Keras: The Python Deep Learning library

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. *Being able to go from idea to result with the least possible delay is key to doing good research.*

Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

Keras is compatible with: **Python 2.7-3.6**.

Keras is an Open Source Neural Network library written in Python that runs on top of Theano or Tensorflow. It is designed to be modular, fast and easy to use. It was developed by François Chollet, a Google engineer.

Keras doesn't handle low-level computation. Instead, it uses another library to do it, called the "Backend. So Keras is high-level API wrapper for the low-level API, capable of running on top of TensorFlow, CNTK, or Theano.

Keras High-Level API handles the way we make models, defining layers, or set up multiple input-output models. In this level, Keras also compiles our model with loss and optimizer functions, training process with fit function. Keras doesn't handle Low-Level API such as making the computational graph, making tensors or other variables because it has been handled by the "backend" engine.

What is a Backend?

Backend is a term in Keras that performs all low-level computation such as tensor products, convolutions and many other things with the help of other libraries such as Tensorflow or Theano. So, the "backend engine" will perform the computation and development of the models. Tensorflow is the default "backend engine" but we can change it in the configuration.

Theano, Tensorflow, and CNTK Backend

Theano is an open source project that was developed by the MILA group at the University of Montreal, Quebec, Canada. It was the first widely used Framework. It is a Python library that helps in multi-dimensional arrays for mathematical operations using Numpy or Scipy. Theano can use GPUs for faster computation, it also can automatically build symbolic graphs for computing gradients. On its website, Theano claims that it can recognize numerically unstable expressions and compute them with more stable algorithms, this is very useful for our unstable expressions.

On the other hand, Tensorflow is the rising star in deep learning framework. Developed by Google's Brain team it is the most popular deep learning tool. With a lot of features, and researchers contribute to help develop this framework for deep learning purposes.

Another backend engine for Keras is The Microsoft Cognitive Toolkit or CNTK. It is an open-source deep learning framework that was developed by Microsoft Team. It can run on multi GPUs or multi-machine for training deep learning model on a massive scale. In some cases, CNTK was reported faster than other frameworks such as Tensorflow or Theano.

Comparing the Backends

We need to do a benchmark In order to know the comparison between this two backends. As you can see in Jeong-Yoon Lee's benchmark, the performance of 3 different backends on different hardware is compared. And the result is Theano is slower than the other backend, it is reported **50 times** slower, but the accuracy is close to each other.

Another benchmark test is performed by Jasmeet Bhatia. He reported that Theano is slower than Tensorflow for some test. But overall accuracy is nearly the same for every network that was tested.

So, between Theano, Tensorflow and CTK it's obvious that TensorFlow is better than Theano. With TensorFlow, the computation time is much shorter and CNN is better than the others.

Keras vs Tensorflow

Parameters	Keras	Tensorflow
Type	High-Level API Wrapper	Low-Level API
Complexity	Easy to use if you Python language	You need to learn the syntax of using some of Tensorflow function
Purpose	Rapid deployment for making model with standard layers	Allows you to make an arbitrary computational graph or model layers
Tools	Uses other API debug tool such as TFDBG	You can use Tensorboard visualization tools
Community	Large active communities	Large active communities and widely shared resources

Guiding principles

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

Advantages of Keras

Fast Deployment and Easy to understand

Keras is very quick to make a network model. If you want to make a simple network model with a few lines, Keras can help you with that. Look at the example below:

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

```
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=50)) #input shape of 50
model.add(Dense(28, activation='relu')) #input shape of 50
model.add(Dense(10, activation='softmax'))
```

Because of friendly the API, we can easily understand the process. Writing the code with a simple function and no need to set multiple parameters.

Disadvantages of Keras

Cannot handle low-level API

Keras only handles high-level API which runs on top other framework or backend engine such as Tensorflow, Theano, or CNTK. So it's not very useful if you want to make your own abstract layer for your research purposes because Keras already have pre-configured layers.

Keras Fundamentals for Deep Learning

The main structure in Keras is the Model which defines the complete graph of a network. You can add more layers to an existing model to build a custom model that you need for your project.

Here's how to make a Sequential Model and a few commonly used layers in deep learning

1. Sequential Model

The core data structure of Keras is a **model**, a way to organize layers. The simplest type of model is the `Sequential` model, a linear stack of layers. For more complex architectures, you should use the Keras functional API, which allows to build arbitrary graphs of layers.

```
from keras.models import Sequential
from keras.layers import Dense, Activation,Conv2D,MaxPooling2D,Flatten,Dropout

model = Sequential()
```

Stacking layers is as easy as `.add()`:

```
from keras.layers import Dense
```

```
model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

2. Convolutional Layer

This is an example of convolutional layer as the input layer with the input shape of 320x320x3, with 48 filters of size 3x3 and use ReLU as an activation function.

```
input_shape=(320,320,3) #this is the input shape of an image 320x320x3
model.add(Conv2D(48, (3, 3), activation='relu', input_shape= input_shape))
```

another type is

```
model.add(Conv2D(48, (3, 3), activation='relu'))
```

3. MaxPooling Layer

To downsample the input representation, use MaxPool2d and specify the kernel size

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

4. Dense Layer

adding a Fully Connected Layer with just specifying the output Size

```
model.add(Dense(256, activation='relu'))
```

5. Dropout Layer

Adding dropout layer with 50% probability

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

Compiling, Training, and Evaluate

After we define our model, let's start to train them. It is required to compile the network first with the loss function and optimizer function. This will allow the network to change weights and minimized the loss.

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

Now to start training, use fit to fed the training and validation data to the model. This will allow you to train the network in batches and set the epochs.

```
model.fit(X_train, X_train, batch_size=32, epochs=10, validation_data=(x_val, y_val))
```

Our final step is to evaluate the model with the test data.

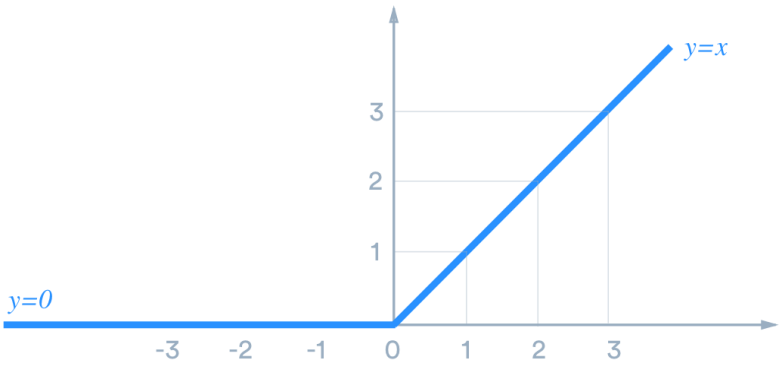
```
score = model.evaluate(x_test, y_test, batch_size=32)
```

The Rectified Linear Unit (RELU)

The Rectified Linear Unit, or ReLU, is not a separate component of the convolutional neural networks' process.

It's a supplementary step to the convolution operation.

ReLU stands for rectified linear unit, and is a type of activation function. Mathematically, it is defined as *y = max(0, x)*. Visually, it looks like the following:



ReLU is the most commonly used activation function in neural networks, especially in CNNs. If you are unsure what activation function to use in your network, ReLU is usually a good first choice.

How does ReLU compare

ReLU is linear (identity) for all positive values, and zero for all negative values. This means that:

- It’s cheap to compute as there is no complicated math. The model can therefore take less time to train or run.

- It converges faster. Linearity means that the slope doesn't plateau, or "saturate," when *x* gets large. It doesn't have the vanishing gradient problem suffered by other activation functions like **sigmoid** or **tanh**.
- It's sparsely activated. Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all.

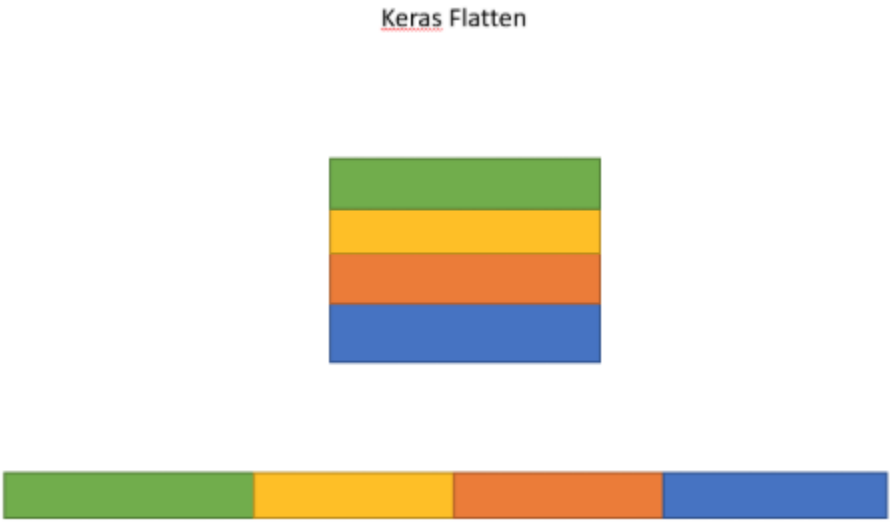
Sparsity

Note: We are discussing model sparsity here. Data sparsity (missing information) is different and usually bad.

Why is sparsity good? It makes intuitive sense if we think about the biological neural network, which artificial ones try to imitate. While we have billions of neurons in our bodies, not all of them fire all the time for everything we do. Instead, they have different roles and are activated by different signals.

Sparsity results in concise models that often have better predictive power and less overfitting/noise. In a sparse network, it's more likely that neurons are actually processing meaningful aspects of the problem. For example, in a model detecting cats in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is about a building.

Role of the Flatten Layer in CNN



This is how Flatten works converting Matrix to single array.

Adam

Adam: Adaptive moment estimation

Adam = RMSprop + Momentum

Some advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters.

In Keras, we can define it like this.

```
keras.optimizers.Adam(lr=0.001)
```

What is Momentum?

Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.

Verbose in CNN

By setting verbose 0, 1 or 2 you just say how do you want to 'see' the training progress for each epoch.

`verbose=0` will show you nothing (silent)
`verbose=1` will show you an animated progress bar like this:

```
[==============================]
```

`verbose=2` will just mention the number of epoch like this:

```
Epoch 1/10
```

Epochs in CNN

One epoch consists of *one* full training cycle on the training set. Once every sample in the set is seen, you start again - marking the beginning of the 2nd epoch.

You can't be sure if 5 epochs or 500 is enough for convergence since it will vary from data to data. You can stop training when the error converges or gets lower than a certain threshold.

sparse_categorical_crossentropy

As one of the multi-class, single-label classification datasets, the task is to classify grayscale images of (28 pixels by 28 pixels), into their ten categories (0 to 9). Let's build a Keras CNN model to handle it with the last layer applied with "**softmax**" activation which outputs an array of ten probability scores (summing to 1). Each score will be the probability that the current image belongs to one of our 10 classes.

For such a model with output shape of (None, 10), the conventional way is to have the target outputs converted to the one-hot encoded array to match with the output shape, however, with the help of the `sparse_categorical_crossentropy` loss function, we can skip that step and keep the integers as targets.

TensorBoard: TensorFlow's visualization toolkit

TensorBoard provides the visualization and tooling needed for machine learning experimentation:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases, or other tensors as they change over time
- Projecting embeddings to a lower dimensional space
- Displaying images, text, and audio data
- Profiling TensorFlow programs
- And much more