Overfit and underfit

As always, the code in this example will use the `tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras) API, which you can learn more about in the TensorFlow Keras guide (https://www.tensorflow.org/guide/keras).

In both of the previous examples—classifying text (/tutorials/keras/text_classification_with_hub) and predicting fuel efficiency (/tutorials/keras/regression)—the accuracy of models on the validation data would peak after training for a number of epochs and then stagnate or start decreasing.

In other words, your model would *overfit* to the training data. Learning how to deal with overfitting is important. Although it's often possible to achieve high accuracy on the *training set*, what you really want is to develop models that generalize well to a *testing set* (or data they haven't seen before).

The opposite of overfitting is *underfitting*. Underfitting occurs when there is still room for improvement on the train data. This can happen for a number of reasons: If the model is not powerful enough, is over-regularized, or has simply not been trained long enough. This means the network has not learned the relevant patterns in the training data.

If you train for too long though, the model will start to overfit and learn patterns from the training data that don't generalize to the test data. You need to strike a balance. Understanding how to train for an appropriate number of epochs as you'll explore below is a useful skill.

To prevent overfitting, the best solution is to use more complete training data. The dataset should cover the full range of inputs that the model is expected to handle. Additional data may only be useful if it covers new and interesting cases.

A model trained on more complete data will naturally generalize better. When that is no longer possible, the next best solution is to use techniques like regularization. These place constraints on the quantity and type of information your model can store. If a network can

only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

In this notebook, you'll explore several common regularization techniques, and use them to improve on a classification model.

Setup

Before getting started, import the necessary packages:

```
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import regularizers

print(tf.__version__)
```

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

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

```
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', 'http://mlphysics.ics.uci.edu/dat
```

```
FEATURES = 28
```

The **`tf.data.experimental.CsvDataset`** (https://www.tensorflow.org/api_docs/python/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_t
```

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 **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) that takes batches of 10,000 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()
```

Inspect 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)
```

To keep this tutorial relatively short, use just the first 1,000 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
STEPS_PER_EPOCH = N_TRAIN//BATCH_SIZE
```

The `Dataset.skip` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#skip) and `Dataset.take` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#take) methods make this easy.

At the same time, use the `Dataset.cache` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#cache) method to ensure that the loader doesn't need to re-read the data from 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
```

These datasets return individual examples. Use the `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) method to create batches of an appropriate size for training. Before batching, also remember to use `Dataset.shuffle` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) and `Dataset.repeat` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) on the training set.

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

Demonstrate overfitting

The simplest way to prevent overfitting is to start with a small model: A model with a small number of learnable parameters (which is determined by the number of layers and the number of units per layer). In deep learning, the number of learnable parameters in a model is often referred to as the model's "capacity".

Intuitively, a model with more parameters will have more "memorization capacity" and therefore will be able to easily learn a perfect dictionary-like mapping between training samples and their targets, a mapping without any generalization power, but this would be useless when making predictions on previously unseen data.

Always keep this in mind: deep learning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting.

On the other hand, if the network has limited memorization resources, it will not be able to learn the mapping as easily. To minimize its loss, it will have to learn compressed representations that have more predictive power. At the same time, if you make your model too small, it will have difficulty fitting to the training data. There is a balance between "too much capacity" and "not enough capacity".

Unfortunately, there is no magical formula to determine the right size or architecture of your model (in terms of the number of layers, or the right size for each layer). You will have to experiment using a series of different architectures.

To find an appropriate model size, it's best to start with relatively few layers and parameters, then begin increasing the size of the layers or adding new layers until you see diminishing returns on the validation loss.

Start with a simple model using only densely-connected layers (`tf.keras.layers.Dense` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)) as a baseline, then create larger models, and compare them.

Training procedure

Many models train better if you gradually reduce the learning rate during training. Use `tf.keras.optimizers.schedules` (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules) to reduce the learning rate over time:

```
lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.001,
  decay_steps=STEPS_PER_EPOCH*1000,
  decay_rate=1,
  staircase=False)

def get_optimizer():
  return tf.keras.optimizers.Adam(lr_schedule)
```

The code above sets a `tf.keras.optimizers.schedules.InverseTimeDecay` (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/InverseTimeDecay) to hyperbolically decrease the learning rate to 1/2 of the base rate at 1,000 epochs, 1/3 at 2,000 epochs, and so on.

```
step = np.linspace(0,100000)
lr = lr_schedule(step)
plt.figure(figsize = (8,6))
plt.plot(step/STEPS_PER_EPOCH, lr)
plt.ylim([0,max(plt.ylim())])
plt.xlabel('Epoch')
_ = plt.ylabel('Learning Rate')
```

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 prints a `.` for each epoch, and a full set of metrics every 100 epochs.

Next include `tf.keras.callbacks.EarlyStopping` (https://www.tensorflow.org/api_docs/python/tf/keras/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`** (https://www.tensorflow.org/api_docs/python/tf/keras/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', patier
    tf.keras.callbacks.TensorBoard(logdir/name),
  ]
```

Similarly each model will use the same **`Model.compile`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) and **`Model.fit`** (https://www.tensorflow.org/api_docs/python/tf/keras/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),
                metrics=[
                  tf.keras.metrics.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=0)
  return history
```

Tiny model

Start by training a model:

```
tiny_model = tf.keras.Sequential([
    layers.Dense(16, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(1)
])
```

```
size_histories = {}
```

```
size_histories['Tiny'] = compile_and_fit(tiny_model, 'sizes/Tiny')
```

Now check how the model did:

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

Small model

To check if you can beat the performance of the small model, progressively train some larger models.

Try two hidden layers with 16 units each:

```
small_model = tf.keras.Sequential([
    # `input_shape` is only required here so that `.summary` works.
    layers.Dense(16, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(16, activation='elu'),
    layers.Dense(1)
])
```

```
size_histories['Small'] = compile_and_fit(small_model, 'sizes/Small')
```

Medium model

Now try three hidden layers with 64 units each:

```
medium_model = tf.keras.Sequential([
    layers.Dense(64, activation='elu', input_shape=(FEATURES,)),
    layers.Dense(64, activation='elu'),
    layers.Dense(64, activation='elu'),
    layers.Dense(1)
])
```

And train the model using the same data:

```
size_histories['Medium']  = compile_and_fit(medium_model, "sizes/Medium")
```

Large model

As an exercise, you can create an even larger model and check how quickly it begins overfitting. Next, add to this benchmark a network that has much more capacity, far more than the problem would warrant:

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

And, again, train the model using the same data:

```
size_histories['large'] = compile_and_fit(large_model, "sizes/large")
```

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 figure out 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]")
```

Note: All the above training runs used the **`callbacks.EarlyStopping`** (https://www.tensorflow.org/api_docs/python/tf/keras/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.

Open an embedded TensorBoard viewer inside a notebook (Sorry, this doesn't display on tensorflow.org):

```
# Load the TensorBoard notebook extension
%load_ext tensorboard

# Open an embedded TensorBoard viewer
%tensorboard --logdir {logdir}/sizes
```

You can view the results of a previous run (https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97) of this notebook on TensorBoard.dev (https://tensorboard.dev/).

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')
```

```
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 demonstrated 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 (https://developers.google.com/machine-learning/glossary/#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 (https://developers.google.com/machine-learning/glossary/#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` (https://www.tensorflow.org/api_docs/python/tf/keras), weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Add L2 weight regularization:

```
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 demonstrated in the diagram above, the "`L2`" regularized model is now much more competitive with 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:

1. If you are writing your own training loop, then you need to be sure to ask the model for its regularization losses.

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

1. 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 used in optimizers like `tf.keras.optimizers.Ftrl` (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Ftrl) and `tfa.optimizers.AdamW` (https://www.tensorflow.org/addons/api_docs/python/tfa/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. For example, 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 Keras, you can introduce dropout in a network via the **`tf.keras.layers.Dropout`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout) layer, which gets applied to the output of layer right before.

Add two dropout layers to your network to check 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, "regularizer
```

```
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, "regulari
```

```
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 run the following into a code-cell (Sorry, this doesn't display on tensorflow.org):

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

You can view the results of a previous run (https://tensorboard.dev/experiment/vW7jmmF9TmKmy3rbheMQpw/#scalars&_smoothingWeight=0.97) of this notebook on TensorBoard.dev (https://tensorboard.dev/).

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 (https://www.tensorflow.org/tutorials/images/data_augmentation)
- Batch normalization (`tf.keras.layers.BatchNormalization` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization))

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

```
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```


Last updated 2024-04-03 UTC.