Introduction to TensorFlow 2 and Keras

Thomas Timmermann

codecentric AG | February 18, 2020

1. Let's get started!

This handout accompanies the workshop "An introduction to TensorFlow 2 and Keras" and contains the **slides** and the **notebooks** which can both be found at the github repository https://github.com/thomastimmermann/TensorFlow-2-Workshop.

1.1 Introduction round

Thomas Timmermann

- **Mathematician**
  PhD + Habilitation
  16 years
- **Data Scientist**
  codecentric
  1.4 years
- father of three kids
  (all ill last week...)

1.2 Our plan for today

What is the core of TensorFlow?

hands-on: **online linear regression**

Building neural networks with keras

hands-on: **classifying digits** and **matching them**

TensorFlow Eco-System Goodies

hands-on: **titanic survivals**

1.3 Setting up the environment

No... I haven't set up anything yet:

`REPO_URL` being `https://github.com/thomastimmermann/TensorFlow-2-Workshop.git`, do

```
git clone REPO_URL
cd TensorFlow-2-Workshop
python3 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip setuptools
pip install -r requirements.txt
```

I did all that... but not this morning:

```
cd TensorFlow-2-Workshop
git pull
source .venv/bin/activate
pip install -r requirements.txt
```

...or go to https://mybinder.org/v2/gh/thomastimmermann/TensorFlow-2-Workshop/master

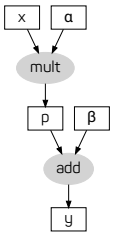
2. TensorFlow in 10 Minutes...

2.1 The Core of TensorFlow

1. Tensors

0	scalars
1	vectors
2	matrices
n	**tensors**

2. Flow



3. Gradients

implement chain rule

$$\frac{\partial y}{\partial \alpha} = \frac{\partial y}{\partial p} \cdot \frac{\partial p}{\partial \alpha}$$

via **back-propagation**

→ optimize ml/dl models via gradient descent, efficiently

2.2 The TensorFlow Eco-System

Execution Engine
- CPUs, GPUs, TPUs
- Linux, iOS, Windows, Android

low-level API
- tensors and gradients
- dl basics
- I/O and preprocessing
- deployment, logging...

high-level API
- pre-built estimators
- keras for dl

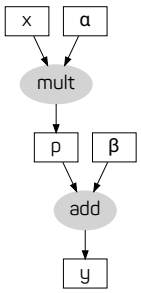
TensorFlow Extended
- data pipelines
- data validation
- model analysis
- deployment

TFLite

TensorFlow.js

2.3 So, why does TensorFlow not rule the world?

this is what happened in **TensorFlow 1.xx**...



- code creates **computation graph**

```
import tensorflow as tf

x, alpha, beta = 2, 3, 4
p = tf.multiply(x, alpha, name='p')
y = tf.add(p, beta, name='y')
```

- computation = **run** the graph

```
with tf.Session() as sess:
    sess.run(y)
```

→ poor debugging, slow prototyping, research turns to pytorch

2.4 How will TensorFlow 2.0 win back?

eager execution by default

```
import tensorflow as tf


def linear(x, alpha, beta):
    return x * alpha + beta
```

computes the result **immediately**
→ **fast prototyping, easy debugging**

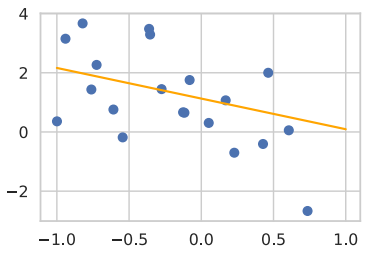
graph mode if wantd

```
import tensorflow as tf

@tf.function
def linear(x, alpha, beta):
    return x * alpha + beta
```

creates **computation graph** for later runs
→ **optimized performance**

3. Hands-on: Fun with tensors and gradient

3.1 The simplest ML model: linear regression



Data

- samples $(x_1, y_1), \ldots, (x_n, y_n)$

Linear Model

- parameters α, β
- predictions $\tilde{y}_i = \alpha x_i + \beta$

Task

- minimize loss $\sum_i (\tilde{y}_i - y_i)^2$

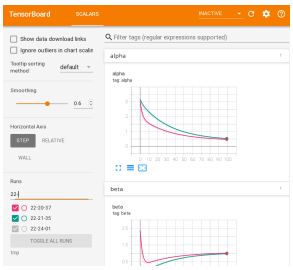
3.2 Online learning via gradient descent

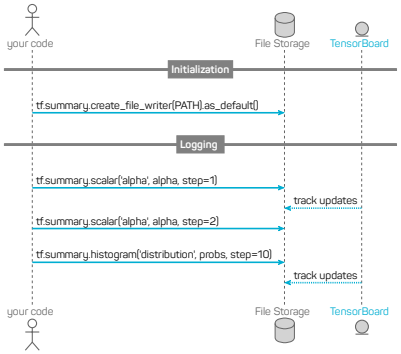
1. choose α, β randomly
2. repeat **gradient descent** steps
   compute
   - **predictions** $\tilde{y}_i = \alpha x_i + \beta$
   - **loss** $L = \sum_i (y_i - \tilde{y}_i)^2$
   - **gradients** $\partial L / \partial \alpha$ and $\partial L / \partial \beta$

   update
   - **parameter** $\alpha \leftarrow \alpha - \eta \cdot \partial L / \partial \alpha$
   - **parameter** $\beta \leftarrow \beta - \eta \cdot \partial L / \partial \beta$

:

3.3 Logging progress with TensorBoard





3.4 TensorFlow Core Packages

tensor operations, gradients and other core functionality available as top-level elements in `tensorflow`

extended math and signal processing `math`, `linalg`, `signal`, `random`, `bitwise`

I/O and preprocessing `data`, `io`, `queue`, `image`, `strings`

special data structures `sparse`, `ragged`, `sets`

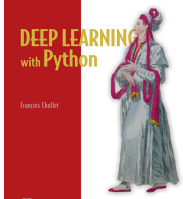
low-level and high-level deep learning API `nn`, `rnn`, `train` and `estimator`, `feature_column`, `keras`

deployment and optimization `saved_model`, `distribute`, `autograph`, `quantization`, `graph_util`

logging/visualization `summary`

4. tf.keras in 10 Minutes...

4.1 From Keras to tf.keras



05/2015 **Keras** as a **high-level API** to Torch, Theano, Caffee
09/2016 **tensorflow** becomes **default backend** of Keras
11/2017 **tf.keras integrates** Keras API into tensorflow
09/2019 **tensorflow 2.0** with tf.keras as **official high-level API**

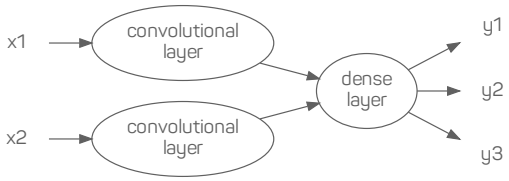
4.2 Build and train a model with tf.keras

1. **Build a model** as a **stack** or **graph** of **layers**
2. **Configure training** choosing the **loss function** and **optimizer**
3. **Train the model** with your **training data**

4.3 *Step 1:* Build a model..

- as a **stack of layers** using `tf.keras.Sequential`

- or as a **graph of layers** using `tf.keras.Model`



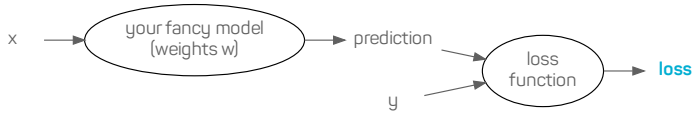
...using a wide range of layers, for example

tensor operations	`Add`, `Multiply`, `Concatenate`, ...
basic layers	`Dense`, `Activation`, `Lambda`, ...
convolutional layers	`Conv1D`, `Conv2D`, ..., `MaxPool1D`, ...
recurrent layers	`RNN`, `GRU`, `LSTM`
regularizing layers	`Dropout`, `BatchNormalization`, ...

4.4 *Step 2:* Configure the training of your model

choosing

- a **loss function** telling **what** to optimize

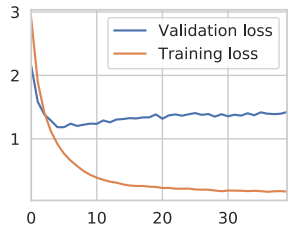


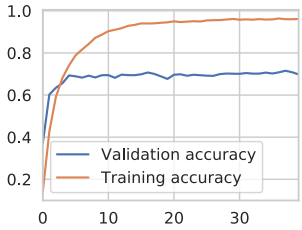
- an **optimizer** telling **how** to optimize

$$w_i^{t+1} = w_i^t - \eta(w^0, \dots, w^t) \frac{\partial \text{loss}}{\partial w_i}$$

4.5 *Step 3:* Train your model

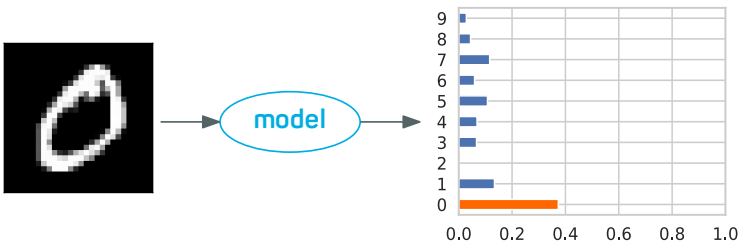
feeding training data in **batches** over several **epochs**





5. Hands-on #2: Fun with Keras and digits

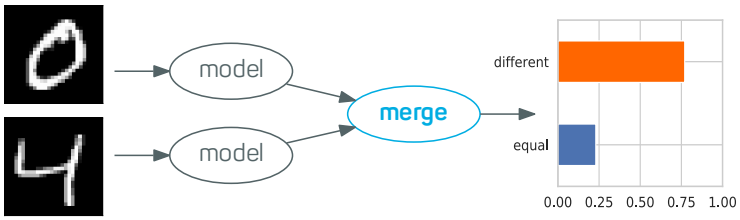
5.1 First Task: Classifying Digits



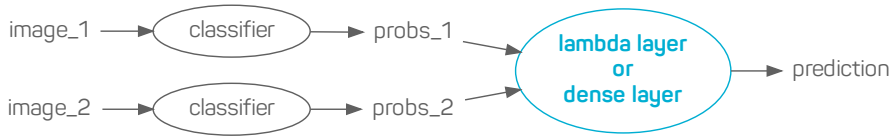

```
from tensorflow.keras import layers, Sequential

model = Sequential(
    [
```

```
        layers.Input((28, 28, 1)),
        layers.Flatten(),
        layers.Dense(10, activation="softmax"),
    ]
)
model.compile(optimizer="adam", loss="categorical_crossentropy")
history = model.fit(X, y, epochs=10, batch_size=32, validation_split=0.2)
```

5.2 Second Task: Matching Digits



5.3 Using functional API to build models




```
from tensorflow.keras import layers, Model

image_1 = layers.Input((28,28,1))
image_2 = layers.Input((28,28,1))

probs_1 = classifier(image_1)
probs_2 = classifier(image_2)
both_probs = layers.Concatenate()([probs_1, probs_2])

dense = layers.Dense(32, activation='relu')(both_probs)
prediction = layers.Dense(1, activation='sigmoid')(dense)
matcher = Model(inputs=[image_1, image_2], outputs=[prediction])
```

5.4 tf.keras Packages

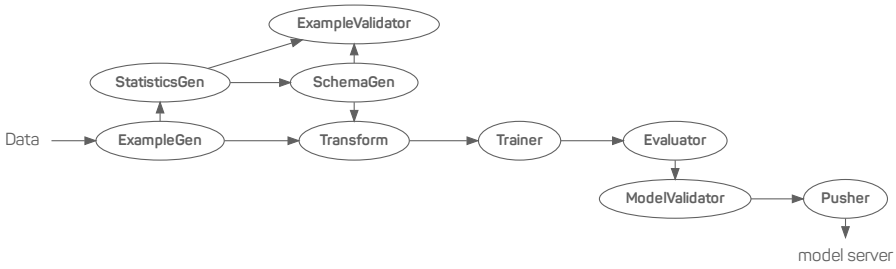
basic ingredients for building models `layers`, `activations`, `losses`, `models`
control over training process `optimizers`, `callbacks`, `metrics`
fine-grained control over weights `initializers`, `regularizers`, `contraints`
pre-trained models (DenseNet, MobileNet, ResNet, ...) `application`

helpful utilities `preprocessing`, `utils`, `estimator`
legacy APIs `backends`, `datasets`

6. TensorFlow Eco-System Goodies

6.1 TensorFlow Extended

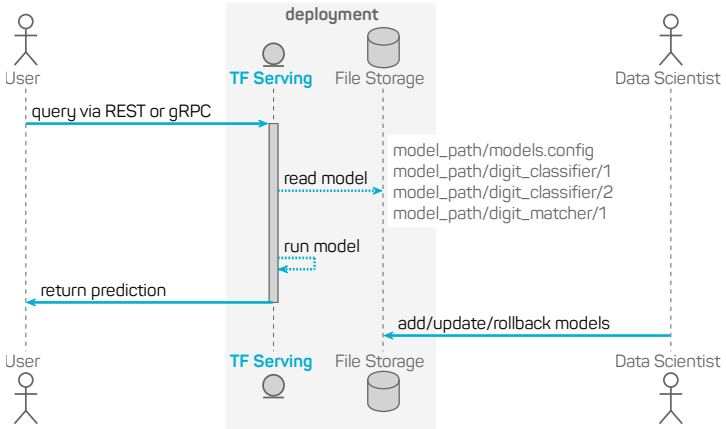
orchestrate **ml pipelines** with pre-built **components**



using

- `Apache Beam` / `AirFlow` / `Kubernetes`, and `ML MetaData` library
- `TF Data Validation`, `TF Model Analysis`, `TF Transform`
- `TF Serving` / `TF Lite` / `TF.js`

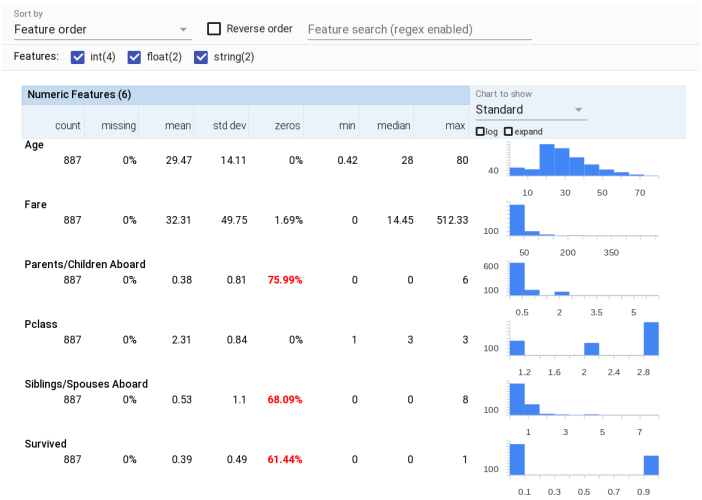
6.2 TensorFlow Serving



6.3 TensorFlow Data Validation

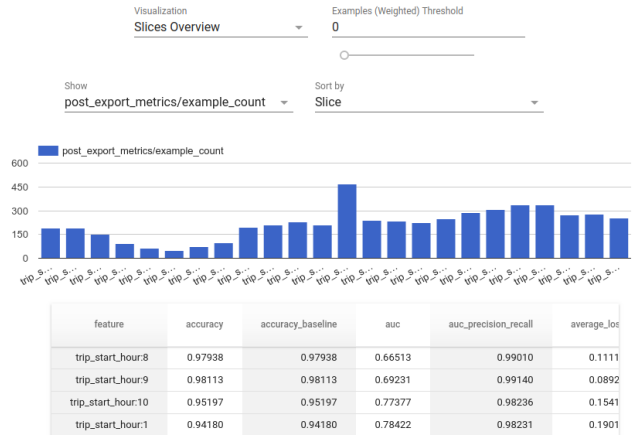
generate statistics from training data
visualize statistics in notebooks or standalone

infer data schema from statistics
validate new data against a schema



6.4 TensorFlow Model Analysis

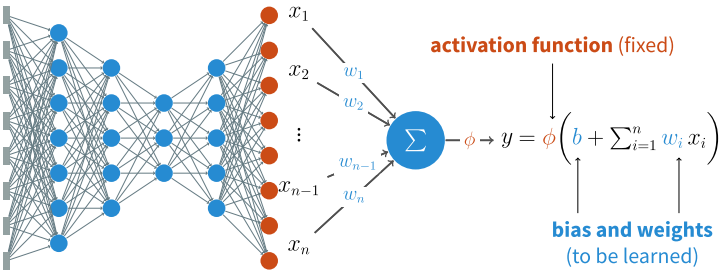
distributed evaluation on large validation data
in-depth analysis on **slices** of the data



7. Appendix Neural network basics

7.1 Artificial neural networks

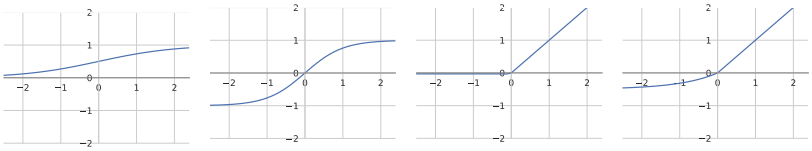
nets of perceptrons **perceptrons** = artificial neurons



7.2 Common activation functions

purpose: induce non-linearity
(compositions of linear functions are... linear)

sigmoid **tanh** **relu** **selu**



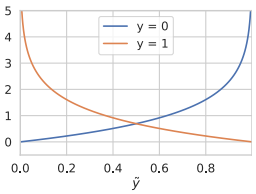
7.3 The SoftMax activation function

vector $(z_1, \dots, z_n)$ $\mapsto$ **probabilities** $(y_1, \dots, y_n)$ where $y_i = \dfrac{e^{z_i}}{\sum_j e^{z_j}}$

7.4 Common loss functions for classification

binary case

- true value y is either 0 or 1
- prediction $\tilde{y}$ is a probability
- binary crossentropy



n-class case

- $y = (0, \dots, 0, 1, 0, \dots, 0)$
  indicates the true class
- $\tilde{y} = (\tilde{y}_1, \dots, \tilde{y}_n)$
  is a probability distribution
- categorical crossentropy
  $-\sum_i y_i \log \tilde{y}_i$

7.5 Optimize the optimizer!

Momentum

move like a ball rolling downhill

$$\begin{cases} v^{(t+1)} = \gamma \cdot v^{(t)} + \eta^{(t)} \cdot \sum_{x,y} \frac{\partial L(y, \tilde{y}(x, w^{(t)}))}{\partial w^{(t)}} \\ w^{(t+1)} = w^{(t)} - v^{(t+1)} \end{cases} \quad (\gamma \approx 0.9)$$

Nesterov accelerated gradients
let the ball look ahead

Adagrad

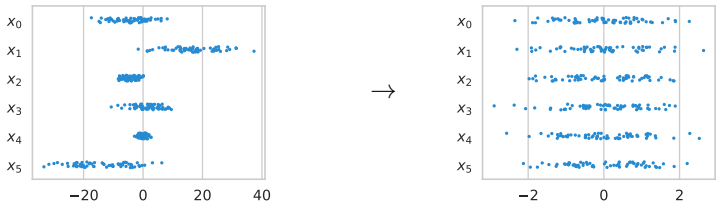
- high η for rarely changed weights
- low η for often changed weights

Adadelta
same but look at recent changes

Adam
think Adadelta with momentum

⇒ S. Ruder. An overview of gradient descent optimization algorithms. arXiv:1609.04747

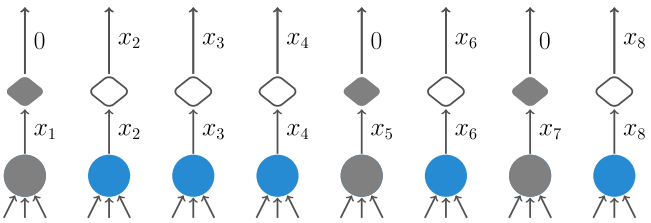
7.6 Batch Normalization Layer



What for each batch, rescale input components
Why regularization

When after convolutional layers and before activation
Keras Layer `BatchNormalization()`

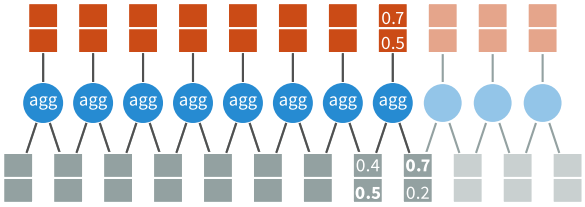
7.7 Dropout Layer



What randomly drop input components
Why regularization

When after dense layers
Keras Layer `Dropout(rate=0.5)`

7.8 Pooling layers

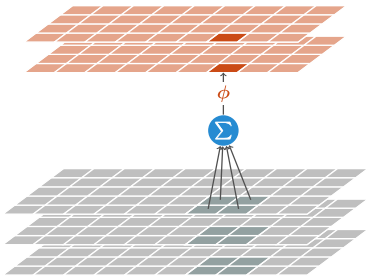


pool size=2
stride=1
agg=max

What sliding-window aggregation
Why get macroscopic features

When after convolutional layers
Keras `MaxPooling1D()`, `AveragePooling1D()`, ...

7.9 2D-Convolutional layers



What sliding-window perceptron
Why extract patterns

Keras Layer
`Conv2D(filters, kernel_size, strides,...)`

8. Hands-on: Hand-made Linear Regression

The purpose of this notebook is to get acquainted with **tensors** and **gradients**. As a guiding toy example, we shall implement linear regression using gradient descent.

Let us start with the necessary imports.

```
import tensorflow as tf
import numpy as np

import matplotlib.pyplot as plt
from IPython.display import set_matplotlib_formats
%matplotlib inline
set_matplotlib_formats('svg')
```

8.1 Step 1: Tensors, and Creating our Dataset

TensorFlow **tensors** are very similar to Numpy **arrays** — they have a certain **rank** and **shape**, the values in a tensor all have the same specified **dtype** (which can not be `object`, though), and

many array operations known from Numpy have an analogue in TensorFlow. To get started with our linear regression example, let's

- fix some initial parameters α, β,
- fix the sample size n,
- choose uniformly spaced $x_1, \dots, x_n$ in the intervall $[0, 1]$ and
- choose samples $y_i = \alpha x_i + \beta + e_i$ with some Gaussian noise e_i.

```
alpha = tf.constant(0.5)
beta = tf.constant(1.)
alpha, beta
```

```
n = 21
x = tf.linspace(0., 1., n)
x
```

```
y = alpha * x + beta + tf.random.normal((n,), stddev=0.1)
y
```

Let's plot the dataset including the line:

```
plt.scatter(x,y) # the points
plt.plot([0, 1], [beta, alpha + beta], color='orange') # the line
```

8.2 Step 2: Variables, and Initializing our Parameters

A TensorFlow **variable** is a wrapper for a tensor and

> the best way to represent shared, persistent state manipulated by your program.

They should be used to store **model parameters** and receive special treatment when tracking gradients and optimizing or saving models.

For our linear regression example, we want to start with random parameters:

```
alpha = tf.Variable(tf.random.uniform((1,),-5,5)[0],
                    name="alpha")
beta = tf.Variable(tf.random.uniform((1,),-5,5)[0],
                   name="beta")
alpha, beta
```

To update the value of a TensorFlow variable, one has to use the `assign` method or variants like `assing_add`, `assign_sub` and so on:

8.3 Step 3: Gradients and Gradient Tapes

To compute **gradients** of a function with respect TensorFlow tensors or variables, use the class `tf.GradientTape` and its methods `watch` and `gradient` as follows:

```
z = tf.constant(3.)
with tf.GradientTape() as tape:
    tape.watch(z)
    y = alpha * z + beta
tape.gradient(y, [alpha, z, beta])
```

A gradient tape records the gradients

- of the functions computed in its scope
- with respect to `tf.Variable` instances and with respect to tensors that are watched explicitly.

```
del tape
```

8.4 Step 4: Implementing Gradient Descent

To perform gradient descent, we need to compute the gradients of our **loss** with respect to the **model parameters**. Let us first write down the loss function:

```
def compute_loss(alpha, beta, x, y):
    y_predict = alpha * x + beta
    errors = y - y_predict
    loss = tf.reduce_sum(errors * errors)
    return loss

compute_loss(alpha, beta, x, y)
```

This could be shortened using suitable tensor operations (see the exercises).

Let us turn to the gradient descent step.

In the function above, the loss is computed from the parameters and the samples using several elementary functions. To compute the gradient of the loss with respect to the parameters, we need to

- keep track of the gradients of the elementary functions and
- combine these one-step gradients using the chain rule.

```
def gradient_step(alpha, beta, x, y, learning_rate):
    with tf.GradientTape() as tape:
        loss = compute_loss(alpha, beta, x, y)
    grad_alpha, grad_beta = tape.gradient(loss, [alpha, beta])
    alpha.assign_sub(grad_alpha * learning_rate)
    beta.assign_sub(grad_beta * learning_rate)
    return loss

def gradient_descent(alpha, beta, x, y,
                     nr_steps=100, learning_rate=0.01):
    alphas, betas, losses = [], [], []
    for step in range(0, nr_steps):
        loss = gradient_step(alpha, beta, x, y, learning_rate)
        alphas.append(alpha.read_value())
        betas.append(beta.read_value())
        losses.append(loss)
    return tf.stack(alphas), tf.stack(betas), tf.stack(losses)

def choose_params():
    alpha = tf.Variable(tf.random.uniform((1,),-5,5)[0],
                        name="alpha")
    beta = tf.Variable(tf.random.uniform((1,),-5,5)[0],
                       name="beta")
    return alpha, beta
```

```
alphas, betas, losses = gradient_descent(*choose_params(), x, y)
alphas, betas, losses
```

Let us visualize the results quickly:

```
import pandas as pd
import seaborn as sns

sns.set_style('whitegrid')

def visualize(alphas, betas, losses):
    pd.Series(losses, name='loss').plot()
    pd.DataFrame({'alpha': alphas, 'beta': betas}).plot()

visualize(*gradient_descent(*choose_params(), x, y))
```

8.5 Visualizing training with TensorBoard

For long-running computations, TensorFlow offers a convenient tool to log and visualize data: TensorBoard. To log the data, use the `tf.summary` module as, for example, in the following function:

```
import os
import time

LOGDIR = 'tmp'

def tb_gradient_descent(alpha, beta, x, y,
                        nr_steps=100, learning_rate=0.01):
    path = os.path.join(LOGDIR, time.strftime('%H-%M-%S'))
    with tf.summary.create_file_writer(path).as_default():
        for step in range(0, nr_steps):
            loss = gradient_step(alpha, beta, x, y,
                                 learning_rate)
            tf.summary.scalar('alpha', alpha.read_value(),
                              step=step)
            tf.summary.scalar('beta', beta.read_value(),
                              step=step)
            tf.summary.scalar('loss', loss, step=step)

tb_gradient_descent(*choose_params(), x, y, nr_steps=100)
```

```
!tensorboard --logdir=$LOGDIR
```

8.6 Eager mode versus graph mode

By default, TensorFlow 2 performs all tensor operations eagerly, which helps debugging and prototyping. But if we decorate a tensor function with `@tf.function`, on first run, the function gets compiled to a computation graph which then may run much more quickly.

```
alpha, beta = choose_params()
%timeit gradient_descent(alpha, beta, x, y, nr_steps=100)
```

```
alpha, beta = choose_params()
quick_descent = tf.function(gradient_descent)
%timeit quick_descent(alpha, beta, x, y, nr_steps=100)
```

The speed-up is impressive.

8.7 Exercises

Exercise 1: Tensor operations

Shorten the following function `get_loss` using `tf.square` or `tf.norm`:

```
def compute_loss(alpha, beta, x, y):
    y_predict = alpha * x + beta
    errors = y - y_predict
    loss = tf.reduce_sum(errors * errors)
    return loss
```

Exercise 2: Computing gradients

Make TensorFlow compute the derivative of the function $t \mapsto \cos t * \sin t$ at $t = 1$.

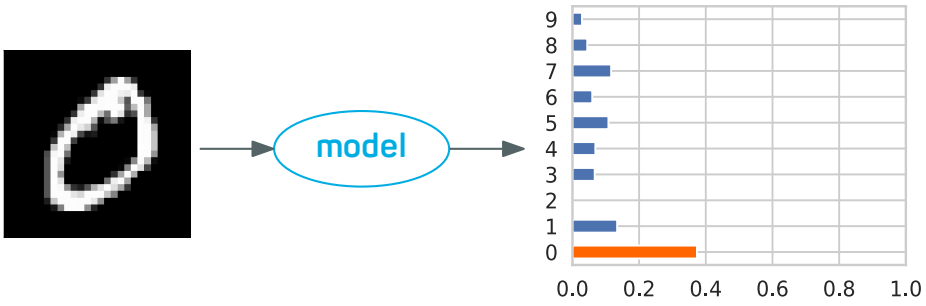
Exercise 3: Computing second-order differentials

Compute the second derivative of the function $f(t) = t\cos(t)$ at $t = 1$ and check that it equals $cos(1) - 2\sin(1)$:

9. Hands-on: Classifying digits

In this notebook we have a look at `tf.keras`, the high-level API of TensorFlow for building and training neural networks.

Our simple task will be to classify hand-written digits:



We start with some imports.

```
import tensorflow as tf
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
from IPython.display import set_matplotlib_formats
%matplotlib inline
set_matplotlib_formats('svg')
```

9.1 Step 1: Loading the MNIST dataset as a `tf.data.Dataset`

We use the MNIST dataset of images of hand-written digits:

```
import tensorflow_datasets as tdfs

tdfs.disable_progress_bar()

mnist_train = tdfs.load(name='mnist', split='train')
mnist_test = tdfs.load(name='mnist', split='test')

mnist_train, mnist_test
```

Let us have a look at the first three images:

```
GRAY_CMAP = plt.get_cmap('gray')

for sample in mnist_train.take(1):  # take one sample
    image, label = sample["image"], sample["label"]
    print(f'Label: {label}')
    plt.imshow(image[:, :, 0], cmap=GRAY_CMAP)
    plt.show()
```

9.2 Step 2: Preprocessing for classification

Our first aim is to classify the digits using a neural network. We therefore

- scale the image tensor so that the greyscale values are between 0 and 1, and
- one-hot-encode the labels:

```
eye = tf.eye(10, dtype='float32') # identity matrix of size 10x10


def scale_ohe(sample):
    scaled_image = tf.cast(sample['image'], 'float32') / 255.
    ohe_label = eye[tf.cast(sample['label'], 'int32')]
    return scaled_image,  ohe_label

Xy_train = mnist_train.map(scale_ohe)
Xy_test = mnist_test.map(scale_ohe)

print([(img.shape, label) for img, label in Xy_train.take(4)])
```

To make the task more interesting, we'll add some noise to the digits:

```
NOISE_STDDEV = 0.6

def add_noise(image, label):
    noise = tf.random.normal(image.shape, 0, NOISE_STDDEV)
    return image + noise, label
```

```
Xy_noisy_train = Xy_train.map(add_noise, 1)
Xy_noisy_test = Xy_test.map(add_noise, 1)
```

Let's check what the images look like now:

```
def show_images(dataset, nr_samples=3):
    _, axes = plt.subplots(1, nr_samples)
    for i, sample in enumerate(dataset.take(nr_samples)):
        axes[i].imshow(sample[0][:,:,0], cmap=GRAY_CMAP)
    plt.show()

show_images(Xy_train)
show_images(Xy_noisy_train)
```

9.3 Step 3: Building and training a sequential model

Let us build and train a neural network to classify the digits.

First, we build it as a sequential model, that is, as a stack of layers:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28,1)),
    tf.keras.layers.Dense(10, activation='softmax'),
])
model.summary()
```

Next, we specify how the model should learn, that is,

- which loss function should be optimized and
- which optimizer should be used.

Additionally, we declare a metrics that should be watched during training.

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Finally, we train the model, and obtain a training history that we plot:

```
history = model.fit(Xy_train.batch(32).take(200),
                    validation_data=Xy_test.batch(32).take(100),
                    epochs=5)

def plot_history(history):
    pd.DataFrame(history.history).plot.line()

plot_history(history)
```

Unsurprisingly, the results are quite good already.

Before we turn to the noisy data and play around with the model, we write a short function to compile and train a given model:

```
def train(model, train_data=Xy_train, test_data=Xy_test,
          nr_batches=200, nr_epochs=5):
```

```
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    train = train_data.batch(32).take(nr_batches)
    test = test_data.batch(32).take(100)
    history = model.fit(train,
                        validation_data=test,
                        epochs=nr_epochs)
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28,1)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(10, activation='softmax'),
])
train(model, Xy_noisy_train, Xy_noisy_test)
```

9.4 Step 4: Play around!

Now it's time for you to play around. For example, try to - add a `'Dense'` layer between the first and the last layer with activation `'relu'` or `'sigmoid'`, - add a `'Dropout'` layer before the last layer with a dropout rate of 0.3, - insert 2-dimensional convolutional layers `Conv2D` before the `Flatten` layer.

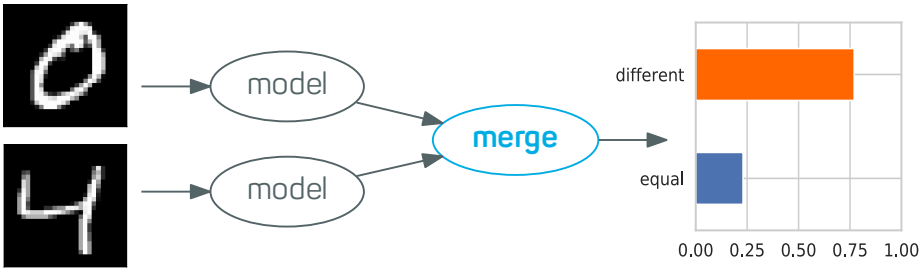
```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(8, kernel_size=3,
                           input_shape=(28,28,1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(pool_size=3, strides=2),
    tf.keras.layers.Conv2D(16, kernel_size=3),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.ReLU(),
    tf.keras.layers.MaxPooling2D(pool_size=3, strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax'),
])
train(model, Xy_train, Xy_test, nr_batches=400)
```

At the end, save your model:

```
CLASSIFIFER_PATH = 'classifier'

model.save(CLASSIFIFER_PATH, save_format='tf')
```

10. Hands-on: Matching digits

We now turn to the second task and build a model that - receives two images of hand-written digits as input and - outputs a probability that both images show the same digit:




```
import tensorflow as tf
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from IPython.display import set_matplotlib_formats
%matplotlib inline
set_matplotlib_formats('svg')
```

10.1 Step 1: Preparing the data

We use the MNIST dataset again:

```
import tensorflow_datasets as tdfs

tdfs.disable_progress_bar()

mnist_train = tdfs.load(name='mnist', split='train')
mnist_test = tdfs.load(name='mnist', split='test')

mnist_train, mnist_test
```

We now take pairs of subsequent samples, scale the images as before, and check whether the labels coincide:

```
mnist_train.batch(2)
```

```
def match_pairs(samples):
    images, digits = samples['image'], samples['label']
    matching = 1. if digits[0] == digits[1] else 0.
    return (images[0] / 255, images[1]/255), matching


Xy_train = mnist_train.batch(2).map(match_pairs)
Xy_test = mnist_test.batch(2).map(match_pairs)

Xy_train
```

Let us see how many matching samples we have in our training set:

```
Xy_train.reduce(tf.constant((0,0)),
                lambda count, sample: count + (sample[1], 1))
```

10.2 Step 2: Building the model

The `Sequential` class allows us to conveniently construct a neural network by stacking layers.

But if we need more flexibility, for example, to construct

- a model with multiple inputs or multiple outputs, or
- a general directed acyclic graph of layers,

we need to use the `tf.keras.Model` class, also known as the functional API of keras.

The idea for our model is that we

1. apply our pre-trained digit-classifier to both images,
2. obtain two probability distributions $p^{(1)}$ and $p^{(2)}$,
3. and use a dense layer to deduce the desired probability:

```
from tensorflow.keras import layers

CLASSIFIFER_PATH = 'classifier'

classifier = tf.keras.models.load_model(CLASSIFIFER_PATH)
classifier.trainable = False

def build_matcher_dense():
    image_1 = layers.Input((28,28,1))
    image_2 = layers.Input((28,28,1))
    probs = [classifier(image_1), classifier(image_2)]
    concat = layers.Concatenate()(probs)
    dense = layers.Dense(32, activation='relu')(concat)
    prediction = layers.Dense(1, activation='sigmoid')(dense)
    matcher = tf.keras.Model(inputs=[image_1, image_2],
                             outputs=[prediction])
    return matcher
```

Let's train our matcher!

```
def train(model, nr_batches=400, nr_epochs=5):
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    train = Xy_train.repeat().batch(32).take(nr_batches)
    test = Xy_test.repeat().batch(32).take(nr_batches // 2)
    history = model.fit(train,
                        validation_data=test,
                        epochs=nr_epochs)

matcher = build_matcher_dense()
train(matcher)
```

Let us now evaluate the matcher:

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

def evaluate(model, nr_samples=1000):
    y_list = list(Xy_test.map(lambda _, p: p).take(nr_samples))
    y_true = np.stack(y_list)
    X_batch= Xy_test.batch(nr_samples).take(1)
    y_pred = np.round(np.concatenate(model.predict(X_batch)))
    print(confusion_matrix(y_true, y_pred))
    print(classification_report(y_true, y_pred))

evaluate(matcher)
```

10.3 Step 3: Building a model that need not learn

We now replace the last dense classification layer with a lambda layer that need not be trained, using the following observation:

Given the two probability distributions $p^{(1)}$ and $p^{(2)}$, the probability that both digits coincide is $\sum_i p_i^{(1)} p_i^{(2)}$.

We can implement the formula in 3. using a `Lambda` layer. The catch is that all tensors flowing through the neural network are batches of data:

```
p1_batch = tf.constant([[0.1, 0.9], [0.5, 0.5]])
p2_batch = tf.constant([[0.2, 0.8],  [0.4, 0.6]])

def compute_prob(ps):
    return tf.reduce_sum(ps[0] * ps[1], axis=-1, keepdims=True)

compute_prob_equality([p1_batch, p2_batch])
```

```
def build_matcher_lambda():
    image_1 = tf.keras.layers.Input((28,28,1))
    image_2 = tf.keras.layers.Input((28,28,1))
    probs = [classifier(image_1), classifier(image_2)]
    prediction = tf.keras.layers.Lambda(compute_prob)(probs)
    matcher = tf.keras.Model(inputs=[image_1, image_2],
                             outputs=[prediction])
    return matcher
```

Let's see how this model performs!

```
matcher = build_matcher_lambda()
evaluate(matcher)
```

This is quite good, isn't it?

11. Hands-on #3: Titanic disaster

In this notebook, we'll take a look at TensorFlow Data Validation, using the Titanic dataset.

11.1 Step 1: Loading the data

```
import os
import pandas as pd
titanic_file = '../data/titanic/titanic.csv'
df = pd.read_csv(titanic_file)
df.head()
```

11.2 Step 2: Let TensorFlow Data Validation analyze the data

The DataValidation library can read data in form of TFRecord files, CSV files or pandas DataFrames, and generate statistics.

```
import tensorflow_data_validation as tfdv

stats = tfdv.generate_statistics_from_dataframe(df)
```

These statistics can be visualized directly in the notebook (but you may need to use Chrome or Chromium...)

```
tfdv.visualize_statistics(stats)
```

11.3 Step 3: Infer a schema and spot anomalies

From the statistics, we can infer a schema which can later be used as a blueprint to checck new data for anomalies:

```
schema = tfdv.infer_schema(stats)
schema
```

Let's create data with missing some value, and see whether this gets detected:

```
faulty_csv = '../data/faulty.csv'
df.iloc[[0],:].assign(Age=None).to_csv(faulty_csv)

options = tfdv.StatsOptions(schema=schema)
anomalies = tfdv.validate_examples_in_csv(faulty_csv, options)
```

```
tfdv.visualize_statistics(anomalies)
```

11.4 Step 4: Train a simple estimator to predict survivals

We now want to train a pre-built estimator on the dataset. First, we split the data:

```
from sklearn.model_selection import train_test_split
train, val = train_test_split(df, test_size=0.2)
```

Next, we use some preprocessing to make the pandas dataframe digestible for pre-built TensorFlow estimators.

```
import tensorflow as tf
import tensorflow.feature_column as tfc

def df_to_dataset(dataframe, shuffle=True, batch_size=32):
    dataframe = dataframe.copy()
```

```
    labels = dataframe.pop('Survived')
    ds = tf.data.Dataset.from_tensor_slices((dict(dataframe),
                                             labels))
    if shuffle:
        ds = ds.shuffle(buffer_size=len(dataframe))
        ds = ds.batch(batch_size)
    return ds

def input_fn():
    return df_to_dataset(train)

def input_fn_eval():
    return df_to_dataset(val)


age = tfc.numeric_column('Age')
sex = tfc.categorical_column_with_vocabulary_list('Sex',
                                                  df.Sex.unique())
sex_ohe = tfc.indicator_column(sex)
pclass = tfc.categorical_column_with_vocabulary_list('Pclass',
                                                  df.Pclass.unique())
pclass_ohe = tfc.indicator_column(pclass)

feature_columns = [age, sex_ohe, pclass_ohe]
```

Now comes the training...

```
classifier = tf.estimator.BoostedTreesClassifier(feature_columns,
                                                 n_batches_per_layer=5)
classifier.train(input_fn)
```

And now the validation:

```
classifier.evaluate(input_fn_eval)
```

As a final step, it would be nice to analyze our estimator with the TensorFlow Model Analysis library. But to do so, we still need to export the validation data in form of a TFRecord file, and it is too late for that right now... Good bye!