CME 216, ME 343 - Spring 2020

Eric Darve, ICME

Different ways of building DNNs in TF

- Sequential models
- Functional API
- Custom models

The last approach is the most general but also the most complicated.

We will review it briefly for now.

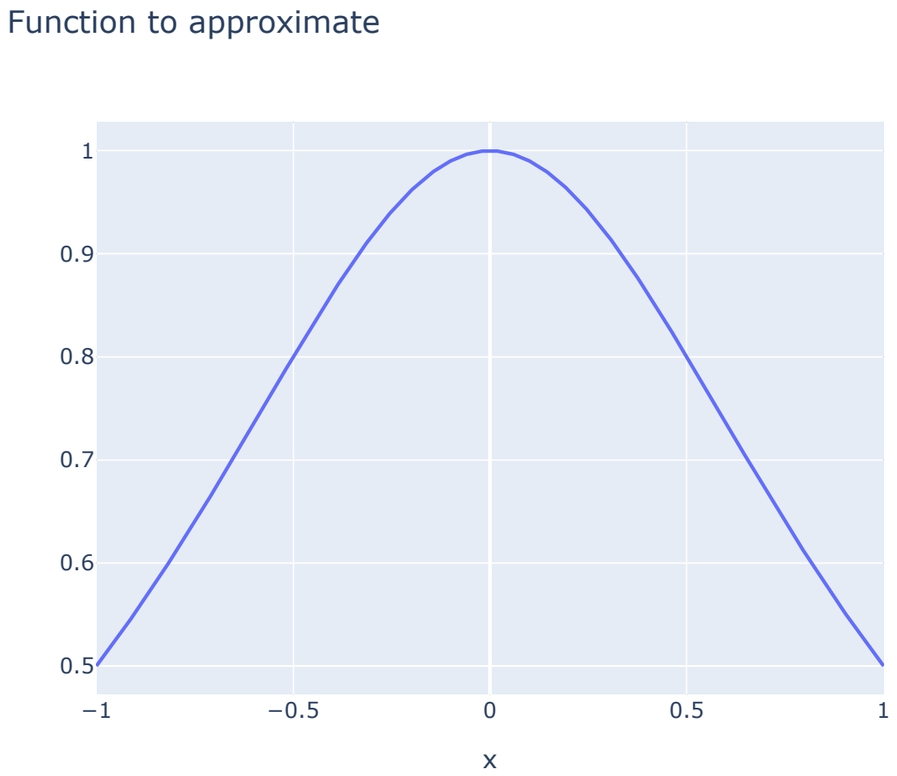
Sequential models are the easiest to work with.

They correspond to a sequence of layers where layers are simply "stacked."

Layer $i + 1$ takes as input the output of layer i.

The functional API allows writing more general models.

Let us consider an example. Let us say we want to approximate this function.



Our function has 1 input and 1 output.

We are going to build a DNN that takes the same input x and tries to approximate our function.

Let's consider the following DNN model:

- Input has size 1
- The model has 3 hidden layers of size 4. The activation function is tanh.
- The output has size 1 and uses a linear activation function.

In Keras, this is very easy to do.

Declare a sequential model:

```
model = keras.models.Sequential()
```

Construct all the layers:

```
model.add(keras.layers.InputLayer(input_shape=1))
model.add(keras.layers.Dense(4, activation="tanh"))
model.add(keras.layers.Dense(4, activation="tanh"))
model.add(keras.layers.Dense(4, activation="tanh"))
model.add(keras.layers.Dense(1, activation="linear"))
```

`Dense` is what we have been using all along. That we apply a linear transformation of the form

$$\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

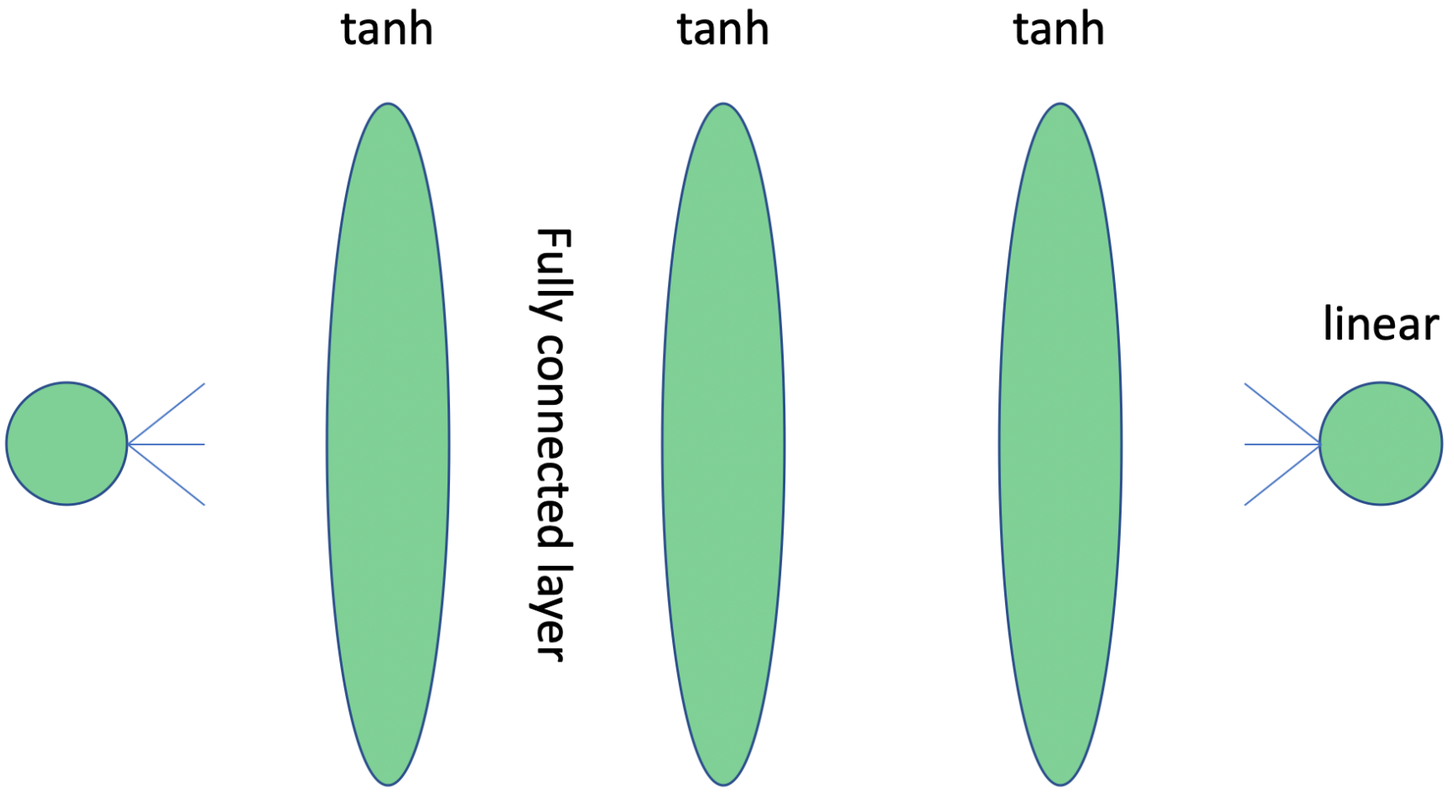
where $\boldsymbol{W}$ is a dense matrix.

`input_shape=1` is the size of the input.

In our case, it's just x.

We use the `tanh` activation function.

The last layer is simply a linear combination of hidden layer 3.


tanh
tanh
tanh
Fully connected layer
linear

Various functions can be used to query the DNN.

```
model.summary()
```

The output is:

Output

```
Model: "sequential"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
dense (Dense) (None, 4) 8
_________________________________________________________________
dense_1 (Dense) (None, 4) 20
_________________________________________________________________
dense_2 (Dense) (None, 4) 20
_________________________________________________________________
dense_3 (Dense) (None, 1) 5
=================================================================
Total params: 53
Trainable params: 53
```

The model is optimized using:

```
sgd = optimizers.SGD(lr=learning_rate)
model.compile(loss='mse', optimizer=sgd, metrics=['mse','mae'])
```

`SGD` is the stochastic gradient descent method.

We will explore later on how it works.

`lr` is the parameter that determines the step size in the gradient descent.

This is basically the parameter α we previously had.

`loss` is the function that the optimizer will try to minimize.

Common options are `mean_absolute_error`, `mean_squared_error`.

We will discuss other types of losses later on.

Documentation page

`metrics` are used to monitor convergence during training.

Documentation page

To optimize or fit the model using the training data:

```
history = model.fit(X_train, y_train, epochs=n_epochs,
validation_data=(X_valid, y_valid))
```

An epoch in deep learning corresponds to 1 iteration of the gradient descent method.

The parameter `n_epochs` controls how many iterations we should perform.

The function

```
model.evaluate(X_test, y_test)
```

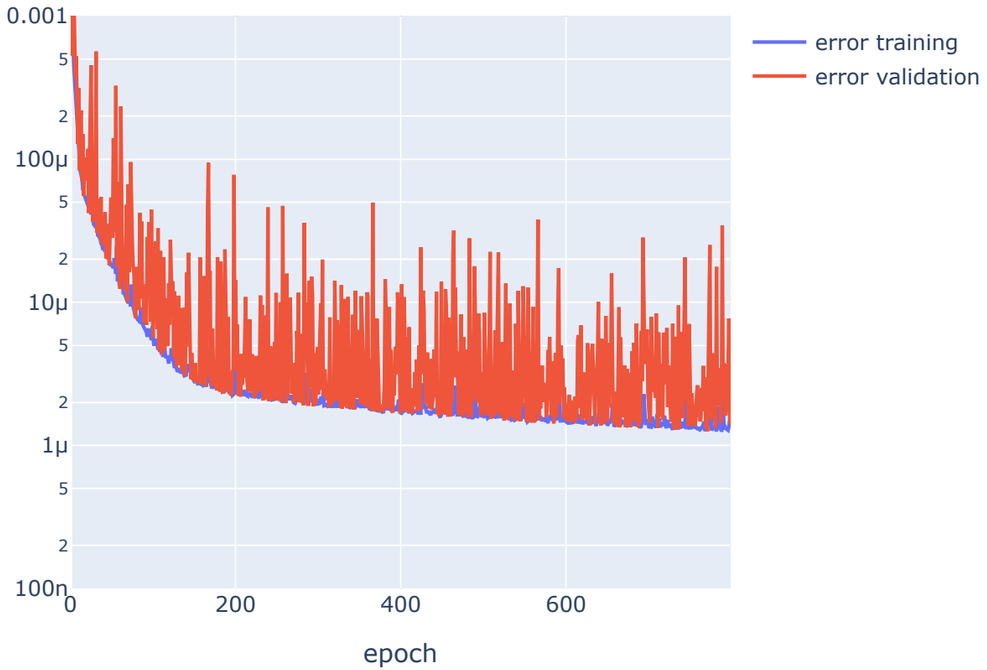
can be used to calculate the score or error for a given test set.

```
model.predict(X_test)
```

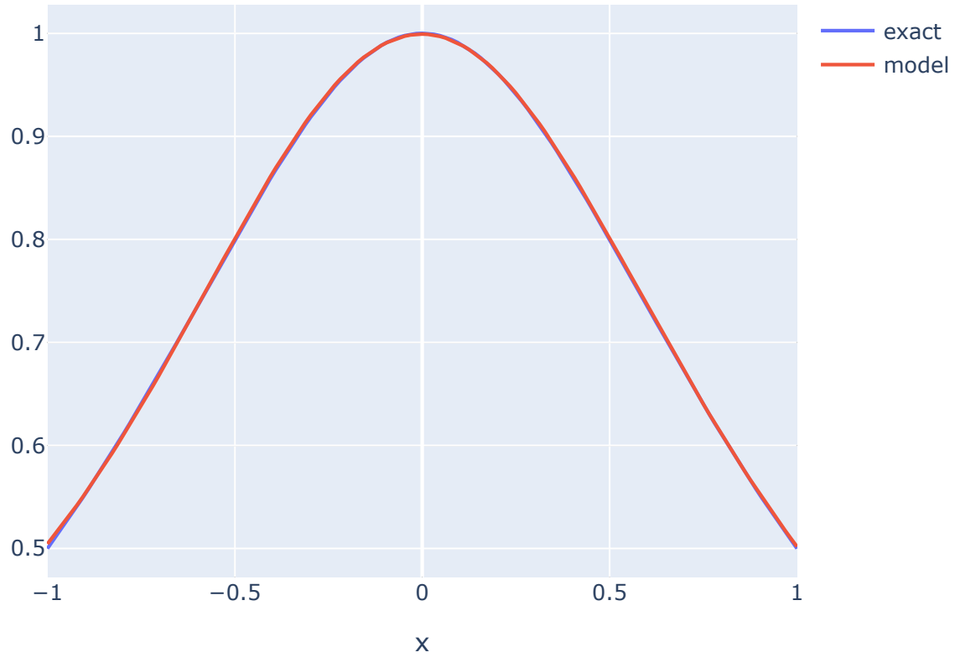
can be used to predict the output using our DNN model.

Please see the Keras guide for additional examples.

Here is the error convergence. We chose the mean squared error.



Model and true function



Absolute error