Lec.6. Creating a keras model

Machine Learning II

Aidos Sarsembayev, IITU, Almaty, 2019

Outline

1. Model building steps
2. Classification models
3. Using models

Model building steps

- Specify Architecture
- Compile
- Fit
- Predict

Model specification

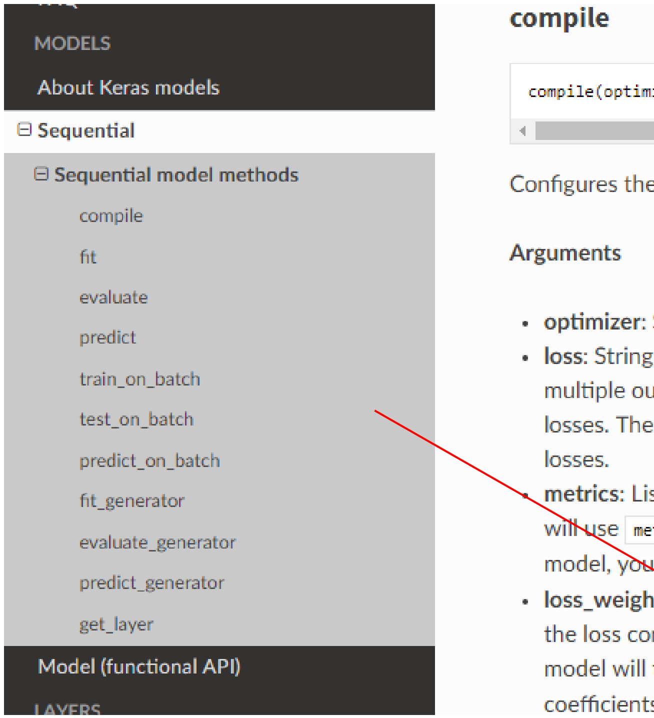
- Specify Architecture

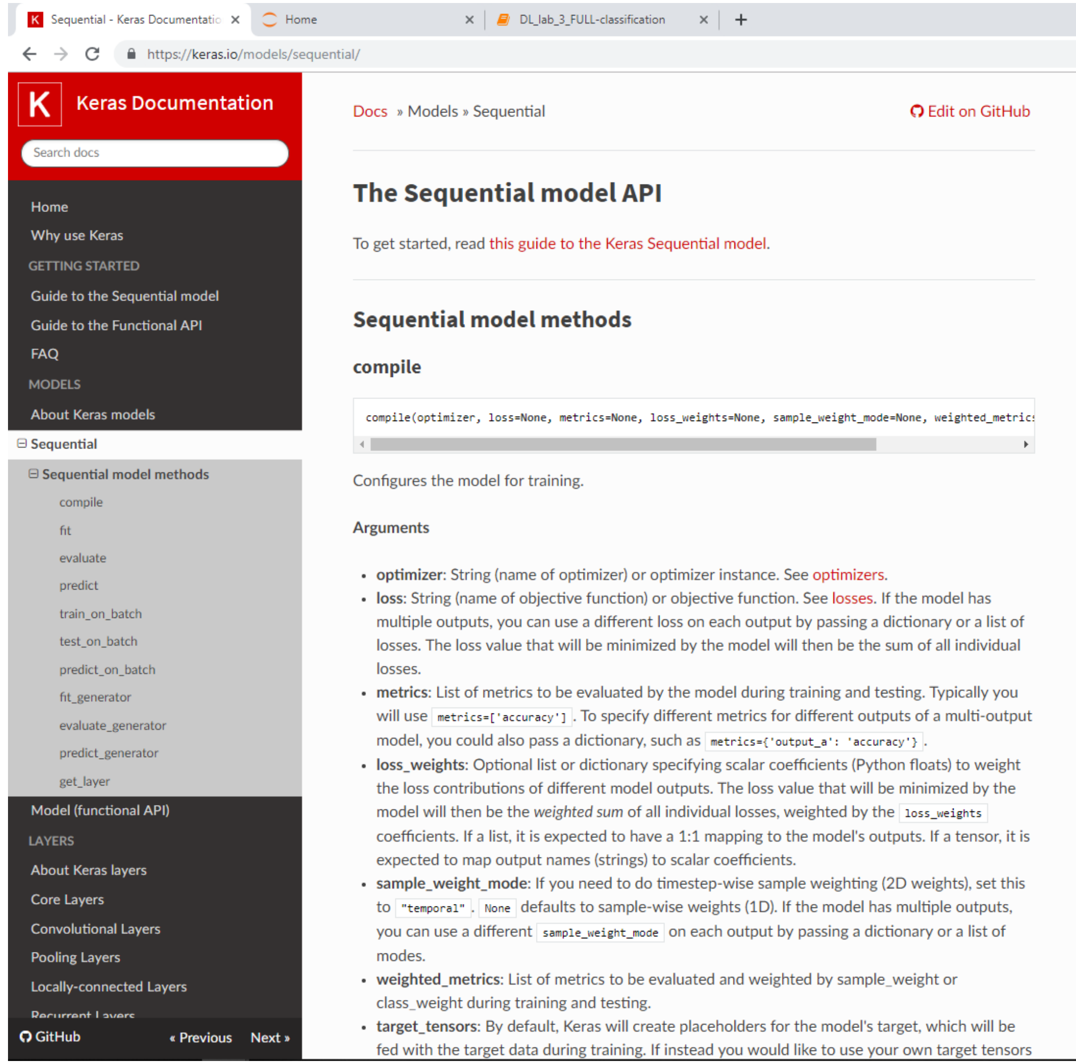
```
In [1]: import numpy as np

In [2]: from keras.layers import Dense

In [3]: from keras.models import Sequential
```

```
In [4]: predictors = np.loadtxt('predictors_data.csv', delimiter=',')

In [5]: n_cols = predictors.shape[1]
```

```
In [6]: model = Sequential()

In [7]: model.add(Dense(100, activation='relu', input_shape = (n_cols,)))

In [8]: model.add(Dense(100, activation='relu'))

In [9]: model.add(Dense(1))
```

Keras documentation





Compiling and fitting a model

Why you need to compile your model

- Specify the optimizer
  - Many options and mathematically complex
  - “Adam” is usually a good choice
- Loss function
  - “mean_squared_error” common for regression

Dense layer type

keras.layers.Dense(units, activation=**None**, use_bias=**True**, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=**None**, bias_regularizer=**None**, activity_regularizer=**None**, kernel_constraint=**None**, bias_constraint=**None**)

Just your regular densely-connected NN layer.

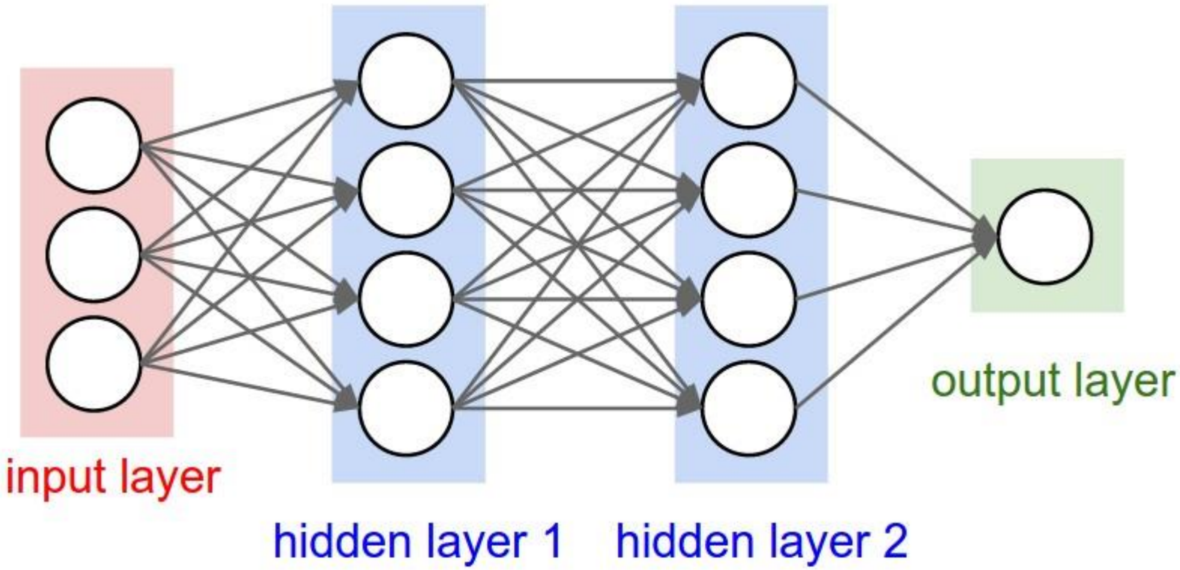
`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`).

this is nothing but: *relu(X.w + b)*

- Also referred to as **Fully-connected NN**

Dense layer type

Also referred to as **Fully-connected NN**



Activation functions in Keras

Activation layers:

- **softmax**
- **elu**
- **selu**
- **softplus**
- **softsign**
- **relu**
- **tanh**
- **sigmoid**
- **hard_sigmoid**
- **exponential**
- **linear**

Advanced Activations Layers:

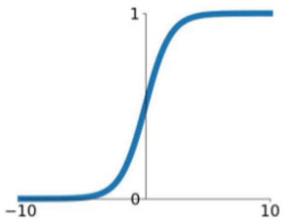
- **LeakyReLU**
- **PReLU**
- **ELU**
- **ThresholdedReLU**
- **Softmax**
- **ReLU**

Activation functions in Keras

Activation Functions

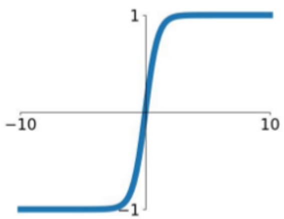
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



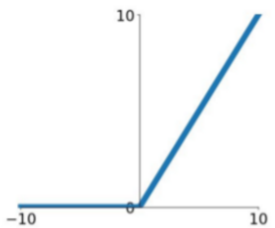
tanh

$\tanh(x)$



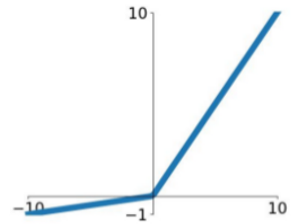
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

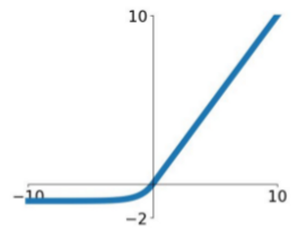


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



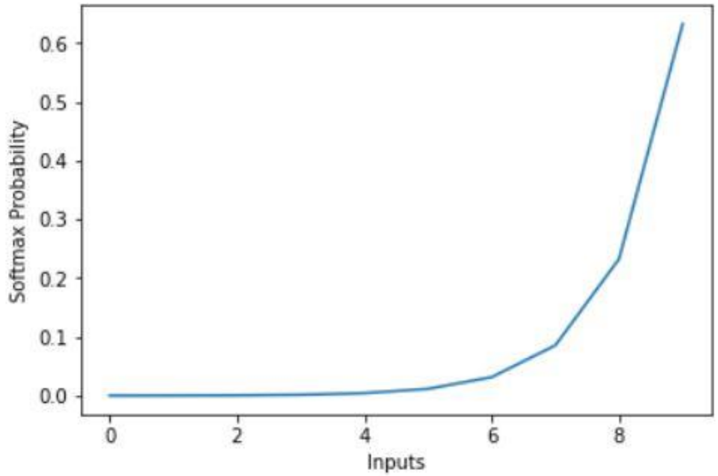
https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c

Softmax

Activation layers:

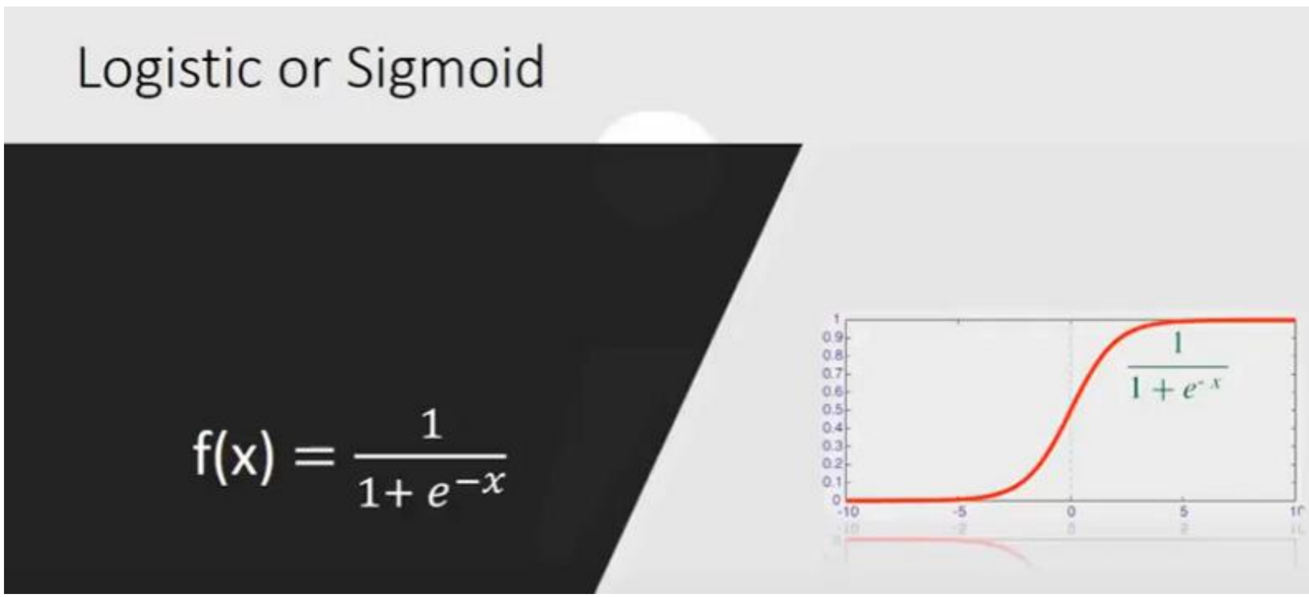
- **Softmax**
  → function is used to impart probabilities when you have more than one outputs you get probability distribution of outputs.
- →Useful for finding most probable occurrence of output with respect to other outputs.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$



Logistic or Sigmoid

- **Logistic or Sigmoid**
  → Maps any sized inputs to outputs in range [0,1].



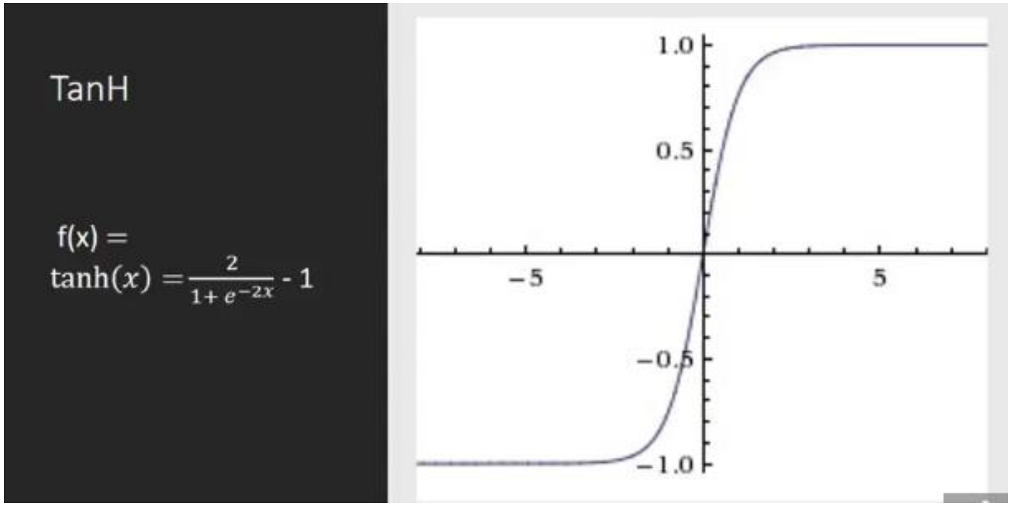
- https://towardsdatascience.com/activation-functions-in-neural-networks-58115cda9c96

Tanh

- **Tanh**

→Maps input to output ranging in [-1,1].

→Similar to sigmoid function except it maps output in [-1,1] whereas sigmoid maps output to [0,1].

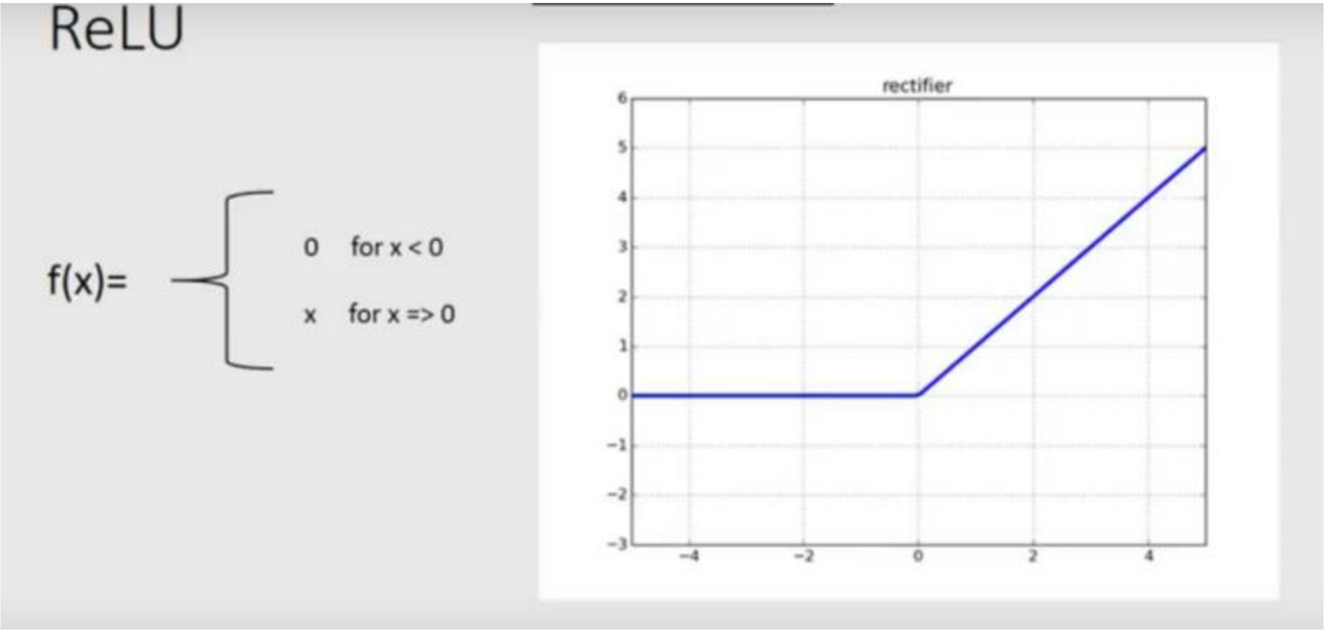


- https://towardsdatascience.com/activation-functions-in-neural-networks-58115cda9c96

Rectified Linear Unit (ReLu)

- **Rectified Linear Unit (ReLu)**

→ It removes negative part of function.

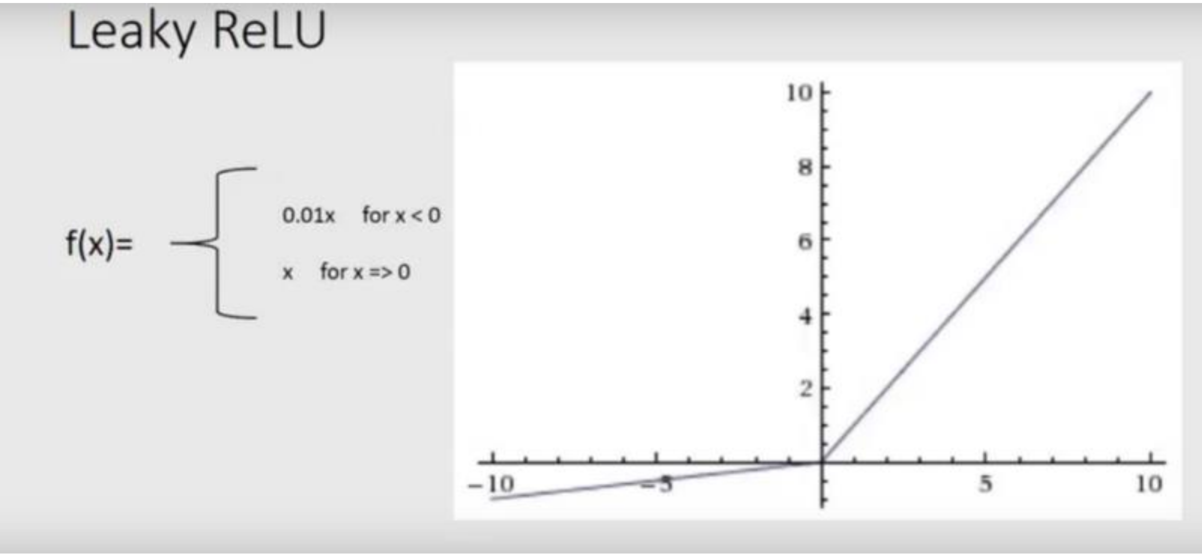


- https://towardsdatascience.com/activation-functions-in-neural-networks-58115cda9c96

Leaky ReLu

- **Leaky ReLu**

→ The only difference between **ReLu** and **Leaky ReLu** is it does not completely vanishes the negative part,it just lower its magnitude.



- https://towardsdatascience.com/activation-functions-in-neural-networks-58115cda9c96

Compiling a model

```
In [1]: n_cols = predictors.shape[1]

In [2]: model = Sequential()

In [3]: model.add(Dense(100, activation='relu', input_shape=(n_cols,)))

In [4]: model.add(Dense(100, activation='relu'))

In [5]: model.add(Dense(1))

In [6]: model.compile(optimizer='adam', loss='mean_squared_error')
```

Compiling parameters

The two mandatory parameters for compiling the model are:

- Optimizer
- Loss

Optimizers

- SGD - Stochastic gradient descent
- RMSprop
- Adagrad
- Adadelta
- **Adam**
- Adamax
- Nadam

Loss functions

- **mean_squared_error**
- mean_absolute_error
- **categorical_crossentropy**
- binary_crossentropy

What is fitting a model

- Applying backpropagation and gradient descent with your data to update the weights
- Scaling data before fitting can ease optimization

Fitting a model

```
In [1]: n_cols = predictors.shape[1]

In [2]: model = Sequential()

In [3]: model.add(Dense(100, activation='relu', input_shape=(n_cols,)))

In [4]: model.add(Dense(100, activation='relu'))

In [5]: model.add(Dense(1))

In [6]: model.compile(optimizer='adam', loss='mean_squared_error')

In [7]: model.fit(predictors, target)
```

Classification models

- ‘categorical_crossentropy’ loss function
- Similar to log loss: Lower is better
- Add metrics = [‘accuracy’] to compile step for easy-to-understand diagnostics
- Output layer has separate node for each possible outcome, and uses ‘softmax’ activation

Quick look at the data

shot_clock	dribbles	touch_time	shot_dis	close_def_dis	shot_result
10.8	2	1.9	7.7	1.3	1
3.4	0	0.8	28.2	6.1	0
0	3	2.7	10.1	0.9	0
10.3	2	1.9	17.2	3.4	0

Transforming to categorical

shot_result
1
0
0
0

Outcome 0	Outcome 1
0	1
1	0
1	0
1	0

Classification

```
In[1]: from keras.utils import to_categorical

In[2]: data = pd.read_csv('basketball_shot_log.csv')

In[3]: predictors = data.drop(['shot_result'], axis=1).as_matrix()

In[4]: target = to_categorical(data.shot_result)

In[5]: model = Sequential()

In[6]: model.add(Dense(100, activation='relu', input_shape = (n_cols,)))

In[7]: model.add(Dense(100, activation='relu'))

In[8]: model.add(Dense(100, activation='relu'))

In[9]: model.add(Dense(2, activation='softmax'))

In[10]: model.compile(optimizer='adam', loss='categorical_crossentropy',
   ...:               metrics=['accuracy'])

In[11]: model.fit(predictors, target)
```

Classification

```
Out[11]:
Epoch 1/10
128069/128069 [==============================] - 4s - loss: 0.7706 - acc: 0.5759
Epoch 2/10
128069/128069 [==============================] - 5s - loss: 0.6656 - acc: 0.6003
Epoch 3/10
128069/128069 [==============================] - 6s - loss: 0.6611 - acc: 0.6094
Epoch 4/10
128069/128069 [==============================] - 7s - loss: 0.6584 - acc: 0.6106
Epoch 5/10
128069/128069 [==============================] - 7s - loss: 0.6561 - acc: 0.6150
Epoch 6/10
128069/128069 [==============================] - 9s - loss: 0.6553 - acc: 0.6158
Epoch 7/10
128069/128069 [==============================] - 9s - loss: 0.6543 - acc: 0.6162
Epoch 8/10
128069/128069 [==============================] - 9s - loss: 0.6538 - acc: 0.6158
Epoch 9/10
128069/128069 [==============================] - 10s - loss: 0.6535 - acc: 0.6157
Epoch 10/10
128069/128069 [==============================] - 10s - loss: 0.6531 - acc: 0.6166
```

Using models

- Save
- Reload
- Make predictions

Saving, reloading and using your Model

```
In [1]: from keras.models import load_model

In [2]: model.save('model_file.h5')

In [3]: my_model = load_model('my_model.h5')

In [4]: predictions = my_model.predict(data_to_predict_with)

In [5]: probability_true = predictions[:,1]
```

Verifying model structure

```
In [6]: my_model.summary()
Out[6]:
____________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
============================================================================================
dense_1 (Dense)                  (None, 100)           1100        dense_input_1[0][0]
____________________________________________________________________________________________
dense_2 (Dense)                  (None, 100)           10100       dense_1[0][0]
____________________________________________________________________________________________
dense_3 (Dense)                  (None, 100)           10100       dense_2[0][0]
____________________________________________________________________________________________
dense_4 (Dense)                  (None, 2)             202         dense_3[0][0]
============================================================================================
Total params: 21,502
Trainable params: 21,502
Non-trainable params: 0
```

To be continued,

Thanks!