Neural Network with Keras

Penyusun Modul: Chairul Aulia

Editor: Citra Chairunnisa

Tensorflow

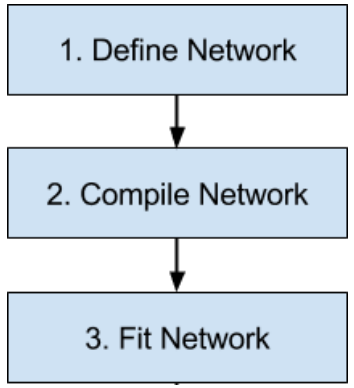
- TensorFlow as one of the greatest gifts to the machine learning community by Google is a **free and open-source** framework for **numerical computing** and solving **complex machine learning** problems.
- TensorFlow works with **tensors** which is a multidimensional array (or ndarray in Numpy terms).
- It was written in Python, C++, and CUDA, and is a Python-friendly open-source library.
- Best used when you have a need for:
  - Deep learning research
  - Complex neural networks
  - Working with large datasets
  - High performance models

Keras

- Keras is a high-level open-source library that runs on top of TensorFlow
- Keras simplifies the implementation of complex neural networks with its easy to use framework.
- When you use Keras, you're really using the TensorFlow library. But you're using the TensorFlow library in a way that's more **intuitive, visual, and modular**.
- Keras makes it easier to use the TensorFlow library, but with some trade-offs. Specifically, you may not be able to access more complex (and confusing) low-level utilities.

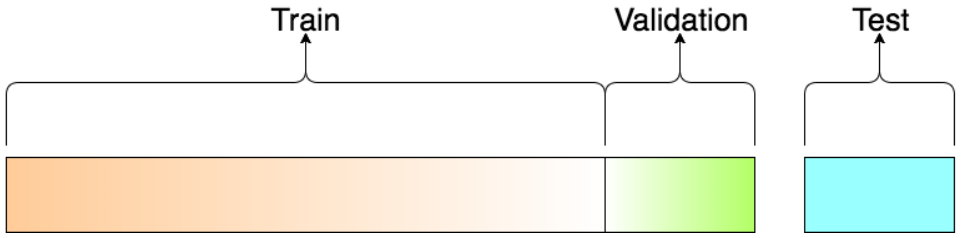
Getting started

- At this point you probably already have pandas, scikit-learn, numpy and matplotlib installed, so before we continue make sure that you have the next 2 packages we need (Keras and Tensorflow)
- The process of data preparation is just the same with what we've done so far in classical machine learning problems (that's why we will still be using sklearn)
- Building a neural network model in Keras comprises of these steps



Validation Dataset

Splitting dataset into training and testing can be done with `sklearn.preprocessing.train_test_split()`
Now let me introduce to you a new term, *validation set*



- Training set: The actual dataset that we use to train the model. The model sees and learns from this data.
- Validation set: The model occasionally sees this data, but never "Learn" from this. We use the results to evaluate a given model by fine-tuning the hyperparameters. So, the validation set affects a model, but only indirectly.
- Testing set: It is only used once a model is completely trained. It provides the gold standard to evaluate the model

Below is the code for splitting data into training, validation and testing sets with 70:15:15 ratio

```
from sklearn.model_selection import train_test_split
X_train, X_val_and_test, Y_train, Y_val_and_test = train_test_split(X, Y, test_size=0.3)
X_val, X_test, Y_val, Y_test = train_test_split(X_val_and_test, Y_val_and_test, test_size=0.5)
```

Defining Network

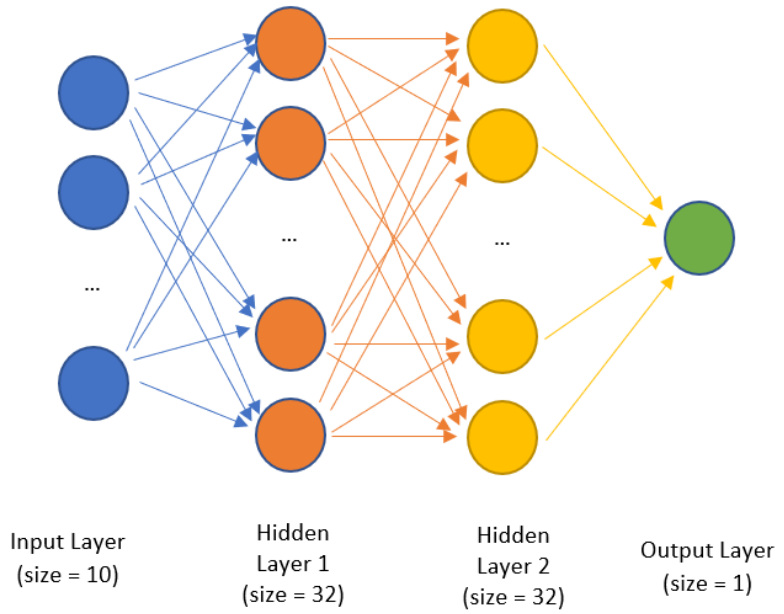
The first thing we have to do is to set up the architecture. Let's first think about what kind of neural network architecture we want.

Suppose we want this simple neural network:

- Hidden layer 1: 32 neurons, ReLU activation
- Hidden layer 2: 32 neurons, ReLU activation
- Output Layer: 1 neuron, Sigmoid activation

```
from keras.models import Sequential
from keras.layers import Dense
```

```
model = Sequential([
    Dense(32, activation='relu', input_shape=(10,)),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid'),
])
```



Compiling the network

Before we start our training, we have to configure the model by

- Telling it which algorithm you want to use to do the optimization
- Telling it what loss function to use
- Telling it what other metrics you want to track apart from the loss function

Configuring the model with these settings requires us to call the function `model.compile`, like this:

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

'sgd' refers to stochastic gradient descent
binary cross entropy is the loss function for outputs that take the values 1 or 0

For more detail documentation please visit
https://keras.io/api/optimizers/

Fitting the network

Training on the data is pretty straightforward and requires us to write one line of code:

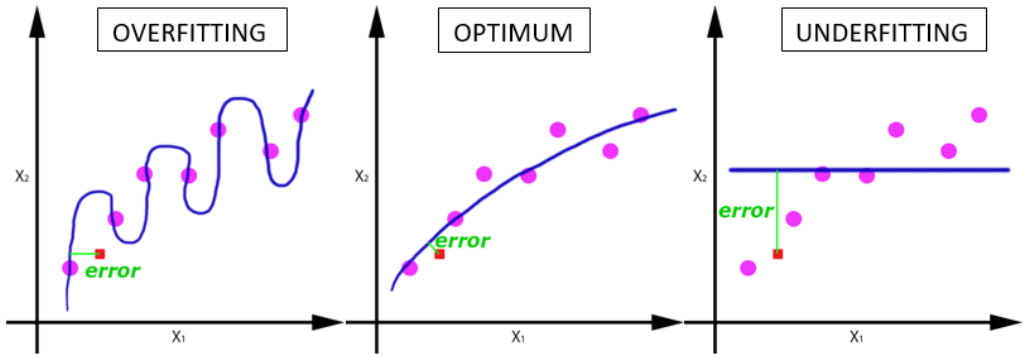
```
model.fit(X_train, Y_train,
   batch_size=32, epochs=100,
   validation_data=(X_val, Y_val))
```

We need terminologies like epochs, batch size, iterations only when the data is too big which happens all the time in machine learning and we can't pass all the data to the computer at once. So, to overcome this problem we need to divide the data into smaller sizes and give it to our computer one by one and update the weights of the neural networks at the end of every step to fit it to the data given

Epochs

One Epoch is when an entire dataset is passed forward and backward through the neural network only once. Keep in mind that we optimize the learning using **Gradient Descent** which is an ***iterative*** process. So, *updating the weights with single pass or one epoch is not enough.*

As the number of epochs increases, more number of times the weight are changed in the neural network and the curve goes from **underfitting** to **optimal** to **overfitting** curve.



So, what is the right numbers of epochs?
Unfortunately, there is no right answer to this question. The answer is different for different datasets but you can say that the numbers of epochs is related to how diverse your data is

Batch Size

Batch size: Total number of training examples present in a single batch.
Since we can't pass the entire dataset into the neural net at once. So, we divide dataset into number of batches or sets or parts.
Just like a book divided into multiple sets/batches/parts like chapter I, chapter II, chapter III and so on.

Let's say we have 2000 training examples that we are going to use.
We can divide the dataset of 2000 examples into batches of 500 then it will take 4 iterations to complete 1 epoch.
(Batch size and number of batches are two different things. Number of batches equals to iterations)

Visualizing Loss and Accuracy

What can we infer from the visualization of training and validation loss and accuracy?

If our model has fit so extremely to the training data that it fails to generalize to other examples, we will see low training loss but high validation loss or much higher training accuracy than validation accuracy

This is what we called overfitting.

Strategy to Tackle Overfitting

- L2 Regularization. The problem with over-fitted models is that we often have some wild crazy model that works well only for the training examples. If we penalize wild crazy models (by penalizing extremely large parameters), we can reduce over-fitting. How can we introduce this penalty? Simple — add the penalty to the loss function! The penalty is the squared value of the parameters (scaled by some constant number), so the larger the parameters, the higher the loss will be.

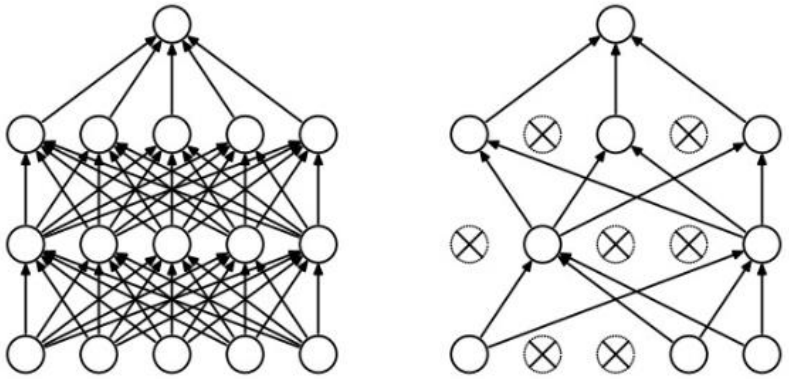
$$\text{New Loss} = \text{Training Loss} + \lambda ||w||^2$$

Penalty for wild models: square of parameters, scaled by a constant λ

New loss when we add L2-regularization that penalizes overly large parameters (i.e. wild and crazy models)

Strategy to Tackle Overfitting

- Dropout. Dropout works as follows: during the training of our model, we randomly cause some of the neurons to be excluded from the neural network (i.e. drop out) at each step with a fixed probability. Since the model won't know which neurons will be dropped out at each step, it is forced to not overly rely on key neurons, thereby preventing wild and crazy models.



(a) Standard Neural Net

(b) After applying dropout.

Exercise

Use **housepricedata** and predict whether the house price is above median or not

Let's implement neural network model below with Keras.