Advanced Deep Learning with Keras

Deeper machine learning technique

Black Raven (James Ng)
08 Mar 2021 · 32 min read

This is a memo to share what I have learnt in Advanced Deep Learning with Keras, capturing the learning objectives as well as my personal notes. The course is taught by Zachary Deane-Mayer from DataCamp, and it includes 4 chapters:

Chapter 1: The Keras Functional API
Chapter 2: Two Input Networks Using Categorical Embeddings, Shared Layers, and Merge Layers
Chapter 3: Multiple Inputs: 3 Inputs (and Beyond!)
Chapter 4: Multiple Outputs

This course shows you how to solve a variety of problems using the versatile Keras functional API. You will start with simple, multi-layer dense networks (also known as multi-layer perceptrons), and continue on to more complicated architectures. The course will cover how to build models with multiple inputs and a single output, as well as how to share weights between layers in a model. We will also cover advanced topics such as category embeddings and multiple-output networks. If you've ever wanted to train a network that does both classification and regression, then this course is for you!

Chapter 1. The Keras Functional API

In this chapter, you'll become familiar with the basics of the Keras functional API. You'll build a simple functional network using functional building blocks, fit it to data, and make predictions.

Keras input and dense layers

```
from keras.layers import Input
input_tensor = Input(shape=(1,))
print(input_tensor)

<tf.Tensor 'input_1:0' shape=(?, 1) dtype=float32>
```


```
from keras.layers import Dense
output_layer = Dense(1)
print(output_layer)

<keras.layers.core.Dense at 0x7f22e0295a58>
```

```
from keras.layers import Input, Dense
input_tensor = Input(shape=(1,))
output_layer = Dense(1)
output_tensor = output_layer(input_tensor)
print(output_tensor)

<tf.Tensor 'dense_1/BiasAdd:0' shape=(?, 1) dtype=float32>
```

Input layers

The first step in creating a neural network model is to define the *Input* layer. This layer takes in raw data, usually in the form of numpy arrays. The shape of the Input layer defines how many variables your neural network will use. For example, if the input data has 10 columns, you define an Input layer with a shape of `(10,)`.

In this case, you are only using one input in your network.

*This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the **Keras Cheat Sheet** and keep it handy!*

```
# Import Input from keras.layers
from keras.layers import Input

# Create an input layer of shape 1
input_tensor = Input(shape=(1,))
```

Remember that the input layer allows your model to load data.

Dense layers

Once you have an Input layer, the next step is to add a *Dense* layer.

Dense layers learn a weight matrix, where the first dimension of the matrix is the dimension of the input data, and the second dimension is the dimension of the output data. Recall that your Input layer has a shape of 1. In this case, your output layer will also have a shape of 1. This means that the Dense layer will learn a 1x1 weight matrix.

In this exercise, you will add a dense layer to your model, after the input layer.

```
# Load layers
from keras.layers import Input, Dense

# Input layer (from previous exercise)
input_tensor = Input(shape=(1,))

# Create a dense layer
output_layer = Dense(1)

# Connect the dense layer to the input_tensor
output_tensor = output_layer(input_tensor)
```

This network will take the input, apply a linear coefficient to it, and return the result.

Output layers

Output layers are simply Dense layers! Output layers are used to reduce the dimension of the inputs to the dimension of the outputs. You'll learn more about output dimensions in chapter 4, but for now, you'll always use a single output in your neural networks, which is equivalent to `Dense(1)` or a dense layer with a single unit.

```
# Load layers
from keras.layers import Input, Dense

# Input layer
input_tensor = Input(shape=(1,))

# Create a dense layer and connect the dense layer to the input_tensor in one step
# Note that you did this in 2 steps in the previous exercise, but are doing it in one step now
output_tensor = Dense(1)(input_tensor)
```

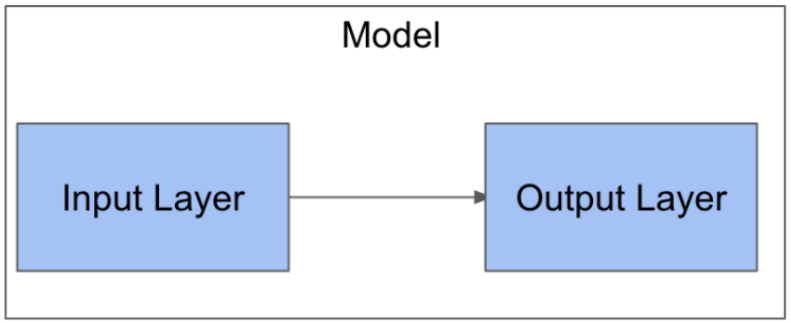
The output layer allows your model to make predictions.

Build and compile a model

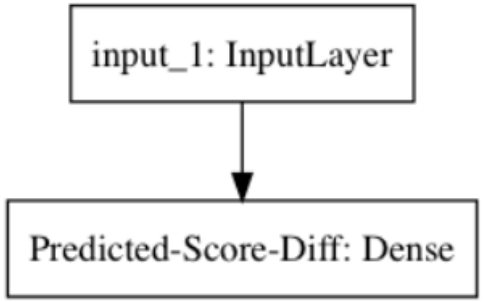
```
from keras.layers import Input, Dense
input_tensor = Input(shape=(1,))
output_tensor = Dense(1)(input_tensor)
model.compile(optimizer='adam', loss='mae')
```

```
from keras.models import Model
model = Model(input_tensor, output_tensor)
```

```
model.summary()


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 1)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```



```
input_tensor = Input(shape=(1,))
output_layer = Dense(1, name='Predicted-Score-Diff')
output_tensor = output_layer(input_tensor)
model = Model(input_tensor, output_tensor)
plot_model(model, to_file ='model.png')

from matplotlib import pyplot as plt
img = plt.imread('model.png')
plt.imshow(img)
plt.show()
```



Build a model

Once you've defined an input layer and an output layer, you can build a Keras model. The model object is how you tell Keras where the model starts and stops: where data comes in and where predictions come out.

```
# Input/dense/output layers
from keras.layers import Input, Dense
input_tensor = Input(shape=(1,))
output_tensor = Dense(1)(input_tensor)

# Build the model
from keras.models import Model
model = Model(input_tensor, output_tensor)
```

This model is a complete neural network, ready to learn from data and make prediction.

Compile a model

The final step in creating a model is *compiling* it. Now that you've created a model, you have to compile it before you can fit it to data. This finalizes your model, freezes all its settings, and prepares it to meet some data!

During compilation, you specify the optimizer to use for fitting the model to the data, and a loss function. `'adam'` is a good default optimizer to use, and will generally work well. Loss function depends on the problem at hand. Mean squared error is a common loss function and will optimize for predicting the mean, as is done in *least squares regression*.

Mean absolute error optimizes for the median and is used in quantile regression. For this dataset, `'mean_absolute_error'` works pretty well, so use it as your loss function.

```
# Compile the model
model.compile(optimizer='adam', loss='mean_absolute_error')
```

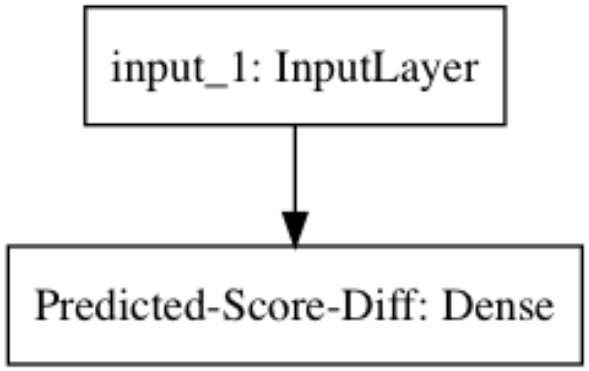
Compiling a model is the final step before fitting it.

Visualize a model

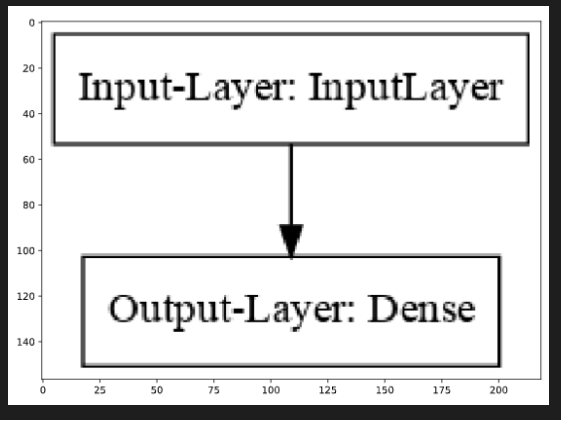
Now that you've compiled the model, take a look a the result of your hard work! You can do this by looking at the model summary, as well as its plot.

The summary will tell you the names of the layers, as well as how many units they have and how many parameters are in the model.

The plot will show how the layers connect to each other.



```
# Import the plotting function
from keras.utils import plot_model
import matplotlib.pyplot as plt

# Summarize the model
model.summary()

# Plot the model
plot_model(model, to_file='model.png')

# Display the image
data = plt.imread('model.png')
plt.imshow(data)
plt.show()
```



It turns out neural networks aren't really black boxes after all!

Fit and evaluate a model

Goal: Predict tournament outcomes

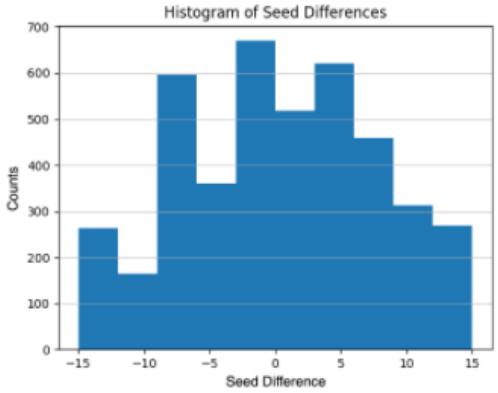
Data Available: team ratings from the tournament organizers

```
import pandas as pd
games_tourney = pd.read_csv('datasets/games_tourney.csv')
games_tourney.head()

Out[1]:
```

	season	team_1	team_2	home	seed_diff	score_diff	score_1	score_2	won
0	1985	288	73	0	-3	-9	41	50	0
1	1985	5929	73	0	4	6	61	55	1
2	1985	9884	73	0	5	-4	59	63	0
3	1985	73	288	0	3	9	50	41	1
4	1985	3920	410	0	1	-9	54	63	0

Input = seed_diff
Output = score_diff

- Seed difference: 15
  - Team 1: 16
  - Team 2: 1
- Seed difference: -15
  - Team 1: 1
  - Team 2: 16




```
from keras.models import Model
from keras.layers import Input, Dense
input_tensor = Input(shape=(1,))
output_tensor = Dense(1)(input_tensor)
model = Model(input_tensor, output_tensor)
model.compile(optimizer='adam', loss='mae')
```

```
from pandas import read_csv
games = read_csv('datasets/games_tourney.csv')
model.fit(games['seed_diff'],
          games['score_diff'],
          batch_size=64,
          validation_split=.20,
          verbose=True)
```

```
model.evaluate(games_test['seed_diff'],
               games_test['score_diff'])

1000/1000 [==============================] - 0s 26us/step
Out[1]: 9.742335235595704
```

error metrics ‘mae’ = 9.74

Fit the model to the tournament basketball data

Now that the model is compiled, you are ready to fit it to some data!

In this exercise, you'll use a dataset of scores from US College Basketball tournament games. Each row of the dataset has the team ids: `team_1` and `team_2`, as integers. It also has the seed difference between the teams (seeds are assigned by the tournament committee and represent a ranking of how strong the teams are) and the score difference of the game (e.g. if `team_1` wins by 5 points, the score difference is `5`).

To fit the model, you provide a matrix of X variables (in this case one column: the seed difference) and a matrix of Y variables (in this case one column: the score difference).

The `games_tourney` DataFrame along with the compiled `model` object is available in your workspace.

```
# Now fit the model
model.fit(games_tourney_train['seed_diff'], games_tourney_train['score_diff'],
          epochs=1,
          batch_size=128,
          validation_split=0.10,
          verbose=True)
```

```
<script.py> output:
    Train on 3087 samples, validate on 343 samples
    Epoch 1/1

     128/3087 [>.............................] - ETA: 9s - loss: 12.6147
    1664/3087 [===============>..............] - ETA: 0s - loss: 12.8104
    3087/3087 [==============================] - 1s 176us/step - loss: 12.6617 - val_loss: 11.8746
```

Now your model has learned something about the basketball data!

Evaluate the model on a test set

After fitting the model, you can evaluate it on new data. You will give the model a new `X` matrix (also called test data), allow it to make predictions, and then compare to the known `y` variable (also called target data).

In this case, you'll use data from the post-season tournament to evaluate your model. The tournament games happen after the regular season games you used to train our model, and are therefore a good evaluation of how well your model performs out-of-sample.

The `games_tourney_test` DataFrame along with the fitted `model` object is available in your workspace.

```
# Load the X variable from the test data
```

```
X_test = games_tourney_test['seed_diff']

# Load the y variable from the test data
y_test = games_tourney_test['score_diff']

# Evaluate the model on the test data
print(model.evaluate(X_test, y_test, verbose=False))
```

```
<script.py> output:
    10.06973339669147
```

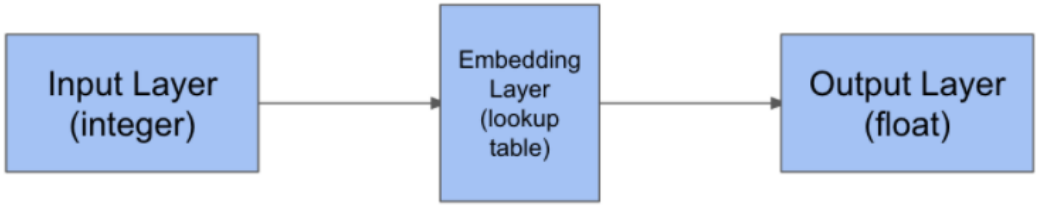
Looks like your model makes pretty good predicitions!

Chapter 2. Two Input Networks Using Categorical Embeddings, Shared Layers, and Merge Layers

In this chapter, you will build two-input networks that use categorical embeddings to represent high-cardinality data, shared layers to specify re-usable building blocks, and merge layers to join multiple inputs to a single output. By the end of this chapter, you will have the foundational building blocks for designing neural networks with complex data flows.

Category embeddings

- Input: integers
- Output: floats
- Note: Increased dimensionality: output layer flattens back to 2D




```
from keras.layers import Embedding
input_tensor = Input(shape=(1,))
n_teams = 10887
embed_layer = Embedding(input_dim=n_teams,
                        input_length=1,
                        output_dim=1,
                        name='Team-Strength-Lookup')
embed_tensor = embed_layer(input_tensor)
```

```
from keras.layers import Flatten
flatten_tensor = Flatten()(embed_tensor)
```

```
input_tensor = Input(shape=(1,))
n_teams =  10887
embed_layer = Embedding(input_dim=n_teams,
                        input_length=1,
                        output_dim=1,
                        name='Team-Strength-Lookup')
embed_tensor = embed_layer(input_tensor)
flatten_tensor = Flatten()(embed_tensor)
model = Model(input_tensor, flatten_tensor)
```

Define team lookup

Shared layers allow a model to use the same weight matrix for multiple steps. In this exercise, you will build a "team strength" layer that represents each team by a single number. You will use this number for both teams in the model. The model will learn a number for each team that works well both when the team is `team_1` and when the team is `team_2` in the input data.

The `games_season` DataFrame is available in your workspace.

```
# Imports
from keras.layers import Embedding
from numpy import unique

# Count the unique number of teams
n_teams = unique(games_season['team_1']).shape[0]

# Create an embedding layer
team_lookup = Embedding(input_dim=n_teams,
                        output_dim=1,
                        input_length=1,
                        name='Team-Strength')
```

The embedding layer is a lot like a dictionary, but your model learns the values for each key.

Define team model

The team strength lookup has three components: an input, an embedding layer, and a flatten layer that creates the output.

If you wrap these three layers in a model with an input and output, you can re-use that stack of three layers at multiple places.

Note again that the weights for *all three* layers will be shared everywhere we use them.

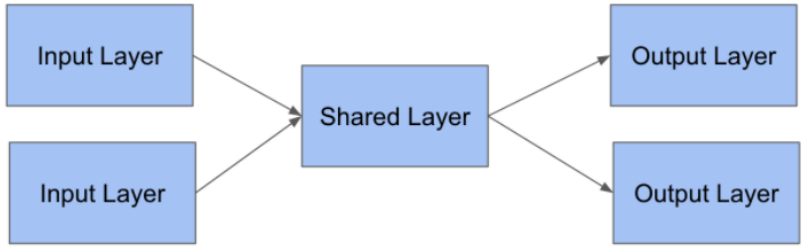
```
# Imports
from keras.layers import Input, Embedding, Flatten
from keras.models import Model

# Create an input layer for the team ID
teamid_in = Input(shape=(1,))

# Lookup the input in the team strength embedding layer
strength_lookup = team_lookup(teamid_in)

# Flatten the output
strength_lookup_flat = Flatten()(strength_lookup)

# Combine the operations into a single, re-usable model
team_strength_model = Model(teamid_in, strength_lookup_flat, name='Team-Strength-Model')
```

The model will be reusable, so you can use it in two places in your final model.

Shared layers

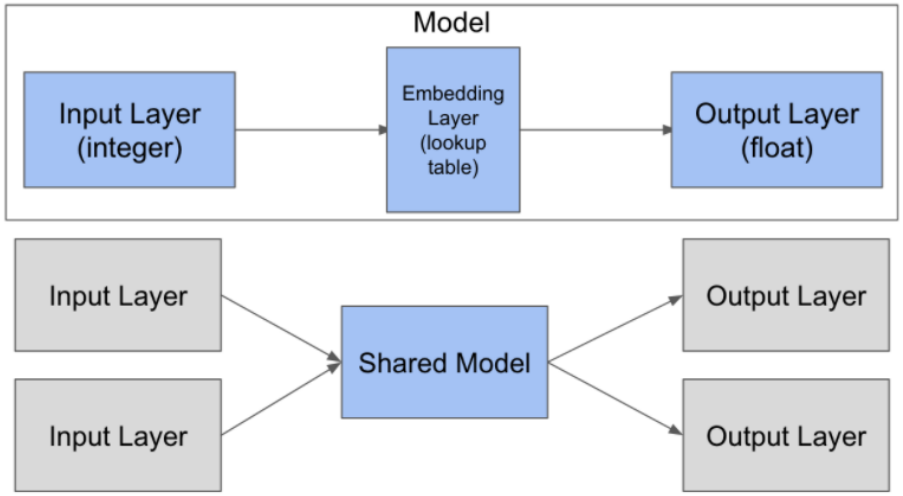
```
shared_layer = Dense(1)
output_tensor_1 = shared_layer(input_tensor_1)
output_tensor_2 = shared_layer(input_tensor_2)
```

- Require the functional API
- Very flexible



```
input_tensor = Input(shape=(1,))
n_teams = 10887
embed_layer = Embedding(input_dim=n_teams,
                        input_length=1,
                        output_dim=1,
                        name='Team-Strength-Lookup')
embed_tensor = embed_layer(input_tensor)
flatten_tensor = Flatten()(embed_tensor)
model = Model(input_tensor, flatten_tensor)
```

```
input_tensor_1 = Input((1,))
input_tensor_2 = Input((1,))
output_tensor_1 = model(input_tensor_1)
output_tensor_2 = model(input_tensor_2)
```



Defining two inputs

In this exercise, you will define two input layers for the two teams in your model. This allows you to specify later in the model how the data from each team will be used differently.

```
# Load the input layer from keras.layers
from keras.layers import Input

# Input layer for team 1
team_in_1 = Input(shape=(1,), name='Team-1-In')

# Separate input layer for team 2
team_in_2 = Input(shape=(1,), name='Team-2-In')
```

These two inputs will be used later for the shared layer.

Lookup both inputs in the same model

Now that you have a team strength model and an input layer for each team, you can lookup the team inputs in the shared team strength model. The two inputs will share the same weights.

In this dataset, you have 10,888 unique teams. You want to learn a strength rating for each team, such that if any pair of teams plays each other, you can predict the score, even if those two teams have never played before. Furthermore, you want the strength rating to be the same, regardless of whether the team is the home team or the away team.

To achieve this, you use a shared layer, defined by the re-usable model (`team_strength_model()`) you built in exercise 3 and the two input layers (`team_in_1` and `team_in_2`) from the previous exercise, all of which are available in your workspace.

```
# Lookup team 1 in the team strength model
team_1_strength = team_strength_model(team_in_1)

# Lookup team 2 in the team strength model
team_2_strength = team_strength_model(team_in_2)
```

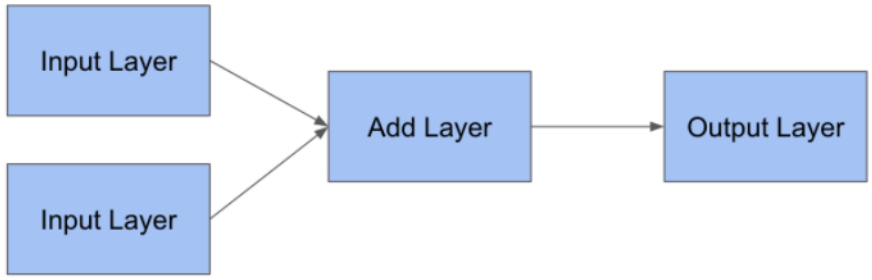
Now your model knows how strong each team is.

Merge layers

- add, subtract, multiply, concatenate

```
from keras.layers import Input, Add
in_tensor_1 = Input((1,))
in_tensor_2 = Input((1,))
out_tensor = Add()([in_tensor_1, in_tensor_2])
```

```
in_tensor_3 = Input((1,))
out_tensor = Add()([in_tensor_1, in_tensor_2, in_tensor_3])
```




```
from keras.models import Model
model = Model([in_tensor_1, in_tensor_2], out_tensor)
```

```
model.compile(optimizer='adam', loss='mean_absolute_error')
```

Output layer using shared layer

Now that you've looked up how "strong" each team is, subtract the team strengths to determine which team is expected to win the game.

This is a bit like the seeds that the tournament committee uses, which are also a measure of team strength. But rather than using seed differences to predict score differences, you'll use the difference of your own team strength model to predict score differences.

The subtract layer will combine the weights from the two layers by subtracting them.

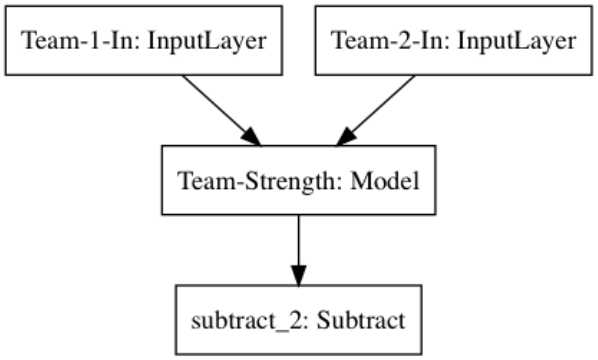
```
# Import the Subtract layer from keras
from keras.layers import Subtract

# Create a subtract layer using the inputs from the previous exercise
score_diff = Subtract()([team_1_strength, team_2_strength])
```

This setup subracts the team strength ratings to determine a winner.

Model using two inputs and one output

Now that you have your two inputs (team id 1 and team id 2) and output (score difference), you can wrap them up in a model so you can use it later for fitting to data and evaluating on new data.

Your model will look like the following diagram:




```
# Imports
from keras.layers import Subtract
```

```
from keras.models import Model

# Subtraction layer from previous exercise
score_diff = Subtract()([team_1_strength, team_2_strength])

# Create the model
model = Model([team_in_1, team_in_2], score_diff)

# Compile the model
model.compile(optimizer='adam', loss='mean_absolute_error')
```

Now your model is finalized and ready to fit to data.

Predict from your model

`model.fit([data_1, data_2], target)` list of inputs

```
model.predict([np.array([[1]]), np.array([[2]])])
array([[3.]], dtype=float32)
```

```
model.predict([np.array([[42]]), np.array([[119]])])
array([[161.]], dtype=float32)
```

```
model.evaluate([np.array([[-1]]), np.array([[-2]])], np.array([[-3]]))
```

```
1/1 [==============================] - 0s 801us/step
Out[21]: 0.0
```

Fit the model to the regular season training data

Now that you've defined a complete team strength model, you can fit it to the basketball data! Since your model has two inputs now, you need to pass the input data as a list.

```python
# Get the team_1 column from the regular season data
input_1 = games_season['team_1']

# Get the team_2 column from the regular season data
input_2 = games_season['team_2']

# Fit the model to input 1 and 2, using score diff as a target
model.fit([input_1, input_2],
          games_season['score_diff'],
          epochs=1,
          batch_size=2048,
          validation_split=0.10,
          verbose=True)
```

```
<script.py> output:
    Train on 280960 samples, validate on 31218 samples
    Epoch 1/1

      2048/280960 [..............................] - ETA: 16s - loss: 12.0254
     34816/280960 [==>...........................] - ETA: 1s - loss: 12.2352
     73728/280960 [======>.......................] - ETA: 0s - loss: 12.1595
    108544/280960 [==========>...................] - ETA: 0s - loss: 12.1196
    143360/280960 [==============>...............] - ETA: 0s - loss: 12.1285
    180224/280960 [==================>...........] - ETA: 0s - loss: 12.1261
    215040/280960 [=====================>........] - ETA: 0s - loss: 12.1222
    241664/280960 [========================>.....] - ETA: 0s - loss: 12.1231
    274432/280960 [============================>.] - ETA: 0s - loss: 12.1246
    280960/280960 [==============================] - 1s 2us/step - loss: 12.1203 - val_loss: 11.8384
```

Now our model has learned a strength rating for every team.

Evaluate the model on the tournament test data

The model you fit to the regular season data (`model`) in the previous exercise and the tournament dataset (`games_tourney`) are available in your workspace.

In this exercise, you will evaluate the model on this new dataset. This evaluation will tell you how well you can predict the tournament games, based on a model trained with the regular season data. This is interesting because many teams play each other in the tournament that did not play in the regular season, so this is a very good check that your model is not overfitting.

```
# Get team_1 from the tournament data
input_1 = games_tourney['team_1']

# Get team_2 from the tournament data
input_2 = games_tourney['team_2']

# Evaluate the model using these inputs
print(model.evaluate([input_1, input_2], games_tourney['score_diff'], verbose=False))
```

```
<script.py> output:
    11.649009290595183
```

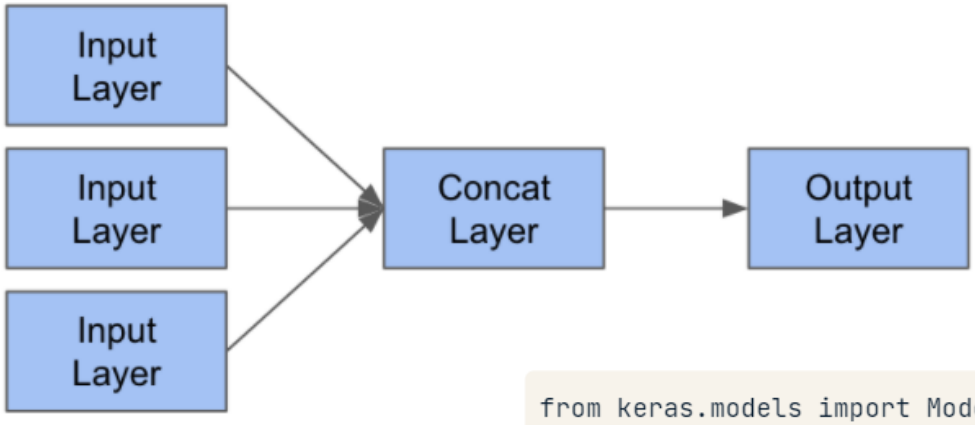
Its time to move on to models with more than two inputs.

Chapter 3. Multiple Inputs: 3 Inputs (and Beyond!)

In this chapter, you will extend your 2-input model to 3 inputs, and learn how to use Keras' summary and plot functions to understand the parameters and topology of your neural networks. By the end of the chapter, you will understand how to extend a 2-input model to 3 inputs and beyond.

Three-input models

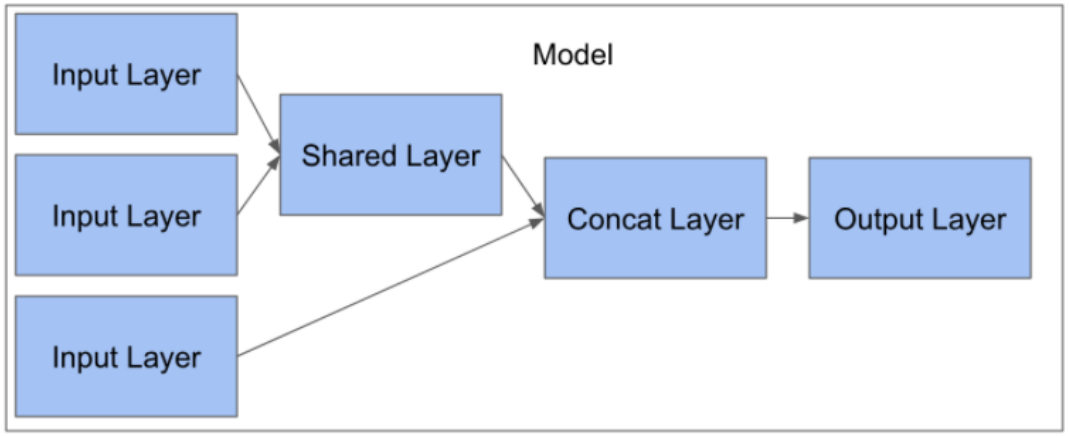
```
from keras.layers import Input, Concatenate, Dense
in_tensor_1 = Input(shape=(1,))
in_tensor_2 = Input(shape=(1,))
in_tensor_3 = Input(shape=(1,))
out_tensor = Concatenate()([in_tensor_1, in_tensor_2, in_tensor_3])
output_tensor = Dense(1)(out_tensor)
```




```
from keras.models import Model
model = Model([in_tensor_1, in_tensor_2, in_tensor_3], out_tensor)
```

```
shared_layer = Dense(1)
shared_tensor_1 = shared_layer(in_tensor_1)
shared_tensor_2 = shared_layer(in_tensor_1)
out_tensor = Concatenate()([shared_tensor_1, shared_tensor_2, in_tensor_3])
out_tensor = Dense(1)(out_tensor)
```

```
from keras.models import Model
model = Model([in_tensor_1, in_tensor_2, in_tensor_3], out_tensor)
```




```
from keras.models import Model
model = Model([in_tensor_1, in_tensor_2, in_tensor_3], out_tensor)
model.compile(loss='mae', optimizer='adam')
```

```
model.fit([[train['col1'], train['col2'], train['col3']],
          train_data['target'])
```

```
model.evaluate([[test['col1'], test['col2'], test['col3']],
               test['target'])
```

Make an input layer for home vs. away

Now you will make an improvement to the model you used in the previous chapter for regular season games. You know there is a well-documented home-team advantage in basketball, so you will add a new input to your model to capture this effect.

This model will have three inputs: `team_id_1`, `team_id_2`, and `home`. The team IDs will be integers that you look up in your team strength model from the previous chapter, and home will be a binary variable, 1 if `team_1` is playing at home, 0 if they are not.

The `team_strength_model` you used in the previous chapter has been loaded into your workspace. After applying it to each input, use a Concatenate layer to join the two team strengths and with the home vs away variable, and pass the result to a Dense layer.

```
# Create an Input for each team
team_in_1 = Input(shape=(1,), name='Team-1-In')
team_in_2 = Input(shape=(1,), name='Team-2-In')

# Create an input for home vs away
home_in = Input(shape=(1,), name='Home-In')

# Lookup the team inputs in the team strength model
team_1_strength = team_strength_model(team_in_1)
team_2_strength = team_strength_model(team_in_2)

# Combine the team strengths with the home input using a Concatenate layer, then add a Dense layer
out = Concatenate()([team_1_strength, team_2_strength, home_in])
out = Dense(1)(out)
```

Now you have a model with 3 inputs!

Make a model and compile it

Now that you've input and output layers for the 3-input model, wrap them up in a Keras model class, and then compile the model, so you can fit it to data and use it to make predictions on new data.

```
# Import the model class
from keras.models import Model

# Make a Model
model = Model([team_in_1, team_in_2, home_in], out)

# Compile the model
model.compile(optimizer='adam', loss='mean_absolute_error')
```

Now our 3-input model is ready to meet some data!

Fit the model and evaluate

Now that you've defined a new model, fit it to the regular season basketball data.

Use the `model` you fit in the previous exercise (which was trained on the regular season data) and evaluate the model on data for tournament games (`games_tourney`).

```
# Fit the model to the games_season dataset
model.fit([games_season['team_1'], games_season['team_2'], games_season['home']],
          games_season['score_diff'],
          epochs=1,
          verbose=True,
          validation_split=.10,
          batch_size=2048)

# Evaluate the model on the games_tourney dataset
print(model.evaluate([games_tourney['team_1'], games_tourney['team_2'], games_tourney['home']],
               games_tourney['score_diff'], verbose=False))
```

```
<script.py> output:
    Train on 280960 samples, validate on 31218 samples
    Epoch 1/1

      2048/280960 [..............................] - ETA: 14s - loss: 12.0596
     36864/280960 [==>...........................] - ETA: 1s - loss: 11.9829
     71680/280960 [======>.......................] - ETA: 0s - loss: 12.0097
    104448/280960 [==========>...................] - ETA: 0s - loss: 11.9995
    137216/280960 [=============>................] - ETA: 0s - loss: 12.0185
    163840/280960 [================>.............] - ETA: 0s - loss: 12.0125
    188416/280960 [===================>..........] - ETA: 0s - loss: 12.0251
    219136/280960 [======================>.......] - ETA: 0s - loss: 12.0267
    241664/280960 [========================>.....] - ETA: 0s - loss: 12.0104
    270336/280960 [===========================>..] - ETA: 0s - loss: 12.0038
    280960/280960 [==============================] - 1s 2us/step - loss: 12.0003 - val_loss: 12.3391
    11.68379570821799
```

Its time to further explore this model.

Summarizing and plotting models

The summary for a Keras model shows you all the layers in the model, as well as how many parameters each layer has. Importantly, Keras models can have non-trainable parameters that are fixed and do not change, as well as trainable parameters, that are learned from the data when the model is fit.

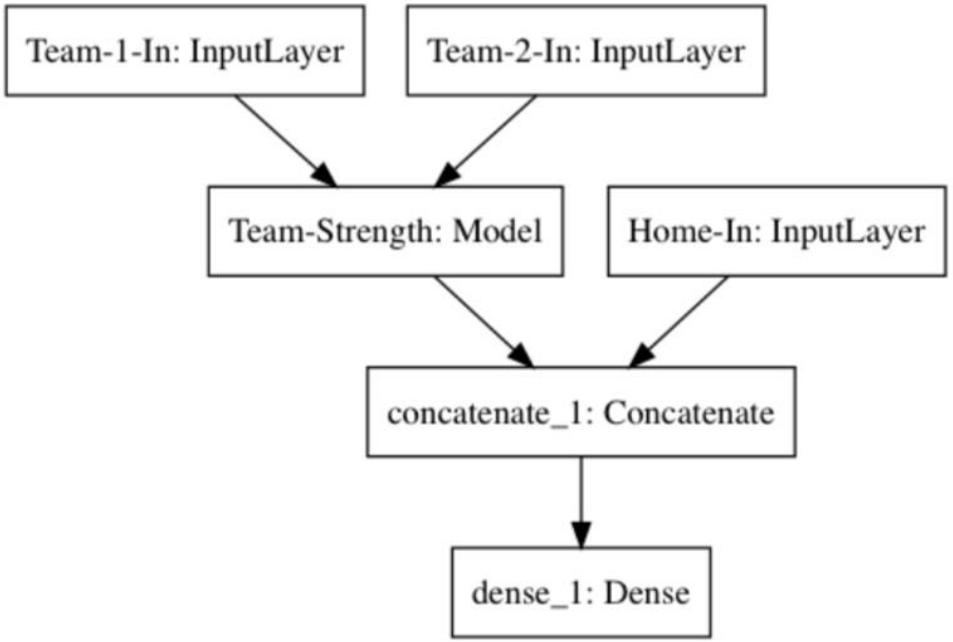
Models with more trainable parameters are typically more flexible. This can also make them more prone to overfitting. Models with fewer trainable parameters are less flexible, but therefore less likely to overfit.

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_1 (InputLayer)             (None, 1)             0
____________________________________________________________________________________________________
input_2 (InputLayer)             (None, 1)             0
____________________________________________________________________________________________________
input_3 (InputLayer)             (None, 1)             0
____________________________________________________________________________________________________
concatenate_1 (Concatenate)      (None, 3)             0           input_1[0][0]
                                                                   input_2[0][0]
                                                                   input_3[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 1)             4           concatenate_1[0][0]
====================================================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
____________________________________________________________________________________________________
```

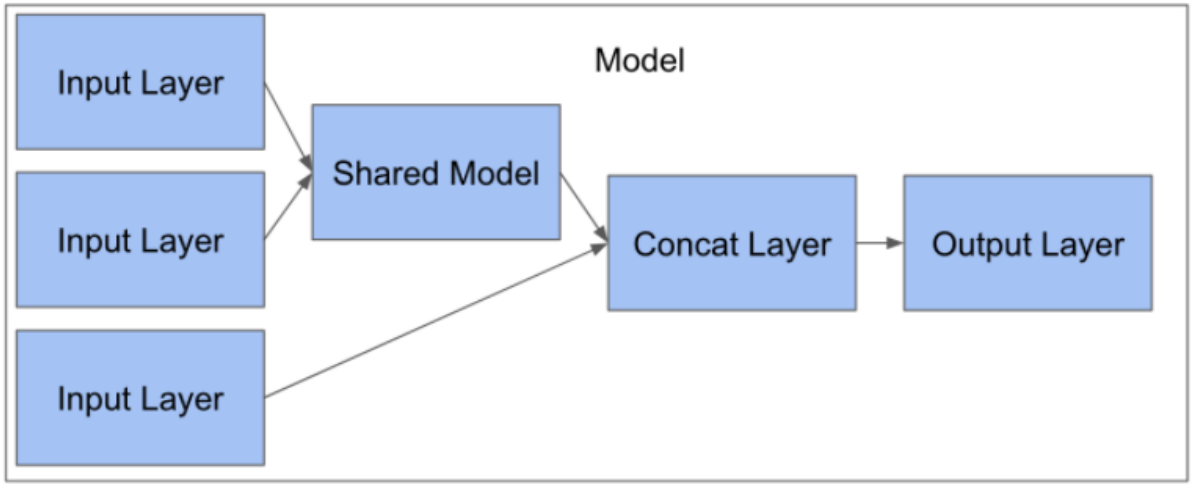
In this case, the model has three inputs. Since all three of them feed into one Dense layer, the model has four parameters: one per input plus a bias, or intercept. All of these parameters are trainable. A model's trainable parameters are usually in its Dense layers.

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 1)            0
__________________________________________________________________________________________________
embedding_1 (Embedding)         (None, 1, 1)         10887       input_1[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 1)            0           embedding_1[0][0]
__________________________________________________________________________________________________
input_2 (InputLayer)            (None, 1)            0
__________________________________________________________________________________________________
input_3 (InputLayer)            (None, 1)            0
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 3)            0           flatten_1[0][0]
                                                                 input_2[0][0]
                                                                 input_3[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 1)            4           concatenate_1[0][0]
==================================================================================================
Total params: 10,891
Trainable params: 10,891
Non-trainable params: 0
__________________________________________________________________________________________________
```

Here is the summary of a slightly more complicated model. You can see that this model has an embedding layer. Even though the dense layer still only has 4 parameters, the model has many more trainable parameters, because of the embedding layer. It's important to remember that embedding layers often add a very large number of trainable parameters to a model. Recall that embedding layers map integers to floats: each unique value of the embedding input gets a parameter for its output.



Here is the plot for a very complicated model. The box at the bottom of the image represents the model's output. In this case, the model has one output. Note that output layers have arrows coming in, but no arrows going out. The boxes in the middle of the image represent intermediate steps in the model. These boxes have arrows coming in, and arrows going out. Note that this model has a shared model: the team strength model, which is applied to two of the inputs before they are combined in the concatenate layer with the third input. The boxes at the top of the image represent the inputs. These boxes only have one arrow going out and none coming in.



Here's another way of looking at the same model, using the network diagrams I've made for the previous chapter's models. Shared models work exactly the same as shared layers. This is a useful abstraction because you can put together a sequence of layers to define a custom model, and then share the entire model in exactly the same way you'd share a layer. It's a little prettier when you make them by hand, but the auto-plotting function in Keras does a good job representing the actual structure of the model.

Model summaries

In this exercise, you will take a closer look at the summary of one of your 3-input models available in your workspace as `model`. Note how many layers the model has, how many parameters it has, and how many of those parameters are trainable/non-trainable.

How many *total* parameters does this model have?

IPython Shell | Slides

```
In [1]: model.summary()
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
Team-1-In (InputLayer)       (None, 1)                 0
_________________________________________________________________________________________________
Team-2-In (InputLayer)       (None, 1)                 0
_________________________________________________________________________________________________
Team-Strength (Model)        (None, 1)                 10888       Team-1-In[0][0]
                                                                   Team-2-In[0][0]
_________________________________________________________________________________________________
Home-In (InputLayer)         (None, 1)                 0
_________________________________________________________________________________________________
concatenate_1 (Concatenate)  (None, 3)                 0           Team-Strength[1][0]
                                                                   Team-Strength[2][0]
                                                                   Home-In[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 1)                 4           concatenate_1[0][0]
=================================================================================================
Total params: 10,892
Trainable params: 10,892
Non-trainable params: 0
_________________________________________________________________________________________________
```

- 0
- 4
- 10,888
- 10,892 (selected)

How many *trainable* parameters does this model have?
Zero trainable parameters would imply the model can't learn anything.
This model has an embedding layer, so it will have many more than 4 trainable parameters.

- 0
- 4
- 10,888
- 10,892 (selected)

Which layer of your model has the most trainable parameters?

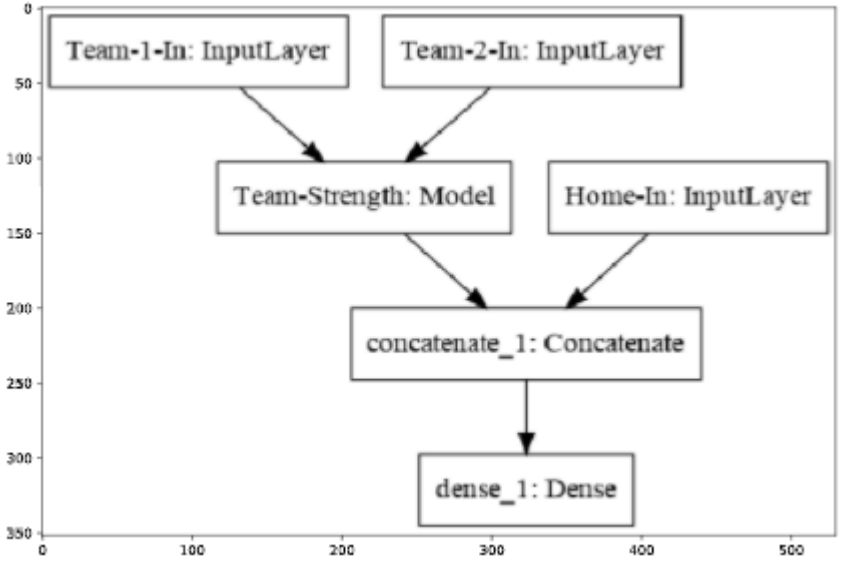
- Team-1-In (InputLayer)
- Team-2-In (InputLayer)
- Team-Strength (Model) (selected)
- Home-In (InputLayer)
- concatenate_1 (Concatenate)
- dense_1 (Dense)

Its time to plot this model.

Plotting models

In addition to summarizing your model, you can also plot your model to get a more intuitive sense of it. Your `model` is available in the workspace.

```
# Imports
import matplotlib.pyplot as plt
from keras.utils import plot_model

# Plot the model
plot_model(model, to_file='model.png')

# Display the image
data = plt.imread('model.png')
plt.imshow(data)
plt.show()
```



How many *inputs* does this model have?

- 1
- 2
- 3
- 4

How many *outputs* does this model have?

- (•) 1
- ( ) 2
- ( ) 3
- ( ) 4

Which layer is *shared* between 2 inputs?

- ( ) Team-1-In
- ( ) Team-2-In
- (•) Team-Strength
- ( ) dense_1

Its time to move on to stacked models.

Stacking models

"model stacking" = using the predictions from one model as an input to another model.
Model stacking is a very advanced data science concept. It is the most sophisticated way of combining models, and when done right can yield some of the most accurate models in existence. Model stacking is often employed to win popular predictive modeling competitions.

```
from pandas import read_csv
games_season = read_csv('datasets/games_season.csv')
games_season.head()
```

	team_1	team_2	home	score_diff
0	3745	6664	0	17
1	126	7493	1	7
2	288	3593	1	7
3	1846	9881	1	16
4	2675	10298	1	12

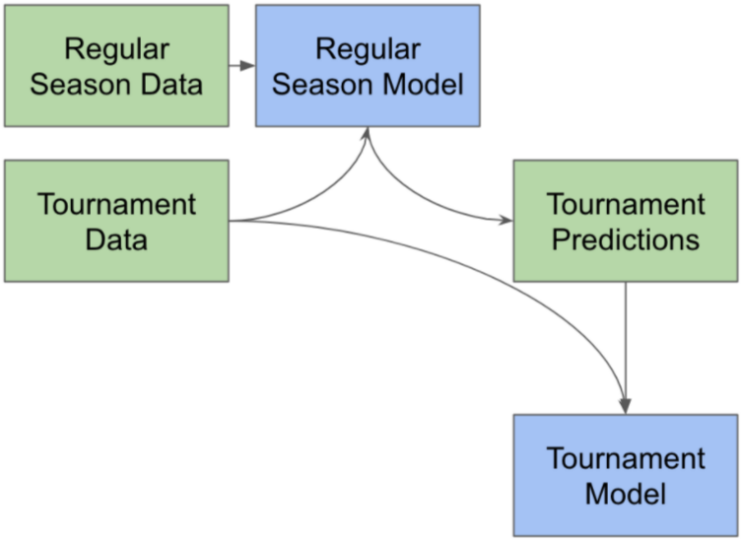
```
games_tourney = read_csv('datasets/games_tourney.csv')
games_tourney.head()
```

	team_1	team_2	home	seed_diff	score_diff
0	288	73	0	-3	-9
1	5929	73	0	4	6
2	9884	73	0	5	-4
3	73	288	0	3	9
4	3920	410	0	1	-9

two datasets:

- the college basketball data from the regular season tournament (over 300,000 rows)
- the college basketball data from the post-season tournament (about 4,000 rows)

Recall that our embedding layer has about 11,000 inputs. So 4,000 rows of data is not enough to learn all 11,000 parameters in our embedding layer. In the previous lesson, you built a three-input model on the regular season data. You can re-use this model to add predictions from the regular season model to the tournament dataset.

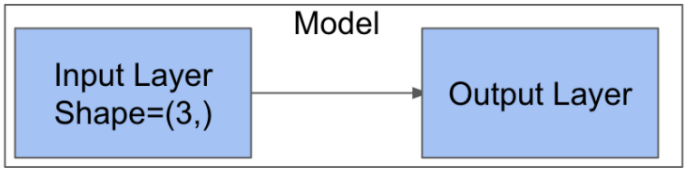


This diagram shows the process for stacking these 2 models. You start with the regular season dataset and fit a model to it. You then predict on the tournament dataset, using this model. This gives you predicted tournament outcomes, which you can now use to build a better model of the actual tournament outcomes. You additionally use the tournament seeds when modeling the tournament. These tournament seeds come from a committee, and are intended to, like your model, capture each team's "strength," without using an embedding layer. The tournament seeds can be thought of as a simplified version of your team strength model, determined by a human rather than a computer.

```
games_tourney[['home','seed_diff','pred']].head()
```

	home	seed_diff	pred
0	0	-3	0.582556
1	0	4	0.707279
2	0	5	1.364844
3	0	3	0.699145
4	0	1	0.833066

The prediction from the regular season model captures the effects of team_1 and team_2, which means you now don't need to use those two variables in the tournament model, and can avoid the use of an embedding layer. You can focus your modeling efforts on the purely numeric data, which is a little easier to work with. With purely numeric inputs, you can pass all of them to a single input layer.



In other words, an input layer with a shape of 3 is another way of defining a 3 input model. The only drawback of this approach is that all the inputs must be numeric.

```
from keras.layers import Input, Dense
in_tensor = Input(shape=(3,))
out_tensor = Dense(1)(in_tensor)
```

```
from keras.models import Model
model = Model(in_tensor, out_tensor)
model.compile(optimizer='adam', loss='mae')
train_X = train_data[['home','seed_diff','pred']]
train_y = train_data['score_diff']
model.fit(train_X,train_y, epochs=10, validation_split=.10)
```

```
test_X = test_data[['home','seed_diff','pred']]
test_y = test_data['score_diff']
model.evaluate(test_X, test_y)
1066/1066 [==============================] - 0s 14us/step
9.11321775461451
```

A huge advantage of this approach is simplicity. You can create a model with a single input tensor and an output tensor, and fit it using a single dataset. Similarly, evaluating the model requires a single dataset, rather than a list. As you can see, this stacked model is pretty accurate! It's off, on average, by about 9 points in a given game.

To recap: stacking keras models means using the predictions from one model as an input to a second model. When stacking, it's important to use different datasets for each model. In this case, you use the regular season data for one model and the tournament dataset for the second model. Finally, if your input dataset is purely numeric, you can put multiple inputs in a single input layer.

Add the model predictions to the tournament data

In lesson 1 of this chapter, you used the regular season model to make predictions on the tournament dataset, and got pretty good results! Try to improve your predictions for the tournament by modeling it specifically.

You'll use the prediction from the regular season model as an input to the tournament model. This is a form of "model stacking."

To start, take the regular season model from the previous lesson, and predict on the tournament data. Add this prediction to the tournament data as a new column.

```
# Predict
games_tourney['pred'] = model.predict([games_tourney['team_1'],
                                       games_tourney['team_2'],
                                       games_tourney['home']])
```

Now you can try building a model for the tournament data based on your regular season predictions.

Create an input layer with multiple columns

In this exercise, you will look at a different way to create models with multiple inputs. This method only works for purely numeric data, but its a much simpler approach to making multi-variate neural networks.

Now you have three numeric columns in the tournament dataset: `'seed_diff'`, `'home'`, and `'pred'`. In this exercise, you will create a neural network that uses a single input layer to process all three of these numeric inputs.

This model should have a single output to predict the tournament game score difference.

```
# Create an input layer with 3 columns
input_tensor = Input((3,))

# Pass it to a Dense layer with 1 unit
output_tensor = Dense(1)(input_tensor)

# Create a model
model = Model(input_tensor, output_tensor)

# Compile the model
model.compile(optimizer='adam', loss='mean_absolute_error')
```

Now your model is ready to meet some data!

Fit the model

Now that you've enriched the tournament dataset and built a model to make use of the new data, fit that model to the tournament data.

Note that this `model` has only one input layer that is capable of handling all 3 inputs, so it's inputs and outputs do not need to be a list.

Tournament games are split into a training set and a test set. The tournament games before 2010 are in the training set, and the ones after 2010 are in the test set.

```
# Fit the model
model.fit(games_tourney_train[['home', 'seed_diff', 'pred']],
          games_tourney_train['score_diff'],
          epochs=1,
          verbose=True)
```

```
<script.py> output:
    Epoch 1/1

     32/3168 [..............................] - ETA: 10s - loss: 21.2439
    864/3168 [=======>......................] - ETA: 0s - loss: 18.3908
   1472/3168 [============>.................] - ETA: 0s - loss: 18.1892
   2144/3168 [===================>..........] - ETA: 0s - loss: 18.1753
   3040/3168 [===========================>..] - ETA: 0s - loss: 17.8417
   3168/3168 [==============================] - 0s 100us/step - loss: 17.8147
```

In the next exercise, you'll see if our model is any good!

Evaluate the model

Now that you've fit your model to the tournament training data, evaluate it on the tournament test data. Recall that the tournament test data contains games from after 2010.

```
# Evaluate the model on the games_tourney_test dataset
print(model.evaluate(games_tourney_test[['home', 'seed_diff', 'prediction']],
                     games_tourney_test['score_diff'], verbose=False))
```

```
<script.py> output:
    9.07315489498804
```

Your model works pretty well on data in the future!

Chapter 4. Multiple Outputs

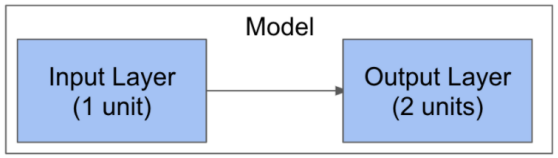
In this chapter, you will build neural networks with multiple outputs, which can be used to solve regression problems with multiple targets. You will also build a model that solves a regression problem and a classification problem simultaneously.

Two-output models

Make prediction for 2 targets at once, eg. use 1 model to predict scores of both teams in a basketball match.

```
from keras.layers import Input, Concatenate, Dense
input_tensor = Input(shape=(1,))
output_tensor = Dense(2)(input_tensor)
```

```
from keras.models import Model
model = Model(input_tensor, output_tensor)
model.compile(optimizer='adam', loss='mean_absolute_error')
```




```
games_tourney_train[['seed_diff', 'score_1', 'score_2']].head()
```

	seed_diff	score_1	score_2
0	-3	41	50
1	4	61	55
2	5	59	63
3	3	50	41
4	1	54	63

```
X = games_tourney_train[['seed_diff']]
y = games_tourney_train[['score_1', 'score_2']]
model.fit(X, y, epochs=500)
```

Now that the model is fit, you can take a look at what it learned. The dense layer has two weights and two biases.

```
model.get_weights()
```

```
[array([[ 0.60714734, -0.5988793 ]], dtype=float32),
 array([70.39491, 70.39306], dtype=float32)]
```

The weights indicate that each additional unit of seed difference for the input data equals about .60 additional points for team 1 (and .60 fewer points for team 2). The bias, or intercept term for each team is about 70 points, indicating that we expect an average basketball team to score about 70 points in an average game. In other words, 2 teams with a 1 point seed difference would be expected to have a score of about 69 to 71, while 2 teams with a 10 point seed difference would be expected to have a score of about 64 to 76.

```
X = games_tourney_test[['seed_diff']]
y = games_tourney_test[['score_1', 'score_2']]
model.evaluate(X, y)
```

```
11.528035634635021
```

Evaluating a model with two outputs is very similar to evaluating a model with 1 output, except you provide the evaluation function a dataset with 2 columns of data for the target.

Simple two-output model

In this exercise, you will use the tournament data to build one model that makes two predictions: the scores of both teams in a given game. Your inputs will be the seed difference of the two teams, as well as the predicted score difference from the model you built in chapter 3.

The output from your model will be the predicted score for team 1 as well as team 2. This is called "multiple target regression": one model making more than one prediction.

```
# Define the input
input_tensor = Input(shape=(2,))

# Define the output
output_tensor = Dense(2)(input_tensor)
```

```
# Create a model
model = Model(input_tensor, output_tensor)

# Compile the model
model.compile(loss='mean_absolute_error', optimizer='adam')
```

Now you have a multiple output model!

Fit a model with two outputs

Now that you've defined your 2-output model, fit it to the tournament data. I've split the data into `games_tourney_train` and `games_tourney_test`, so use the training set to fit for now.

This model will use the pre-tournament seeds, as well as your pre-tournament predictions from the regular season model you built previously in this course.

As a reminder, this model will predict the scores of both teams.

```
model.fit(games_tourney_train[['seed_diff', 'pred']],
          games_tourney_train[['score_1', 'score_2']],
          verbose=True,
          epochs=100,
          batch_size=16384)
```

```
<script.py> output:
    Epoch 1/100

    3430/3430 [==============================] - 0s 37us/step - loss: 71.6213
    Epoch 2/100

    3430/3430 [==============================] - 0s 1us/step - loss: 70.6213
    Epoch 3/100

    3430/3430 [==============================] - 0s 1us/step - loss: 69.6213
    Epoch 4/100
```

```
    3430/3430 [==============================] - 0s 1us/step - loss: 9.7397
    Epoch 98/100

    3430/3430 [==============================] - 0s 1us/step - loss: 9.7300
    Epoch 99/100

    3430/3430 [==============================] - 0s 1us/step - loss: 9.7209
    Epoch 100/100

    3430/3430 [==============================] - 0s 1us/step - loss: 9.7136
```

Let's look at the model weights.

Inspect the model (I)

Now that you've fit your model, let's take a look at it. You can use the `.get_weights()` method to inspect your model's weights.

The input layer will have 4 weights: 2 for each input times 2 for each output.

The output layer will have 2 weights, one for each output.

```
# Print the model's weights
print(model.get_weights())

# Print the column means of the training data
print(games_tourney_train.mean())
```

```
<script.py> output:
    [array([[ 0.13067713, -0.10371894],
           [ 0.38644195, -0.35632333]], dtype=float32), array([72.38115, 72.38473], dtype=float32)]
    season        1.998074e+03
    team_1        5.556771e+03
    team_2        5.556771e+03
    home          0.000000e+00
    seed_diff     0.000000e+00
    score_diff    0.000000e+00
    score_1       7.162128e+01
    score_2       7.162128e+01
    won           5.000000e-01
    pred         -1.625470e-14
    dtype: float64
```

Did you notice that both output weights are about ~72? This is because, on average, a team will score about 72 points in the tournament.

Evaluate the model

Now that you've fit your model and inspected it's weights to make sure it makes sense, evaluate it on the tournament test set to see how well it performs on new data.

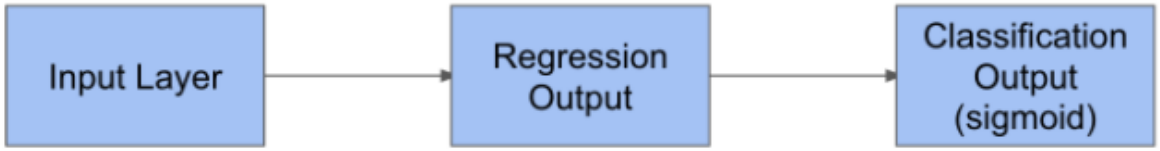
```
print(model.evaluate(games_tourney_test[['seed_diff', 'pred']],
                     games_tourney_test[['score_1', 'score_2']], verbose=False))
```

```
<script.py> output:
    8.986760239102948
```

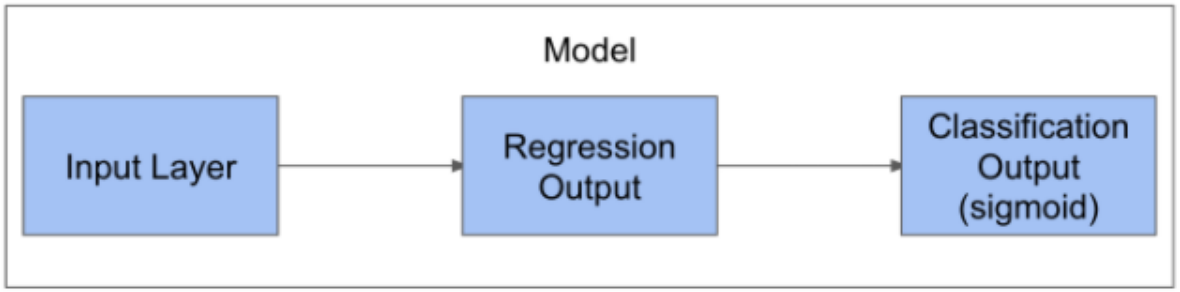
Loss metrics (mae) is quite low. This model is pretty accurate at predicting tournament scores!

Single model for classification and regression

```
from keras.layers import Input, Dense
input_tensor = Input(shape=(1,))
output_tensor_reg = Dense(1)(input_tensor)
output_tensor_class = Dense(1, activation='sigmoid')(output_tensor_reg)
```

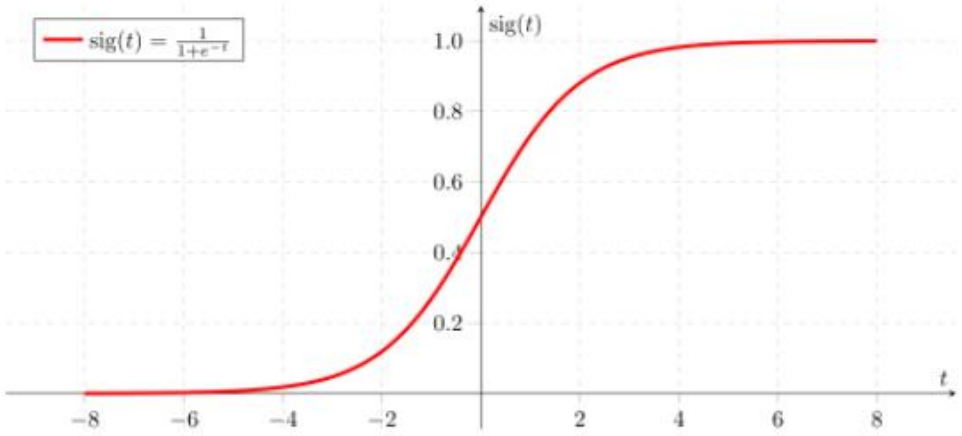



```
from keras.models import Model
model = Model(input_tensor, [output_tensor_reg, output_tensor_class])
model.compile(loss=['mean_absolute_error', 'binary_crossentropy'],
              optimizer='adam')
```




```
X = games_tourney_train[['seed_diff']]
y_reg = games_tourney_train[['score_diff']]
y_class = games_tourney_train[['won']]
model.fit(X, [y_reg, y_class], epochs=100)
```

```
model.get_weights()
[array([[1.2371823]], dtype=float32),
 array([-0.05451894], dtype=float32),
 array([[0.13870609]], dtype=float32),
 array([0.00734114], dtype=float32)]
```



This model's weight structure is a bit different from the last model, where both outputs were regression targets. The first layer has a weight of 1.24 and a bias of almost zero. This means that a 1 unit change in the teams' seed difference yields about 1.24 additional points in their score difference. So 2 teams with a seed difference of 1 would be expected to have team 1 win by 1.2 points. But 2 teams with a seed difference of 10 would be expected to have team 1 win by 12 points. The next layer maps predicted score difference to predicted win/loss. Recall that the final layer in the model uses sigmoid activation.

```
model.get_weights()
[array([[1.2371823]], dtype=float32),
 array([-0.05451894], dtype=float32),
 array([[0.13870609]], dtype=float32),
 array([0.00734114], dtype=float32)]
```

```
from scipy.special import expit as sigmoid
print(sigmoid(1 * 0.13870609 + 0.00734114))
```

```
0.5364470465211318
```

Let's manually calculate the win probability for 2 teams that are predicted to have a score difference of 1. First, multiply 1 by the weight for the final layer in the model: 0.14. Add the bias for the final layer: 0.0007. Since the bias is very close to zero, the result is still 0.14. Finally, we apply the sigmoid function to 0.14, which yields a prediction of 0.54. In other words, the model has learned that an expected score difference of 1 point is equal to an expected win probability of 54%.

```
X = games_tourney_test[['seed_diff']]
y_reg = games_tourney_test[['score_diff']]
y_class = games_tourney_test[['won']]
model.evaluate(X, [y_reg, y_class])
```

```
[9.866300069455413, 9.281179495657208, 0.585120575627864]
```

Finally, you can evaluate the model on new data. First, split the evaluation dataset into a regression target and a classification target, and provide the same list of 2 targets to the evaluate() method. This outputs 3 numbers now, instead of 1 as with the models we looked at in other chapters. The first number is the loss function used by the model, which is the sum of all the output losses. The second number is the loss for the regression part of the model, and the third number is the loss for the classification part of the model. So our model has a mean absolute error of 9.28 and a logloss of 0.58, which is pretty good

Classification and regression in one model

Now you will create a different kind of 2-output model. This time, you will predict the score difference, instead of both team's scores and then you will predict the probability that team 1 won the game. This is a pretty cool model: it is going to do both classification and regression!

In this model, turn off the bias, or intercept for each layer. Your inputs (seed difference and predicted score difference) have a mean of very close to zero, and your outputs both have means that are close to zero, so your model shouldn't need the bias term to fit the data well.

```
# Create an input layer with 2 columns
input_tensor = Input(shape=(2,))

# Create the first output
output_tensor_1 = Dense(1, activation='linear', use_bias=False)(input_tensor)

# Create the second output (use the first output as input here)
output_tensor_2 = Dense(1, activation='sigmoid', use_bias=False)(output_tensor_1)

# Create a model with 2 outputs
model = Model(input_tensor, [output_tensor_1, output_tensor_2])
```

This kind of model is only possible with a neural network.

Compile and fit the model

Now that you have a model with 2 outputs, compile it with 2 loss functions: mean absolute error (MAE) for `'score_diff'` and binary cross-entropy (also known as logloss) for `'won'`. Then fit the model with `'seed_diff'` and `'pred'` as inputs. For outputs, predict `'score_diff'` and `'won'`.

This model can use the scores of the games to make sure that close games (small score diff) have lower win probabilities than blowouts (large score diff).

The regression problem is easier than the classification problem because MAE punishes the model less for a loss due to random chance. For example, if `score_diff` is -1 and `won` is 0, that means `team_1` had some bad luck and lost by a single free throw. The data for the easy problem helps the model find a solution to the hard problem.

```
# Import the Adam optimizer
from keras.optimizers import Adam

# Compile the model with 2 losses and the Adam optimzer with a higher learning rate
model.compile(loss=['mean_absolute_error', 'binary_crossentropy'], optimizer=Adam(.01))

# Fit the model to the tournament training data, with 2 inputs and 2 outputs
model.fit(games_tourney_train[['seed_diff', 'pred']],
          [games_tourney_train[['score_diff']], games_tourney_train[['won']]],
          epochs=10,
          verbose=True,
          batch_size=16384)
```

```
<script.py> output:
    Epoch 1/10

    3430/3430 [==============================] - 0s 40us/step - loss: 9.5718 - dense_1_loss: 8.9779 - dense_2_loss: 0.5939
    Epoch 2/10

    3430/3430 [==============================] - 0s 1us/step - loss: 9.5437 - dense_1_loss: 8.9614 - dense_2_loss: 0.5823
    Epoch 3/10
    3430/3430 [==============================] - 0s 2us/step - loss: 9.4618 - dense_1_loss: 8.9208 - dense_2_loss: 0.5408
    Epoch 9/10

    3430/3430 [==============================] - 0s 1us/step - loss: 9.4585 - dense_1_loss: 8.9201 - dense_2_loss: 0.5385
    Epoch 10/10

    3430/3430 [==============================] - 0s 2us/step - loss: 9.4562 - dense_1_loss: 8.9190 - dense_2_loss: 0.5372
```

You just fit a model that is both a classifier and a regressor!

Inspect the model (II)

Now you should take a look at the weights for this model. In particular, note the last weight of the model. This weight converts the predicted score difference to a predicted win probability. If you multiply the predicted score difference by the last weight of the model and then apply the sigmoid function, you get the win probability of the game.

```
# Print the model weights
print(model.get_weights())

# Print the training data means
print(games_tourney_train.mean())
```

```
<script.py> output:
    [array([[0.9695341 ],
           [0.22126554]], dtype=float32), array([[0.1428871]], dtype=float32)]
    season        1.998074e+03
    team_1        5.556771e+03
    team_2        5.556771e+03
    home          0.000000e+00
    seed_diff     0.000000e+00
    score_diff    0.000000e+00
    score_1       7.162128e+01
    score_2       7.162128e+01
    won           5.000000e-01
    pred         -1.625470e-14
    dtype: float64
```

```
# Import the sigmoid function from scipy
from scipy.special import expit as sigmoid

# Weight from the model
weight = 0.14

# Print the approximate win probability predicted close game
print(sigmoid(1 * weight))

# Print the approximate win probability predicted blowout game
print(sigmoid(10 * weight))
```

```
<script.py> output:
    0.5349429451582145
    0.8021838885585818
```

So sigmoid(1 * 0.14) is 0.53, which represents a pretty close game and sigmoid(10 * 0.14) is 0.80, which represents a pretty likely win. In other words, if the model predicts a win of 1 point, it is less sure of the win than if it predicts 10 points. Who says neural networks are black boxes?

Evaluate on new data with two metrics

Now that you've fit your model and inspected its weights to make sure they make sense, evaluate your model on the tournament test set to see how well it does on new data.

Note that in this case, Keras will return 3 numbers: the first number will be the sum of both the loss functions, and then the next 2 numbers will be the loss functions you used when defining the model.

Ready to take your deep learning to the next level? Check out **"Convolutional Neural Networks for Image Processing"**.

```
# Evaluate the model on new data
print(model.evaluate(games_tourney_test[['seed_diff', 'pred']],
                     [games_tourney_test[['score_diff']], games_tourney_test[['won']]], verbose=False))
```

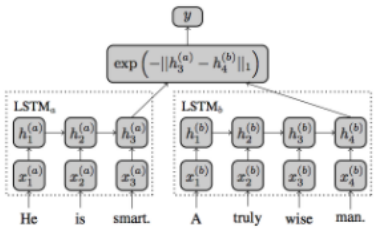
```
<script.py> output:
    [9.685116468970456, 9.11585681592647, 0.5692595753503676]
```

Turns out you can have your cake and eat it too! This model is both a good regressor and a good classifier!

Shared layers

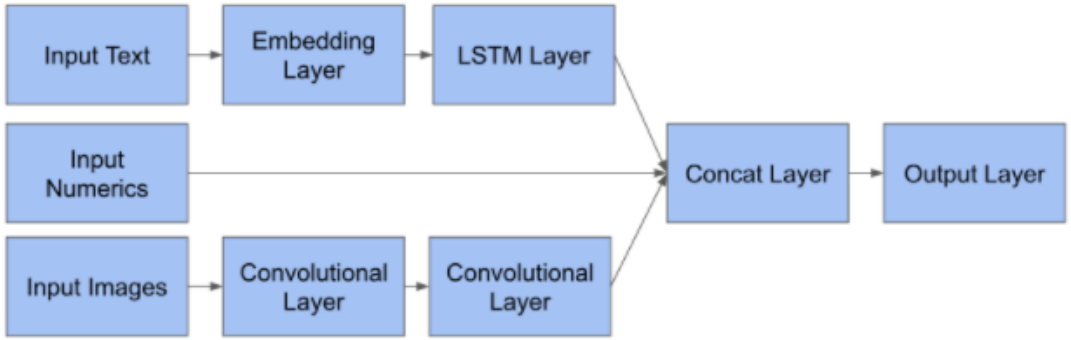
Useful for making comparisons

- Basketball teams
- Image similarity / retrieval
- Document similarity



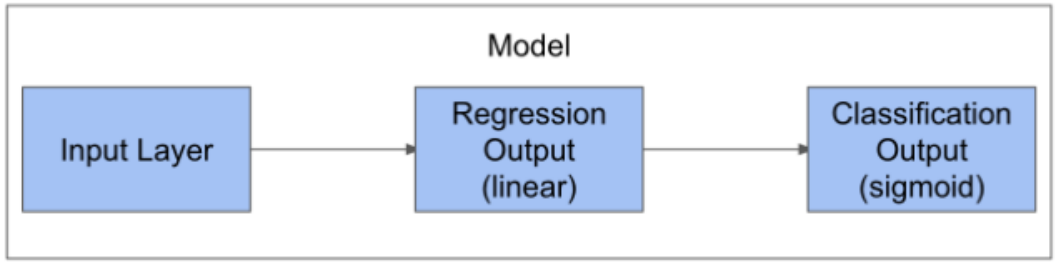
In academic research, shared models are known as "siamese networks", which are used to calculate things like document similarity, using shared embedding layer and a shared long-short-term memory layer (or LSTM layer), and then comparing the LSTM outputs. Since both documents are encoded with the same embedding layer and the same LSTM layer, the model learns a representation of the documents that can be used to compare them.

Multiple inputs



Multiple input networks are especially useful when you want to process different types of data within your model. For example, in our basketball models, we processed team IDs separately, using an embedding layer. For numeric data such as home vs away, we skipped the embedding step and passed it directly to the output. You can extend this concept to build a network that, for example, uses an LSTM to process text, a standard Dense layer to process numerics, and a convolutional layer (or CNN) to process images.

Multiple outputs

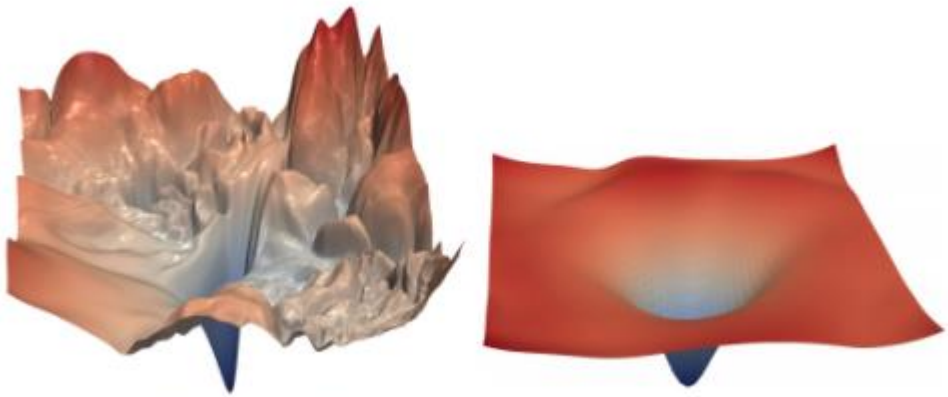


The multiple output network can do both classification AND regression! The regression problem turns out to be a lot easier than the classification problem. In the regression problem, the neural network gets penalized less for random chance. For example, a team that wins by 1 point got really lucky, and the model can use that information. However, in the classification model, winning by 1 point is the same as winning by 10 points. It's pretty cool that we can use the output from the easier regression problem to help solve the more difficult classification problem.

Skip connections

```
input_tensor = Input((100,))
hidden_tensor = Dense(256, activation='relu')(input_tensor)
hidden_tensor = Dense(256, activation='relu')(hidden_tensor)
hidden_tensor = Dense(256, activation='relu')(hidden_tensor)
output_tensor = Concatenate()([input_tensor, hidden_tensor])
output_tensor = Dense(256, activation='relu')(output_tensor)
```

Visualizing the Loss Landscape of Neural Nets



(a) without skip connections (b) with skip connections

In a paper called "Visualizing the Loss Landscape of Neural Nets", Li et al. propose a method called "skip connections" to simplify the optimization of neural networks. To summarize a very long and interesting paper, skip connections make it a lot easier for the adam optimizer to find the global minimum of the network's loss function. In Keras, implementing a skip connection is as simple as using the "Concatenate" layer to concatenate the inputs to the deep network's outputs, right before the final output layer.

Course completed!

Recap topics covered:

- Functional keras models (API)
- Shared layers
- Categorical embeddings
- Multiple inputs
- Multiple outputs
- Regression / Classification in one model

Happy learning!