A Guide to using Keras

Author: Louis Miranda-Smedley

Intended as a guide for future years or those interested in using Keras for creating neural networks.

Backend

Keras is made available by different machine learning environments namely, Tensorflow and Theano. It is down to the user which backend they prefer to use. Currently Tensorflow is one of the most widely used open source libraries in industry. Whilst the base language of Tensorflow is C++ or Python, Theano completely runs on Python. The backend can be changed from using Tensorflow to Theano and vice versa, by editing the keras.json file that comes from installing keras. In the Cartpole-v0 solution we used Theano as the backend for Keras. Theano and Tensorflow both have their advantages, and these will be compared below to allow a better informed choice based on the problem.

Tensorflow

Benefits

- Supports reinforcement learning and other algorithms
- Offers computational graph abstraction
- Faster compile time than Theano
- TensorBoard feature for visualisation
- Can be deployed on multiple CPUs and GPUs

Drawbacks

- Matrix operations not supported
- Significantly slower to run
- Computational graph can be slow
- Doesn't have pertained models
- Not commercially supported

Theano

Benefits

- Open-source deep libraries such as Keras, Lasagne and Blocks have been built on top of Theano
- Raw Theno is somewhat low level
- It has some high level wrappers such Keras, Lasagne which increases it usability

Drawbacks

- Can be troublesome to run on AWS (Amazon Web Services)
- Can only deploy on single GPU
- Large models can demand long compile times
- Error messages can be unhelpful at times for debugging

Models

Which model is used to create the neural network is important for ensuring success of its implementation. The following models are compared for their aptness for given problems. For the CartPole-v0 environment, the Sequential() model is sufficient as this problem requires only optimizing weights in the direction from input to output layer.

Sequential API

The Sequential() model is the preferred Keras model for most use cases. The Sequential() Model is a linear stack of layers, with input layer at the top, then the hidden layers, and finally the output layer. For this structure, an input shape is required in the input layer, and since the next layer will assume to be connected by the previous layer, the input shape is not required after the first layer. To get Keras to show your model in the way as seen in figure 1, you need to install graphviz and use the plot_model() function.

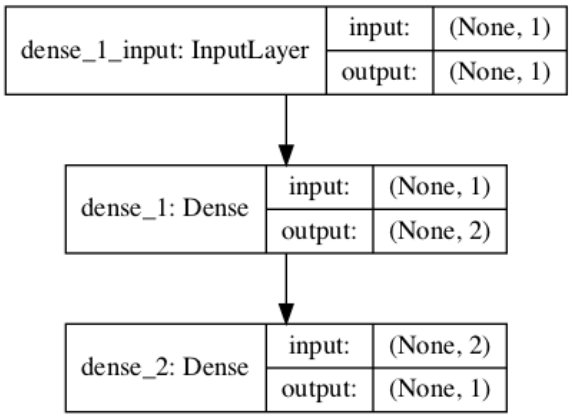

Figure 1: Sequential() model visualisation.

Benefits:

- Great for Deep Learning Models
- Can be used for most problems
- Logical layer by layer structuring

Drawbacks:

- Not straight-forward for layers to be shared by other layers
- Not straight-forward for including multiple input sources and multiple output sources
- Not straight-forward for re-use of layers

Functional API

Model() can be used instead of Sequential() instead for its greater flexibility in designing the architecture of the network. The Model() function allows for greater complexity in networks, essentially improving on the limitations that face using Sequential(). This makes networks such as Siamese and Residual networks to be possible to model using the functional API.

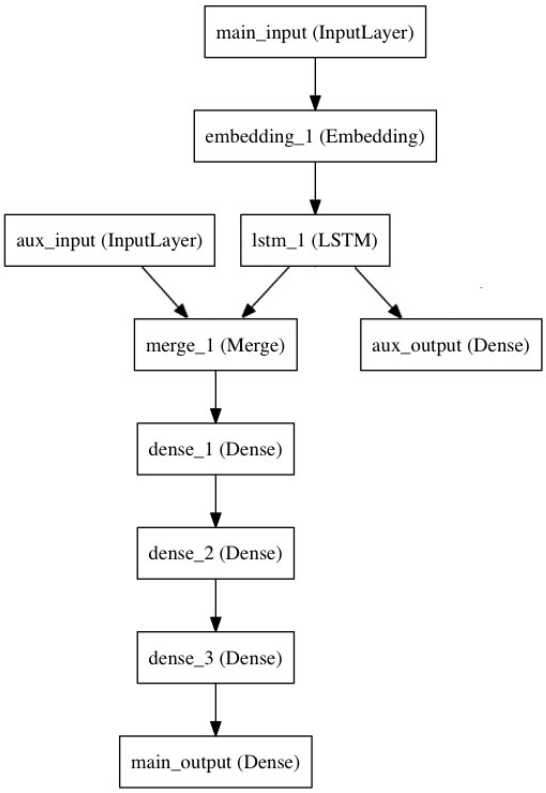

Figure 2: Model() Functional API visualisation.

Benefits:

- Layer sharing between layers
- Multiple input sources
- Multiple output destinations
- Cyclic layers (connected to themselves)

Activation functions

All forward layers in the Keras model of the neural net have the option to specify an activation. An activation function describes the mathematical operation being done on the inputs to a node in the network. Therefore, the activation function decides whether or not the node should be activated or not, based on whether the inputs are relevant to the model's prediction. Since this is an operation that is done at each node, in a network which could have millions of nodes, it is important that each of these activations are computationally efficient.

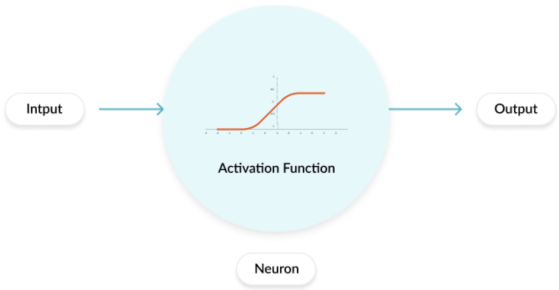

Figure 3: Activation function in neural network.

Keras includes 11 types of activation function which may be used.

- 'elu' - Exponential linear unit
- 'softmax' - Softmax activation function
- 'selu' - Scaled exponential linear unit
- 'softplus' - Softplus activation function
- 'softsign' - Softsign activation function
- 'relu' - Rectified linear unit
- 'tanh' - Hyperbolic tangent activation function
- 'sigmoid' - Sigmoid activation function
- 'hard_sigmoid' - Hard Sigmoid activation function
- 'exponential' - Exponential activation function
- 'linear' - Linear activation function

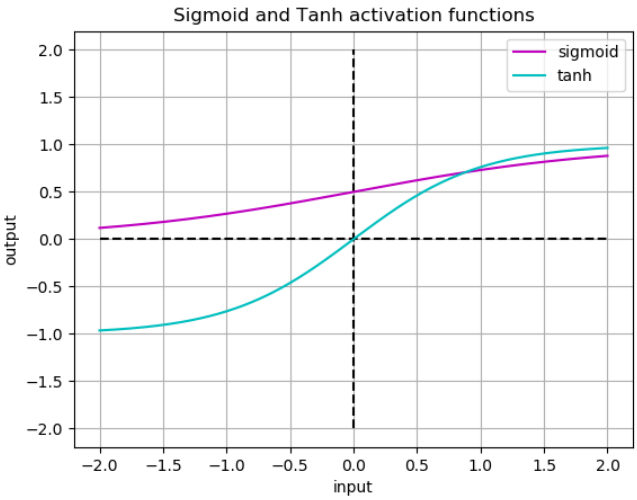

Figure 4: Sigmoid and Tanh activation functions

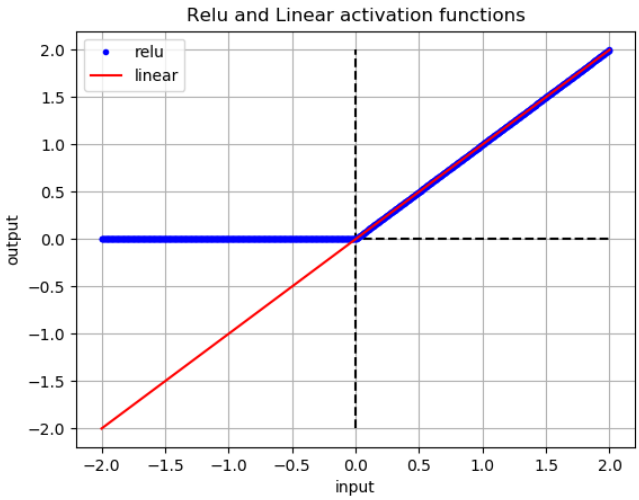

Figure 5: Relu and Linear activation functions

The most common activation functions are shown above, containing a mixture of linear and non-linear activation functions. Linear activation functions have a constant gradient so cannot be used in back-propagation to allow the model to learn from the error in predictions on training data. This is why non-linear functions such as the sigmoid function are commonly used as activation functions. The gradient of the sigmoid function is proportional to the adjustment in weights since the gradient decreases close to 0 and 1 suggesting growing confidence in the network.

Loss functions

Deep learning neural networks are based on optimization algorithms. As part of the optimizer algorithm (which optimizes the weights), the error for the current state of the system must be estimated repeatedly. This requires a choice of an error function, or otherwise known as a loss function, which can be used to estimate the loss of the model so the weights may be updated to reduce loss of the next run. There are three types of loss function: (1) Regression Loss functions, (2) Binary Classification Loss functions and (3) Multi-Class Classification Loss functions. The mean-squared error ('mse') is the most common loss function used in regression problems and was used in the CartPole-v0 Deep Q-Network solution. The loss function is a specified argument of the model.compile() function in Keras which compiles the layers to form a neural network.

Optimizers

The choice of whether to use the Adam or Stochastic Gradient Descent (SGD) optimizer in a machine learning problem is subjective. Adam is an adaptive learning rate optimization algorithm that has been designed to specifically train deep neural networks. The learning rate, commonly denoted as α, is measured between 0 and 1 and essentially quantifies the weight given of model predictions based on new information (learning). With time the model should improve and therefore, more of a weight should be given on past actions that goes into future actions rather than current actions and so the learning rate usually decays. This is how the Adam optimiser is used to solve problems. Adam is much faster than SGD and its default parameters usually work fine, there is also the option to define your own learning rate and decay. Keras takes 'Adam()' as an optimizer as the second argument to loss function in the model.compile() function. In the Cartpole-v0 solution, the Adam optimiser was used as well as defining a specific learning rate and decay rate. The point of these optimizers is to converge to a solution which optimally solves a problem and for those using the optimizers, convergence time is important. In some cases Adam struggles to converge faster than SGD and there are other optimizers out there which claim to also have better convergence times.

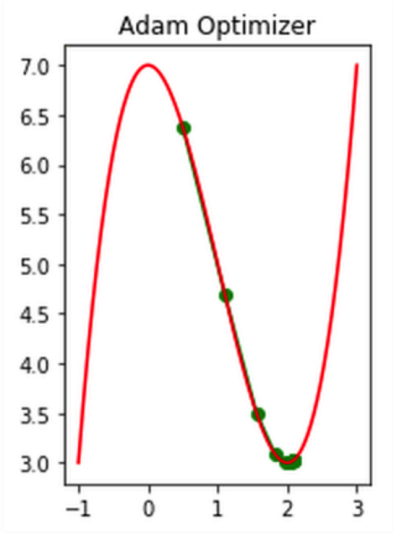

Figure 6: Adam optimizer

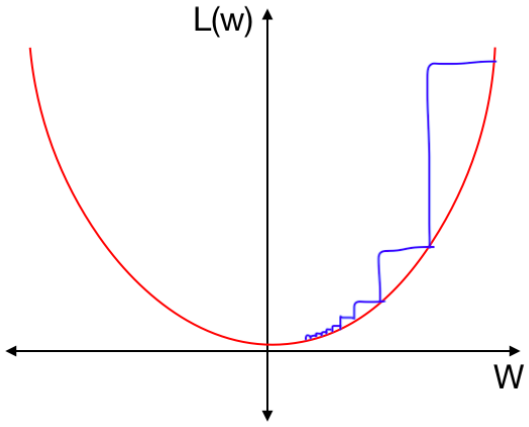

Figure 7: SGD optimizer

CartPole-v0

Considering this project is focused on swinging and the agent finding an optimal way to swing in order to build higher amplitudes, we started with trying to solve to CartPole problem. This is made available from the OpenAI Gym environment and great way to be introduced to machine learning. This problem concerns a cart which is connected to a pole by a pivot and the cart is able to apply forces to the left and right in order to balance the pole.

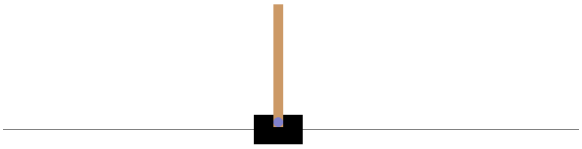

Figure 8: CartPole-v0 environment

In order to solve this problem, we used a Deep Q-Network which is based on reinforcement learning. To start with the agent takes random actions and a reward is given based on the wellness of the action, then with time the agent uses the model to take actions instead of taking random actions. This merges reinforcement learning and Q-Learning. Instead of a Q-Table a neural network is used, using the Keras Sequential model. An input layer of 4 nodes is densely connected to two hidden layers, containing 24 and 48 nodes respectively, and then finally fed into 2 output nodes. The loss function was chosen to be mean-squared error and the optimiser used was Adam. We also used the 'tanh' and 'linear' activation functions which worked well, we also tried 'relu' which took longer to solve.

```
#Neural Net
self.model = Sequential()
self.model.add(Dense(24, input_dim=4, activation='tanh'))
self.model.add(Dense(48, activation='tanh'))
self.model.add(Dense(2, activation='linear'))

self.model.compile(loss='mse', optimizer=Adam(lr=self.alpha, decay=self.alpha_decay))
```

Figure 9: Building neural network using Sequential model.

The CartPole problem states on the OpenAI Gym website that a winning score is a mean survival time of 195 ticks (environment steps). Using the model defined above and the code ran on the cartpole_class.py file in this folder we achieved the following convergence to the solution.

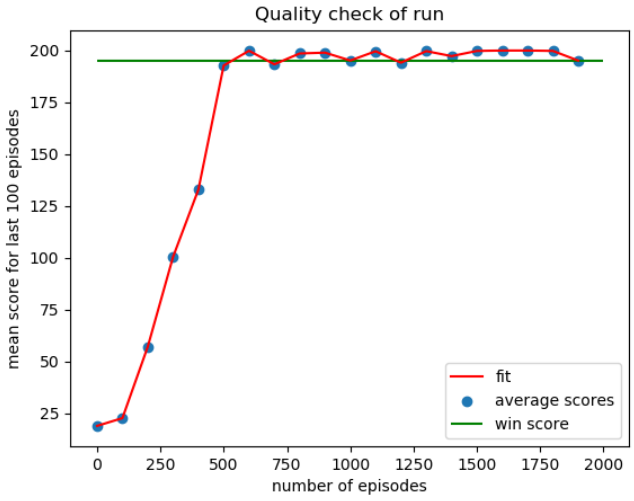

Figure 10: Example plot of balancing the pole

References

[1] https://keras.io/getting-started/sequential-model-guide/

[2] https://keras.io/getting-started/functional-api-guide/

[3] https://keras.io/losses/

[4] https://keras.io/optimizers/

[5] https://keras.io/activations/