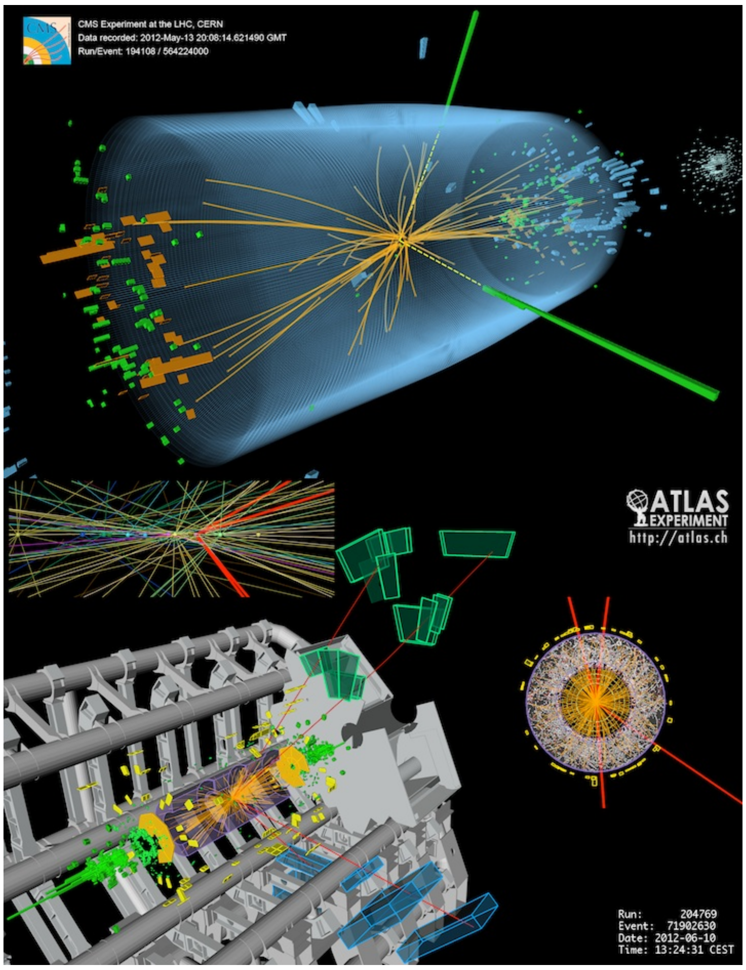
Searching for the Higgs Boson

The Standard Model is a theory in particle physics that describes some of the most basic forces of nature. One fundamental particle, the Higgs boson, is what accounts for the *mass* of matter. First theorized in the 1964, the Higgs boson eluded observation for almost fifty years. In 2012 it was finally observed experimentally at the Large Hadron Collider. These experiments produced millions of gigabytes of data.

Large and complicated datasets like these are where deep learning excels. In this notebook, we'll build a Wide and Deep neural network to determine whether an observed particle collision produced a Higgs boson or not.


CMS Experiment at the LHC, CERN
Data recorded: 2012-May-13 20:08:14.621490 GMT
Run/Event: 194108 / 564224000
ATLAS EXPERIMENT
http://atlas.ch
Run: 204769
Event: 71902630
Date: 2012-06-10
Time: 13:24:31 CEST

The Collision Data

The collision of protons at high energy can produce new particles like the Higgs boson. These particles can't be directly observed, however, since they decay almost instantly. So to detect the presence of a new particle, we instead obesrve the behavior of the particles they decay into, their "decay products".

The *Higgs* dataset contains 21 "low-level" features of the decay products and also 7 more "high-level" features derived from these.

In [ ]:

```
# TensorFlow
import tensorflow as tf
print("Tensorflow version " + tf.__version__)

import pandas as pd
```

Note 1 😀

Let us inspect a couple of records from the training dataset

Useful links:
TFRecord and tf.train.Example: https://www.tensorflow.org/tutorials/load_data/tfrecord
How to inspect the structure of a TFRecord file: https://stackoverflow.com/questions/55861893/how-to-inspect-the-structure-of-a-tfrecord-file-in-tensorflow-1-13

In [ ]:

```
iteration, records = 0, 2
for record in tf.compat.v1.python_io.tf_record_iterator("../input/higgs-boson/training/s
hard_00.tfrecord"):
    iteration+=1
    print('==== Record ', iteration, ' ====')
    print(tf.train.Example.FromString(record))

    if iteration == records:
        break
```

Note 1 continued 😀

We can also try and convert the features and label from TFRecord format to uint6 or float16 datatypes

Note that without knowing the exact initial datatype before it was encoded as TFRecord, decoding it will not be accurate.

For example, below when we create the feature_tensor object, its size will depend on the datatype we will use to decode: tf.unit8 will give us 122 features while tf.float16 will return 61 features

In [ ]:

```
#Useful dataframe that will store our features and label currently in a tfrecord
feature_label_df = pd.DataFrame()
feature_label_df['label'] = None

# a simple limit on the number of records we would like to inspect, feel free to adjust
iteration, records = 0, 100

#Let's iterate over each tfrecord
for record in tf.compat.v1.python_io.tf_record_iterator("../input/higgs-boson/training/s
hard_00.tfrecord"):
    iteration+=1
    #print('==== Record ', iteration, ' ====')
    #print(tf.train.Example.FromString(record))

    #parse each tfrecord
    example = tf.train.Example()
    example.ParseFromString(record)

    #looking at the output of the cell above, we are interested in the features.bytes_lis
t entry
    string = example.features.feature['features'].bytes_list.value[0]

    #convert it to an int or float, see note above
    feature_tensor = tf.io.decode_raw(string, tf.float16) #tf.uint8

    #print('==== Features ', iteration, ' ====')
    #print(feature_tensor)

    #append current decoded record to our hepful dataframe
    feature_label_df = feature_label_df.append(pd.DataFrame(feature_tensor).transpose())
.reset_index(drop=True)

    ##looking at the output of the cell above, we are alsoe interested in the label.float
_list field as our target, i.e presence or not of Higgs boson "decay products"
```

```
    label = example.features.feature['label'].float_list.value[0]
    #print('==== Label ', iteration, ' ====')
    #print(label)
    feature_label_df.loc[iteration-1, 'label'] = label

    if iteration == records:
        break
```

In [ ]:

```
#Inspect our helpful dataframe
feature_label_df
```

In [ ]:

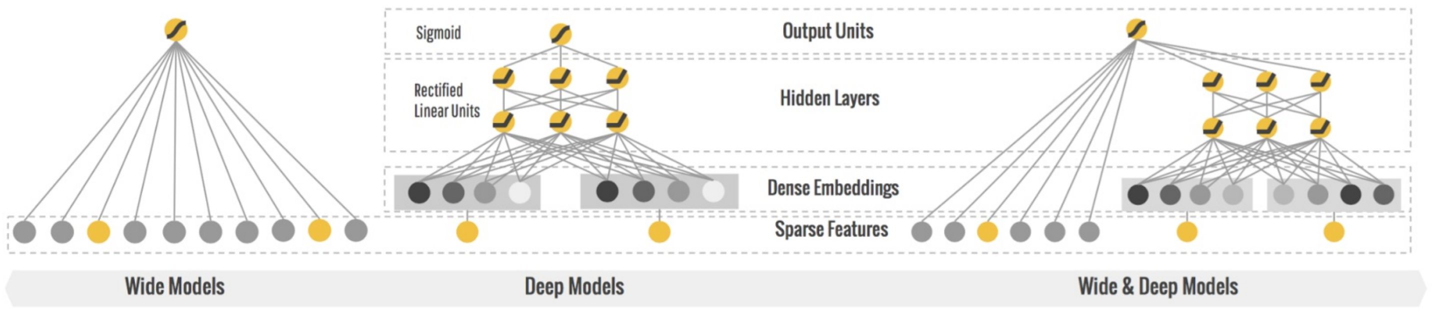
```
#Inspect class balance
feature_label_df.label.value_counts().sort_index()
```

Wide and Deep Neural Networks

A *Wide and Deep* network trains a linear layer side-by-side with a deep stack of dense layers. Wide and Deep networks are often effective on tabular datasets.[1]

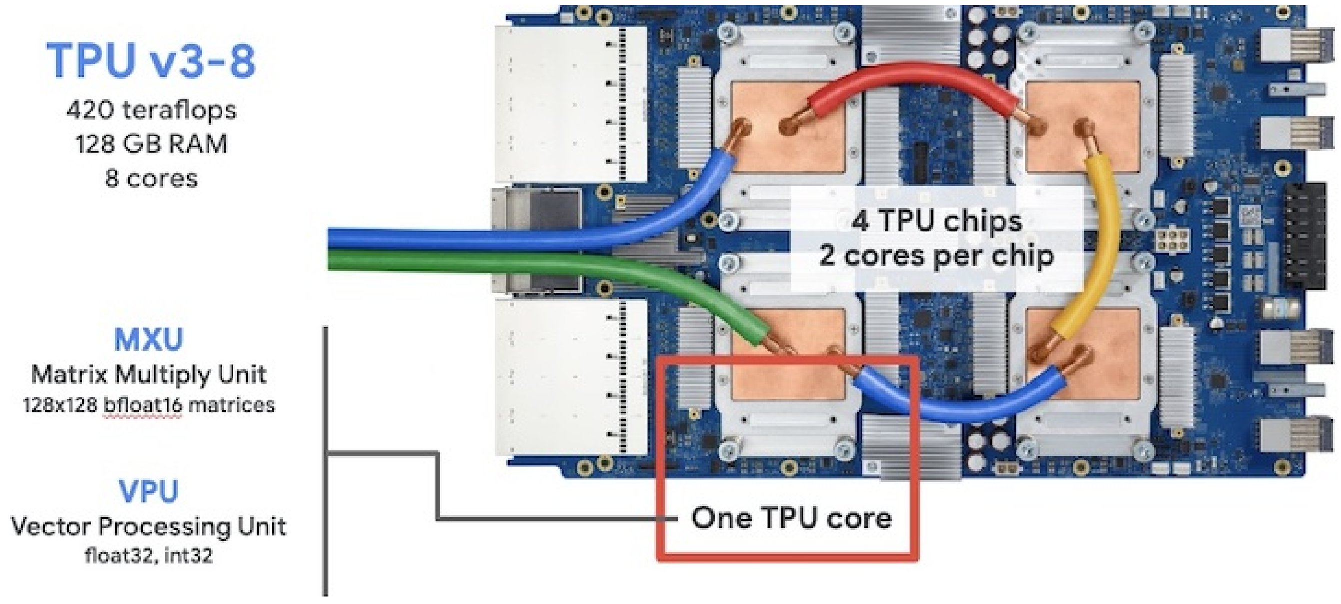
Note 2 😀

The differences of all 3 network types



1. **In the original implementation, categorical features were one-hot encoded and crossed to produce the interaction features. This "wide" dataset was used with the linear component. For the deep component, the categories were encoded into a much narrower embedding layer.↩**

Both the dataset and the model are much larger than what we used in the course. To speed up training, we'll use Kaggle's Tensor Processing Units (TPUs), an accelerator ideal for large workloads.



We've collected some hyperparameters here to make experimentation easier. Fork this notebook by [clicking here] to try it yourself!

In [ ]:

```
# Model Configuration
UNITS = 2 ** 11  # 2048
ACTIVATION = 'relu'
DROPOUT = 0.1

# Training Configuration
BATCH_SIZE_PER_REPLICA = 2 ** 11 # powers of 128 are best
```

The next few sections set up the TPU computation, data pipeline, and neural network model. If you'd just like to see the results, feel free to skip to the end!

Setup

In addition to our imports, this section contains some code that will connect our notebook to the TPU and create a distribution strategy. Each TPU has eight computational cores acting independently. With a distribution strategy, we define how we want to divide up the work between them.

In [ ]:

```
# TF 2.3 version
# Detect and init the TPU
# try: # detect TPUs
#     tpu = tf.distribute.cluster_resolver.TPUClusterResolver.connect() # TPU detection
#     strategy = tf.distribute.TPUStrategy(tpu)
# except ValueError: # detect GPUs
#     strategy = tf.distribute.get_strategy() # default strategy that works on CPU and si
ngle GPU
# print("Number of accelerators: ", strategy.num_replicas_in_sync)

# TF 2.2 version
try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
except ValueError:
    strategy = tf.distribute.get_strategy()  # default strategy that works on CPU and sing
le GPU

print("Number of accelerators: ", strategy.num_replicas_in_sync)
```

Note 3 😀

Let us inspect `strategy` object

In [ ]:

```
strategy.__dir__()
```

In [ ]:

```
# Plotting
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = [12,7]

# Matplotlib defaults
plt.style.use('seaborn-whitegrid')
plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
       titleweight='bold', titlesize=18, titlepad=10)
```

In [ ]:

```
# Data
from kaggle_datasets import KaggleDatasets
from tensorflow.io import FixedLenFeature     #Configuration for parsing a fixed-length in
put feature.
AUTO = tf.data.experimental.AUTOTUNE          #See Note 5 below 😀
```

In [ ]:

```
# Model
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import callbacks
```

Notice that TensorFlow now detects eight accelerators. Using a TPU is a bit like using eight GPUs at once.

Load Data

The dataset has been encoded in a binary file format called *TFRecords*. These two functions will parse the TFRecords and build a TensorFlow `tf.data.Dataset` object that we can use for training.

In [ ]:

```
def make_decoder(feature_description):

    def decoder(example):
        example = tf.io.parse_single_example(example, feature_description)
        features = tf.io.parse_tensor(example['features'], tf.float32)
        features = tf.reshape(features, [28])
        label = example['label']
        return features, label

    return decoder
```

Note 4 😀

`make_decoder(feature_description)` **returns a function, in this case** `decoder(example)` **which takes** `example` **as input when called. This concept is also used in decorators in Python.**

Here is a beginner friendly article on them: https://realpython.com/inner-functions-what-are-they-good-for/#closures-and-factory-functions

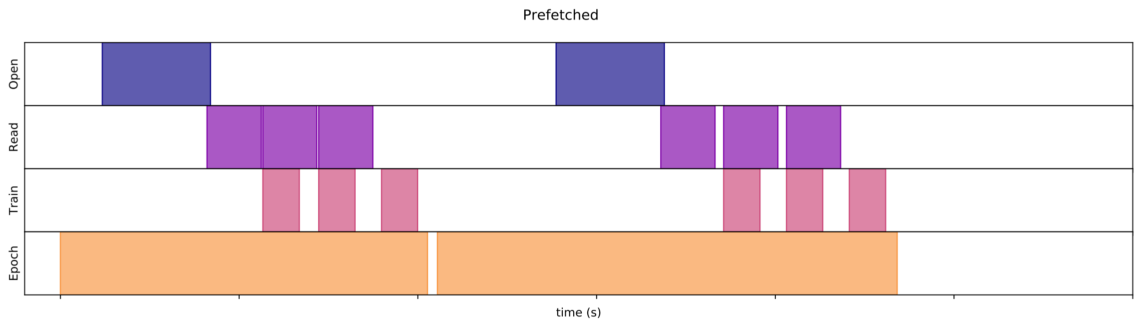
In [ ]:

```
def load_dataset(filenames, decoder, ordered=False):
    AUTO = tf.data.experimental.AUTOTUNE
    ignore_order = tf.data.Options()

    if not ordered:
        ignore_order.experimental_deterministic = False

    dataset = (
        tf.data
        .TFRecordDataset(filenames, num_parallel_reads=AUTO)
        .with_options(ignore_order)
        .map(decoder, AUTO)
    )

    return dataset
```

Note 5 😀

`AUTO = tf.data.experimental.AUTOTUNE`

GPUs and TPUs can radically reduce the time required to execute a single training step. Achieving peak performance requires an efficient input pipeline that delivers data for the next step before the current step has finished. The tf.data API helps to build flexible and efficient input pipelines.
https://www.tensorflow.org/guide/data_performance



The tf.data API provides the tf.data.Dataset.prefetch transformation. It can be used to decouple the time when data is produced from the time when data is consumed. In particular, the transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of the time they are requested. The number of elements to prefetch should be equal to (or possibly greater than) the number of batches consumed by a single training step. You could either manually tune this value, or set it to tf.data.experimental.AUTOTUNE which will prompt the tf.data runtime to tune the value dynamically at runtime.

Note that the prefetch transformation provides benefits any time there is an opportunity to overlap the work of a "producer" with the work of a "consumer."

In [ ]:

```
dataset_size = int(11e6)
validation_size = int(5e5)
training_size = dataset_size - validation_size

# For model.fit
batch_size = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
steps_per_epoch = training_size // batch_size
validation_steps = validation_size // batch_size

# For model.compile
steps_per_execution = steps_per_epoch
```

In [ ]:

```
print(dataset_size, validation_size, training_size, batch_size, steps_per_epoch, validati
on_steps, steps_per_execution)
```

In [ ]:

```
feature_description = {
    'features': FixedLenFeature([], tf.string),
    'label': FixedLenFeature([], tf.float32),
}

decoder = make_decoder(feature_description)
```

In [ ]:

```
decoder
```

See Note 4 😀 above

In [ ]:

```
data_dir = KaggleDatasets().get_gcs_path('higgs-boson')
train_files = tf.io.gfile.glob(data_dir + '/training' + '/*.tfrecord')
valid_files = tf.io.gfile.glob(data_dir + '/validation' + '/*.tfrecord')
```

In [ ]:

```
ds_train = load_dataset(train_files, decoder, ordered=False)

ds_train = (
    ds_train
    .cache()
    .repeat()
    .shuffle(2 ** 19)
    .batch(batch_size)
    .prefetch(AUTO)
)

ds_valid = load_dataset(valid_files, decoder, ordered=False)

ds_valid = (
    ds_valid
    .batch(batch_size)
    .cache()
    .prefetch(AUTO)
)
```

Model

Now that the data is ready, let's define the network. We're defining the deep branch of the network using Keras's *Functional API*, which is a bit more flexible that the `Sequential` method we used in the course.

In [ ]:

```
def dense_block(units, activation, dropout_rate, l1=None, l2=None):
    def make(inputs):
        x = layers.Dense(units)(inputs)
        x = layers.BatchNormalization()(x)
        x = layers.Activation(activation)(x)
        x = layers.Dropout(dropout_rate)(x)
        return x
    return make
```

In [ ]:

```
with strategy.scope():
    # Wide Network
    wide = keras.experimental.LinearModel()

    # Deep Network
    inputs = keras.Input(shape=[28])
    x = dense_block(UNITS, ACTIVATION, DROPOUT)(inputs)
    x = dense_block(UNITS, ACTIVATION, DROPOUT)(x)
    x = dense_block(UNITS, ACTIVATION, DROPOUT)(x)
    x = dense_block(UNITS, ACTIVATION, DROPOUT)(x)
    x = dense_block(UNITS, ACTIVATION, DROPOUT)(x)
    outputs = layers.Dense(1)(x)
    deep = keras.Model(inputs=inputs, outputs=outputs)

    # Wide and Deep Network
    wide_and_deep = keras.experimental.WideDeepModel(
        linear_model=wide,
        dnn_model=deep,
        activation='sigmoid',
    )
```

In [ ]:

```
wide_and_deep.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['AUC', 'binary_accuracy'],
#     experimental_steps_per_execution=steps_per_execution,
)
```

Training

During training, we'll use the `EarlyStopping` callback as usual. Notice that we've also defined a learning rate schedule. It's been found that gradually decreasing the learning rate over the course of training can improve performance (the weights "settle in" to a minimum). This schedule will multiply the learning rate by `0.2` if the validation loss didn't decrease after an epoch.

In [ ]:

```
early_stopping = callbacks.EarlyStopping(
    patience=2,
    min_delta=0.001,
    restore_best_weights=True,
)

lr_schedule = callbacks.ReduceLROnPlateau(
    patience=0,
    factor=0.2,
    min_lr=0.001,
)
```

In [ ]:

```
history = wide_and_deep.fit(
    ds_train,
    validation_data=ds_valid,
    epochs=50,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping, lr_schedule],
)
```

In [ ]:

```
history_frame = pd.DataFrame(history.history)
history_frame.loc[:, ['loss', 'val_loss']].plot(title='Cross-entropy Loss')
history_frame.loc[:, ['auc', 'val_auc']].plot(title='AUC');
```

Note 6 😀

Let us inspect these 3 models and their architecture**

In [ ]:

```
wide.summary()
```

In [ ]:

```
tf.keras.utils.plot_model(wide, show_shapes=True)
```

In [ ]:

```
deep.summary()
```

In [ ]:

```
tf.keras.utils.plot_model(deep, show_shapes=True)
```

In [ ]:

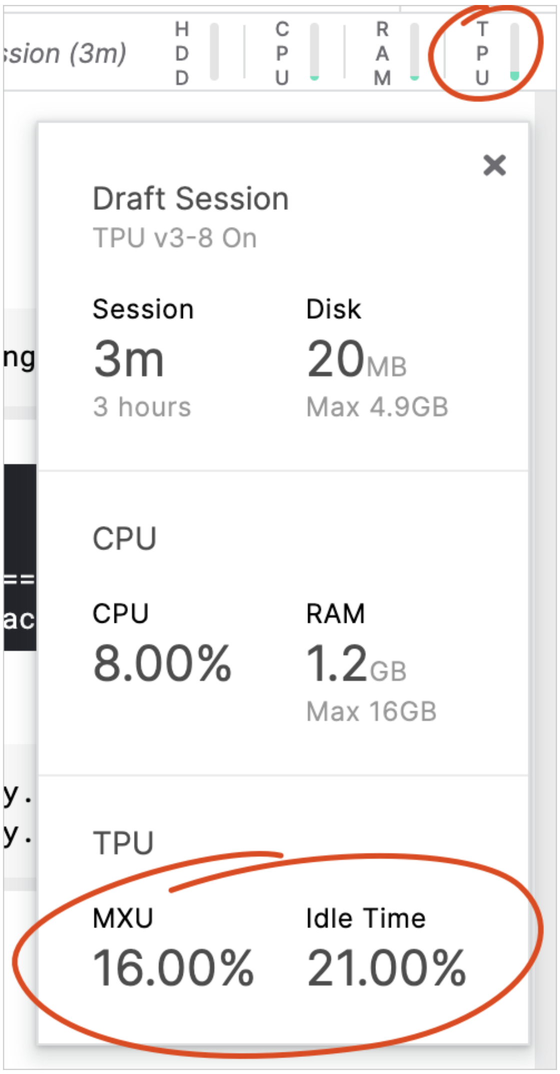
```
wide_and_deep.summary()
```

In [ ]:

```
tf.keras.utils.plot_model(wide_and_deep, show_shapes=True)
```

Note 7 😀

Let us monitor TPU and CPU usage during training


sion (3m)

H D D

C P U

R A M

T P U

Draft Session

TPU v3-8 On

Session

3m

3 hours

Disk

20MB

Max 4.9GB

CPU

CPU

8.00%

RAM

1.2GB

Max 16GB

TPU

MXU

16.00%

Idle Time

21.00%

References

- Baldi, P. et al. *Searching for Exotic Particles in High-Energy Physics with Deep Learning*. (2014) (arXiv)
- Cheng, H. et al. *Wide & Deep Learning for Recommender Systems*. (2016) (arXiv)
- *What Exactly is the Higgs Boson?* Scientific American. (1999) (article)]

Have questions or comments? Visit the Learn Discussion forum to chat with other Learners.