Lecture 03

Learning Process with tf.Keras

CSCI E-89 Deep Learning

Fall 2024

Zoran B. Djordjević & Rahul B. Joglekar

Sources

- Material in this lecture follows:
  - Keras Documentation https://keras.io
  - https://www.tensorflow.org/tutorials and https://keras.io/examples/
  - *Chapter 4, 2nd Edition of Deep Learning with Python* by François Chollet, Chapter 3, 1st Edition. Material is almost identical.
  - Chapters 10 and 11 in the 3rd Edition of Aurelien Geron's book

Objectives

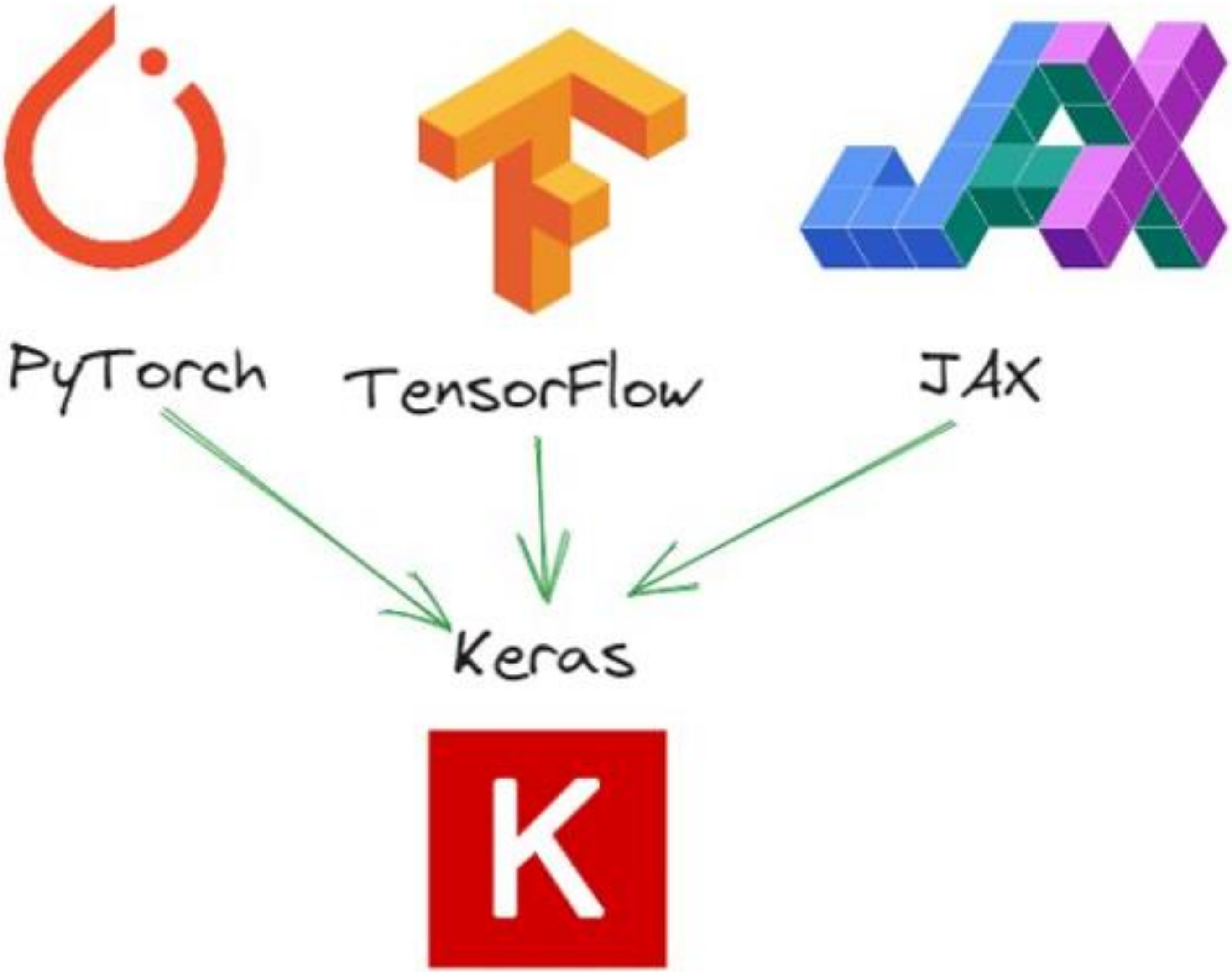
- Objectives of today's lecture are:
  - Introduce Keras
  - Keras 2 Vs Keras 3
  - Introduce Layers, building blocks of NN
  - Describe Learning process, Optimizers, Loss Function, `compile()` and `fit()` methods
  - Illustrate Sequential Neural Networks on
    - Binary classification
    - Regression
  - Understand K-fold Validation
  - Understand underfitting and overfitting

Keras

Keras

- Keras was initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).
- Keras (κέρας) means *horn* in ancient Greek.
- Francois Chollet, a senior Google developer, is the pioneer developer behind Keras and introduced it as an independent API intended to be a user-friendly interface for Theano, a deep learning API developed at Universite de Montreal.
- Keras was developed with a focus on enabling fast experimentation, giving developers *an ability* to *go from idea to result with the least possible delay.*
- A few months later, in 2015, TensorFlow was released and Keras was refactored to work atop of TensorFlow, as well. Eventually, Keras included support for Microsoft's CNTK and Amazon's MXNet.
- Theano and CNTK are not being developed any more and very few people outside Amazon use MXNet. Keras was practically left with TensorFlow as its sole background.
- In 2019 Google adopted Keras as TensorFlow's official high level API. With release of TensorFlow 2.0, tf.Keras API is the front and center of TensorFlow.
- With the release of **Keras 3.0** it now supports JAX, Tensorflow & Pytorch as backends

Keras 3



Keras 2 Vs Keras 3

- **Keras 3** is a multi-framework API, and can be used to develop modular components that are compatible with any framework – **JAX, TensorFlow, or PyTorch.**
- **Keras 2** only supports **tensorflow** as a backend.
- Starting Keras 3, any Keras model can be instantiated as a PyTorch Module, can be exported as a TensorFlow SavedModel, or can be instantiated as a stateless JAX function.
- Also we can use it with PyTorch ecosystem packages, with the full range of TensorFlow deployment & production tools, and with JAX large-scale TPU training infrastructure. Write one model.py using Keras APIs, and get access to everything the ML world has to offer.
- **Cross-framework compatibility** : The Keras fit()/evaluate()/predict() routines are compatible with tf.data.Dataset objects, with PyTorch DataLoader objects, with NumPy arrays, Pandas dataframes – regardless of the backend you're using. You can train a Keras + TensorFlow model on a PyTorch DataLoader or train a Keras + PyTorch model on a tf.data.Dataset.

Keras 2 Vs Keras 3

- Starting **TensorFlow 2.16**, doing pip install tensorflow will install **Keras 3**. When you have TensorFlow >= 2.16 and Keras 3, then by default

  from tensorflow import keras will be Keras 3.

- From **TensorFlow 2.0 to TensorFlow 2.15** (included), doing pip install tensorflow will also install the corresponding version of **Keras 2** – for instance, pip install tensorflow==2.14.0 will install keras==2.14.0. That version of Keras is then available via both import keras and from tensorflow import keras (the tf.keras namespace).

- We DONOT recommend this however should you want tf.keras to stay on Keras 2 after upgrading to **TensorFlow 2.16+,** you can configure your TensorFlow installation so that tf.keras points to tf_keras. Export the environment variable TF_USE_LEGACY_KERAS=1.

Keras 2 Vs Keras 3 Compatibility Matrix

- **JAX compatibility**

  The following Keras + JAX versions are compatible with each other:

  jax==0.4.20 & keras~=3.0

- **TensorFlow compatibility**

  The following Keras + TensorFlow versions are compatible with each other:

  To use Keras 2:

  tensorflow~=2.13.0 & keras~=2.13.0

  tensorflow~=2.14.0 & keras~=2.14.0

  tensorflow~=2.15.0 & keras~=2.15.0

  To use Keras 3:

  tensorflow~=2.16.1 & keras~=3.0

- **PyTorch compatibility**

  The following Keras + PyTorch versions are compatible with each other:

  torch~=2.1.0 & keras~=3.0

Keras 3 Ecosystem

The Keras project isn't limited to the core Keras API for building and training neural networks. It spans a wide range of related initiatives that cover every step of the machine learning workflow.

- **KerasTuner** : KerasTuner is an easy-to-use, scalable hyperparameter optimization framework that solves the pain points of hyperparameter search. KerasTuner comes with Bayesian Optimization, Hyperband, and Random Search algorithms built-in, and is also designed to be easy for researchers to extend in order to experiment with new search algorithms.
- **KerasNLP**: KerasNLP is a toolbox of modular building blocks ranging from pretrained state-of-the-art models, to low-level Transformer Encoder layers.
- **Keras CV** : KerasCV is a repository of modular building blocks (layers, metrics, losses, data-augmentation) that applied computer vision engineers can leverage to quickly assemble production-grade, state-of-the-art training and inference pipelines for common use cases such as image classification, object detection, image segmentation, image data augmentation, etc.
- **AutoKeras** : An AutoML system based on Keras. It is developed by DATA Lab at Texas A&M University. The goal of AutoKeras is to make machine learning accessible to everyone.

Keras CV pre-trained models: https://keras.io/api/keras_cv/models/
Keras NLP pre-trained models: https://keras.io/api/keras_nlp/models/

tf.Keras Principles

- **User friendliness.** tf.Keras is an API designed for human beings, not machines. It puts user experience front and center. tf.Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
- **Modularity.** A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
- **Easy extensibility.** New modules are simple to add (as new classes and functions), and existing modules provide ample examples. tf.Keras lets you easily create new modules allows for total expressiveness, making tf.Keras suitable for advanced research.
- **Work with Python**. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easy to debug, and easy to extend.

tf.Keras Principles

- **Progressive disclosure of complexity** is a design principle used in user interfaces and information presentation to gradually reveal information and functionality as needed
- Using Keras we can start out with simple workflows — such as using Sequential and Functional models and training them with fit() — and when you need more flexibility, can easily customize different components while reusing most of your prior code.user's familiarity and the task at hand.
- For example: We can customize what happens in your training loop while still leveraging the power of fit(), without having to write our own training loop from scratch — just by overriding the train_step method.

tf.Keras Vocabulary

These are some common definitions that are necessary to know and understand to correctly utilize `tf.Keras`:

- **Sample**: one element of a dataset.
  - Example: one image is a sample in image classification using CNN.
  - Example: one audio file is a sample for a speech recognition model
- **Batch**: a set of N samples.
  - All samples in a batch are processed independently, in parallel. In training, a batch results in only one update to the model.
  - A batch generally approximates the distribution of the input data better than a single input. The larger the batch, the better the approximation. Large batches take longer to process and still result in only one update.
- **Micro or Mini Batches** are groups of samples: 64, 128, … we use for training in Stochastic Gradient Descent. We pick a (mini-)batch size that is as large as we could afford without running out of memory. Larger batches will usually result in faster evaluating/prediction).
- **Epoch**: an arbitrary cutoff, usually defined as "one pass over the entire dataset", used to separate training into distinct phases, which is useful for logging and periodic evaluation.
  - When using `validation_data` or `validation_split` with the `fit` method of `tf.Keras` models, evaluation will be run at the end of every epoch.
  - Within `tf.Keras`, there is the ability to add `callbacks` specifically designed to be run at the end of an epoch. Examples of callbacks are ***learning rate changes*** and ***model checkpointing*** (saving).

Installing tf.keras & Keras

- If you have TensorFlow installed, tf.Keras is already there as `tf.keras`
- In this class, we will use `tf.keras,` and later switch to Keras 3.
- At the beginning of your Python program or Jupyter notebook you import Keras in the following way:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
```

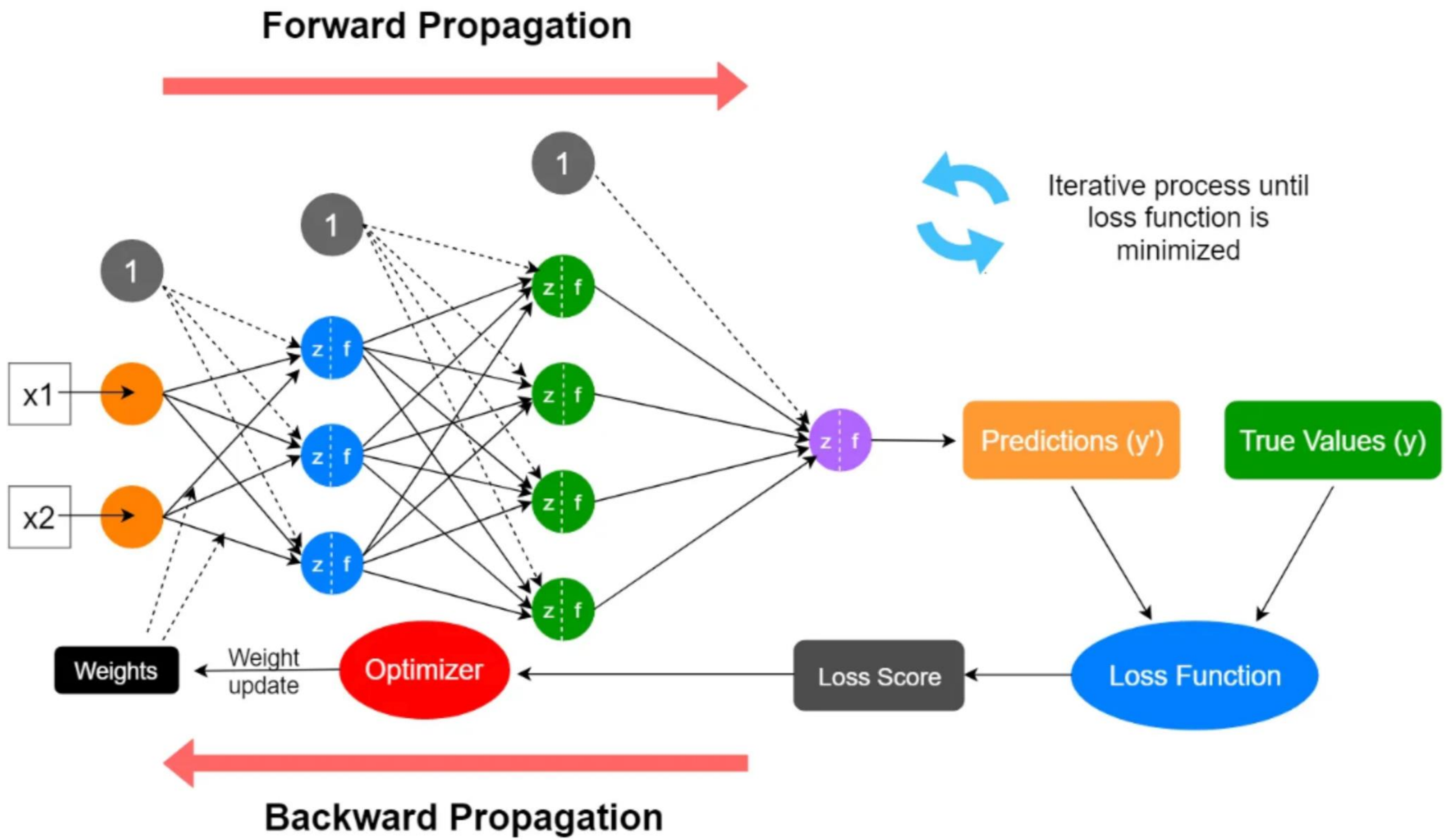
- You might need the following tools as well:
  - cuDNN (recommended if you plan on running tf.Keras on GPU).
  - IHDF5 and `h5py` (required if you plan on saving tf.Keras models to disk).
- Install `graphviz` and `pydot` (used by visualization utilities to plot model graphs).
- If you want to install Keras 3, use Python's `pip:`

```
$ pip install keras. < DO NOT do this at this time >
```

We will use “Keras 2 only” for next few lectures & later move to Keras 3

The building blocks of deep neural networks

Building Blocks of NN

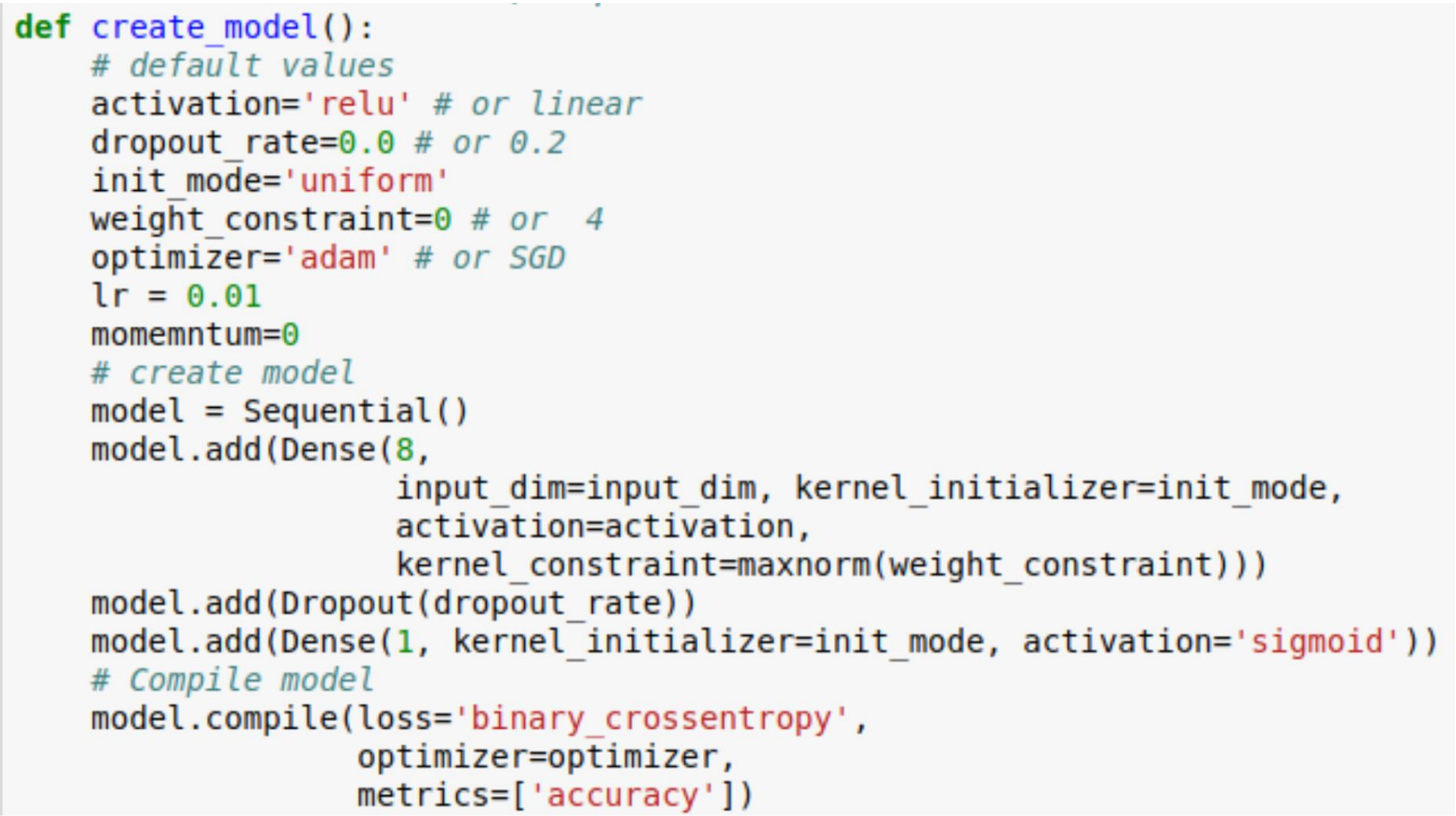


Building Blocks of NN

- In previous lectures we spoke about layers of artificial neurons as the building blocks of neural networks. Keras, TensorFlow, PyTorch and other frameworks organize those layers as objects of type `Layer`. `tf.keras` module containing different layers is called `tf.keras.layers`
- A layer in which every neuron accepts all inputs or outputs from the previous layer is of the type: `tf.keras.layers.Dense`.
- Training a neural network revolves around the following objects:
  - ***Layers***, which are combined into a *network* (or *model*)
  - ***Input data*** and corresponding ***targets*** *or* ***labels***
  - ***Loss function***, which defines the feedback signal used for learning
  - ***Optimizer***, which determines how learning proceeds
- Network, composed of ***layers*** that are chained together, maps the input data to predictions.
- ***Loss function*** compares these predictions to the targets, producing a loss value.
- ***Loss value*** is a measure of how well the network's predictions match what was expected.
- ***Optimizer*** uses the loss values to update the network's weights in the process of Gradient Descent.
- As we will see in the following slides, `tf.Keras` makes assembly of neural networks from the above building blocks quite straightforward.

Let’s make it real !

- Below is a simple Vanilla model written in Keras using all building blocks.
- We will see these in detail



```
def create_model():
    # default values
    activation='relu' # or linear
    dropout_rate=0.0 # or 0.2
    init_mode='uniform'
    weight_constraint=0 # or  4
    optimizer='adam' # or SGD
    lr = 0.01
    momemntum=0
    # create model
    model = Sequential()
    model.add(Dense(8,
                    input_dim=input_dim, kernel_initializer=init_mode,
                    activation=activation,
                    kernel_constraint=maxnorm(weight_constraint)))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, kernel_initializer=init_mode, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
```

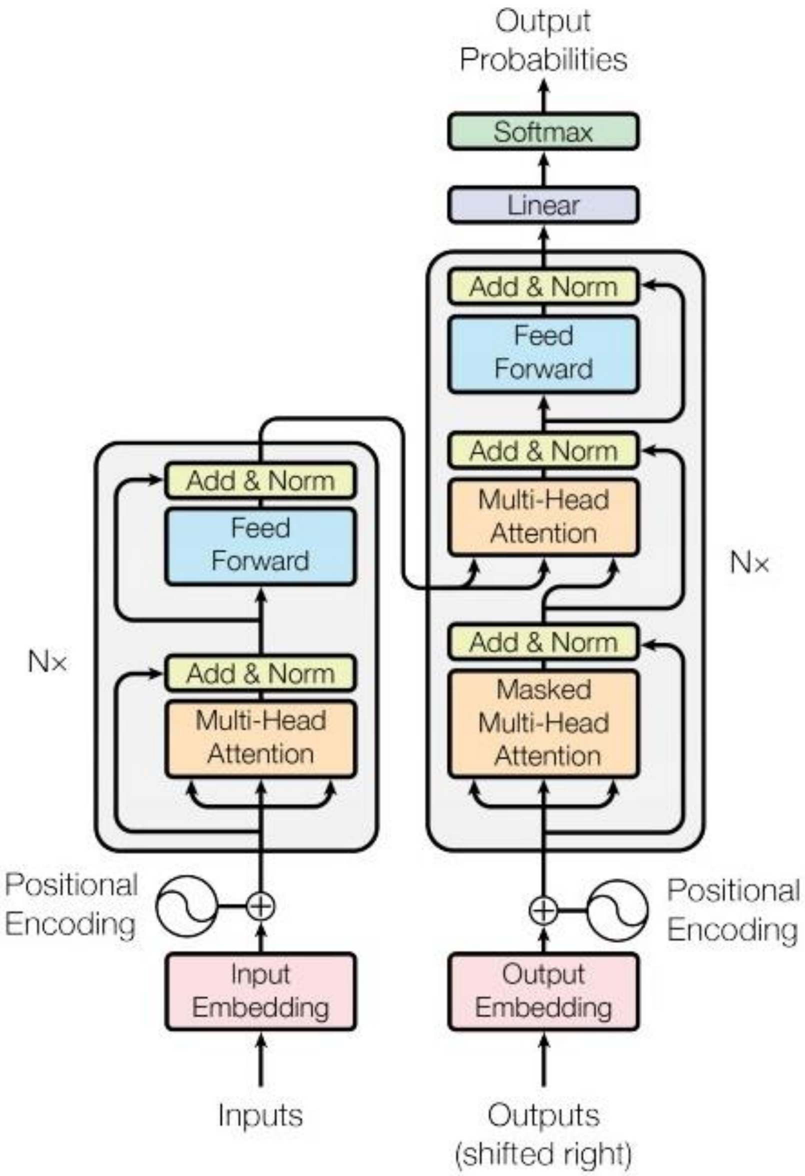
Layers

Layers

- The most fundamental data structure in a neural network is the *layer.*
- A layer is a data-processing module that takes as input one or more tensors and outputs one or more tensors. Some layers are stateless, but many types of layers have a state.
- *The state is represented by the layer's weights*, or one or several tensors learned during the stochastic gradient descent, which together contain the network's *knowledge*.
- Different layers are appropriate for different tensor formats and different types of data processing.
- For instance, simple vector data, stored in 2D tensors of shape (`samples, features`), is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the `Dense` class in `tf.keras.layers` package).
- Sequence data, stored in 3D tensors of shape (`samples, timesteps, features`), is typically processed by *recurrent* layers such as an `LSTM` layer.
- Image data, stored in 4D tensors, is usually processed by 2D convolution layers (`Conv2D`).
- `tf.Keras` treats layers as the LEGO blocks of deep learning. “Lego blocks” is a very appropriate metaphor for `tf.keras` layers.
- Deep-learning models in `tf.keras` is done by clipping together compatible layers to form useful data-transformation pipelines.
- When Layers are just stacked one atop another we speak about **`Sequential`** network or topology.

From Layers to Models

- A deep-learning model is a graph of layers. In Keras, that is the Model class.
- `Sequential` models (a subclass of Model), are simple stack of layers, mapping a single tensor input to a single tensor output.
- NN can work with much broader variety of network topologies. Some common topologies are:
  - Two-branch networks
  - Multi-head networks
  - Residual connections
  - Inception, others
- Network topology can be involved. On the right is the topology of the graph of layers of a `Transformer`, a kay architecture pattern used for analysis of text, in Large Language Models (LLMs) like ChatGPT.
- The topology of a network defines a *hypothesis space*. Within every hypothesis space we will search for an optimal set of values for the weight tensors of all layers.
- Collection of weights is learned knowledge.
- Picking the right network architecture is more an art than a science



Creating Layers

- In several examples, we will work with standard MNIST images of handwritten digits of dimension 28x28 = 784 pixels.
- For example, we create a layer that will only accept as its input tensors, the first dimension of those tensors is 784. Axis 1, the batch dimension, is unspecified, and thus any number of samples could be accepted.

```
from tensorflow.keras import layers
layer = layers.Dense(32, input_shape=(784,))
```

- This layer will return a tensor where the first dimension has been set to 32. Thus, this layer can only be connected to a downstream layer that expects 32- dimensional vectors as its input.
- When using `tf.Keras`, as said previously, we do not worry about downstream compatibility. Subsequent layers we are dynamically built to match the shape of the incoming tensors, i.e the outputs of preceding layers.
- For instance, suppose we write the following:

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(16))
```

- The second layer did not receive an input shape argument. Instead, it automatically inferred its input shape as being the output shape of the layer that came before.

Automatic Shape Recognition, Model

- Just like with LEGO bricks, you can only “clip” together layers that are *compatible*. The notion of *layer compatibility* here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape. Consider the following dense layer with 32 output units:

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation='relu’, input_shape=(784,))
model = models.Sequential()
model.add(layer)
model.add(layers.Dense(16))
```

- The second layer did not receive an input shape argument. Instead, it automatically inferred its input shape as being the output shape of the layer that came before.
- This layer returns a tensor where the first dimension has been transformed to 32. It can only be connected to a downstream layer that expects 32-dimensional vectors as its input.
- When using Keras, you don’t worry about size compatibility most of the time, because the layers are dynamically built to match the shape of the incoming data emitted by the previous layer.
- Object `model`, we introduced above, is the key container of network architecture. In `tf.keras`, the `model` is a directed, acyclic graph of layers.
- The most common `model` is a linear stack of layers, mapping a single input tensor to a single output tensor.

Sequential vs. Functional Models

- Below is a two-layer model defined using the `Sequential` class. Note that we are passing the expected shape of the input data to the first layer.

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

- The same model defined using the functional API reads:

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

- With the functional API, you're manipulating the data tensors that the model processes and applying layers to this tensor as if they were functions.

Loss Function, Optimizer, Metrics
The `compile()` & `fit()` steps

Optimizers, Losses, Metrics

- In general, you do not create your own losses, metrics, or optimizers from scratch, because Keras offers a wide range of built-in options that is likely to include what you need:
- Optimizers:
  - SGD() (with or without momentum)
  - RMSprop()
  - Adam()
  - Adagrad(), Etc.

- Losses:
  - CategoricalCrossentropy()
  - SparseCategoricalCrossentropy()
  - BinaryCrossentropy()
  - MeanSquaredError()
  - KLDivergence()
  - CosineSimilarity(), Etc.

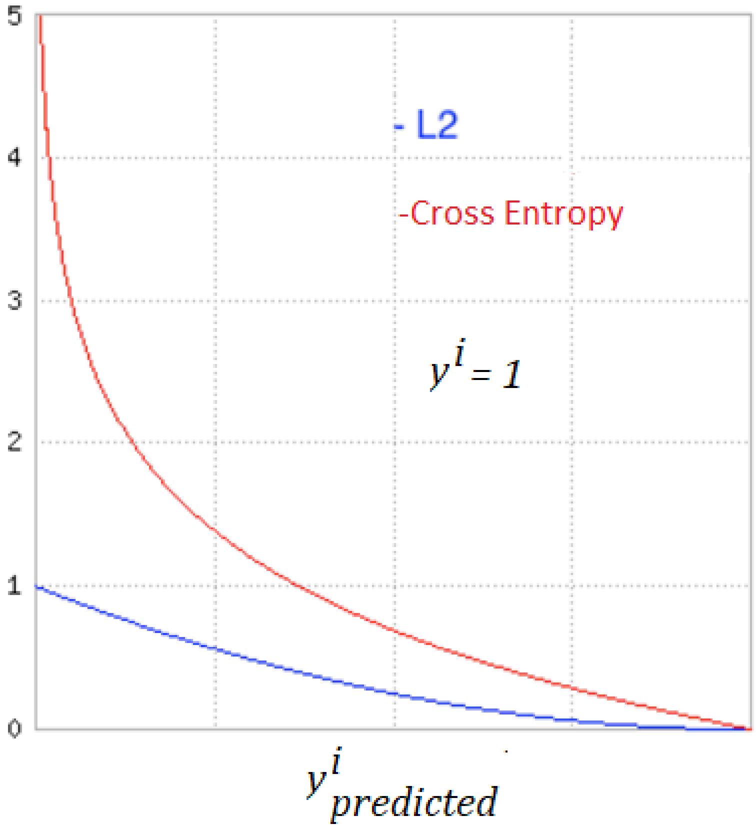
- Metrics:
  - CategoricalAccuracy()
  - SparseCategoricalAccuracy()
  - BinaryAccuracy()
  - AUC()
  - Precision()
  - Recall(), Etc.

Loss functions and Optimizers

- Once the network architecture is defined, you still must choose :
  - ***Loss** (cost or objective) **function** is* the quantity that is minimized during training. The loss function represents a measure of network's success of in predicting values.
  - ***Optimizer*** *is* a class which determines network weights based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).
  - ***Metrics*** are the measures of success of learning process.
- A neural network that has multiple outputs may have multiple loss functions (one per output). The gradient-descent process must be based on a *single* scalar loss value; so, for multi-loss networks, all losses are combined (via averaging) into a single scalar quantity.
- Choosing the right loss function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss;
- When it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss function.
- We use
  - *binary cross entropy* for a *two-class classification problem,*
  - *categorical cross entropy* for a *many-class classification problem,*
  - *mean-squared-error* for a *regression problem,*
  - *connectionist temporal classification* (CTC) for a *sequence-learning problem*, etc.
- Only when you work on truly new research problems, will you have to develop your own loss functions.

Cross Entropy Loss Function

- Cross entropy loss function is defined as:
- $L = \sum_{i=1}^{N}[y^{i}\log(y^{i}_{predicted}) + (1 - y^{i})\log(1 - y^{i}_{predicted}])$
- Below we visualize this function for a sample with label $y^i = 1$.



- Cross entropy produces a much greater value ("penalty"), when prediction is farther from what is expected than the standard square error.
- For a sample with label $y^i = 1$, *the* cross entropy, as the predicted value comes closer to 0, the penalty grows towards infinity.
- This makes it very expensive for the model to make that misprediction.
- Cross entropy is better suited as a loss function for classification models than simple square error function. The maximal value of a square error function, presented in blue is 1.

compile() and fit()

- Once we select our loss, optimizer, and metrics, we use the built-in compile() method configures the training process

```
        model.compile(loss='mean_squared_error', optimizer=sgd)
```

- The fit() method executes the actual learning. Its key arguments are:
  - The data (inputs and targets) to train on. It will typically be passed either in the form of NumPy arrays, of a TensorFlow Dataset object.
  - The number of epochs to train for: how many times the training loop should iterate over the data passed.
  - The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.

`compile()` and Optimizers

- An optimizer is one of the two arguments required for compiling a `tf.Keras` model:

```
import tensoflow as tf
from tensoflow import keras
from tensorflow.keras import optimizers
from tensorflow.keras import layers
model = keras.Sequential()
model.add(layers.Dense(64,kernel_initializer='uniform',
input_shape=(10,)))
model.add(layers.Activation('softmax'))
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9,
nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

- You can instantiate an optimizer before passing it to `model.compile()`, as above, or you can call it by its short name. Then the default parameters are used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

- The parameters `clipnorm` and `clipvalue` can be used with all optimizers to control gradient clipping:

```
# All parameter gradients will be clipped to a maximum norm of 1.
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
# All parameter gradients will be clipped to a max 0.5 and min of -0.5.
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```

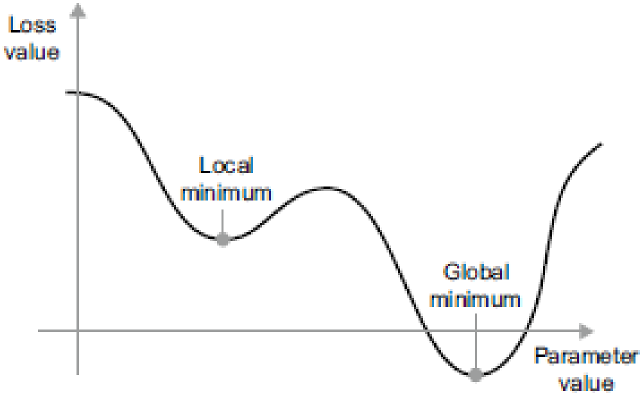
fit() method

- Trains the model for a fixed number of epochs (iterations on a dataset).

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose="auto",
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

Momentum

- When reading descriptions of many optimizers you will notice the concept f “momentum”. Momentum addresses two issues with Stochastic Gradient Descent (SGD): convergence speed and local minima. Consider figure below, which shows the curve of a loss as a function of a model parameter.
- There is a *local minimum*: around that point, moving left would result in the loss increasing, but so would moving right. If the parameter under consideration were being optimized via SGD with a small learning rate, the optimization process could get stuck at the local minimum instead of making its way to the global minimum.



- You can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve. If it has enough momentum, the ball won’t get stuck in a ravine and will end up at the global minimum.
- Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration).
- In practice, this means updating the parameter w based not only on the current gradient value but also on the previous parameter update

Implementation of Momentum

- A somewhat simplified implementation of the momentum idea would read :

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

Developing with tf.Keras

Developing with tf.Keras

- Typical `tf.Keras` program requires the following steps:
  1. Define your training data: input tensors and target tensors.
  2. Define a network of layers (or *model*) that maps your inputs to your targets.
  3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
  4. Iterate on your training data by calling the `fit()` method of your model.

- Two typical ways to define a model are using the
  – `Sequential` class (only for linear stacks of layers, which is the most common network architecture by far) or the
  – `Functional` *API* (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures).

Learning Process

- The following steps are the same for both Sequential model and Functional API.
- The learning process is configured in the `compile()` step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you monitor during training.
- The following is an example with a single loss function, which is by far the most common case:

```
from tensorflow.keras import optimizers
model.compile(optimizer=optimizers.RMSprop(lr=0.001), loss='mse',
metrics=['accuracy’])
```

- Finally, the learning process consists of passing Numpy array of input data (and the corresponding target data) to the model via the `fit()` method, similar to what you would do in Scikit-Learn and several other machine-learning libraries:

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

Inference (Predicting)

- Once you trained your model, you can use it to make predictions on new data. This is called "inference". To do this, a naive approach would simply be to call the model:

```
predictions = model(new_inputs)
```

- The `model()` will take a `NumPy array` or TensorFlow `tensor` and return a TensorFlow `tensor`
- This processes all inputs in `new_inputs` at once, which may not be feasible with at a lot of data.
- A better way to do inference is to use the `predict()` method. It will iterate over the data in small batches and return a `NumPy array` of predictions.

```
predictions = model.predict(new_inputs, batch_size=128)
```

- Takes a `NumPy array` or a `Dataset` and returns a `NumPy array`
- For instance, if we use `predict()` on some of our validation data with the linear model we trained earlier, we get scalar scores between 0 and 1 — below 0.5 indicates that the model considers the corresponding point to belong to class 0, and above 0.5 indicates that the model considers the corresponding point to belong to class 1.

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10]) [[0.3590725 ]
[0.82706255] [0.74428225] [0.682058 ] [0.7312616 ] [0.6059811
] [0.78046083] [0.025846 ] [0.16594526]
[0.72068727]]
```

Binary Classification

Classification and regression glossary

- *Sample or input* — One data point that goes into your model.
- *Prediction or output* — What comes out of your model.
- *Target* — The truth. What your model should ideally have predicted, according to an external source of data.
- *Prediction error or loss value* — The distance between your model's prediction and the target.
- *Classes* — A set of possible labels to choose from in a classification problem. For example, when classifying cat and dog pictures, dog" and cat" are the two classes.
- *Label* — A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class dog," then dog" is a label of picture #1234.
- *Ground-truth or annotations* — All targets for a dataset, typically collected by humans.
- *Binary classification* — Each input sample should be categorized into two exclusive categories.
- *Multiclass classification* — A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
- *Multilabel classification* — A classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the cat" label and the “dog” label. The number of labels per image is usually variable.
- *Scalar regression* — A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
- *Vector regression* — A task where the target is a set of continuous values: for example, a continuous vector. If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you're doing vector regression.

Classifying Movie Reviews

- In a practical demonstration of the use of tf.Keras we will examine "IMDB dataset", a set of 50,000 highly-polarized reviews from the Internet Movie Database.
- Reviews are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.
- Just like the MNIST dataset, the IMDB dataset comes packaged with tf.Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

```
import numpy
from tensorflow import keras
from tensorflow.keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels)=
imdb.load_data(num_words=10000)
Downloading data from https://s3.amazonaws.com/text-datasets/imdb.npz
17465344/17464789 [==============================] - 3s 0us/step
```

- The argument `num_words=10000` means that we will only keep the top `10,000` most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.
- The variables `train_data` and `test_data` are lists of reviews, each review being a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for "negative" and 1 stands for "positive":

Movie Reviews

- To see the content of `train_data` and `train_labels`, we could ask for values in the first sample, with index 0:

```
train_data[0]
[1, 14, 22, 16, 43,

train_labels[0]
1
```

- We restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

```
max([max(sequence) for sequence in train_data])
9999
```

- To decode the review, see the text in English, we have to pay attention to the fact that indices are offset by 3. 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".

Decoded Review

- Here is how you can quickly decode one of these reviews back to English words

```
# word_index is a dictionary mapping words to an integer index
word_index = imdb.get_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])
# We decode the review; indices are offset by 3
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?')
   for i in train_data[0]]
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/imdb_word_index.json
1646592/1641221 [==============================] - 1s 1us/step
```

```
decoded_review
```

```
"? this film was just brilliant casting location scenery story direction everyone's
really suited the part they played and you could just imagine being there robert ? is
an amazing actor and now the same being director ? father came from the same scottish
island as myself so i loved the fact there was a real connection with this film the
witty remarks throughout the film were great it was just brilliant so much that i
bought the film as soon
```

Preparing the Data

- We have to turn our lists of number into tensors.
- There are two ways we could do that:
  - We could pad our lists so that they all have the same length, and turn them into an integer tensor of shape (`samples, word_indices`).
  - We could one-hot-encode our lists to turn 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 are set to ones. Then we could use as first layer in our network a Dense layer, capable of handling floating point vector data.
- We will go with the latter solution. We will vectorize our data
- Our original data look like this:

```
train_data
```

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838,
112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447,
4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, . .
. . . . ., 19, 178, 32]),
list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119,
954, 189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116,
9, 35, 8163, 4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 100 . . . . . ]),
```

One-hot-encoding

- The following will replace lists with vectors of 0-s and 1-s

```
import numpy as np
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)
```

- Every review (document) is now an array of floating point 0-s and 1-s.

```
x_train[0]
array([0., 1., 1., ..., 0., 0., 0.])
```

- We also vectorize labels

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Bag of Words Model

- On the previous slide we see that we replaced every review (a collection of ordered words), each of which is 10,000 elements long vector of zeros and a single 1, with a single 10,000 elements long vector of zeros and 1-s.
- In effect we said, the order of words in a review is not all that important and it might not be important how many times a word appears in a document.
- We will just record whether a word appears in a review or not. The resulting vector has 1 for every word that appears at least once and 0 for words that do not appear in that review.
- This is what in computational linguistics we call “bag of words model”. There are many more elaborate and more precise models. Still, bag of words model is used quite often and with a considerable success.

Building the network

- Our input data is simple vectors, and our labels are scalars (1s and 0s).
- A type of network that performs well on such a problem is a simple stack of fully-connected (`Dense`) layers with `relu` activations: `Dense(16,activation='relu')`
- We choose a network with 3 layers. The first two layers have 16 neurons. The first layer accepts 10,000 units long vectors. The last layer is a single neuron with a simple sigmoid since we want to make a decision: a review is positive (1) or negative (0).

```
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- We need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss function. It is not the only viable choice: you could use, for instance, `mean_squared_error`. But cross-entropy is usually the best choice when you are dealing with models that output probabilities.

Optimizers

- `TensorFlow, i.e. tf.Keras` documentation is not very verbose about various optimizers `tf.Keras` supports.
- You could take a look at this page:

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/

- There are many articles on the Internet and Wikipedia that clarify major features of most popular optimizers. For example

http://ruder.io/optimizing-gradient-descent/index.html#rmsprop

Loss function and Optimizer

- Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`.

- But cross-entropy is usually the best choice when you are dealing with models that output probabilities.

```
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Testing Results, Validating the Approach

- In order to monitor the accuracy of the model during the training, we will use the data that the model has not seen before. For that purpose we create a "validation set" made of 10,000 samples from the original training data:

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

- We will train our model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At this same time we will monitor `loss` and `accuracy` on the 10,000 samples that we set apart. This is done by passing the validation data as the `validation_data` argument:

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Train on 15000 samples, validate on 10000 samples Epoch 1/20 15000/15000
[==============================] - 5s 308us/step - loss: 0.5834 - acc: 0.7413 - val_loss: 0.4437 -
val_acc: 0.8386 Epoch 2/20 15000/15000 [==============================] - 3s 212us/step -
loss: 0.3364 - acc: 0.8907 - val_loss: 0.3136 - val_acc: 0.8817  . . . .
.   .   .   .   .
Epoch 20/20 15000/15000 [==============================] - 4s 289us/step - loss: 0.0043 - acc:
1.0000 - val_loss: 0.5788 - val_acc: 0.8699
```

`history` Object

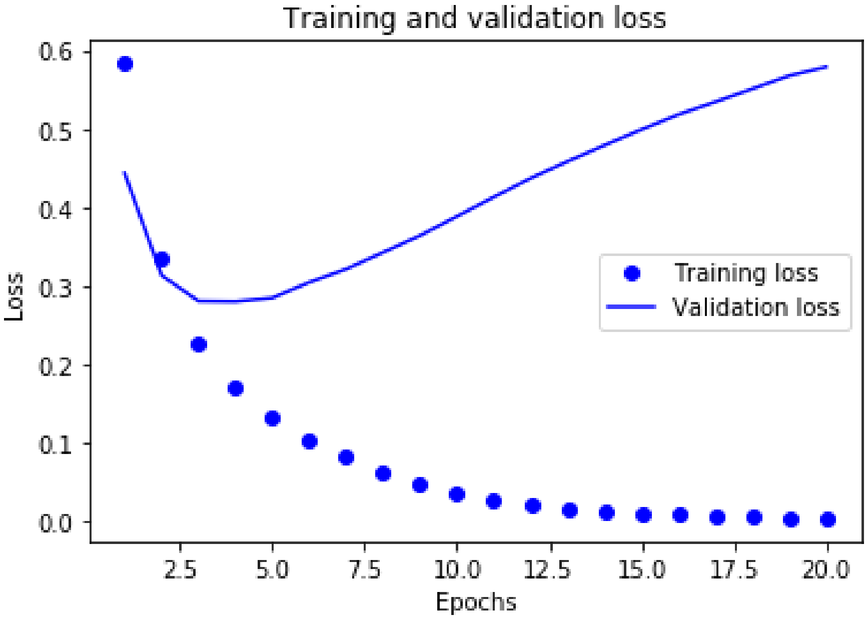
- The call to `model.fit()` returns a `history` object. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let us examine it:

```
history_dict = history.history
history_dict.keys()
dict_keys(['val_acc', 'acc', 'val_loss', 'loss’])
```

- It contains 4 entries: one per metric that was being monitored, during training and during validation.
- We could use Matplotlib to plot the training and validation loss side by side, next slide, as well as the training and validation accuracy, on the following slide.

Training and Validation Loss

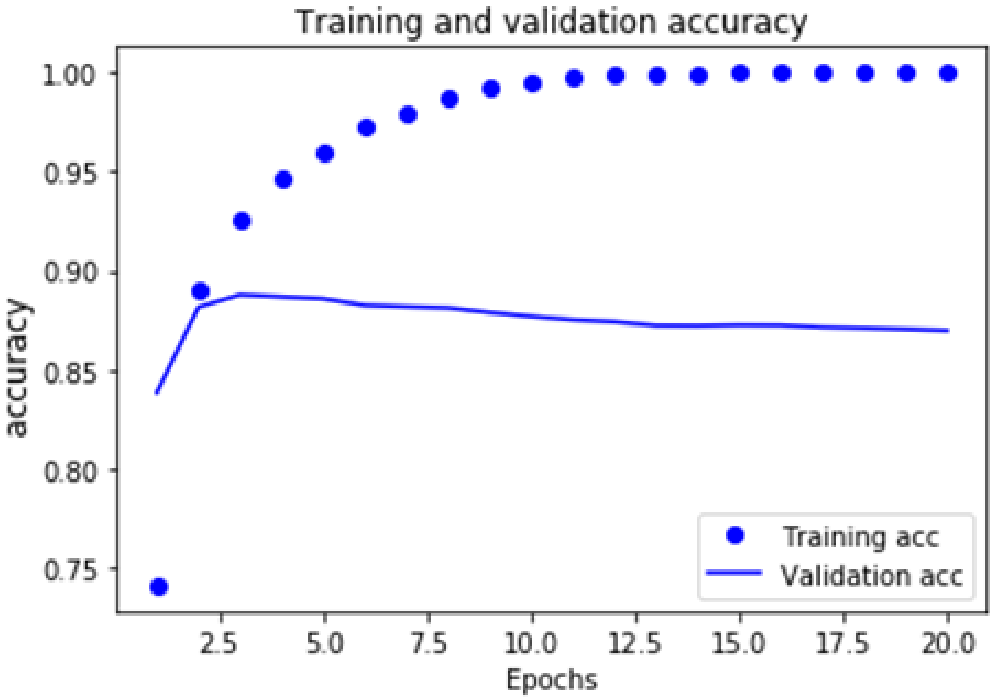
```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)
```



```
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Training and Validation Accuracy

```
plt.clf()   # clear figure
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

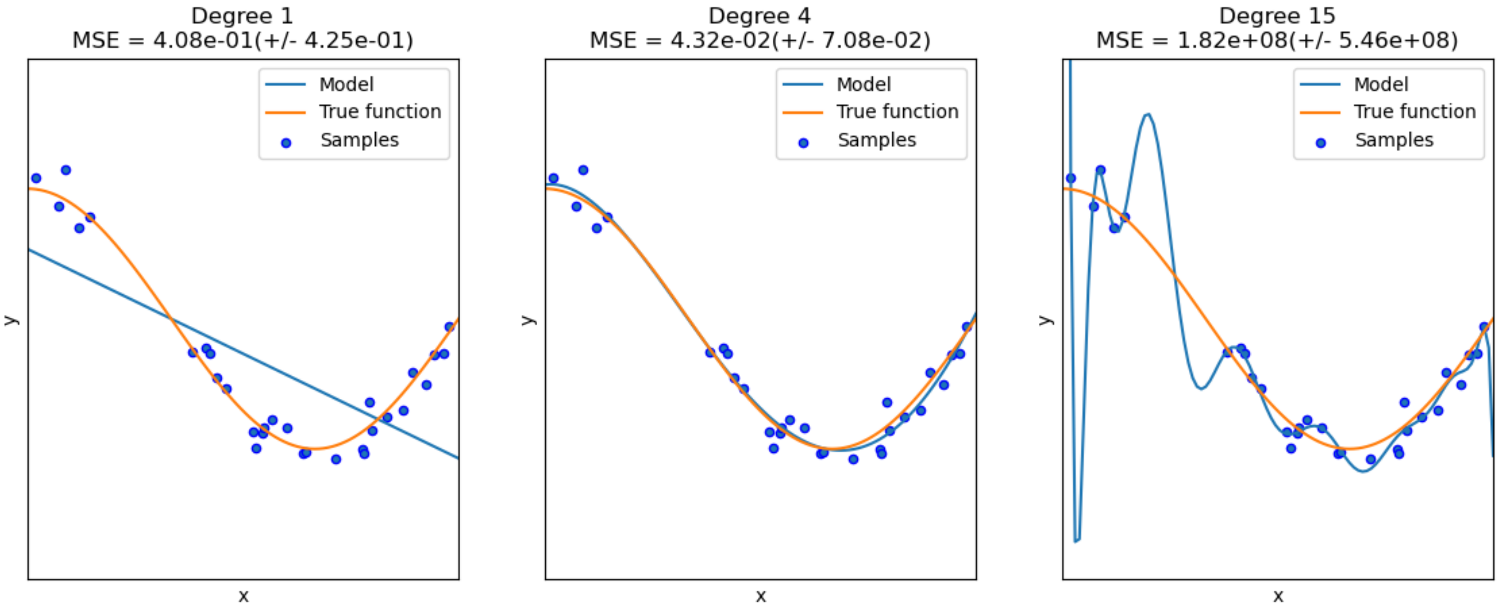


Overfitting

- As you can see, the training loss decreases with every epoch and the training accuracy increases with every epoch. That's what you would expect when running gradient descent optimization -- the quantity you are trying to minimize should get lower with every iteration.
- But that isn't the case for the validation loss and accuracy: they seem to peak at the third or fourth epoch.
- A model that performs well on the training data isn't necessarily a model that will do better on data it has never seen before.
- What you are seeing is "overfitting": after the second epoch, we are over-optimizing on the training data, and we ended up learning representations that are specific to the training data and do not generalize to data outside of the training set.
- In this case, to prevent overfitting, we could simply stop training after three epochs. In general, there is a range of techniques you can leverage to mitigate overfitting, most notably the regularization.

Overfitting Review

- The plot shows a function (cosine(x)) that we want to approximate. The models have polynomial features of different degrees.
- A linear function (polynomial with degree 1) is not sufficient to fit the training samples. This is called *underfitting*.
- A polynomial of degree 4 approximates the true function almost perfectly.
- A model with higher degrees (15) has a very small error with respect to training data but will have very high error with respect to any new data point. Such model "*overfits*" the training data, i.e., it learns the noise of the training data. Polynomial of 26-th degree could have zero training error.
- Neural networks often have very large number of trainable parameters (weights and biases) and could similarly overfit the training data, as we see on previous slides.



Train network on “optimal” number of epochs

- We will train the network for four epochs, then evaluate it on our test data:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

```
Epoch 1/4 25000/25000 [==============================] - 4s 165us/step - loss: 0.4749 - acc:
0.8217
Epoch 2/4 25000/25000 [==============================] - 3s 138us/step - loss: 0.2658 - acc:
0.9097
Epoch 3/4 25000/25000 [==============================] - 4s 143us/step - loss: 0.1982 - acc:
0.9299
Epoch 4/4 25000/25000 [==============================] - 4s 159us/step - loss: 0.1679 - acc:
0.9404 25000/25000 [==============================] - 4s 171us/step

Results
[0.3231498811721802, 0.87352]
```

- Our fairly naive approach achieves an accuracy of 87.35%. With state-of-the-art approaches, one should be able to get close to 95%.

Predicting results on new data

- After training a network, you will want to use it in a practical setting.
- You generate the likelihood of reviews being positive by using the predict method:

```
model.predict(x_test)
array([[0.14028221], [0.9997029 ], [0.29558158], ..., [0.07234909], [0.04342627], [0.4815999 ]], dtype=float32)
```

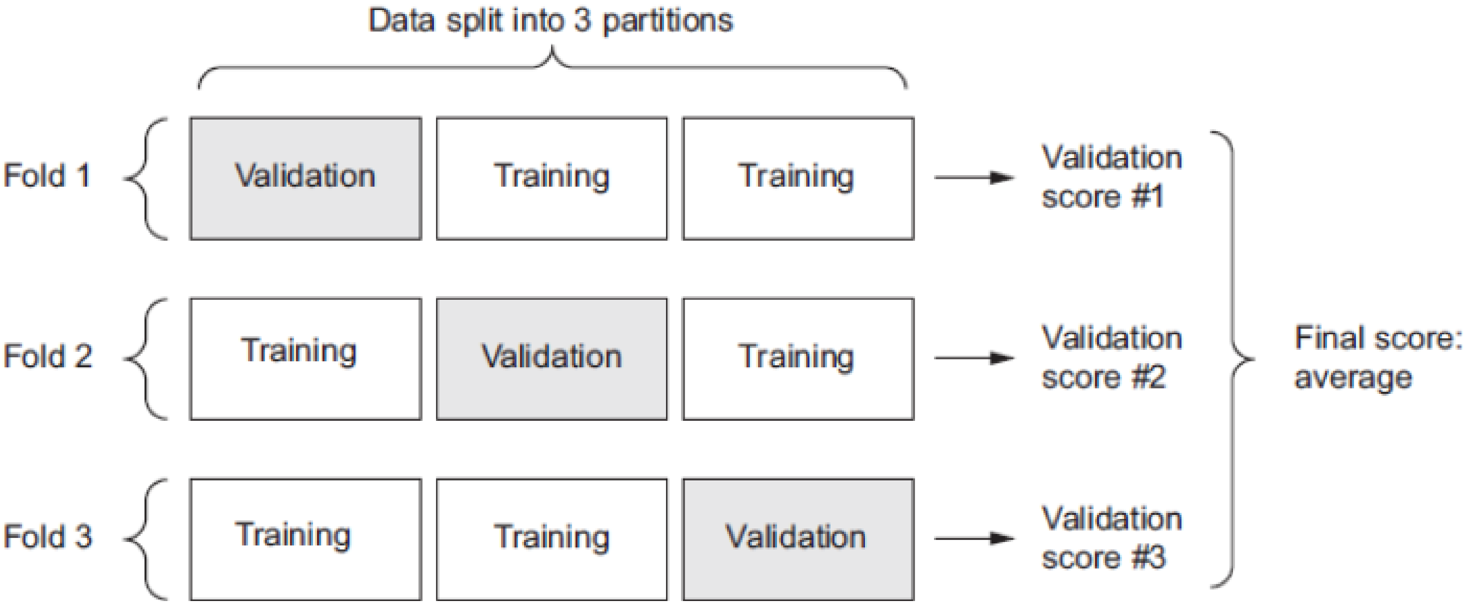
- Network is predicting that some samples are positive [0.9997029] and some negative, ..., [0.07234909], [0.04342627]. For some samples, the network gives inconclusive results [0.4815999]]
- As they get better on their training data, neural networks eventually start *overfitting* and end up obtaining increasingly worse results on data never-seen-before. Make sure to always monitor performance on data that is outside of the training set.

K-fold Validation

- To evaluate our network while we keep adjusting its parameters (such as the number of epochs used for training), we could simply split the data into a training set and a validation set, as we were doing in our previous examples.
- However, because we have so few data points, the validation set would end up being very small (e.g. about 100 examples). A consequence is that our validation scores may change a lot depending on *which* data points we choose to use for validation and which we choose for training, i.e. the validation scores may have a high *variance* with regard to the validation split. This would prevent us from reliably evaluating our model.
- The best practice in such situations is to use K-fold cross-validation. It consists of splitting the available data into K partitions (typically K=4 or 5), then 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 would then be the average of the K validation scores obtained.
- Please note that this technique is invaluable when analyzing small data sets. However, it is considered too expensive for massive data sets.

3-fold Validation

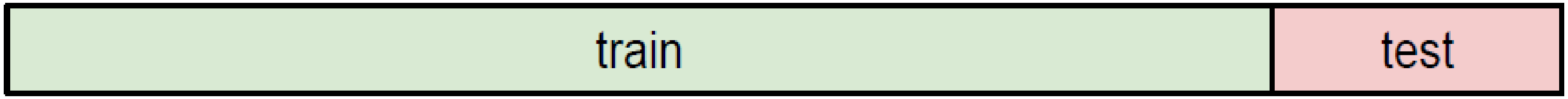
- The following is a symbolic representation of a 3-fold validation



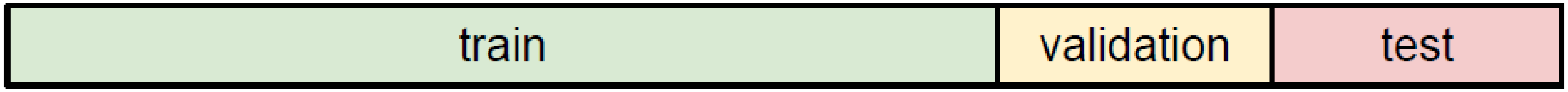
- We train our model 1 on 2 training partitions and validate its accuracy on the first validation partition. We store the “score #1”. The score could be loss, accuracy or both.
- We recreate a new model and repeat the training on the shifted set of data. The Validation set is the second partition this time. Again we keep the score(s).
- Finally, we reinitialize the third model and, train it the first two partitions and validate it on the third partition. Keep the third score(s).
- Average score(s) over three runs. The result is (are) the final score(s).
- Notice that this is a bit different from k-fold validation we introduced previously.
- Some people use this approach and other use the one on the next slide.

Determining Hyper Parameters

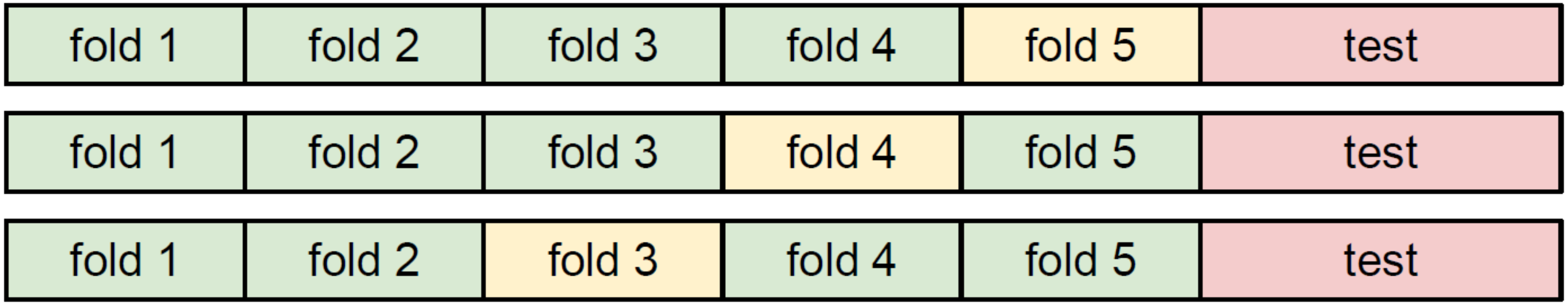
- When training a model we said that we should separate the data into 80% training set and 20% test set. We determine the optimal weights and biases on the training set and then evaluate the accuracy of the model on the test set.



- If we would like to determine the optimal values of hyper-parameters we need to break the data set in 3 parts. In this arrangement, the second portion of the data is used for validation or optimization of the hyper-parameters.



- To make sure that we are not picking hyper-parameters optimized for a particular portion of our data, the experiments are performed several times with the validation and test portion cycled through the data set. Hyper-parameters from various validation regions are averaged at the end. Due to lack of data, this practice is sometimes considered too expensive.



Scalar Regression

Predicting the past Boston Housing Prices

- Another frequent ML task is to predict a numeric value of a variable given a one or multi-dimensional input. Such task is called *"Regression".*
- We will attempt to predict the median price of homes in one Boston suburb in the mid-1970s, given a few data points about the suburb at the time, such as the crime rate, the local property tax rate, etc.
- The dataset we will use is different from our two previous examples: it has very few data points, only 506 in total, split between 404 training samples and 102 test samples, and each "feature" in the input data (e.g. the crime rate is a feature) has a different scale. For instance some values are proportions, which take a values between 0 and 1, others take values between 1 and 12, others between 0 and 100...

```
from tensorflow.keras.datasets import boston_housing
(train_data,train_targets),(test_data,test_targets) =
 boston_housing.load_data()
Downloading data from https://s3.amazonaws.com/keras-datasets/boston_housing.npz 57344/57026
[==============================] - 0s 1us/step
train_data.shape
(404, 13)
test_data.shape
(102, 13)
```

- As you can see, we have 404 training samples and 102 test samples.

Features

Boston housing prices data set has 13 features

1. Per capita crime rate.
2. Proportion of residential land zoned for lots over 25,000 square feet.
3. Proportion of non-retail business acres per town.
4. Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
5. Nitric oxides concentration (parts per 10 million).
6. Average number of rooms per dwelling.
7. Proportion of owner-occupied units built prior to 1940.
8. Weighted distances to five Boston employment centers.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. 1000 *(Bk - 0.63)* * 2 where Bk is the proportion of Black people by town.
13. % lower status of the population.

- The targets are the median values of owner-occupied homes, in thousands of dollars

Values of Targets

```
train_targets
•   array([ 15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3, 15.6, 15.6, 14.4,
    12.1, 17.9, 23.1, 19.9, 15.7, 8.8, 50. , 22.5, 24.1, 27.5, 10.9, 30.8,
    32.9, 24. , 18.5, 13.3, 22.9, 34.7, 16.6, 17.5, 22.3, 16.1, 14.9, 23.1,
    34.9, 25. , 13.9, 13.1, 20.4, 20. , 15.2, 24.7, 22.2, 16.7, 12.7, 15.6,
    18.4, 21. , 30.1, 15.1, 18.7, 9.6, 31.5, 24.8, 19.1, 22. , 14.5, 11. ,
    32. , 29.4, 20.3, 24.4, 14.6, 19.5, 14.1, 14.3, 15.6, 10.5, 6.3, 19.3,
    19.3, 13.4, 36.4, 17.8, 13.5, 16.5, 8.3, 14.3, 16. , 13.4, 28.6, 43.5,
    20.2, 22. , 23. , 20.7, 12.5, 48.5, 14.6, 13.4, 23.7, 50. , 21.7, 39.8,
    38.7, 22.2, 34.9, 22.5, 31.1, 28.7, 46. , 41.7, 21. , 26.6, 15. , 24.4,
    13.3, 21.2, 11.7, 21.7, 19.4, 50. , 22.8, 19.7, 24.7, 36.2, 14.2, 18.9,
    18.3, 20.6, 24.6, 18.2, 8.7, 44. , 10.4, 13.2, 21.2, 37. , 30.7, 22.9,
    20. , 19.3, 31.7, 32. , 23.1, 18.8, 10.9, 50. , 19.6, 5. , 14.4, 19.8,
    13.8, 19.6, 23.9, 24.5, 25. , 19.9, 17.2, 24.6, 13.5, 26.6, 21.4, 11.9,
    22.6, 19.6, 8.5, 23.7, 23.1, 22.4, 20.5, 23.6,
…..
```

- Prices in 1970 ranged from $10,000 to $50,000. To get today's prices just multiply the above numbers by 100.

Preparing the data

- It might be problematic to feed into a neural network values that all take wildly different ranges.
- The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult.
- A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), we will subtract the mean of the feature and divide by the standard deviation, so that the feature will be centered around 0 and will have a unit standard deviation.
- This is easily done in Numpy: `mean = train_data.mean(axis=0)`

```
train_data -= mean
std = train_data.std(axis=0)
train_data /= std

test_data -= mean
test_data /= std
```

Constructing the network

- Because so few samples are available, we will use a very small network with two hidden layers, each with 64 units. In general, the less training data you have, the worse overfitting. Using a small network is one way to mitigate overfitting.

```
from tensorflow.keras import models
from tensorflow.keras import layers
def build_model():
    # We need to instantiate the same model multiple times,
    # so we will use a function to construct it.
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

- It is a common practice to build our model within a function.
- Our network ends with a single unit, and no activation. This is a “linear” layer.
- This is a typical setup for scalar regression, where we are trying to predict a single continuous value which is not bound by 0 and 1. Price could be an arbitrary number

Characteristics of Regression Networks

- Notice that our last layer has no activation function.
- Applying an activation function would constrain the range of the output;
- Had we applied a sigmoid activation function to our 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.
- Note that we are compiling the network with the `mse` loss function -- Mean Squared Error, the square of the difference between the predictions and the targets, a widely used loss function for regression problems.
- We are also monitoring a new metric during training: `mae`. This stands for Mean Absolute Error. It is simply the absolute value of the difference between the predictions and the targets. For instance, a MAE of 0.5 on this problem would mean that our predictions are off by $500 on average.

Validating our approach with K-fold validation

- We will split the available data into K=4 partitions.
- Then we will instantiate K identical models, and train each one on K-1=3 partitions while evaluating on the remaining partition.
- The validation score for the model used would then be the average of K=4 calculated validation scores.

Validating our approach with K-fold validation

```
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)
    # Prepare the validation data: data from partition # k
    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]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # Build the tf.Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    # Evaluate the model on the validation data
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Results

- While processing the network outputs:

```
processing fold # 0
processing fold # 1
processing fold # 2
processing fold # 3
```

- To display all validation scores we type:

```
all_scores
[2.0035791113825128, 2.212607409694407, 2.8790937650321733, 2.3663983687315837]
```

- The mean score, over all 4 runs, is:

```
np.mean(all_scores)
2.365419663710169
```

- The different runs do indeed show rather different validation scores, from 2.0 to 2.9.
- Their average (2.4) is a much more reliable metric than any single of these scores -- that's the entire point of K-fold cross-validation.
- In this case, we are off by $2,400 on average, which is still significant considering that the prices range from $10,000 to $50,000.

Examining Validation Scores per Epoch

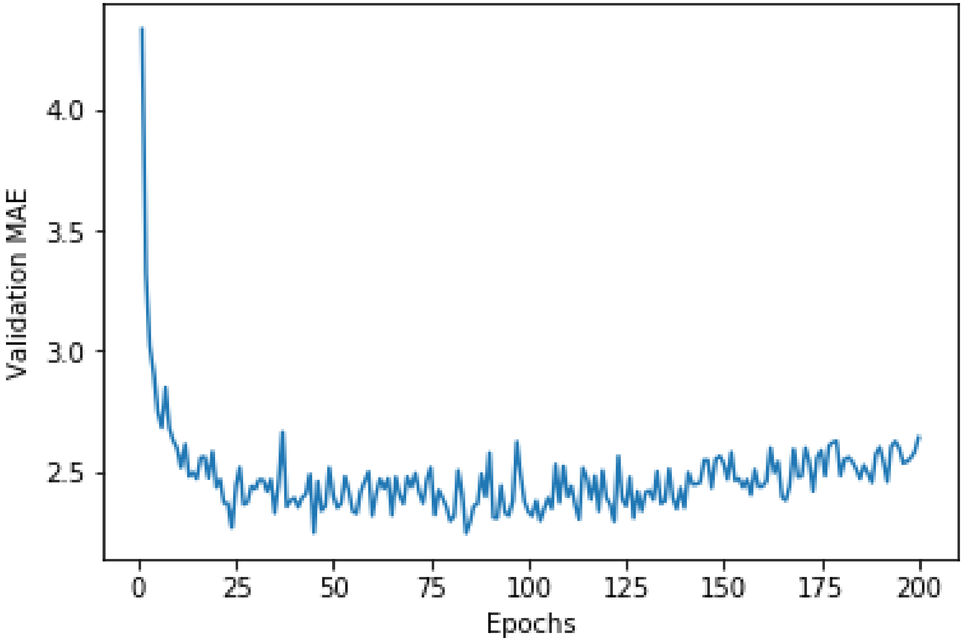
- To keep a record of how well the model did at each epoch, we will modify our training loop to save the per-epoch validation score log. We will run 200 epochs.

```
from keras import backend as K
# Some memory clean-up
K.clear_session()
num_epochs = 200
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    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]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # Build the tf.Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    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_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

Scores

- We can then compute the average of the per-epoch MAE scores for all folds:

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



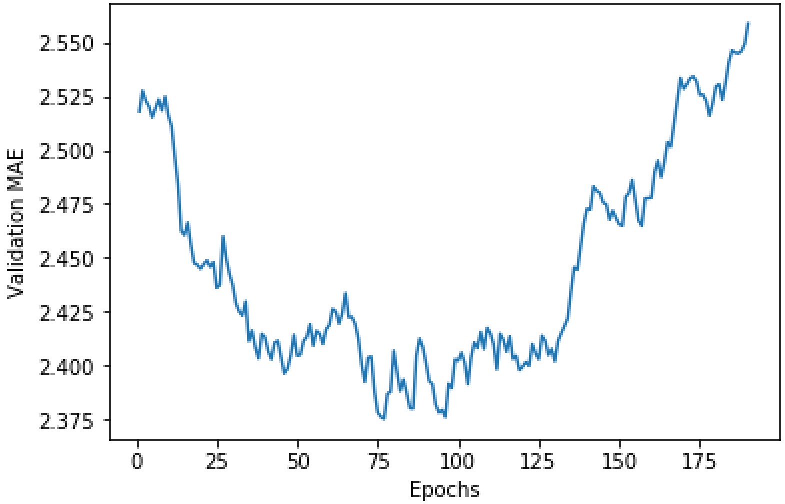
Improve the graph

- It may be a bit hard to see the plot due to scaling issues and relatively high variance. We will do the following
- Omit the first 10 data points, which are on a different scale from the rest of the curve.
- Replace each point with an exponential moving average of the previous points, to obtain a smooth curve.

```
def smooth_curve(points, factor=0.9):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

Optimal Number of Epochs

- Previous code produced the a cleaner graph:



- According to this plot, it seems that validation MAE stops improving significantly after 80 epochs. Past that point, we start overfitting.
- Once we are done tuning other parameters of our model (besides 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, then look at its performance on the test data.

Run the model with optimal # of epochs

- If 80 is the proper number of epochs we could go back to the original code and re-run it:

```
# Get a fresh, compiled model.
model = build_model()
# Train it on the entirety of the data.
model.fit(train_data, train_targets,
          epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)

test_mae_score
2.7169744547675636
```

- Our average error is still high ($2700) but at least our process did not waste time on too many epochs.

Regression, Summary

- Regression is done using different loss functions from classification; Mean Squared Error (MSE) is a commonly used loss function for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally the concept of "accuracy" does not apply for regression. A common regression metric is Mean Absolute Error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-Fold validation is a great way to reliably evaluate a model.
- When little training data is available, it is preferable to use a small network with very few hidden layers (typically only one or two), in order to avoid severe overfitting.

Fighting Overfitting

Overfitting and Underfitting

- We noticed 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* the training data.
- Overfitting happens in every single machine learning problem. Learning how to deal with overfitting is essential.
- The fundamental issue in machine learning is the tension between *optimization* and *generalization*.
- *"Optimization"* refers to the process of adjusting a model to get the best performance possible on the training data (the "learning" in "machine learning").
- *"Generalization*" refers to how well the trained model would perform on data it has never seen before. The goal of the game is to get good generalization, of course, but you do not control generalization; you can only adjust the model based on its training data.
- At the beginning of training, optimization and generalization are correlated: the lower your loss on training data, the lower your loss on test data. While this is happening, your model is said to be *under-fit*: there is still progress to be made; the network hasn't yet modeled all relevant patterns in the training data. After a certain number of iterations on the training data, generalization stops improving, validation metrics stall and then start degrading: the model is then starting to over-fit, i.e. is it starting to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data.

Regularization

- To prevent a model from learning misleading or irrelevant patterns found in the training data, *the best solution is of course to get* ***more high quality training data***. A model trained on more data will naturally generalize better.
- When that is not 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 usually referred to as the *regularization*.
- We will review some of the most common regularization techniques and apply them in practice to improve our movie classification model.

IMDB Test set

- For test with regularization we will use the same IMDB set we have used already:

```
from keras.datasets import imdb
import numpy as np

(train_data, train_labels), (test_data, test_labels) =
imdb.load_data(num_words=10000)

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')
```

Fighting Overfitting, *Reduce Network Size*

- The simplest way to prevent overfitting is to reduce the size of the model, i.e. the number of learnable parameters in the model (so called model capacity, 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.
- Such a model would be useless for classifying new samples.
- 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, we should use models that have enough parameters that they will not underfit. The model should not be starved for memorization resources. There should be a compromise between "too much capacity" and "not enough capacity".
- There is no magical formula to determine what the right number of layers is, or what the right size for each layer is. You have to evaluate an array of different architectures (on your validation set, not on your test set, of course) 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.

Vary the Size of Network

- Our original network had 16 neurons in the first 2 layers. Let is reduce that number to 4 per layer. This is the original network:

```
from keras import models
from keras import layers
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu',
input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))

original_model.compile(optimizer='adam',
                       loss='binary_crossentropy',
                       metrics=['acc'])
```

- This is a resized, smaller, model

```
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(4, activation='relu',
input_shape=(10000,)))
smaller_model.add(layers.Dense(4, activation='relu'))
smaller_model.add(layers.Dense(1, activation='sigmoid'))

smaller_model.compile(optimizer='rmsprop',
                      loss='binary_crossentropy',
                      metrics=['acc'])
```

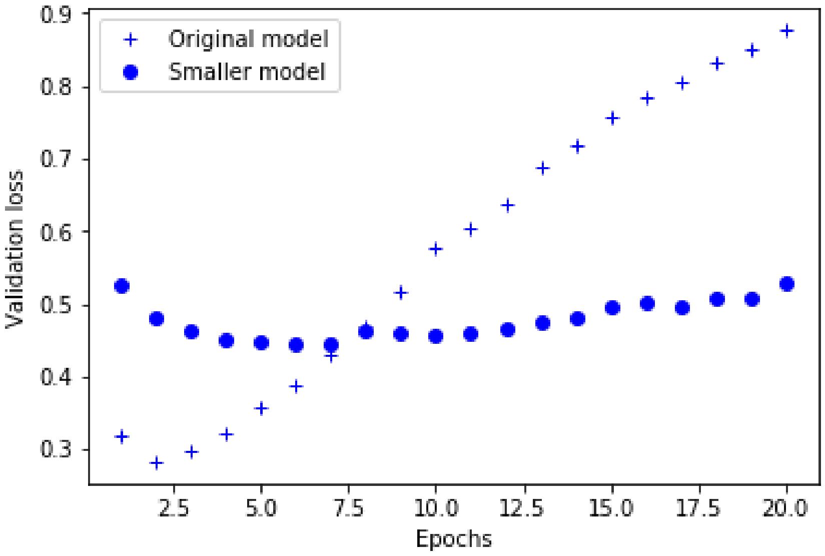
Comparison

- We run both models for 20 epochs with Adam optimizer.
- Then we extract the history objects from both models.

```
epochs = range(1, 21)
original_val_loss = original_hist.history['val_loss']
smaller_model_val_loss = smaller_model_hist.history['val_loss']
```

- And plot both losses as a function of the epoch number

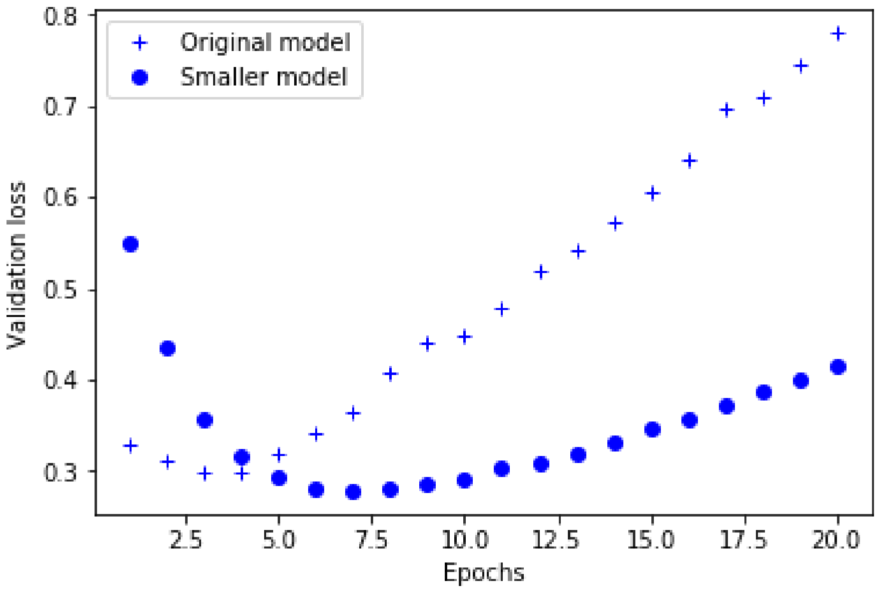
```
import matplotlib.pyplot as plt
# b+ is for "blue cross"
plt.plot(epochs, original_val_loss, 'b+', label='Original model')
# "bo" is for "blue dot"
plt.plot(epochs, smaller_model_val_loss, 'bo', label='Smaller model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```



We see that the smaller model does not overfit as quickly.

Comparison with `optimizer='rmsprop'`

- If in both original and smaller model we replace the optimizer with `rmsprop`, we will get somewhat better results.



- As you can see, the smaller network starts overfitting later than the reference one (after 6 epochs rather than 4) and its performance degrades much more slowly once it starts overfitting.
- You could continue these experiments and change the size of the first two layers to different values and then select an “optimal” size of the first 2 layers.

Weight Regularization

- A practical objective is to have as simple model as possible. This means a model with as few neurons and weights.
- 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.

Regularized linear regression

- To reduce the importance of tail weights, standard loss function is supplemented with a quadratic term in weights (W_{ij} -s) we are trying to determine.

$$L_2 = \sum_{i=0}^{n} (y_i \; - h(x_i))^2 + \lambda \sum_{i=0, j=0}^{n} (W_{ij})^2 \;]$$

- Parameter $\lambda, regularization\ rate,$ on the right hand side is a new hyper parameter and we are setting it ourselves.
- Indexes i, j in expression for weights W_{ij} are meant to indicate that there are many weights we are looking for.
- λ is not a parameter determined by the gradient descent or similar processes.
- λ is determined by an iterative estimation or by trial and error.

Weight Regularizer

- In tf.Keras, weight regularization is added by passing *weight regularizer instances* to layers as keyword arguments. Let's add L2 weight regularization to our movie review classification network:

```
from keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))

l2_model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy',
                 metrics=['acc'])
```

- **`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.
- This penalty is only added at training time, and the loss for this network will be much higher at training than at test time.

Impact of Regularization

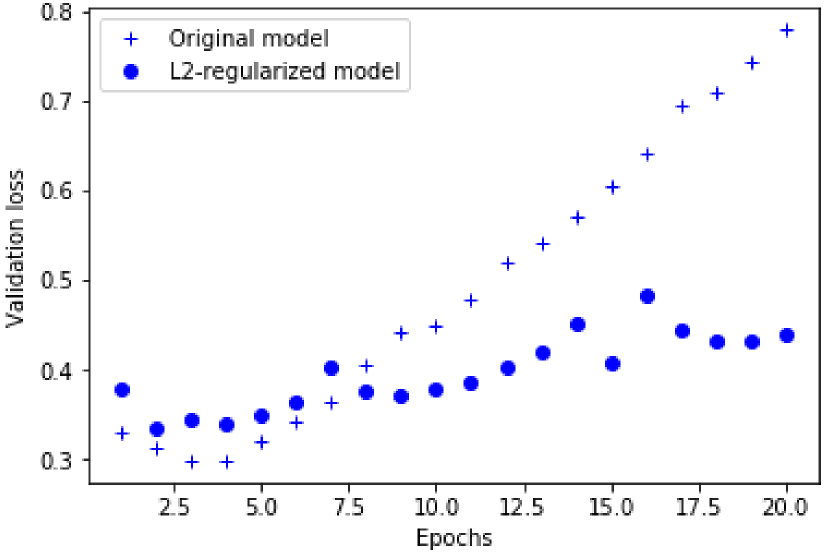
- To examine the impact of the regularization term, we fit (run) the model again

```
l2_model_hist = l2_model.fit(x_train, y_train,
                             epochs=20,
                             batch_size=512,
                             validation_data=(x_test, y_test))
```

- Subsequently, we extract model history and compare losses with the losses of the original unregularized model

```
l2_model_val_loss = l2_model_hist.history['val_loss']
plt.plot(epochs, original_val_loss, 'b+', label='Original model')
plt.plot(epochs, l2_model_val_loss, 'bo', label='L2-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```

Regularized model is more resistant to overfitting, though both models have the same number of parameters



Alternative Regularizers

- As we have seen, the model with L2 regularization (dots) has become much more resistant to overfitting than the reference model (crosses), even though both models have the same number of parameters.
- As alternatives to L2 regularization, one could use one of the following tf.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)
```

Dropout Layers

- Dropout is another effective and most commonly used regularization techniques for neural networks, developed by Hinton and colleagues at the University of Toronto.
- 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.

Dropping-out, Zero-ing

- Consider a Numpy matrix containing the output of a layer, layer_output, of shape (batch_size, features).
- At training time, we would be zero-ing out at random a fraction of the values in the matrix:

```
# At training time: we drop out 50% of the units in the output
layer_output *= np.randint(0, high=2, size=layer_output.shape)
```

- At test time, we would be scaling the output down by the dropout rate. Here we scale by 0.5 (because we were previous dropping half the units):

```
# At test time:
layer_output *= 0.5
```

- This process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it is implemented in practice:

```
# At training time:
layer_output *= np.randint(0, high=2, size=layer_output.shape)
# Note that we are scaling *up* rather scaling *down* in this case
layer_output /= 0.5
```

- The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that are not significant (what Hinton refers to as "conspiracies"), which the network would start memorizing if no noise was present.

Dropout layer

- In tf.Keras you can introduce dropout in a network via the Dropout layer, which gets applied to the output of layer right before it, e.g.:

```
model.add(layers.Dropout(0.5))
```

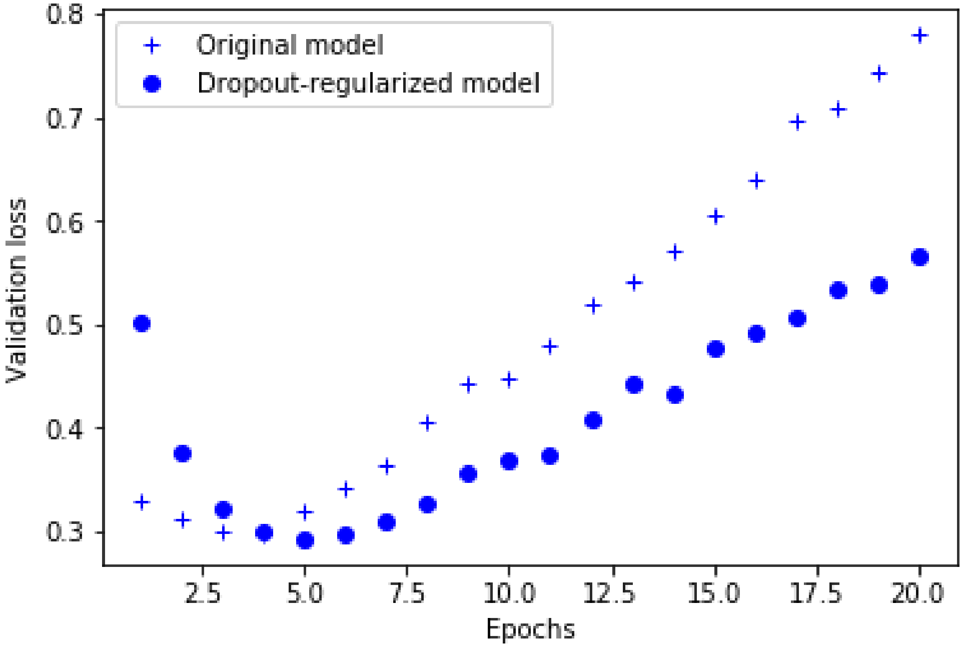
- We will add two Dropout layers in our IMDB network to see how well they do at reducing overfitting:

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['acc'])
dpt_model_hist = dpt_model.fit(x_train, y_train,
                               epochs=20,
                               batch_size=512,
                               validation_data=(x_test, y_test))
```

Compare the original model with drop-out model

- Once we go through all 20 epochs with new model containing the dropout layers, we extract the history of the new run and compare losses and/or accuracies with the losses or accuracies of the original layer.

```
dpt_model_val_loss = dpt_model_hist.history['val_loss']

plt.plot(epochs, original_val_loss, 'b+', label='Original model')
plt.plot(epochs, dpt_model_val_loss, 'bo', label='Dropout-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()

plt.show()
```



Again, we see a clear improvement over the reference network.

Appendix: Keras Documentation

Classification and regression glossary

- *Sample or input* — One data point that goes into your model.
- *Prediction or output* — What comes out of your model.
- *Target* — The truth. What your model should ideally have predicted, according to an external source of data.
- *Prediction error or loss value* — The distance between your model's prediction and the target.
- *Classes* — A set of possible labels to choose from in a classification problem. For example, when classifying cat and dog pictures, dog" and cat" are the two classes.
- *Label* — A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class dog," then dog" is a label of picture #1234.
- *Ground-truth or annotations* — All targets for a dataset, typically collected by humans.
- *Binary classification* — Each input sample should be categorized into two exclusive categories.
- *Multiclass classification* — A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
- *Multilabel classification* — A classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the cat" label and the "dog" label. The number of labels per image is usually variable.
- *Scalar regression* — A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
- *Vector regression* — A task where the target is a set of continuous values: for example, a continuous vector. If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you're doing vector regression.

Module `tf.keras.losses`

- `class BinaryCrossentropy`: Computes the cross-entropy loss between true labels and predicted labels.
- `class BinaryFocalCrossentropy`: Computes focal cross-entropy loss between true labels and predictions.
- `class CategoricalCrossentropy`: Computes the crossentropy loss between the labels and predictions.
- `class CategoricalFocalCrossentropy`: Computes the alpha balanced focal crossentropy loss.
- `class CategoricalHinge`: Computes the categorical hinge loss between y_true & y_pred.
- `class CosineSimilarity:` Computes the cosine similarity between labels and predictions.
- `class Hinge`: Computes the hinge loss between y_true & y_pred.
- `class Huber`: Computes the Huber loss between y_true & y_pred.
- `class KLDivergence:` Computes Kullback-Leibler divergence loss between y_true & y_pred.
- `class LogCosh`: Computes the logarithm of the hyperbolic cosine of the prediction error.
- `class Loss:` Loss base class.
- `class MeanAbsoluteError`: Computes the mean of absolute difference between labels and predictions.
- `class MeanAbsolutePercentageError`: Computes the mean absolute percentage error between y_true & y_pred.
- `class MeanSquaredError:` Computes the mean of squares of errors between labels and predictions.
- `class MeanSquaredLogarithmicError`: Computes the mean squared logarithmic error between y_true & y_pred.
- `class Poisson:` Computes the Poisson loss between y_true & y_pred.
- `class Reduction:` Types of loss reduction.
- `class SparseCategoricalCrossentropy:` The crossentropy loss between the labels and predictions.
- `class SquaredHinge`: Computes the squared hinge loss between y_true & y_pred.

Module `tf.keras.optimizers`

- `class Adadelta`: Optimizer that implements the Adadelta algorithm.
- `class Adafactor`: Optimizer that implements the Adafactor algorithm.
- `class Adagrad`: Optimizer that implements the Adagrad algorithm.
- `class Adam:` Optimizer that implements the Adam algorithm.
- `class AdamW:` Optimizer that implements the AdamW algorithm.
- `class Adamax`: Optimizer that implements the Adamax algorithm.
- `class Ftrl:` Optimizer that implements the FTRL algorithm.
- `class Lion`: Optimizer that implements the Lion algorithm.
- `class Nadam`: Optimizer that implements the Nadam algorithm.
- `class Optimizer`: Abstract optimizer base class.
- `class RMSprop:` Optimizer that implements the RMSprop algorithm.
- `class SGD:` Gradient descent (with momentum) optimizer.

Keras supported Optimizers

- *SGD (*Stochastic gradient descent) optimizer includes support for momentum, learning rate decay, and Nesterov momentum.
- *RMSProp* (for Root Mean Square Propagation) is an optimizer in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- *Adam* (short for Adaptive Moment Estimation) is an update to the *RMSProp* optimizer. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used.
- *Adagrad* (for adaptive gradient algorithm) is a modified stochastic gradient descent with per-parameter learning rate. Informally, this increases the learning rate for more sparse parameters and decreases the learning rate for less sparse ones. This strategy often improves convergence performance over standard stochastic gradient descent in settings where data is sparse and sparse parameters are more informative.
- *Adadelta* is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelta you don't have to set an initial learning rate. In this version, initial learning rate and decay factor can be set, as in most other tf.Keras optimizers. Recommended to leave the parameters at default values.
- *Adamx*: It is a variant of Adam based on the infinity norm.
- *Nadam:* Nesterov Adam optimizer extends Adam with Nestorov accelerated gradient algorithm.
- *FTRL:* "Follow The Regularized Leader" is an optimization algorithm developed at Google for click-through rate prediction in the early 2010s. It is most suitable for shallow models with large and sparse feature spaces.
- (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers)

SGD Optimizer

- SGD (Stochastic gradient descent) optimizer includes support for momentum, learning rate decay, and Nesterov momentum. SGD optimizer is invoke with

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

- Arguments

`lr: float >= 0`. Learning rate.

`momentum: float >= 0`. Parameter that accelerates SGD in the relevant direction and dampens oscillations.

`decay: float >= 0`. Learning rate decay over each update.

`nesterov: boolean`. Whether to apply Nesterov momentum.

RMSProp Optimizer

- RMSProp optimizer divides the gradient by a running average of its recent magnitude. RMSProp is invoked by the following line:

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

- It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).
- This optimizer is usually a good choice for recurrent neural networks.

- Arguments

`lr: float >= 0.` Learning rate.

`rho: float >= 0.`

`epsilon: float >= 0.` Fuzz factor. If None, defaults to K.epsilon().

`decay: float >= 0.` Learning rate decay over each update.

ADAM Optimizer

- Adam (Adaptive Moment Estimation) is an update to the *RMSProp* optimizer. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used.

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
```

- Arguments

`lr: float >= 0.`	Learning rate.
`beta_1: float, 0 < beta < 1.`	Generally close to 1.
`beta_2: float, 0 < beta < 1.`	Generally close to 1.
`epsilon: float >= 0.`	Fuzz factor. If None, defaults to K.epsilon().
`decay: float >= 0.`	Learning rate decay over each update.
`amsgrad: boolean.`	Whether to apply the AMSGrad variant of this algorithm.

Module `tf.keras.metrics` Classes

class AUC: Computes the approximate AUC (Area under the curve) via a Riemann sum.
class Accuracy: Calculates how often predictions equals labels.
class BinaryAccuracy: Calculates how often predictions matches binary labels.
class BinaryCrossentropy: Computes the crossentropy metric between the labels and predictions.
class CategoricalAccuracy: Calculates how often predictions matches one-hot labels.
class CategoricalCrossentropy: Computes the crossentropy metric between the labels and predictions.
class CategoricalHinge: Computes the categorical hinge metric between y_true and y_pred.
class CosineSimilarity: Computes the cosine similarity between the labels and predictions.
class FalseNegatives: Calculates the number of false negatives.
class FalsePositives: Calculates the number of false positives.
class Hinge: Computes the hinge metric between y_true and y_pred.
class KLDivergence: Computes Kullback-Leibler divergence metric between y_true and y_pred.
class LogCoshError: Computes the logarithm of the hyperbolic cosine of the prediction error.
class Mean: Computes the (weighted) mean of the given values.
class MeanAbsoluteError: Computes the mean absolute error between the labels and predictions.
class MeanAbsolutePercentageError: Computes the mean absolute percentage error between y_true and y_pred.
class MeanIoU: Computes the mean Intersection-Over-Union metric.
class MeanRelativeError: Computes the mean relative error by normalizing with the given values.
class MeanSquaredError: Computes the mean squared error between y_true and y_pred.

Module `tf.keras.metrics` Classes

class MeanSquaredLogarithmicError: Computes the mean squared logarithmic error between y_true and y_pred.

class MeanTensor: Computes the element-wise (weighted) mean of the given tensors.

class Metric: Encapsulates metric logic and state.

class Poisson: Computes the Poisson metric between y_true and y_pred.

class Precision: Computes the precision of the predictions with respect to the labels.

class PrecisionAtRecall: Computes best precision where recall is >= specified value.

class Recall: Computes the recall of the predictions with respect to the labels.

class RecallAtPrecision: Computes best recall where precision is >= specified value.

class RootMeanSquaredError: Computes root mean squared error metric between y_true and y_pred.

class SensitivityAtSpecificity: Computes best sensitivity where specificity is >= specified value.

class SparseCategoricalAccuracy: Calculates how often predictions matches integer labels.

class SparseCategoricalCrossentropy: Computes the crossentropy metric between the labels and predictions.

class SparseTopKCategoricalAccuracy: Computes how often integer targets are in the top K predictions.

class SpecificityAtSensitivity: Computes best specificity where sensitivity is >= specified value.

class SquaredHinge: Computes the squared hinge metric between y_true and y_pred.

class Sum: Computes the (weighted) sum of the given values.

class TopKCategoricalAccuracy: Computes how often targets are in the top K predictions.

class TrueNegatives: Calculates the number of true negatives.

class TruePositives: Calculates the number of true positives.

Module `tf.keras.metrics` Functions

KLD(...): Computes Kullback-Leibler divergence loss between y_true and y_pred.

MAE(...): Computes the mean absolute error between labels and predictions.

MAPE(...): Computes the mean absolute percentage error between y_true and y_pred

MSE(...): Computes the mean squared error between labels and predictions.

MSLE(...): Computes the mean squared logarithmic error between y_true and y_pred.

binary_accuracy(...): Calculates how often predictions matches binary labels.

binary_crossentropy(...): Computes the binary crossentropy loss.

categorical_accuracy(...): Calculates how often predictions matches one-hot labels.

categorical_crossentropy(...): Computes the categorical crossentropy loss.

deserialize(...): Deserializes a serialized metric class/function instance.

get(...): Retrieves a Keras metric as a function/Metric class instance.

hinge(...): Computes the hinge loss between y_true and y_pred.

kl_divergence(...): Computes Kullback-Leibler divergence loss between y_true and y_pred.

kld(...): Computes Kullback-Leibler divergence loss between y_true and y_pred.

Module `tf.keras.metrics` Functions

kullback_leibler_divergence(...): Computes Kullback-Leibler divergence loss between y_true and y_pred.

mae(...): Computes the mean absolute error between labels and predictions.

mape(...): Computes the mean absolute percentage error between y_true and y_pred.

mean_absolute_error(...): Computes the mean absolute error between labels and predictions.

mean_absolute_percentage_error(...): Computes the mean absolute percentage error between y_true and y_pred.

mean_squared_error(...): Computes the mean squared error between labels and predictions.

mean_squared_logarithmic_error(...): Computes the mean squared logarithmic error between y_true and y_pred.

mse(...): Computes the mean squared error between labels and predictions.

msle(...): Computes the mean squared logarithmic error between y_true and y_pred.

poisson(...): Computes the Poisson loss between y_true and y_pred.

serialize(...): Serializes metric function or Metric instance.

sparse_categorical_accuracy(...): Calculates how often predictions matches integer labels.

sparse_categorical_crossentropy(...): Computes the sparse categorical crossentropy loss.

sparse_top_k_categorical_accuracy(...): Computes how often integer targets are in the top K predictions.

squared_hinge(...): Computes the squared hinge loss between y_true and y_pred.

top_k_categorical_accuracy(...): Computes how often targets are in the top K predictions.

tf.keras.regularizers.Regularizer

- Regularizers allow you to apply penalties on layer parameters or layer activity during optimization. These penalties are summed into the loss function that the network optimizes.
- Regularization penalties are applied on a per-layer basis. The exact API will depend on the layer, but many layers (e.g. Dense, Conv1D, Conv2D and Conv3D) have a unified API.
- These layers expose 3 keyword arguments:
  - `kernel_regularizer`: Regularizer to apply a penalty on the layer's kernel
  - `bias_regularizer`: Regularizer to apply a penalty on the layer's bias
  - `activity_regularizer`: Regularizer to apply a penalty on the layer's output
- All layers (including custom layers) expose `activity_regularizer` as a settable property, whether or not it is in the constructor arguments.
- The value returned by the `activity_regularizer` is divided by the input batch size so that the relative weighting between the weight regularizers and the activity regularizers does not change with the batch size.
- You can access a layer's regularization penalties by calling `layer.losses` after calling the layer on inputs.

Example

```
layer = tf.keras.layers.Dense(
    5, input_dim=5,
    kernel_initializer='ones',
    kernel_regularizer=tf.keras.regularizers.L1(0.01),
    activity_regularizer=tf.keras.regularizers.L2(0.01))
tensor = tf.ones(shape=(5, 5)) * 2.0
out = layer(tensor
```

Summary

- The most common ways to prevent overfitting in neural networks are:
  - Getting more training data.
  - Reducing the capacity of the network.
  - Adding weight regularization.
  - Adding dropout layers.