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-learning-databases/00280/HIGGS.csv.gz')
```

```
Downloading data from https://archive.ics.uci.edu/ml/machine-learning-databases/00280/HIGGS.csv.gz
2816409600/2816407858 [==============================] - 35s 0us/step
```

```
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="GZIP")
```

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()
```

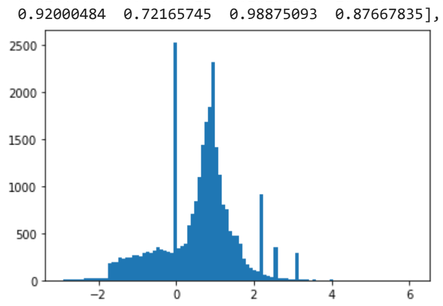
EDA

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, -0.24857314, -1.0920639, 0.0, 1.3749921, -0.6536742, 0.9303491, 1.1074361, 1.1389043, -1.5781983, -1.0469854, 0.0, 0.65792954, -0.01045457, -0.04576717, 3.1019614, 1.35376, 0.9795631, 0.97807616,
(1.0, 0.9075421, 0.32914728, 0.35941187, 1.4979699, -0.31300953, 1.0955306, -0.5575249, -1.5882298, 2.1730762, 0.8125812, -0.21364193, 1.2710146, 2.2148721, 0.49999395, -1.2614318, 0.73215616, 0.0, 0.3987009, -1.1389301, -0.0008191102, 0.0, 0.3022199, 0.83304816, 0.9856996
```

```
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
```

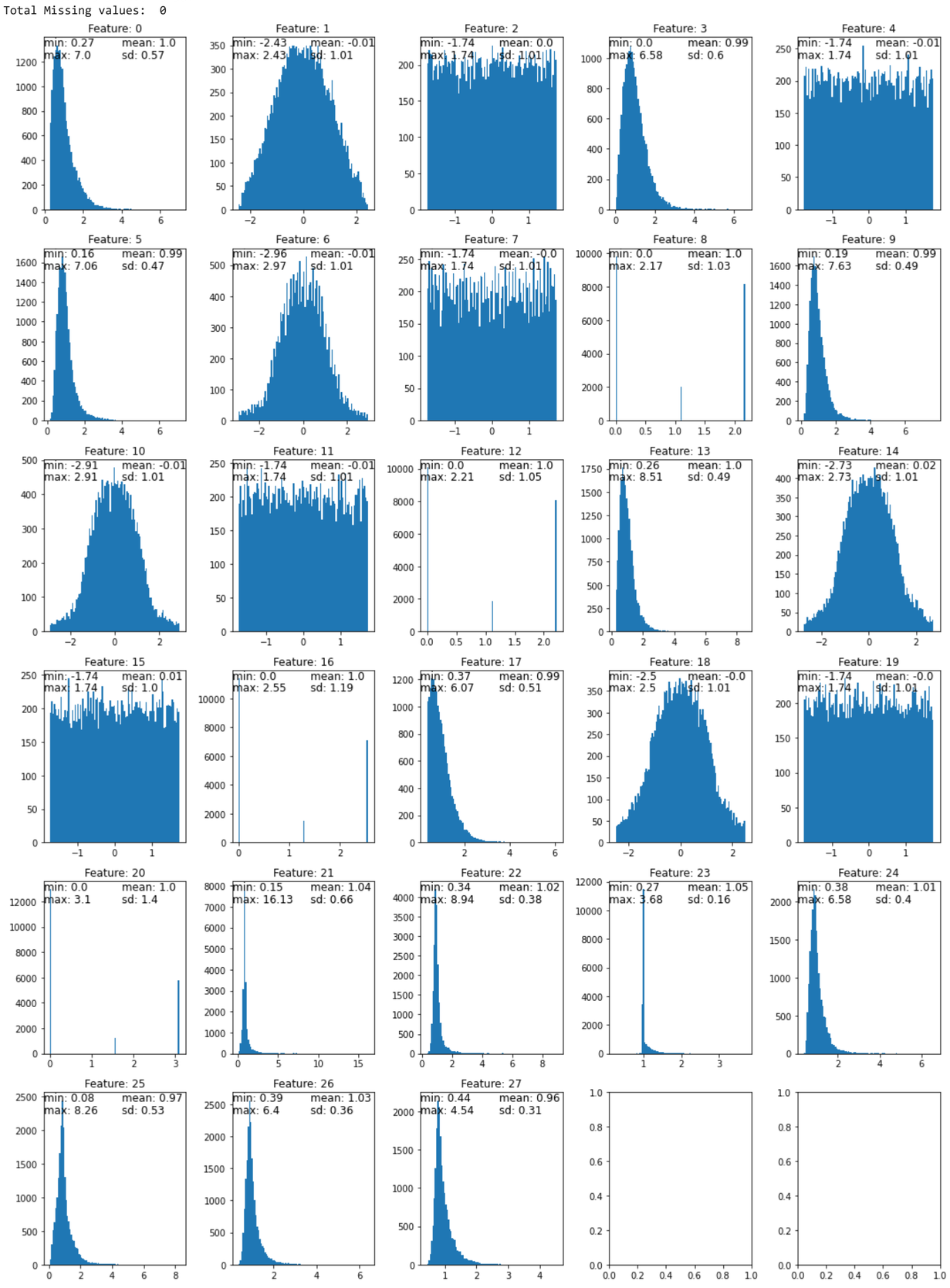
```
#for features,label in packed_ds.batch(1000).take(1):
fig, axs = plt.subplots(6, 5,figsize=(15,20))
fig.subplots_adjust(hspace = .2, wspace=.1)

```

```
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, fontsize=12,verticalalignment='top')
  ax.text(x=0,y=0.93,s='max: '+str(round(max(featVals),2)), transform=ax.transAxes, fontsize=12,verticalalignment='top')
  ax.text(x=0.55,y=1,s='mean: '+str(round(np.mean(featVals),2)), transform=ax.transAxes, fontsize=12,verticalalignment='top')
  ax.text(x=0.55,y=0.93,s='sd: '+str(round(np.std(featVals),2)), transform=ax.transAxes, fontsize=12,verticalalignment='top')
  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 training time.
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 factor every batch. (1/(1.000001^8000)^100
    #                                  min_lr=.000001
    #                                  )

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

```

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

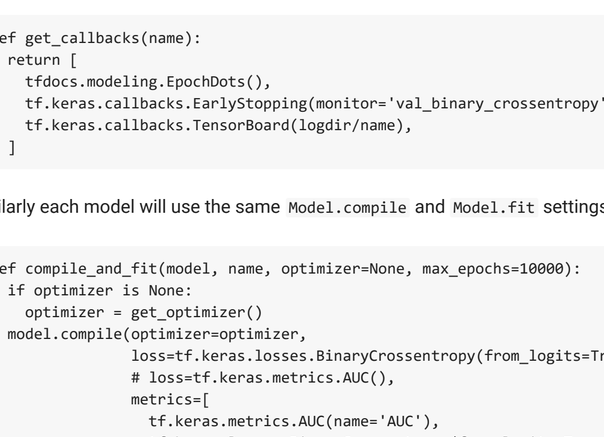
```
<tf.Tensor: shape=(25,), dtype=float32, numpy=
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
```

```
       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=0.00001, patience=10),
    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_crossentropy'),
                  '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-first-layers-or-the-last-layers)
# Weight Decay: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/
weight_decay=0.00001
paper_model = tf.keras.Sequential([
    layers.Dense(300, activation='tanh', input_shape=(FEATURES,), kernel_initializer=first_initializer, kernel_regularizer=l2(weight_decay)),
    layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_regularizer=l2(weight_decay)),
    layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_regularizer=l2(weight_decay)),
    layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_regularizer=l2(weight_decay)),
    layers.Dropout(0.5), # Top Hidden Layer
    layers.Dense(1, activation='sigmoid', kernel_initializer=outer_initializer, kernel_regularizer=l2(weight_decay))

])
```

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

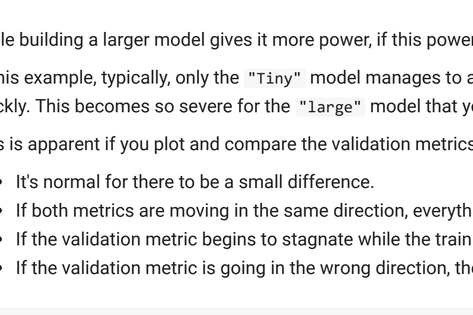
```
Epoch 33/10000
.20/20 - 0s - loss: 0.6694 - AUC: 0.6677 - binary_crossentropy: 0.6618 - accuracy: 0.5994 - val_loss: 0.6729 - val_AUC: 0.6504 - val_binary_crossentropy: 0.6653 - val_accuracy: 0.5570
Epoch 34/10000
.20/20 - 0s - loss: 0.6696 - AUC: 0.6665 - binary_crossentropy: 0.6621 - accuracy: 0.5970 - val_loss: 0.6716 - val_AUC: 0.6622 - val_binary_crossentropy: 0.6640 - val_accuracy: 0.5530
Epoch 35/10000
.20/20 - 0s - loss: 0.6683 - AUC: 0.6723 - binary_crossentropy: 0.6607 - accuracy: 0.5988 - val_loss: 0.6716 - val_AUC: 0.6555 - val_binary_crossentropy: 0.6640 - val_accuracy: 0.5680
Epoch 36/10000
.20/20 - 0s - loss: 0.6689 - AUC: 0.6715 - binary_crossentropy: 0.6613 - accuracy: 0.5972 - val_loss: 0.6711 - val_AUC: 0.6611 - val_binary_crossentropy: 0.6636 - val_accuracy: 0.5580
Epoch 37/10000
.20/20 - 0s - loss: 0.6674 - AUC: 0.6768 - binary_crossentropy: 0.6598 - accuracy: 0.5983 - val_loss: 0.6706 - val_AUC: 0.6664 - val_binary_crossentropy: 0.6630 - val_accuracy: 0.5980
Epoch 38/10000
.20/20 - 0s - loss: 0.6675 - AUC: 0.6785 - binary_crossentropy: 0.6599 - accuracy: 0.6094 - val_loss: 0.6713 - val_AUC: 0.6641 - val_binary_crossentropy: 0.6638 - val_accuracy: 0.5560
Epoch 39/10000
.20/20 - 0s - loss: 0.6681 - AUC: 0.6747 - binary_crossentropy: 0.6605 - accuracy: 0.6014 - val_loss: 0.6692 - val_AUC: 0.6707 - val_binary_crossentropy: 0.6616 - val_accuracy: 0.5710
Epoch 40/10000
.20/20 - 0s - loss: 0.6659 - AUC: 0.6798 - binary_crossentropy: 0.6583 - accuracy: 0.6102 - val_loss: 0.6715 - val_AUC: 0.6627 - val_binary_crossentropy: 0.6640 - val_accuracy: 0.5600
Epoch 41/10000
.20/20 - 0s - loss: 0.6695 - AUC: 0.6693 - binary_crossentropy: 0.6619 - accuracy: 0.6005 - val_loss: 0.6687 - val_AUC: 0.6677 - val_binary_crossentropy: 0.6611 - val_accuracy: 0.5960
Epoch 42/10000
.20/20 - 0s - loss: 0.6658 - AUC: 0.6778 - binary_crossentropy: 0.6582 - accuracy: 0.6088 - val_loss: 0.6684 - val_AUC: 0.6689 - val_binary_crossentropy: 0.6608 - val_accuracy: 0.6040
Epoch 43/10000
.20/20 - 0s - loss: 0.6655 - AUC: 0.6820 - binary_crossentropy: 0.6579 - accuracy: 0.6133 - val_loss: 0.6687 - val_AUC: 0.6755 - val_binary_crossentropy: 0.6611 - val_accuracy: 0.6130
Epoch 44/10000
.20/20 - 0s - loss: 0.6647 - AUC: 0.6823 - binary_crossentropy: 0.6571 - accuracy: 0.6154 - val_loss: 0.6703 - val_AUC: 0.6677 - val_binary_crossentropy: 0.6627 - val_accuracy: 0.5640
Epoch 45/10000
.20/20 - 0s - loss: 0.6683 - AUC: 0.6748 - binary_crossentropy: 0.6607 - accuracy: 0.6062 - val_loss: 0.6668 - val_AUC: 0.6685 - val_binary_crossentropy: 0.6592 - val_accuracy: 0.5850
Epoch 46/10000
.20/20 - 0s - loss: 0.6641 - AUC: 0.6851 - binary_crossentropy: 0.6565 - accuracy: 0.6160 - val_loss: 0.6709 - val_AUC: 0.6611 - val_binary_crossentropy: 0.6634 - val_accuracy: 0.5570
Epoch 47/10000
.20/20 - 0s - loss: 0.6652 - AUC: 0.6834 - binary_crossentropy: 0.6576 - accuracy: 0.6083 - val_loss: 0.6642 - val_AUC: 0.6828 - val_binary_crossentropy: 0.6566 - val_accuracy: 0.6030
Epoch 48/10000
.20/20 - 0s - loss: 0.6643 - AUC: 0.6824 - binary_crossentropy: 0.6567 - accuracy: 0.6135 - val_loss: 0.6669 - val_AUC: 0.6837 - val_binary_crossentropy: 0.6593 - val_accuracy: 0.6250
Epoch 49/10000
.20/20 - 0s - loss: 0.6629 - AUC: 0.6880 - binary_crossentropy: 0.6553 - accuracy: 0.6212 - val_loss: 0.6694 - val_AUC: 0.6833 - val_binary_crossentropy: 0.6618 - val_accuracy: 0.6280
Epoch 50/10000
.20/20 - 0s - loss: 0.6671 - AUC: 0.6707 - binary_crossentropy: 0.6595 - accuracy: 0.6022 - val_loss: 0.6670 - val_AUC: 0.6635 - val_binary_crossentropy: 0.6594 - val_accuracy: 0.5760
Epoch 51/10000
.20/20 - 0s - loss: 0.6632 - AUC: 0.6886 - binary_crossentropy: 0.6556 - accuracy: 0.6222 - val_loss: 0.6680 - val_AUC: 0.6651 - val_binary_crossentropy: 0.6604 - val_accuracy: 0.5710
Epoch 52/10000
.20/20 - 0s - loss: 0.6633 - AUC: 0.6834 - binary_crossentropy: 0.6557 - accuracy: 0.6178 - val_loss: 0.6617 - val_AUC: 0.6904 - val_binary_crossentropy: 0.6541 - val_accuracy: 0.6060
Epoch 53/10000
.20/20 - 0s - loss: 0.6635 - AUC: 0.6873 - binary_crossentropy: 0.6559 - accuracy: 0.6146 - val_loss: 0.6639 - val_AUC: 0.6857 - val_binary_crossentropy: 0.6563 - val_accuracy: 0.5980
Epoch 54/10000
.20/20 - 0s - loss: 0.6613 - AUC: 0.6897 - binary_crossentropy: 0.6537 - accuracy: 0.6216 - val_loss: 0.6662 - val_AUC: 0.6790 - val_binary_crossentropy: 0.6586 - val_accuracy: 0.5960
Epoch 55/10000
.20/20 - 0s - loss: 0.6618 - AUC: 0.6899 - binary_crossentropy: 0.6542 - accuracy: 0.6244 - val_loss: 0.6647 - val_AUC: 0.6841 - val_binary_crossentropy: 0.6571 - val_accuracy: 0.6200
Epoch 56/10000
.20/20 - 0s - loss: 0.6632 - AUC: 0.6903 - binary_crossentropy: 0.6555 - accuracy: 0.6234 - val_loss: 0.6673 - val_AUC: 0.6757 - val_binary_crossentropy: 0.6597 - val_accuracy: 0.5720
Epoch 57/10000
.20/20 - 0s - loss: 0.6613 - AUC: 0.6948 - binary_crossentropy: 0.6537 - accuracy: 0.6284 - val_loss: 0.6631 - val_AUC: 0.6788 - val_binary_crossentropy: 0.6555 - val_accuracy: 0.5880
Epoch 58/10000
.20/20 - 0s - loss: 0.6585 - AUC: 0.7015 - binary_crossentropy: 0.6508 - accuracy: 0.6348 - val_loss: 0.6667 - val_AUC: 0.6812 - val_binary_crossentropy: 0.6590 - val_accuracy: 0.5810
Epoch 59/10000
.20/20 - 0s - loss: 0.6602 - AUC: 0.6943 - binary_crossentropy: 0.6526 - accuracy: 0.6260 - val_loss: 0.6622 - val_AUC: 0.6891 - val_binary_crossentropy: 0.6545 - val_accuracy: 0.6100
Epoch 60/10000
.20/20 - 0s - loss: 0.6636 - AUC: 0.6824 - binary_crossentropy: 0.6560 - accuracy: 0.6228 - val_loss: 0.6743 - val_AUC: 0.6526 - val_binary_crossentropy: 0.6667 - val_accuracy: 0.5340
```

```
Epoch 61/10000
.20/20 - 0s - loss: 0.6658 - AUC: 0.6721 - binary_crossentropy: 0.6582 - accuracy: 0.6052 - val_loss: 0.6626 - val_AUC: 0.6767 - val_binary_crossentropy: 0.6550 - val_accuracy: 0.6010
Epoch 62/10000
.20/20 - 0s - loss: 0.6574 - AUC: 0.7022 - binary_crossentropy: 0.6498 - accuracy: 0.6360 - val_loss: 0.6627 - val_AUC: 0.6841 - val_binary_crossentropy: 0.6551 - val_accuracy: 0.5990
```

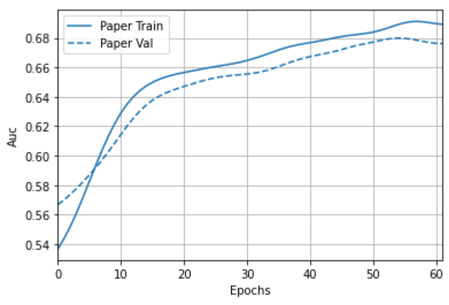
```
size_histories['paper'].history['AUC'][0:10]
```

```
[0.5014192461967468,
 0.5259501934051514,
 0.5493761897087097,
 0.567119836807251,
 0.5775632858276367,
 0.5930601954460144,
 0.6003478765487671,
 0.6114186644554138,
 0.6216607093811035,
 0.6321367621421814]
```

```
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.

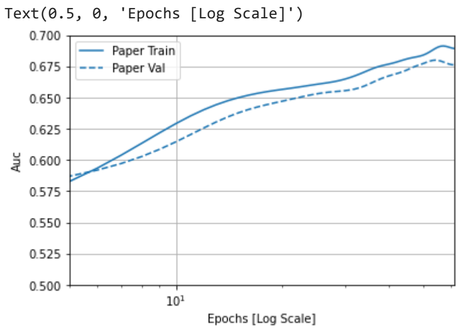
In this example, typically, only the `"Tiny"` model manages to avoid overfitting altogether, and each of the larger models overfit the data more quickly. This becomes so severe for the `"large"` model that you need to switch the plot to a log-scale to really see what's happening.

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:

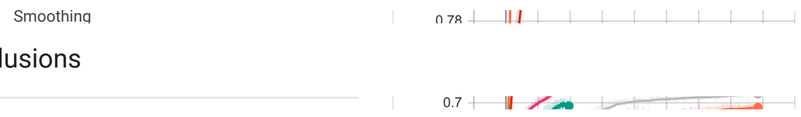
```
display.IFrame(
    src="https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97",
    width="100%", height="800px")
```

If you want to share TensorBoard results you can upload the logs to [TensorBoard.dev](https://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.



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.