Notes On TensorFlow & Keras

Ara Patapoutian, 2017

CONTENTS

I. INTRODUCTION

- General:
  - *TensorFlow*, or TF, is an open source software library for numerical computation, machine learning & NN
  - TF is an *application program interface* (API) which is a set of routines, protocols, and tools for building software applications
    - ∗ the core of TF is written in C++
  - the primary API is accessed through Python
    - ∗ the variable that displays version `tf.__version__`
    - ∗ TF can also be accessed through the C & C++
  - TF has better support for distributed systems than competing software packages
- TF includes three packages
  - TensorFlow
  - TensorBoard
  - TensorFlow Serving
- TensorBoard:
  - *TensorBoard* is a visualizations software that is included with TF installation
- TensorFlow Serving:
  - *TF Serving* is a software that facilitates easy deployment of pre-trained TF models
  - a user can export their model to a file which can then be read by TF serving
  - TF Serving seamlessly switches old models with new ones
  - TF Serving enables users to avoid re-implement their models for production
  - TF Serving is is written in C++

A. *Graphs*

- Computational graph:
  - TF uses *data-flow graphs* or *computational graph*
  - a computational graph is a directed graph
  - no computation is performed while a graph is constructed
  - graph representation allows distributing computations across multiple CPU/GPUs
  - a NN is naturally modeled on a computational graph
- Nodes and edges:
  - nodes constitute operations or *ops*
    * the nodes are objects of type `tf.Operation`
    * each node is differentiable
  - edges represent data by tensors
    * the edges are objects of type `tf.Tensor`
    * only tensors may be passed between nodes
    * hence the name TensorFlow
- Differentiation:
  - differentiation is used for gradient based optimization or training
  - TF can take the derivative of any node with respect to any other
  - as a rule of thumb, back propagation takes about twice the memory and twice the computation as the forward path
- `Graph`
  - graphs belong to the class `Graph`
  - `default_graph = tf.get_default_graph()`
    * `get_default_graph()` provides a handle of the default graph
    * a default `Graph` is created when TF is loaded
  - `g = tf.Graph()` constructs a graph explicitly
  - `g.as_default`
    * to add an operation to graph `g`, use `g.as_default`
      ```
      with g.as_default():
          a = tf.mult(2,3)
      ```
    * see below for more on `with` statement
  - `w1 = graph.get_tensor_by_name("w1:0")`
    * returns the Tensor with the given name
    * allows access to saved variables, tensors, or placeholders
  - `tf.reset_default_graph()` resets a graph
- `with`
  - the `with` statement is used to wrap the execution of a block with methods defined by a context manager
  - a *context manager* is an object that defines the runtime context to be established when executing a `with` statement
  - the context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code

B. *Environments*

- TF through `anaconda`
  - go to the desired directory
  - to activate
    ```
    source activate env_name
    ```
    * my environment names are `tensorflow` and `carnd-term1`
  - choose `python` or `jupyter notebook`
  - to deactivate
    ```
    source deactivate
    ```
- TF through `virtualenv`
  - to activate type
    ```
    source ~tf_env/bin/activate
    ```
- TF through Python:

- in `virtualenv`, enter Python by typing
  `python`
- from the Python cursor type
  `import tensorflow as tf`
- to avoid overhead, TF describes a graph of interacting operations that run entirely outside Python
- the role of the Python code is to build this external computation graph, and to dictate which parts of the computation graph should be run

II. Tensors

- Variables in TF & Python:
  - names in Python-space are temporary pointers to TF variables during the run of the script
  - saving & restoring variables are done through TF namespace, not Python
- Tensor:
  - the *tensor* data-structure is used to represent all data
  - a tensor has a dynamic rank, shape & a static type
    - *rank* is the number of dimensions of the tensor
    - *shape* describes the the sizes of each dimension
  - list of static TF data *types* and the corresponding Python `dtype`:

TF	Python	bits
`DT_FLOAT16`	`tf.float16`	16
`DT_FLOAT`	`tf.float`	32
`DT_DOUBLE`	`tf.double`	64
`DT_INT8`	`tf.int8`	8
`DT_INT16`	`tf.int16`	16
`DT_UINT8`	`tf.uint8`	8
`DT_UINT16`	`tf.unit16`	16
`DT_QINT8`	`tf.qint8`	8
`DT_STRING`	`tf.string`	8*
`DT_BOOL`	`tf.bool`	1

    - letters `U` & `Q` respectively indicate unsigned and quantized
- `tf.Variable()`
  - a `Variable()` is a value that lives in TensorFlow's computation graph
    - the value returned by `tf.Variable()` is an instance of the Python class `tf.Variable`
  - a TF variable reside in memory buffer, containing tensors
    - variables can also be saved to disk
    - variables are mutable tensors
  - the value of a `Variable()` can be changed using the `Variable.assign()` method
    - they are executed in `Session`, see Section IV

```
b = a.assign(a*2)
```

  - addition & subtraction can be performed using specific methods

```
b = a.assign_add(2)      #b=a+2
c = a.assign_sub(1)      #c=a-1
```

  - if a variable should not change during training, define it as

```
a = tf.Variable(1, trainable=False)
```

  - calling `tf.Variable()` adds several operations to the graph:
    1) a variable operation that holds the variable value
    2) an initializer operation that initializes the variable
       - initialization is a `tf.assign()` operation
    3) the ops for the initial value (e.g. zeros op), are also added to the graph
- Special tensors:

```
tf.zeros(shape, dtype=tf.float32)
   e.g., tf.zeros([3, 4], tf.int32)
tf.zeros_like(tensor, dtype=None)
tf.ones(shape, dtype=tf.float32)
tf.ones_like(tensor, dtype=None)
tf.fill([3,4], value, name=None)
tf.constant(value,dtype=None,shape=None)
   e.g., tf.constant([4, 2, 3])
tf.linspace(start, stop, num)
tf.range(start,limit=None,delta=1)
tf.random_normal(shape, mean=0.0,stddev=1.0,dtype=tf.float32,seed=None)
tf.truncated_normal(shape, mean=0.,stddev=1.0, dtype=tf.float32,seed=None)
```

```
tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32,seed=None)
tf.random_shuffle(value, seed=None)   #along first dimension
tf.random_crop(tensor,size,seed=None) #randomly crops a tensor to a given size
```

- Random seeding:
  - random operations use two seeds: graph-level & operation-level seeds
  - the following command sets the graph-level seed

    `tf.set_random_seed(seed)`

- `tf.Variable()` Initialization:
  - variables proceed through 3 steps:
    * specified
    * added, or initialized, to graph
    * executed
  - when a variable is created, a tensor is specified as its initial value to the `Variable()` constructor
  - example:
    * `x = tf.Variable([1.0, 2.0], name="y")`
    * `tf.Variable()` only *specifies* the initial value, it does not perform the initialization
    * the state of a variable is managed by class `Session`, see Section IV
    * a `Variable` should be initialized within a `Session`
  - special tensors can be used to initialize a `tf.Variable()`

    `a = tf.Variable(tf.zeros([3, 4], tf.int32))`
  - to initialize a variable from the value of another variable, use `initialized_value()` method

    `w2 = tf.Variable(w1.initialized_value())`
  - an op that can be used within a `Session`, to initialize all the variables is

    `init_op = tf.global_variables_initializer()`
    * note that the above command adds the op the graph, it still does not run it
    * to initialize need to use `run()` command within session
  - an op that is used to initialize some of the variables is

    `init_op = tf.initialize_variables([v1])`
  - variable initializers must be run explicitly, before other operations can be run
- `tf.device()`
  - a variable can be pinned to a particular device,
  - examples of devices are CPUs & GPUs,

    ```
    tf.device("/cpu:0"):
       v = tf.Variable(...)
    ```
- `tf.placeholder()`
  - `tf.placeholder()` is a variable that will be assigned data, at a later date
  - `tf.placeholder()` creates an input node on graph
  - examples:

    ```
    a = tf.placeholder(tf.float32, name="my_input")
    b = tf.placeholder(tf.float32, shape=[None, 784])
    ```
  - the Python data type argument (`dtype`) is required
  - `shape` is an optional parameter

III. Operators

- `tensor.get_shape()`
  - `print(x.get_shape())`
  - `x.get_shape()` is the static shape of `x`
  - the dynamic shape (during a session) can be accessed using `x.shape`
- `tf.TensorShape.as_list()`
  - returns a list of integers or None for each dimension
  - `x.get_shape().as_list()`

- `tf.reshape()`
  - `reshape(tensor,shape)`
  - `tf.reshape(x, [-1,28,28,1])`
- `tf.transpose()`
  - `transpose(x,perm=None)`
  - transposes `x` and permutes the dimensions according to `perm`
- `tf.one_hot()`
  - `tf.one_hot(x,_size)`
  - converts x to one-hot format
- `tf.split()`
  - `split(value,num_or_size_splits,axis=0)`
  - splits a tensor into sub tensors
  - `num_or_size_splits` can be an integer that evenly divides `value.shape[axis]`
  - `num_or_size_splits` can also be a tensor that partitions `value` into `len(size_splits)` pieces
  - `split0, split1, split2 = tf.split(value, [4, 15, 11], 1)`
- `tf.pack()`
  - `tf.pack([a, b, c])`
  - packs a list of rank-R tensors into 1 rank-(R+1) tensor
- `tf.assign()`
  - `tf.assign(x_ref, _value)`
  - update `x_ref` by assigning `_value` to it
- `tf.cast()`
  - `tf.cast(x, tf.float32))`
- General operator examples:

```
tf.add(x,y)                  #addition
tf.sub(x,y)                  #subtraction
tf.div(x,y)                  #elementwise integer or floating point division
tf.truediv(x,y)              #elementwise floating point division
tf.mul(x,y)                  #multiplication
tf.matmul(x,W)               #matrix multiplication
tf.mod(x,y)                  #x % y
tf.pow(x,y)                  #x.^y
tf.log(x)                    #natural log
tf.reduce_mean()             #take the average
tf.reduce_sum()              #take the sum
tf.argmax(X,axis)            #index of the highest entry along some axis
tf.squared_difference(x,y)   #elementwise
tf.sigmoid(x)                #elementwise sigmoid fn
tf.equal(x,y)                #1 if equal
tf.less(x,y)                 #<
tf.less_equal(x,y)           #<=
tf.greater(x,y)              #>
tf.greater_equal(x,y)        #>=
tf.logical_and(x,y)          #&
tf.logical_or(x,y)           #|
tf.logical_xor(x,y)          #^
```

- NN specific operators:

```
tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1], padding='SAME')
tf.nn.relu(y)
tf.nn.tanh(y)
tf.nn.softmax(y)
tf.nn.dropout(h_fc, keep_prob) #it multiplies kept units by 1/keep_prob.
tf.nn.sigmoid_cross_entropy_with_logits(y, y_) #cross entropy with sigmoid
tf.nn.softmax_cross_entropy_with_logits(y, y_) #cross entropy with softmax
tf.nn.sparse_softmax_cross_entropy_with_logits(y, y_) #sparse/one-shot
```

```
tf.nn.bias_add(n_layer,bias)|                        #adds bias, addition
y = tf.nn.xw_plus_b(_x, _w, _b)                      # X W + B
_v, _i=tf.nn.top_k(input, k=3) #returns k highest values and their indices
```

- Image input format:
  - TF's input pipeline format is optimized to work with a batch of images
  - a rank-four tensor is used as the NN input structure with shape `[in_batch_size, in_height, in_width, in_channels]`
    1) `in_batch_size` is the batch size
    2) `in_height` is the rows of an image
    3) `in_width` is the columns of an image
    4) `in_channels` is the color channels
- `tf.layers`
  - a library that provides a set of high-level neural networks layers
  - e.g. `Conv2d`, `Conv2d_transpose`, `Dense`, `MaxPooling2D`, etc.
- `out=tf.nn.conv2d(input_batch, filter, strides, padding)`
  - `input_batch` is rank-4 tensor as described above
    * data type is `float16`, `float32` or `float64`
  - `filter` is the kernel which is a tensor with the same type as `input_batch`
    * shape is `[filter_height, filter_width, in_channels, out_channels]`
  - `strides` causes the filter to skip over pixels
    * a list of integers, of length 4, corresponding to the four dimensions of `input_batch`
    * e.g `strides = [1 , 2, 2, 1]`
  - `padding`, which is assigned to `'SAME'` or `'VALID'`, describes how to fill the missing area in image
    * if `padding='SAME'`, then

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
pad_along_height = ((out_height - 1) * strides[1] +
                    filter_height - in_height)
pad_along_width = ((out_width - 1) * strides[2] +
                   filter_width - in_width)
pad_top = pad_along_height / 2
pad_left = pad_along_width / 2
```

    * if `padding='VALID'`, then

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
pad_along_height = ((out_height - 1) * strides[1] +
                    filter_height - in_height)
pad_along_width = ((out_width - 1) * strides[2] +
                   filter_width - in_width)
pad_top = pad_along_height / 2
pad_left = pad_along_width / 2
```

  - in detail, with the default NHWC format, the function computes

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
                    filter[di, dj, q, k]
```

- `conv2d_transpose`
  - sometimes called deconvolution after Deconvolutional Networks, but is actually the transpose (gradient) of `conv2d` rather than an actual deconvolution
  - a stride of 2, and "SAME" padding would double the output dimensions
- `tf.metrics`
  - metric functions in this library return a Tensor for the metric result and a Tensor Operation to generate the result
    * need to run operation before getting the result
  - the other characteristic of TensorFlow metric functions is the usage of local TensorFlow variables

    - these are temporary TensorFlow Variables that must be initialized by running `tf.local_variables_initializer()`
  - examples include
    `accuracy()`
    `mean_squared_error()`
    `root_mean_squared_error()`
    `mean_iou()`

IV. Session & Train

A. Session

- `tf.Session()` & `tf.InteractiveSession()`
  - sessions are responsible for graph execution
  - Tensorflow variables are only alive inside a session
  - launch one of two classes `tf.Session()` or `tf.InteractiveSession()`
  - `tf.InteractiveSession()` class makes TensorFlow more flexible about how it is structured
    - it allows to interleave operations that build a computation graph with ones that run the graph
  - these classes make the connection to the highly efficient C++ back-end to do its computation
  - will focus on `tf.Session()`
- `sess   = tf.Session()`
  - `tf.Session()` is a constructor for a session
  - `tf.Session()` takes three *optional* arguments:
    - `target` specifies the execution engine
    - `graph` specifies the `Graph` object
    - `config` allows users to specify options to configure the session
  - the constructor places the graph operations onto specific devices
  - the associated graph attribute is `sess.graph`
- `sess.run()`
  - two common arguments of the `run()` method are
    - `fetches`
    - `feed_dict`
  - `fetches` is not optional
- `sess.run(fetches)`
  - `fetches` can be any graph element
    - in other words, `fetches` can be an operation or a tensor
  - if `fetches` is a tensor, the output will be a NumPy array
    `output = sess.run(v)`
    - computes tensor `v` and assigns it to `output`
    - it is also possible to pass a list of graph elements
      `sess.run([v, u])`
    - if the argument is a list so will be the output
  - if `fetches` is an operator the output will be `None`
    - `sess.run(tf.global_variables_initializer())` initializes all the variables
- `feed_dict`
  - `feed_dict` is an optional argument to `run()`
    - it overrides tensor values in the graph
  - `feed_dict` expects a Python dictionary object as input
    - the keys in the dictionary are handles to tensor objects that should be overridden
    - the values can be numbers, strings, lists or NumPy arrays
  - `sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})`
    - in the above example, the `session` runs an element called `optimizer` for `x = batch_xs` & `y = batch_y`
- `tensor.eval()`
  - `t.eval()` is a shortcut for calling `tf.get_default_session().run(t)`

  - can also use it to print tensors
    `print(t.eval())`
- `sess.close()`
  - release a `Session` by typing `sess.close()`
- `with`
  - there are two ways of using `with` to run a `Session`:
  - the first approach is as follows

```
sess = tf.Session()
with sess.as_default():
    a.eval()
    ...
sess.close()
```

  - the second approach is

```
with tf.Session() as sess:
    ...
```

    in this second approach, there is no need to `close()` a `Session` at the end

B. Train

- `tf.train`
  - `tf.train.Optimizer` is a base class for gradient based optimizations
  - `tf.train.SummaryWriter` is a class that creates an event file in a given directory and adds summaries to it
  - `tf.train.Saver` saves the environment in proprietary format
  - `tf.train.string_input_producer()` to read files
- `tf.train.Optimizer`
  - TF uses automatic differentiation to find the gradients of the loss function with respect to each of the variables
  - specific classes include

```
GradientDescentOptimizer
AdamOptimizer
AdagradOptimizer
MomentumOptimizer
```

  - examples:

```
optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(total_loss)
optimizer = tf.train.AdamOptimizer(1e-4).minimize(total_loss)
```

  - in the above two examples, new operations are added to the graph
  - the optimizer returns a single operation `optimizer`
  - when run, this operation performs a step of training

```
training_operation = optimizer.minimize(cost_function)
sess.run(training_operation,feed_dict={x: batch[0], y_: batch[1]})
```

- `tf.stop_gradient()`
  - `tf.stop_gradient()` prevents the gradient from flowing backwards past a specified point
  - `layer_n = tf.stop_gradient(layer_n)`
- `tf.train.SummaryWriter`
  - used by `TensorBoard`
  - `writer = tf.train.SummaryWriter('./my_graph',sess.graph)`
  - graph description will be stored in directory `'./my_graph'`
  - `SummaryWriter` will output the description of the graph, i.e. `sess.graph`, into `my_graph` directory
- `tf.train.Saver()`
  - the `Saver` class adds operations to save and restore variables to and from checkpoints
    * *checkpoints* are binary files in a proprietary format which map variable names to tensor values.
    * the role of the class `Saver` is to save the tf environment
    * `Saver` class provides the functionality to save any `tf.Variable` to a file system

- the constructor is
  `saver = tf.train.Saver()` or
  `saver = tf.train.Saver({'v1': v1, 'v2': v2})` where specific variables are passed as dictionary, or
  `saver = tf.train.Saver([v1, v2])`, where variables are passed as a list
- `saver.save(sess, "mymodel", global_step=3)`
  * the `save` method should be run within a `Session`
  * multiple filenames will be created or updated under the name `mymodel-3`
  * by default, the `saver` will keep only the most recent five files
- four files are generated with `save` method:
  * `mymodel-3.index` is a checkpoint file
  * `mymodel-3.data` is a checkpoint file that contains the values of training variables
  * `checkpoint` file has the list of checkpoint filenames
  * `mymodel-3.meta` is a `meta graph` protocol buffer which saves the complete Tensorflow graph
- to save a subset of the variables, specify the names and variables to be saved by passing a Python dictionary: keys are the names to use, values are the variables to manage,
  ```
  v2 = tf.Variable(..., name="v2")
  saver = tf.train.Saver({"my_v2": v2})
  ```
- to verify if a checkpoint is available, use command
  `ckpt = tf.train.get_checkpoint_state(os.path.dir-name(__file__))`
- ro recreate the model (not values), type
  `saver = tf.train.import_meta_graph('mymodel-3.meta')`
- to restore the variable values, type
  `saver.restore(sess, "/tmp/model.ckpt")`, or
  `saver.restore(sess, tf.train.latest_checkpoint('.'))`
  * while restoring `Session` should be active
  * all restored variables should be pre-defined in Python

• `saved_model`
- to save or restore a whole model it is recommended using `saved_model`
- `saved_model` is a language-neutral, that enables higher-level systems and tools to produce, consume, and transform TensorFlow models
- `saved_model.loader` can load & restore pre-existing models
- `saved_model.loader.load(sess,[tag_constants.TRAINING], export_dir)` where
  * `sess` is the session in which the graph definition & variables are restored to
  * `tag_constants.TRAINING` are the tags used to identify the MetaGraphDef to load
  * `export_dir` is the location (directory) of the SavedModel

• Testing it two ways:
- `tensor_name.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels})`
- `sess.run(tensor_name, feed_dict={x: mnist.test.images,y_: mnist.test.labels})`

• Reading data:
- there are three main methods of getting data into a TensorFlow program
  1) through `feed_dict` as described in `Session()`
  2) reading from files, where an input pipeline reads the data from files at the beginning of a TF graph
  3) for small data sets can use preloaded data, where a constant or variable in the TF graph holds all the data

• Reading from files:
- `tf.train.string_input_producer()`
  * pass the list of filenames to the `tf.train.string_input_producer()` function
  * `string_input_producer` creates a FIFO queue for holding the filenames until the reader needs them
  `filename_queue = tf.train.string_input_producer(["file0.csv", "file1.csv"])`
- readers:
  * readers convert string work-units into records
  * *work-units* are typically filenames, while *records* are (key, value) pairs extracted from the contents of those files
    · the key would be the associated filename
  * to read text files use `TextLineReader()` operation
    ```
    reader = tf.TextLineReader()                    #constructor
    ```

  `key, value = reader.read(filename_queue)`
    * to convert comma-separated value (CSV) format to tensors, use `tf.decode_csv()` operation
      `tf.decode_csv(value, record_defaults)`
      · `record_defaults` is a list of Tensor objects with types from `float32`, `int32`, `int64`, `string`
      · returns a list of Tensor objects with same type as `record_defaults`
    * `tf.train.shuffle_batch()` creates a batch by randomly shuffling tensors

V. Image Processing

- `tf.image.resize_images()`
  - `resize_images(images,size)`
  - `resized = tf.image.resize_images(x, (227, 227))`

A. *MNIST*

- MNIST repository in TF:
  - MNIST data sits in the `~/MNIST_data/` directory,
    * `'train-images-idx3-ubyte.gz'` train
    * `'train-labels-idx1-ubyte.gz'` train
    * `'t10k-images-idx3-ubyte.gz'` test
    * `'t10k-labels-idx1-ubyte.gz'` test
- `input_data.py`
  - `input_data.py` imports `read_data_sets()` function from `mnist.py` file
    * `input_data.py` also includes multiple `import` commands
  - `input_data.py` is in directory `~/tensorflow/tensorflow/examples/tutorials/mnist`
  - first load the program
    `from tensorflow.examples.tutorials.mnist import input_data`
  - `mnist.py` is in directory
    `~/tensorflow/tensorflow/contrib/learn/python/learn/datasets`
- `read_data_sets()`
  - to execute `read_data_sets()`, type
    `mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)`
  - the complete command is as follows
    `read_data_sets(train_dir,fake_data=False,one_hot=False, dtype=dtypes.float32, reshape=True,validation_size=5000)`
- `Datasets` & `Dataset`
  - `mnist` is a user defined `collections.namedtuple()` called `Datasets`, that contains three instantiations of a class called `Dataset`,
    * `Datasets.train`
    * `Datasets.test`
    * `Datasets.validate`
  - `Dataset`
    * the `Dataset` class contains two NumPy arrays: `images` and `labels` both of type `numpy.ndarray`,
    * each image is of size 784,
    * each label is of size 10.
    * `Dataset` class includes a method that can iterate through mini-batches,
      `batch_xs, batch_ys = mnist.train.next_batch(batch_size)`

VI. Keras

- *Keras* is a higher level wrapper around Tensorflow and other tools
- Models:
  - there are two types of models available in Keras:
    * the *Sequential model* class and
    * the *Model* class used with functional API
- `Sequential`

- `from keras.models import Sequential`
- `model = Sequential()`
  - the `keras.models.Sequential` class is a wrapper for the neural network model
- the class `Sequential()` provides common functions like
  - `hist=model.fit(X,y,nb_epoch=10,validation_split=0.2)`
  - `model.compile('adam', 'categorical_crossentropy', ['accuracy'])`
  - `model.compile(loss='mse',optimizer='adam')`
  - `model.fit(X,y,nb_epoch=1,batch_size=8,validation_data=(Xv,yv), shuffle=True)`
    - `h = fit()` returns a history object, where `h.history` contains `loss` and `val_loss` as keys,
    - `h.history['loss']` & `h.history['val_loss']` can be plotted as a function of epochs
  - `mode.evaluate(x, y, batch_size=32, verbose=1, sample_weight=None)`
  - `model.add()` adds a layer, see next
  - `model.save(fn)`

- Layers:
  - `from keras.layers.core import Dense, Activation, Flatten, Dropout`
  - `from keras.layers import Input, Lambda`
  - to add input to model type
    `model.add(Flatten(input_shape=(32, 32, 3)))`
  - to normalize input, type
    `model.add(Lambda(lambda x: (x / 255.0) - 0.5))`
    - more generally, `Lambda` layers can be used to create arbitrary functions that operate on each input, as it passes through the layer
  - to add a second FC layer, type
    `model.add(Dense(100))`
  - to add various activations, type
    `model.add(Activation('relu'))`
    `model.add(Activation('softmax'))`
  - to add dropout, type
    `model.add(Dropout(0.5))`
  - Keras infers the shape of subsequent layers after the first layer
- Convolution & pooling:
  - `from keras.layers.convolutional import Convolution2D`
  - `from keras.layers.pooling import MaxPooling2D`
  - `model.add(Convolution2D(32, 3, 3, input_shape=(32, 32, 3)))`
    where `32` are the features, and `3,3` is the kernel
  - `model.add(MaxPooling2D((2,2)))`
- Datasets:
  `from keras.datasets import cifar10`
  `(X_train, y_train), (X_test, y_test) = cifar10.load_data()`
- `Model`
  - `from keras.models import Model`
  - `from keras.layers import Input, Dense`
  - `a = Input(shape=724,)`
  - `b = Dense(32)(a)`
  - `model = Model(inputs=a, outputs=b)`
- Callback:
  - a `callback` is a set of functions to be applied at given stages of the training procedure
  - can use callbacks to get a view on internal states and statistics of the model during training
  - can pass a list of callbacks (as the keyword argument `callbacks`) to the `.fit()` method of the `Sequential` or `Model` classes
  - the relevant methods of the callbacks will then be called at each stage of the training
  - `ModelCheckpoint()` saves the model after every epoch
    `keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', save_best_only=False, save_weights_only=False, period=1)`
  - `EarlyStopping()` stops training when a monitored quantity has stopped improving
    `keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0)`