Setup

Before getting started, import the necessary packages:.

```
from __future__ import absolute_import, division, print_function, unicode_literals

try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras.regularizers import l1, l2

print(tf.__version__)
```

```
2.3.0
```

```
!pip install -q git+https://github.com/tensorflow/docs

import tensorflow_docs as tfdocs
import tensorflow_docs.modeling
import tensorflow_docs.plots
```

```
  Building wheel for tensorflow-docs (setup.py) ... done
```

```
from  IPython import display
from matplotlib import pyplot as plt

import numpy as np

import pathlib
import shutil
import tempfile

```

```
logdir = pathlib.Path(tempfile.mkdtemp())/"tensorboard_logs"
shutil.rmtree(logdir, ignore_errors=True)
```

The Higgs Dataset

The goal of this tutorial is not to do particle physics, so don't dwell on the details of the dataset. It contains 11 000 000 examples, each with 28 features, and a binary class label.

```
gz = tf.keras.utils.get_file('HIGGS.csv.gz', 'https://archive.ics.uci.edu/ml/machine-le
```

```
FEATURES = 28
```

The `tf.data.experimental.CsvDataset` class can be used to read csv records directly from a gzip file with no intermediate decompression step.

```
ds = tf.data.experimental.CsvDataset( gz,[float(),]*(FEATURES+1), compression_type="GZI
```

That csv reader class returns a list of scalars for each record. The following function repacks that list of scalars into a (feature_vector, label) pair.

```
def pack_row(*row):
  label = row[0]
  features = tf.stack(row[1:],1)
  return features, label
```

TensorFlow is most efficient when operating on large batches of data.

So instead of repacking each row individually make a new `Dataset` that takes batches of 10000-examples, applies the `pack_row` function to each batch, and then splits the batches back up into individual records:

```
packed_ds = ds.batch(10000).map(pack_row).unbatch()
```

EDA

Have a look at some of the records from this new `packed_ds`.

The features are not perfectly normalized, but this is sufficient for this tutorial.

```
for features,label in packed_ds.batch(1000).take(1):
  print(features[0])
  plt.hist(features.numpy().flatten(), bins = 101)
```

```
tf.Tensor(
[ 0.8692932  -0.6350818   0.22569026  0.32747006 -0.6899932   0.75420225
 -0.24857314 -1.0920639   0.          1.3749921  -0.6536742   0.9303491
  1.1074361   1.1389043  -1.5781983  -1.0469854   0.          0.65792954
 -0.01045457 -0.04576717  3.1019614   1.35376     0.9795631   0.97807616
  0.92000484  0.72165745  0.98875093  0.87667835], shape=(28,), dtype=float32)
```


```
import numpy as np
import matplotlib.pyplot as plt
print(list(ds.take(2).as_numpy_iterator())[0])
print(list(ds.take(2).as_numpy_iterator())[1])

```

```
(1.0, 0.8692932, -0.6350818, 0.22569026, 0.32747006, -0.6899932, 0.75420225, -0.2485
(1.0, 0.9075421, 0.32914728, 0.35941187, 1.4979699, -0.31300953, 1.0955306, -0.55752
```

```
featVals=[features.numpy()[0] for features,label in packed_ds.take(1000)]

t=packed_ds.take(20000)
```

Target Variable

```
t = [label.numpy() for features,label in packed_ds.take(20000)]
print("Values 1:",sum(t))
print("Values 0:",sum(np.equal(t,0)))

```

```
Values 1: 10457.0
Values 0: 9543
```

Predictors

```
t = np.array([features.numpy() for features,label in packed_ds.take(20000)])
print("Min Value:",min(t.flatten()))
print("Max Value:",max(t.flatten()))
```

```
Min Value: -2.960813
Max Value: 16.131908
```

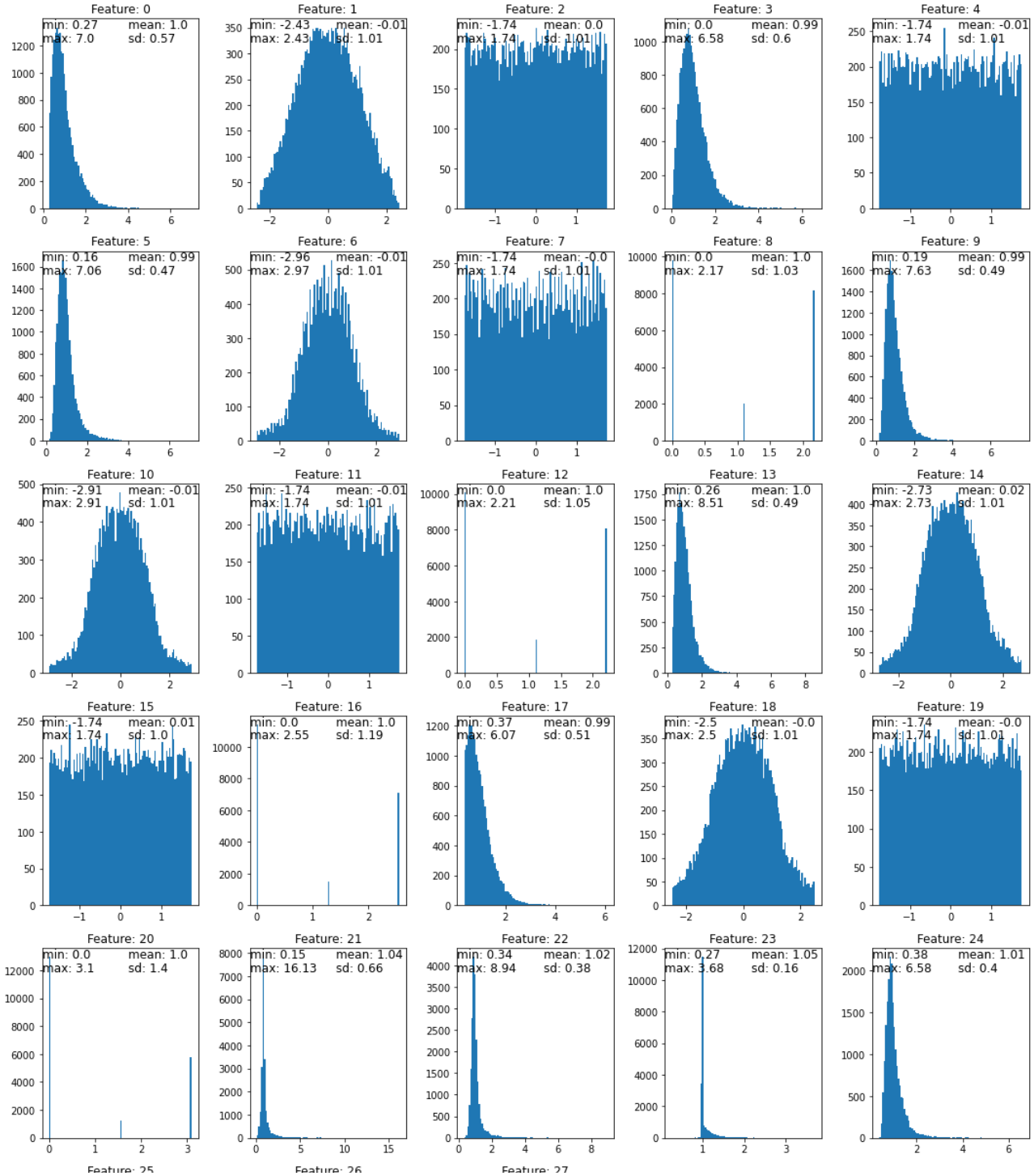
```
#for features,label in packed_ds.batch(1000).take(1):
fig, axs = plt.subplots(6, 5,figsize=(15,20))
fig.subplots_adjust(hspace = .2, wspace=.1)

axs = axs.ravel()
totMissing=0
for f in  range([len(x) for x in ds.take(1)][0]-1):
```

```
  print(f % 10,end='')
  featVals=[features.numpy()[f] for features,label in packed_ds.take(20000)]
  ax=axs[f]
  ax.hist(featVals ,bins = 100)
  ax.set_title('Feature: ' + str(f))
  ax.text(x=0,y=1,s='min: '+str(round(min(featVals),2)), transform=ax.transAxes, fontsi
  ax.text(x=0,y=0.93,s='max: '+str(round(max(featVals),2)), transform=ax.transAxes, for
  ax.text(x=0.55,y=1,s='mean: '+str(round(np.mean(featVals),2)), transform=ax.transAxes
  ax.text(x=0.55,y=0.93,s='sd: '+str(round(np.std(featVals),2)), transform=ax.transAxes
  totMissing += sum(np.isnan(featVals))
  #if(f % 4 !=0):
    #ax.get_yaxis().set_visible(False)
plt.tight_layout()
print("\nTotal Missing values: ", totMissing)


```

```
01234567890123456789012345678
Total Missing values:  0
```



Sampling

To keep this tutorial relatively short use just the first 1000 samples for validation, and the next 10 000 for training:

```
N_VALIDATION = int(1e3)
N_TRAIN = int(1e4)
BUFFER_SIZE = int(1e4)
BATCH_SIZE = 500  # TODO: Note that the paper used 100, we can increase this to reduce
STEPS_PER_EPOCH = N_TRAIN//BATCH_SIZE
```

The `Dataset.skip` and `Dataset.take` methods make this easy.

At the same time, use the `Dataset.cache` method to ensure that the loader doesn't need to re-read the data form the file on each epoch:

```
validate_ds = packed_ds.take(N_VALIDATION).cache()
train_ds = packed_ds.skip(N_VALIDATION).take(N_TRAIN).cache()
```

```
train_ds
```

```
<CacheDataset shapes: ((28,), ()), types: (tf.float32, tf.float32)>
```

These datasets return individual examples. Use the `.batch` method to create batches of an appropriate size for training. Before batching also remember to `.shuffle` and `.repeat` the training set.

```
validate_ds = validate_ds.batch(BATCH_SIZE)
train_ds = train_ds.shuffle(BUFFER_SIZE).repeat().batch(BATCH_SIZE)
```

Training procedure

Many models train better if you gradually reduce the learning rate during training. Use `optimizers.schedules` to reduce the learning rate over time:

```
# https://www.tensorflow.org/tutorials/text/transformer
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self):
    super(CustomSchedule, self).__init__()

  def __call__(self, step):
    ## Original Paper
    # update_callbacks=pylearn2.training_algorithms.sgd.ExponentialDecay(
    #                                     decay_factor=1.0000002, # Decreases by this f
    #                                     min_lr=.000001
    #                                     )

    # Implementation in TensorFlow
    lr = tf.clip_by_value(0.05 / 1.0000002**step, clip_value_min=0.000001, clip_value_m
    return lr

```

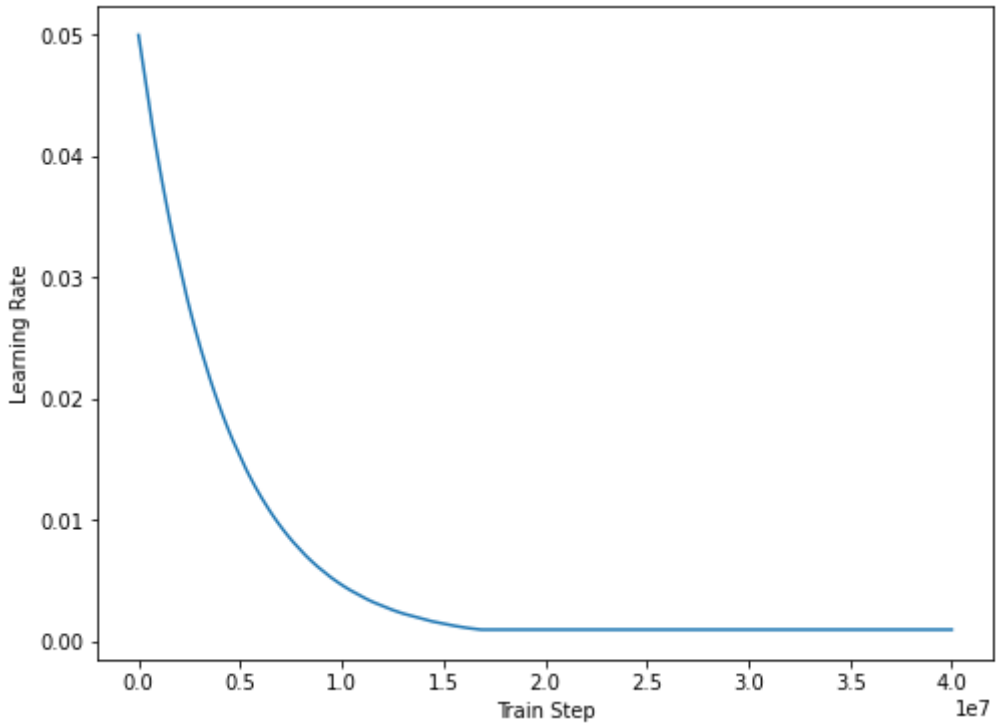
```
tf.range(25, dtype=tf.float32)
```

```
<tf.Tensor: shape=(25,), dtype=float32, numpy=
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
       13., 14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24.],
      dtype=float32)>
```

```
temp_lr_schedule = CustomSchedule()
plt.figure(figsize = (8,6))
lrs = temp_lr_schedule(tf.range(40000000, dtype=tf.float32))
plt.plot(lrs)
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")

```

```
Text(0.5, 0, 'Train Step')
```




```
lrs
```

```
<tf.Tensor: shape=(40000000,), dtype=float32, numpy=
array([0.05      , 0.04999999, 0.04999998, ..., 0.00091578, 0.00091578,
       0.00091578], dtype=float32)>
```

```
def get_optimizer():
  lr_schedule = CustomSchedule()
  return tf.keras.optimizers.SGD(lr_schedule, momentum=0.9)
```

Each model in this tutorial will use the same training configuration. So set these up in a reusable way, starting with the list of callbacks.

The training for this tutorial runs for many short epochs. To reduce the logging noise use the `tfdocs.EpochDots` which simply a `.` for each epoch and, and a full set of metrics every 100 epochs.

Next include `callbacks.EarlyStopping` to avoid long and unnecessary training times. Note that this callback is set to monitor the `val_binary_crossentropy`, not the `val_loss`. This difference will be important later.

Use `callbacks.TensorBoard` to generate TensorBoard logs for the training.

```
def get_callbacks(name):
  return [
    tfdocs.modeling.EpochDots(),
    tf.keras.callbacks.EarlyStopping(monitor='val_binary_crossentropy', min_delta=0.000
    tf.keras.callbacks.TensorBoard(logdir/name),
  ]
```

Similarly each model will use the same `Model.compile` and `Model.fit` settings:

```
def compile_and_fit(model, name, optimizer=None, max_epochs=10000):
  if optimizer is None:
    optimizer = get_optimizer()
  model.compile(optimizer=optimizer,
                loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                # loss=tf.keras.metrics.AUC(),
                metrics=[
                  tf.keras.metrics.AUC(name='AUC'),
                  tf.keras.losses.BinaryCrossentropy(from_logits=True, name='binary_cro
                  'accuracy'])

  model.summary()

  history = model.fit(
    train_ds,
    steps_per_epoch = STEPS_PER_EPOCH,
    epochs=max_epochs,
    validation_data=validate_ds,
    callbacks=get_callbacks(name),
    verbose=2)
  return history
```

```
size_histories = {}
```

Model from Paper

```
# https://www.tensorflow.org/api_docs/python/tf/keras/initializers/RandomNormal
first_initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=0.1, seed=42)
outer_initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=0.001, seed=42)
other_initializers = tf.keras.initializers.RandomNormal(mean=0., stddev=0.05, seed=42)

# Top Layer (https://www.quora.com/Are-the-top-layers-of-a-deep-neural-network-the-firs
# Weight Decay: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-le
weight_decay=0.00001
paper_model = tf.keras.Sequential([
    layers.Dense(300, activation='tanh', input_shape=(FEATURES,), kernel_initializer=fi
    layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_
    layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_
    layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_
    layers.Dropout(0.5),  # Top Hidden Layer
```

```
    layers.Dropout(0.5), # Top hidden Layer
    layers.Dense(1, activation='sigmoid', kernel_initializer=outer_initializer, kernel_

])
```

```
size_histories['paper'] = compile_and_fit(paper_model, 'sizes/paper')
```

```
Epoch 60/10000
.20/20 - 0s - loss: 0.6590 - AUC: 0.6982 - binary_crossentropy: 0.6513 - accurac
Epoch 61/10000
.20/20 - 0s - loss: 0.6619 - AUC: 0.6909 - binary_crossentropy: 0.6543 - accurac
Epoch 62/10000
.20/20 - 0s - loss: 0.6580 - AUC: 0.7005 - binary_crossentropy: 0.6503 - accurac
Epoch 63/10000
.20/20 - 0s - loss: 0.6569 - AUC: 0.7043 - binary_crossentropy: 0.6493 - accurac
Epoch 64/10000
.20/20 - 0s - loss: 0.6559 - AUC: 0.7062 - binary_crossentropy: 0.6483 - accurac
Epoch 65/10000
.20/20 - 0s - loss: 0.6558 - AUC: 0.7078 - binary_crossentropy: 0.6482 - accurac
Epoch 66/10000
.20/20 - 0s - loss: 0.6579 - AUC: 0.6993 - binary_crossentropy: 0.6503 - accurac
Epoch 67/10000
.20/20 - 0s - loss: 0.6577 - AUC: 0.7015 - binary_crossentropy: 0.6501 - accurac
Epoch 68/10000
.20/20 - 0s - loss: 0.6559 - AUC: 0.7072 - binary_crossentropy: 0.6483 - accurac
Epoch 69/10000
.20/20 - 0s - loss: 0.6596 - AUC: 0.6985 - binary_crossentropy: 0.6520 - accurac
Epoch 70/10000
.20/20 - 0s - loss: 0.6631 - AUC: 0.6843 - binary_crossentropy: 0.6554 - accurac
Epoch 71/10000
.20/20 - 0s - loss: 0.6584 - AUC: 0.7000 - binary_crossentropy: 0.6508 - accurac
Epoch 72/10000
.20/20 - 0s - loss: 0.6566 - AUC: 0.7056 - binary_crossentropy: 0.6489 - accurac
Epoch 73/10000
.20/20 - 0s - loss: 0.6565 - AUC: 0.7061 - binary_crossentropy: 0.6489 - accurac
Epoch 74/10000
.20/20 - 0s - loss: 0.6523 - AUC: 0.7142 - binary_crossentropy: 0.6446 - accurac
Epoch 75/10000
.20/20 - 0s - loss: 0.6535 - AUC: 0.7102 - binary_crossentropy: 0.6459 - accurac
Epoch 76/10000
.20/20 - 0s - loss: 0.6602 - AUC: 0.6877 - binary_crossentropy: 0.6525 - accurac
Epoch 77/10000
.20/20 - 0s - loss: 0.6563 - AUC: 0.7024 - binary_crossentropy: 0.6487 - accurac
Epoch 78/10000
.20/20 - 0s - loss: 0.6509 - AUC: 0.7120 - binary_crossentropy: 0.6433 - accurac
Epoch 79/10000
.20/20 - 0s - loss: 0.6513 - AUC: 0.7138 - binary_crossentropy: 0.6436 - accurac
Epoch 80/10000
.20/20 - 0s - loss: 0.6503 - AUC: 0.7178 - binary_crossentropy: 0.6426 - accurac
Epoch 81/10000
.20/20 - 0s - loss: 0.6501 - AUC: 0.7164 - binary_crossentropy: 0.6424 - accurac
Epoch 82/10000
.20/20 - 0s - loss: 0.6483 - AUC: 0.7198 - binary_crossentropy: 0.6406 - accurac
Epoch 83/10000
.20/20 - 0s - loss: 0.6487 - AUC: 0.7181 - binary_crossentropy: 0.6410 - accurac
Epoch 84/10000
.20/20 - 0s - loss: 0.6511 - AUC: 0.7151 - binary_crossentropy: 0.6434 - accurac
Epoch 85/10000
.20/20 - 0s - loss: 0.6491 - AUC: 0.7185 - binary_crossentropy: 0.6414 - accurac
Epoch 86/10000
.20/20 - 0s - loss: 0.6536 - AUC: 0.7085 - binary_crossentropy: 0.6459 - accurac
```

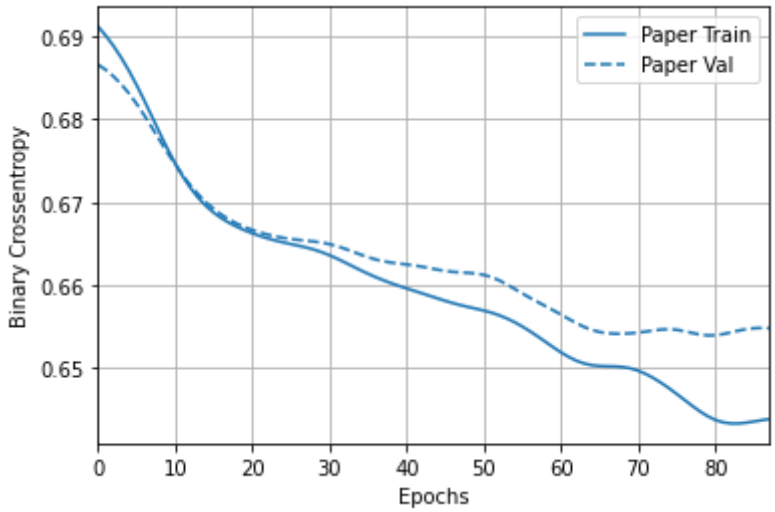
```
Epoch 87/10000
.20/20 - 0s - loss: 0.6570 - AUC: 0.6924 - binary_crossentropy: 0.6493 - accurac
Epoch 88/10000
.20/20 - 0s - loss: 0.6482 - AUC: 0.7201 - binary_crossentropy: 0.6405 - accurac
```

```
1 size_histories['paper'].history['AUC']
```

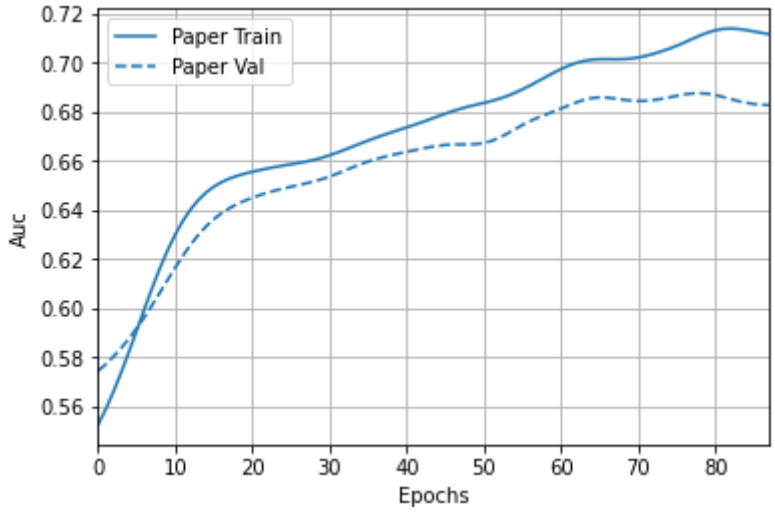
```
 0.6616893410682678,
 0.663505494594574,
 0.65963214635849,
 0.6632059216499329,
 0.6671822667121887,
 0.6692437529563904,
 0.671256422996521,
 0.6672073602676392,
 0.6700705885887146,
 0.6698629856109619,
 0.675326406955719,
 0.6713006496429443,
 0.6778548359870911,
 0.6828531622886658,
 0.6589682698249817,
 0.6810590028762817,
 0.6813192963600159,
 0.6843479871749878,
 0.6796518564224243,
 0.6906561851501465,
 0.6844614744186401,
 0.6785256266593933,
 0.6791103482246399,
 0.6831559538841248,
 0.6851184368133545,
 0.6891517639160156,
 0.6927095055580139,
 0.6821547746658325,
 0.6899731159210205,
 0.7005809545516968,
 0.6981526017189026,
 0.6909323334693909,
 0.700454592704773,
 0.7043041586875916,
 0.7062296867370605,
 0.7077540755271912,
 0.6992825865745544,
 0.7015367150306702,
 0.7072222232818604,
 0.6984642744064331,
 0.6842929720878601,
 0.6999787092208862,
 0.7056492567062378,
 0.7060688734054565,
 0.7141595482826233,
 0.7102193236351013,
 0.6877254247665405,
 0.7023689150810242,
 0.7119570374488831,
 0.7138358950614929,
 0.717795193195343,
 0.7163913249969482,
```

```
0.7197931408882141,
0.7180827260017395,
0.7151320576667786,
0.7185297608375549,
0.7084594368934631,
0.6923511028289795,
0.7200726866722107]
```

```
plotter = tfdocs.plots.HistoryPlotter(metric = 'binary_crossentropy', smoothing_std=10)
plotter.plot(size_histories)
# plt.ylim([0.5, 0.7])
```




```
plotter = tfdocs.plots.HistoryPlotter(metric = 'AUC', smoothing_std=10)
plotter.plot(size_histories)
# plt.ylim([0.5, 0.7])
```



Plot the training and validation losses

The solid lines show the training loss, and the dashed lines show the validation loss (remember: a lower validation loss indicates a better model).

While building a larger model gives it more power, if this power is not constrained somehow it can easily overfit to the training set.

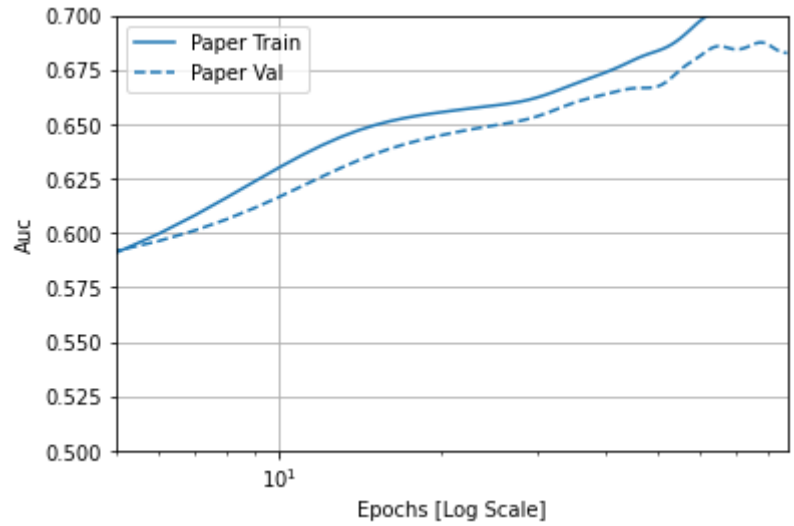
In this example, typically, only the `"Tiny"` model manages to avoid overfitting altogether, and each of the larger models overfit the data more quickly. This becomes so severe for the `"large"` model that you need to switch the plot to a log-scale to really see what's happening.

This is apparent if you plot and compare the validation metrics to the training metrics.

- It's normal for there to be a small difference.
- If both metrics are moving in the same direction, everything is fine.
- If the validation metric begins to stagnate while the training metric continues to improve, you are probably close to overfitting.
- If the validation metric is going in the wrong direction, the model is clearly overfitting.

```
plotter.plot(size_histories)
a = plt.xscale('log')
plt.xlim([5, max(plt.xlim())])
plt.ylim([0.5, 0.7])
plt.xlabel("Epochs [Log Scale]")
```

```
Text(0.5, 0, 'Epochs [Log Scale]')
```



Note: All the above training runs used the `callbacks.EarlyStopping` to end the training once it was clear the model was not making progress.

View in TensorBoard

These models all wrote TensorBoard logs during training.

To open an embedded TensorBoard viewer inside a notebook, copy the following into a code-cell:

```
%tensorboard --logdir {logdir}/sizes
```

You can view the [results of a previous run](#) of this notebook on [TensorBoard.dev](#).

TensorBoard.dev is a managed experience for hosting, tracking, and sharing ML experiments with everyone.

It's also included in an `<iframe>` for convenience:

```
display.IFrame(
    src="https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingW
    width="100%", height="800px")
```

TensorBoard.dev SCALARS SEND FEEDBACK

Add a name and description to the experiment to provide more context and details for these results. Learn more *Created on Nov 15, 20…*

Show data download links

Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing 0.97

Horizontal Axis

STEP RELATIVE

WALL

Runs

Write a regex to filter runs

large/train

large/validation

medium/train

medium/validation

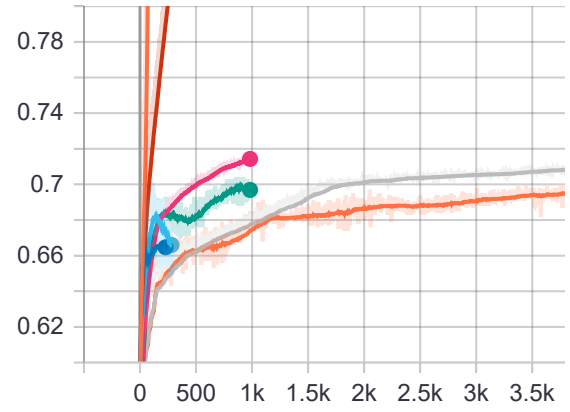
TOGGLE ALL RUNS

experiment vW7jmmF9TmKmy3rbheMQpw

Filter tags (regular expressions supported)

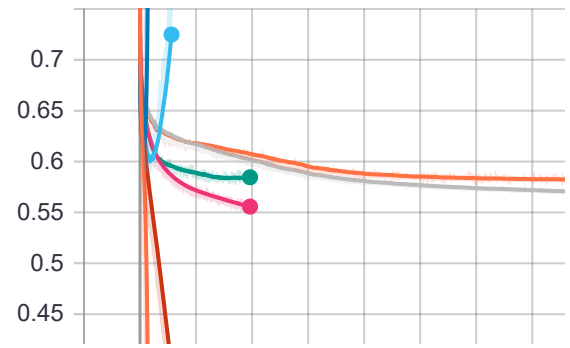
epoch_accuracy

epoch_accuracy



epoch_binary_crossentropy

epoch_binary_crossentropy



If you want to share TensorBoard results you can upload the logs to TensorBoard.dev by copying the following into a code-cell.

Note: This step requires a Google account.

```
!tensorboard dev upload --logdir  {logdir}/sizes
```

Caution: This command does not terminate. It's designed to continuously upload the results of long-running experiments. Once your data is uploaded you need to stop it using the "interrupt execution" option in your notebook tool.

Strategies to prevent overfitting

Before getting into the content of this section copy the training logs from the `"Tiny"` model above, to use as a baseline for comparison.

```
shutil.rmtree(logdir/'regularizers/Tiny', ignore_errors=True)
shutil.copytree(logdir/'sizes/Tiny', logdir/'regularizers/Tiny')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-74-de68c6b319db> in <module>()
      1 shutil.rmtree(logdir/'regularizers/Tiny', ignore_errors=True)
----> 2 shutil.copytree(logdir/'sizes/Tiny', logdir/'regularizers/Tiny')

/usr/lib/python3.6/shutil.py in copytree(src, dst, symlinks, ignore, copy_function,
ignore_dangling_symlinks)
    313
    314     """
--> 315     names = os.listdir(src)
    316     if ignore is not None:
    317         ignored_names = ignore(src, names)

FileNotFoundError: [Errno 2] No such file or directory:
'/tmp/tmp00k9m44g/tensorboard_logs/sizes/Tiny'
```

```
regularizer_histories = {}
regularizer_histories['Tiny'] = size_histories['Tiny']
```

Add weight regularization

You may be familiar with Occam's Razor principle: given two explanations for something, the explanation most likely to be correct is the "simplest" one, the one that makes the least amount of assumptions. This also applies to the models learned by neural networks: given some training data and a network architecture, there are multiple sets of weights values (multiple

models) that could explain the data, and simpler models are less likely to overfit than complex ones.

A "simple model" in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters altogether, as we saw in the section above). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more "regular". This is called "weight regularization", and it is done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

- L1 regularization, where the cost added is proportional to the absolute value of the weights coefficients (i.e. to what is called the "L1 norm" of the weights).
- L2 regularization, where the cost added is proportional to the square of the value of the weights coefficients (i.e. to what is called the squared "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 exact same as L2 regularization.

L1 regularization pushes weights towards exactly zero encouraging a sparse model. L2 regularization will penalize the weights parameters without making them sparse since the penalty goes to zero for small weights. one reason why L2 is more common.

In `tf.keras`, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Let's add L2 weight regularization now.

```
l2_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001),
                 input_shape=(FEATURES,)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1)
])

regularizer_histories['l2'] = compile_and_fit(l2_model, "regularizers/l2")
```

`l2(0.001)` means that every coefficient in the weight matrix of the layer will add `0.001 * weight_coefficient_value**2` to the total **loss** of the network.

That is why we're monitoring the `binary_crossentropy` directly. Because it doesn't have this regularization component mixed in.

So, that same `"Large"` model with an `L2` regularization penalty performs much better:

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

As you can see, the `"L2"` regularized model is now much more competitive with the the `"Tiny"` model. This `"L2"` model is also much more resistant to overfitting than the `"Large"` model it was based on despite having the same number of parameters.

More info

There are two important things to note about this sort of regularization.

First: if you are writing your own training loop, then you need to be sure to ask the model for it's regularization losses.

```
result = l2_model(features)
regularization_loss=tf.add_n(l2_model.losses)
```

Second: This implementation works by adding the weight penalties to the model's loss, and then applying a standard optimization procedure after that.

There is a second approach that instead only runs the optimizer on the raw loss, and then while applying the calculated step the optimizer also applies some weight decay. This "Decoupled Weight Decay" is seen in optimizers like `optimizers.FTRL` and `optimizers.AdamW`.

Add dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Hinton and his students at the University of Toronto.

The intuitive explanation for dropout is that because individual nodes in the network cannot rely on the output of the others, each node must output features that are useful on their own.

Dropout, applied to a layer, consists of randomly "dropping out" (i.e. set to zero) a number of output features of the layer during training. Let's say a given layer would normally have returned a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training; after applying dropout, this vector will have a few zero entries distributed at random, e.g. [0, 0.5, 1.3, 0, 1.1].

The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5. At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

In `tf.keras` you can introduce dropout in a network via the Dropout layer, which gets applied to the output of layer right before.

Let's add two Dropout layers in our network to see how well they do at reducing overfitting:

```
dropout_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['dropout'] = compile_and_fit(dropout_model, "regularizers/dropout
```

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

It's clear from this plot that both of these regularization approaches improve the behavior of the `"Large"` model. But this still doesn't beat even the `"Tiny"` baseline.

Next try them both, together, and see if that does better.

Combined L2 + dropout

```
combined_model = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['combined'] = compile_and_fit(combined_model, "regularizers/combi
```

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

This model with the `"Combined"` regularization is obviously the best one so far.

View in TensorBoard

These models also recorded TensorBoard logs.

To open an embedded tensorboard viewer inside a notebook, copy the following into a code-cell:

```
%tensorboard --logdir {logdir}/regularizers
```

You can view the results of a previous run of this notebook on TensorDoard.dev.

It's also included in an `<iframe>` for convenience:

```
display.IFrame(
    src="https://tensorboard.dev/experiment/fGInKDo8TXes1z7HQku9mw/#scalars&_smoothingW
    width = "100%",
    height="800px")

```

This was uploaded with:

```
!tensorboard dev upload --logdir  {logdir}/regularizers
```

Conclusions

To recap: here are the most common ways to prevent overfitting in neural networks:

- Get more training data.
- Reduce the capacity of the network.
- Add weight regularization.
- Add dropout.

Two important approaches not covered in this guide are:

- data-augmentation
- batch normalization

Remember that each method can help on its own, but often combining them can be even more effective.