Explore overfitting and underfitting

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 movie reviews, and predicting fuel efficiency—we saw that the accuracy of our model on the validation data would peak after training for a number of epochs, and would then start decreasing.

In other words, our 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 we really want is to develop models that generalize well to a *testing data* (or data they haven't seen before).

The opposite of overfitting is *underfitting*. Underfitting occurs when there is still room for improvement on the test 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. We need to strike a balance. Understanding how to train for an appropriate number of epochs as we'll explore below is a useful skill.

To prevent overfitting, the best solution is to use more training data. A model trained on more 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, we'll explore two common regularization techniques—weight regularization and dropout—and use them to improve our IMDB movie review classification notebook.

```
from __future__ import absolute_import, division, print_function

import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
1.13.0-rc2
```

Download the IMDB dataset

Rather than using an embedding as in the previous notebook, here we will multi-hot encode the sentences. This model will quickly overfit to the training set. It will be used to demonstrate when overfitting occurs, and how to fight it.

Multi-hot-encoding our lists means turning them into vectors of 0s and 1s. Concretely, this would mean for instance turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all-zeros except for indices 3 and 5, which would be ones.

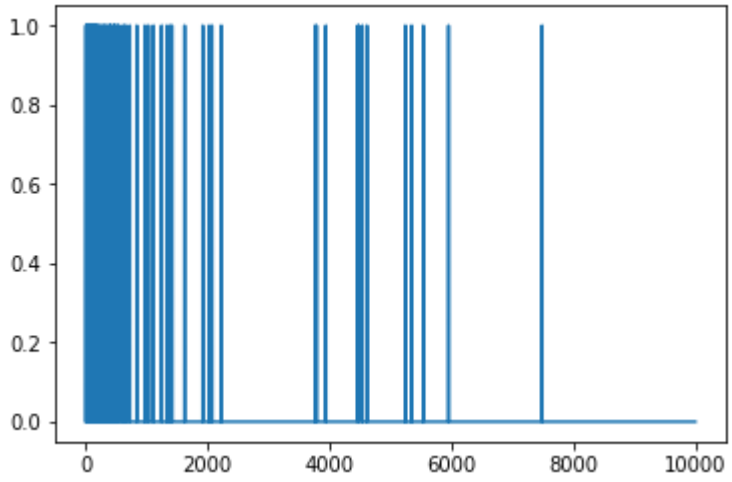
```
NUM_WORDS = 10000

(train_data, train_labels), (test_data, test_labels) = keras.datasets.imdb.load_

def multi_hot_sequences(sequences, dimension):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, word_indices in enumerate(sequences):
        results[i, word_indices] = 1.0  # set specific indices of results[i] to
    return results


train_data = multi_hot_sequences(train_data, dimension=NUM_WORDS)
test_data = multi_hot_sequences(test_data, dimension=NUM_WORDS)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
17465344/17464789 [==============================] - 0s 0us/step
```

Let's look at one of the resulting multi-hot vectors. The word indices are sorted by frequency, so it is expected that there are more 1-values near index zero, as we can see in this plot:

```
plt.plot(train_data[0])
```

```
[<matplotlib.lines.Line2D at 0x7f2a85ff6a20>]
```



Demonstrate overfitting

The simplest way to prevent overfitting is to reduce the size of the model, i.e. the number of learnable parameters in the model (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. Let's try this on our movie review classification network.

We'll create a simple model using only `Dense` layers as a baseline, then create smaller and larger versions, and compare them.

Create a baseline model

```
baseline_model = keras.Sequential([
    # `input_shape` is only required here so that `.summary` works.
    keras.layers.Dense(16, activation=tf.nn.relu, input_shape=(NUM_WORDS,)),
    keras.layers.Dense(16, activation=tf.nn.relu),
    keras.layers.Dense(1, activation=tf.nn.sigmoid)
])

baseline_model.compile(optimizer='adam',
                       loss='binary_crossentropy',
                       metrics=['accuracy', 'binary_crossentropy'])

baseline_model.summary()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-packages/tensorflow/pytho
Instructions for updating:
Colocations handled automatically by placer.
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                160016
_________________________________________________________________
dense_1 (Dense)              (None, 16)                272
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 160,305
Trainable params: 160,305
```

```
baseline_history = baseline_model.fit(train_data,
                                      train_labels,
                                      epochs=20,
```

```
                                    batch_size=512,
                                    validation_data=(test_data, test_labels),
                                    verbose=2)
```

```
Train on 25000 samples, validate on 25000 samples
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-packages/tensorflow/pytho
Instructions for updating:
Use tf.cast instead.
Epoch 1/20
 - 3s - loss: 0.4825 - acc: 0.8103 - binary_crossentropy: 0.4825 - val_loss: 0.
Epoch 2/20
 - 3s - loss: 0.2450 - acc: 0.9130 - binary_crossentropy: 0.2450 - val_loss: 0.
Epoch 3/20
 - 3s - loss: 0.1798 - acc: 0.9373 - binary_crossentropy: 0.1798 - val_loss: 0.
Epoch 4/20
 - 3s - loss: 0.1464 - acc: 0.9488 - binary_crossentropy: 0.1464 - val_loss: 0.
Epoch 5/20
```

Create a smaller model

Let's create a model with less hidden units to compare against the baseline model that we just created:

```
smaller_model = keras.Sequential([
    keras.layers.Dense(4, activation=tf.nn.relu, input_shape=(NUM_WORDS,)),
    keras.layers.Dense(4, activation=tf.nn.relu),
    keras.layers.Dense(1, activation=tf.nn.sigmoid)
])

smaller_model.compile(optimizer='adam',
                loss='binary_crossentropy',
                metrics=['accuracy', 'binary_crossentropy'])

smaller_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

```
=================================================================
dense_3 (Dense)              (None, 4)                 40004
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 20
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 5
=================================================================
Total params: 40,029
Trainable params: 40,029
Non-trainable params: 0
_________________________________________________________________
```

And train the model using the same data:

```
smaller_history = smaller_model.fit(train_data,
                                    train_labels,
                                    epochs=20,
                                    batch_size=512,
                                    validation_data=(test_data, test_labels),
                                    verbose=2)
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/20
 - 3s - loss: 0.6322 - acc: 0.7314 - binary_crossentropy: 0.6322 - val_loss: 0.
Epoch 2/20
 - 3s - loss: 0.4423 - acc: 0.8709 - binary_crossentropy: 0.4423 - val_loss: 0.
Epoch 3/20
 - 3s - loss: 0.3204 - acc: 0.9000 - binary_crossentropy: 0.3204 - val_loss: 0.
Epoch 4/20
 - 3s - loss: 0.2566 - acc: 0.9160 - binary_crossentropy: 0.2566 - val_loss: 0.
Epoch 5/20
 - 3s - loss: 0.2176 - acc: 0.9285 - binary_crossentropy: 0.2176 - val_loss: 0.
Epoch 6/20
 - 3s - loss: 0.1912 - acc: 0.9363 - binary_crossentropy: 0.1912 - val_loss: 0.
Epoch 7/20
```

Create a bigger model

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

```
bigger_model = keras.models.Sequential([
    keras.layers.Dense(512, activation=tf.nn.relu, input_shape=(NUM_WORDS,)),
    keras.layers.Dense(512, activation=tf.nn.relu),
    keras.layers.Dense(1, activation=tf.nn.sigmoid)
])

bigger_model.compile(optimizer='adam',
                     loss='binary_crossentropy',
                     metrics=['accuracy','binary_crossentropy'])

bigger_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 512)               5120512
_________________________________________________________________
dense_7 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 513
=================================================================
Total params: 5,383,681
Trainable params: 5,383,681
Non-trainable params: 0
_________________________________________________________________
```

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

```
bigger_history = bigger_model.fit(train_data, train_labels,
                                  epochs=20,
                                  batch_size=512,
                                  validation_data=(test_data, test_labels),
                                  verbose=2)
```

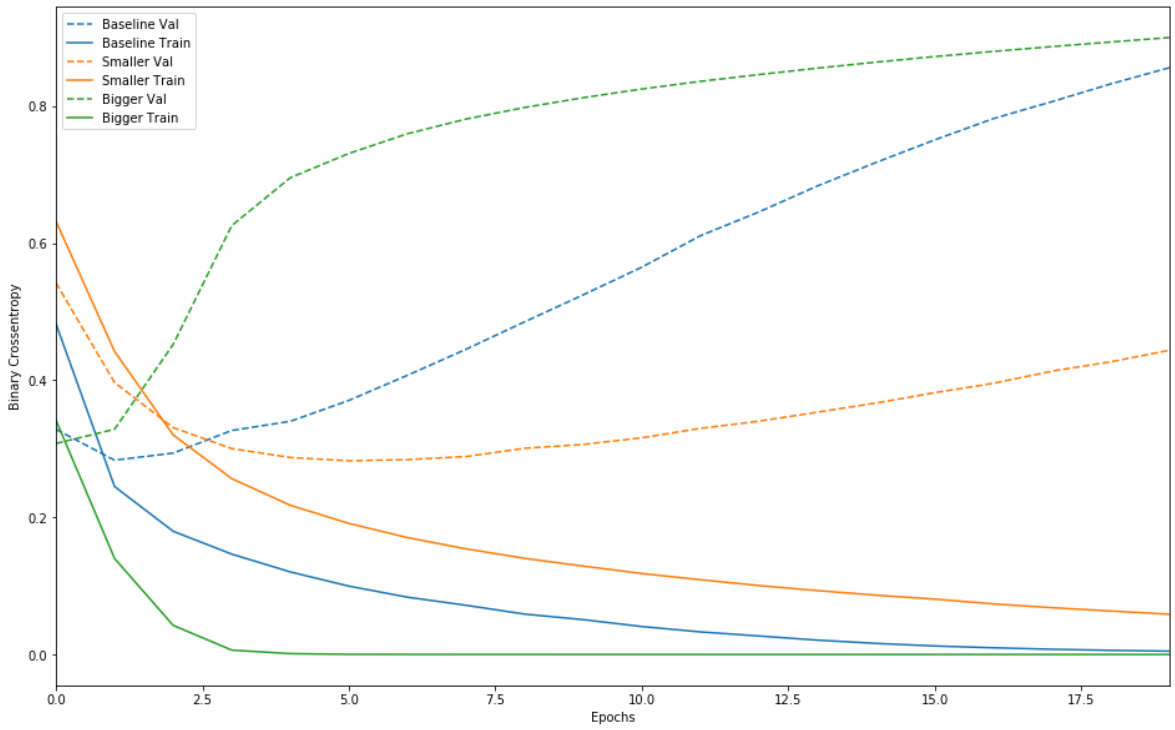
```
Train on 25000 samples, validate on 25000 samples
Epoch 1/20
 - 7s - loss: 0.3419 - acc: 0.8534 - binary_crossentropy: 0.3419 - val_loss: 0.
Epoch 2/20
```

```
 - 6s - loss: 0.1401 - acc: 0.9491 - binary_crossentropy: 0.1401 - val_loss: 0.
Epoch 3/20
 - 6s - loss: 0.0425 - acc: 0.9879 - binary_crossentropy: 0.0425 - val_loss: 0.
Epoch 4/20
 - 6s - loss: 0.0063 - acc: 0.9991 - binary_crossentropy: 0.0063 - val_loss: 0.
Epoch 5/20
 - 6s - loss: 0.0012 - acc: 1.0000 - binary_crossentropy: 0.0012 - val_loss: 0.
Epoch 6/20
 - 7s - loss: 2.2473e-04 - acc: 1.0000 - binary_crossentropy: 2.2473e-04 - val_
```

Plot the training and validation loss

The solid lines show the training loss, and the dashed lines show the validation loss (remember: a lower validation loss indicates a better model). Here, the smaller network begins overfitting later than the baseline model (after 6 epochs rather than 4) and its performance degrades much more slowly once it starts overfitting.

```
def plot_history(histories, key='binary_crossentropy'):
  plt.figure(figsize=(16,10))

  for name, history in histories:
    val = plt.plot(history.epoch, history.history['val_'+key],
                   '--', label=name.title()+' Val')
    plt.plot(history.epoch, history.history[key], color=val[0].get_color(),
             label=name.title()+' Train')

  plt.xlabel('Epochs')
  plt.ylabel(key.replace('_',' ').title())
  plt.legend()

  plt.xlim([0,max(history.epoch)])


plot_history([('baseline', baseline_history),
              ('smaller', smaller_history),
              ('bigger', bigger_history)])
```



Notice that the larger network begins overfitting almost right away, after just one epoch, and overfits much more severely. The more capacity the network has, the quicker it will be able to model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

Strategies

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 (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.

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. Let's add L2 weight regularization now.

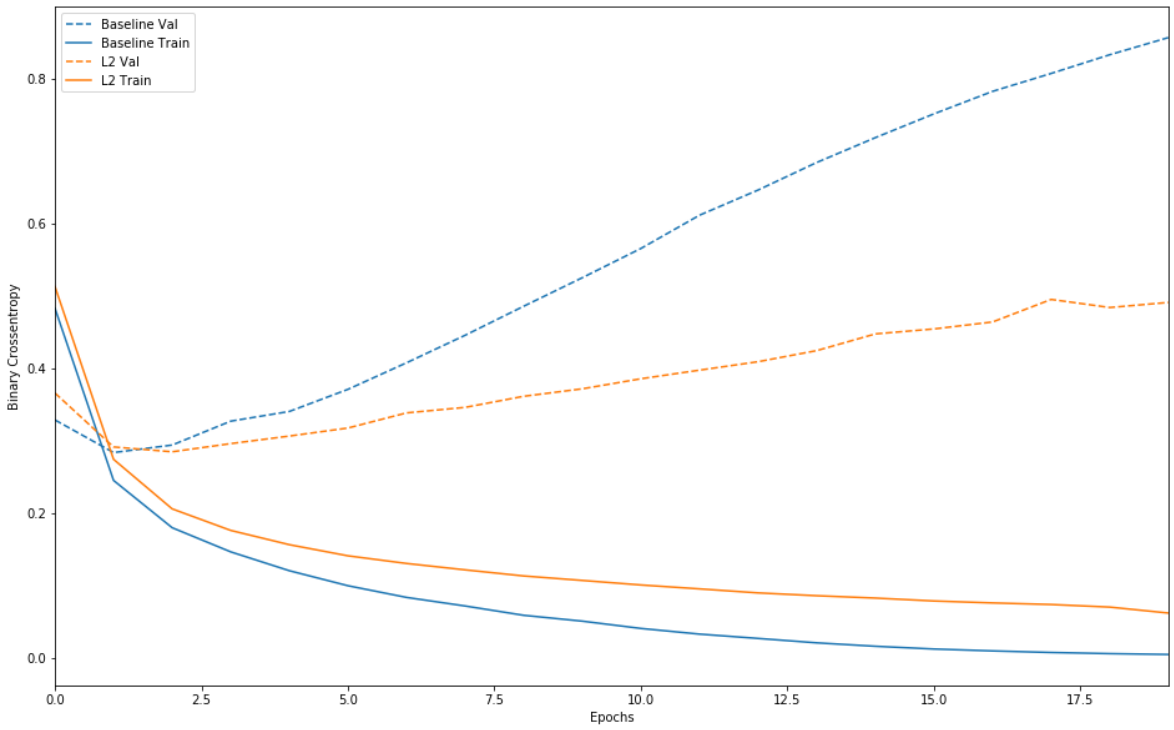
```
l2_model = keras.models.Sequential([
    keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001),
                       activation=tf.nn.relu, input_shape=(NUM_WORDS,)),
    keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001),
                       activation=tf.nn.relu),
    keras.layers.Dense(1, activation=tf.nn.sigmoid)
])

l2_model.compile(optimizer='adam',
                 loss='binary_crossentropy',
                 metrics=['accuracy', 'binary_crossentropy'])

l2_model_history = l2_model.fit(train_data, train_labels,
                                epochs=20,
                                batch_size=512,
                                validation_data=(test_data, test_labels),
                                verbose=2)
```

```
Train on 25000 samples, validate on 25000 samples
Epoch 1/20
 - 3s - loss: 0.5516 - acc: 0.7788 - binary_crossentropy: 0.5125 - val_loss: 0.
Epoch 2/20
 - 2s - loss: 0.3171 - acc: 0.9066 - binary_crossentropy: 0.2741 - val_loss: 0.
Epoch 3/20
 - 3s - loss: 0.2547 - acc: 0.9286 - binary_crossentropy: 0.2058 - val_loss: 0.
Epoch 4/20
 - 2s - loss: 0.2287 - acc: 0.9406 - binary_crossentropy: 0.1760 - val_loss: 0.
Epoch 5/20
 - 3s - loss: 0.2117 - acc: 0.9483 - binary_crossentropy: 0.1563 - val_loss: 0.
Epoch 6/20
 - 3s - loss: 0.1986 - acc: 0.9536 - binary_crossentropy: 0.1410 - val_loss: 0.
Epoch 7/20
```

`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. Note that because this penalty is only added at training time, the loss for this network will be much higher at training than at test time.

Here's the impact of our L2 regularization penalty:

```
plot_history([('baseline', baseline_history),
              ('l2', l2_model_history)])
```



As you can see, the L2 regularized model has become much more resistant to overfitting than the baseline model, even though both models have the same number of parameters.

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. 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.

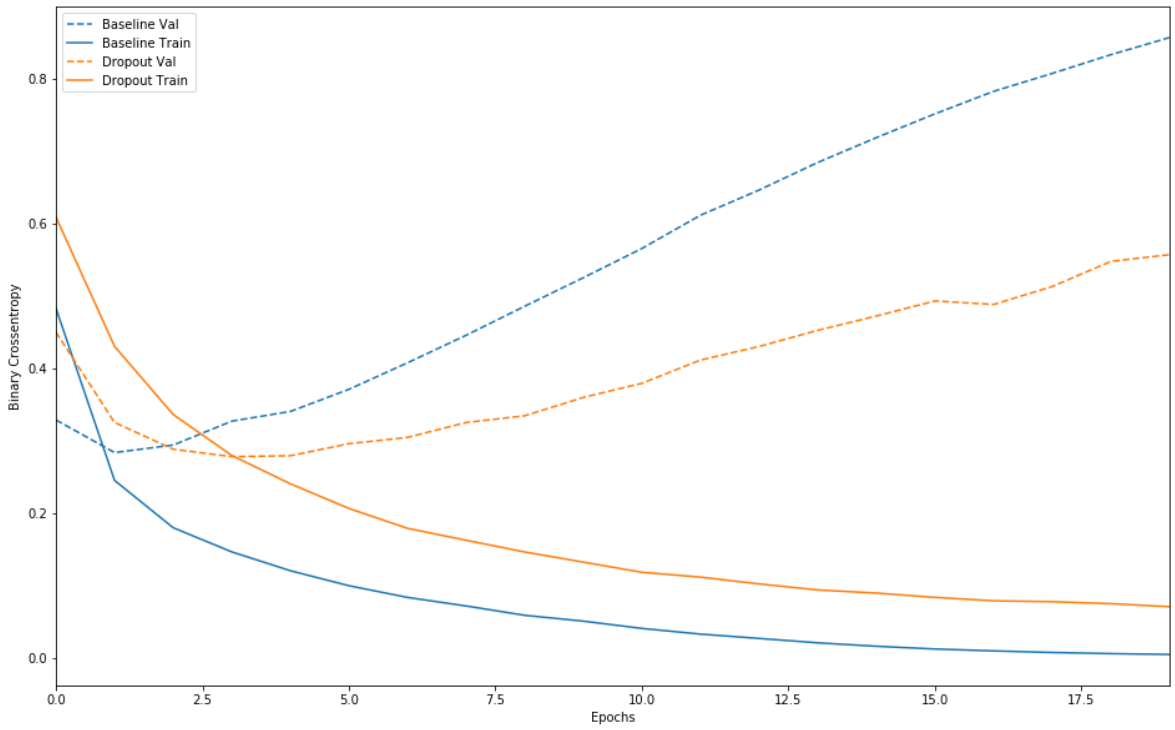
In tf.keras you can introduce dropout in a network via the Dropout layer, which gets applied to the output of layer right before.

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

```
dpt_model = keras.models.Sequential([
    keras.layers.Dense(16, activation=tf.nn.relu, input_shape=(NUM_WORDS,)),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(16, activation=tf.nn.relu),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1, activation=tf.nn.sigmoid)
])

dpt_model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy','binary_crossentropy'])

dpt_model_history = dpt_model.fit(train_data, train_labels,
                                  epochs=20,
                                  batch_size=512,
                                  validation_data=(test_data, test_labels),
                                  verbose=2)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-packages/tensorflow/pytho
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - kee
Train on 25000 samples, validate on 25000 samples
Epoch 1/20
 - 3s - loss: 0.6084 - acc: 0.6597 - binary_crossentropy: 0.6084 - val_loss: 0.
Epoch 2/20
 - 3s - loss: 0.4300 - acc: 0.8197 - binary_crossentropy: 0.4300 - val_loss: 0.
Epoch 3/20
 - 3s - loss: 0.3360 - acc: 0.8773 - binary_crossentropy: 0.3360 - val_loss: 0.
Epoch 4/20
 - 3s - loss: 0.2795 - acc: 0.9024 - binary_crossentropy: 0.2795 - val_loss: 0.
Epoch 5/20
```

```
plot_history([('baseline', baseline_history),
              ('dropout', dpt_model_history)])
```



Adding dropout is a clear improvement over the baseline model.

To recap: here 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.

And two important approaches not covered in this guide are data-augmentation and batch normalization.

```
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# 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.
```