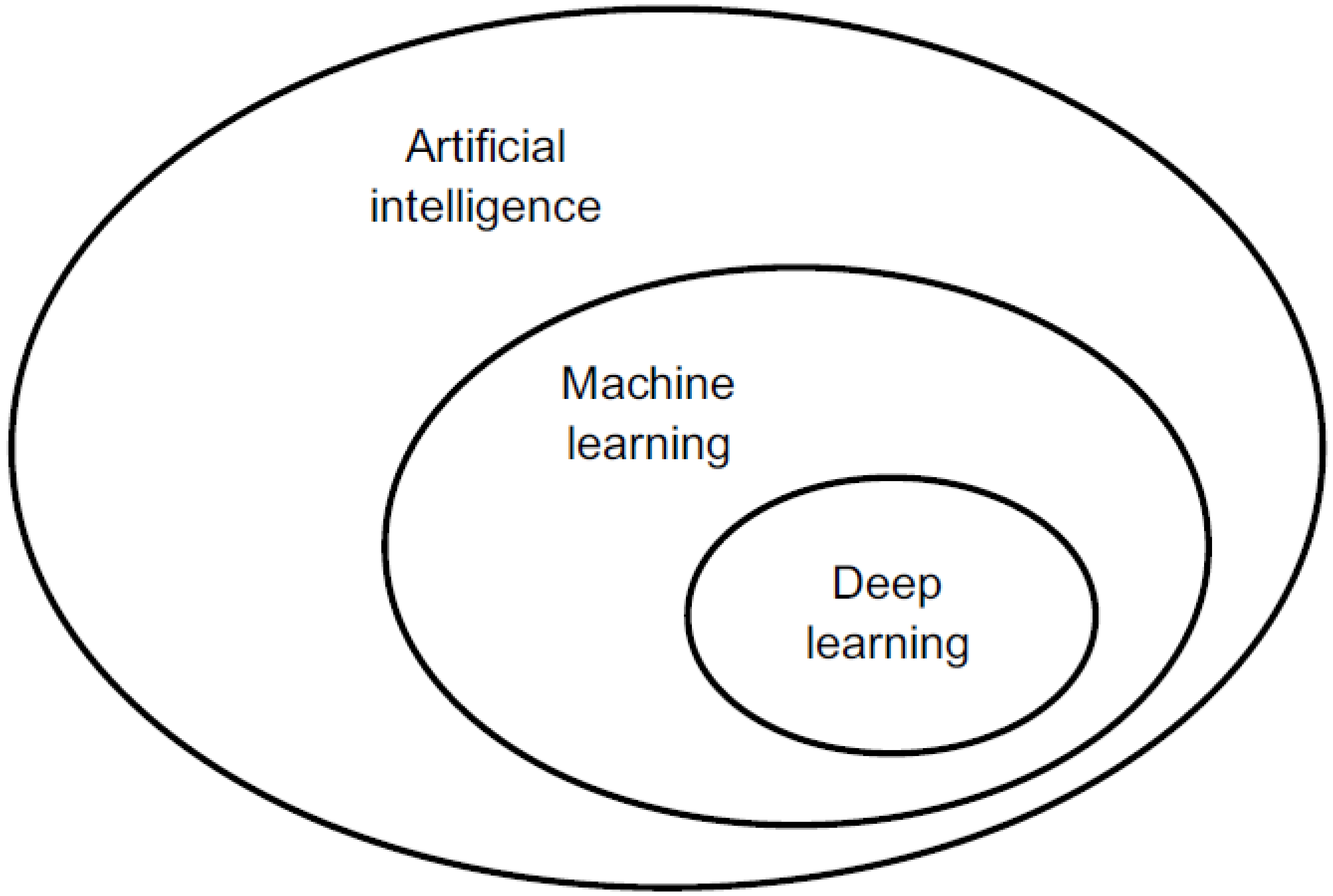
Notebooks on GitHub:

- https://github.com/fchollet/deep-learning-with-python-notebooks

Discussion Forum:

- https://livebook.manning.com/book/deep-learning-with-python-second-edition/discussion

AI > ML > DL



Concise Definitions

- AI can be described as the effort to automate intellectual tasks normally performed by humans
- Machine learning is ... searching for useful representations and rules over some input data, within a predefined space of possibilities, using guidance from a feedback signal

Ada Lovelace, 1843

“The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. . . . Its province is to assist us in making available what we’re already acquainted with.”

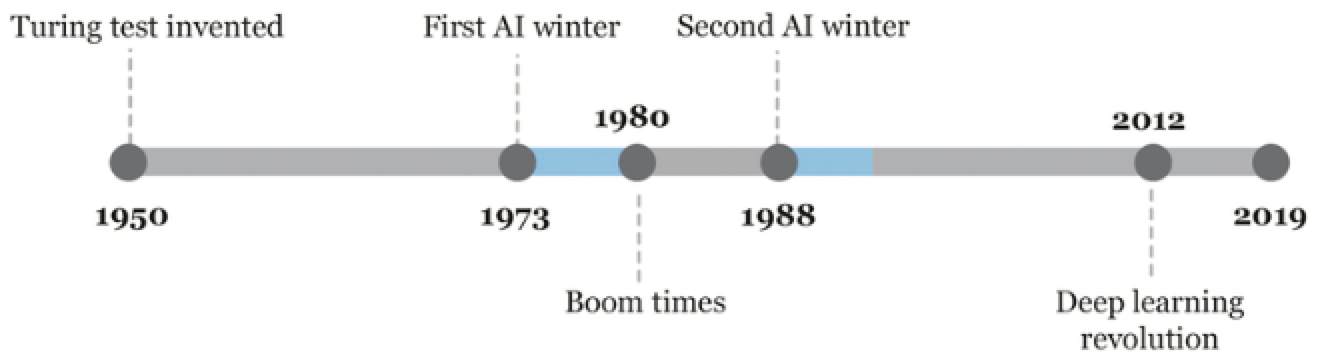
John McCarty, Dartmouth

1956: Summer Workshop Proposal

- The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.
- An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves.
- We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.

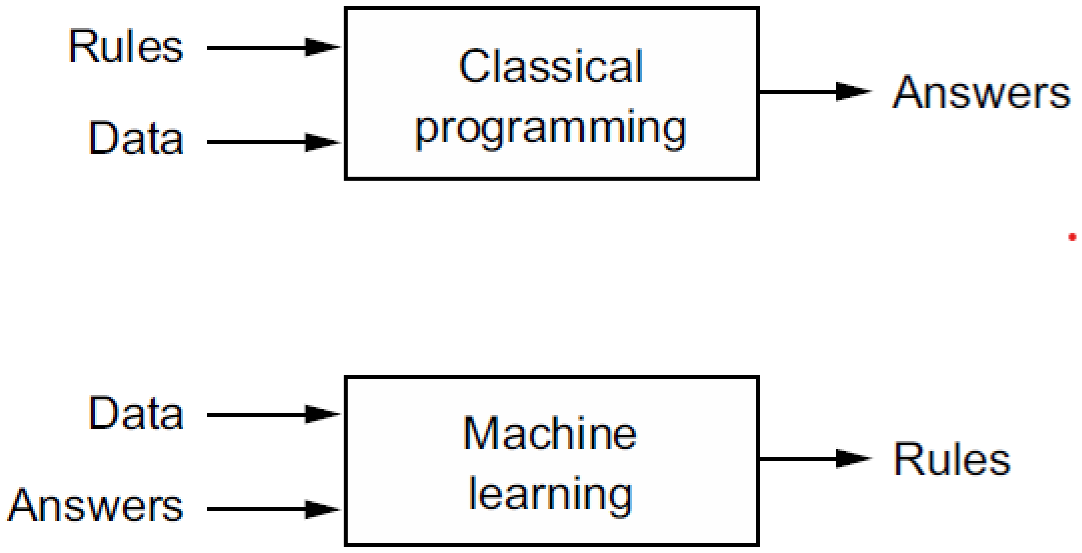
AI Winters

- Winter: researchers and funds turn away from the field
- Hype surrounding the possibility of a “machine with the general intelligence of an average human being”; but it failed to materialize
- Initial success stories for expert systems fueled investment; but they proved to be expensive to maintain, difficult to scale, and limited in scope



https://www.researchgate.net/figure/Timeline-of-the-AI-winters_fig1_333039347

Machine Learning vs Classical Programming

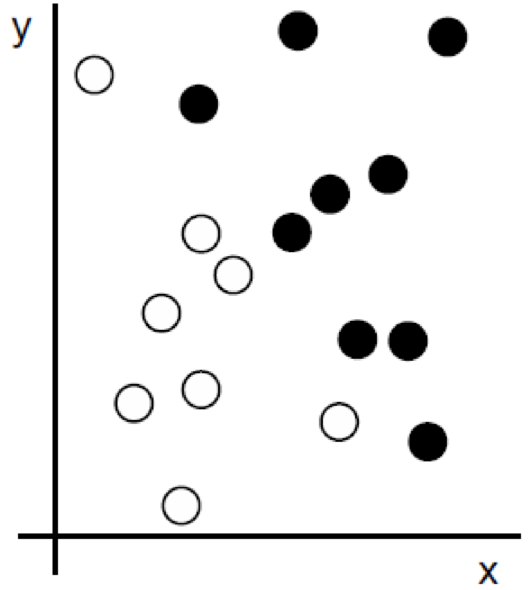


Requirements for Machine Learning

1. Input data points
2. Examples of the expected output
3. A way to measure whether the algorithm is doing a good job

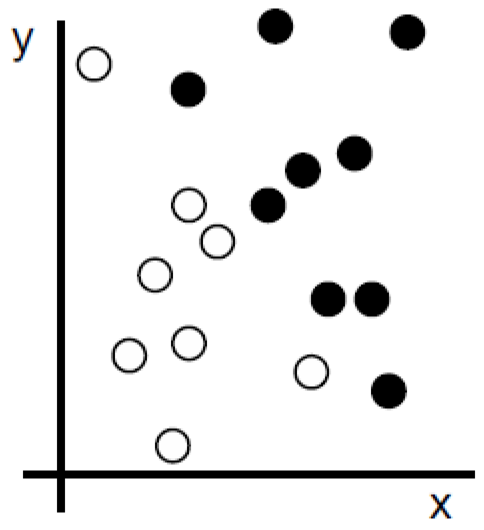
Example Problem

1. The inputs are the coordinates of our points.
2. The expected outputs are the colors of our points.
3. A way to measure whether our algorithm is doing a good job could be, for instance, the percentage of points that are being correctly classified.

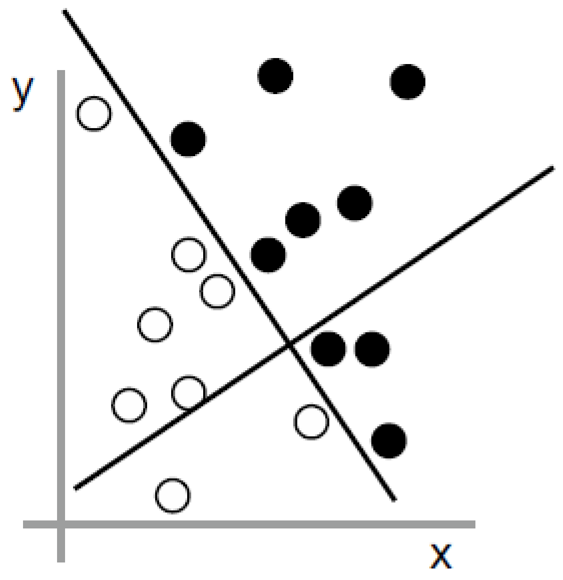


Improved Representation

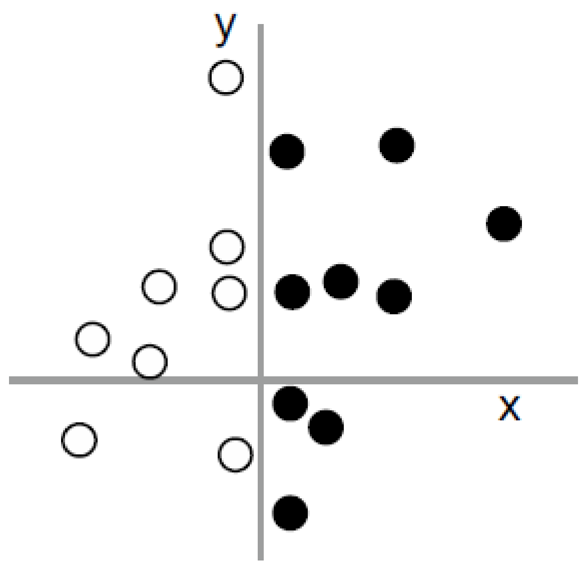
1: Raw data



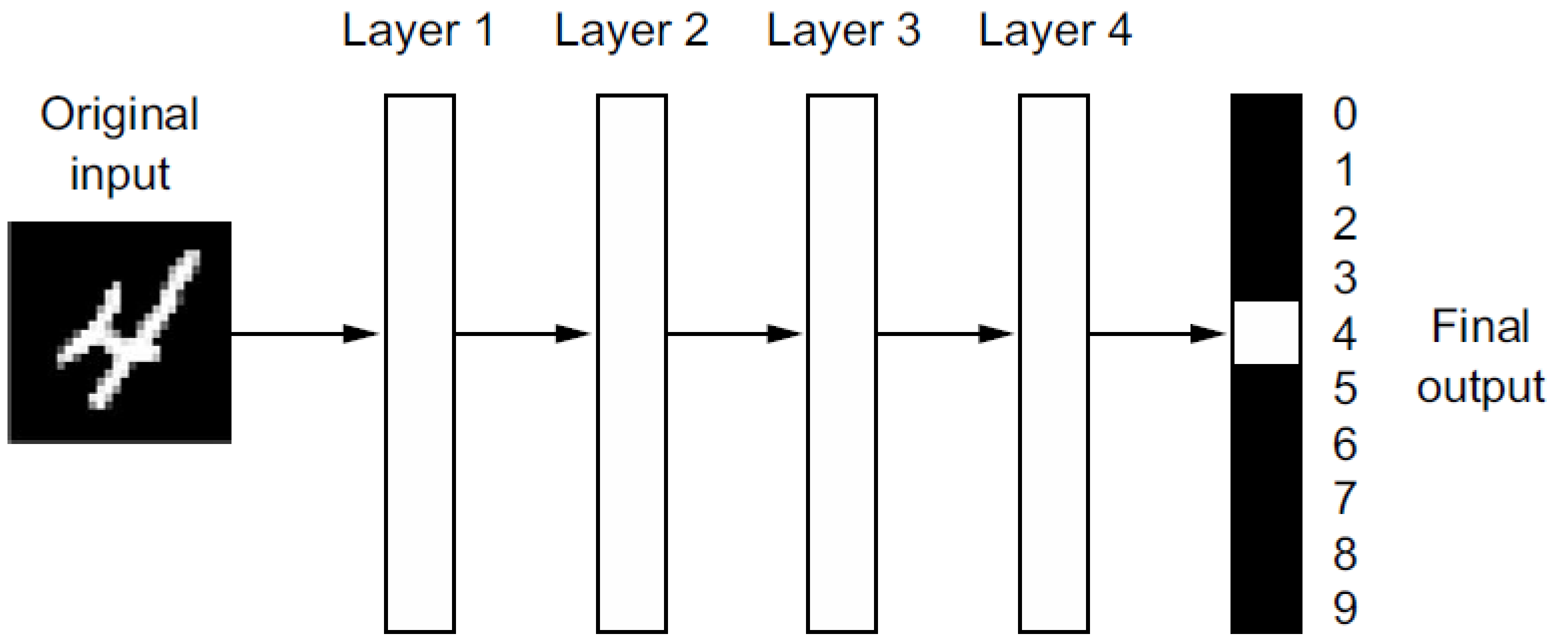
2: Coordinate change



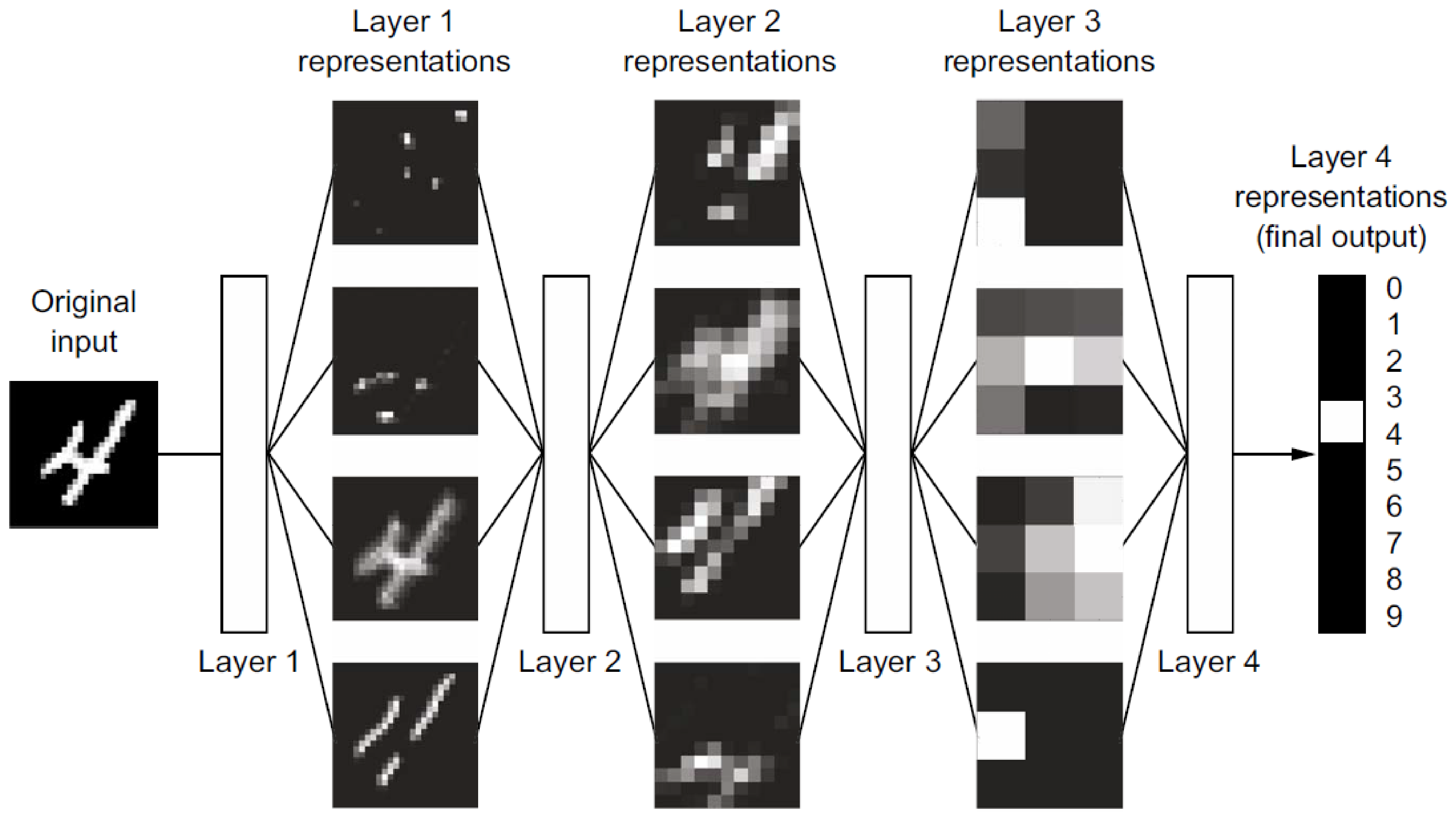
3: Better representation



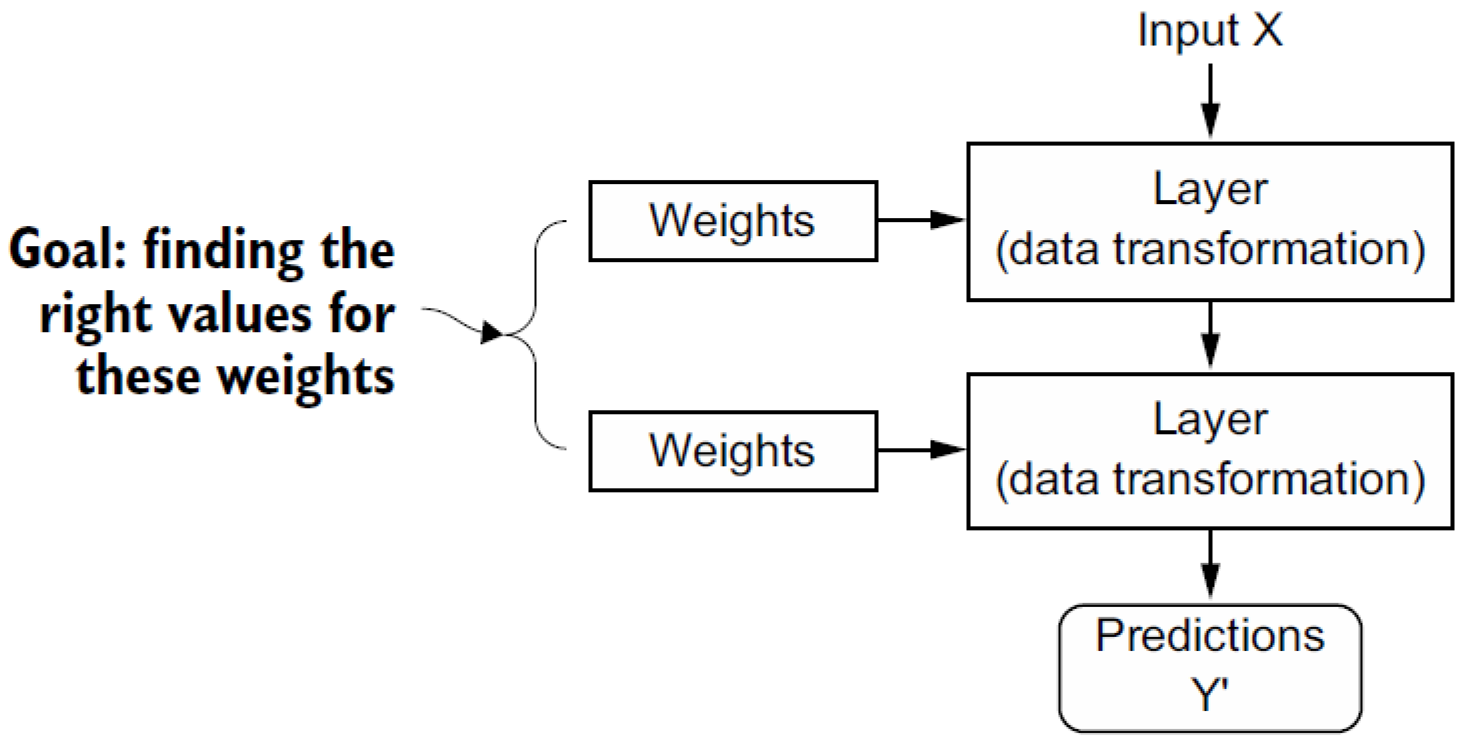
Deep Neural Network for Digit Classification



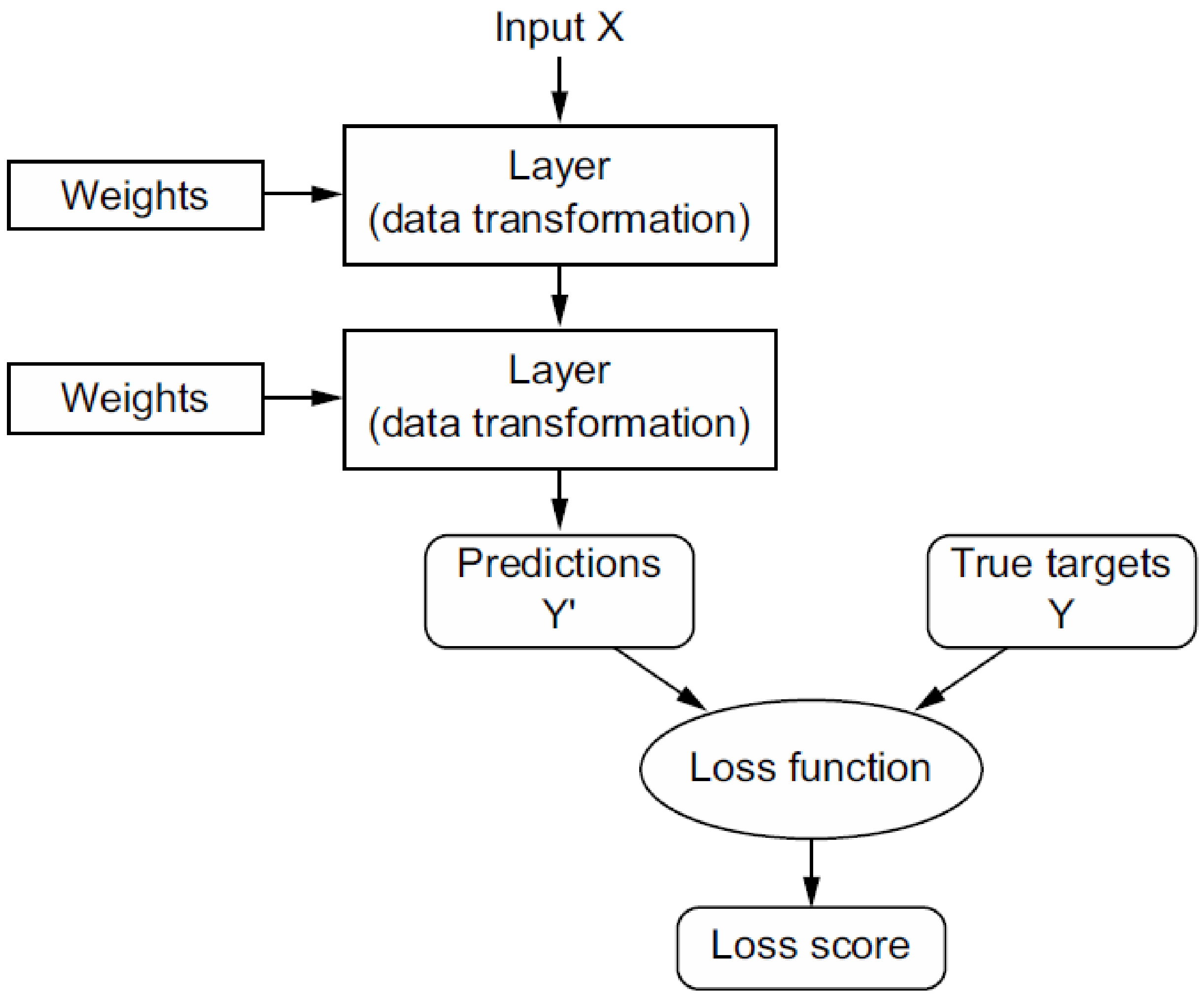
Learned Data Representations



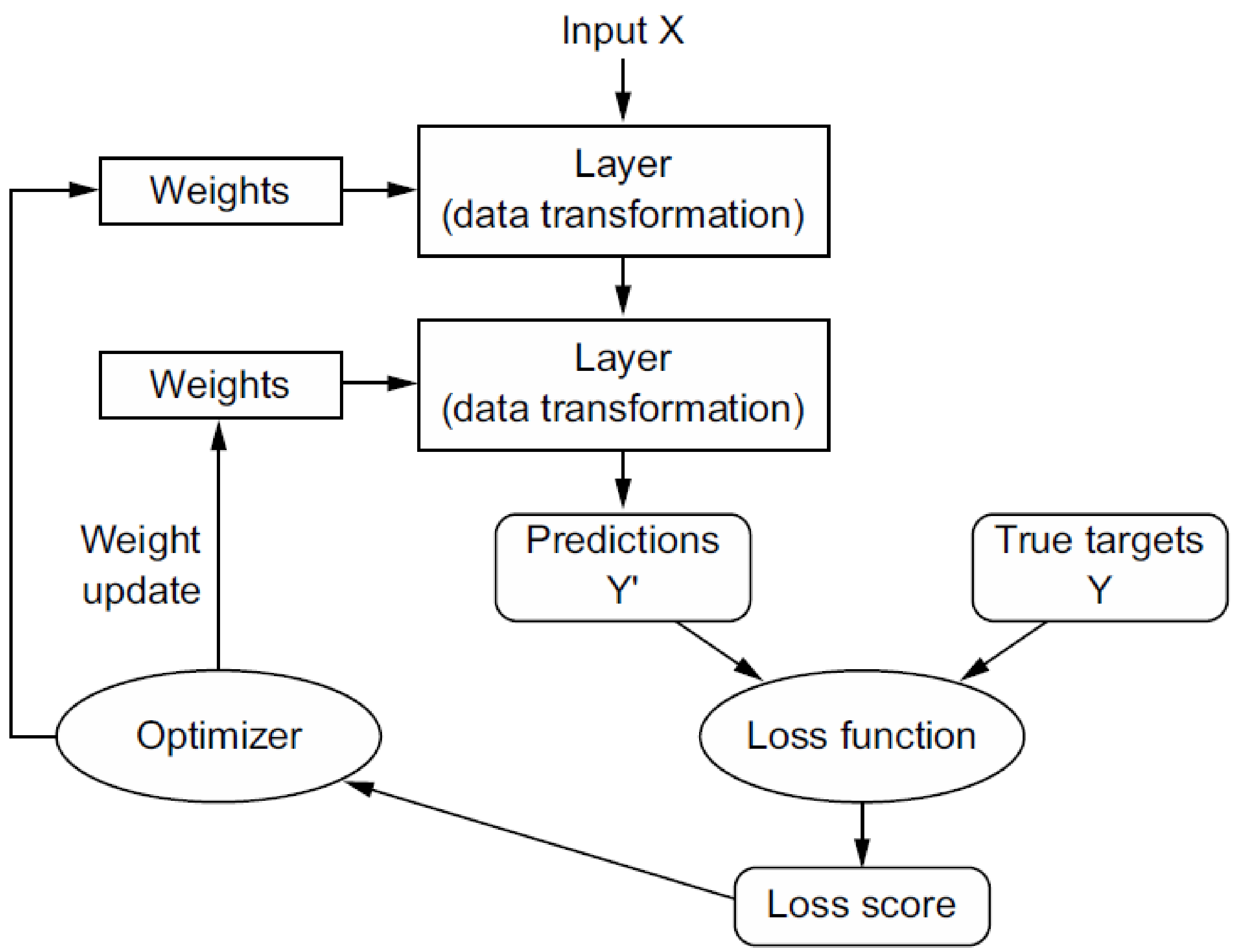
Neural Network is Parameterized by Weights



Loss Function Measures Prediction Quality



Loss Function is Used to Update Weights



Deep Learning Breakthroughs include …

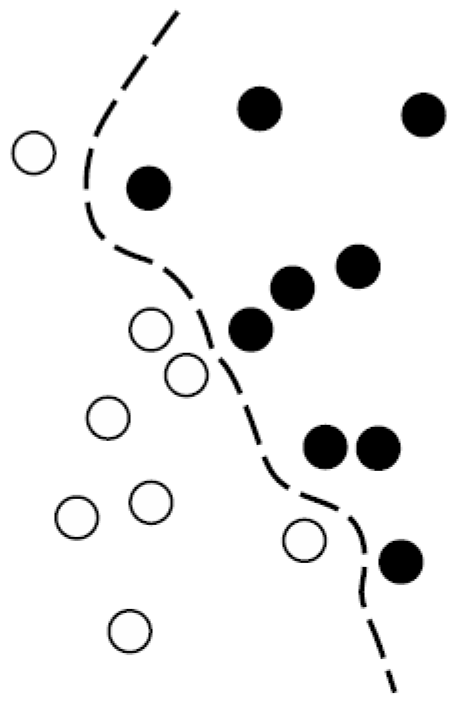
- Near-human-level image classification
- Near-human-level speech transcription
- Near-human-level handwriting transcription
- Dramatically improved machine translation
- Dramatically improved text-to-speech conversion
- Digital assistants such as Google Assistant and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, or Bing
- Improved search results on the web
- Ability to answer natural language questions
- Superhuman Go playing

Alternative Machine Learning Methods

- Probabilistic Modeling
  - Naïve Bayes vs Logistic Regression: Which generative? Which discriminative?
- Shallow Neural Networks
  - Multi-Layer Perceptron (MLP) with one hidden layer and one output layer
- Kernel Methods
  - Support Vector Machines and Gaussian Processes
  - Loss Function for SVMs?
- Decision Trees, Random Forests, and Gradient Boosting Machines
  - Bootstrap Aggregation (bagging) vs Boosting

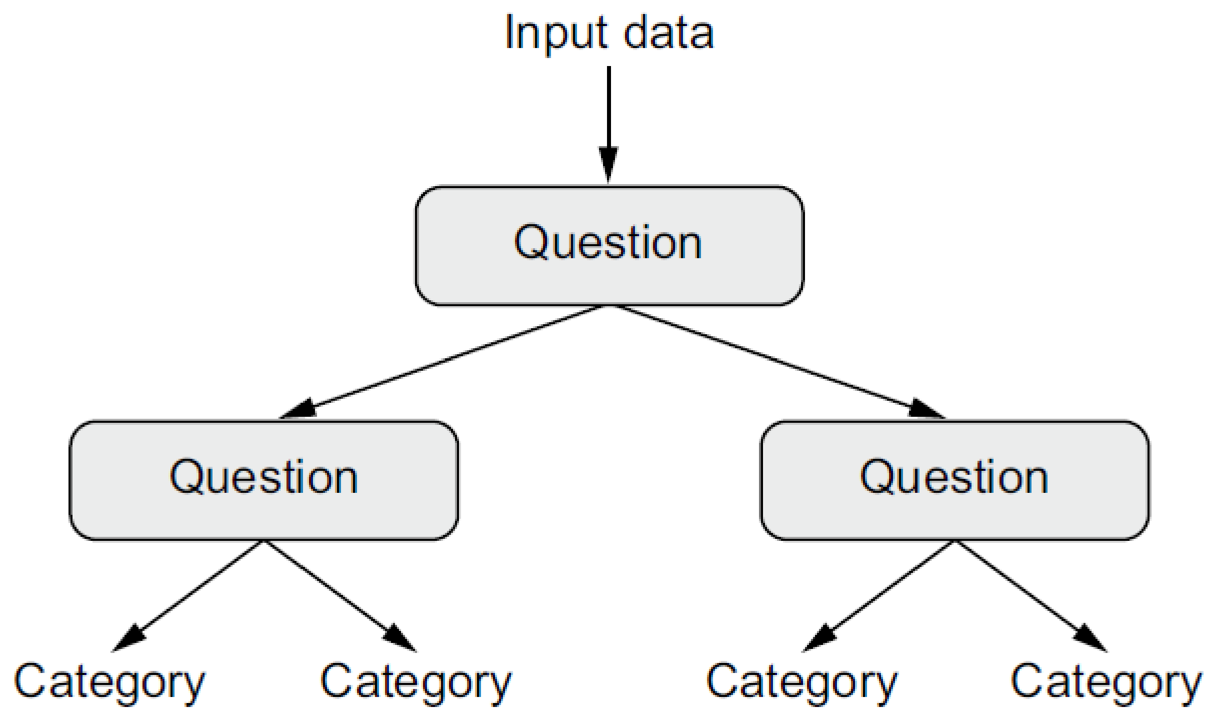
Example Support Vector Machine (SVM)

Where do we find support vectors?



Example Decision Tree

- Questions can incorporate either categorical or numeric features
- What can we do about missing values?

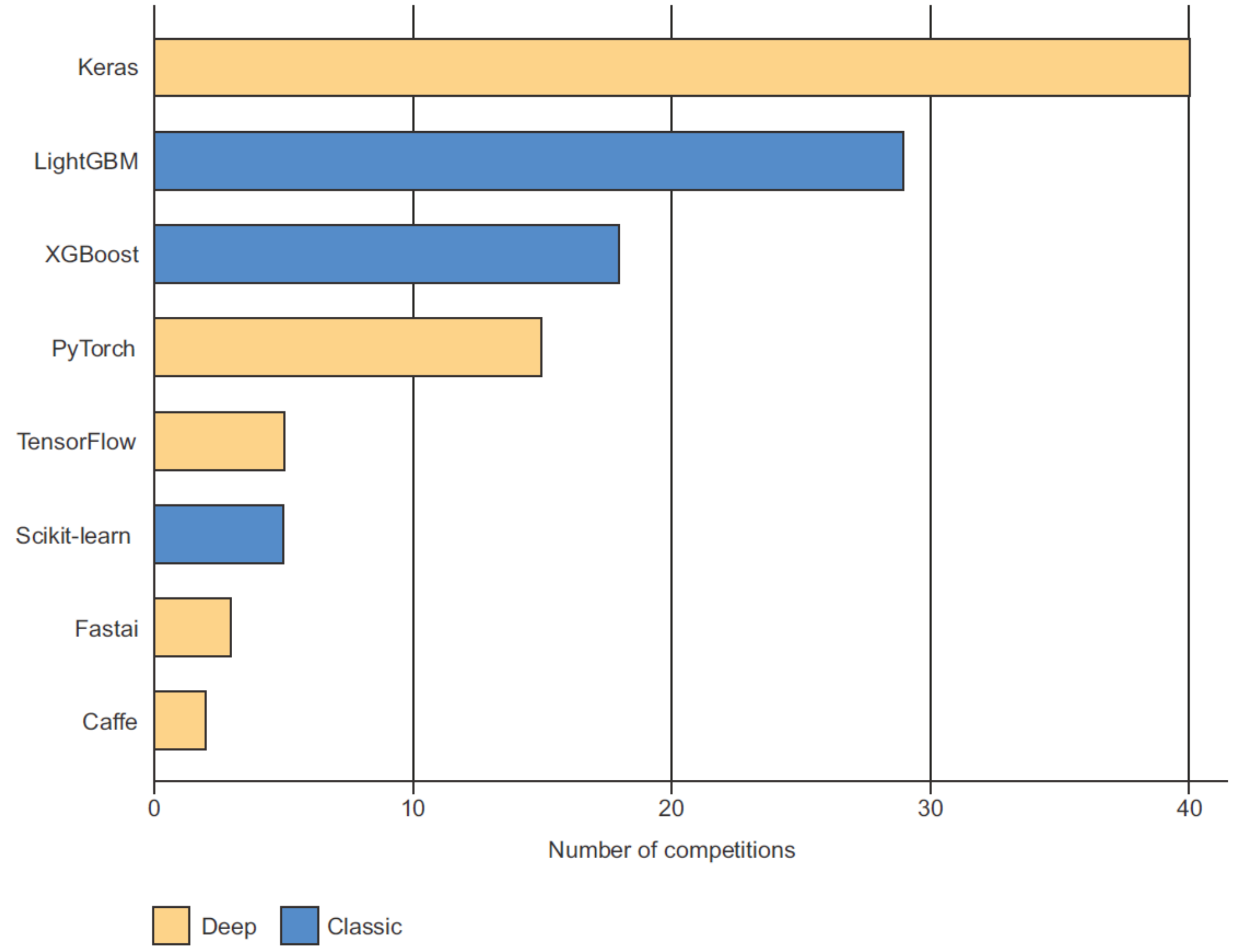


Deep Learning

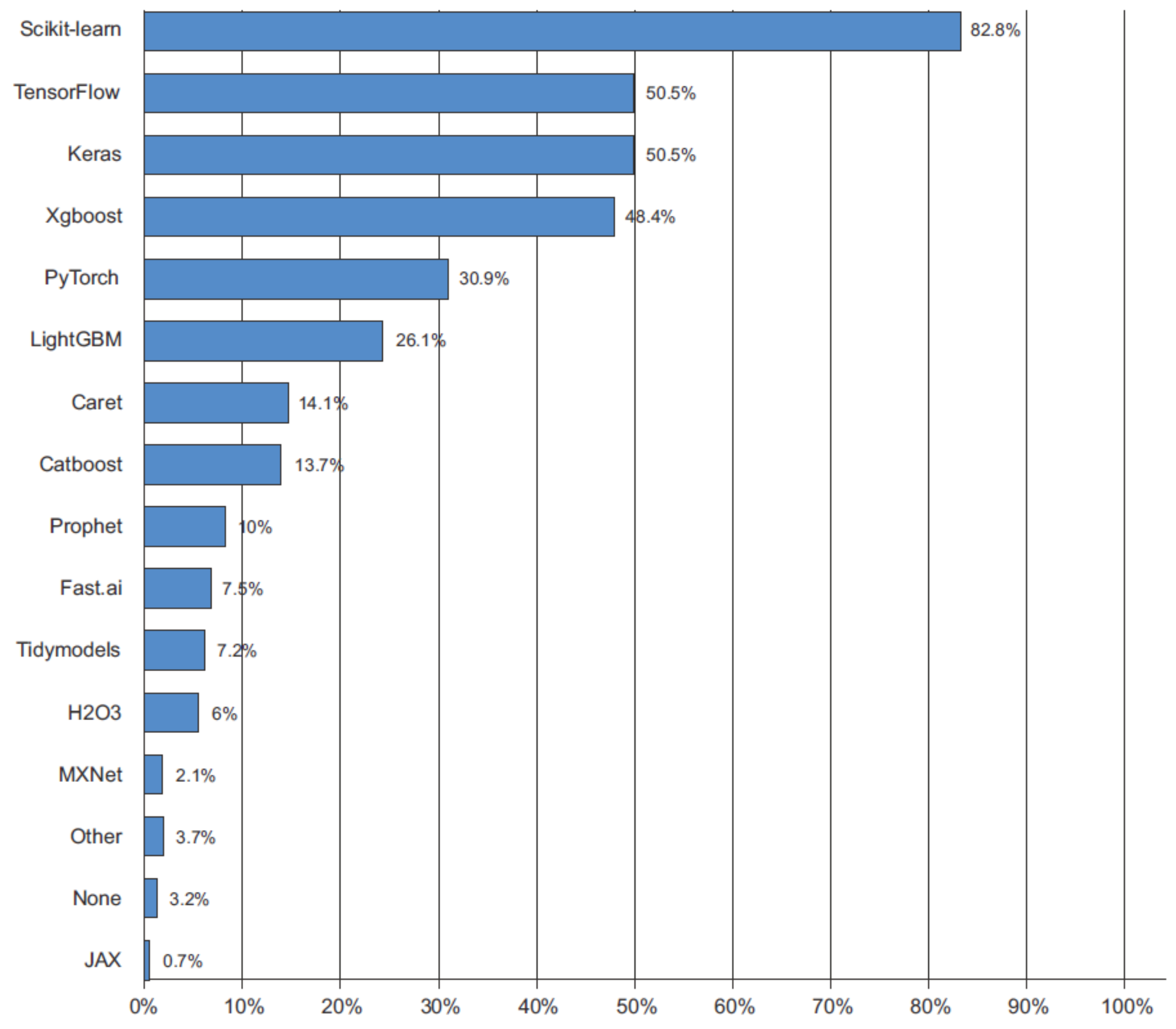
- 1989: Yann LeCun et al publish “Handwritten Digit Recognition with a Back-Propagation Network” at the Neural Information Processing Systems conference [the 3rd year for this long running conference]
- 2011: Dan Ciresan, from Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA), wins academic image-classification competitions with a convolutional neural network
- 2012: Alex Krizhenvsky, from University of Toronto (advisor: Geoff Hinton), wins the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)
  - 1.4 million images with 1,000 classes
  - “Classical approach”: 74.3% Accuracy@5 vs “Deep”: 83.6% Accuracy@5

Commonly used for “perceptual” (e.g. visual) and other unstructured data

Tool Survey: Top 5 Entries for Kaggle Contests 2017-2019



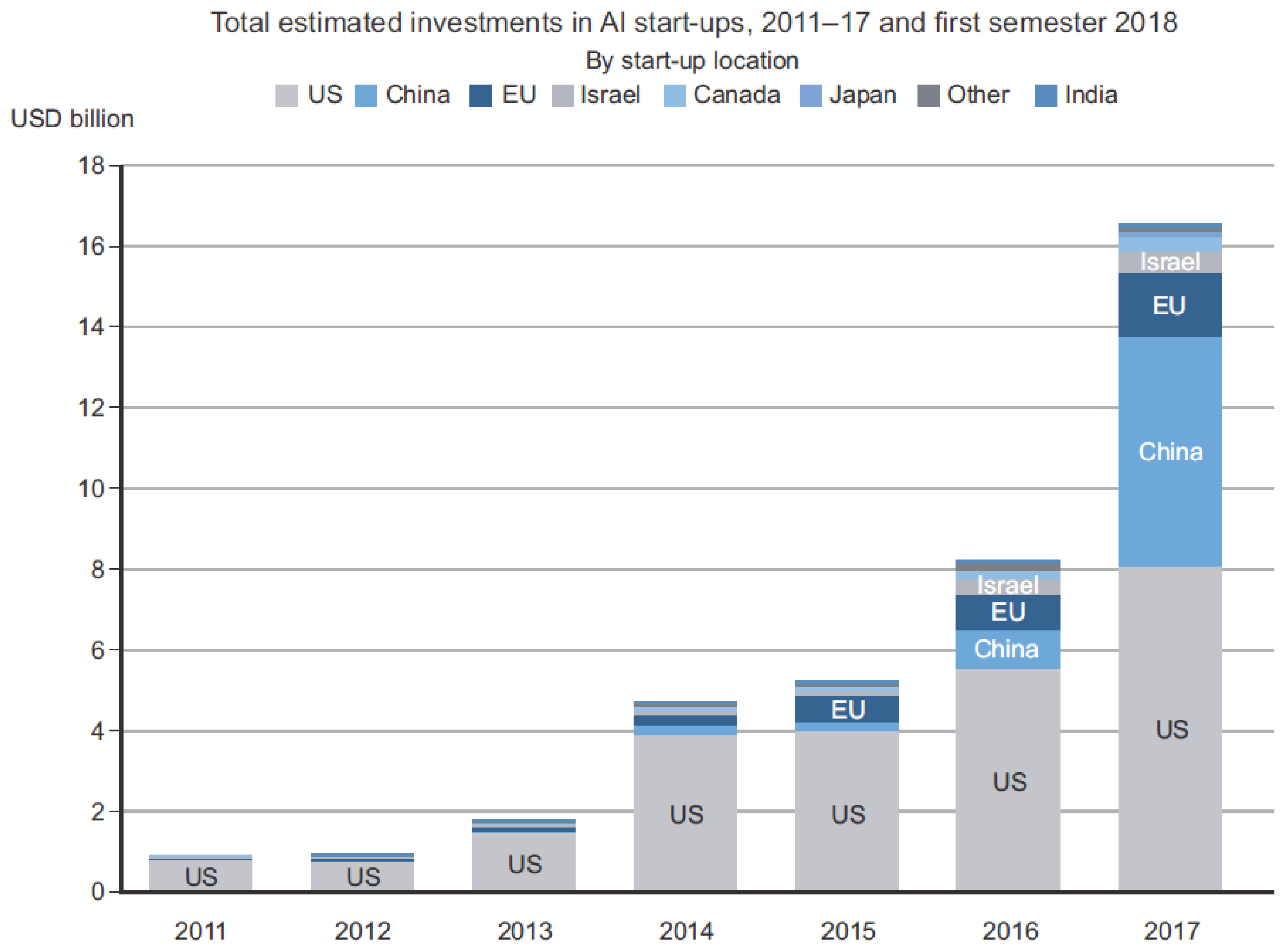
Tool Survey: All Contestants 2020



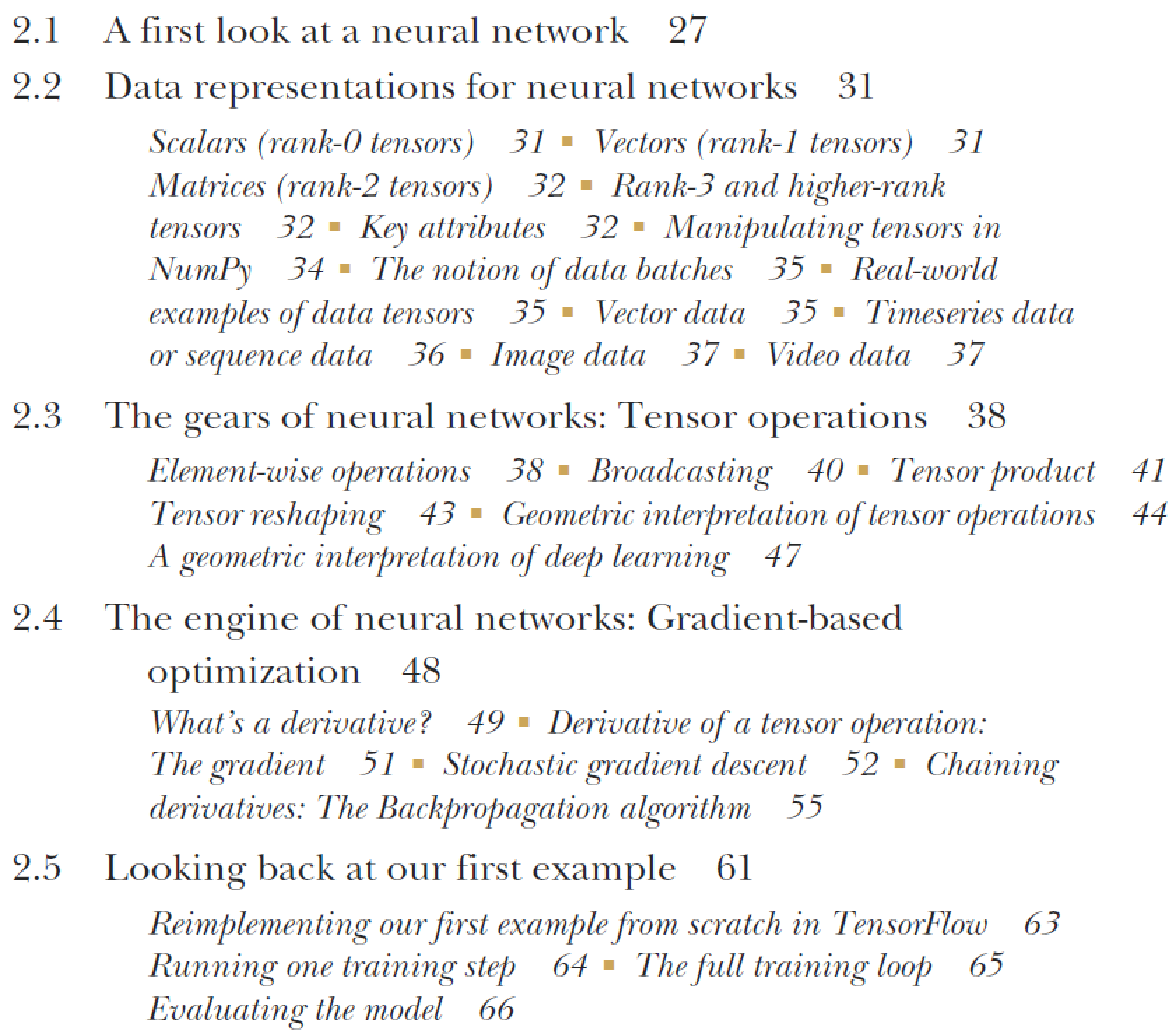
Why Deep Learning? Why Now?

- Hardware
  - Graphics Processing Units (GPUs): Nvidia [Common Unified Device Architecture (CUDA) API] and Advanced Micro Devices (AMD)
  - Google Tensor Processing Units (TPUs)
  - FLOPS: FLoating-point Operations Per Second; giga-, tera-, peta-, exa-
- Datasets and benchmarks
  - ImageNet Large Scale Visual Recognition Challenge: https://www.image-net.org/
  - The Pile: https://pile.eleuther.ai/
- Algorithmic advances
  - Better activation functions: e.g. ReLU
  - Better weight initialization; e.g. glorot uniform
  - Better optimization schemes; e.g. AdaM

Investments in AI Start-Ups



The Mathematical Building Blocks of Neural Networks



MNIST Sample Digits

- In machine learning, a category in a classification problem is called a class
- Data points are called samples
- The class associated with a specific sample is called a label

MNIST Data

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

MNIST Model Architecture

Initialization happens when you the layer is instantiated, as shown here … [use model.get_weights() to see initial weights]

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

MNIST Model “Compilation”

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

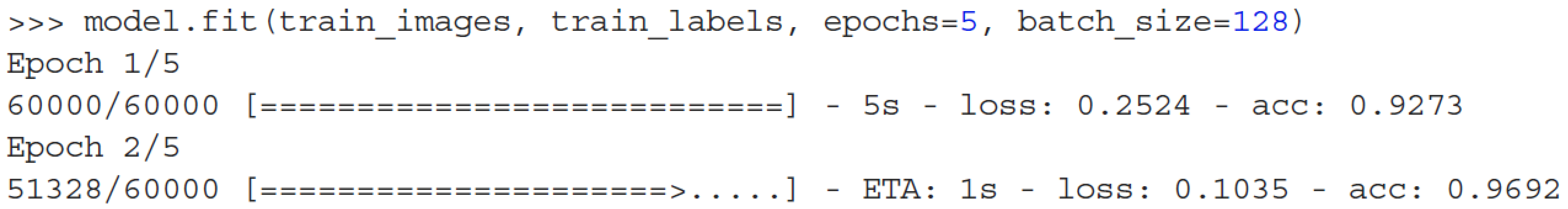
MNIST Data Preprocessing

- Images are “flattened”
- Min-max normalization is applied

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

MNIST Model Training

- Please use validation to sanity check performance
- Never more than an hour or so without a sanity check on hold-out data



```
>>> model.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [==========================] - 5s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=====================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

“We quickly reach an accuracy of 0.989 (98.9%) on the training data”

MNIST Model Predictions

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
>>> predictions[0].argmax()
7
>>> predictions[0][7]
0.99999106

>>> test_labels[0]
7
```

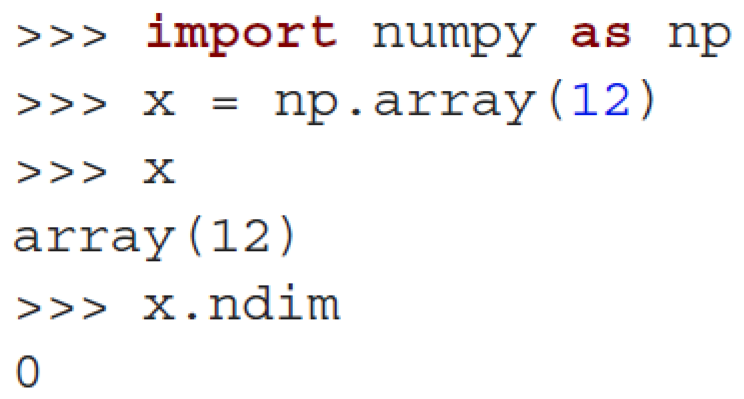
MNIST Model Testing

Nota bene: validation data should be used for model selection

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

“This gap between training accuracy and test accuracy is an example of *overfitting*”
… not so fast …
If validation accuracy continues to improve, we’re going to keep training.
It’s when validation accuracy deteriorates while training accuracy continues to improve that we need to stop.

Scalars: Rank-0 Tensors



```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

Vectors: Rank-1 Tensors

```
>>> x = np.array([12, 3, 6, 14, 7])
>>> x
array([12, 3, 6, 14, 7])
>>> x.ndim
1
```

Matrices: Rank-2 Tensors

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim
2
```

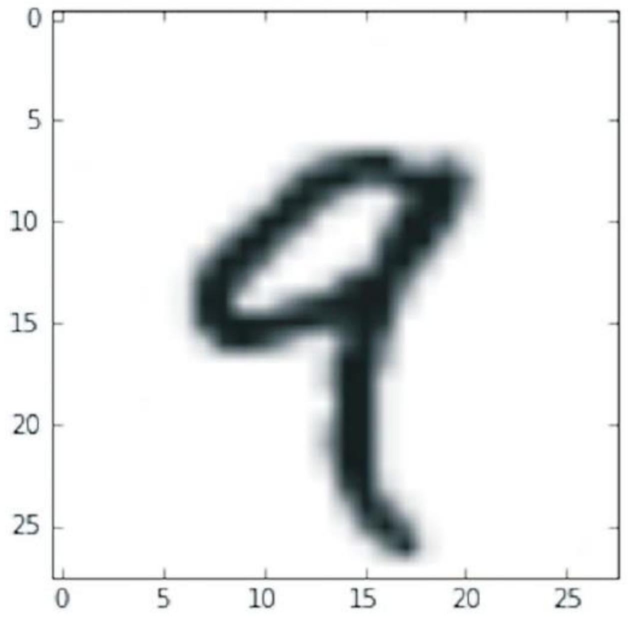
Rank-3 and Higher-Rank Tensors

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Numpy Attributes

- ndim: number of dimensions/axes (aka “rank”)
- shape: index count along each axis
- dtype: the data type; e.g. uint8, int32, float32, float16, float64

Plotting an MNIST Data



```
import matplotlib.pyplot as plt
digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

Slicing Tensors

```
>>> my_slice = train_images[10:100]
>>> my_slice.shape

>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Equivalent to the previous example

Also equivalent to the previous example

```
my_slice = train_images[:, 14:, 14:]

my_slice = train_images[:, 7:-7, 7:-7]
```

Data Batches

Slicing along the “samples” dimension …

```
batch = train_images[:128]

batch = train_images[128:256]

n = 3
batch = train_images[128 * n:128 * (n + 1)]
```

Real-world Examples of Data Tensors

- Vector data: Rank-2 tensors of shape (samples, features), where each sample is a vector of numerical attributes (“features”)
- Timeseries data or sequence data: Rank-3 tensors of shape (samples, timesteps, features), where each sample is a sequence (of length timesteps) of feature vectors
- Images: Rank-4 tensors of shape (samples, height, width, channels), where each sample is a 2D grid of pixels, and each pixel is represented by a vector of values (“channels”)
- Video: Rank-5 tensors of shape (samples, frames, height, width, channels), where each sample is a sequence (of length frames) of images

Vector Data Examples

- An actuarial dataset of people, where we consider each person’s age, gender, and income: (100000, 3)
- A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words): (500, 20000)

Timeseries and Sequence Data Examples

- A dataset of stock prices: (250, 390, 3)
  - only 1 stock?
  - 250 days worth of data
  - 390 trading minutes in the day
  - current price; highest price in the previous minute; lowest price in the previous minute
- A dataset of tweets, where we encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters: (1000000, 280, 128)

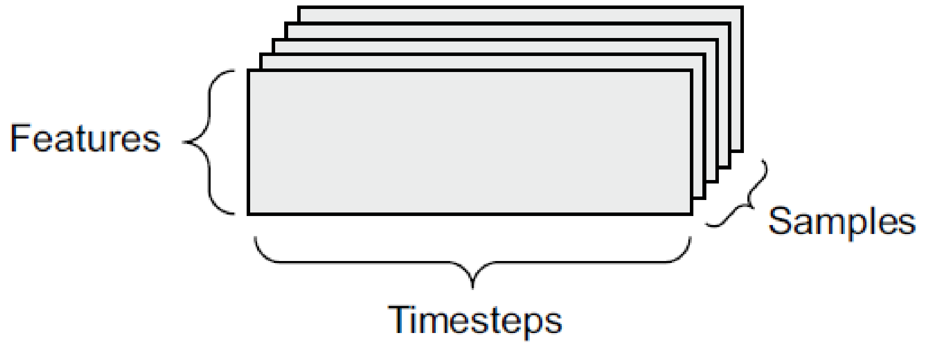


Image Data

- Channels first

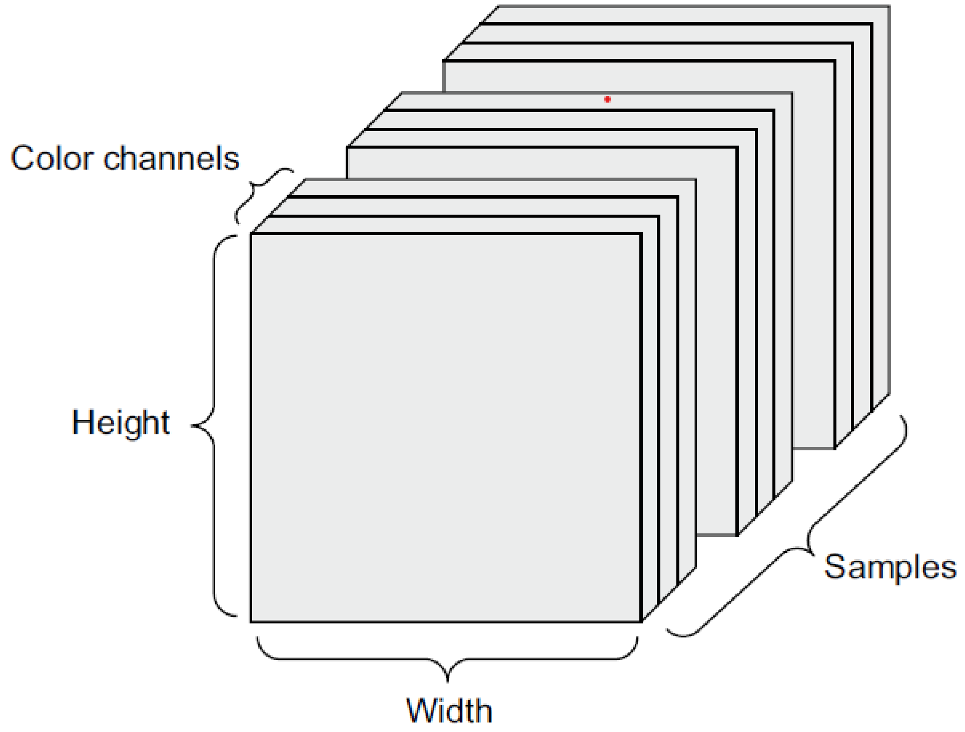
  (samples, color_depth, height, width)

  pytorch

- Channels last

  (samples, height, width, color_depth)

  tensorflow



Video Data

- (samples, frames, height, width, color_depth)
- 60-second video clip
- 4-frames per second
- 144x256 frames
- color: each picture element (pixel) has 3 color values
  { red, green, blue }
- (4, 240, 144, 256, 3): 106,168,320 float32 values; 4 bytes each
  405 megabytes

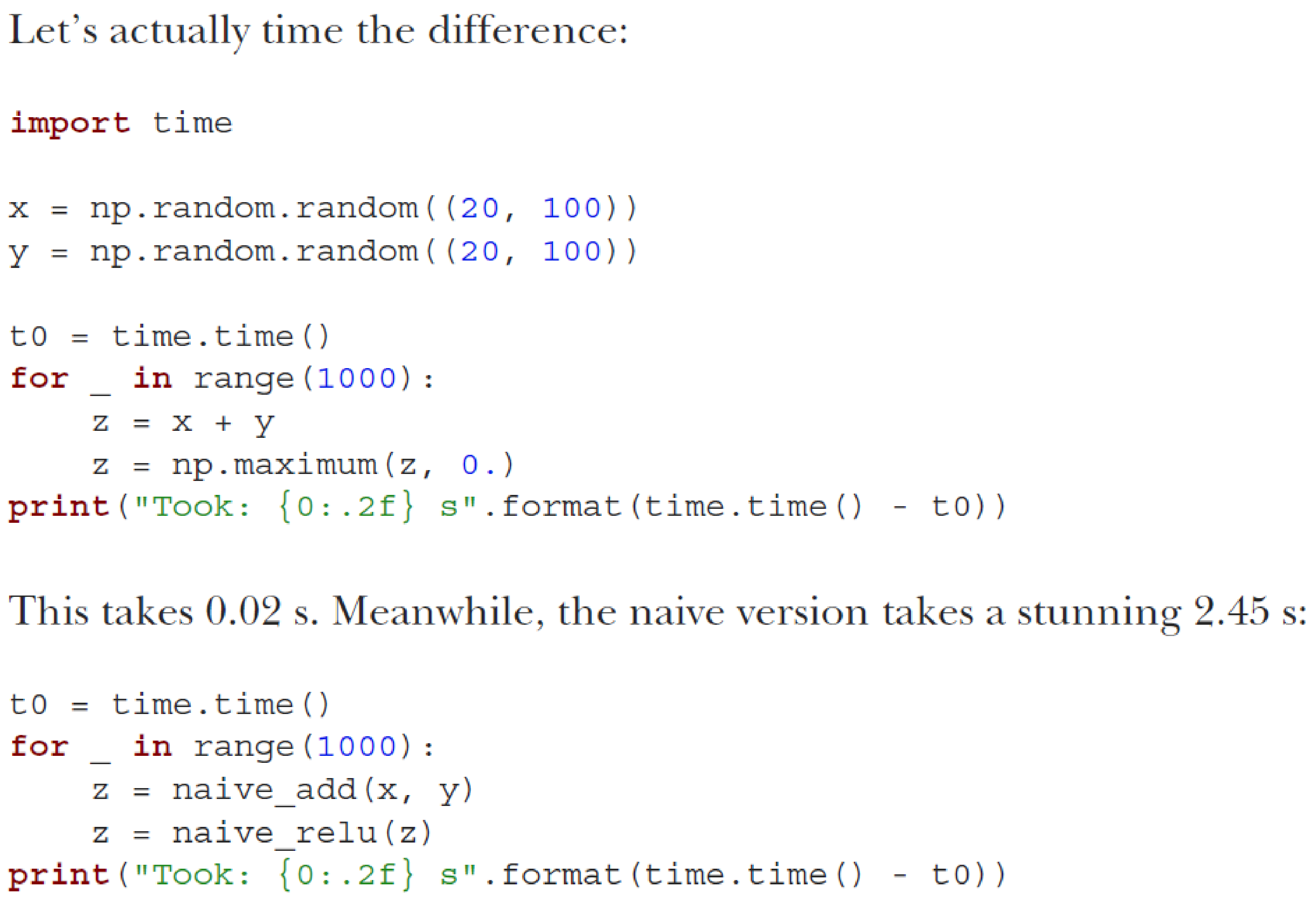
Tensor Operations

```
keras.layers.Dense(512, activation="relu")

output = relu(dot(input, W) + b)

def naive_relu(x):
    assert len(x.shape) == 2
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

Tensor Add

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

Naïve Implementation is Wildly Slower



Let’s actually time the difference:

```
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.)
print("Took: {0:.2f} s".format(time.time() - t0))
```

This takes 0.02 s. Meanwhile, the naive version takes a stunning 2.45 s:

```
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {0:.2f} s".format(time.time() - t0))
```

Broadcasting

```
import numpy as np
X = np.random.random((32, 10))
y = np.random.random((10,))
y = np.expand_dims(y, axis=0)
Y = np.concatenate([y] * 32, axis=0)

def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

# What does this do, and what is the output size?

```
import numpy as np
x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

Vector Dot Product

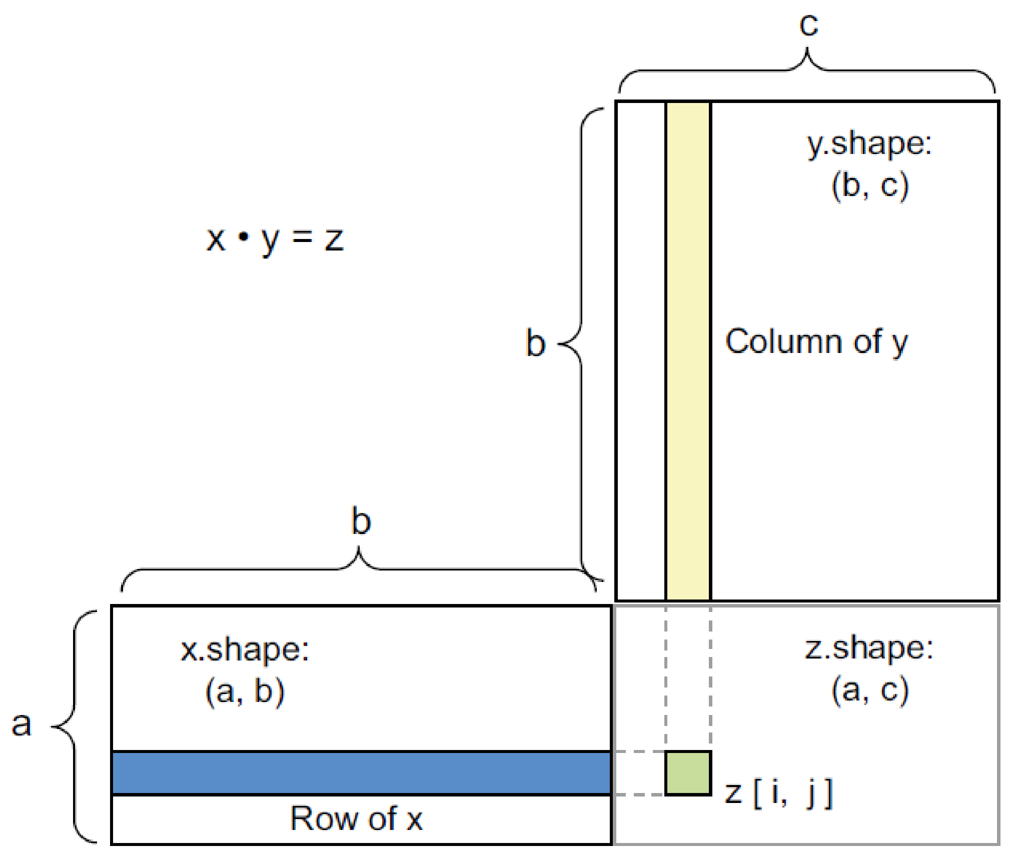
```
x = np.random.random((32,))
y = np.random.random((32,))
z = np.dot(x, y)
z = x • y
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Matrix-Vector Dot Product

```
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z

def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

Matrix-Matrix Dot Product

```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```



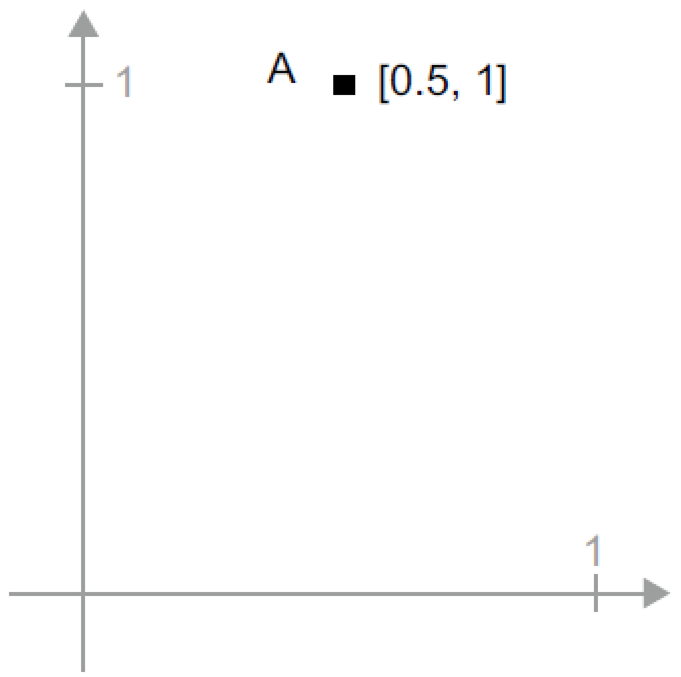
Tensor Reshaping

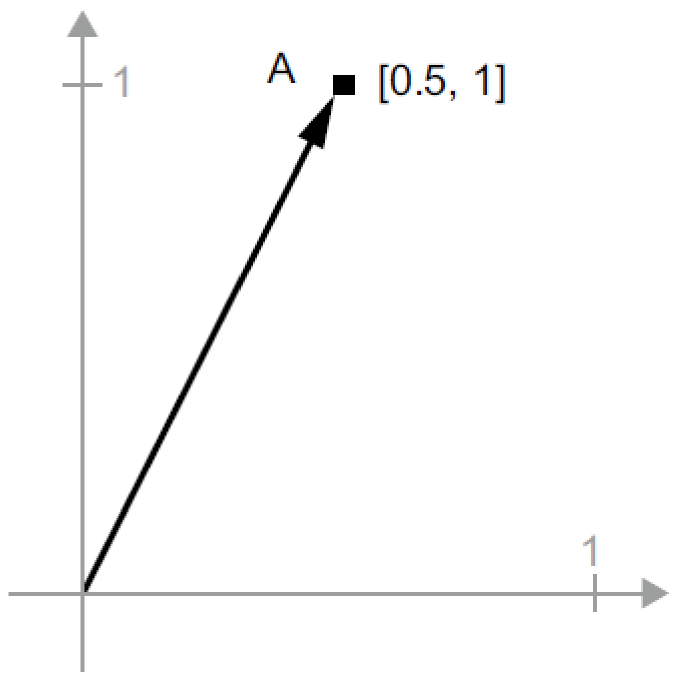
```
>>> x = np.array([[0., 1.],
[2., 3.],
[4., 5.]])
>>> x.shape
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```

```
>>> x = x.reshape((2, 3))
>>> x
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])

>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> x.shape
(20, 300)
```

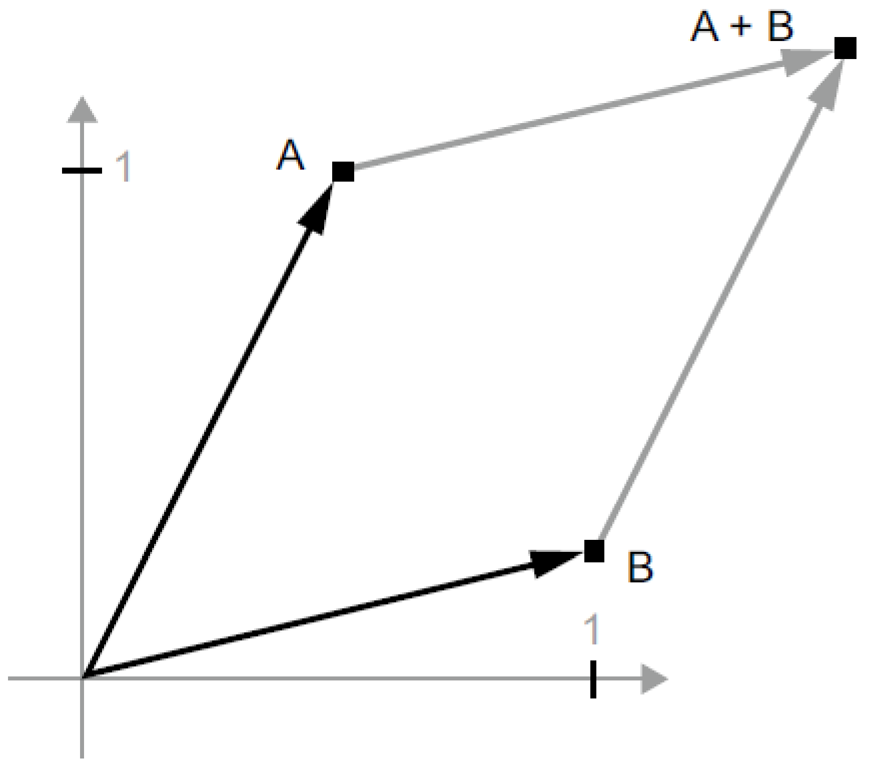
```
train_images = train_images.reshape((60000, 28 * 28))
```

Example Vector

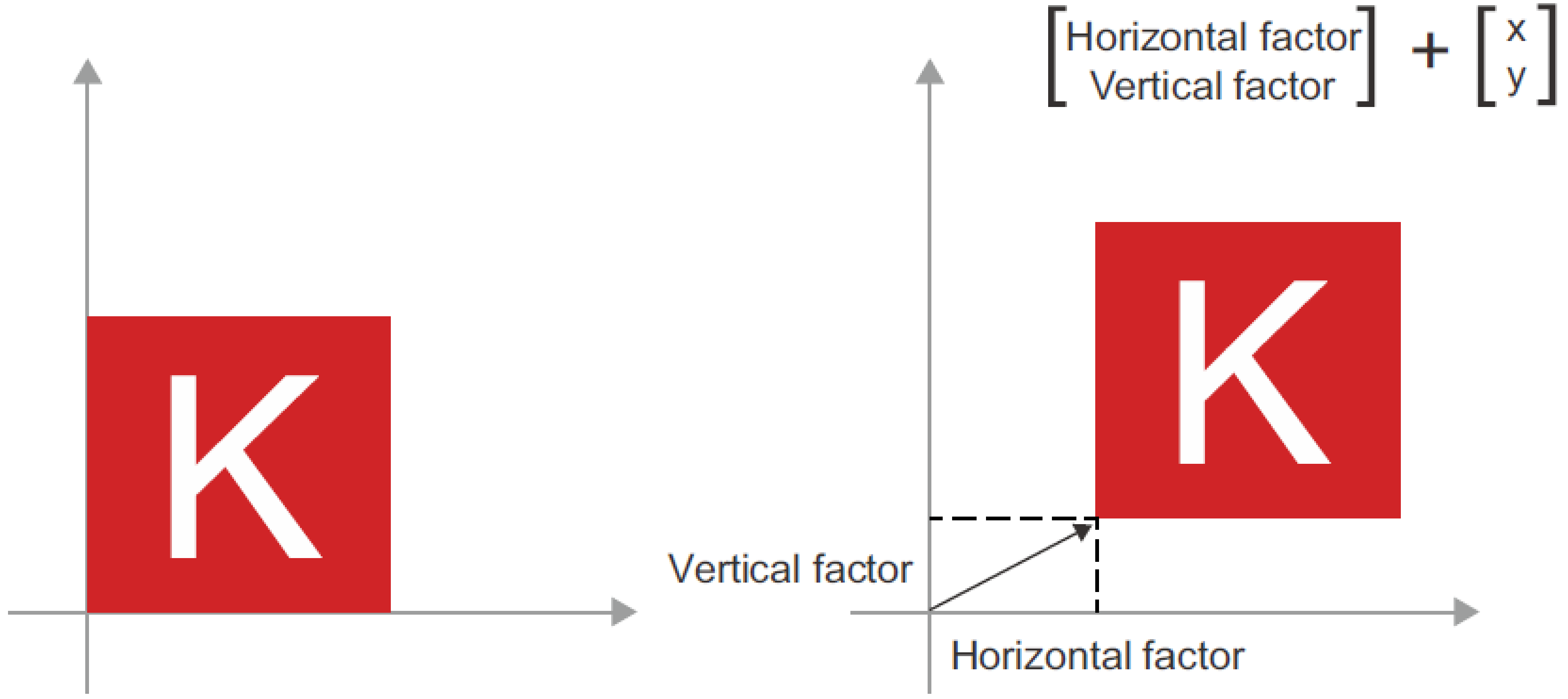




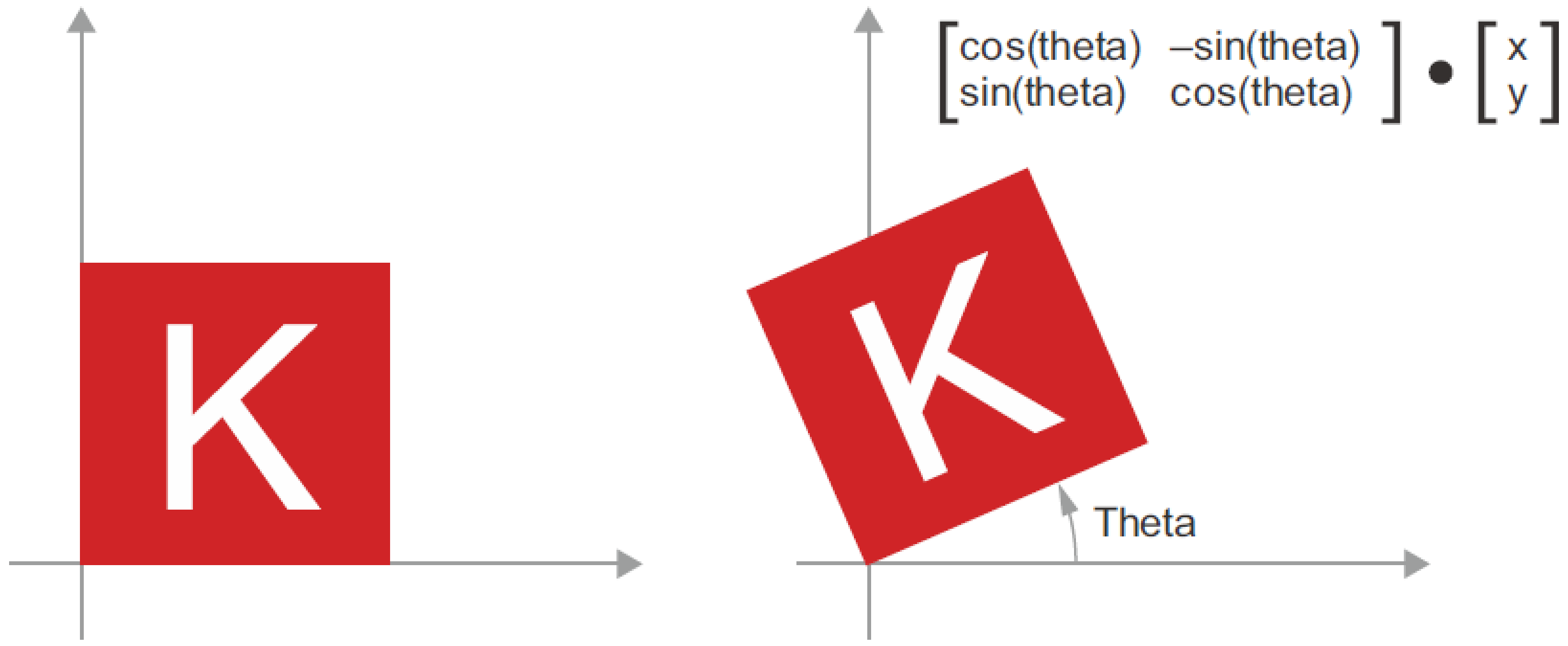
Geometric Interpretation of a Sum of Vectors



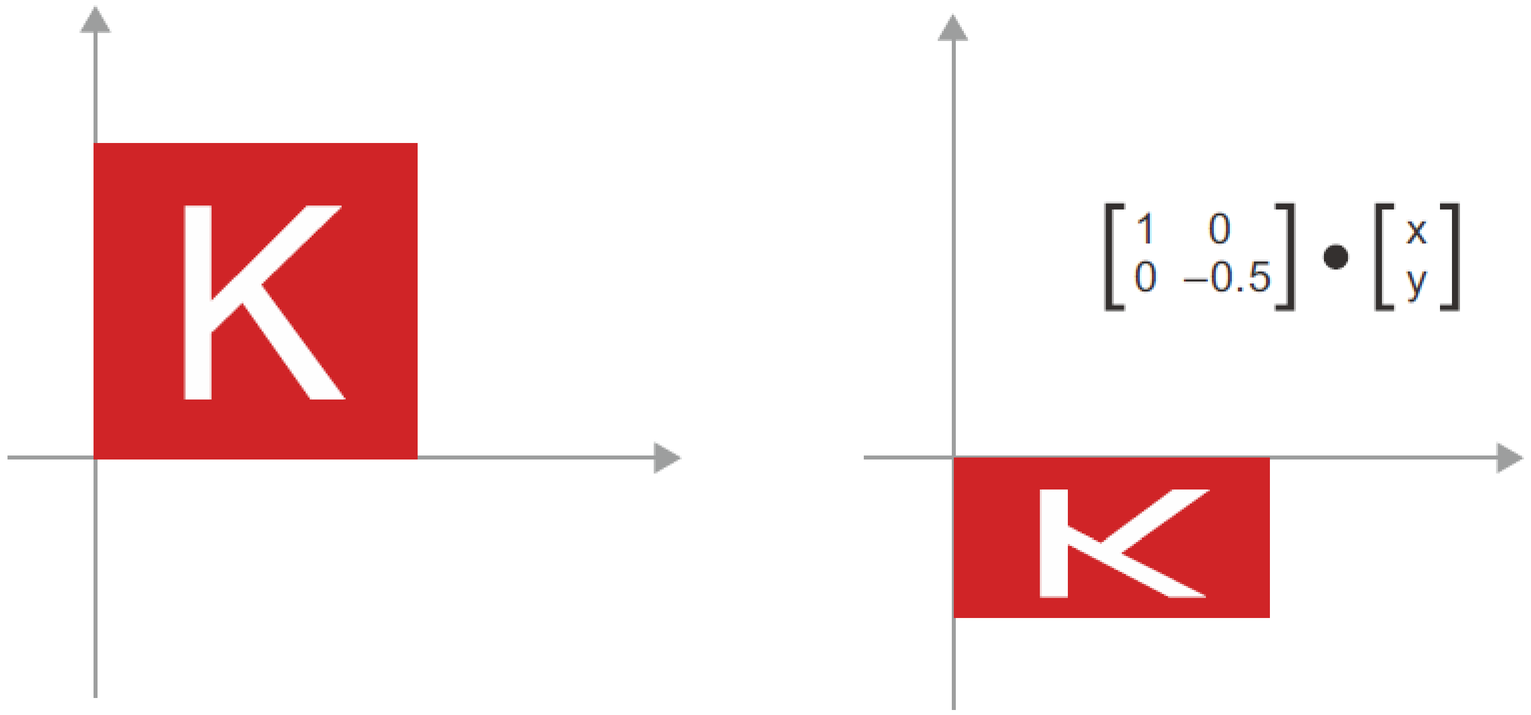
Translation as Vector Addition


K
$\begin{bmatrix} \text{Horizontal factor} \\ \text{Vertical factor} \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$
K
Vertical factor
Horizontal factor

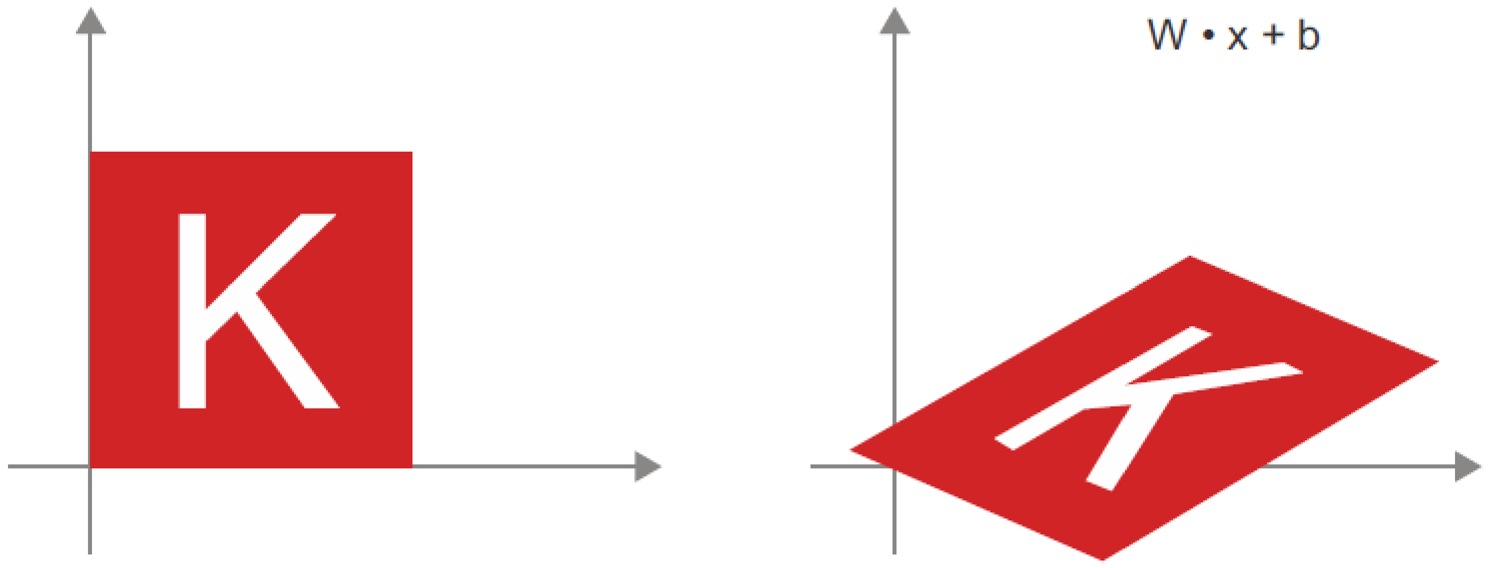
2-D Rotation as a Dot Product



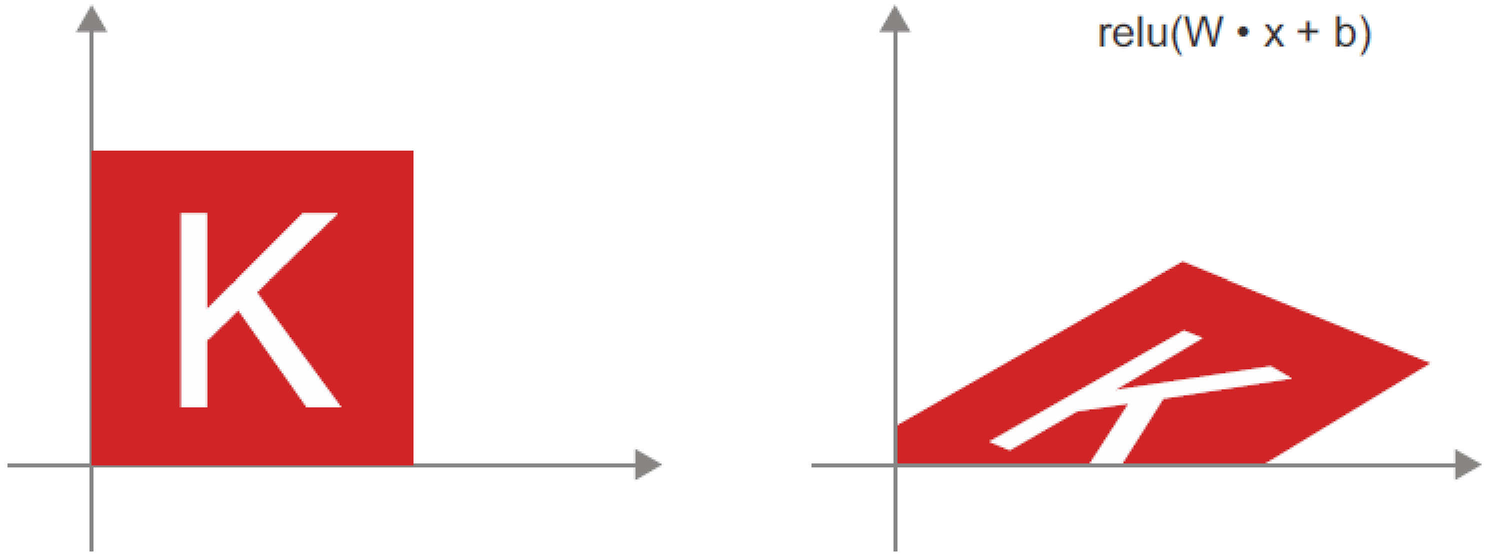
2-D Scaling as a Dot Product


K
$\begin{bmatrix} 1 & 0 \\ 0 & -0.5 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix}$
K

Affine Transform in the Plane

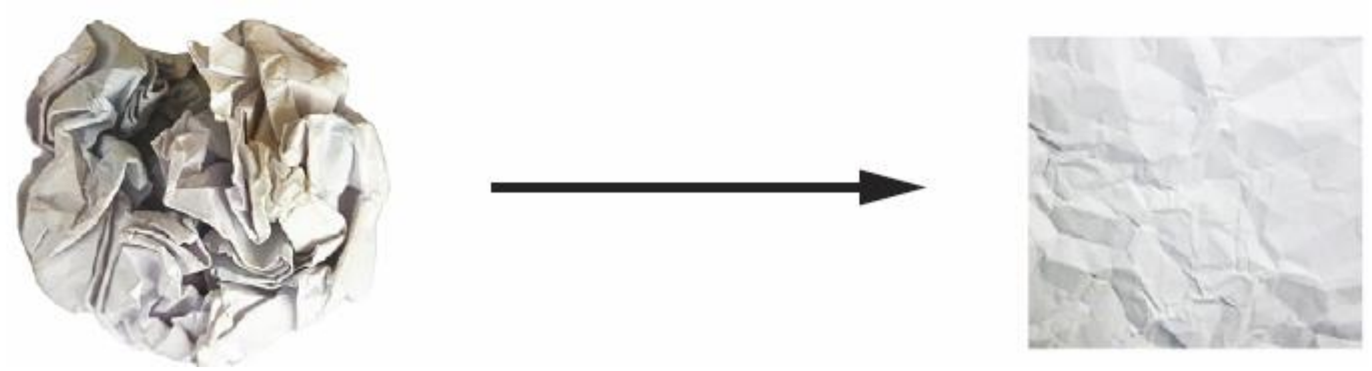


Affine Transform Followed by ReLU Activation



Uncrumpling a Complicated Manifold of Data

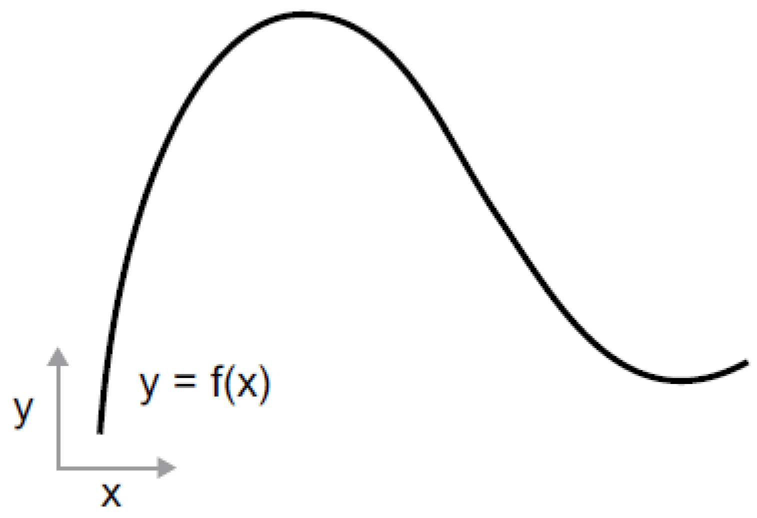
“Uncrumpling paper balls is what machine learning is about” ☺



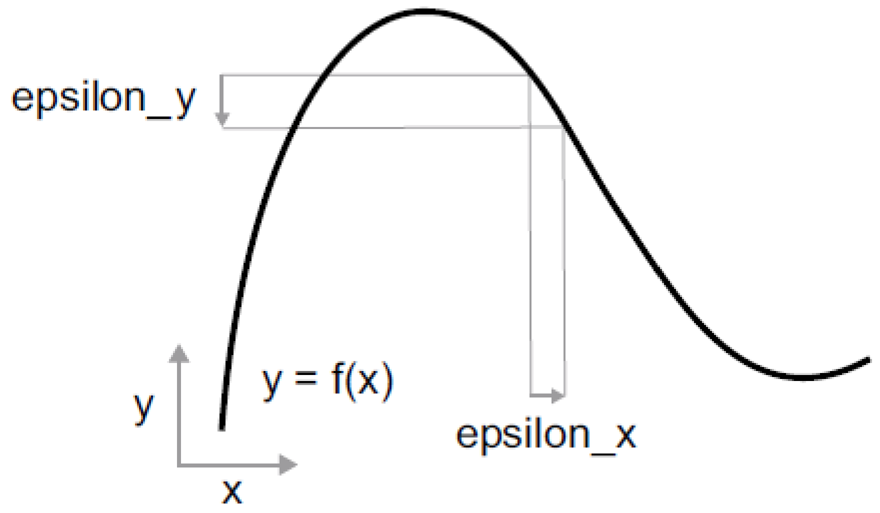
Training Loop

- Draw a batch of training samples, x, and corresponding targets, y_true
- Run the model on x (a step called the forward pass) to obtain predictions, y_pred
- Compute the loss of the model on the batch, a measure of the mismatch between y_pred and y_true
- Update all weights of the model in a way that slightly reduces the loss on this batch

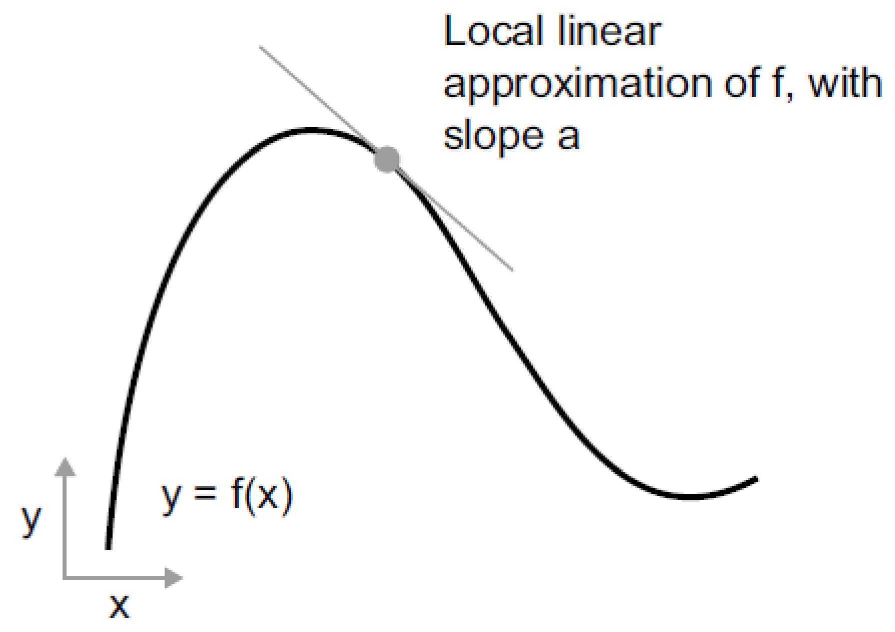
A Continuous Smooth Function



Small Change in x Results in Small Change in y



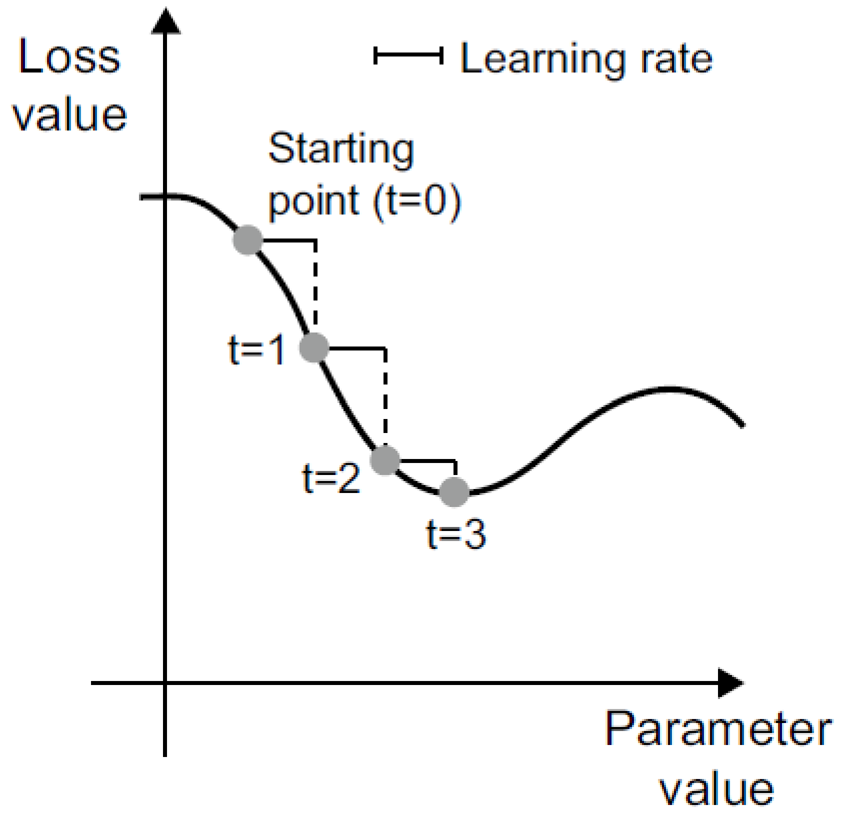
Derivative of f(x) with respect to x



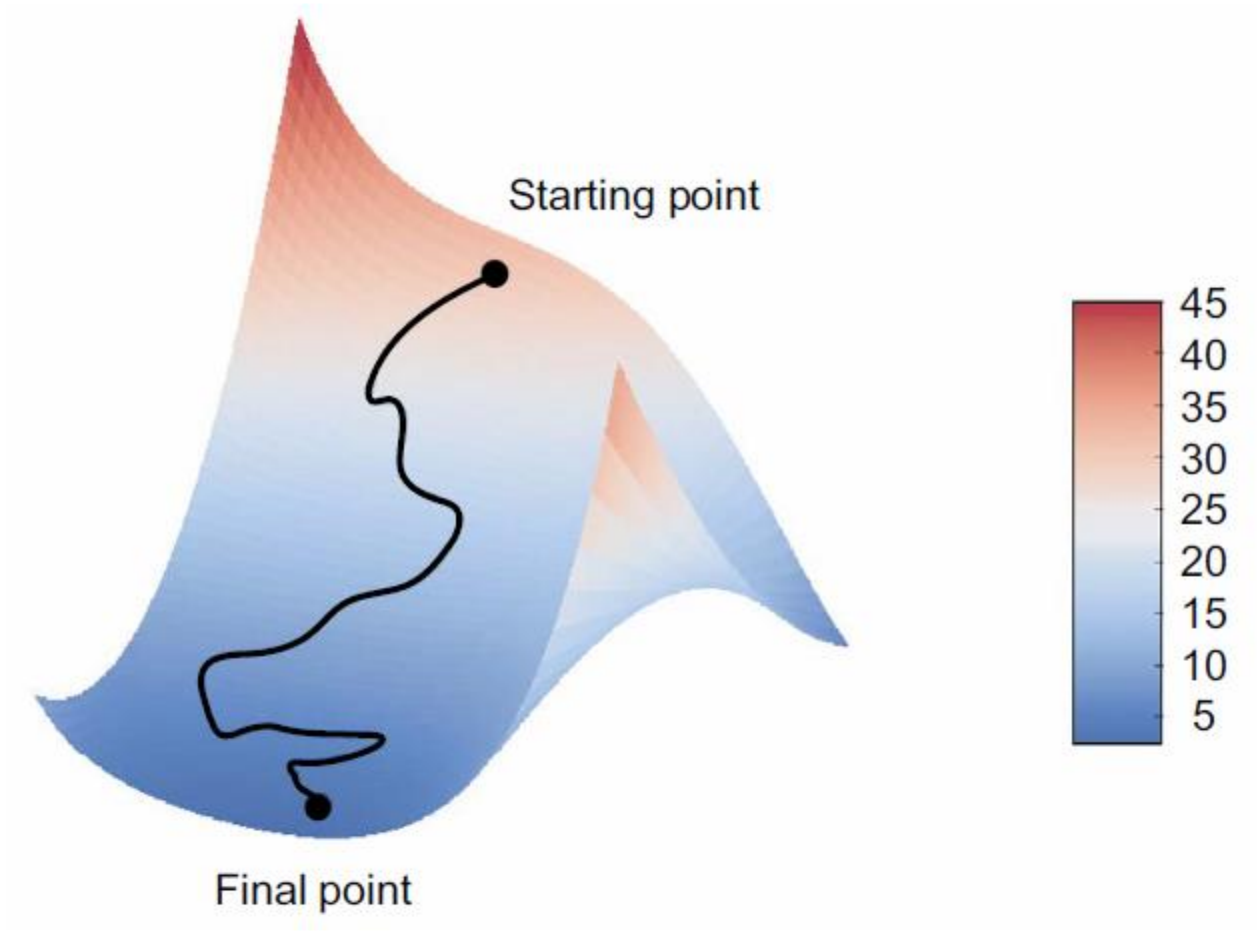
Training Loop Revisited

- Draw a batch of training samples, x, and corresponding targets, y_true
- Run the model on x to obtain predictions, y_pred (this is called the forward pass)
- Compute the loss of the model on the batch, a measure of the mismatch between y_pred and y_true
- Compute the gradient of the loss with regard to the model's parameters (this is called the backward pass)
- Move the parameters a little in the opposite direction from the gradient: for example, W -= learning_rate * gradient, thus reducing the loss on the batch a bit

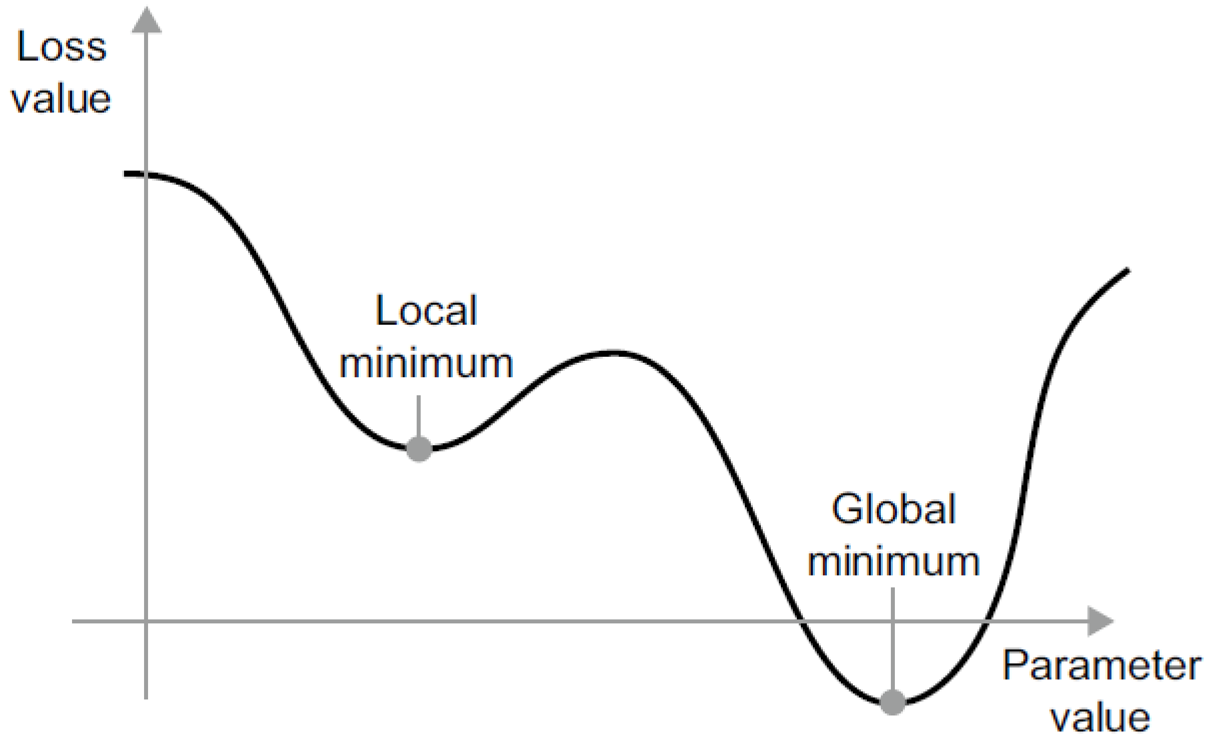
SGD on a 1-D Loss Curve



Gradient Descent on a 2-D Loss Surface



Local Minimum vs Global Minimum



Momentum Example

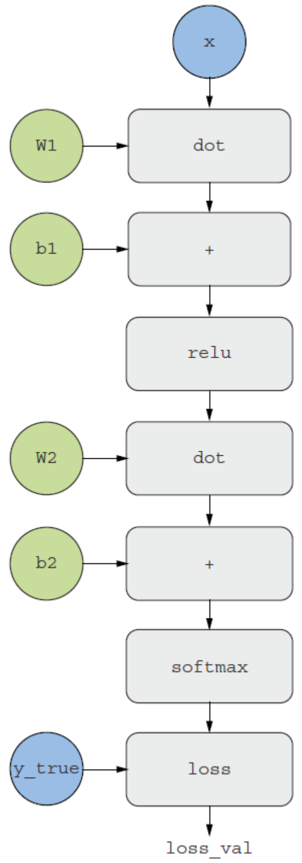
```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

Chain Rule

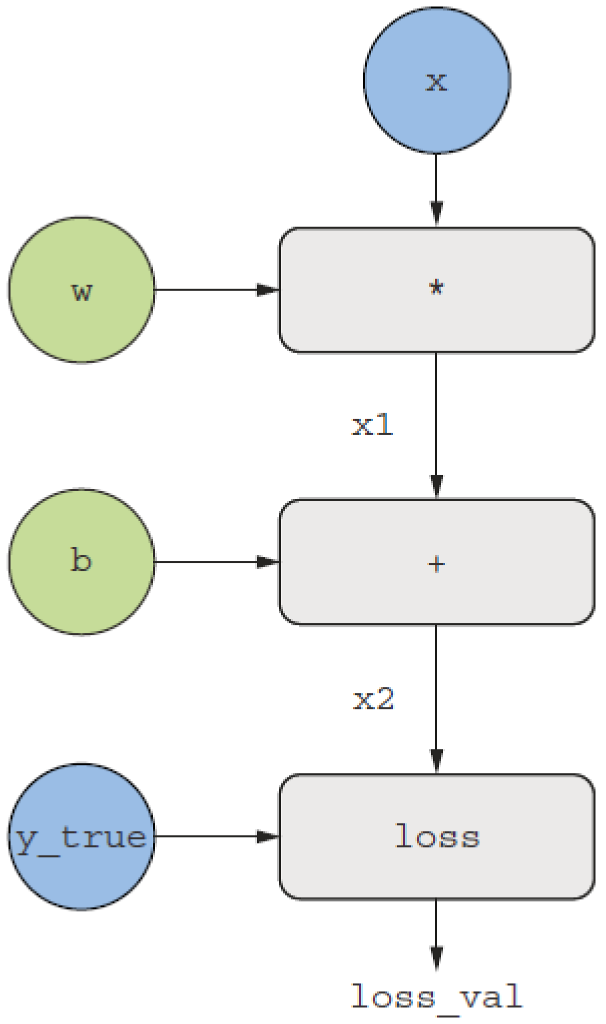
```
loss_value = loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))
```

```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y = f(x3)
    return y
```

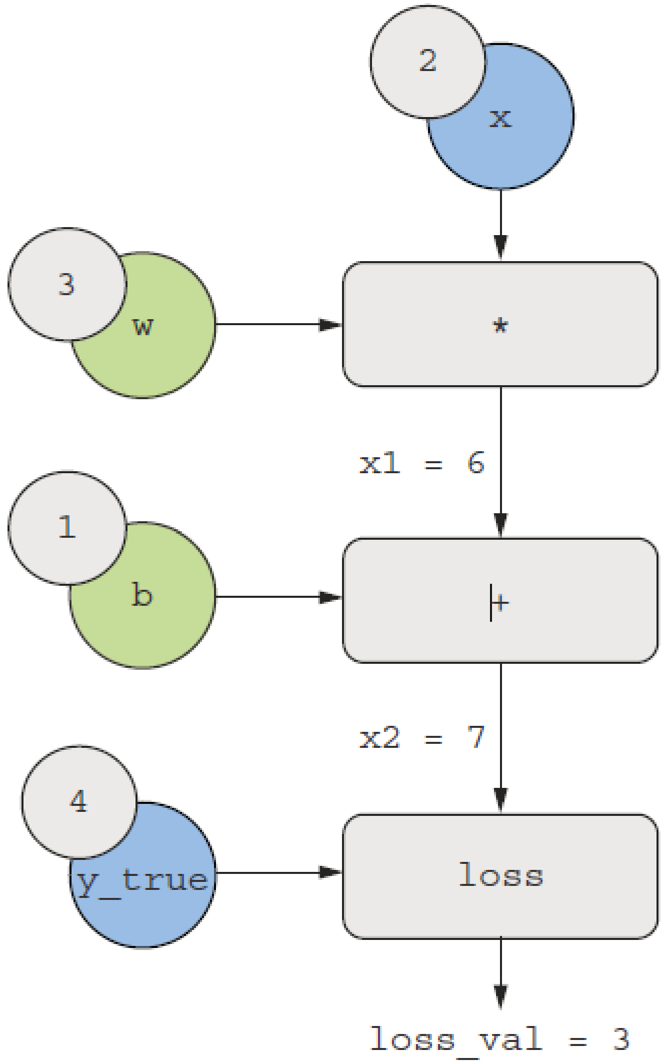
```
grad(y, x) == (grad(y, x3) * grad(x3, x2) * grad(x2, x1) * grad(x1, x))
```



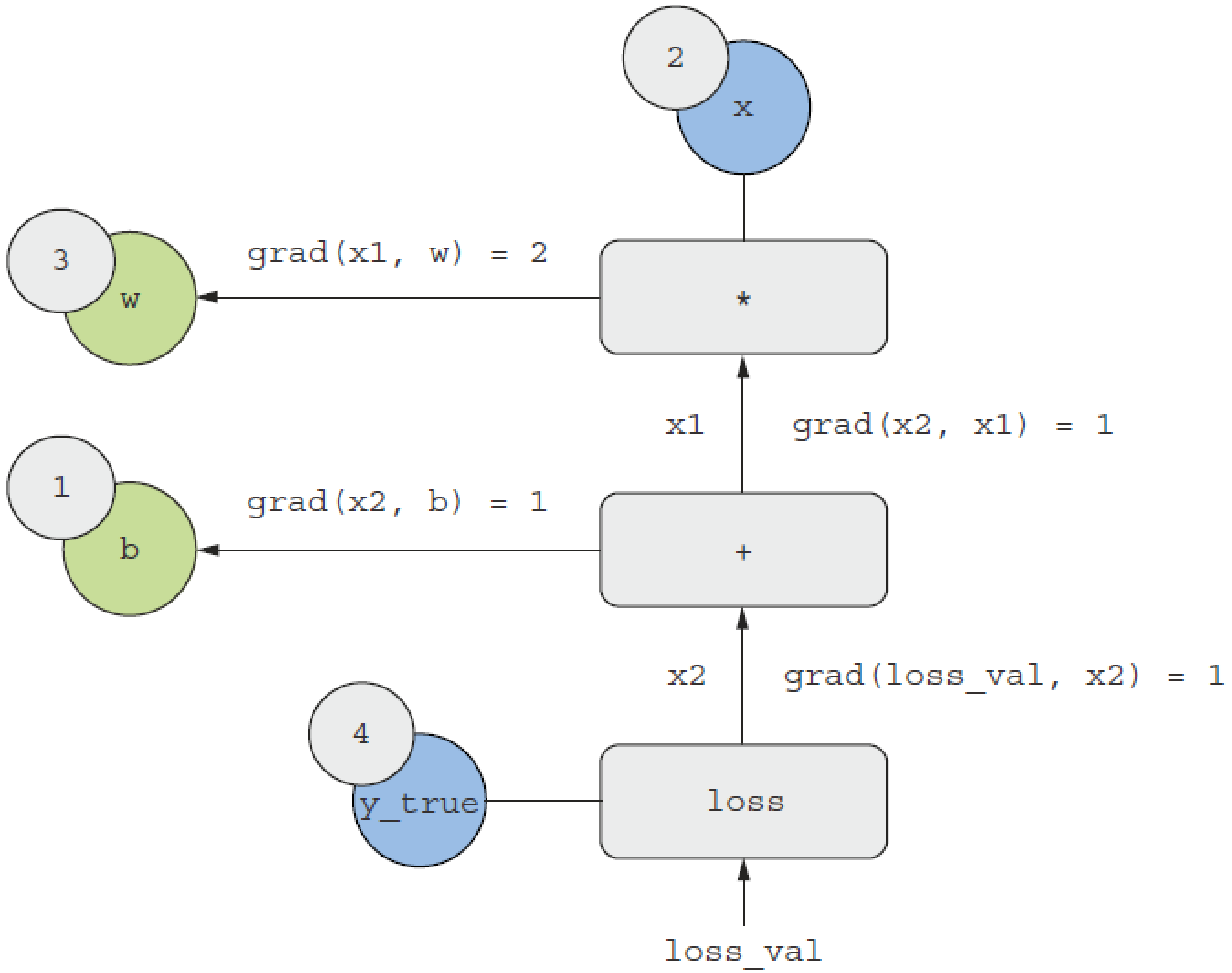
Example Computation Graph



Forward: Regression with Absolute Error



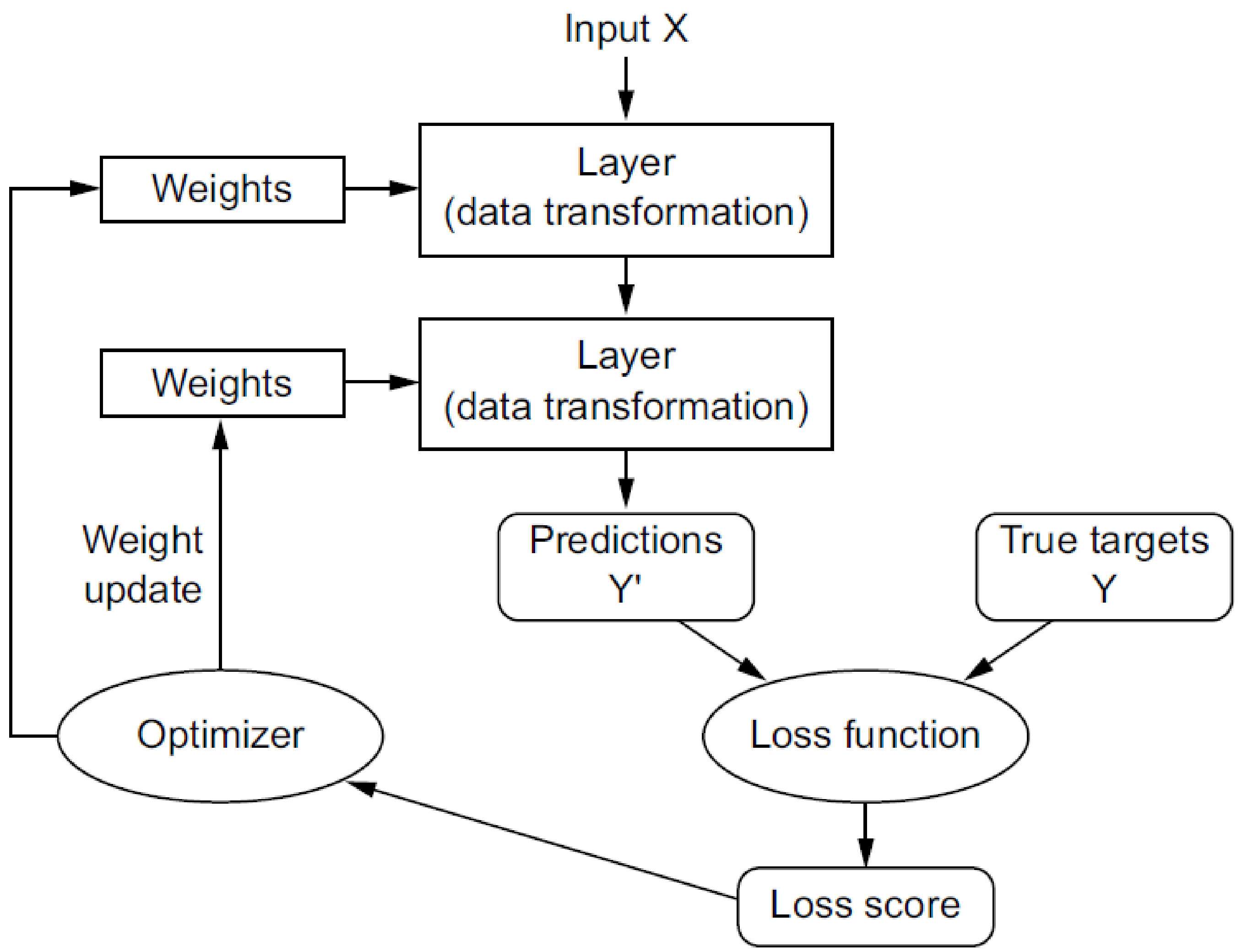
Backward: Regression with Absolute Error



Gradient Tape Examples

```
import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)

x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)

W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

Training Loop Review: Graph



Training Loop Review: Code

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

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

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

model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Implementing Keras: Dense Class

```
import tensorflow as tf
class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        self.activation = activation
        w_shape = (input_size, output_size)
        w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
        self.W = tf.Variable(w_initial_value)
        b_shape = (output_size,
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)
    def __call__(self, inputs)::
        return self.activation(tf.matmul(inputs, self.W) + self.b)
    @property
    def weights(self):
        return [self.W, self.b]
```

Implementing Keras: Sequential Class

```
class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers
    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
        x = layer(x)
        return x
    @property
    def weights(self):
        weights = []
        for layer in self.layers:
            weights += layer.weights
        return weights
```

Implementing Keras: Batch Generator

```
import math
class BatchGenerator:
    def __init__(self, images, labels, batch_size=128):
        assert len(images) == len(labels)
        self.index = 0
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.num_batches = math.ceil(len(images) / batch_size)
    def next(self):
        images = self.images[self.index : self.index + self.batch_size]
        labels = self.labels[self.index : self.index + self.batch_size]
        self.index += self.batch_size
        return images, labels
```

Implementing Keras: One Training Step

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        per_sample_losses = tf.keras.losses.sparse_categorical_crossentropy(
            labels_batch, predictions)
        average_loss = tf.reduce_mean(per_sample_losses)
    gradients = tape.gradient(average_loss, model.weights)
    update_weights(gradients, model.weights)
    return average_loss
```

Implementing Keras: Optimizer

```
learning_rate = 1e-3
def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign_sub(g * learning_rate)
```

Implementing Keras: Full Training Loop

```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")
        batch_generator = BatchGenerator(images, labels)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            if batch_counter % 100 == 0:
                print(f"loss at batch {batch_counter}: {loss:.2f}")
```

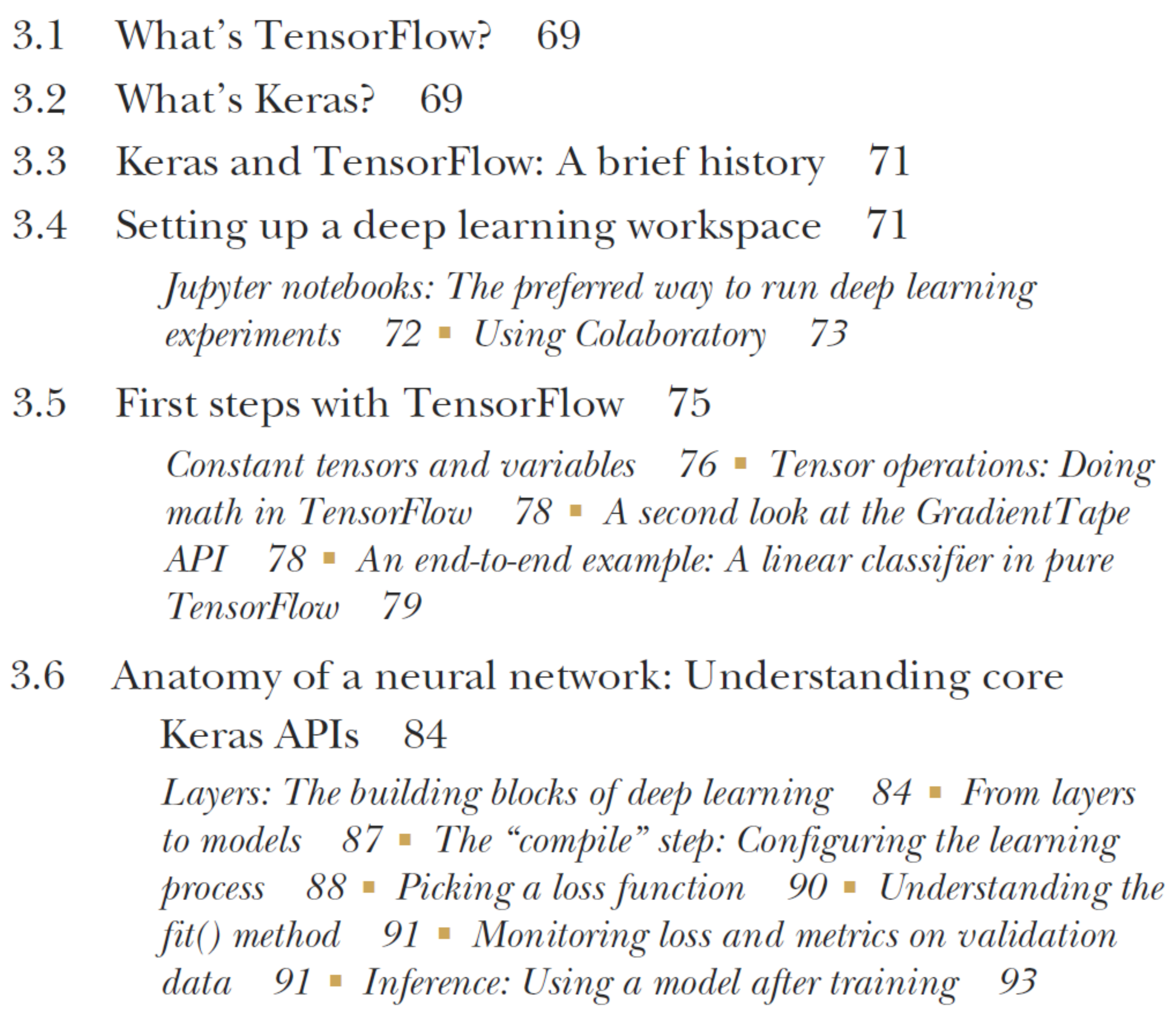
Implementing Keras: “Go” Time!

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

fit(model, train_images, train_labels, epochs=10, batch_size=128)

predictions = model(test_images)
predictions = predictions.numpy()
predicted_labels = np.argmax(predictions, axis=1)
matches = predicted_labels == test_labels
print(f"accuracy: {matches.mean():.2f}")
```

Summary Concepts

- Tensors: ndim, shape, dtype
- Tensor operations: addition, tensor product, element-wise multiplication
- Model is parameterized by weights and biases
- Learning: updating weights to minimize a loss function
- Mini-batch stochastic gradient descent
- Chain rule used to find gradient of loss with respect to a parameters
- Loss used to evaluate a prediction
- Optimizer specifies how to update weights

Introduction to Keras and Tensorflow

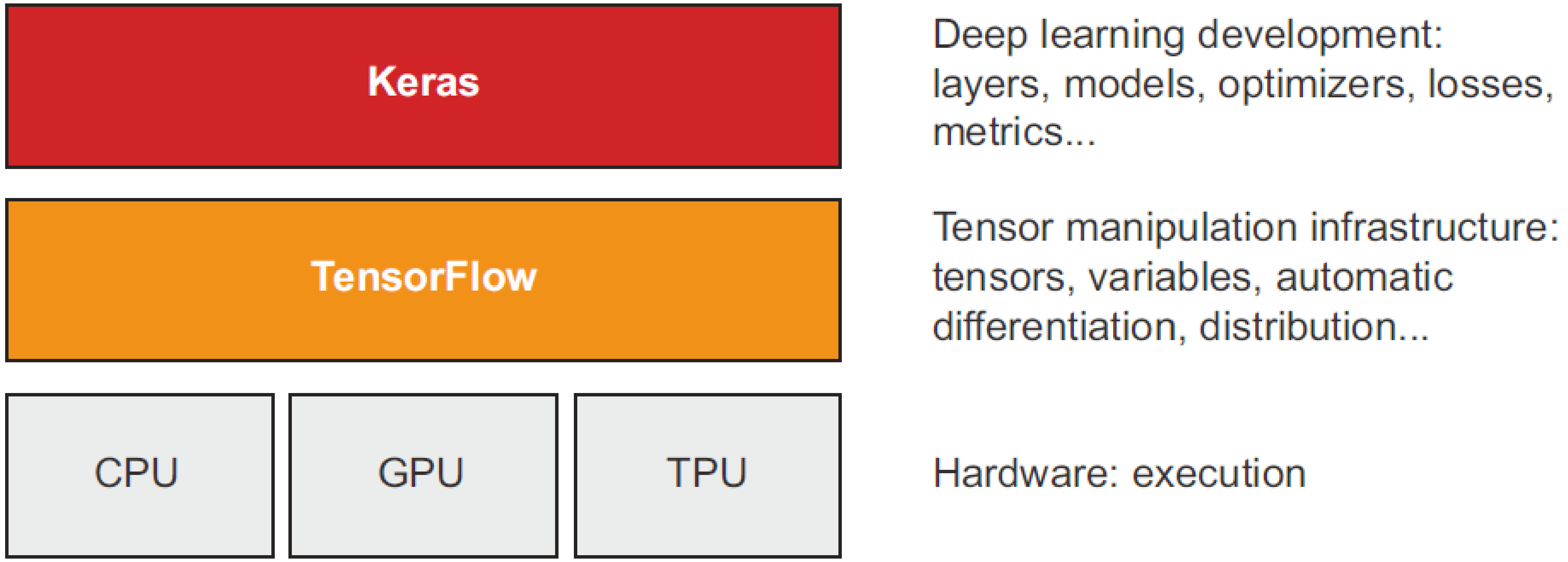


Tensorflow

- It can automatically compute the gradient of any differentiable expression (as you saw in chapter 2), making it highly suitable for machine learning.
- It can run not only on CPUs, but also on GPUs and TPUs, highly parallel hardware accelerators.
- Computation defined in TensorFlow can be easily distributed across many machines.
- TensorFlow programs can be exported to other runtimes, such as C++, Java-Script (for browser-based applications), or TensorFlow Lite (for applications running on mobile devices or embedded devices), etc. This makes TensorFlow applications easy to deploy in practical settings.

Keras

TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API


Keras
TensorFlow
CPU
GPU
TPU
Deep learning development: layers, models, optimizers, losses, metrics...
Tensor manipulation infrastructure: tensors, variables, automatic differentiation, distribution...
Hardware: execution

Deep Learning on a GPU

You have 4 options:

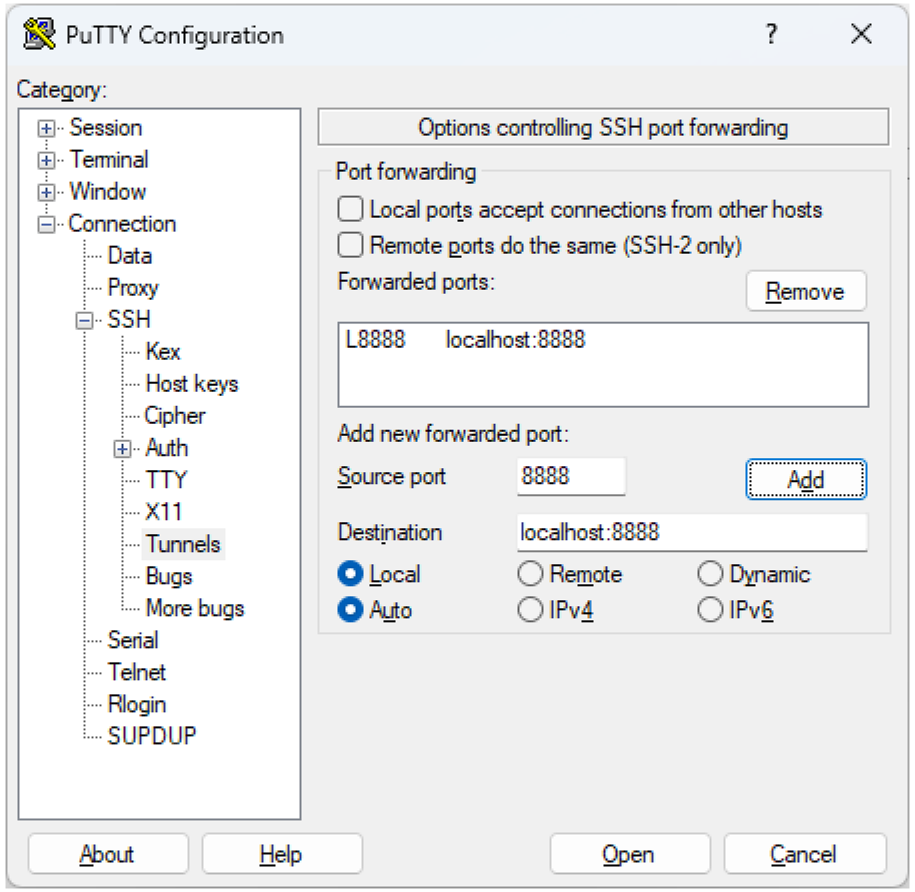
- Use https://labs.azure.com/virtualmachines: your lab fee paid for this
- Buy and install a physical NVIDIA GPU on your workstation
- Use GPU instances on Google Cloud or AWS EC2
- Use the free GPU runtime from Colaboratory, a hosted notebook service offered by Google (for details about what a “notebook” is, see the next section)

Configuring Putty for Notebook Tunneling

Connection > SSH > Tunnels

- Source port: 8888
- Destination: localhost:8888
- Click “Add”

$ jupyter-notebook



Options controlling SSH port forwarding

Port forwarding

[ ] Local ports accept connections from other hosts

[ ] Remote ports do the same (SSH-2 only)

Forwarded ports: Remove

```
L8888      localhost:8888
```

Add new forwarded port:

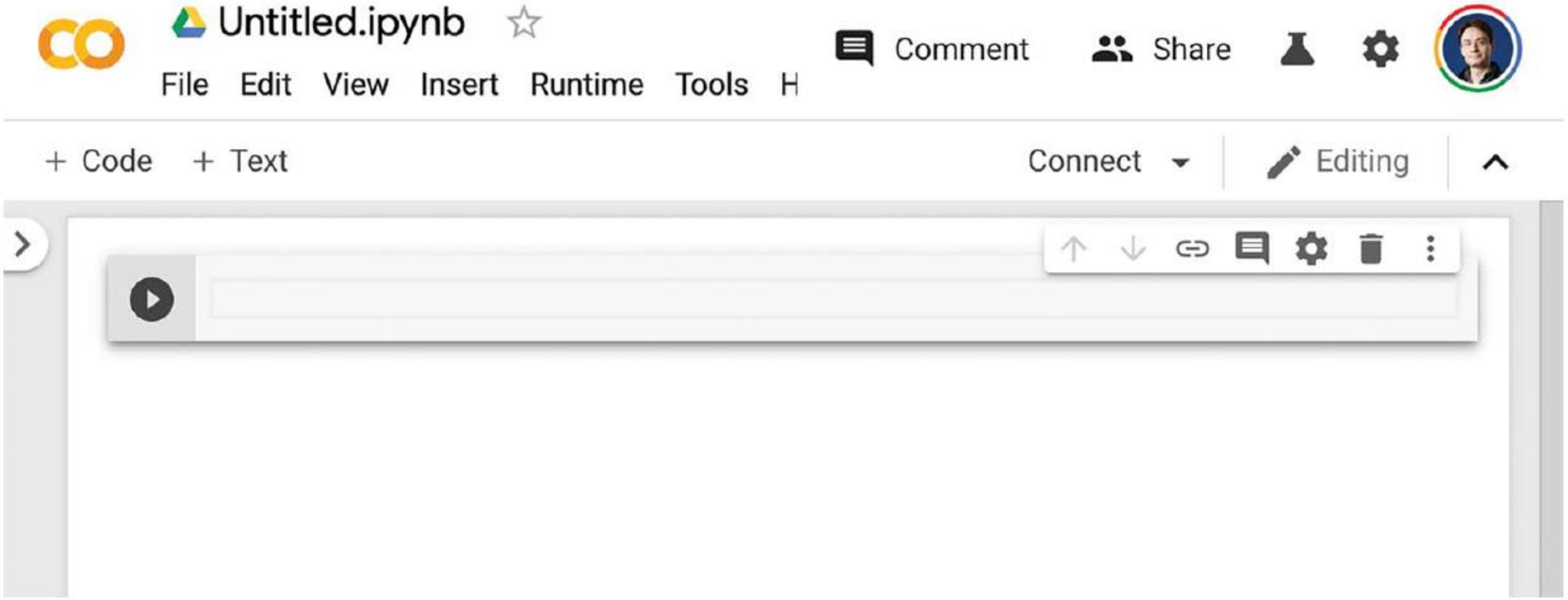
Source port: 8888 — Add

Destination: localhost:8888

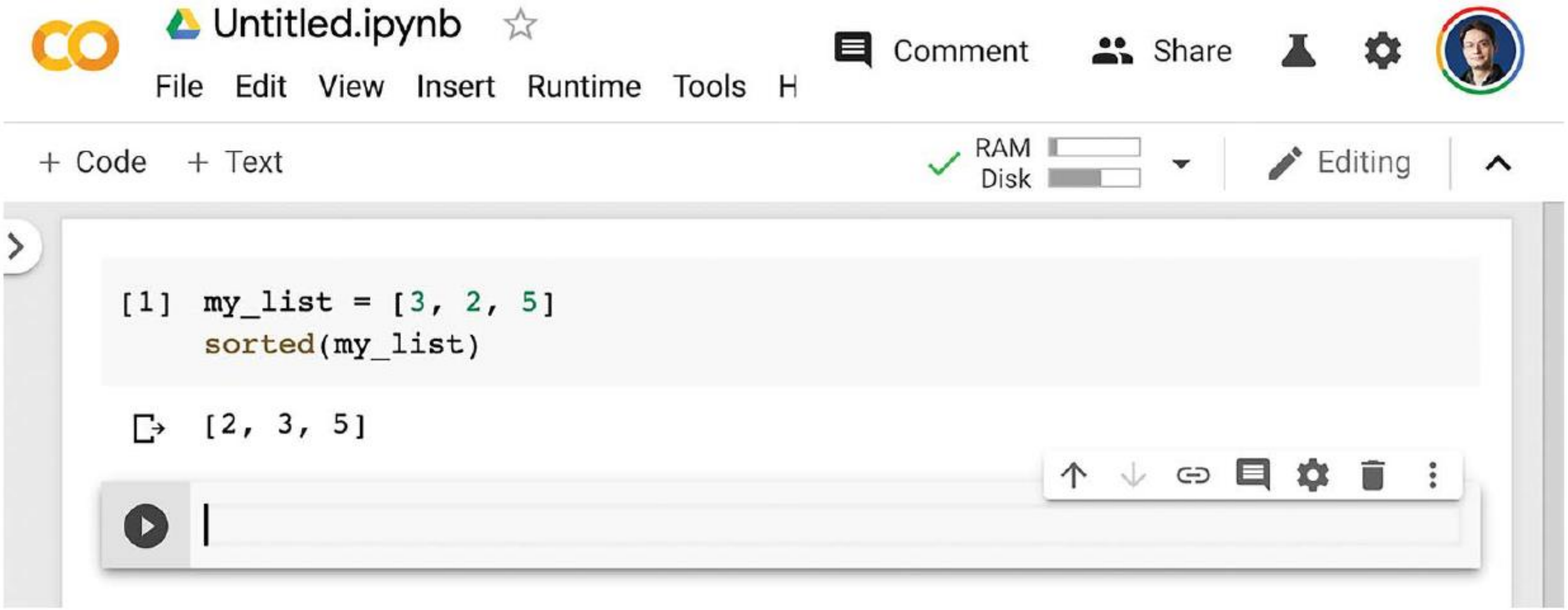
[✓ Local] Remote Dynamic

[✓ Auto] IPv4 IPv6

Colab Notebook



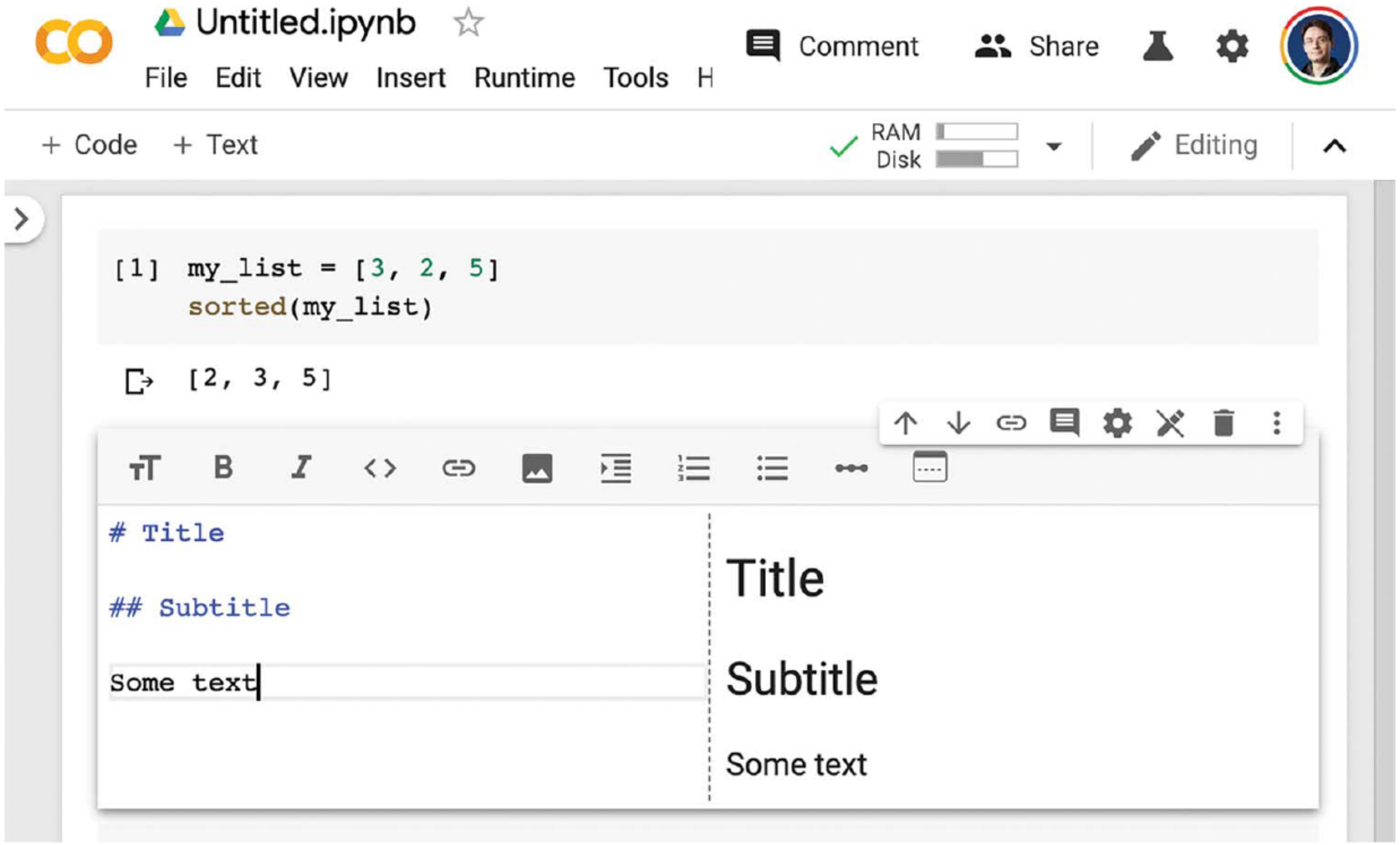
Example Code Cell



```
[1] my_list = [3, 2, 5]
    sorted(my_list)
```

```
[2, 3, 5]
```

Example Text Cell



```
[1] my_list = [3, 2, 5]
    sorted(my_list)

[2, 3, 5]
```

```
# Title

## Subtitle

Some text
```

Title

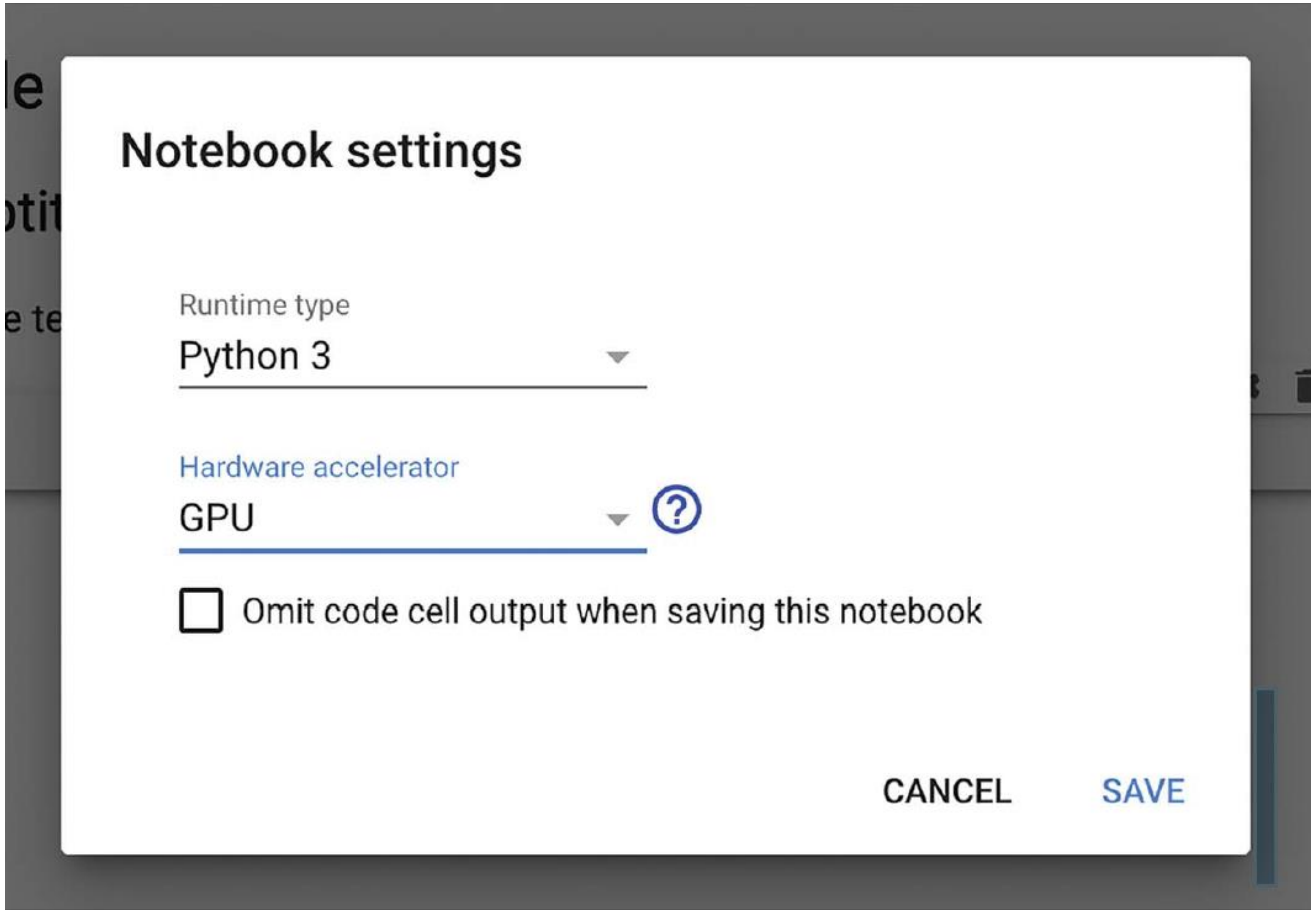
Subtitle

Some text

Running Commands in Notebook

```
!pip install package_name
```

Selecting Runtime and Accelerator



Notebook settings

Runtime type
Python 3

Hardware accelerator
GPU

[ ] Omit code cell output when saving this notebook

Training a Neural Network

- First, low-level tensor manipulation—the infrastructure that underlies all modern machine learning. This translates to TensorFlow APIs:
  - *Tensors*, including special tensors that store the network's state (*variables*)
  - *Tensor operations* such as addition, `relu`, `matmul`
  - *Backpropagation*, a way to compute the gradient of mathematical expressions (handled in TensorFlow via the `GradientTape` object)
- Second, high-level deep learning concepts. This translates to Keras APIs:
  - *Layers*, which are combined into a *model*
  - A *loss function*, which defines the feedback signal used for learning
  - An *optimizer*, which determines how learning proceeds
  - *Metrics* to evaluate model performance, such as accuracy
  - A *training loop* that performs mini-batch stochastic gradient descent

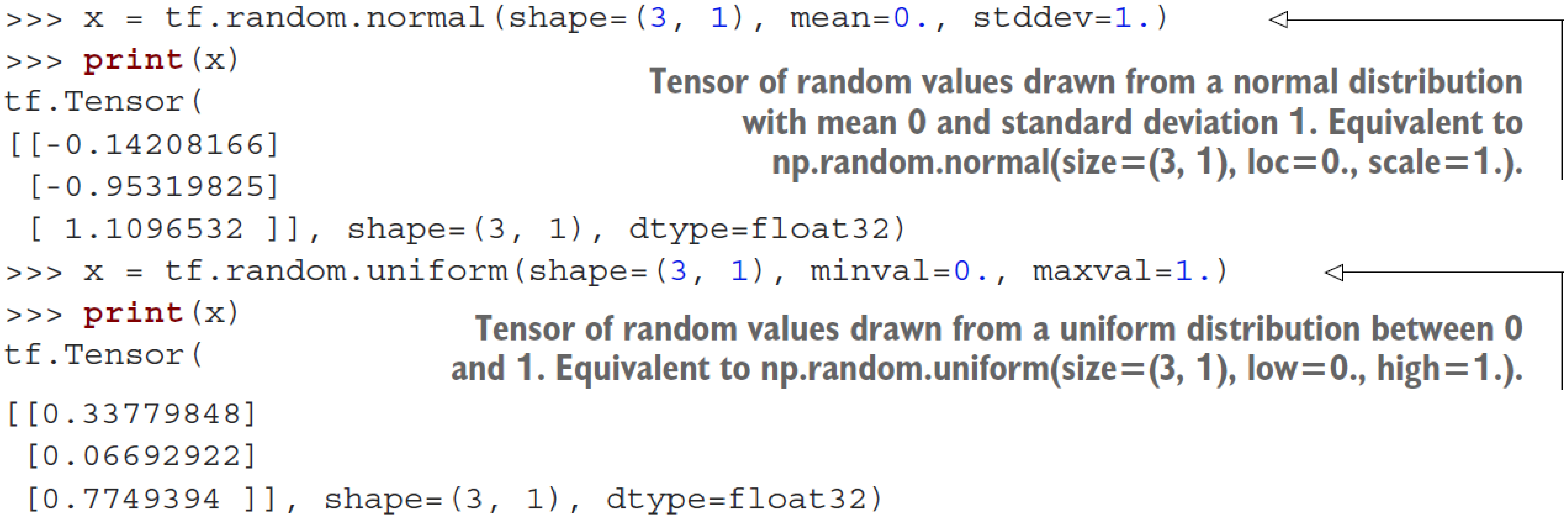
All Ones or All Zeros

```
>>> import tensorflow as tf
>>> x = tf.ones(shape=(2, 1))
>>> print(x)
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
>>> x = tf.zeros(shape=(2, 1))
>>> print(x)
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

Equivalent to np.ones(shape=(2, 1))

Equivalent to np.zeros(shape=(2, 1))

Random Tensors



```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
>>> print(x)
tf.Tensor(
[[-0.14208166]
 [-0.95319825]
 [ 1.1096532 ]], shape=(3, 1), dtype=float32)
>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
>>> print(x)
tf.Tensor(
[[0.33779848]
 [0.06692922]
 [0.7749394 ]], shape=(3, 1), dtype=float32)
```

Tensor of random values drawn from a normal distribution with mean 0 and standard deviation 1. Equivalent to np.random.normal(size=(3, 1), loc=0., scale=1.).

Tensor of random values drawn from a uniform distribution between 0 and 1. Equivalent to np.random.uniform(size=(3, 1), low=0., high=1.).

NumPy Arrays are Assignable TensorFlow Tensors are Not Assignable

```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

```
x = tf.ones(shape=(2, 2))
x[0, 0] = 0.
```

This will fail, as a tensor isn't assignable.

TensorFlow Variables Can Be Modified

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
>>> print(v)
array([[-0.75133973],
       [-0.4872893 ],
       [ 1.6626885 ]], dtype=float32)>

>>> v.assign(tf.ones((3, 1)))
array([[1.],
       [1.],
       [1.]], dtype=float32)>

>>> v[0, 0].assign(3.)
array([[3.],
       [1.],
       [1.]], dtype=float32)>

>>> v.assign_add(tf.ones((3, 1)))
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

Math in TensorFlow

```
a = tf.ones((2, 2))
b = tf.square(a)
c = tf.sqrt(a)
d = b + c
e = tf.matmul(a, b)
e *= d
```

Take the square.

Take the square root.

Add two tensors (element-wise).

Take the product of two tensors (as discussed in chapter 2).

Multiply two tensors (element-wise).

Using the Gradient Tape

```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
   result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

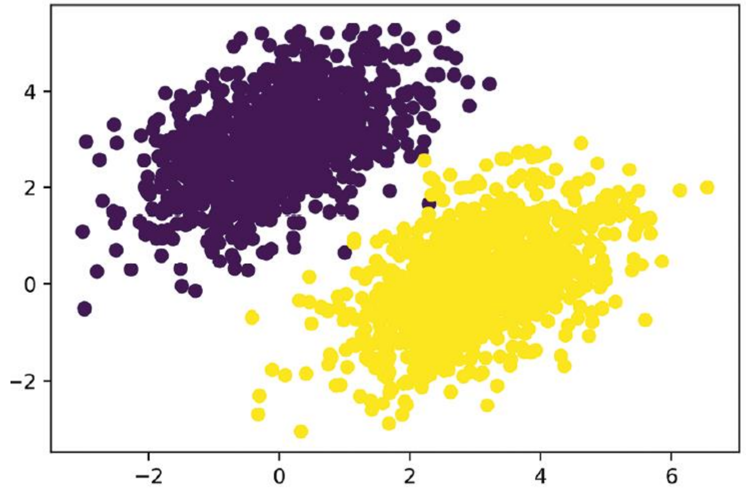
```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
   tape.watch(input_const)
   result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

```
time = tf.Variable(0.)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position =  4.9 * time ** 2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

We use the outer tape to compute the gradient of the gradient from the inner tape. Naturally, the answer is 4.9 * 2 = 9.8.

First Steps

Generating Two Classes



```
num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
```

Generate the first class of points: 1000 random 2D points. cov=[[1, 0.5],[0.5, 1]] corresponds to an oval-like point cloud oriented from bottom left to top right.

Generate the other class of points with a different mean and the same covariance matrix.

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)

targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

TensorFlow: Variables and Model

The inputs will be 2D points.

The output predictions will be a single score per sample (close to 0 if the sample is predicted to be in class 0, and close to 1 if the sample is predicted to be in class 1).

```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))



def model(inputs):
    return tf.matmul(inputs, W) + b
```

TensorFlow: Loss

per_sample_losses will be a tensor with the same shape as targets and predictions, containing per-sample loss scores.

```
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

We need to average these per-sample loss scores into a single scalar loss value: this is what reduce_mean does.

TensorFlow: Training Step

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(targets, predictions)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss

for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

Retrieve the gradient of the loss with regard to weights.

Forward pass, inside a gradient tape scope

Update the weights.

TensorFlow: Predictions

w1 * x + w2 * y + b = 0.5

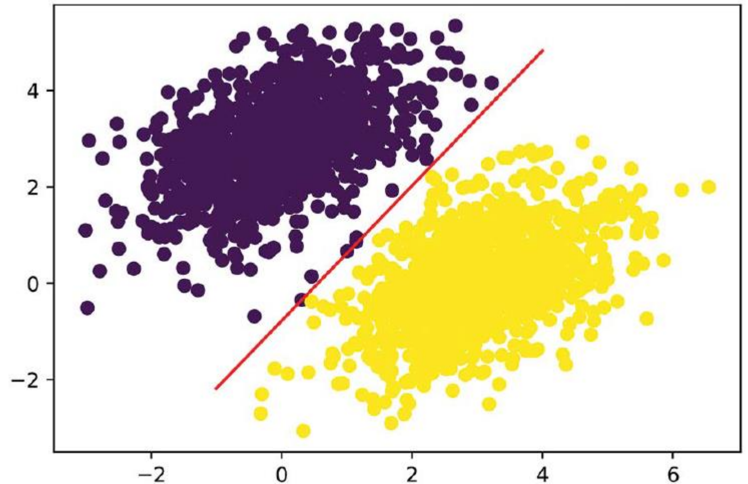
Generate 100 regularly spaced numbers between –1 and 4, which we will use to plot our line.

This is our line's equation.

Plot our line ("-r" means "plot it as a red line").

```
x = np.linspace(-1, 4, 100)
y = - W[0] /  W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

Plot our model's predictions on the same plot.



SimpleDense

```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                                 initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,),
                                 initializer="zeros")

    def call(self, inputs):
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

All Keras layers inherit from the base Layer class.

Weight creation takes place in the build() method.

add_weight() is a shortcut method for creating weights. It is also possible to create standalone variables and assign them as layer attributes, like self.W = tf.Variable(tf.random.uniform(w_shape)).

We define the forward pass computation in the call() method.

Using the SimpleDense Layer

```
>>> my_dense = SimpleDense(units=32, activation=tf.nn.relu)
>>> input_tensor = tf.ones(shape=(2, 784))
>>> output_tensor = my_dense(input_tensor)
>>> print(output_tensor.shape)
(2, 32))
```

Instantiate our layer, defined previously.

Create some test inputs.

Call the layer on the inputs, just like a function.

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

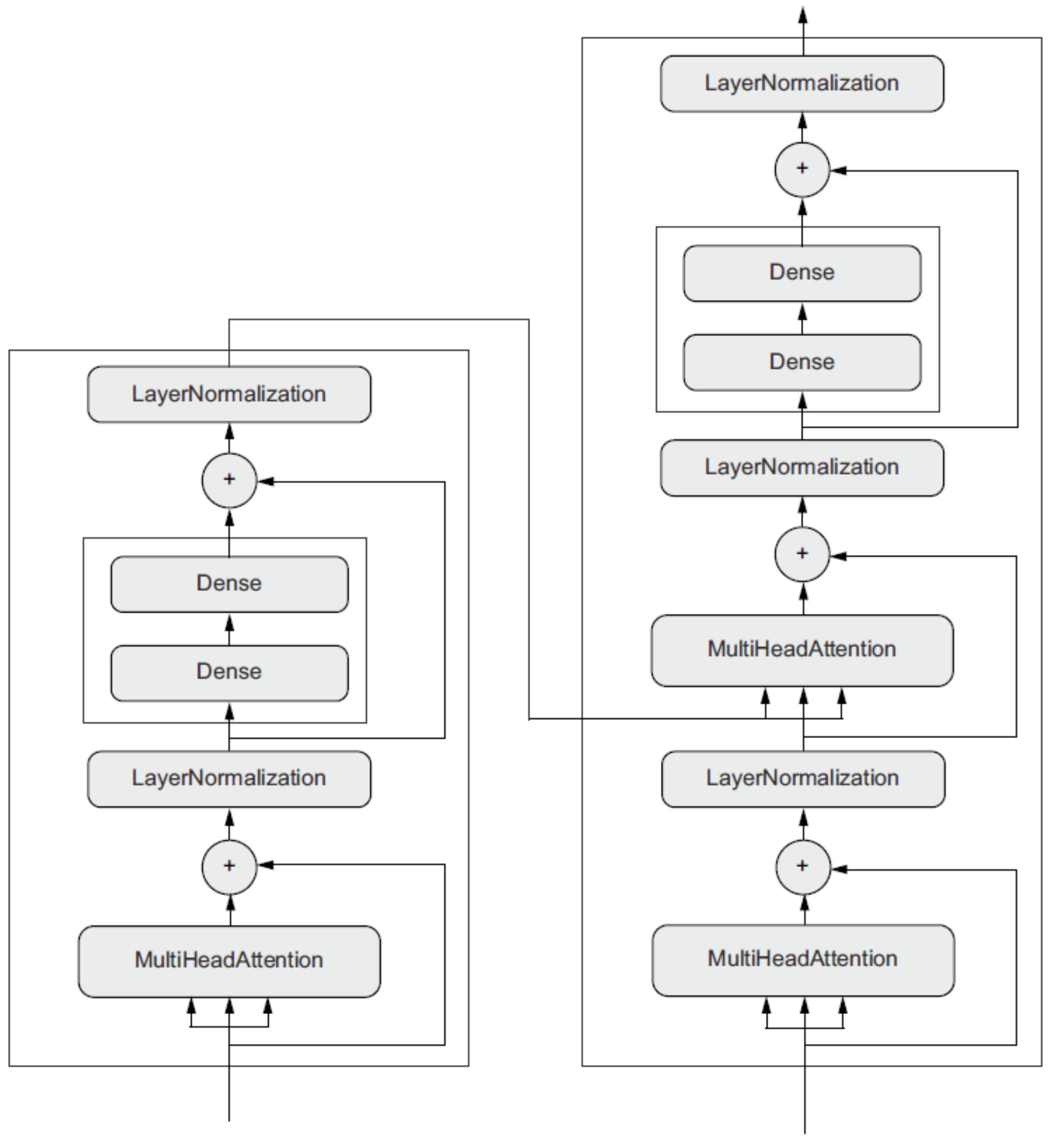
Non-Sequential Models

- Two-branch Networks; e.g. process a pair of sentences
- Multihead Networks; e.g. predict a bounding box and a classification
- Residual Connections; e.g. create short-cut connections to mitigate the problem of vanishing gradients

Surprise Transformer Reference

Sequence-to-Sequence (seq2seq) Example

- Example Applications
  - Language Translation
  - Text Summarization
- Encoder stack on the left [only propagated once
- Decoder stack on the right
  - Start with start-of-text token, to predict the first token
  - Repeat until end-of-text token predicted, or max length achieved
- Residual connections around the MultiHeadAttention blocks and Dense blocks



Compile (Configuration) Step

- Loss function (objective function)—The quantity that will be minimized during training. It represents a measure of success for the task at hand.
- Optimizer—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).
- Metrics—The measures of success you want to monitor during training and validation, such as classification accuracy. Unlike the loss, training will not optimize directly for these metrics. As such, metrics don’t need to be differentiable.

Compile Examples

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer="rmsprop",
              loss="mean_squared_error",
              metrics=["accuracy"])
```

Define a linear classifier.

Specify the optimizer by name: RMSprop (it's case-insensitive).

Specify the loss by name: mean squared error.

Specify a list of metrics: in this case, only accuracy.

```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
              loss=my_custom_loss,
              metrics=[my_custom_metric_1, my_custom_metric_2])
```

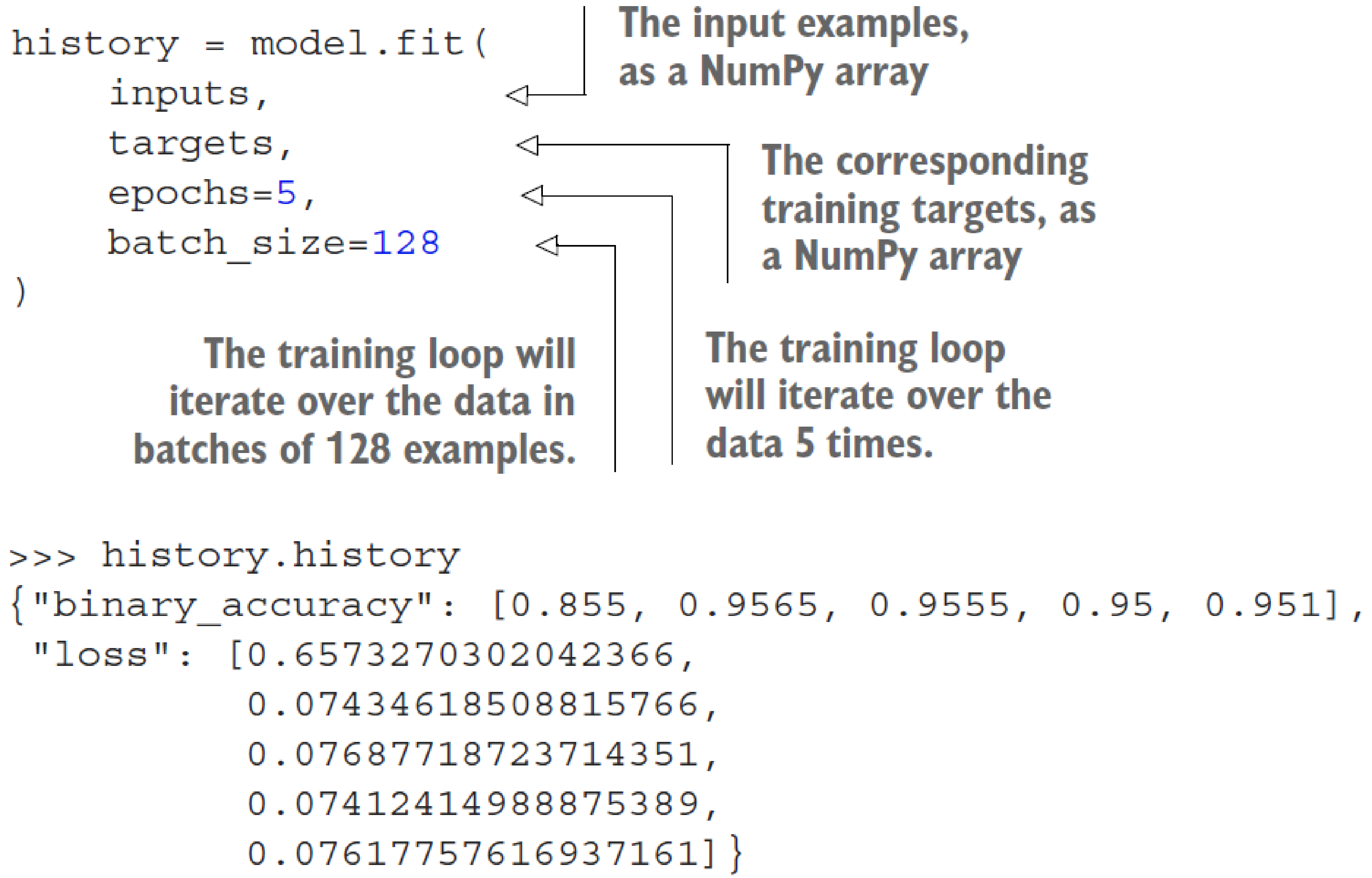
Compile Options

- Optimizers
  - SGD
  - RMSProp
  - Adam
  - Adagrad
  - Etc.
- Losses
  - CategoricalCrossentropy
  - SparseCategoricalCrossentropy
  - BinaryCrossentropy
  - MeanSquaredError
  - KLDivergence
  - CosineSimilarity
  - Etc.
- Metrics
  - CategoricalAccuracy
  - SparseCategoricalAccuracy
  - BinaryAccuracy
  - AUC
  - Precision
  - Recall
  - Etc.

The Fit Method

- The data (inputs and targets) to train on. It will typically be passed either in the form of NumPy arrays or a TensorFlow Dataset object. You'll learn more about the Dataset API in the next chapters.
- The number of epochs to train for: how many times the training loop should iterate over the data passed.
- The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.

Calling the Fit Method



```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

The input examples, as a NumPy array

The corresponding training targets, as a NumPy array

The training loop will iterate over the data 5 times.

The training loop will iterate over the data in batches of 128 examples.

```
>>> history.history
{"binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],
 "loss": [0.6573270302042366,
          0.07434618508815766,
          0.07687718723714351,
          0.07412414988875389,
          0.07617757616937161]}
```

Using the validation_data Argument

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

To avoid having samples from only one class in the validation data, shuffle the inputs and targets using a random indices permutation.

Reserve 30% of the training inputs and targets for validation (we'll exclude these samples from training and reserve them to compute the validation loss and metrics).

Training data, used to update the weights of the model

Validation data, used only to monitor the validation loss and metrics

Using the Model After Training

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

```
predictions = model(new_inputs)
```

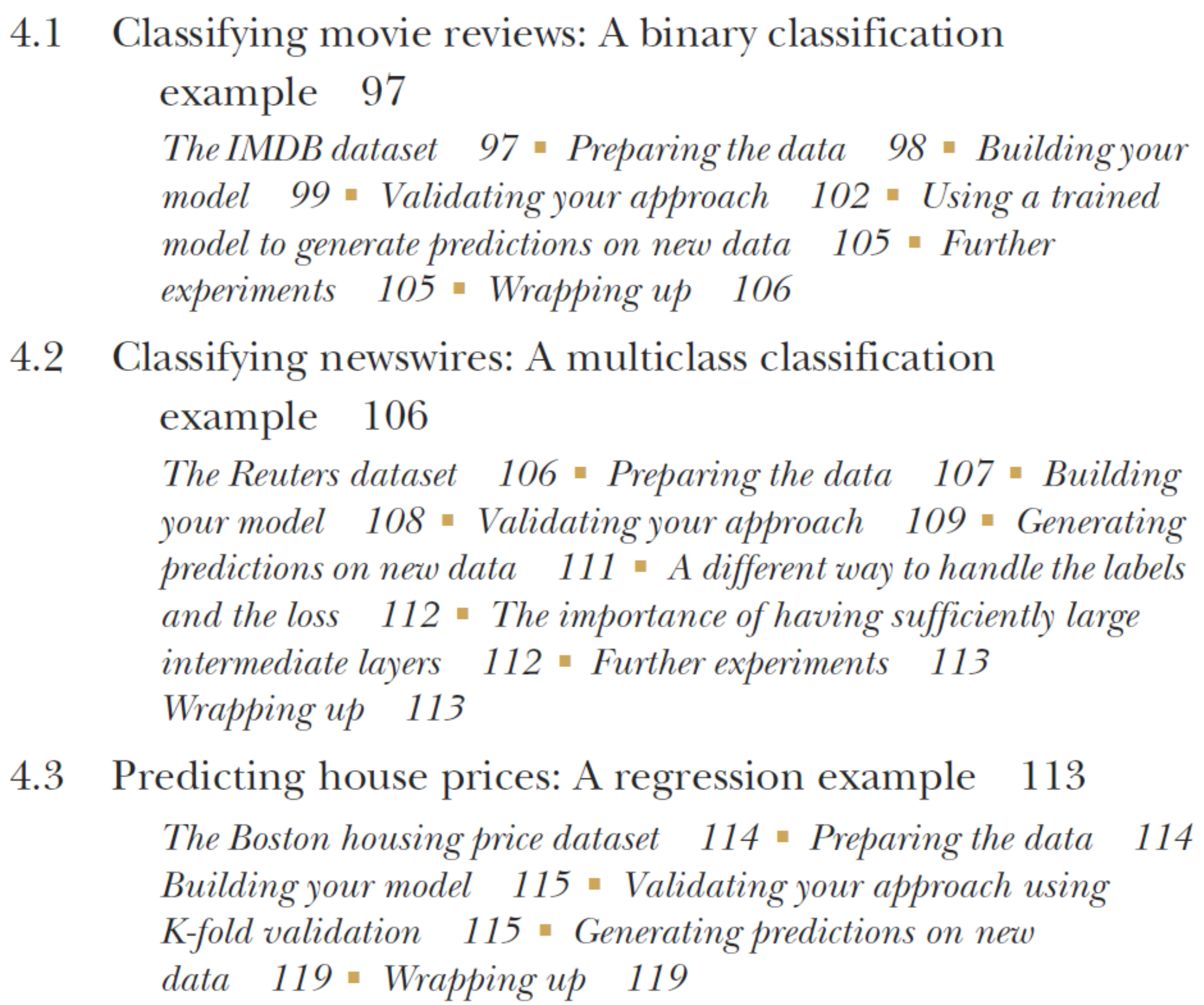
Takes a NumPy array or TensorFlow tensor and returns a TensorFlow tensor

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10])
[[0.3590725 ]
 [0.82706255]
 [0.74428225]
 [0.682058  ]
 [0.7312616 ]
 [0.6059811 ]
 [0.78046083]
 [0.025846  ]
 [0.16594526]
 [0.72068727]]
```

Summary

- TensorFlow is an industry-strength numerical computing framework that can run on CPU, GPU, or TPU. It can automatically compute the gradient of any differentiable expression, it can be distributed to many devices, and it can export programs to various external runtimes—even JavaScript.
- Keras is the standard API for doing deep learning with TensorFlow. It's what we'll use throughout this book.
- Key TensorFlow objects include tensors, variables, tensor operations, and the gradient tape.
- The central class of Keras is the Layer. A layer encapsulates some weights and some computation. Layers are assembled into models.
- Before you start training a model, you need to pick an optimizer, a loss, and some metrics, which you specify via the model.compile() method.
- To train a model, you can use the fit() method, which runs mini-batch gradient descent for you. You can also use it to monitor your loss and metrics on validation data, a set of inputs that the model doesn't see during training.
- Once your model is trained, you use the model.predict() method to generate predictions on new inputs.

Getting Started: Classification and Regression



3 Case Studies

- Classifying movie reviews as positive or negative (binary classification)
- Classifying news wires by topic (multiclass classification)
- Estimating the price of a house, given real-estate data (scalar regression)

Classification and Regression Glossary

- Sample or input
- Prediction or output
- Target
- Prediction error or loss value
- Classes
- Label
- Ground-truth or annotations
- Binary classification
- Multiclass classification
- Multilabel classification
- Scalar regression
- Vector regression (e.g. bounding box)
- Mini-batch or batch

Internet Movie DataBase (IMDB)

- 25,000 positive: {7, 8, 9, 10} stars
- 25,000 negative: {1, 2, 3, 4 } stars

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)

>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
>>> train_labels[0]
1

>>> max([max(sequence) for sequence in train_data])
9999
```

Decoding Reviews

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in train_data[0]])
```

word_index is a dictionary mapping words to an integer index.

Reverses it, mapping integer indices to words

Decodes the review. Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

Multi-Hot Encoding of Documents (Reviews)

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Creates an all-zero matrix of shape (len(sequences), dimension)

Sets specific indices of results[i] to 1s

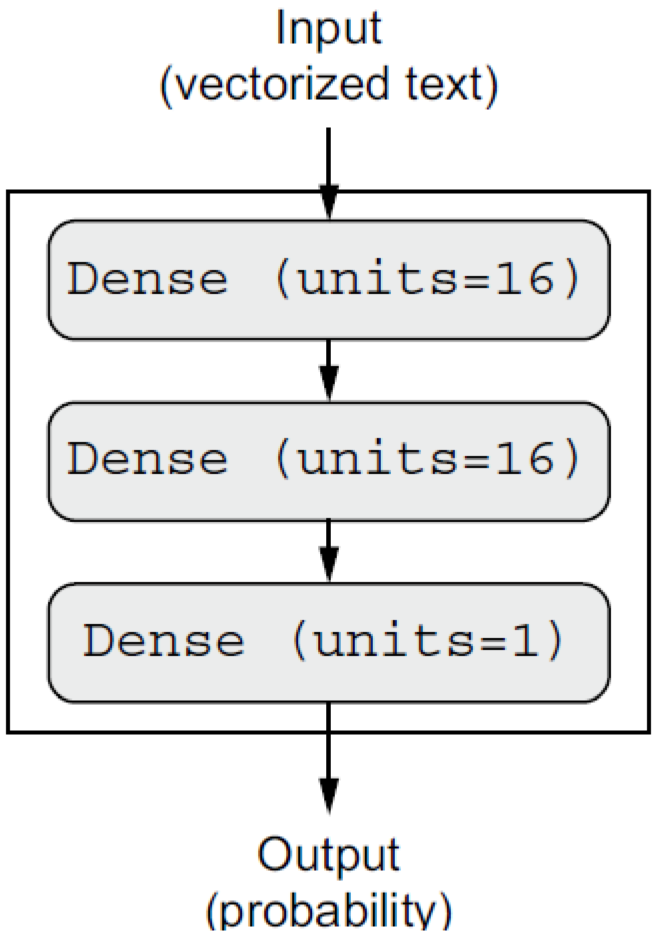
Vectorized training data

Vectorized test data

```
>>> x_train[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.])
```

```
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

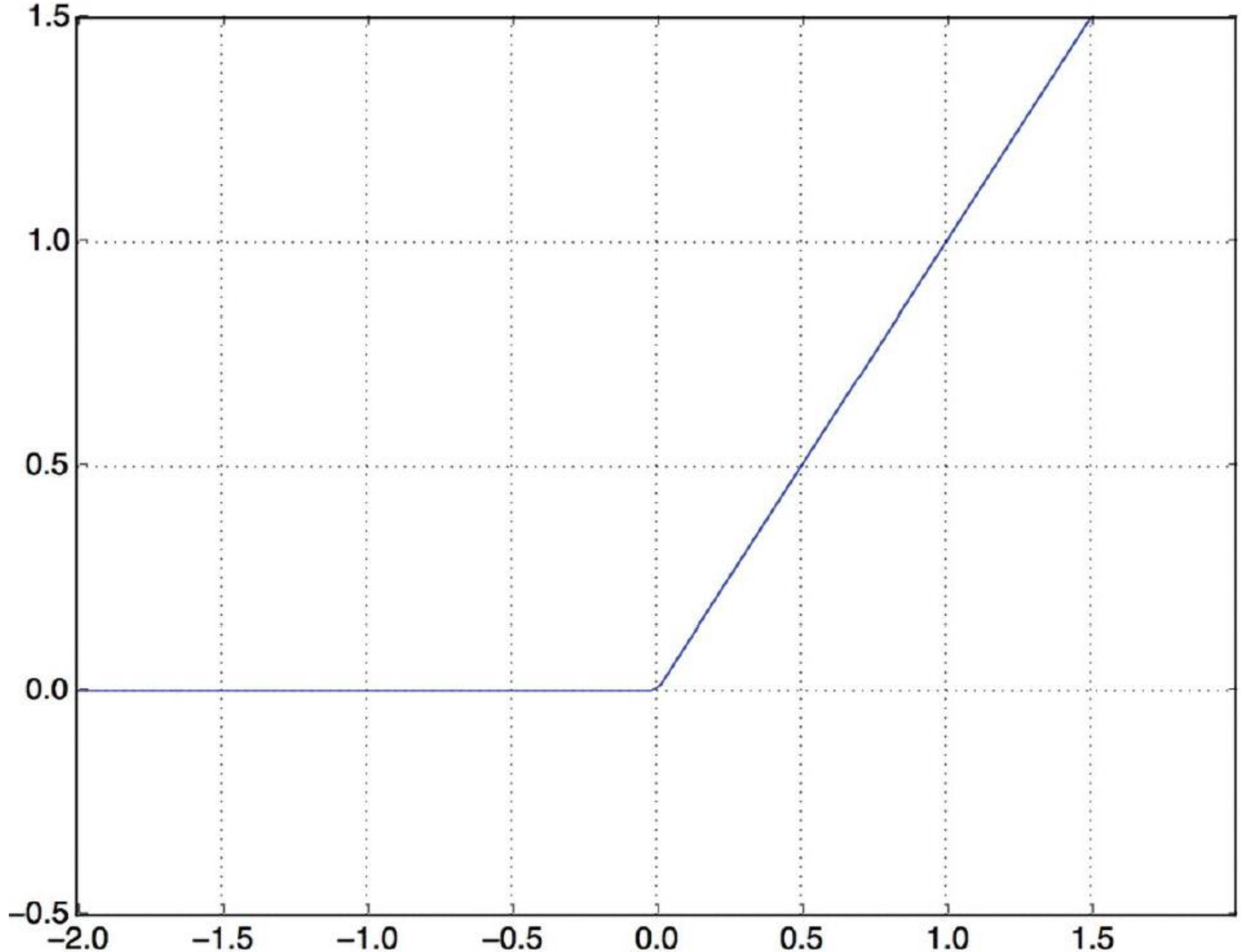
The IMDB Model



```
from tensorflow import keras
from tensorflow.keras import layers

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


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

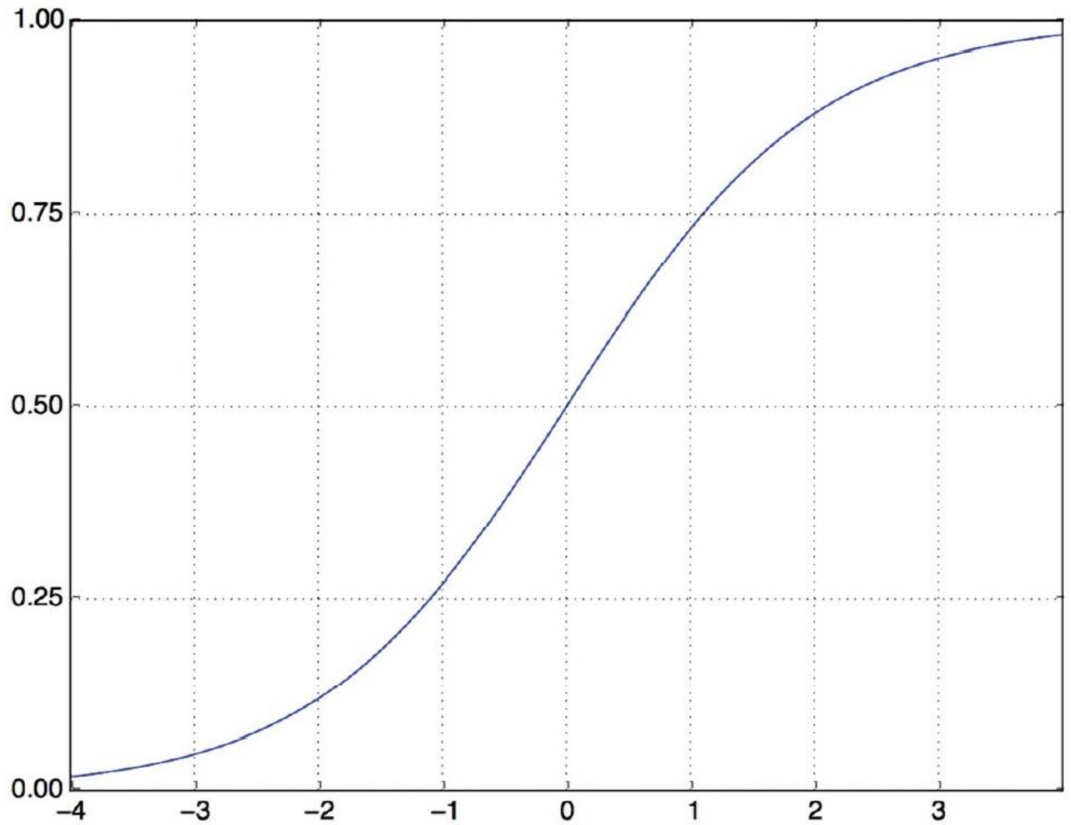
Rectified Linear Unit (ReLU) Activation

It was linear 'til someone rect it ☺



Sigmoid Activation

Also Known As (AKA) the Logistic Function, mapping log odds to a probability estimate



Without the Activation Function?

- The activation function is required to learn non-linear transforms, like the eXclusive OR (XOR) example we reviewed earlier
- Without a linear activation function, any adjacent pair of Dense() layers could be combined into a single equivalent Dense() layer

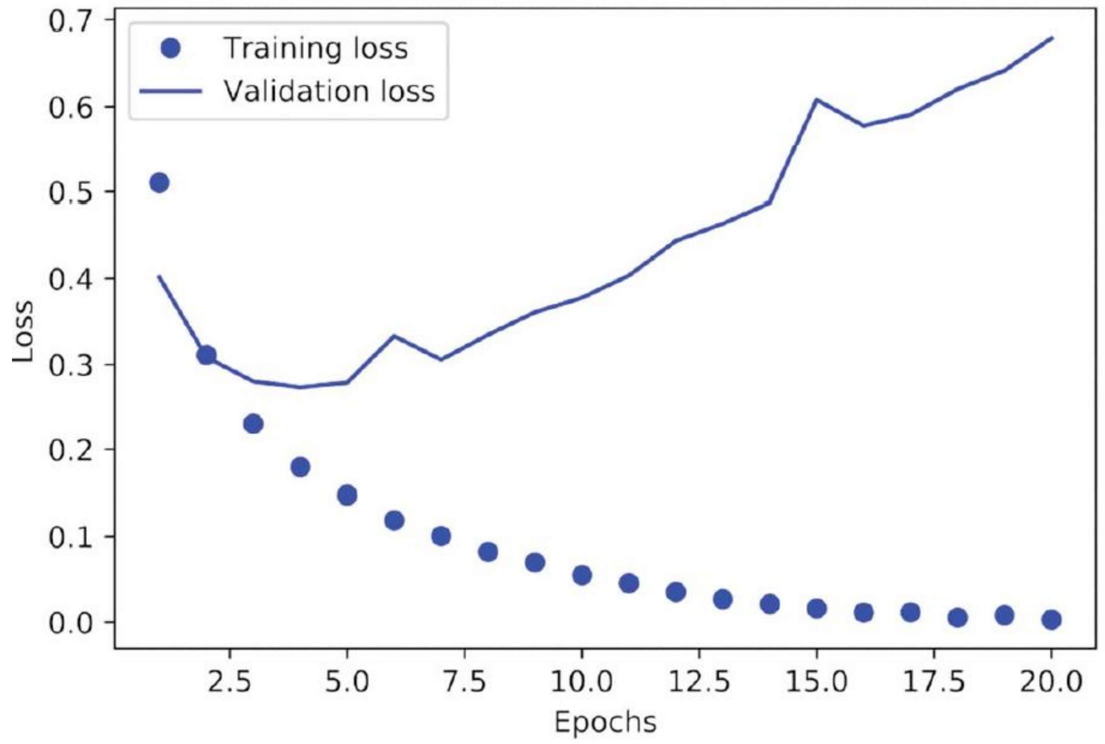
Training the Model

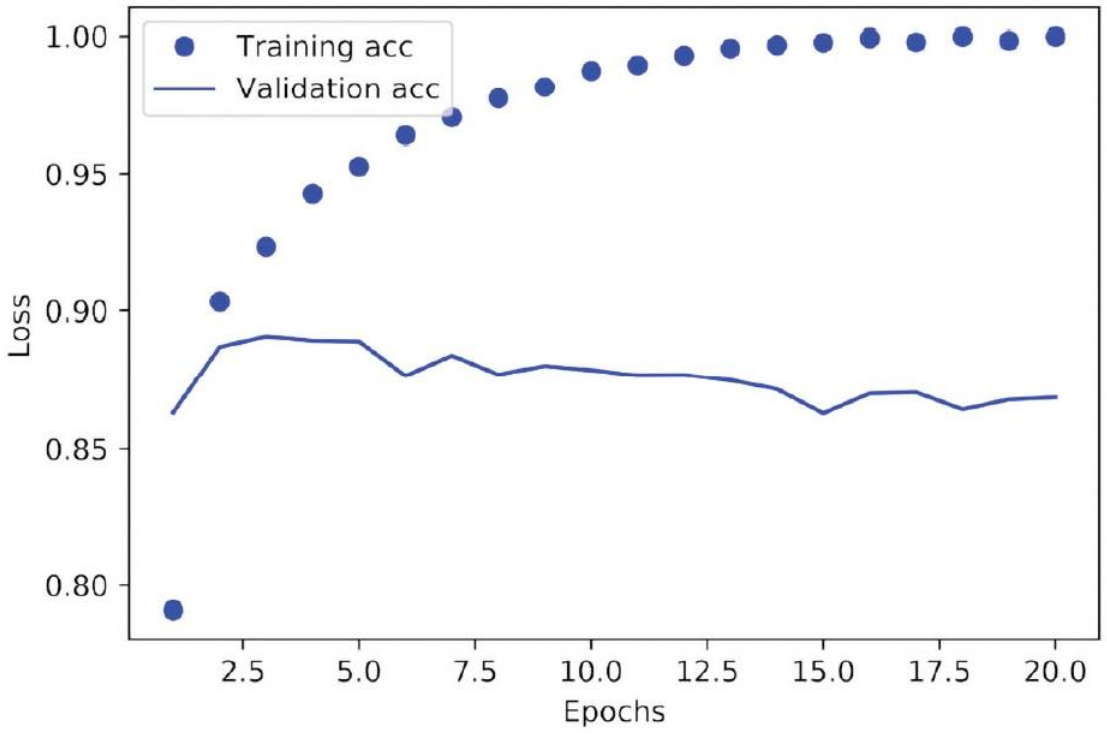
```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
>>> history_dict = history.history
>>> history_dict.keys()
[u"accuracy", u"loss", u"val_accuracy", u"val_loss"]
```

IMDB: Crossentropy and Accuracy





Plotting Crossentropy

```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, "bo", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

"bo" is for "blue dot."

"b" is for "solid blue line."

Plotting Accuracy

```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Clears the figure

Retraining a Model from Scratch

Nota bene: we're using all of the data; e.g. x_train vs partial_x_train

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

The first number, 0.29, is the test loss, and the second number, 0.88, is the test accuracy.

IMDB: Stuff to Try

- You used two representation layers before the final classification layer. Try using one or three representation layers, and see how doing so affects validation and test accuracy.
- Try using layers with more units or fewer units: 32 units, 64 units, and so on.
- Try using the mse loss function instead of binary_crossentropy.
- Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

IMDB: Wrapping Up

- You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it—as tensors—into a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options too.
- Stacks of Dense layers with relu activations can solve a wide range of problems (including sentiment classification), and you'll likely use them frequently.
- In a binary classification problem (two output classes), your model should end with a Dense layer with one unit and a sigmoid activation: the output of your model should be a scalar between 0 and 1, encoding a probability.
- With such a scalar sigmoid output on a binary classification problem, the loss function you should use is binary_crossentropy.
- The rmsprop optimizer is generally a good enough choice, whatever your problem. That's one less thing for you to worry about.
- As they get better on their training data, neural networks eventually start overfitting and end up obtaining increasingly worse results on data they've never seen before. Be sure to always monitor performance on data that is outside of the training set.

Reuters News Articles

46 Topics

- 8,982 training
- 2,246 testing

```
from tensorflow.keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)

>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Vectorized training data

Vectorized test data

Dense CategoricalCrossentropy ☹

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
y_train = to_one_hot(train_labels)      Vectorized training labels
y_test = to_one_hot(test_labels)        Vectorized test labels
```

```
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)
```

Reuters Model

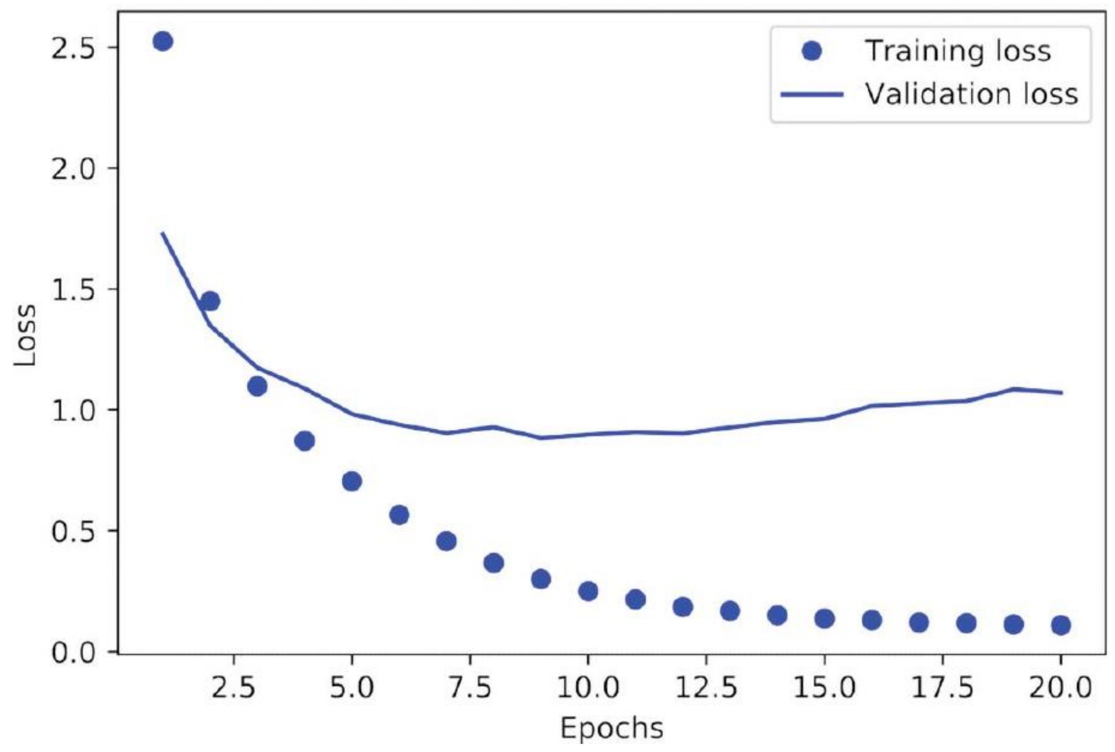
```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
```

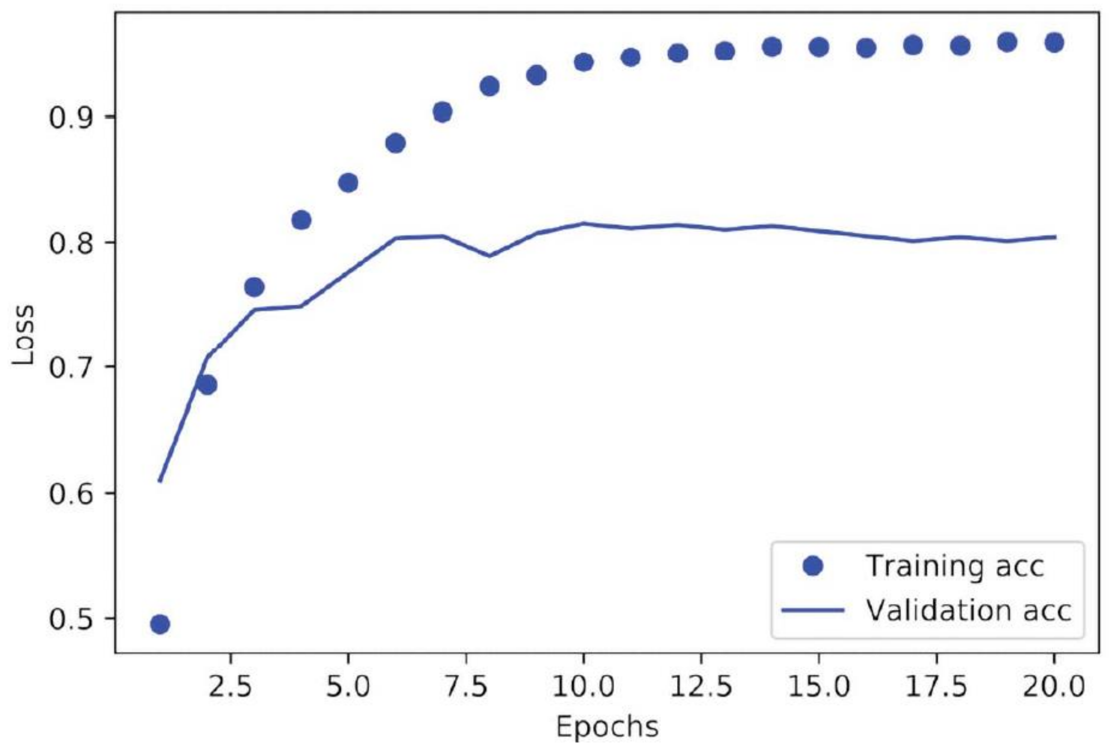
```
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

Training the Reuters Model

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = y_train[:1000]
partial_y_train = y_train[1000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Reuters: Crossentropy and Accuracy





Retrain the Model

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train,
          y_train,
          epochs=9,
          batch_size=512)
results = model.evaluate(x_test, y_test)
```

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

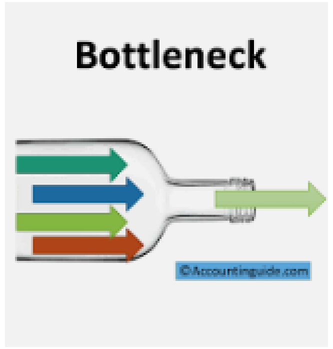
Baseline [null model; no predictors]

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> hits_array.mean()
0.18655387355298308
```

Alternatively … [majority classifier for the win (FTW)]

```
>>> import numpy as np
>>> import pandas as pd
>>> (trnX, trnY), (tstX, tstY) = datasets.reuters.load_data()
>>> np.max(pd.DataFrame(trnY)[0].value_counts())
3159
>>> np.max(pd.DataFrame(trnY)[0].value_counts()) / len(trnY)
0.3517034068136273
```

Information “Bottleneck”



```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

Reuters: Things to Try

- Try using larger or smaller layers: 32 units, 128 units, and so on.
- You used two intermediate layers before the final softmax classification layer. Now try using a single intermediate layer, or three intermediate layers.

Reuters: Wrapping Up

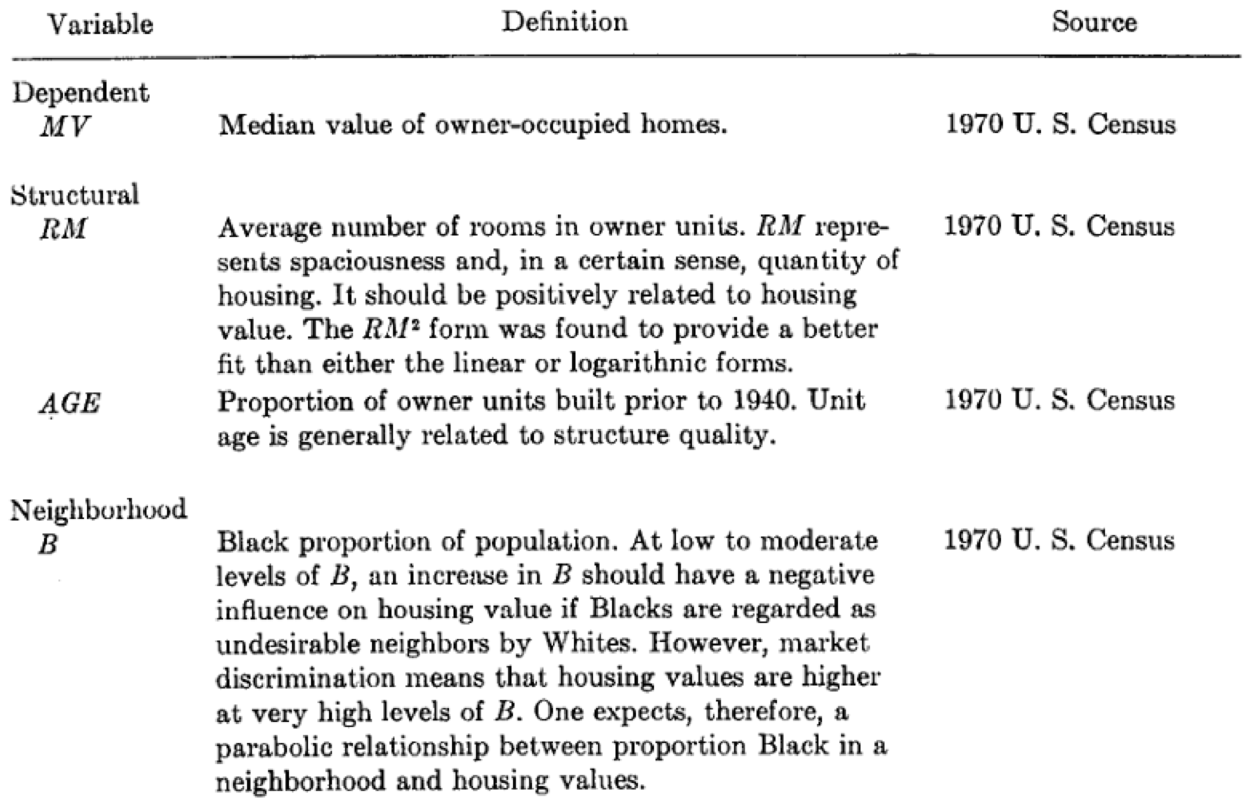
- If you’re trying to classify data points among N classes, your model should end with a Dense layer of size N.
- In a single-label, multiclass classification problem, your model should end with a softmax activation so that it will output a probability distribution over the N output classes.
- Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the model and the true distribution of the targets.
- There are two ways to handle labels in multiclass classification:
  - Encoding the labels via categorical encoding (also known as one-hot encoding) and using categorical_crossentropy as a loss function
  - Encoding the labels as integers and using the sparse_categorical_crossentropy loss function
- If you need to classify data into a large number of categories, you should avoid creating information bottlenecks in your model due to intermediate layers that are too small.

Regression vs Regression?

- “Don’t confuse regression and the logistic regression algorithm. Confusingly, logistic regression isn’t a regression algorithm—it’s a classification algorithm.”
- Pop quiz: in logistic regression, what is the linear model trying to predict?
- Hint: what is the input to a logistic function?

Boston Housing Data: Beware Bias!

- https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html
- Originally published with this paper: https://www.law.berkeley.edu/files/Hedonic.PDF



Variable	Definition	Source
Dependent		
MV	Median value of owner-occupied homes.	1970 U. S. Census
Structural		
RM	Average number of rooms in owner units. *RM* represents spaciousness and, in a certain sense, quantity of housing. It should be positively related to housing value. The RM^2 form was found to provide a better fit than either the linear or logarithmic forms.	1970 U. S. Census
AGE	Proportion of owner units built prior to 1940. Unit age is generally related to structure quality.	1970 U. S. Census
Neighborhood		
B	Black proportion of population. At low to moderate levels of *B*, an increase in *B* should have a negative influence on housing value if Blacks are regarded as undesirable neighbors by Whites. However, market discrimination means that housing values are higher at very high levels of *B*. One expects, therefore, a parabolic relationship between proportion Black in a neighborhood and housing values.	1970 U. S. Census

Strongly disagree with using this predictor: ethics are for everyone

Boston

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) = (
    boston_housing.load_data())
```

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

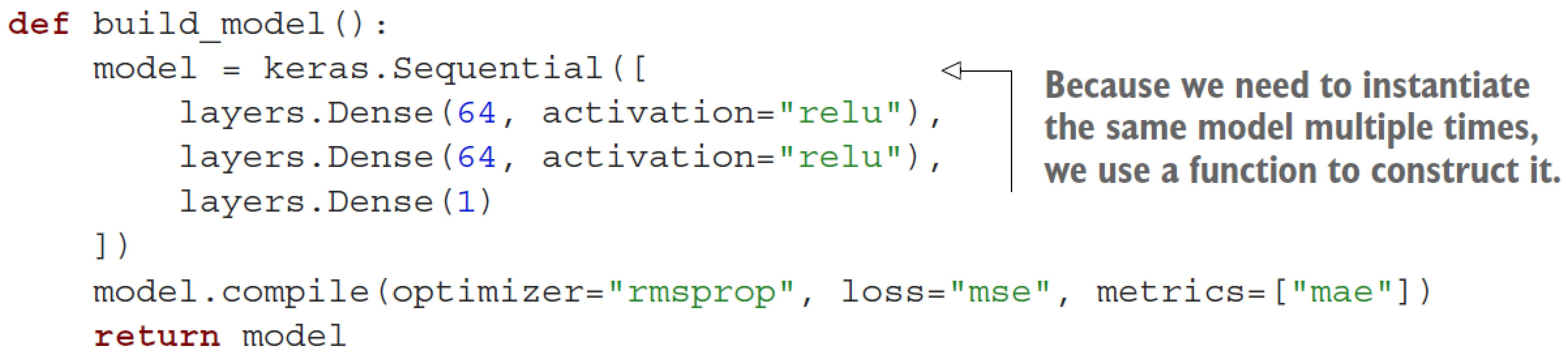
```
>>> train_targets
[ 15.2,  42.3,  50. ...  19.4,  19.4,  29.1]
```

Normalizing: Centering and Scaling the Data

```
mean = train_data.mean(axis=0)
train_data -= mean

std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Boston: Model

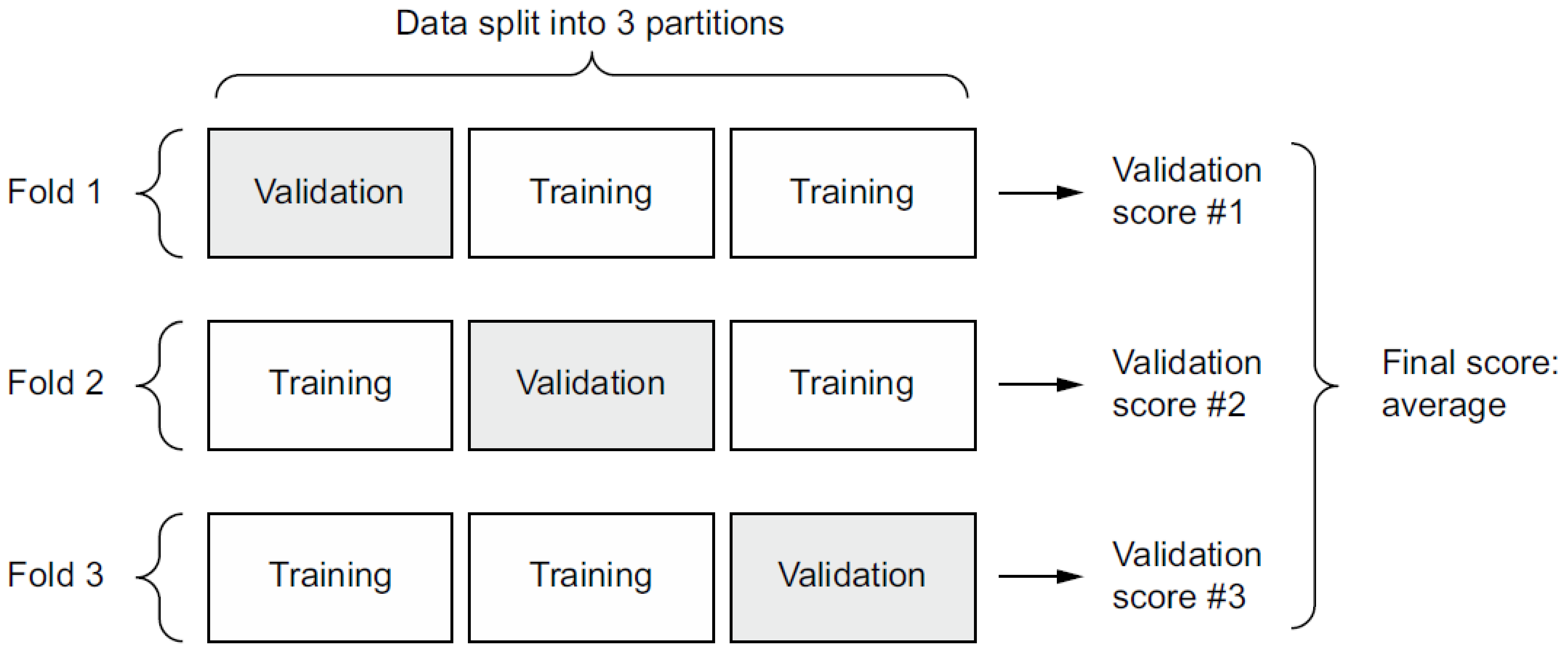
What’s sketchy about this function?



```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

Because we need to instantiate the same model multiple times, we use a function to construct it.

K-Fold Cross Validation (K=3)



K-Fold Cross Validation Implementation

```
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print(f"Processing fold #{i}")
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=16, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose = 0)

Evaluates the model on the validation data

K-Fold Cross Validation Results

```
>>> all_scores
[2.112449, 3.0801501, 2.6483836, 2.4275346]
>>> np.mean(all_scores)
2.5671294
```

Saving Histories

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print(f"Processing fold #{i}")
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=16, verbose=0)
    mae_history = history.history["val_mae"]
    all_mae_histories.append(mae_history)

average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

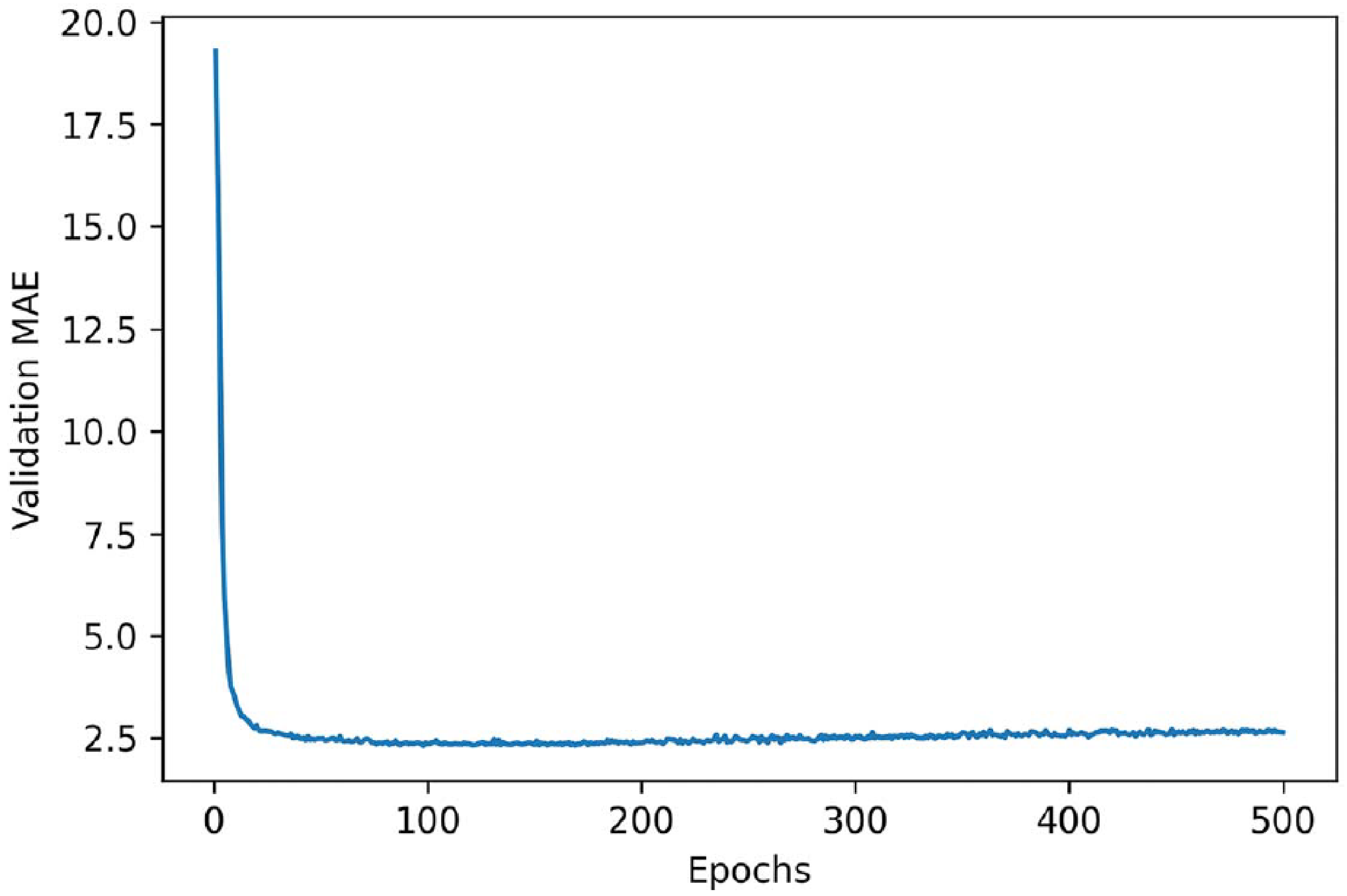
Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

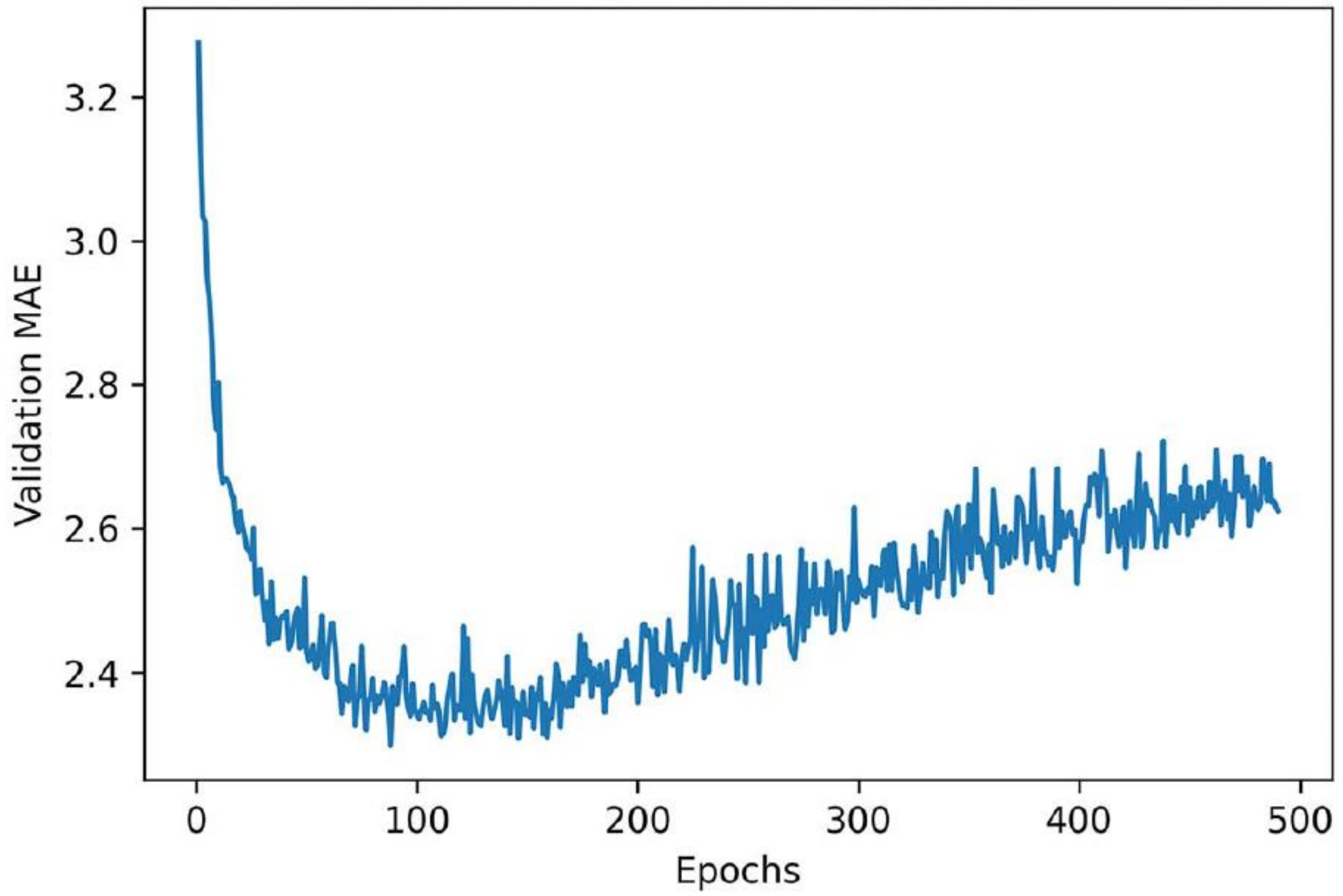
Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose=0)

Averaged Mean Absolute Error Values: Take 1



Excluding the First 10 Points



Building the Final Model

This was necessary, as we're using all of the data …

Gets a fresh, compiled model

Trains it on the entirety of the data

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=130, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

```
>>> test_mae_score
2.4642276763916016
```

Predictions

```
>>> predictions = model.predict(test_data)
>>> predictions[0]
array([9.990133], dtype=float32)
```

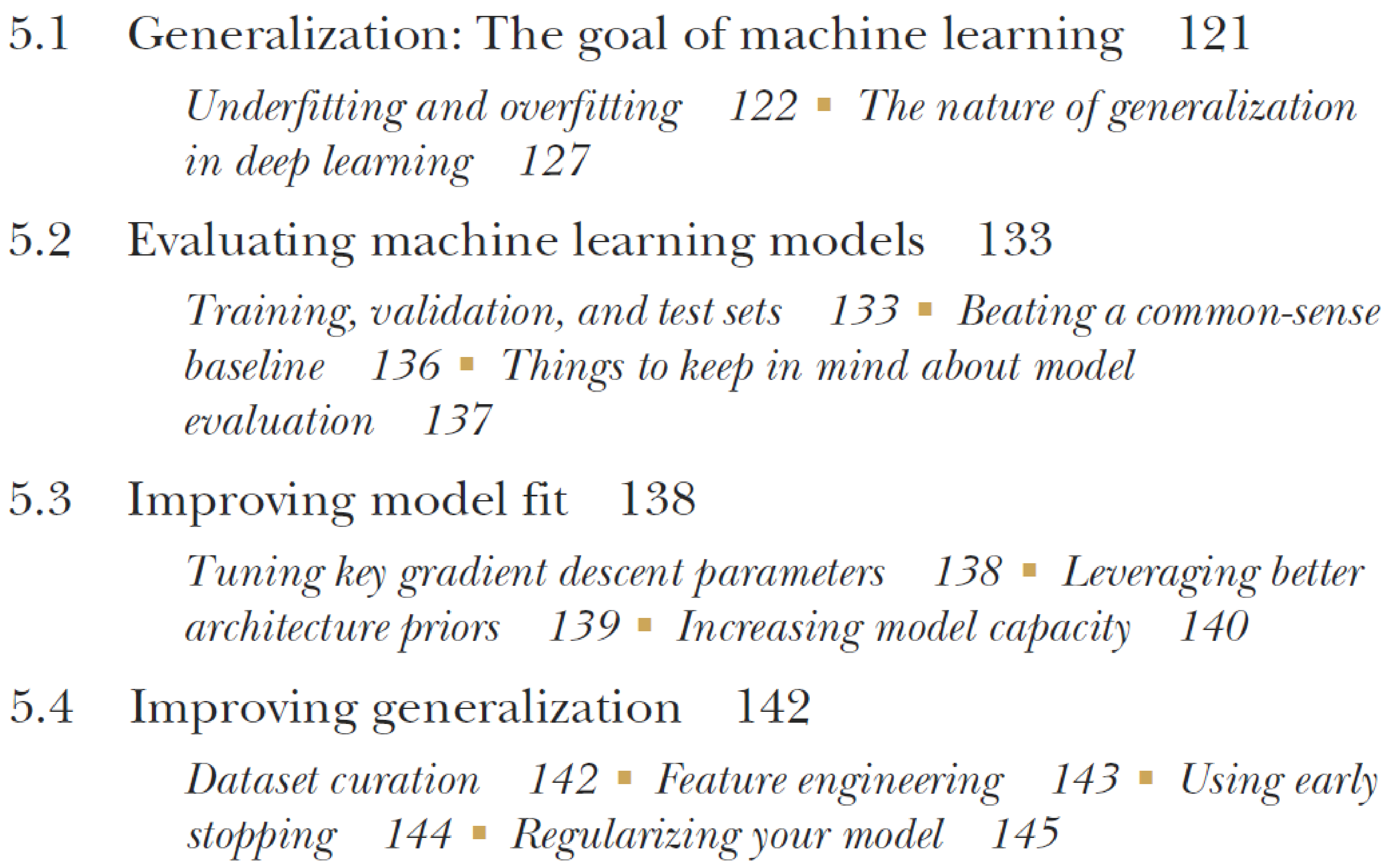
Boston: Wrapping Up

- Regression is done using different loss functions than we used for classification. Mean squared error (MSE) is a loss function commonly used for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn’t apply for regression. A common regression metric is mean absolute error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- When little training data is available, it’s preferable to use a small model with few intermediate layers (typically only one or two), in order to avoid severe overfitting.

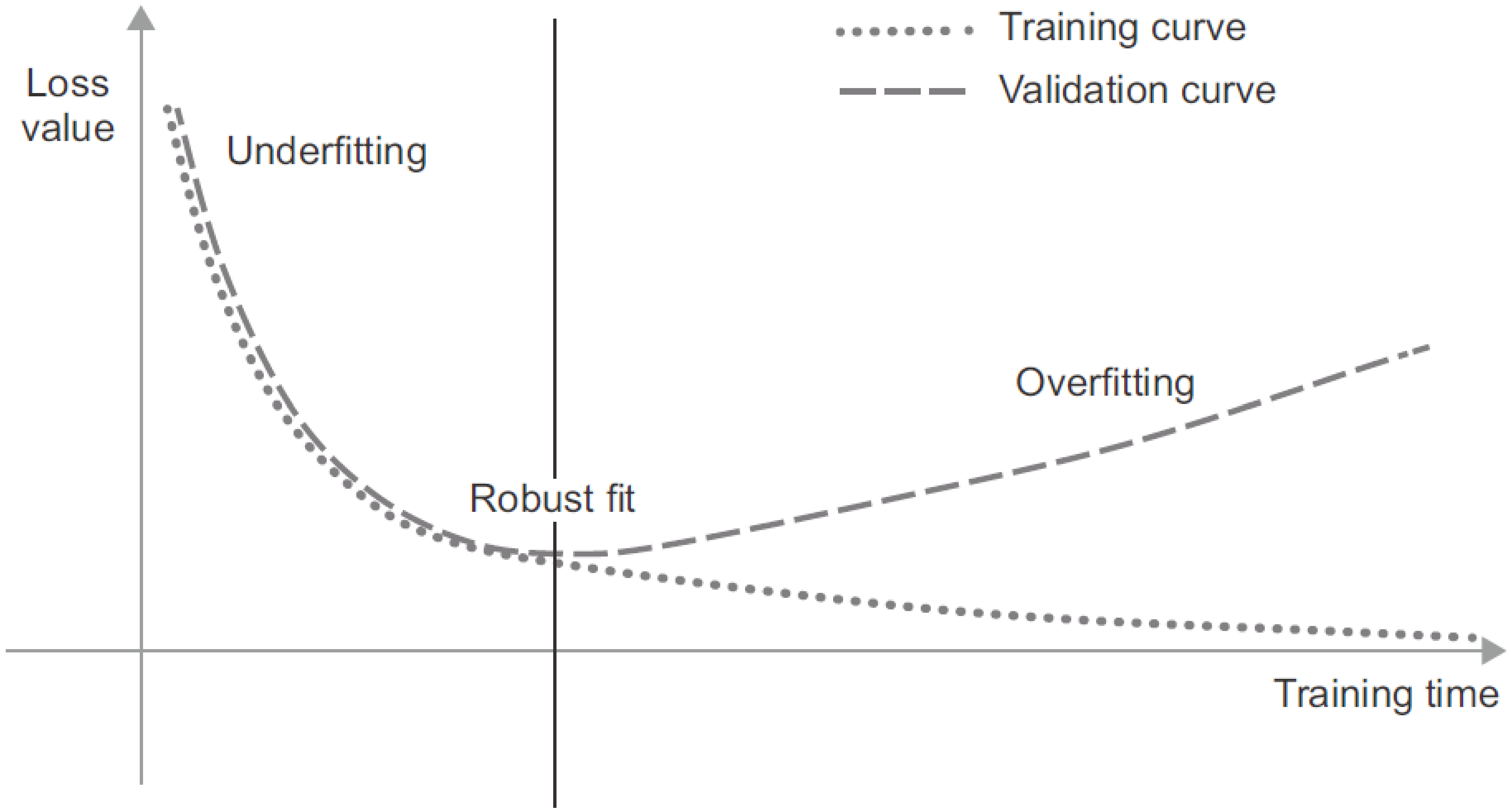
Chapter Summary

- You’ll usually need to preprocess raw data before feeding it into a neural network.
- When your data has features with different ranges, scale each feature independently as part of preprocessing.
- As training progresses, neural networks eventually begin to overfit and obtain worse results on never-before-seen data.
- If you don’t have much training data, use a small model with only one or two intermediate layers, to avoid severe overfitting.
- If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small.
- When you’re working with little data, K-fold validation can help reliably evaluate your model.

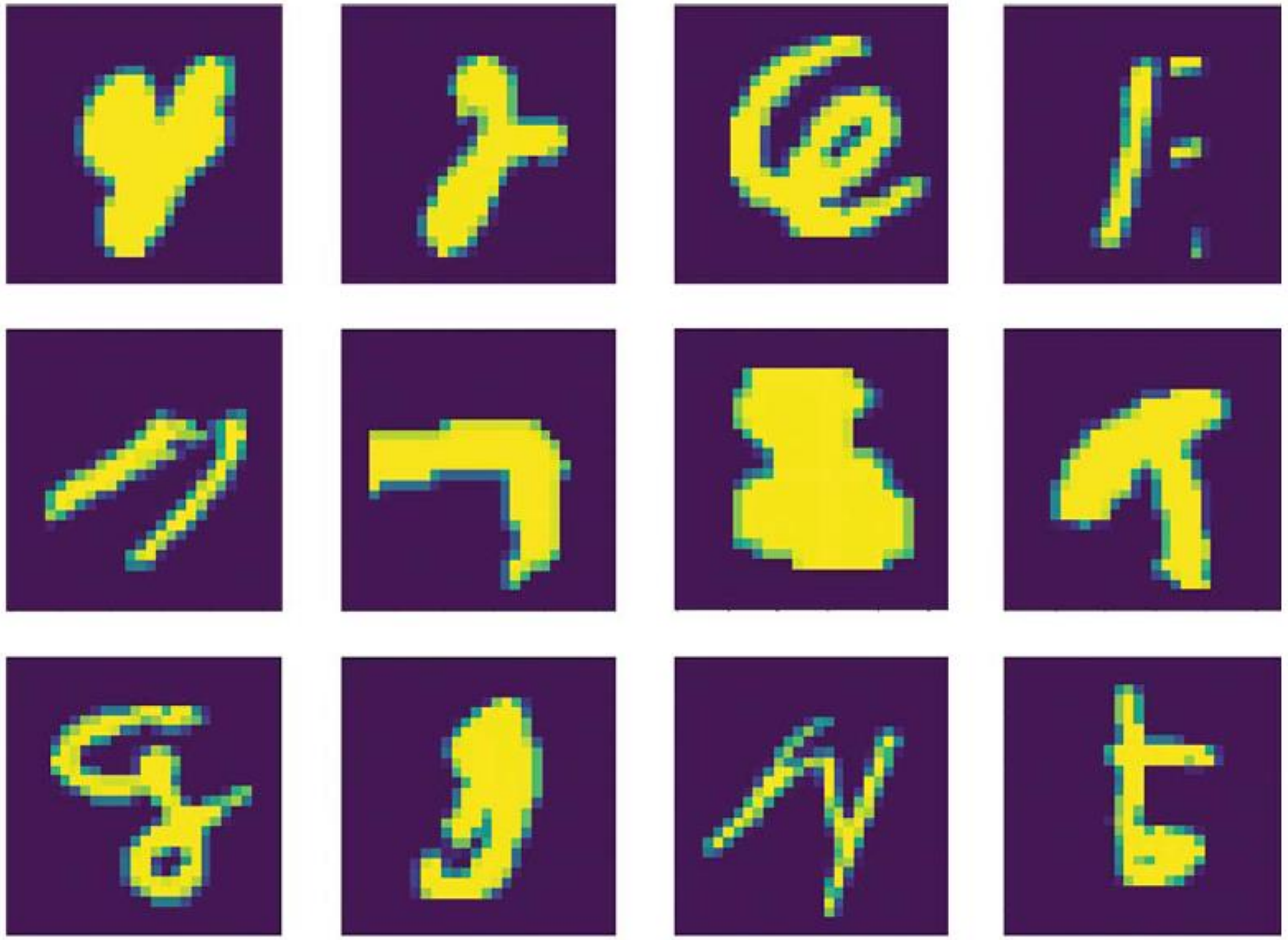
Fundamentals of Machine Learning



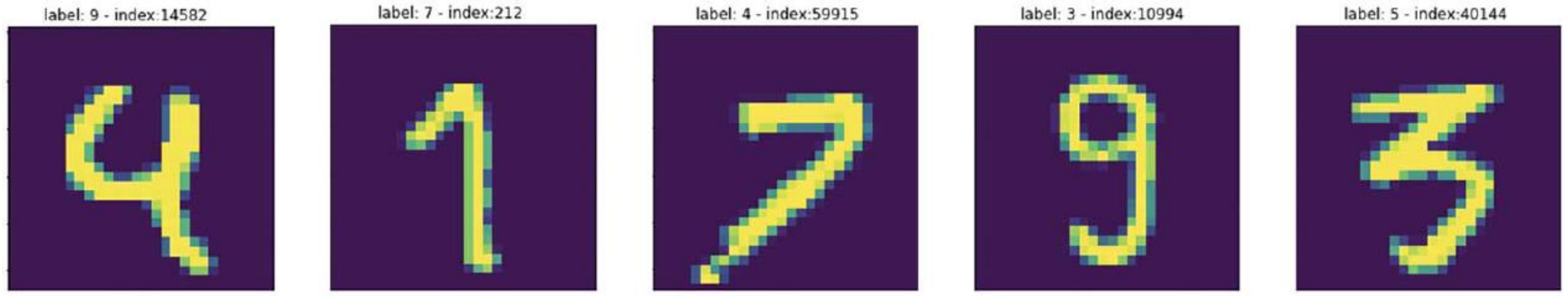
Canonical Overfitting Behavior



Noisy Data: Some Funky MNIST Digits

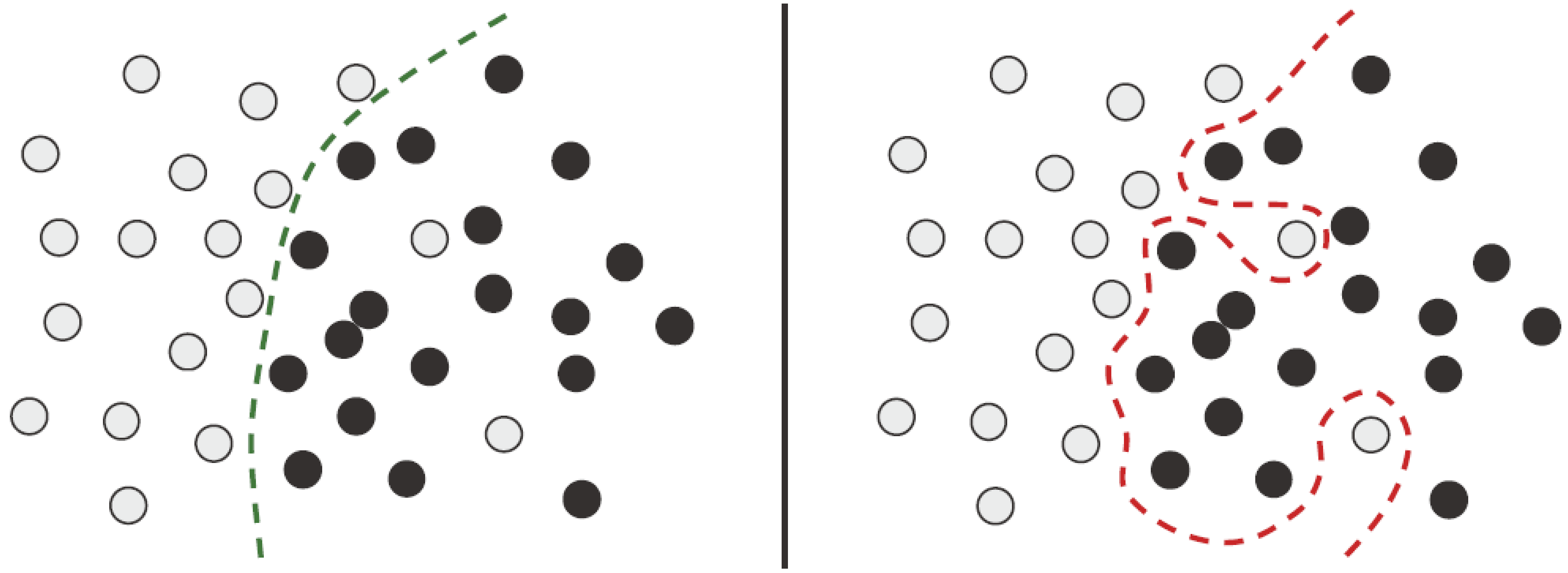


Noisy Labels

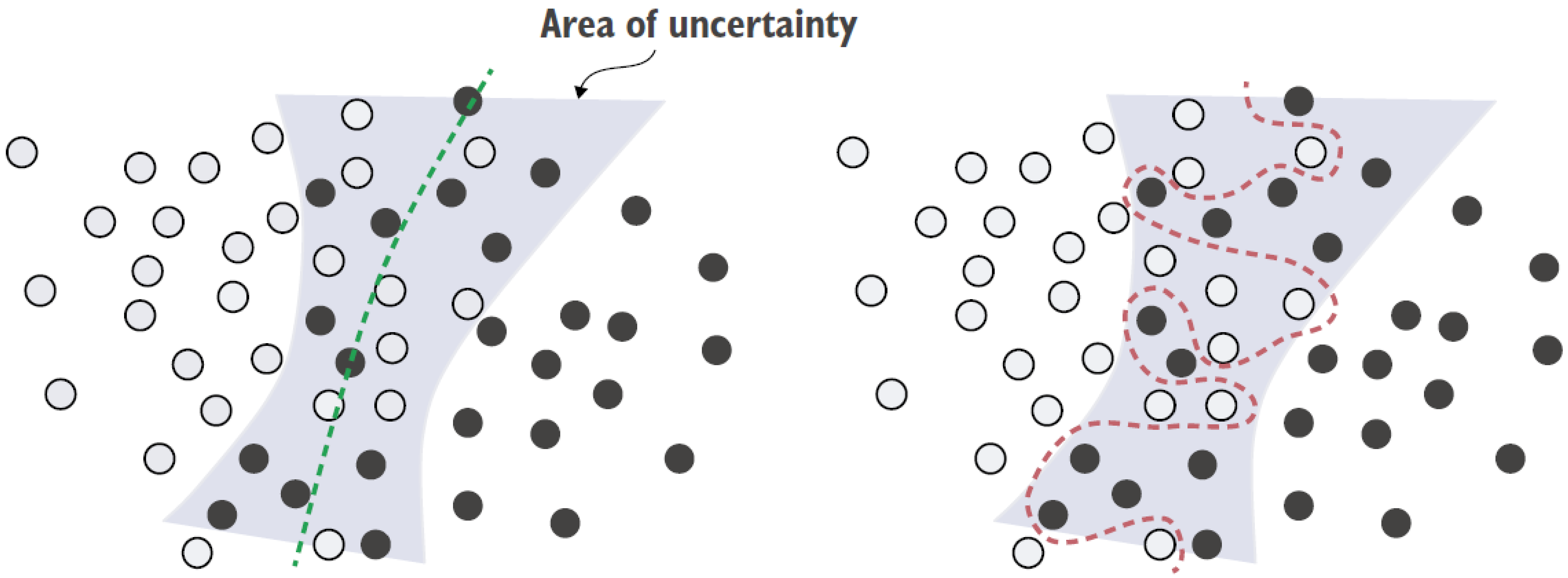


Label: 9 Label: 7 Label: 4 Label: 3 Label: 5

Dealing with Outliers: Robust Fit vs Over Fit



Class Overlap: Robust Fit vs Over Fit

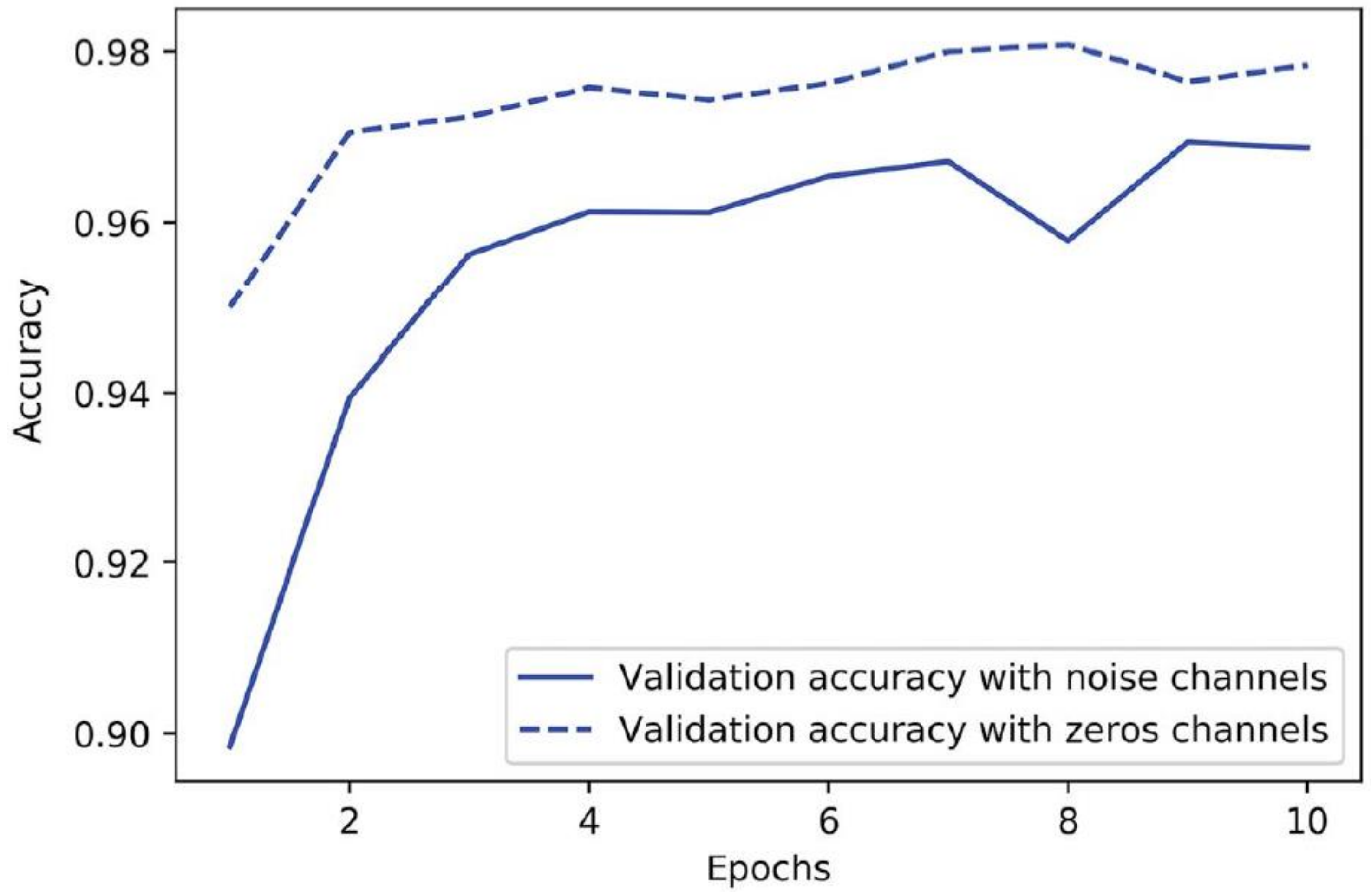


Adding Noise to MNIST

If I was into torture, this is how I would roll …

```
from tensorflow.keras.datasets import mnist
import numpy as np

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

train_images_with_noise_channels = np.concatenate(
    [train_images, np.random.random((len(train_images), 784))], axis=1)

train_images_with_zeros_channels = np.concatenate(
    [train_images, np.zeros((len(train_images), 784))], axis=1)
```

Effects of Noise on Validation Accuracy



Fitting a Model with Randomly Shuffled Labels

The point: if we're desperate enough, we can hallucinate that we're making progress [learn to recognize when stuff has gone off the rails] …

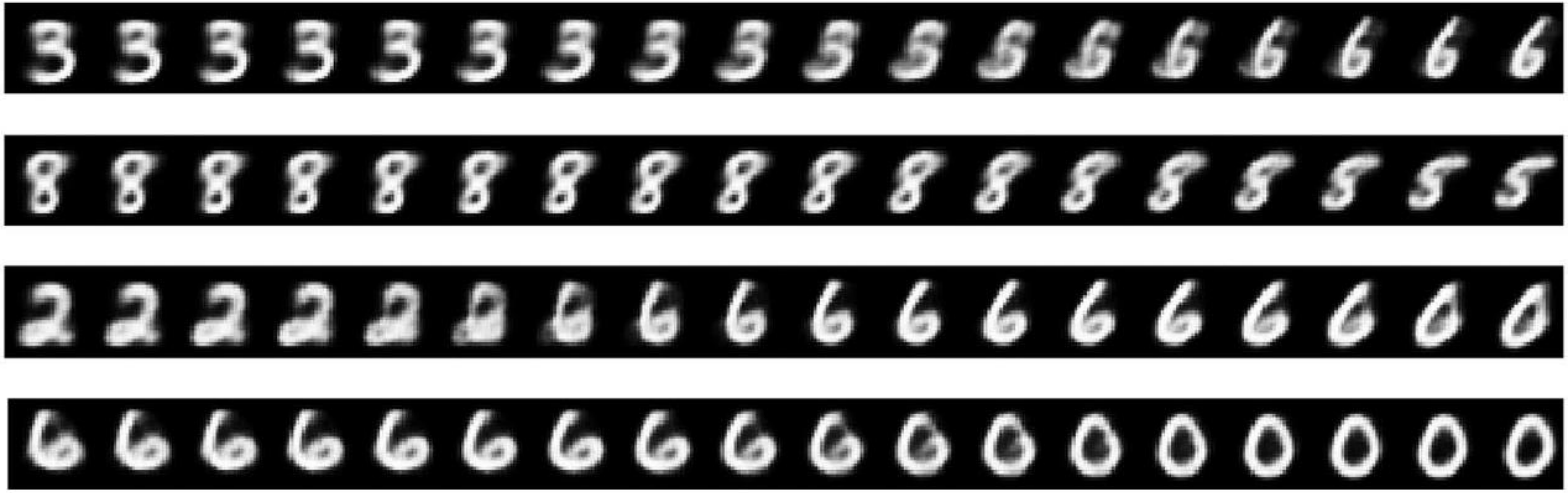
```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

random_train_labels = train_labels[:]
np.random.shuffle(random_train_labels)

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, random_train_labels,
          epochs=100,
          batch_size=128,
          validation_split=0.2)
```

The Manifold Hypothesis

- The manifold hypothesis posits that all natural data lies on a low-dimensional manifold within the high-dimensional space where it is encoded
- There is structure in the natural world; e.g. only a small fraction of the 256**784 possible MNIST "digits" are actually "likely"

Examples of Variational AutoEncoder Outputs

"showing that the space of handwritten digits forms a 'manifold' " …

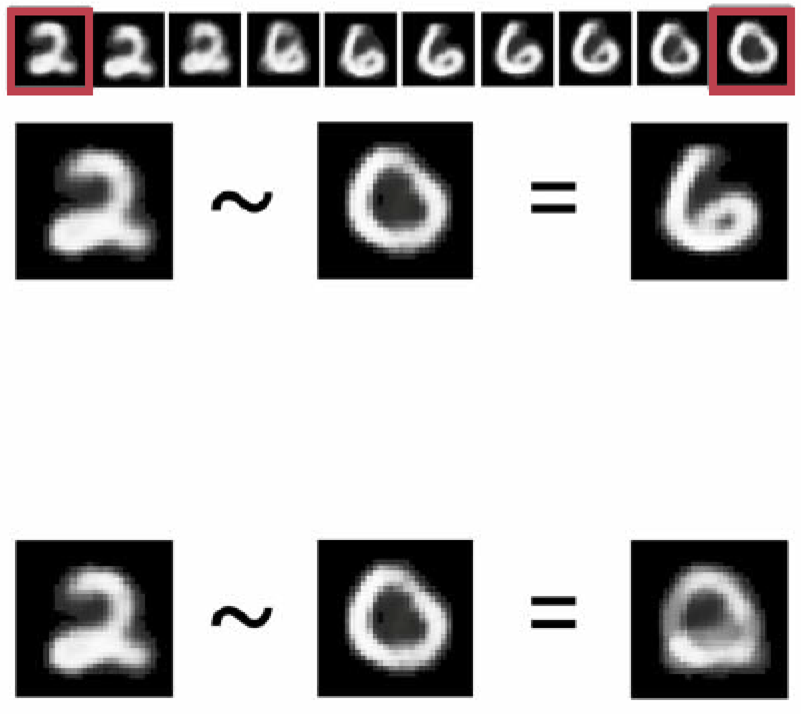


Implications

- Machine learning models only have to fit relatively simple, low-dimensional, highly structured subspaces within their potential input space (latent manifolds).
- Within one of these manifolds, it's always possible to interpolate between two inputs, that is to say, morph one into another via a continuous path along which all points fall on the manifold.
- "The ability to interpolate between samples is the key to understanding generalization in deep learning."

Manifold Interpolation

See Figure 12.18: we're moving from "left to right" along a single latent variable dimension

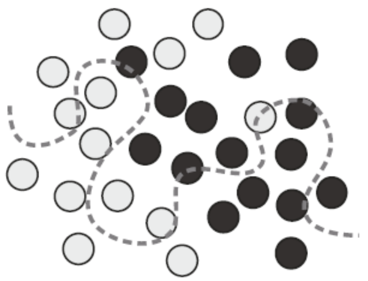


Manifold interpolation (intermediate point on the latent manifold)

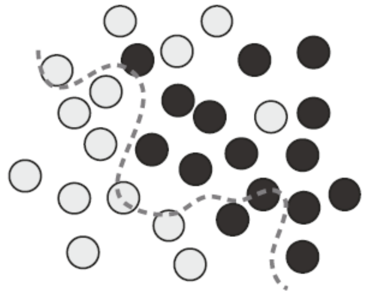
Linear interpolation (average in the encoding space)

Going from Random to Robust to Over Fit

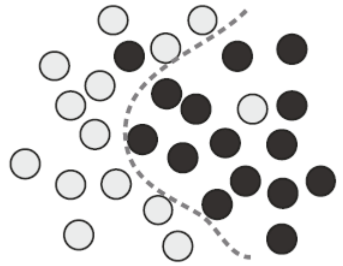
Before training: the model starts with a random initial state.



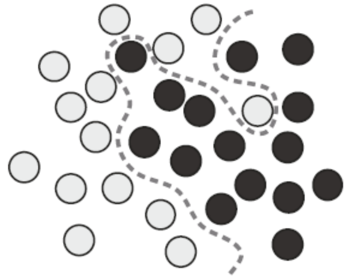
Beginning of training: the model gradually moves toward a better fit.



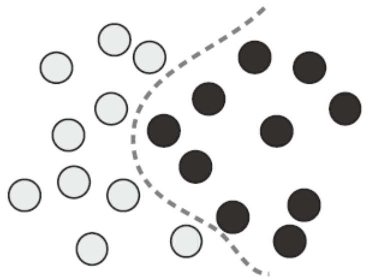
Further training: a robust fit is achieved, transitively, in the process of morphing the model from its initial state to its final state.



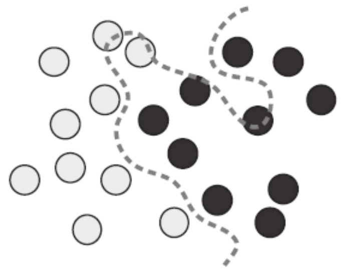
Final state: the model overfits the training data, reaching perfect training loss.



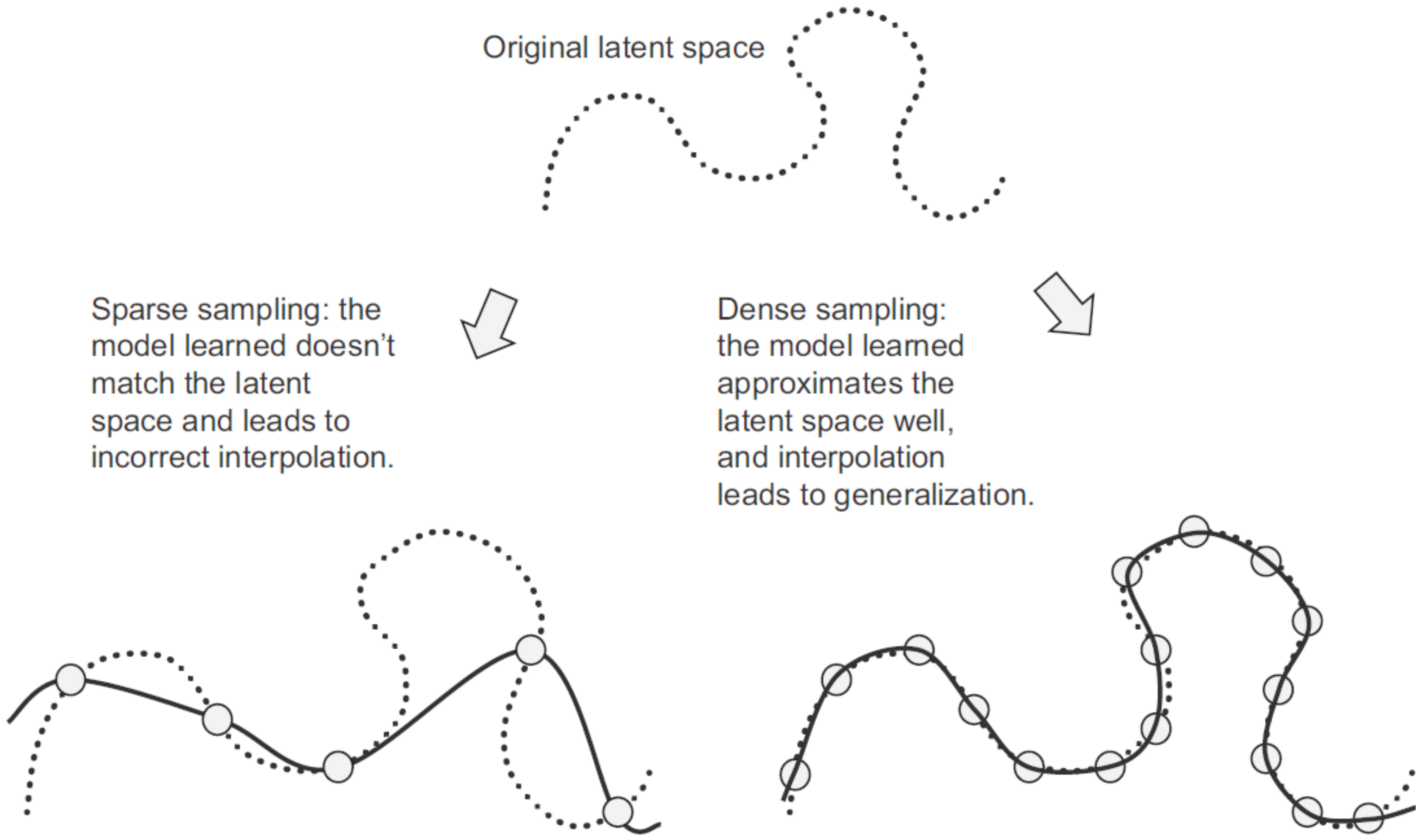
Test time: performance of robustly fit model on new data points



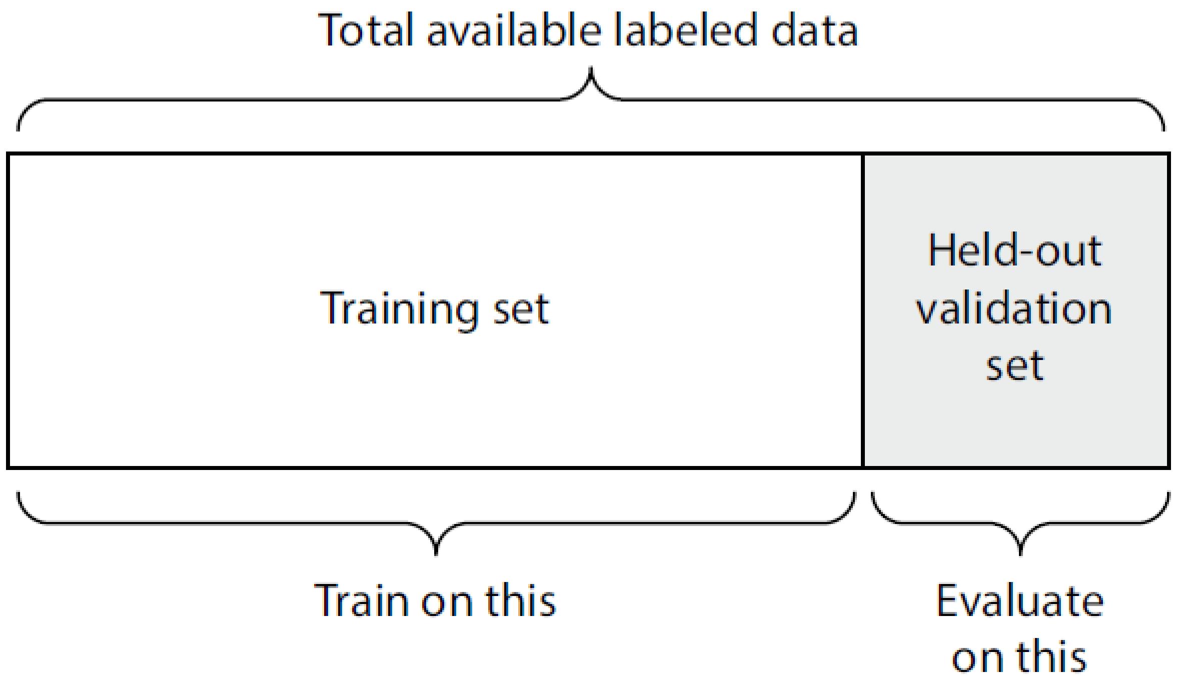
Test time: performance of overfit model on new data points



Dense vs Sparse Sampling

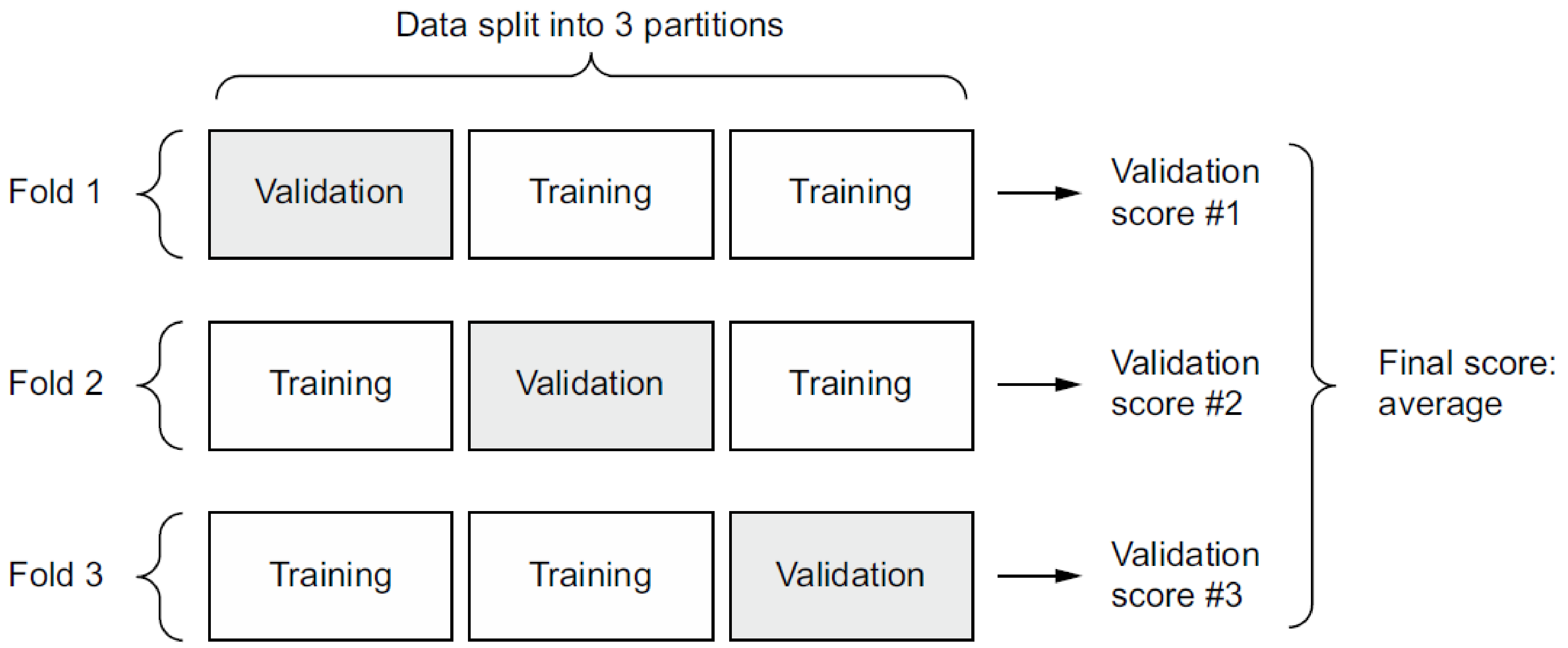

Original latent space
Sparse sampling: the model learned doesn't match the latent space and leads to incorrect interpolation.
Dense sampling: the model learned approximates the latent space well, and interpolation leads to generalization.

Hold-Out Validation [hold-out test too!]



K-Fold Cross Validation

Helpful when model performance shows significant variance based on your train/validation split



Iterated K-Fold Cross Validation

- For situations in which you have relatively little data available and you need to evaluate your model as precisely as possible
- Found it to be extremely helpful in Kaggle competitions
- You end up training and evaluating P * K models (where P is the number of iterations you use), which can be very expensive

Stuff to Keep in Mind

- Data representativeness: You want both your training set and test set to be representative of the data at hand. Training on MNIST digits 0-7 won't help with testing on digits 8-9 :)
- The arrow of time: If you’re trying to predict the future given the past (for example, tomorrow’s weather, stock movements, and so on), you should not randomly shuffle your data before splitting it, because doing so will create a temporal leak: your model will effectively be trained on data from the future.
- Redundancy in your data: If some data points in your data appear twice (fairly common with real-world data), then shuffling the data and splitting it into a training set and a validation set will result in redundancy between the training and validation sets. In effect, you’ll be testing on part of your training data.

3 Common Problems

- Training doesn't get started: your training loss doesn't go down over time.
- Training gets started just fine, but your model doesn't meaningfully generalize: you can't beat the common-sense baseline you set.
- Training and validation loss both go down over time, and you can beat your baseline, but you don't seem to be able to overfit, which indicates you're still underfitting.

Tuning Gradient Descent Parameters

Try high learning rate with MNIST just so you can recognize the behavior?

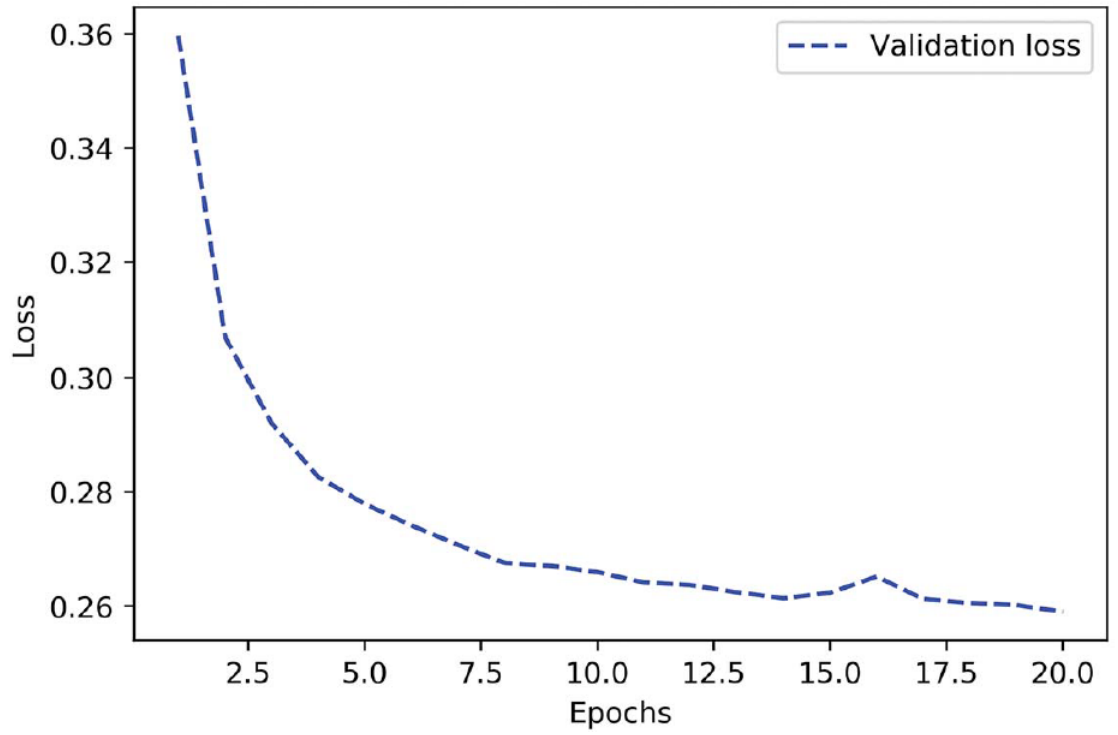
- Lowering or increasing the learning rate: a learning rate that is too high may lead to updates that vastly overshoot a proper fit, like in the preceding example, and a learning rate that is too low may make training so slow that it appears to stall.
- Increasing the batch size: a batch with more samples will lead to gradients that are more informative and less noisy (lower variance).

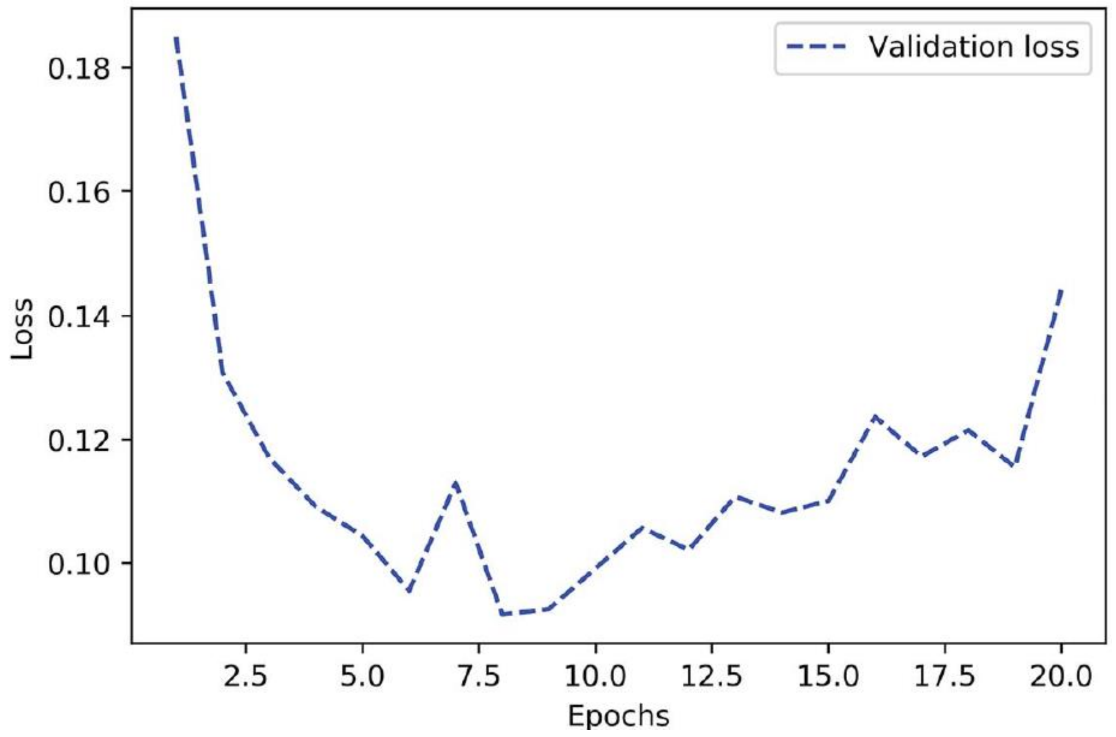
```
model.compile(optimizer=keras.optimizers.RMSprop(1.),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Better Architecture

- Using a model that makes the right assumptions about the problem is essential to achieve generalization: you should leverage the right architecture priors.
- In the following chapters, you'll learn about the best architectures to use for a variety of data modalities—images, text, timeseries, and so on. In general, you should always make sure to read up on architecture best practices for the kind of task you're attacking—chances are you're not the first person to attempt it.
- https://paperswithcode.com/sota

Effect of Insufficient Model Capacity



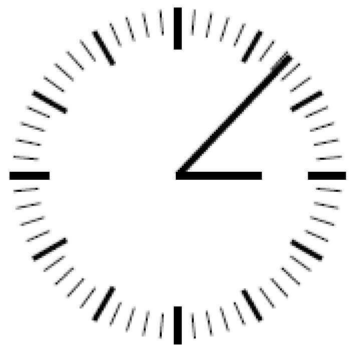
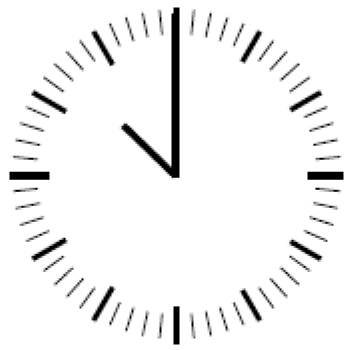


Which curve is demonstrating insufficient model capacity?

Dataset Curation

- Make sure you have enough data. Remember that you need a dense sampling of the input-cross-output space. More data will yield a better model. Sometimes, problems that seem impossible at first become solvable with a larger dataset.
- Minimize labeling errors—visualize your inputs to check for anomalies, and proofread your labels.
- Clean your data and deal with missing.
- If you have many features and you aren’t sure which ones are actually useful, do feature selection.

Feature Engineering

Raw data: pixel grid			Hard to learn
Better features: clock hands' coordinates	{x1: 0.7, y1: 0.7} {x2: 0.5, y2: 0.0}	{x1: 0.0, y2: 1.0} {x2: -0.38, y2: 0.32}	
Even better features: angles of clock hands	theta1: 45 theta2: 0	theta1: 90 theta2: 140	Easy to learn

Feature Engineering

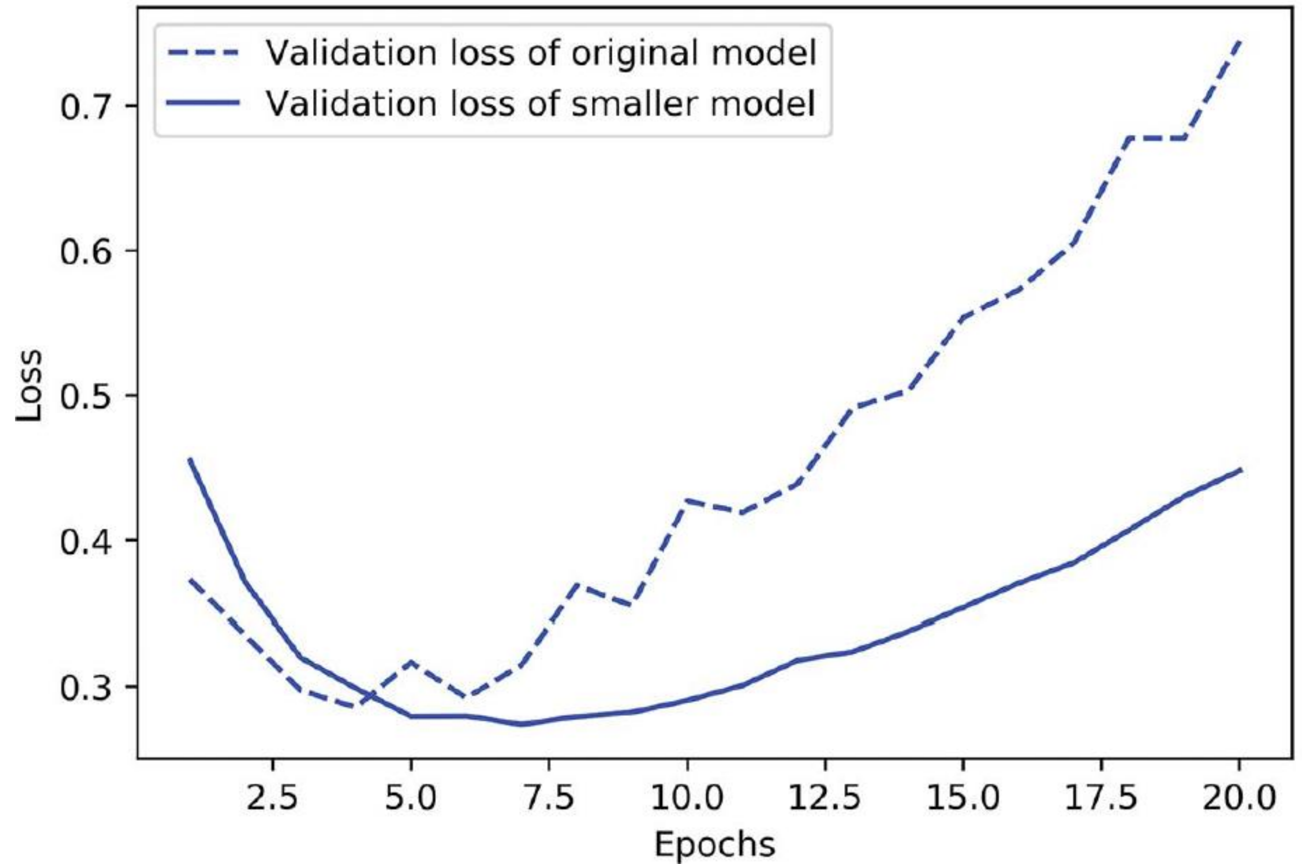
- Good features still allow you to solve problems more elegantly while using fewer resources. For instance, it would be ridiculous to solve the problem of reading a clock face using a convolutional neural network.
- Good features let you solve a problem with far less data. The ability of deep learning models to learn features on their own relies on having lots of training data available; if you have only a few samples, the information value in their features becomes critical.

Early Stopping

- In deep learning, we always use models that are vastly overparameterized: they have way more degrees of freedom than the minimum necessary to fit to the latent manifold of the data.
- This overparameterization is not an issue, because you never fully fit a deep learning model. Such a fit wouldn’t generalize at all.

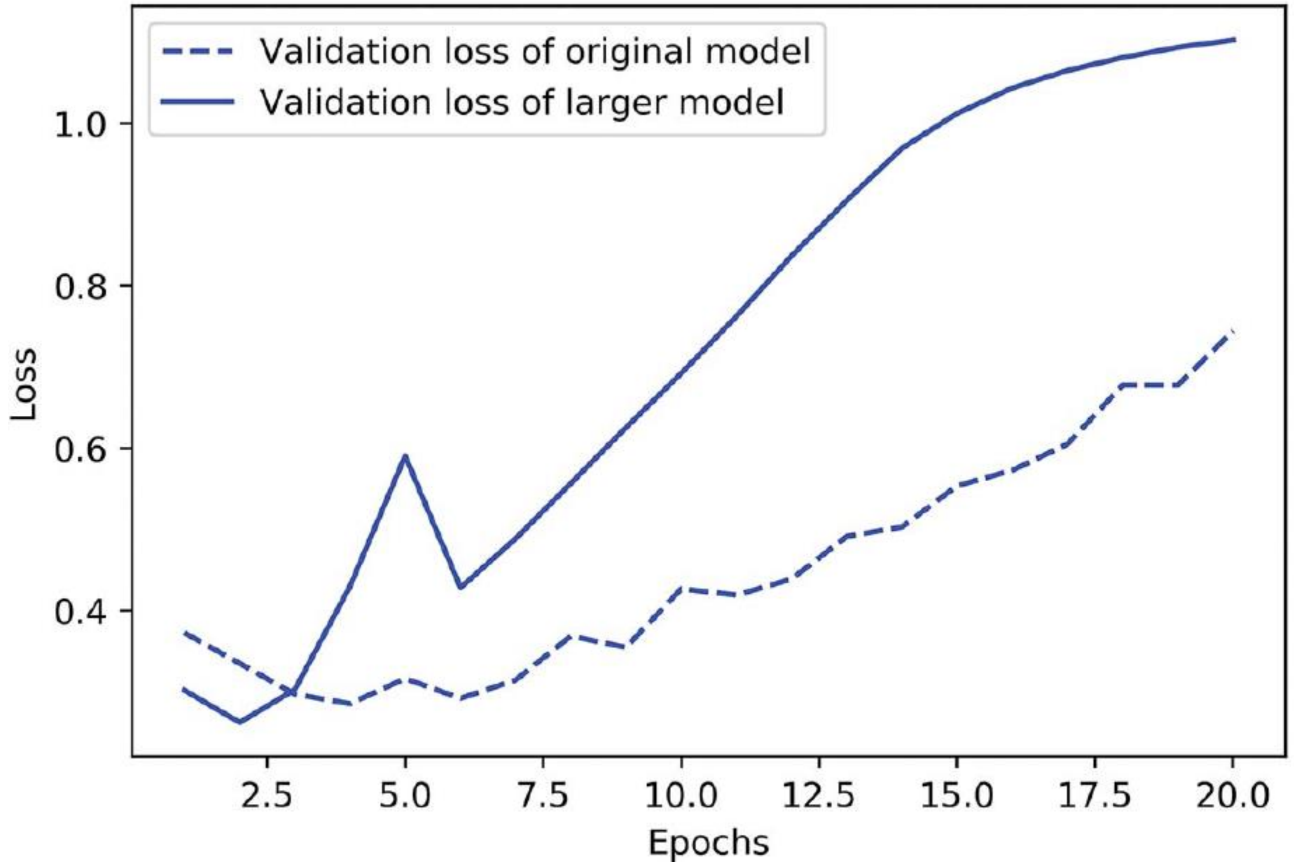
Model Regularization: Capacity

Example: Width 16 vs Width 4 Layers



Model Regularization: Capacity

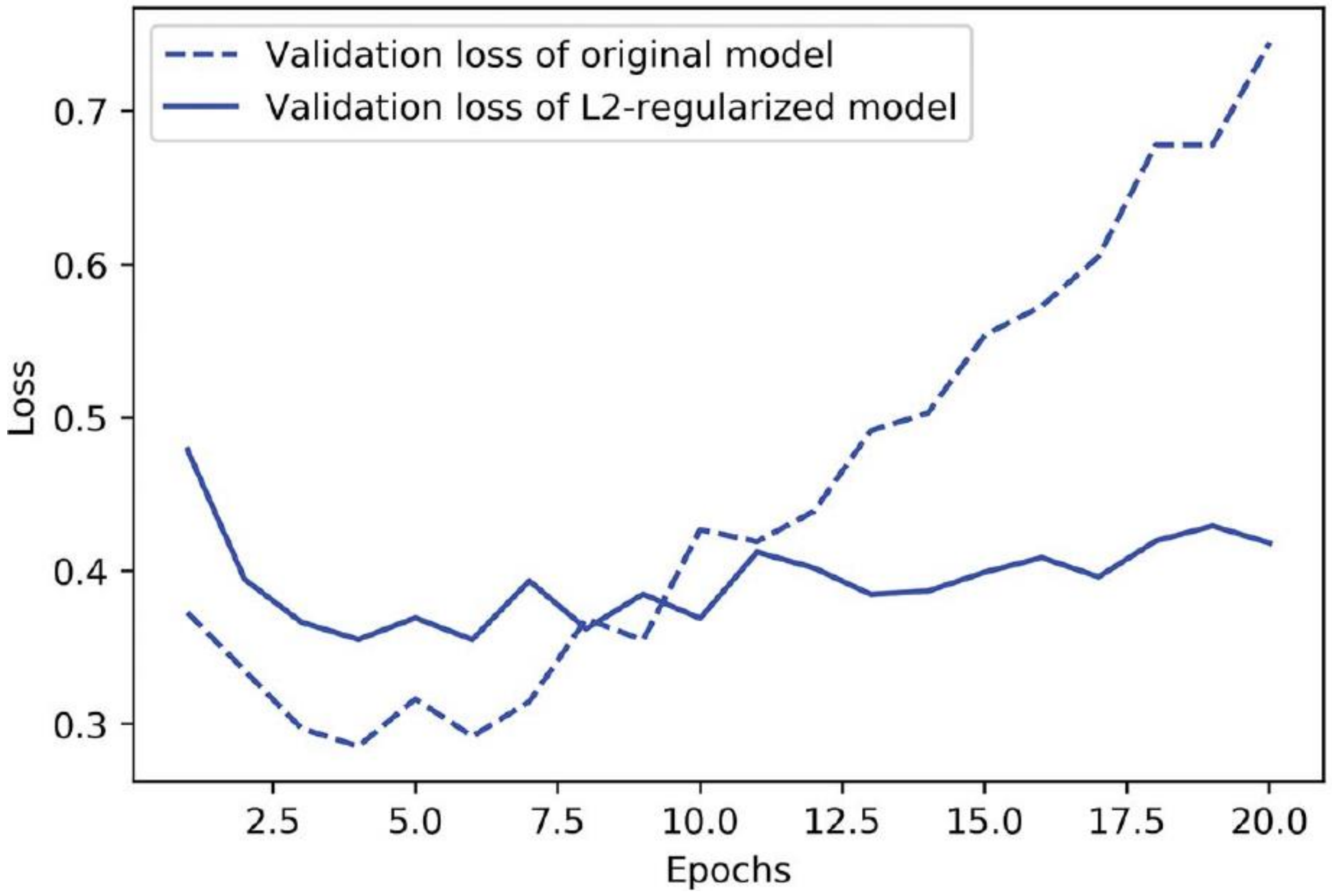
Example: Width 512 vs Width 16 Capacity



Weight Regularization

- 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.

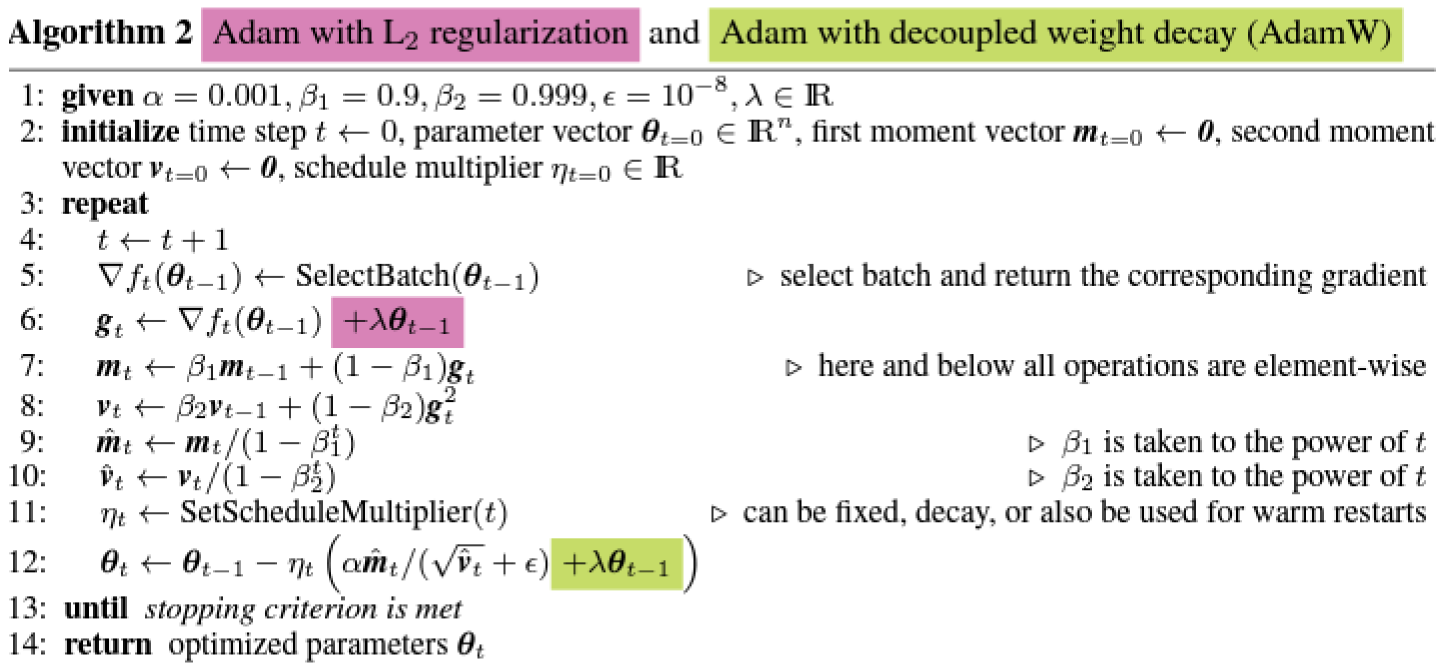
Weight Regularization



Weight Regularization

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/experimental/AdamW [keras.io]

https://arxiv.org/abs/1711.05101v3 [paperswithcode.com/methods]

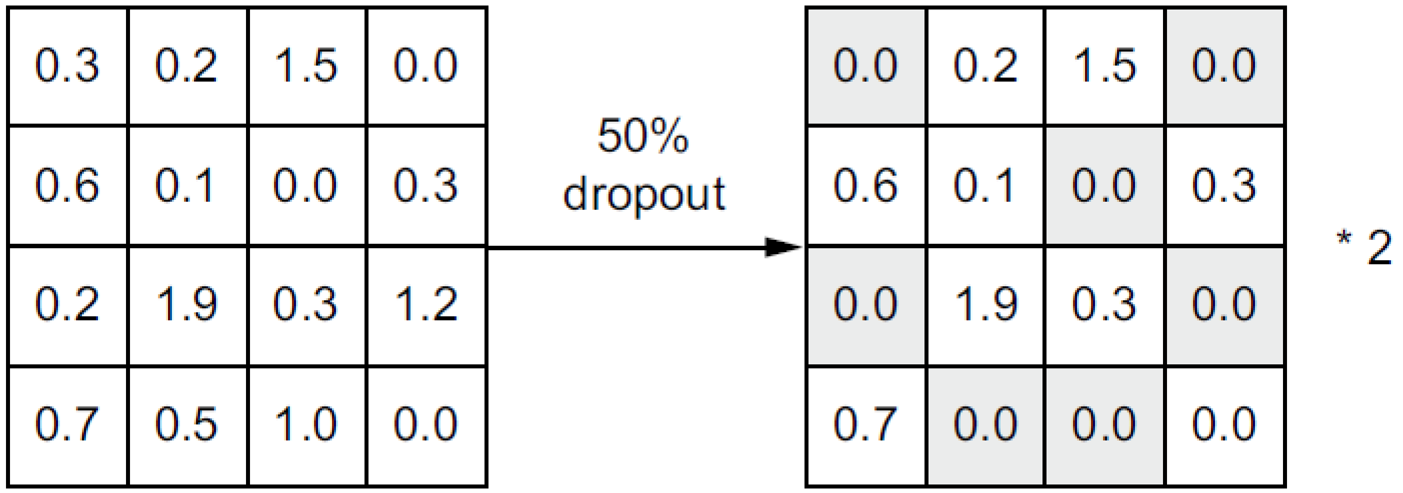


Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

```
1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment
   vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: repeat
4:     $t \leftarrow t + 1$
5:     $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow$ SelectBatch$(\boldsymbol{\theta}_{t-1})$    ▷ select batch and return the corresponding gradient
6:     $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$ $+\lambda\boldsymbol{\theta}_{t-1}$
7:     $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$    ▷ here and below all operations are element-wise
8:     $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9:     $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$    ▷ $\beta_1$ is taken to the power of $t$
10:    $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$    ▷ $\beta_2$ is taken to the power of $t$
11:    $\eta_t \leftarrow$ SetScheduleMultiplier$(t)$    ▷ can be fixed, decay, or also be used for warm restarts
12:    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left( \alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) +\lambda\boldsymbol{\theta}_{t-1} \right)$
13: until stopping criterion is met
14: return optimized parameters $\boldsymbol{\theta}_t$
```

Dropout

We randomly select x% of the activations of a layer and set their value to zero



0.3	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.2	1.9	0.3	1.2
0.7	0.5	1.0	0.0

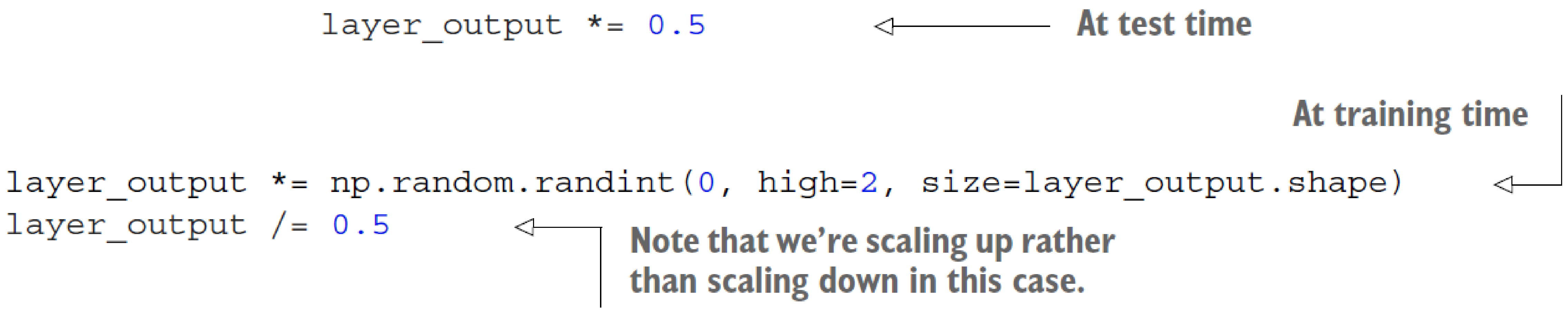
0.0	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.0	1.9	0.3	0.0
0.7	0.0	0.0	0.0

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
```

At training time, drops out 50% of the units in the output

Alternatives to Compensate for Dropout

- Dropout (random zeros) only happens at training time
- To keep the same expected value (between training and testing), we need to either
  - scale down the activations at testing time
  - scale up the activations at training time [this is common]
- The rate below is the “keep” rate, where keep_rate = 1 – dropout_rate



```
layer_output *= 0.5
```

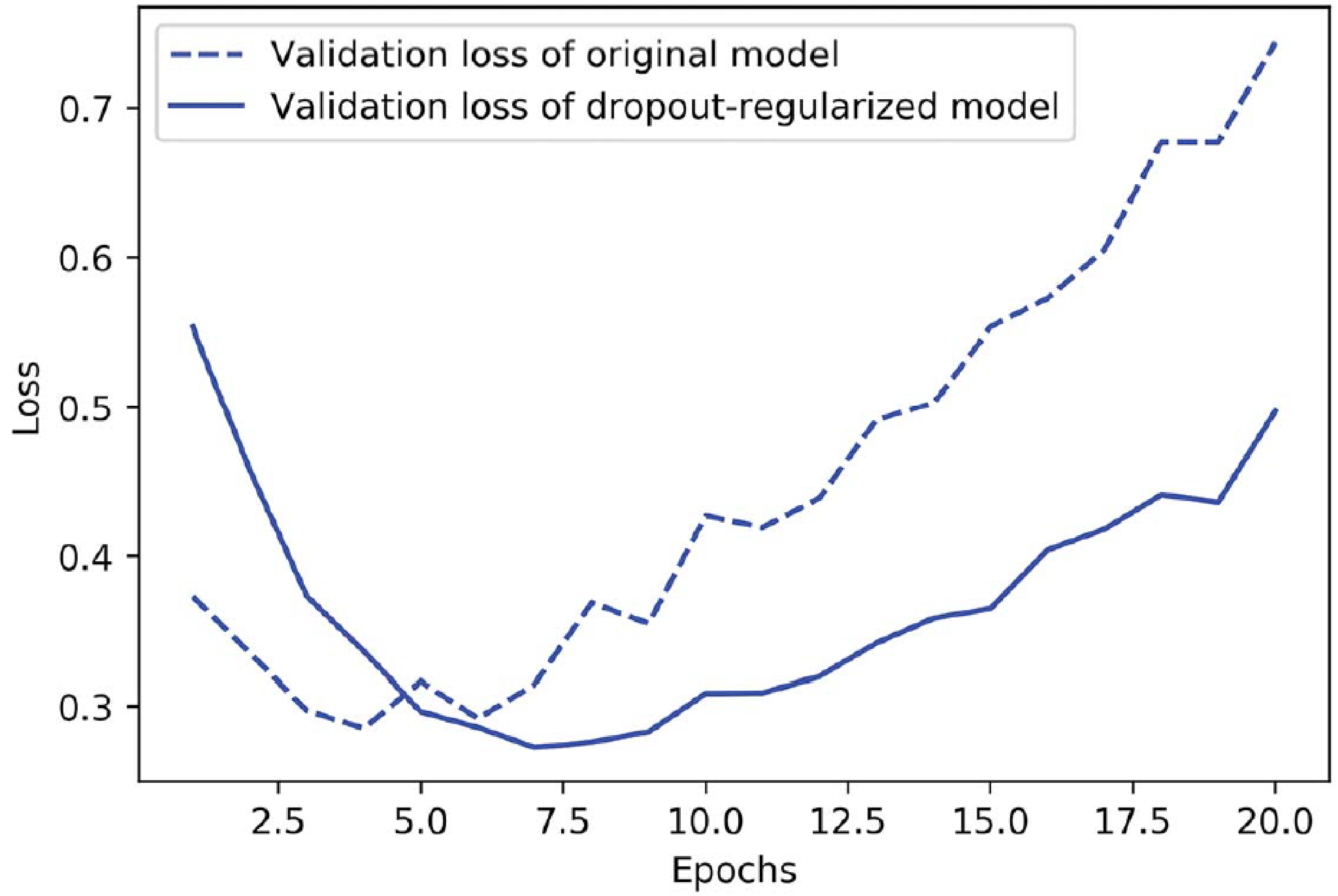
At test time

At training time

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
layer_output /= 0.5
```

Note that we’re scaling up rather than scaling down in this case.

Dropout Effect



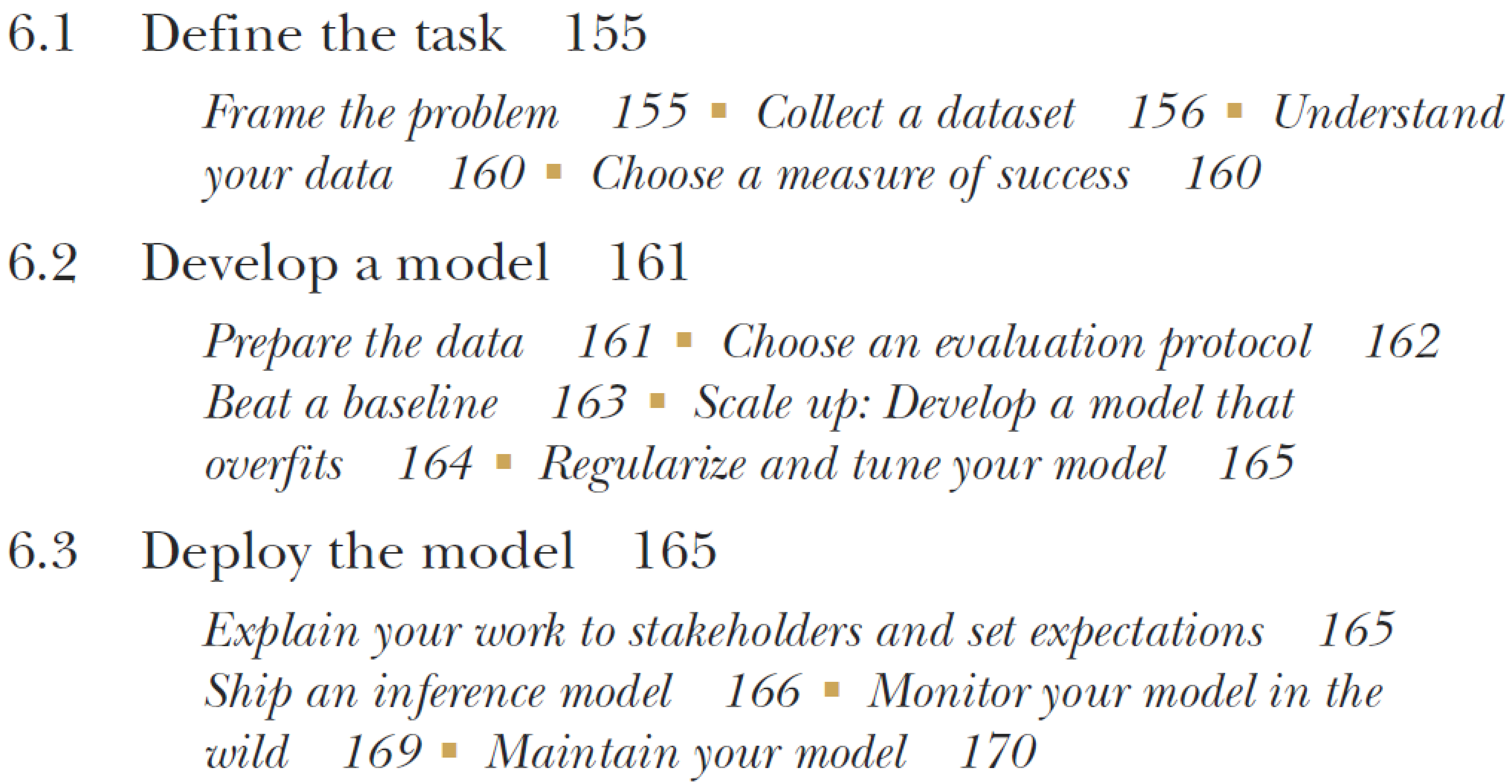
Tactics to Improve Generalization

- Get more training data, or better training data
- Develop better features
- Reduce the capacity of the model
- Add weight regularization (for smaller models)
- Add dropout

Summary

- The purpose of a machine learning model is to generalize: to perform accurately on never-before-seen inputs. It's harder than it seems.
- A deep neural network achieves generalization by learning a parametric model that can successfully interpolate between training samples—such a model can be said to have learned the "latent manifold" of the training data. This is why deep learning models can only make sense of inputs that are very close to what they've seen during training.
- The fundamental problem in machine learning is the tension between optimization and generalization: to attain generalization, you must first achieve a good fit to the training data, but improving your model's fit to the training data will inevitably start hurting generalization after a while. Every single deep learning best practice deals with managing this tension.
- The ability of deep learning models to generalize comes from the fact that they manage to learn to approximate the latent manifold of their data, and can thus make sense of new inputs via interpolation.
- It's essential to be able to accurately evaluate the generalization power of your model while you're developing it. You have at your disposal an array of evaluation methods, from simple holdout validation to K-fold cross-validation and iterated K-fold cross-validation with shuffling. Remember to always keep a completely separate test set for final model evaluation, since information leaks from your validation data to your model may have occurred.
- When you start working on a model, your goal is first to achieve a model that has some generalization power and that can overfit. Best practices for doing this include tuning your learning rate and batch size, leveraging better architecture priors, increasing model capacity, or simply training longer.
- As your model starts overfitting, your goal switches to improving generalization through model regularization. You can reduce your model's capacity, add dropout or weight regularization, and use early stopping. And naturally, a larger or better dataset is always the number one way to help a model generalize.

The Universal Workflow of Machine Learning



Machine Learning Consulting Projects

- A personalized photo search engine for a picture-sharing social network: type in "wedding" and retrieve all the pictures you took at weddings, without any manual tagging needed.
- Flagging spam and offensive text content among the posts of a budding chat app
- Building a music recommendation system for users of an online radio
- Detecting credit card fraud for an e-commerce website
- Predicting display ad click-through rate to decide which ad to serve to a given user at a given time
- Flagging anomalous cookies on the conveyor belt of a cookie-manufacturing line
- Using satellite images to predict the location of as-yet unknown archeological sites

Note on Ethics

- You may sometimes be offered ethically dubious projects, such as “building an AI that rates the trustworthiness of someone from a picture of their face”
- It isn’t clear why trustworthiness would be reflected on someone’s face
- Collecting a dataset for this task would amount to recording the biases and prejudices of the people who label the pictures
- The models you would train on such data would merely encode these same biases into a black-box algorithm that would give them a thin veneer of legitimacy
- Your model would be laundering and operationalizing at scale the worst aspects of human judgement, with negative effects on the lives of real people

The Universal Workflow of Machine Learning

- Define the task
- Develop a model
- Deploy the model

Define the Task

- Frame the problem
- Collect a dataset
- Understand your data
- Choose a measure of success

Frame the Problem

- What are you trying to predict? What will your input data be?
- What type of machine learning task are you facing?
- What do existing solutions look like?
- Are there any particular constraints you will need to deal with?

Machine Learning Tasks

- The photo search engine project is a multiclass, multilabel classification task.
- The spam detection project is a binary classification task. If you set “offensive content” as a separate class, it’s a three-way classification task.
- The music recommendation engine turns out to be better handled not via deep learning, but via matrix factorization (collaborative filtering).
- The credit card fraud detection project is a binary classification task.
- The click-through-rate prediction project is a scalar regression task.
- Anomalous cookie detection is a binary classification task, but it will also require an object detection model as a first stage in order to correctly crop out the cookies in raw images. Note that the set of machine learning techniques known as “anomaly detection” would not be a good fit in this setting!
- The project for finding new archeological sites from satellite images is an image-similarity ranking task: you need to retrieve new images that look the most like known archeological sites.

Collect a Dataset

- The photo search engine project requires you to first select the set of labels you want to classify—you settle on 10,000 common image categories. Then you need to manually tag hundreds of thousands of your past user-uploaded images with labels from this set.
- For the chat app's spam detection project, because user chats are end-to-end encrypted, you cannot use their contents for training a model. You need to gain access to a separate dataset of tens of thousands of unfiltered social media posts, and manually tag them as spam, offensive, or acceptable.
- For the music recommendation engine, you can just use the "likes" of your users. No new data needs to be collected. Likewise for the click-through-rate prediction project: you have an extensive record of click-through rate for your past ads, going back years.
- For the cookie-flagging model, you will need to install cameras above the conveyor belts to collect tens of thousands of images, and then someone will need to manually label these images. The people who know how to do this currently work at the cookie factory, but it doesn't seem too difficult. You should be able to train people to do it.
- The satellite imagery project will require a team of archeologists to collect a database of existing sites of interest, and for each site you will need to find existing satellite images taken in different weather conditions. To get a good model, you're going to need thousands of different sites.

Investing in Annotation Infrastructure

- Should you annotate the data yourself?
- Should you use a crowdsourcing platform like Mechanical Turk to collect labels?
- Should you use the services of a specialized data-labeling company?

Annotation Constraints

- Do the data labelers need to be subject matter experts, or could anyone annotate the data?
- If annotating the data requires specialized knowledge, can you train people to do it?
- Do you, yourself, understand the way experts come up with the annotations?

Beware of Non-Representative Data

- Suppose you’re developing an app where users can take pictures of a plate of food to find out the name of the dish.
- Come deployment time, feedback from angry users starts rolling in: your app gets the answer wrong 8 times out of 10. What’s going on? Your accuracy on the test set was well over 90%!
- A quick look at user-uploaded data reveals that mobile picture uploads of random dishes from random restaurants taken with random smartphones look nothing like the professional-quality, well-lit, appetizing pictures you trained the model on: your training data wasn’t representative of the production data.

https://www.gwern.net/Tanks

Concept Drift

- A music recommendation engine trained in the year 2013 may not be very effective today
- Likewise, the IMDB dataset you worked with was collected in 2011, and a model trained on it would likely not perform as well on reviews from 2020 compared to reviews from 2012, as vocabulary, expressions, and movie genres evolve over time
- Concept drift is particularly acute in adversarial contexts like credit card fraud detection, where fraud patterns change practically every day
- Dealing with fast concept drift requires constant data collection, annotation, and model retraining

The Problem of Sampling Bias

- The editor of the Tribune had trusted the results of a phone survey—but phone users in 1948 were not a random, representative sample of the voting population
- They were more likely to be richer, conservative, and to vote for Dewey, the Republican candidate

Understand Your Data

- If your data includes images or natural language text, take a look at a few samples (and their labels) directly.
- If your data contains numerical features, it's a good idea to plot the histogram of feature values to get a feel for the range of values taken and the frequency of different values.
- If your data includes location information, plot it on a map. Do any clear patterns emerge?
- Are some samples missing values for some features? If so, you'll need to deal with this when you prepare the data (we'll cover how to do this in the next section).
- If your task is a classification problem, print the number of instances of each class in your data. Are the classes roughly equally represented? If not, you will need to account for this imbalance.
- Check for target leaking: the presence of features in your data that provide information about the targets and which may not be available in production. If you're training a model on medical records to predict whether someone will be treated for cancer in the future, and the records include the feature "this person has been diagnosed with cancer," then your targets are being artificially leaked into your data. Always ask yourself, is every feature in your data something that will be available in the same form in production?

Choose a Measure of Success

- To achieve success on a project, you must first define what you mean by success
    - Accuracy?
    - Precision and recall?
    - Customer retention rate?
- Your metric for success will guide all of the technical choices you make throughout the project
- It should directly align with your higher-level goals, such as the business success of your customer

Develop a Model

- Prepare the data
- Choose an evaluation protocol
- Beat a baseline
- Scale up: develop a model that overfits
- Regularize and tune your model

Prepare the Data

- Prepare the data
  - Vectorization
  - Value normalization
    - Values should be small; e.g. in the interval [0, 1]
    - Values should be homogeneous: roughly the same range
  - Handling missing values
    - Categorical: create a "value is missing" category
    - Numerical: consider using the mean or median value for the feature (or predict it)

Choose an Evaluation Protocol

- Validation data is used for hyperparameter optimization
  - Hold-out validation
  - K-fold cross-validation
    - Too few folds (e.g. k = 2) may yield high bias
    - Too many folds (e.g. k = n) may yield high variance
  - Iterated k-fold cross-validation
- You’ll also need hold-out testing data

Beating a Simple Baseline

- Feature engineering/selection
- Select an architecture (e.g. dense, convolutional, recurrent, transformer)
- Configure training (e.g. loss, optimizer)

Picking the Right Loss Function

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`

****

Scale Up: Develop a Model that Overfits

- Add layers
- Make the layers bigger
- Train for more epochs

****

Regularize and Tune Your Model

- Try different architectures; add or remove layers
- Add dropout
- If your model is small, add L1 or L2 regularization
- Try different hyperparameters (such as the number of units per layer or the learning rate of the optimizer) to find the optimal configuration
- Optionally, iterate on data curation or feature engineering: collect and annotate more data, develop better features, or remove features that don’t seem to be informative
- Once you’ve selected a satisfactory model configuration, you can train your final production model on all available data and evaluate it one last time on the test set

Deploy the Model

- Explain your work to stakeholders and set expectations
- Ship an inference model
- Monitor your model in the wild
- Maintain your model

Explain Your Work to Stakeholders and Set Expectations

- The expectations of non-specialists towards AI systems are often unrealistic.
- Consider showing some examples of the failures; e.g. false positives and false negatives.
- “With these settings, the fraud detection model would have a 5% false negative rate and a 2.5% false positive rate. Every day, an average of 200 valid transactions would be flagged as fraudulent and sent for manual review, and an average of 14 fraudulent transactions would be missed. An average of 266 fraudulent transactions would be correctly caught.”

Deploying a Model as a REpresentational State Transfer (REST) API

Use this when …

- The application that will consume the model’s prediction will have reliable access to the internet (obviously)
- The application does not have strict latency requirements: the request, inference, and answer round trip will typically take around 500 ms
- The input data sent for inference is not highly sensitive: the data will need to be available on the server in a decrypted form, since it will need to be seen by the model (but note that you should use SSL encryption for the HTTP request and answer)

Deploying a Model on a Device

Use this when …

- Your model has strict latency constraints or needs to run in a low-connectivity environment
- Your model can be made sufficiently small that it can run under the memory and power constraints of the target device
- Getting the highest possible accuracy isn’t mission critical for your task
- The input data is strictly sensitive and thus shouldn’t be decryptable on a remote server

Deploying a Model in the Browser

Use this when …

- You want to offload compute to the end user, which can dramatically reduce server costs.
- The input data needs to stay on the end user's computer or phone. For instance, in our spam detection project, the web version and the desktop version of the chat app (implemented as a cross-platform app written in JavaScript) should use a locally run model.
- Your application has strict latency constraints. While a model running on the end user's laptop or smartphone is likely to be slower than one running on a large GPU on your own server, you don't have the extra 100 ms of network round trip.
- You need your app to keep working without connectivity, after the model has been downloaded and cached.

Inference Model Optimization

- Weight pruning: Not every coefficient in a weight tensor contributes equally to the predictions. It’s possible to considerably lower the number of parameters in the layers of your model by only keeping the most significant ones. This reduces the memory and compute footprint of your model, at a small cost in performance metrics. By deciding how much pruning you want to apply, you are in control of the trade-off between size and accuracy.
- Weight quantization: Deep learning models are trained with single-precision floating-point (float32) weights. However, it’s possible to quantize weights to 8-bit signed integers (int8) to get an inference-only model that’s a quarter the size but remains near the accuracy of the original model.

Monitor Your Model in the Wild

- Measure the effect
  - Is user engagement in your online radio up or down after deploying the new music recommender system?
  - Has the average ad click-through rate increased after switching to the new click-through-rate prediction model?
  - Consider using randomized A/B testing to isolate the impact of the model itself from other changes: a subset of cases should go through the new model, while another control subset should stick to the old process. Once sufficiently many cases have been processed, the difference in outcomes between the two is likely attributable to the model.
- If possible, do a regular manual audit of the model’s predictions on production data. It’s generally possible to reuse the same infrastructure as for data annotation: send some fraction of the production data to be manually annotated, and compare the model’s predictions to the new annotations. For instance, you should definitely do this for the image search engine and the bad-cookie flagging system.
- When manual audits are impossible, consider alternative evaluation avenues such as user surveys (for example, in the case of the spam and offensive-content flagging system).

Maintain Your Model

- Watch out for changes in the production data. Are new features becoming available? Should you expand or otherwise edit the label set?
- Keep collecting and annotating data, and keep improving your annotation pipeline over time. In particular, you should pay special attention to collecting samples that seem to be difficult for your current model to classify—such samples are the most likely to help improve performance.

Summary: Define the Problem

- Understand the broader context of what you’re setting out to do—what’s the end goal and what are the constraints?
- Collect and annotate a dataset; make sure you understand your data in depth.
- Choose how you’ll measure success for your problem—what metrics will you monitor on your validation data?

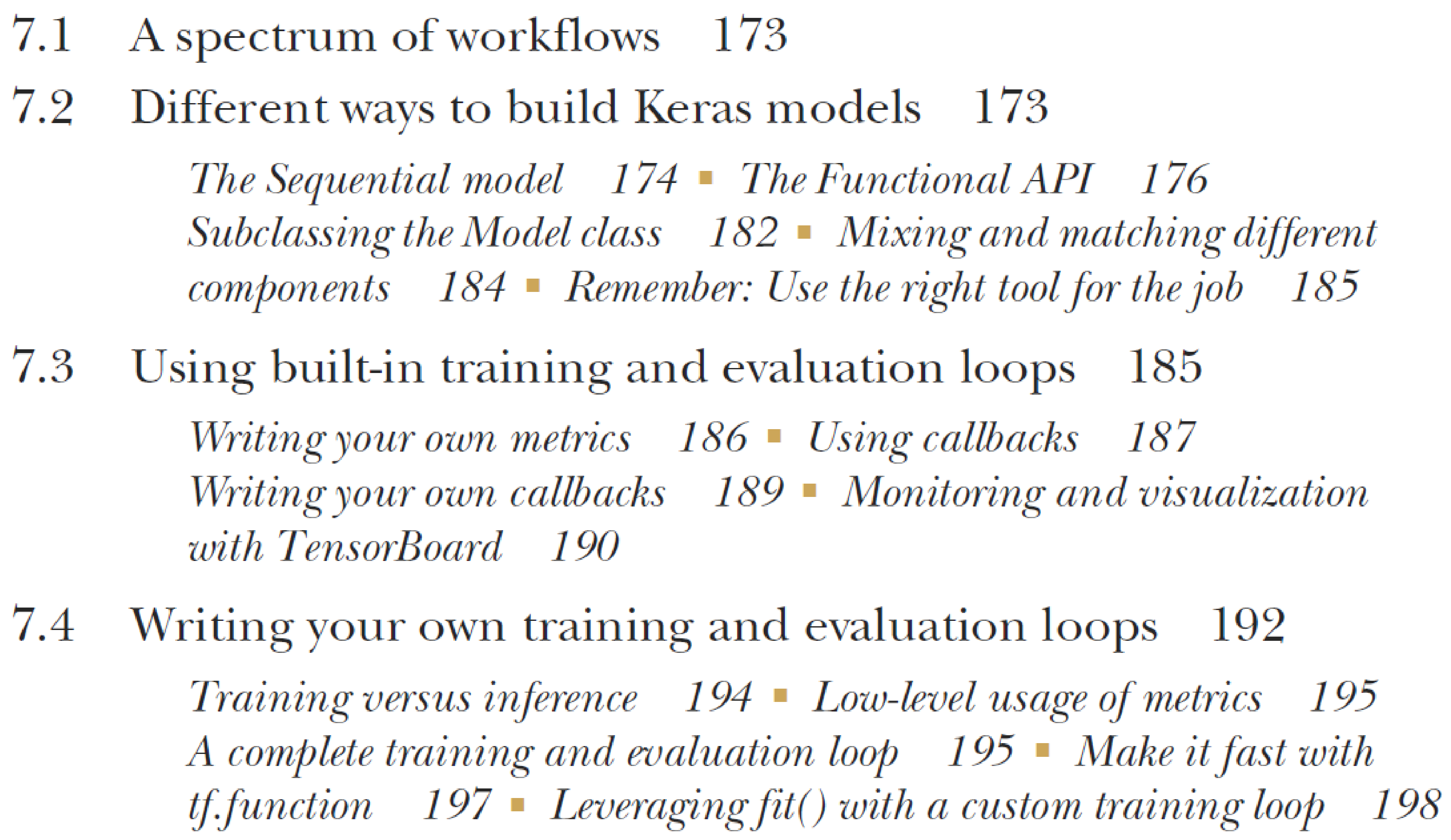
Summary: Develop Your Model

- Prepare your data.
- Pick your evaluation protocol: holdout validation? K-fold validation? Which portion of the data should you use for validation?
- Achieve statistical power: beat a simple baseline.
- Scale up: develop a model that can overfit.
- Regularize your model and tune its hyperparameters, based on performance on the validation data. A lot of machine learning research tends to focus only on this step, but keep the big picture in mind.

Summary: Deployment

- First, make sure you set appropriate expectations with stakeholders.
- Optimize a final model for inference, and ship a model to the deployment environment of choice—web server, mobile, browser, embedded device, etc.
- Monitor your model’s performance in production, and keep collecting data so you can develop the next generation of the model.

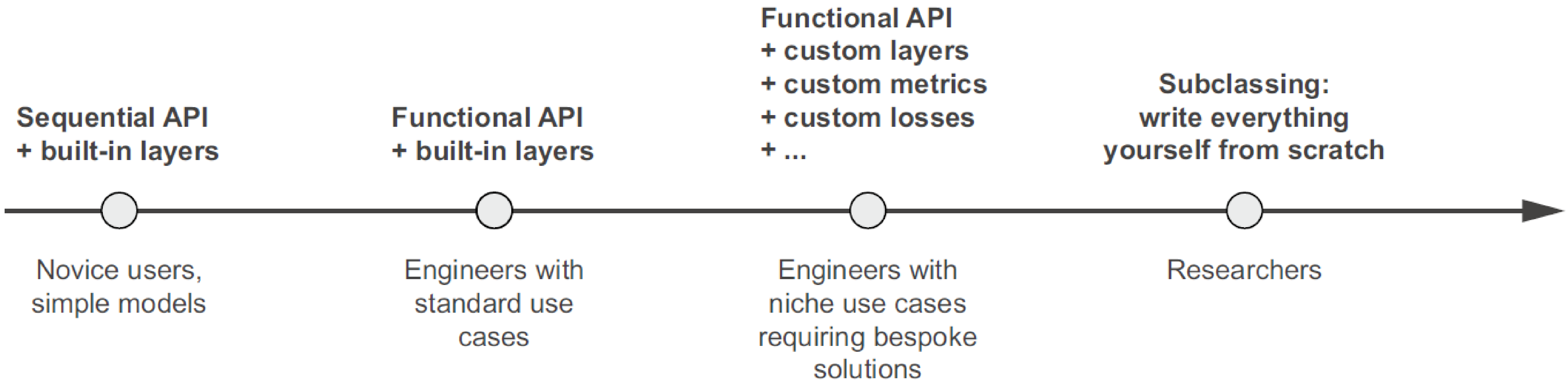
Working with Keras: a Deep Dive



Review

- models.Sequential() vs models.Model()
- layers.Dense()
- .compile(), .fit(), .evaluate(), .predict()

APIs for Building Models



Progressive disclosure of complexity for model building

Sequential API

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")
])

model = keras.Sequential()
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

At that point, the model isn't built yet.

```
>>> model.weights
ValueError: Weights for model sequential_1 have not yet been created.
```

Now you can retrieve the model's weights.

```
>>> model.build(input_shape=(None, 3))
>>> model.weights
[<tf.Variable "dense_2/kernel:0" shape=(3, 64) dtype=float32, ... >,
 <tf.Variable "dense_2/bias:0" shape=(64,) dtype=float32, ... >
 <tf.Variable "dense_3/kernel:0" shape=(64, 10) dtype=float32, ... >,
 <tf.Variable "dense_3/bias:0" shape=(10,) dtype=float32, ... >]
```

Builds the model—now the model will expect samples of shape (3,). The None in the input shape signals that the batch size could be anything.

Named Layers!

```
>>> model = keras.Sequential(name="my_example_model")
>>> model.add(layers.Dense(64, activation="relu", name="my_first_layer"))
>>> model.add(layers.Dense(10, activation="softmax", name="my_last_layer"))
>>> model.build((None, 3))
>>> model.summary()
Model: "my_example_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
my_first_layer (Dense)       (None, 64)                256
_________________________________________________________________
my_last_layer (Dense)        (None, 10)                650
=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
```

Specifying the Input Shape

```
model = keras.Sequential()
model.add(keras.Input(shape=(3,)))
model.add(layers.Dense(64, activation="relu"))
```

Use Input to declare the shape of the inputs. Note that the shape argument must be the shape of each sample, not the shape of one batch.

Functional API

```
inputs = keras.Input(shape=(3,), name="my_input")
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```

```
>>> inputs.shape
(None, 3)
>>> inputs.dtype
float32
```

The model will process batches where each sample has shape (3,). The number of samples per batch is variable (indicated by the None batch size).

These batches will have dtype float32.

```
>>> features.shape
(None, 64)
```

Multi-Input, Multi-Output Model

```
vocabulary_size = 10000
num_tags = 100
num_departments = 4

title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape=(vocabulary_size,), name="text_body")
tags = keras.Input(shape=(num_tags,), name="tags")

features = layers.Concatenate()([title, text_body, tags])
features = layers.Dense(64, activation="relu")(features)

priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department")(features)

model = keras.Model(inputs=[title, text_body, tags],
                    outputs=[priority, department])
```

Define model inputs.

Combine input features into a single tensor, features, by concatenating them.

Apply an intermediate layer to recombine input features into richer representations.

Define model outputs.

Create the model by specifying its inputs and outputs.

Lists for Inputs and Outputs

```
import numpy as np

num_samples = 1280
```

Dummy input data

```
title_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
text_body_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
tags_data = np.random.randint(0, 2, size=(num_samples, num_tags))
```

Dummy target data

```
priority_data = np.random.random(size=(num_samples, 1))
department_data = np.random.randint(0, 2, size=(num_samples, num_departments))

model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[["mean_absolute_error"], ["accuracy"]])
model.fit([title_data, text_body_data, tags_data],
          [priority_data, department_data],
          epochs=1)
model.evaluate([title_data, text_body_data, tags_data],
               [priority_data, department_data])
priority_preds, department_preds = model.predict(
    [title_data, text_body_data, tags_data])
```

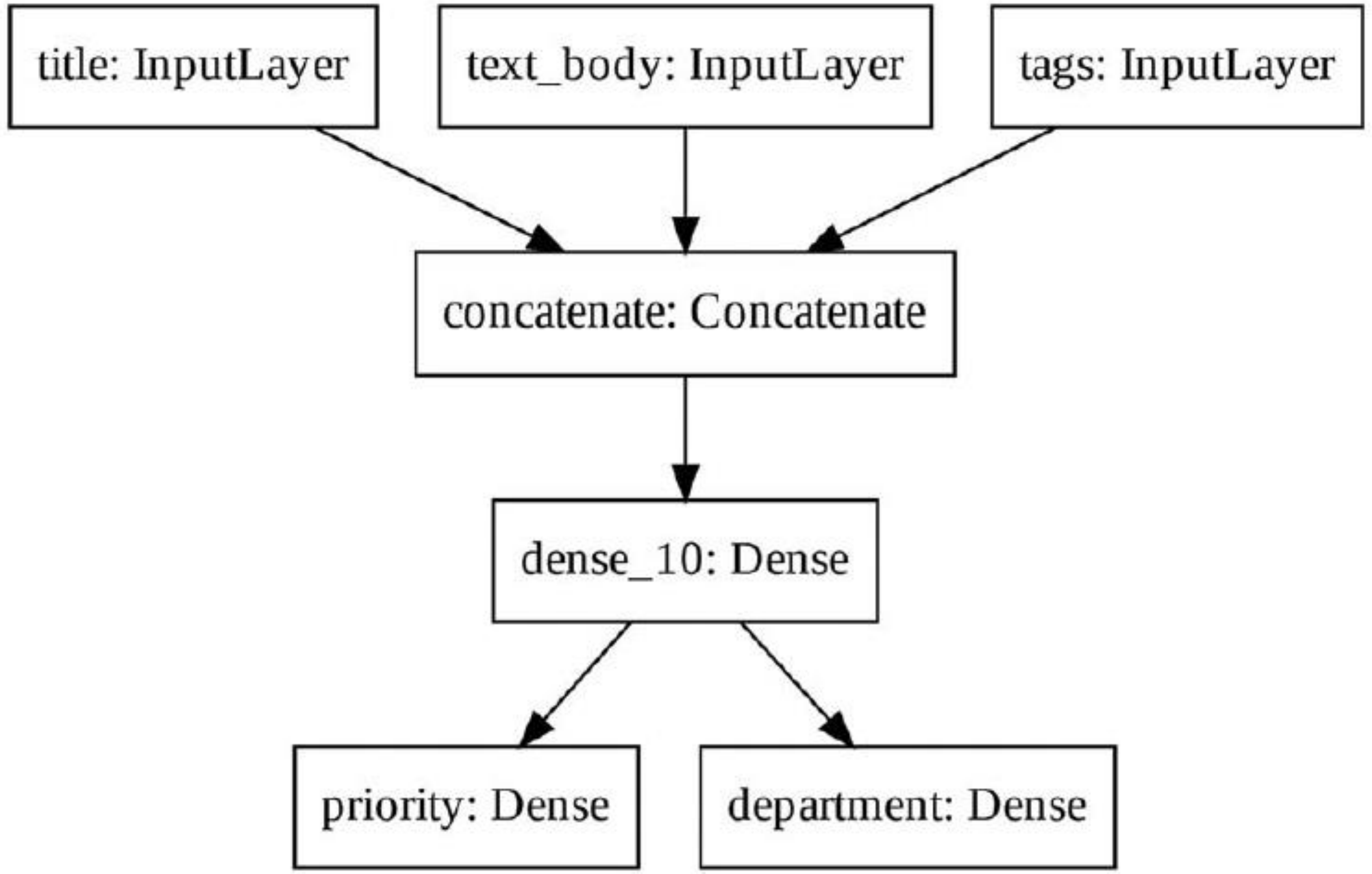
Dictionaries for Inputs and Outputs

```
model.compile(optimizer="rmsprop",
              loss={"priority": "mean_squared_error", "department":
                    "categorical_crossentropy"},
              metrics={"priority": ["mean_absolute_error"], "department":
                        ["accuracy"]})
model.fit({"title": title_data, "text_body": text_body_data,
           "tags": tags_data},
          {"priority": priority_data, "department": department_data},
          epochs=1)
model.evaluate({"title": title_data, "text_body": text_body_data,
                "tags": tags_data},
               {"priority": priority_data, "department": department_data})
priority_preds, department_preds = model.predict(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})
```

```
pip install pydot
sudo apt-get install graphviz
```

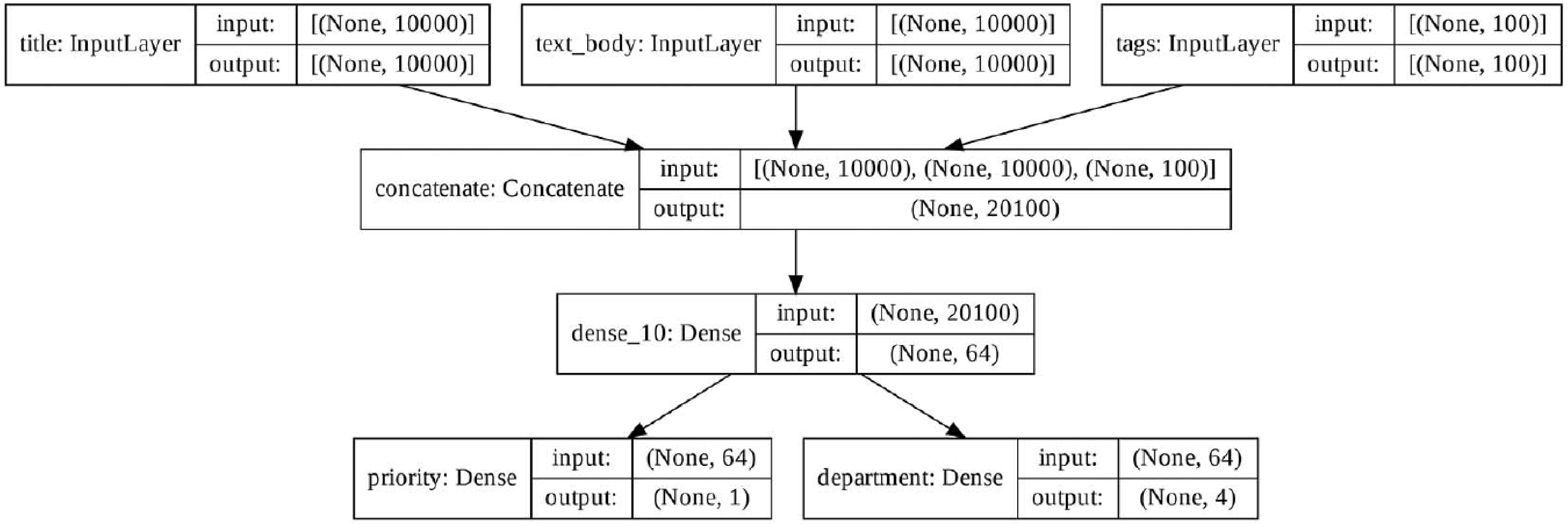
Plotting the Model

```
keras.utils.plot_model(model, "ticket_classifier.png")
```



Plotting the Model with Shape Information

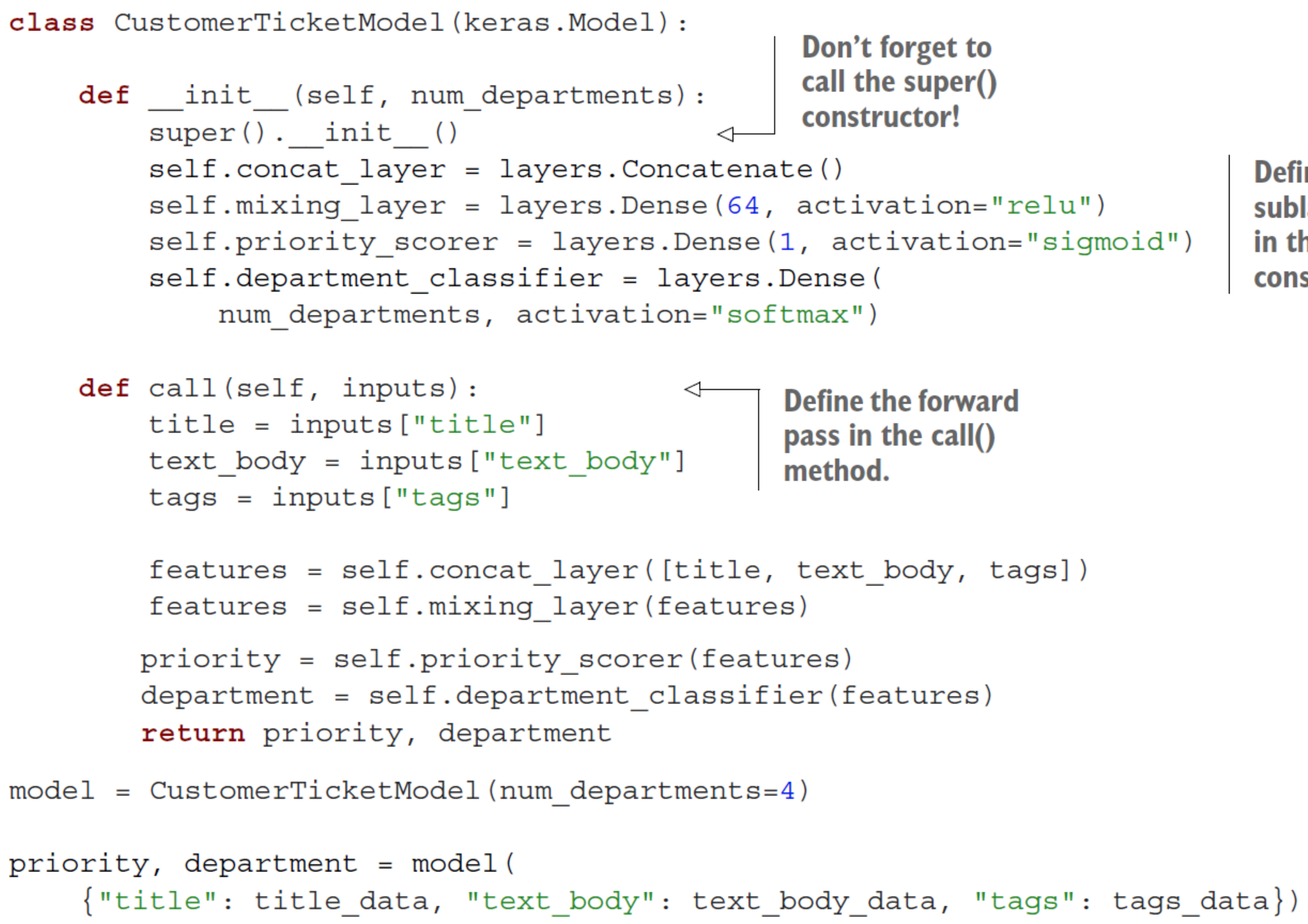
```
keras.utils.plot_model(
    model, "ticket_classifier_with_shape_info.png", show_shapes=True)
```



Retrieving Inputs and Outputs

```
>>> model.layers
[<tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d358>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d2e8>,
 <tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fa963f9d470>,
 <tensorflow.python.keras.layers.merge.Concatenate at 0x7fa963f9d860>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa964074390>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f9d898>,
 <tensorflow.python.keras.layers.core.Dense at 0x7fa963f95470>]
>>> model.layers[3].input
[<tf.Tensor "title:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "text_body:0" shape=(None, 10000) dtype=float32>,
 <tf.Tensor "tags:0" shape=(None, 100) dtype=float32>]
>>> model.layers[3].output
<tf.Tensor "concatenate/concat:0" shape=(None, 20100) dtype=float32>
```

Simple Subclassed Model



```
class CustomerTicketModel(keras.Model):

    def __init__(self, num_departments):
        super().__init__()
        self.concat_layer = layers.Concatenate()
        self.mixing_layer = layers.Dense(64, activation="relu")
        self.priority_scorer = layers.Dense(1, activation="sigmoid")
        self.department_classifier = layers.Dense(
            num_departments, activation="softmax")

    def call(self, inputs):
        title = inputs["title"]
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)

        priority = self.priority_scorer(features)
        department = self.department_classifier(features)
        return priority, department

model = CustomerTicketModel(num_departments=4)

priority, department = model(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})
```

Don't forget to call the super() constructor!

Define sublayers in the constructor.

Define the forward pass in the call() method.

Using the Subclassed Model

The structure of what you pass as the loss and metrics arguments must match exactly what gets returned by call()—here, a list of two elements.

```
model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[["mean_absolute_error"], ["accuracy"]])
model.fit({"title": title_data,
           "text_body": text_body_data,
           "tags": tags_data},
          [priority_data, department_data],
          epochs=1)
model.evaluate({"title": title_data,
                "text_body": text_body_data,
                "tags": tags_data},
               [priority_data, department_data])
priority_preds, department_preds = model.predict({"title": title_data,
                                                  "text_body": text_body_data,
                                                  "tags": tags_data})
```

The structure of the target data must match exactly what is returned by the call() method—here, a list of two elements.

The structure of the input data must match exactly what is expected by the call() method—here, a dict with keys title, text_body, and tags.

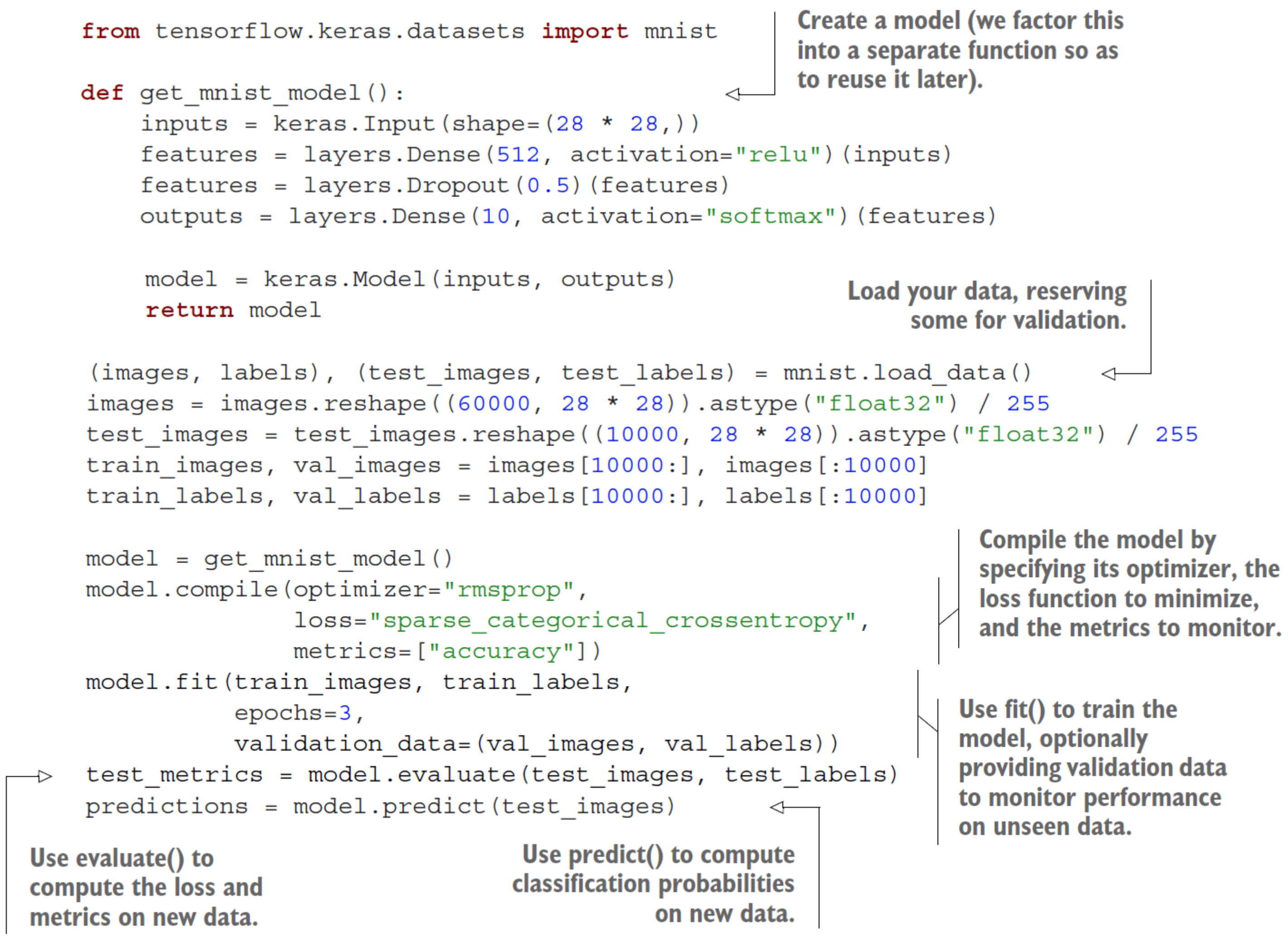
Including a Subclassed Model in a Functional Model

```
class Classifier(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        if num_classes == 2:
            num_units = 1
            activation = "sigmoid"
        else:
            num_units = num_classes
            activation = "softmax"
        self.dense = layers.Dense(num_units, activation=activation)

    def call(self, inputs):
        return self.dense(inputs)

inputs = keras.Input(shape=(3,))
features = layers.Dense(64, activation="relu")(inputs)
outputs = Classifier(num_classes=10)(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Including a Functional Model in a Subclassed Model

```
inputs = keras.Input(shape=(64,))
outputs = layers.Dense(1, activation="sigmoid")(inputs)
binary_classifier = keras.Model(inputs=inputs, outputs=outputs)

class MyModel(keras.Model):

    def __init__(self, num_classes=2):
        super().__init__()
        self.dense = layers.Dense(64, activation="relu")
        self.classifier = binary_classifier

    def call(self, inputs):
        features = self.dense(inputs)
        return self.classifier(features)

model = MyModel()
```

Standard Workflow: compile, fit, evaluate, predict



```
from tensorflow.keras.datasets import mnist

def get_mnist_model():
    inputs = keras.Input(shape=(28 * 28,))
    features = layers.Dense(512, activation="relu")(inputs)
    features = layers.Dropout(0.5)(features)
    outputs = layers.Dense(10, activation="softmax")(features)

    model = keras.Model(inputs, outputs)
    return model

(images, labels), (test_images, test_labels) = mnist.load_data()
images = images.reshape((60000, 28 * 28)).astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28)).astype("float32") / 255
train_images, val_images = images[10000:], images[:10000]
train_labels, val_labels = labels[10000:], labels[:10000]

model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=3,
          validation_data=(val_images, val_labels))
test_metrics = model.evaluate(test_images, test_labels)
predictions = model.predict(test_images)
```

Create a model (we factor this into a separate function so as to reuse it later).

Load your data, reserving some for validation.

Compile the model by specifying its optimizer, the loss function to minimize, and the metrics to monitor.

Use fit() to train the model, optionally providing validation data to monitor performance on unseen data.

Use evaluate() to compute the loss and metrics on new data.

Use predict() to compute classification probabilities on new data.

Creating a Custom Metric

```
import tensorflow as tf

class RootMeanSquaredError(keras.metrics.Metric):

    def __init__(self, name="rmse", **kwargs):
        super().__init__(name=name, **kwargs)
        self.mse_sum = self.add_weight(name="mse_sum", initializer="zeros")
        self.total_samples = self.add_weight(
            name="total_samples", initializer="zeros", dtype="int32")

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = tf.one_hot(y_true, depth=tf.shape(y_pred)[1])
        mse = tf.reduce_sum(tf.square(y_true - y_pred))
        self.mse_sum.assign_add(mse)
        num_samples = tf.shape(y_pred)[0]
        self.total_samples.assign_add(num_samples)

    def result(self):
        return tf.sqrt(self.mse_sum / tf.cast(self.total_samples, tf.float32))

    def reset_state(self):
        self.mse_sum.assign(0.)
        self.total_samples.assign(0)
```

Subclass the Metric class.

Define the state variables in the constructor. Like for layers, you have access to the add_weight() method.

To match our MNIST model, we expect categorical predictions and integer labels.

Implement the state update logic in update_state(). The y_true argument is the targets (or labels) for one batch, while y_pred represents the corresponding predictions from the model. You can ignore the sample_weight argument—we won't use it here.

Using a Custom Metric

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy", RootMeanSquaredError()])
model.fit(train_images, train_labels,
          epochs=3,
          validation_data=(val_images, val_labels))
test_metrics = model.evaluate(test_images, test_labels)
```

Callbacks

- callbacks.ModelCheckpoint: Saving the current state of the model at different points during training
- callbacks.EarlyStopping: Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training)
- callbacks.ReduceLROnPlateau: Dynamically adjusting the value of certain parameters during training—Such as the learning rate of the optimizer
- callbacks.CSVLogger: Logging training and validation metrics during training

callbacks argument of the .fit() method

Callbacks are passed to the model via the callbacks argument in fit(), which takes a list of callbacks. You can pass any number of callbacks.

Interrupts training when improvement stops

Monitors the model's validation accuracy

Interrupts training when accuracy has stopped improving for two epochs

Saves the current weights after every epoch

Path to the destination model file

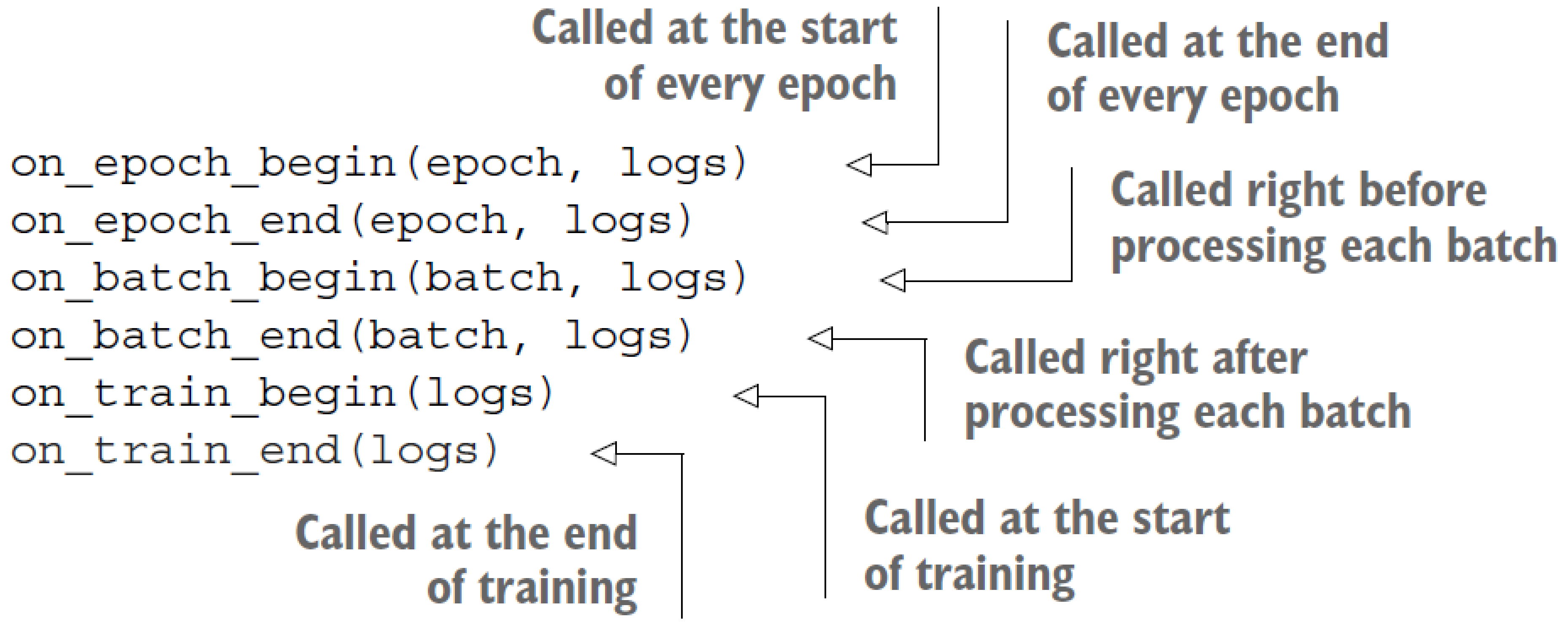
```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=2,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="checkpoint_path.keras",
        monitor="val_loss",
        save_best_only=True,
    )
]
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=callbacks_list,
          validation_data=(val_images, val_labels))
```

These two arguments mean you won't overwrite the model file unless val_loss has improved, which allows you to keep the best model seen during training.

You monitor accuracy, so it should be part of the model's metrics.

Note that because the callback will monitor validation loss and validation accuracy, you need to pass validation_data to the call to fit().

Methods for Callbacks



```
on_epoch_begin(epoch, logs)
on_epoch_end(epoch, logs)
on_batch_begin(batch, logs)
on_batch_end(batch, logs)
on_train_begin(logs)
on_train_end(logs)
```

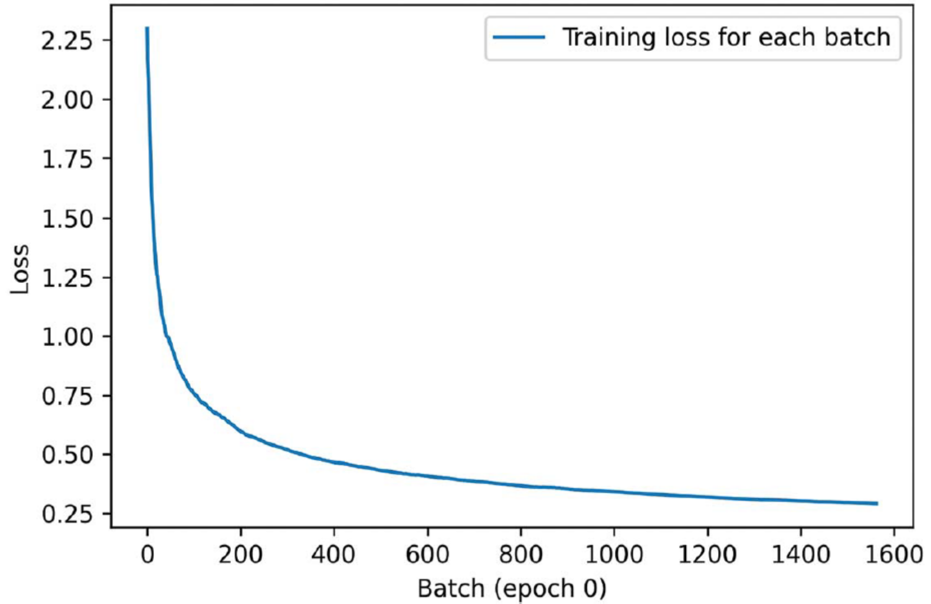
Called at the start of every epoch
Called at the end of every epoch
Called right before processing each batch
Called right after processing each batch
Called at the start of training
Called at the end of training

Creating a Custom Callback

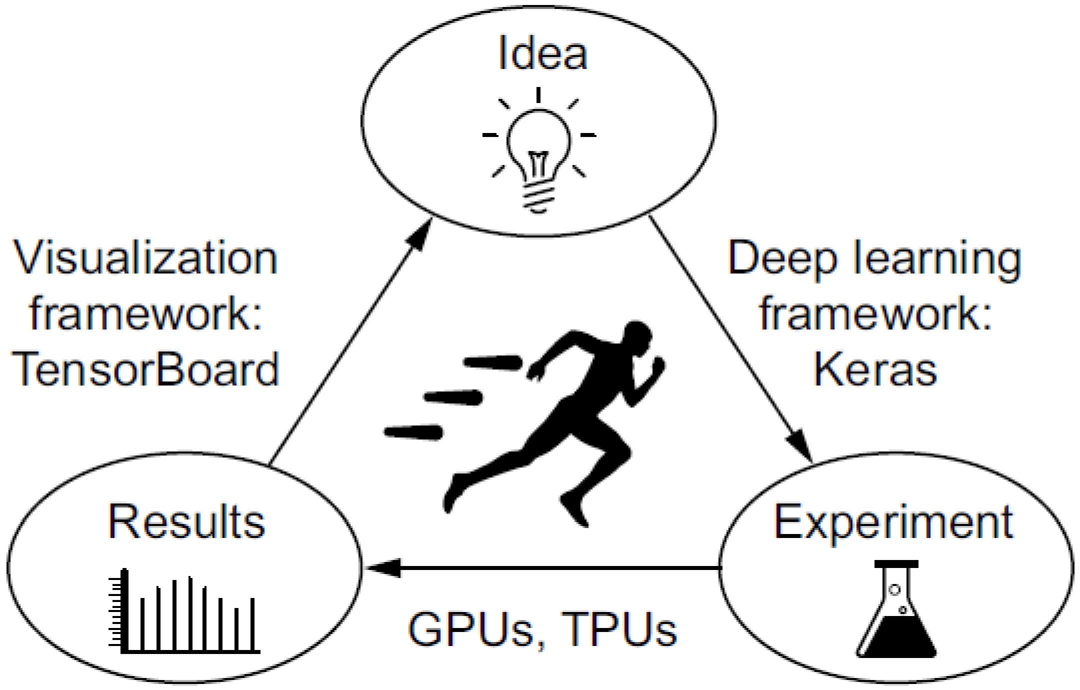
```
from matplotlib import pyplot as plt

class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs):
        self.per_batch_losses = []

    def on_batch_end(self, batch, logs):
        self.per_batch_losses.append(logs.get("loss"))

    def on_epoch_end(self, epoch, logs):
        plt.clf()
        plt.plot(range(len(self.per_batch_losses)), self.per_batch_losses,
                 label="Training loss for each batch")
        plt.xlabel(f"Batch (epoch {epoch})")
        plt.ylabel("Loss")
        plt.legend()
        plt.savefig(f"plot_at_epoch_{epoch}")
        self.per_batch_losses = []
```

Using a Custom Callback

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          callbacks=[LossHistory()],
          validation_data=(val_images, val_labels))
```



Iterative Progress


Idea
Deep learning framework: Keras
Experiment
GPUs, TPUs
Results
Visualization framework: TensorBoard

TensorBoard

- Visually monitor metrics during training
- Visualize your model architecture
- Visualize histograms of activations and gradients
- Explore embeddings in 3D

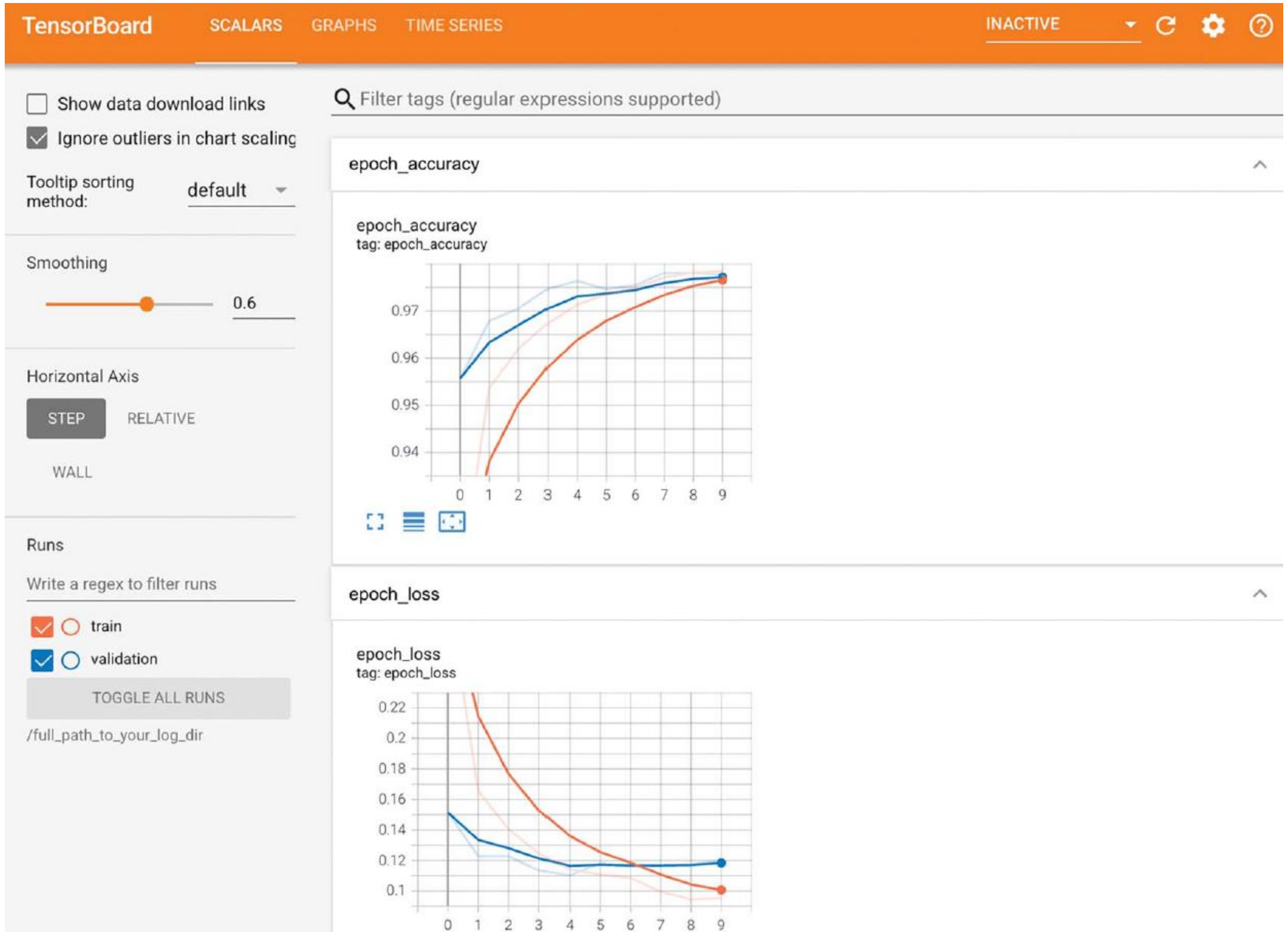
Using TensorBoard

```
model = get_mnist_model()
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

tensorboard = keras.callbacks.TensorBoard(
    log_dir="/full_path_to_your_log_dir",
)
model.fit(train_images, train_labels,
          epochs=10,
          validation_data=(val_images, val_labels),
          callbacks=[tensorboard])
```

```
tensorboard --logdir /full_path_to_your_log_dir
```

```
%load_ext tensorboard
%tensorboard --logdir /full_path_to_your_log_dir
```

Tensorboard

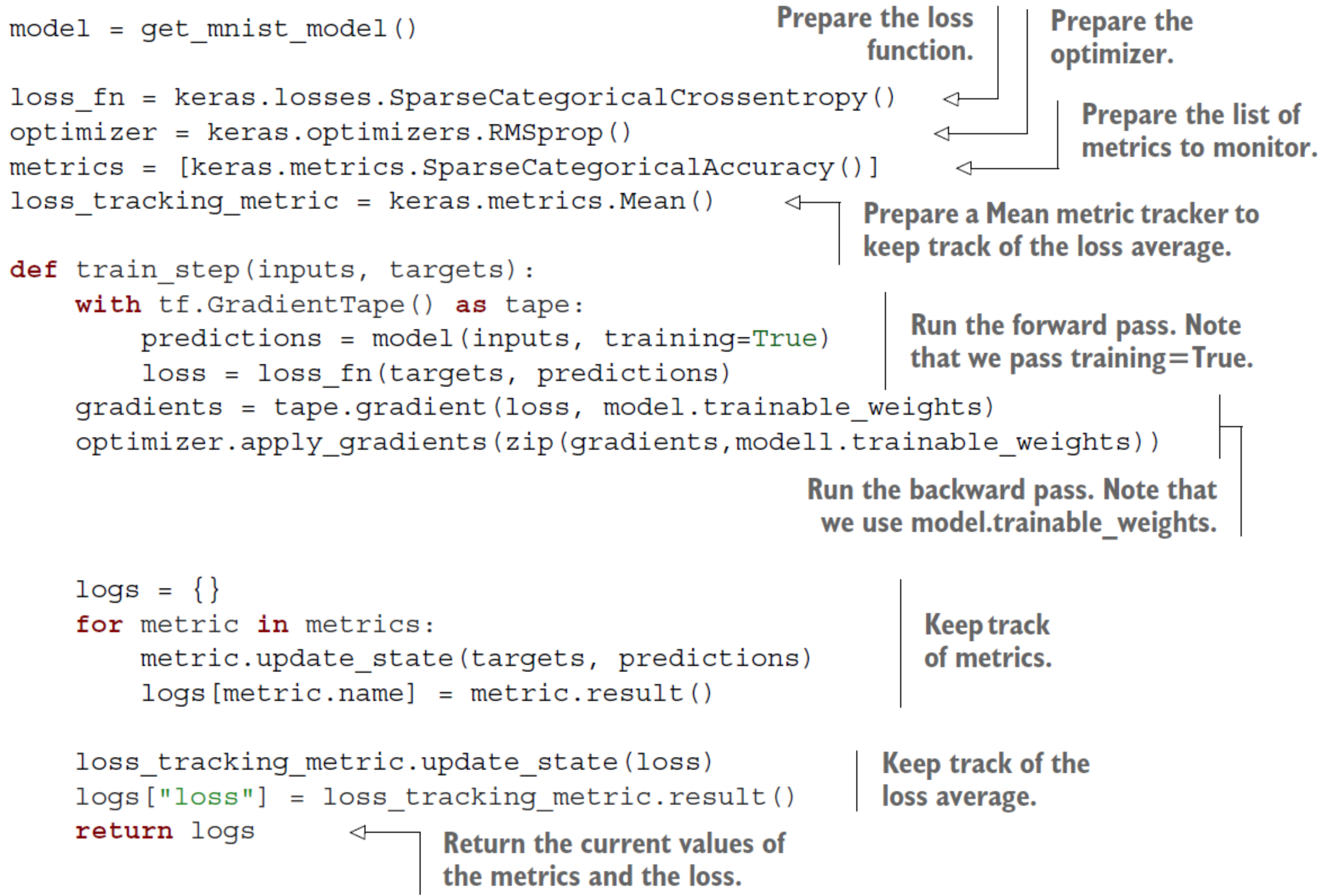


☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis [✓ STEP] RELATIVE WALL
Runs
Write a regex to filter runs
☑ train
☑ validation
/full_path_to_your_log_dir

Trainable vs Non-Trainable Weights

- Trainable weights: These are meant to be updated via backpropagation to minimize the loss of the model, such as the kernel and bias of a Dense layer.
- Non-trainable weights: These are meant to be updated during the forward pass by the layers that own them. For instance, if you wanted a custom layer to keep a counter of how many batches it has processed so far, that information would be stored in a non-trainable weight, and at each batch, your layer would increment the counter by one.

Creating a Training Step



```
model = get_mnist_model()

loss_fn = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.RMSprop()
metrics = [keras.metrics.SparseCategoricalAccuracy()]
loss_tracking_metric = keras.metrics.Mean()

def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        loss = loss_fn(targets, predictions)
    gradients = tape.gradient(loss, model.trainable_weights)
    optimizer.apply_gradients(zip(gradients,modell.trainable_weights))


    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs[metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["loss"] = loss_tracking_metric.result()
    return logs
```

Prepare the loss function.

Prepare the optimizer.

Prepare the list of metrics to monitor.

Prepare a Mean metric tracker to keep track of the loss average.

Run the forward pass. Note that we pass training=True.

Run the backward pass. Note that we use model.trainable_weights.

Keep track of metrics.

Keep track of the loss average.

Return the current values of the metrics and the loss.

Creating a Training Loop

```
def reset_metrics():
    for metric in metrics:
        metric.reset_state()
    loss_tracking_metric.reset_state()

training_dataset = tf.data.Dataset.from_tensor_slices(
    (train_images, train_labels))
training_dataset = training_dataset.batch(32)
epochs = 3
for epoch in range(epochs):
    reset_metrics()
    for inputs_batch, targets_batch in training_dataset:
        logs = train_step(inputs_batch, targets_batch)
    print(f"Results at the end of epoch {epoch}")
    for key, value in logs.items():
        print(f"...{key}: {value:.4f}")
```

Creating an Evaluation Loop

```
def test_step(inputs, targets):
    predictions = model(inputs, training=False)
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()
for inputs_batch, targets_batch in val_dataset:
    logs = test_step(inputs_batch, targets_batch)
print("Evaluation results:")
for key, value in logs.items():
    print(f"...{key}: {value:.4f}")
```

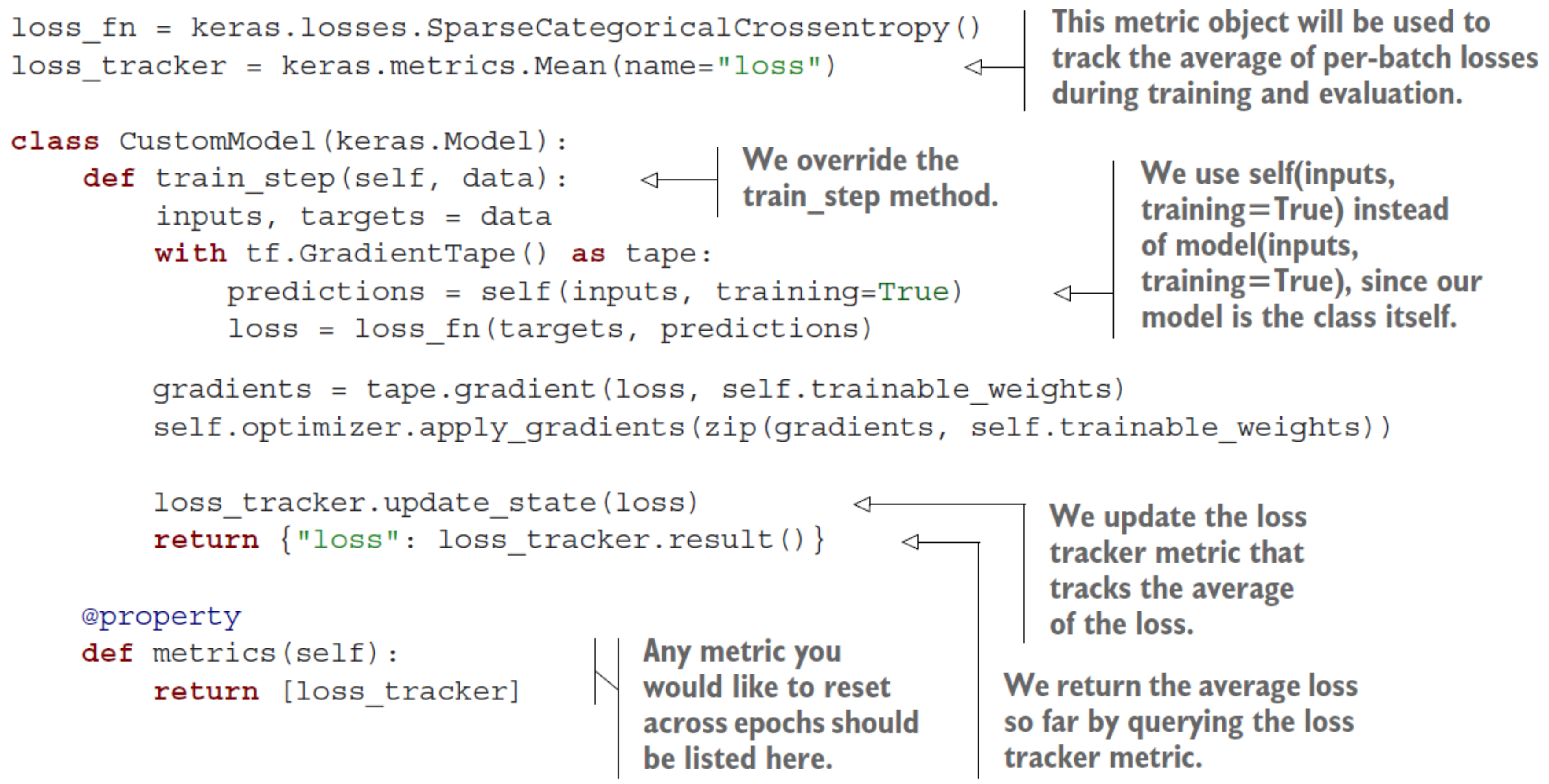
Note that we pass training=False.

Adding @tf.function to compile (faster)

```
@tf.function
def test_step(inputs, targets):
    predictions = model(inputs, training=False)
    loss = loss_fn(targets, predictions)

    logs = {}
    for metric in metrics:
        metric.update_state(targets, predictions)
        logs["val_" + metric.name] = metric.result()

    loss_tracking_metric.update_state(loss)
    logs["val_loss"] = loss_tracking_metric.result()
    return logs

val_dataset = tf.data.Dataset.from_tensor_slices((val_images, val_labels))
val_dataset = val_dataset.batch(32)
reset_metrics()
```

This is the only line that changed.

Custom Training Step



```
loss_fn = keras.losses.SparseCategoricalCrossentropy()
loss_tracker = keras.metrics.Mean(name="loss")

class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = loss_fn(targets, predictions)

        gradients = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(gradients, self.trainable_weights))

        loss_tracker.update_state(loss)
        return {"loss": loss_tracker.result()}

    @property
    def metrics(self):
        return [loss_tracker]
```

This metric object will be used to track the average of per-batch losses during training and evaluation.

We override the train_step method.

We use self(inputs, training=True) instead of model(inputs, training=True), since our model is the class itself.

We update the loss tracker metric that tracks the average of the loss.

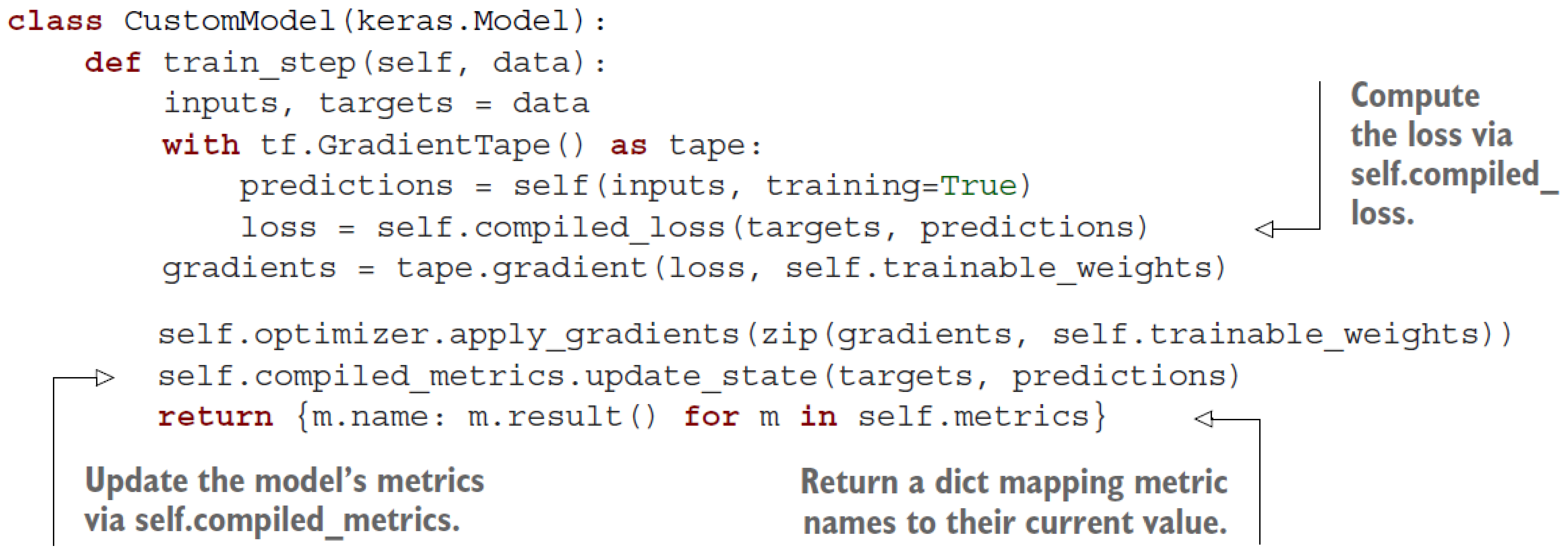
We return the average loss so far by querying the loss tracker metric.

Any metric you would like to reset across epochs should be listed here.

Using a Custom Training Step

```
inputs = keras.Input(shape=(28 * 28,))
features = layers.Dense(512, activation="relu")(inputs)
features = layers.Dropout(0.5)(features)
outputs = layers.Dense(10, activation="softmax")(features)
model = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop())
model.fit(train_images, train_labels, epochs=3)
```

Custom Training Step with compiled_metrics



```
class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = self.compiled_loss(targets, predictions)
        gradients = tape.gradient(loss, self.trainable_weights)

        self.optimizer.apply_gradients(zip(gradients, self.trainable_weights))
        self.compiled_metrics.update_state(targets, predictions)
        return {m.name: m.result() for m in self.metrics}
```

Compute the loss via self.compiled_loss.

Update the model's metrics via self.compiled_metrics.

Return a dict mapping metric names to their current value.

Using a Custom Training Step

```
inputs = keras.Input(shape=(28 * 28,))
features = layers.Dense(512, activation="relu")(inputs)
features = layers.Dropout(0.5)(features)
outputs = layers.Dense(10, activation="softmax")(features)
model = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.SparseCategoricalCrossentropy(),
              metrics=[keras.metrics.SparseCategoricalAccuracy()])
model.fit(train_images, train_labels, epochs=3)
```

Summary

- Keras offers a spectrum of different workflows, based on the principle of progressive disclosure of complexity. They all smoothly inter-operate together.
- You can build models via the Sequential class, via the Functional API, or by subclassing the Model class. Most of the time, you'll be using the Functional API.
- The simplest way to train and evaluate a model is via the default fit() and evaluate() methods.
- Keras callbacks provide a simple way to monitor models during your call to fit() and automatically take action based on the state of the model.
- You can also fully take control of what fit() does by overriding the train_step() method.
- Beyond fit(), you can also write your own training loops entirely from scratch. This is useful for researchers implementing brand-new training algorithms.
- Update the model's metrics via self.compiled_metrics. Return a dict mapping metric names to their current value.