Regularization

Overfitting and Underfitting

In the house price regression exercise, we could notice that the performance of our model on the held-out validation data would always peak after a few epochs and would then start degrading, i.e. our model would quickly start to overfit to the training data. Overfitting happens in every single machine learning problem.
To prevent a model from learning misleading or irrelevant patterns found in the training data, the best solution is of course to get 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 modulate the quantity of information that your model is allowed to store, or to add constraints on what information it is allowed to 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.The processing of fighting overfitting in this way is called regularization.
Exe. 1 Load IMDB dataset from Keras with maximum number of $num\_words = 10000$ to include. It is a large dataset containing the text of $50,000$ movie reviews from the Internet Movie Database. This is a dataset for binary sentiment classification of movies containing $\{0, 1\}$. Check train and test datasets sizes and shapes.
Exe. 2 Prepare the data using the following code:

```
def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension
    )
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of
    results[i] to 1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

check new train and test sets shapes and sizes. What have changed?
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). 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. 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 this mapping as easily, and thus, in order to minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets – precisely the type of representations that we are interested in. At the same time, keep in mind that you should be using models that have enough parameters that they won't be underfitting: your model shouldn't be starved for memorization resources. There is a compromise to be found between "too much capacity" and "not enough capacity".
Unfortunately, there is no magical formula to determine what the right number of layers is, or what the right size for each layer is. You will have to evaluate an array of different architectures (on your validation set, not on your test set) in order to find the right model size for your data. The general workflow to find an appropriate model size is to start with relatively few layers and parameters, and start increasing the size of the layers or adding new layers until you see diminishing returns with regard to the validation loss.
Exe. 3 reducing the network size design a neural network model namely `original_model` with two dense layers of size 16 and each followed with an activation function `relu`. Add a final layer of 1 node and an activation function of `sigmoid`. For your network compilation, define a `rmsprop` optimizer, with a `binary_crossentropy` and `acc` as the metric.
Exe. 4 Design a neural network model namely `smaller_model` with two dense layers of size 4 and each followed with an activation function `relu`. Add a final layer of 1 node and an activation function of `sigmoid`. For your network compilation, define a `rmsprop` optimizer, with a `binary_crossentropy` and `acc` as the metric.
Exe. 5 Fit the original model on the train set with the test set as the `validation_set`, 20 `epochs` and `batch_size = 512` and save it in `original_hist`.
Exe. 6 Fit the smaller model on the train set with the test set as the `validation_set`, 20 `epochs` and `batch_size = 512` and save it in `smaller_hist`.
Exe. 7 Get the `val_loss` from the trained model histogram and save them in new variables.

Exe. 8 Plot the validation loss values w.r.t the epochs (we have 20 epochs) and observe the loss value changing trough the epochs increase. What do you observe? which network starts overfitting later? after hom many epochs each network starts overfitting?. Notice that the loss should reduce during the epochs changing, as soon as the loss starts to increase, overfitting happens. (you should have a figure more or less similar to figure 1)
Exe. 9 Design a network that has much more capacity, far more than the problem would warrant. Call this model `bigger_model` with two dense layers of size

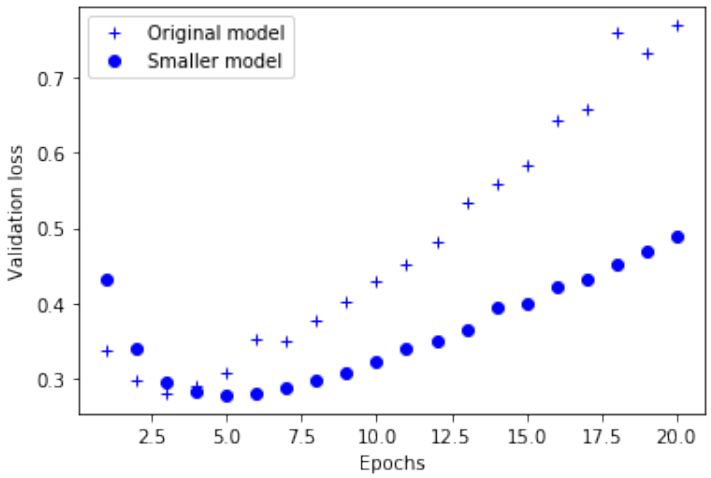

Figure 1: validation loss changing w.r.t epochs for original and small models.

512 and each followed with an activation function `relu`. Add a final layer of 1 node and an activation function of `sigmoid`. For your network compilation, define a `rmsprop` optimizer, with a `binary_crossentropy` and `acc` as the metric.
Exe. 10 Fit the bigger model on the train set with the test set as the `validation_set`, 20 `epochs` and `batch_size = 512` and save it in `bigger_hist`.
Exe. 11 Plot the bigger and original validation loss changing w.r.t epochs changing. Similar to exercise 18. How the bigger network loss changes regarding to the original network. The more capacity the network has, the quicker it will be able to model the training data, but if it converges quickly to 0, this is more susceptible for overfitting!

weight regularization

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 is called "Occam's Razor principle". 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 above). Thus a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to only 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 "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 Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments: for instance, `kernel_regularizer= regularizers.l1(0.001)` where $l2(0.001)$ means that every coefficient in the weight matrix of the layer will add 0.001 * weight_coefficient_value 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.
Exe. 12 Modify the original network as `l2_model` by adding an $L2$ weight regularization to the model[1]. Add the L2 regularizers to the first two layers.
Exe. 13 Fit the model on train set with previously presented parameters.
Exe. 14 Plot the validation loss w.r.t epochs changing for original model and `l2_model`.Observe, the model with L2 regularization and see how it is more resistant to overfitting than the original model, even though both models have the same number of parameters.
As alternatives to $L2$ regularization, you could use one of the following Keras weight regularizers:

```
from keras import regularizers

# L1 regularization
regularizers.l1(0.001)

# L1 and L2 regularization at the same time
regularizers.l1_l2(l1=0.001, l2=0.001)
```

Adding Dropout

Dropout, applied to a layer, consists of randomly "dropping out" (i.e. setting 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.
The dropout can be added to the model in Keras using `layers.Dropout(0.5)` where 0.5 is the dropout rate indicating 50% of nodes should be dropped for the following layer randomly. The dropout should be applied to the output of

[1]For more information use Keras documentation: `https://keras.io/regularizers/`

layer right before the layer.
Exe. 15 Modify the original network by adding a dropout after each layer of 16 nodes. Name this model `dpt_model`
Exe. 16 Fit the model on the set with previously presented parameters.
Exe. 17 Plot the validation loss changing w.r.t the epochs changing for two original and dropout model. What are your observations? Do you see any improvement over the original network?

Exe. 1 Load House Price DataSet The goal is to predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on. Load the `boston_housing` dataset in train and test and check the data shape and sizes.

```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass
from keras.datasets import boston_housing
```

```
(train_data, train_targets), (test_data, test_targets) =
    boston_housing.load_data()
```

The targets are the median values of owner-occupied homes, in thousands of dollars. Print them to check their data [2].
Exe. 2 Normalizing data For each feature in the input data (a column in the input data matrix), subtract the mean of the feature and divide by the standard deviation i.e. $\forall x,\ x = \frac{x-\mu}{\sigma}$. In this way the feature is centered around 0 and has a unit standard deviation. Hint: use `.mean(axis=0)` and `.std(axis=0)` for computing mean and standard deviation.
Exe. 3 Model definition Because so few samples are available, use a very small network with two hidden layers, each with 64 units and *relu* activation. To have a linear layer output, add a final layer with a single unit and no activation function. Compile the network with the `mse` (mean squared error) loss function. Monitoring a new metric during training: mean absolute error (`mae`)[3]. Finally define the `optimizer = 'rmsprop'` in the model compilation. Define the model in a function named `build_model()`.

```
from keras import models
from keras import layers
def build_model():
  model = models.Sequential()
  # complete the model here
  return model
```

[2]The prices are typically between $10,000 and $50,000. If that sounds cheap, remember that this was the mid-1970s
[3]It's the absolute value of the difference between the predictions and the targets.

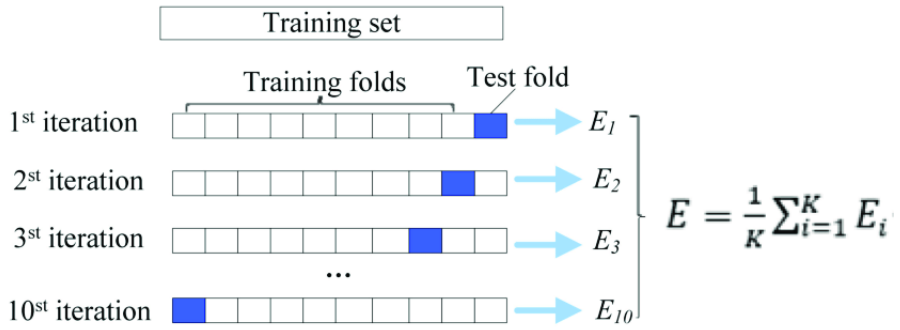

Figure 2: 10-cross validation example.

Notice In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting. Applying an activation function would constrain the range the output can take; for instance, if you applied a `sigmoid` activation function to the last layer,the network could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the network is free to learn to predict values in any range.

Exe. 4 Model Validation To evaluate the network while we keep adjusting its parameters (such as the number of epochs used for training), we can split the data into a training set and a test set (here we name it as a validation set), as you did in machine learning examples. Because of few data points, the validation set would end up being very small (for instance, about 100 examples). As a consequence, the validation scores might change a lot depending on which data points we chose to use for validation and which we chose for training: the validation scores might have a high variance with regard to the validation split. This would prevent you from reliably evaluating our model.

The best practice in such situations is to use K-fold cross-validation (see figure 2). It consists of splitting the available data into K partitions (typically $K = 4$ or 5), instantiating K identical models, and training each one on $K-1$ partitions while evaluating on the remaining partition. The validation score for the model used is then the average of the K validation scores obtained.

```
import numpy as np
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
  print('processing fold #', i)
  val_data = train_data[i * num_val_samples: (i + 1) *
    num_val_samples]
  val_targets = train_targets[i * num_val_samples: (i + 1) *
    num_val_samples]
  partial_train_data = np.concatenate([train_data[:i *
    num_val_samples], train_data[(i + 1) * num_val_samples:]], axis
    =0)
```

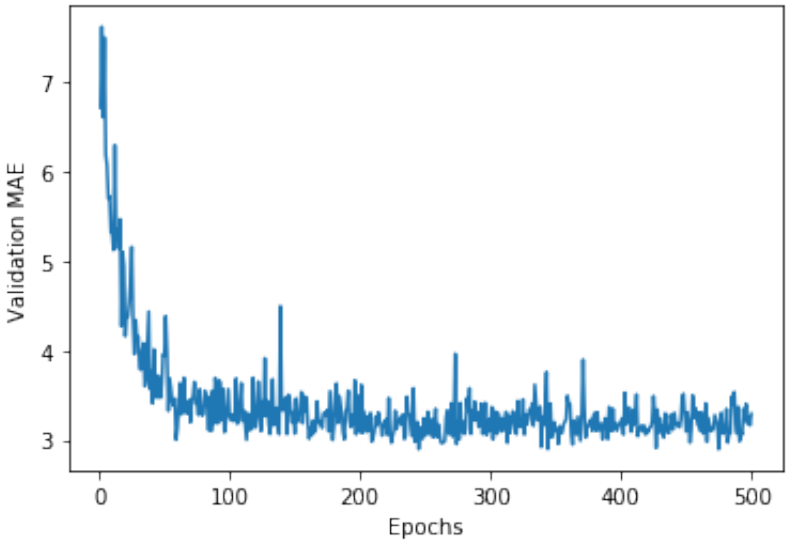

Figure 3: validation mae by epoch

```
  partial_train_targets = np.concatenate([train_targets[:i *
    num_val_samples], train_targets[(i + 1) * num_val_samples:]],
    axis=0)
  # you should call build_model here
  # you should fit model here
  """the following line is for evaluating the test set"""
  val_mse, val_mae = model.evaluate(val_data, val_targets, verbose
    =0)
  all_scores.append(val_mae)
```

Exe. 5 print the overall score of your model and check its average.
Exe. 6 Now training the network a bit longer: 500 epochs. To keep a record of how well the model does at each epoch, we should modify the training loop to save the per-epoch validation score log. Modify the code from the previous exercise in order to save `val_mean_absolute_error` after each epoch for any fold of cross validation. hint: for the aim of this exercise, you can use the following lines in your code:

```
history = model.fit(partial_train_data, partial_train_targets,
    validation_data=(val_data, val_targets), epochs=num_epochs,
    batch_size=1, verbose=0)
  mae_history = history.history['val_mae']
```

Exe. 7 Compute the average of the per-epoch `mae` scores for all folds.
Exe. 8 Plot the average of the per-epoch MAE scores w.r.t epochs changing. You should have a figure similar to 3
Exe. 9 It may be a little difficult to see the plot, due to scaling issues and relatively high variance. Let's do the following:
- Omit the first 10 data points, which are on a different scale than the rest of

the curve.
- Replace each point with an exponential moving average of the previous points,to obtain a smooth curve i.e. $point = previous\_point * factor + point * (1 - factor)$ where factor is a value between 0 and 1. According to this plot, you can check when the validation MAE stops improving or becomes worst. Past the special point on the graph, the model starts overfitting. Once we have finished tuning other parameters of the model (in addition to the number of epochs, we could also adjust the size of the hidden layers), we can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data.

Exe. 10 After finishing tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the hidden layers), train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data from the Exe. 1.