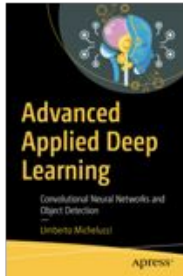
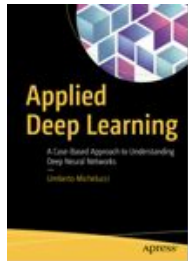


# Who am I?



Google Developer Expert  
in Machine Learning

Responsible for AI and Research  
Helsana Versicherung AG (CH)



Author of two books  
On Deep Learning and  
TensorFlow

Lecturer in Deep Learning  
Zürich university of applied  
science

Zürcher Hochschule  
für Angewandte Wissenschaften



Founder of TOELT llc  
[www.toelt.ai](http://www.toelt.ai)



# How to get in touch with me

---



+41 79 396 7406



[umberto.michelucci@toelt.ai](mailto:umberto.michelucci@toelt.ai)



<https://www.linkedin.com/in/umbertomichelucci/>

# Swiss TensorFlow User Group



The aim of this group is to bring people that work and want to learn how to use TensorFlow together. Regular meetings will provide members of the group with exceptional learning possibilities in many areas: from TensorFlow 2.0, Keras, TFX, Hardware acceleration for deep learning using GPUs and TPUs, distributed learning, on the edge learning, research, and much more.

<https://www.meetup.com/Swiss-TensorFlow-Usergroup>

We are happy to help and support your initiatives, events or learning questions.  
Contact us for any questions or requests at

[umberto.michelucci@gmail.com](mailto:umberto.michelucci@gmail.com) or [fabien.tarrade@gmail.com](mailto:fabien.tarrade@gmail.com)



# TOELT Training are always divided in two types

## Chalk Talk

- Focused on Theory
- Can be taught without beamer or computer
- No programming

### Definition of *chalk talk*\*

: a talk or lecture illustrated at a blackboard

## Hands-on Sessions

- Focused on programming
- Focused on modern tools and methods
- Theoretical understanding is expected

### Definition of *hands-on*\*

**1** : relating to, being, or providing direct practical experience in the operation or functioning of something

// *hands-on* training



**TensorFlow**



# TensorFlow 1.x/2.x



## An open source Deep Learning library

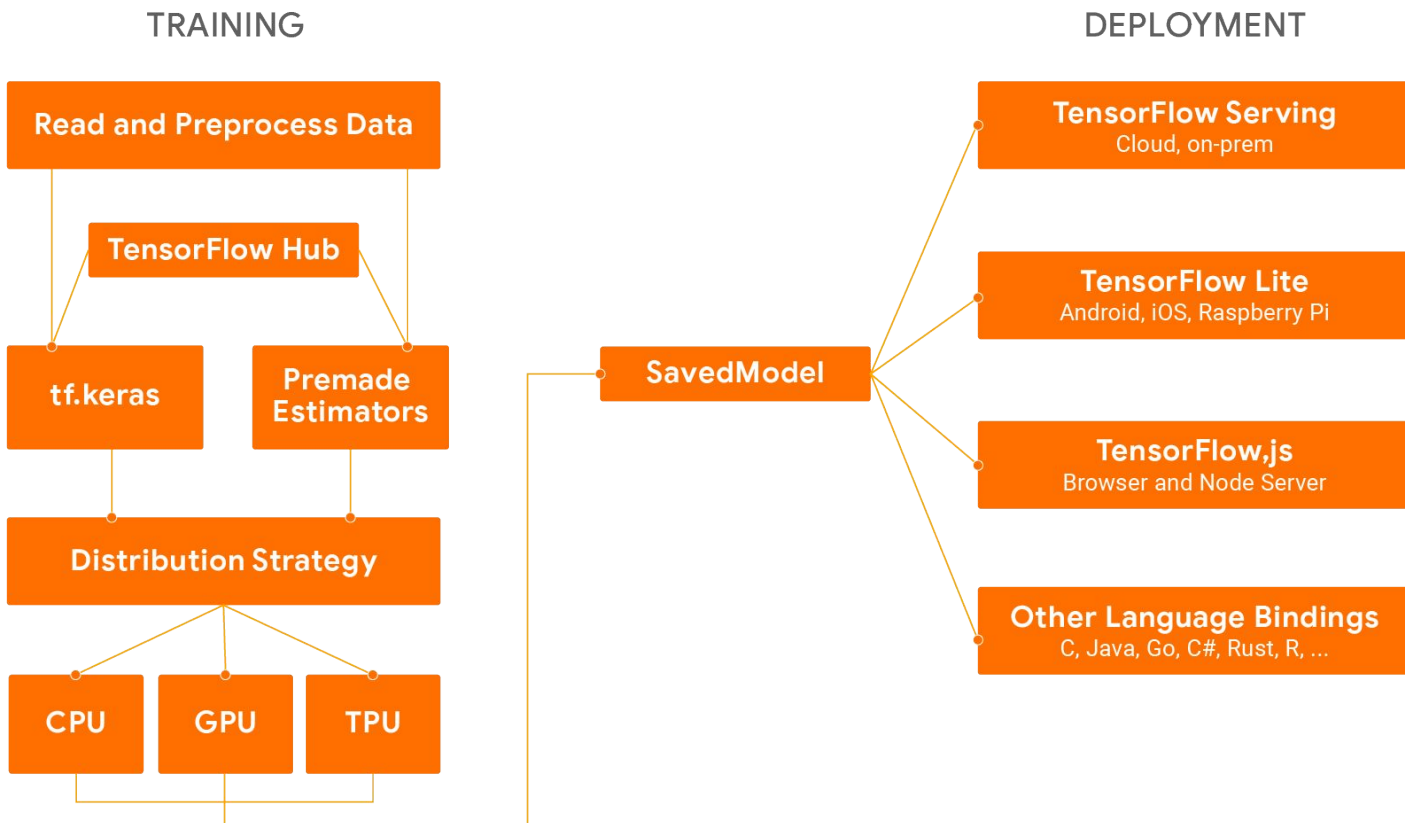
- **>1,800 contributors worldwide**
- Apache 2.0 license
- Released by Google in 2015

## TensorFlow 2.0

- Easier to learn and use
- For **beginners** and **experts**
- Available today



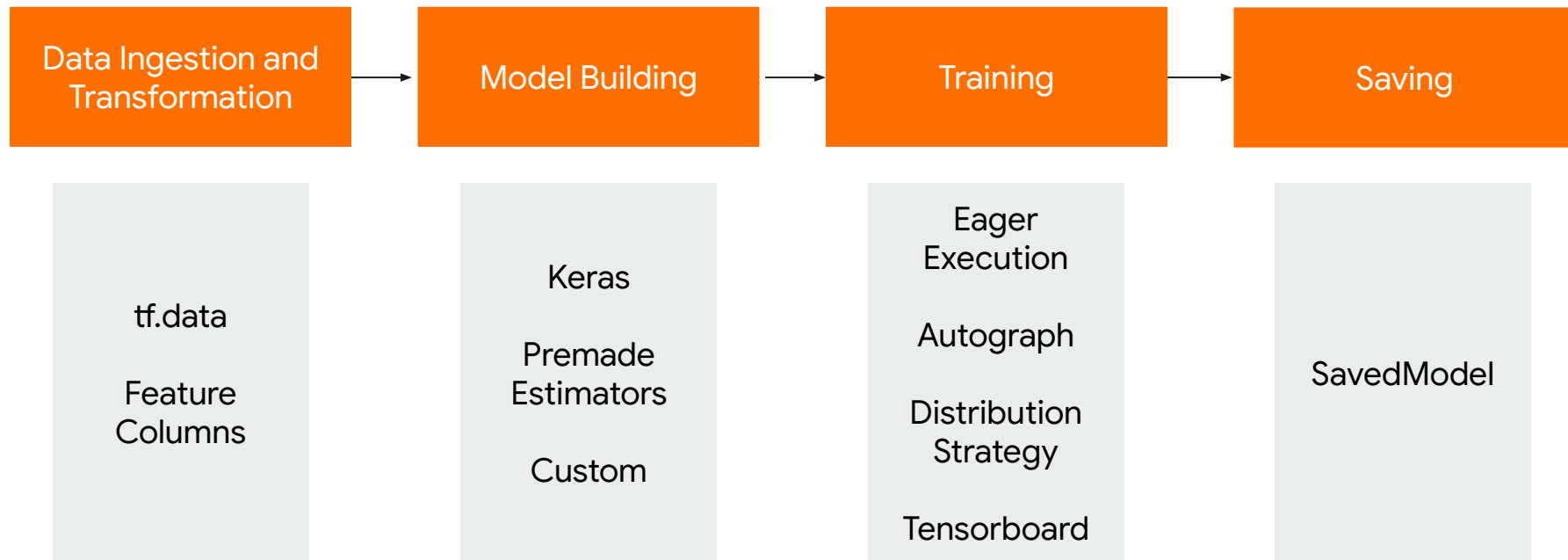
# TensorFlow architecture



Source: [Getting Started with TensorFlow 2.0 presentation Google I/O 2019](#)



# Training Workflow







**TensorFlow**  
**1.x**

# Tensor datatype



A Tensor has a name, type a rank and a shape

- The **name** identifies uniquely the object in the computational graph
- The **type** specify the data type, for example `tf.float32` or `tf.int8`.
- The **rank** is simply the number of dimensions of the tensor: a scalar has rank 0, a vector has rank 1 and so on.
- The **shape** is the number of elements in each dimensions: a scalar has a shape of `()`, a vector a shape of `(d0)`, a matrix shape of `(d0, d1)` and so on (with `d0` and `d1` positive integers).

**NOTE:** the dimension **None** is allowed and indicates an unknown dimension.

# Main Tensor types



- **tf.Variable\*** - will change during training; a variable maintains state in the graph across calls to **run()**
- **tf.constant\*\*** - will remain constant
- **tf.placeholder\*\*\*** - will contain values that will not change during training but that can change between runs (training and validation datasets for example)

\* <https://www.tensorflow.org/guide/variable>

\*\* [https://www.tensorflow.org/api\\_docs/python/tf/constant](https://www.tensorflow.org/api_docs/python/tf/constant)

\*\*\* Only in 1.X [https://www.tensorflow.org/versions/r1.14/api\\_docs/python/tf/placeholder](https://www.tensorflow.org/versions/r1.14/api_docs/python/tf/placeholder)

## TF 1.x - development steps



- **Computational Graph construction**
- Session object creation
- Using the resources
- Releasing resources

# Example of graph construction



```
A = tf.constant(1, dtype = tf.float32)
B = tf.constant(2, dtype = tf.float32)
y = tf.add(A,B)
print(y)
```

```
>> Tensor("Add:0", shape=(), dtype=float32)
```

★ No evaluation has yet taken place

# Example of graph evaluation



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

```
>> 3.0
```

# TF 1.x - Graph evaluation



- **tf.Session** is a class that TF provides to represent a connection between the Python code and the C++ runtime
  - **tf.Session** is the only class that is able to communicate directly with the hardware
1. Acquire the resources instantiating a **tf.Session**
  2. Use resources (**sess.run()**)
  3. Release the resources with **sess.close()**

# TF1.x - Graph evaluation



```
sess = tf.Session()  
sess.run(...)  
sess.close()
```

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



# TF 1.x - Computational Graph - The hardcore way



```
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])

W1 = tf.Variable(tf.random_normal([784, 300], stddev=0.03), name='W1')
b1 = tf.Variable(tf.random_normal([300]), name='b1')
W2 = tf.Variable(tf.random_normal([300, 10], stddev=0.03), name='W2')
b2 = tf.Variable(tf.random_normal([10]), name='b2')
hidden_out = tf.add(tf.matmul(x, W1), b1)
hidden_out = tf.nn.relu(hidden_out)
y_ = tf.nn.softmax(tf.add(tf.matmul(hidden_out, W2), b2))
y_clipped = tf.clip_by_value(y_, 1e-10, 0.9999999)
cross_entropy = -tf.reduce_mean(tf.reduce_sum(y * tf.log(y_clipped)
                                              + (1 - y) * tf.log(1 - y_clipped), axis=1))

optimiser =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cross_entropy)
init_op = tf.global_variables_initializer()
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

# TF 1.x - The slightly less hardcore way



```
input = tf.placeholder(tf.float32, [None,  
image_size*image_size])  
labels = tf.placeholder(tf.float32, [None, labels_size])  
  
hidden = tf.layers.dense(inputs=input, units=1024,  
activation=tf.nn.relu)  
output = tf.layers.dense(inputs=hidden, units=labels_size)
```

# TF 1.x - Training Loop



```
with tf.Session() as sess:  
    sess.run(init_op)  
    for epoch in range(epochs):  
        avg_cost = 0  
        sess.run(optimiser, feed_dict={x:  
x_train, y: y_train})
```



# TensorFlow

## 2.x

# What is new in TF2.0

- Easy model building with **Keras** and **eager execution** (activated by default in TF2.0).
- Robust model **deployment** in production on any platform.
- Powerful experimentation for research.
- Simplifying the API by **cleaning up deprecated APIs** and reducing duplication (relevant in case you have code developed in TensorFlow 1.X and you need to convert it)
- Load your data using **tf.data**. Training data is read using input pipelines which are created using tf.data.
- Build, train and validate your model with **tf.keras**, or use **Premade Estimators**.
- **TensorFlow Hub**.
- **Run and debug with eager execution**, then use **tf.function** for the benefits of graphs.
- Use Distribution Strategies for distributed training.
- hardware accelerators like CPUs, GPUs, and TPUs; you can enable training workloads to be distributed to single-node/multi-accelerator as well as multi-node/multi-accelerator configurations, including TPU Pods.
- Export to **SavedModel**. TensorFlow will standardize on SavedModel as an interchange format for TensorFlow Serving, TensorFlow Lite, TensorFlow.js, TensorFlow Hub, and more.
- **Tensorflow Datasets**

# You can use 2.0 now



```
!pip install tensorflow
```

```
!pip install tensorflow_gpu
```

★ TensorFlow 2 packages  
require a pip version >19.0.

★ This version support GPUs

## Google Colab trick

```
%tensorflow_version 2.x  
import tensorflow as tf
```

★ 1.x and 2.x are available

# All packages



TensorFlow 2 packages are available

- `tensorflow` —Latest stable release for CPU-only
- `tensorflow-gpu` —Latest stable release with [GPU support](#) (*Ubuntu and Windows*)
- `tf-nightly` —Preview build (*unstable*). Ubuntu and Windows include [GPU support](#).

Older versions of TensorFlow

For the 1.15 release, CPU and GPU support are included in a single package:

- `tensorflow==1.15rc2` —1.x release. Ubuntu and Windows include [GPU support](#)






For releases 1.14 and older, CPU and GPU packages are separate:

- `tensorflow==1.14` —Release for CPU-only
- `tensorflow-gpu==1.14` —Release with [GPU support](#) (*Ubuntu and Windows*)

# tensorflow-gpu version dependencies

## Software requirements

The following NVIDIA® software must be installed on your system:

- [NVIDIA® GPU drivers](#)  —CUDA 10.0 requires 410.x or higher.
- [CUDA® Toolkit](#)  —TensorFlow supports CUDA 10.0 (TensorFlow >= 1.13.0)
- [CUPTI](#)  ships with the CUDA Toolkit.
- [cuDNN SDK](#)  (>= 7.4.1)
- *(Optional)* [TensorRT 5.0](#)  to improve latency and throughput for inference on some models.



# Check the installed version



```
import tensorflow as tf  
print(tf.__version__)
```

★ When you change TF version  
you need to restart the runtime in  
Google Colab



# TensorFlow 2.0

## Usability

---

- `tf.keras` as the recommended high-level API.
- Eager execution by default.

```
>>> tf.add(2, 3)
<tf.Tensor: id=2, shape=(), dtype=int32, numpy=5>
```



# TensorFlow 2.0

## Clarity

---

- Remove duplicate functionality
- Consistent, intuitive syntax across APIs
- Compatibility throughout the TensorFlow ecosystem



# TensorFlow 2.0

## Flexibility

---

- Full lower-level API.
- Internal ops accessible in `tf.raw_ops`
- Inheritable interfaces for variables, checkpoints, layers.

# How to study TF2.0



[https://github.com/michelucci/TensorFlow20-Notes/blob/master/howto\\_study\\_TF20.md](https://github.com/michelucci/TensorFlow20-Notes/blob/master/howto_study_TF20.md)

## Basics

---

1. Tensors
2. Computational graphs basics (this is relevant when you are going to study how `@tf.function` works and how it creates a graph in the background. Unless you understand how Computational graphs are working, you will not really understand how `@tf.function` is working)
3. Eager Execution (basics and subtelties)
4. `tf.Variable` (aka no need to initialize them anymore in eager mode)
5. Keras basics ( `Sequential()` models, `.compile()` and `.fit()` calls)
6. Keras functional APIs
7. `tf.data.Dataset` and data processing pipelines

# Use a built-in training loop...



```
model.fit(x_train, y_train, epochs=5)
```

# Or use a custom one...



```
model = MyModel()
```

```
with tf.GradientTape() as tape:
```

```
    logits = model(images)
```

```
    loss_value = loss(logits, labels)
```

```
grads = tape.gradient(loss_value, model.trainable_variables)
```

```
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

---

# Eager execution



# Eager execution introduction



TensorFlow's **eager execution** is an imperative programming environment that evaluates operations immediately.

Eager execution is a flexible machine learning platform for research and experimentation, providing:

- *An intuitive interface*—Structure your code naturally and use Python data structures. Quickly iterate on small models and small data.
- *Easier debugging*—Call ops directly to inspect running models and test changes. Use standard Python debugging tools for immediate error reporting.
- *Natural control flow*—Use Python control flow instead of graph control flow, simplifying the specification of dynamic models.

# Eager execution - setup



Eager execution is enable in TF2.x by default.

```
import tensorflow as tf
tf.executing_eagerly()
```

```
>>> True
```

Operations returns value immediately

```
x = [[2.]]
m = tf.matmul(x, x)
print("hello, {}".format(m))
```

```
>>> hello, [[4.]]
```

# @tf.function - make it faster

```
lstm_cell = tf.keras.layers.LSTMCell(10)
```

```
def fn(input, state):  
    return lstm_cell(input, state)
```

```
input = tf.zeros([10, 10]); state = [tf.zeros([10, 10])] * 2  
lstm_cell(input, state); fn(input, state) # warm up
```

```
# benchmark
```

```
timeit.timeit(lambda: lstm_cell(input, state), number=10) # 0.03
```

# @tf.function - make it faster

```
lstm_cell = tf.keras.layers.LSTMCell(10)
```

```
@tf.function
```

```
def fn(input, state):
```

```
    return lstm_cell(input, state)
```

```
input = tf.zeros([10, 10]); state = [tf.zeros([10, 10])] * 2
```

```
lstm_cell(input, state); fn(input, state) # warm up
```

```
# benchmark
```

```
timeit.timeit(lambda: lstm_cell(input, state), number=10) # 0.03
```

```
timeit.timeit(lambda: fn(input, state), number=10) # 0.004
```

—

Keras



KERAS: high level API



# Keras.io (Reference implementation)



```
import keras
```

## TensorFlow implementation

```
from tensorflow import keras
```

★ TensorFlow's implementation (a superset, built-in to TF)

# For beginners



## Model definition

```
model = tf.keras.models.Sequential(...)
```

## Model compilation

```
model.compile(...)
```

## Model fitting

```
model.fit(...)
```



# Sequential() model



```
model = tf.keras.models.Sequential([  
    tf.keras.layers.Flatten(),  
    tf.keras.layers.Dense(512, activation='relu'),  
    tf.keras.layers.Dropout(0.2),  
    tf.keras.layers.Dense(10, activation='softmax')  
])
```

**A sequence of layers stacked one after the other**

# Sequential() model - beginners



```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

# Sequential() model - experts



```
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__(name='my_model')
        self.dense_1 = layers.Dense(32, activation='relu')
        self.dense_2 = layers.Dense(num_classes, activation='sigmoid')

    def call(self, inputs):
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

---

# Sequential() model

# Sequential() model



The Sequential model is a linear stack of layers. You can create a Sequential model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

```
model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

# Sequential() model



You can also simply add layers via the `.add()` method

```
model = Sequential()  
model.add(Dense(32, input_dim=784))  
model.add(Activation('relu'))
```

# Compilation



Before training a model, you need to configure the learning process

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

# Training



Keras models are trained on Numpy arrays of input data and labels.

```
model.fit(x_train, y_train, epochs=5)
```



---

# Layers in Keras

# keras.layers



## Most used

```
from keras.layers import Dense, Conv2D,  
MaxPooling2D, Dropout, Flatten
```

```
keras.layers.Dense
```

```
keras.layers.Conv2D
```

```
keras.layers.MaxPooling2D
```

```
keras.layers.Dropout
```

```
keras.layers.Flatten
```

# Example of custom layer



```
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()
        w_init = tf.random_normal_initializer()
        self.w = tf.Variable(initial_value=w_init(shape=(input_dim, units),
                                                    dtype='float32'),
                              trainable=True)

        b_init = tf.zeros_initializer()
        self.b = tf.Variable(initial_value=b_init(shape=(units,),
                                                    dtype='float32'),
                              trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
```

---

# Custom callback classes

# Introduction



A custom callback is a powerful tool to customize the behavior of a Keras model during training, evaluation, or inference, including reading/changing the Keras model.

“In Keras, **Callback** is a python class meant to be subclassed to provide specific functionality, with a set of methods called at various stages of training (including batch/epoch start and ends), testing, and predicting. “

# Custom Callback methods



- `on_train_begin`: Called at the beginning of training
- `on_train_end`: Called at the end of training
- `on_epoch_begin`: Called at the start of an epoch
- `on_epoch_end`: Called at the end of an epoch
- `on_batch_begin`: Called right before processing a batch
- `on_batch_end`: Called at the end of a batch

# Example of Callback



```
class CustomCallback1(keras.callbacks.Callback):  
    def on_epoch_end(self, epoch, logs={}):  
        print ("Just finished epoch", epoch)  
        print (logs)  
        return
```

## Callback during training

```
CC1 = CustomCallback1()  
model.fit(train_images, train_labels, epochs = 2, validation_data =  
          (test_images, test_labels), callbacks = [CC1])
```

```
>>> {'val_loss': 0.2545496598124504, 'val_acc': 0.9244, 'loss':  
0.05098680723309517, 'acc': 0.9878}
```

# Example of Callback



```
class CustomCallback3(keras.callbacks.Callback):  
    def on_epoch_end(self, epoch, logs={}):  
        print ("Just finished epoch", epoch)  
        print ('Loss evaluated on the validation dataset  
= ', logs.get('val_loss'))  
        print ('Accuracy reached is', logs.get('acc')) return
```

```
>>> Just finished epoch 0
```

```
>>> Loss evaluated on the validation dataset = 0.2546206972360611
```

## Frequency of logging

```
if (epoch % 10 == 0):
```



# Pre-ready callbacks

## Classes

`class BaseLogger` : Callback that accumulates epoch averages of metrics.

`class CSVLogger` : Callback that streams epoch results to a csv file.

`class Callback` : Abstract base class used to build new callbacks.

`class EarlyStopping` : Stop training when a monitored quantity has stopped improving.

`class History` : Callback that records events into a `History` object.

`class LambdaCallback` : Callback for creating simple, custom callbacks on-the-fly.

`class LearningRateScheduler` : Learning rate scheduler.

`class ModelCheckpoint` : Save the model after every epoch.

`class ProgbarLogger` : Callback that prints metrics to stdout.

`class ReduceLR0nPlateau` : Reduce learning rate when a metric has stopped improving.

`class RemoteMonitor` : Callback used to stream events to a server.

`class TensorBoard` : Enable visualizations for TensorBoard.

`class TerminateOnNaN` : Callback that terminates training when a NaN loss is encountered.



---

# Keras functional APIs

# Keras Functional APIs - an introduction



The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, models with shared layers or for example residual neural networks.

# Example of Keras Functional APIs



## Model definition

```
from keras.layers import Input, Dense
from keras.models import Model

inputs = Input(shape=(784,))
output_1 = Dense(64, activation='relu')(inputs)
output_2 = Dense(64, activation='relu')(output_1)
predictions = Dense(10, activation='softmax')(output_2)
model = Model(inputs=inputs, outputs=predictions)
```


# Example of Keras Functional APIs



Compiling and training remain the same

```
model.compile(optimizer='rmsprop',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])  
model.fit(data, labels)
```

# Keras Functional APIs - everything is a layer



With the functional API, it is easy to reuse trained models: you can treat any model as if it were a layer, by calling it on a tensor. Note that by calling a model you aren't just reusing the architecture of the model, you are also reusing its weights.

# Keras Functional APIs - everything is a layer



This can allow, for instance, to quickly create models that can process sequences of inputs. You could turn an image classification model into a video classification model, in just one line.

```
from keras.layers import TimeDistributed

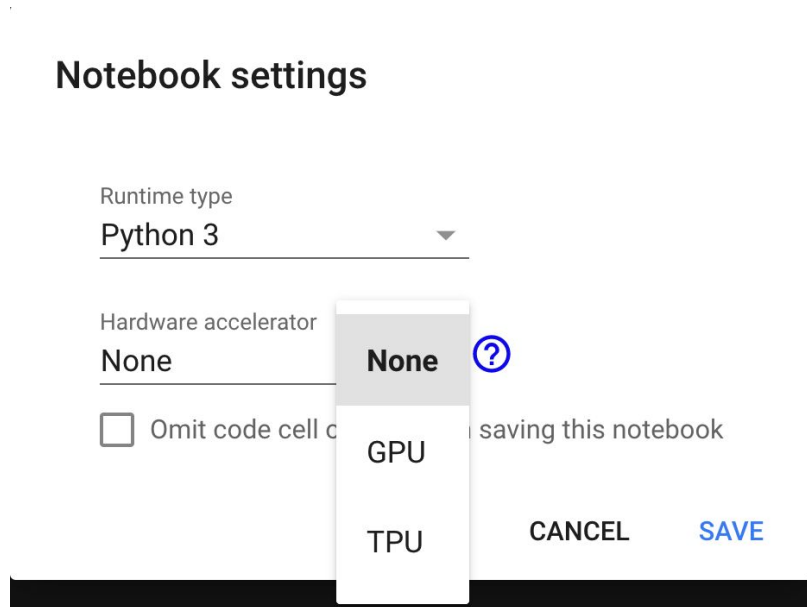
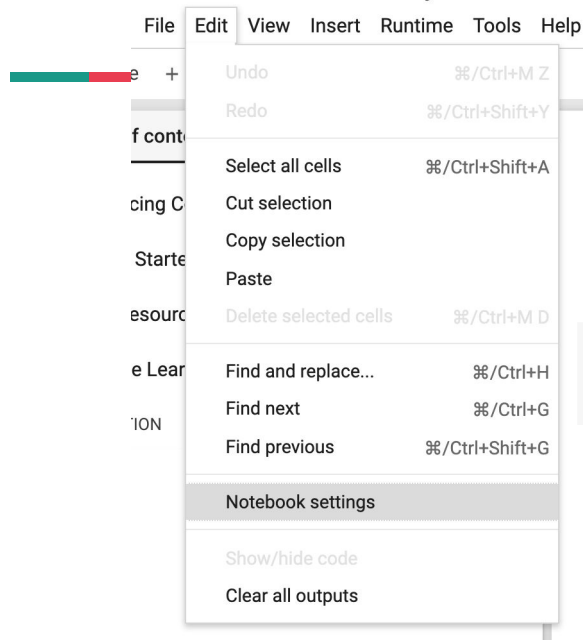
input_sequences = Input(shape=(20, 784))
processed_sequences = TimeDistributed(model)(input_sequences)
```



---

# Hardware acceleration

# Using hardware acceleration in Google Colab



Changing the Hardware accelerator setting will make restarting the runtime necessary

# Testing presence of a GPU



```
print(tf.test.is_gpu_available())

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found.')
print('Found GPU at: {}'.format(device_name))
```

# Putting operations on a device

```
with tf.device('/gpu:0'):
```

```
    # Your code here
```

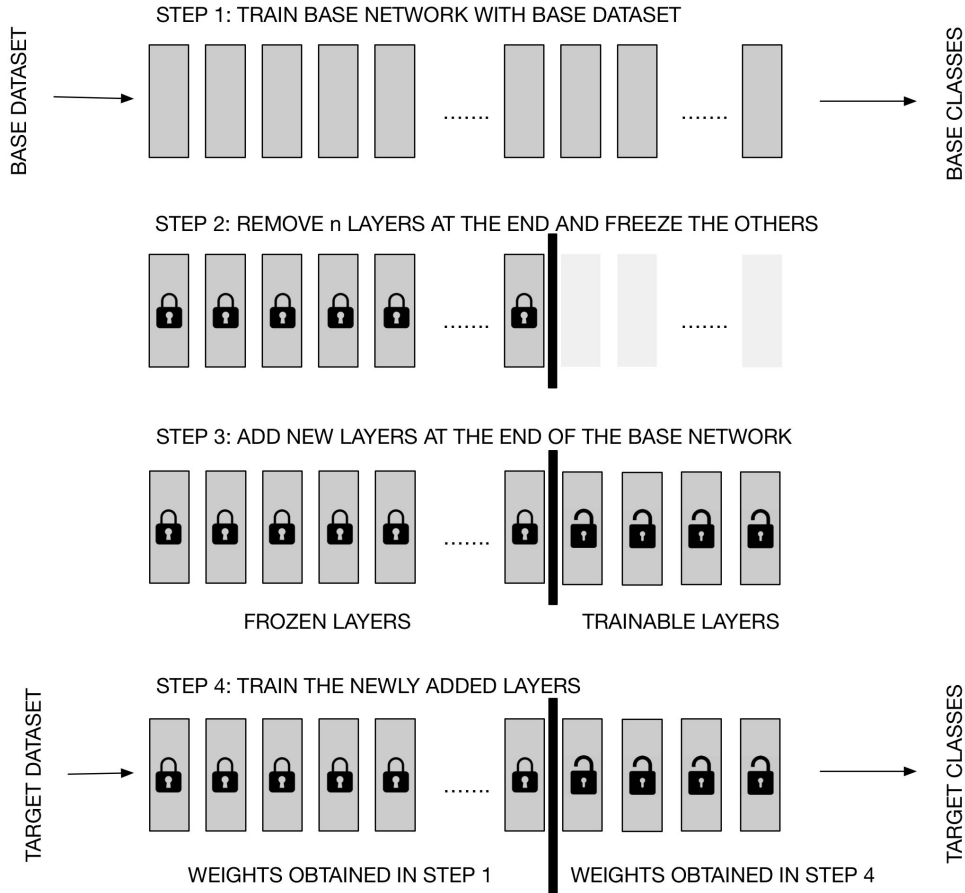
```
with tf.device('/cpu:0'):
```

```
    # Your code here
```

---

# Transfer Learning

# Transfer Learning - an introduction

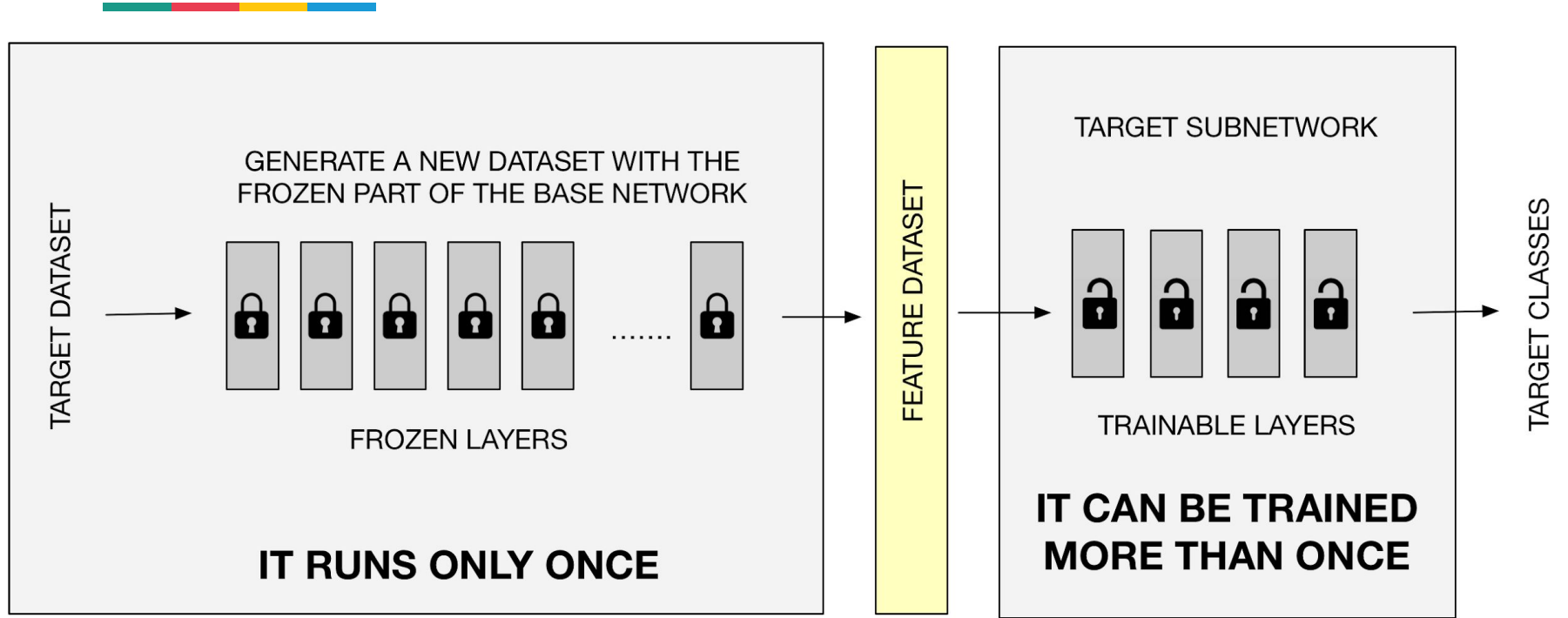


# Transfer Learning in practice

```
base_model=vgg16.VGG16(include_top=False, weights='imagenet')
x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x)
preds=Dense(1,activation='softmax')(x)
model=Model(inputs=base_model.input,outputs=preds)
```

...

```
for layer in model.layers[:20]:
    layer.trainable=False
for layer in model.layers[20:]:
    layer.trainable=True
```



SOURCE: "Advanced Applied Deep Learning - Convolutional Neural Networks and Object Detection" by U. Michelucci - <https://www.apress.com/gp/book/9781484249758>





# APPENDIX



# TOELT

Our network





# TOELT - Our network

