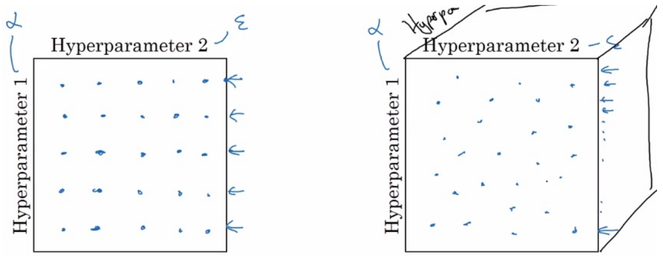
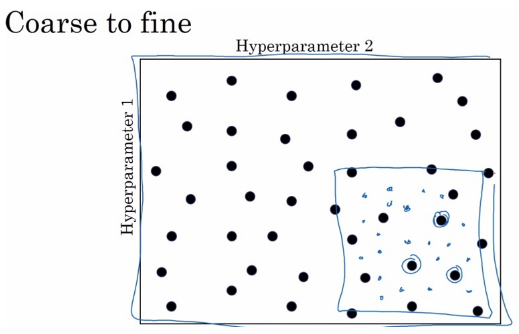
II-Improve DNN - L3

- Hyperparameter tuning
  - Tuning process
    - The importance of hp (hyperparameters) - Ng's opinion
      - most IM
        - learning_rate
      - second most IM
        - momentum β
        - #hidden units
        - mini-batch size
      - third most IM
        - #hidden layers
        - learning_rate decay
      - use default
        - Adam parameters: β1, β2, e (0.9, 0.999, 10^-8)
    - how to try out the hp
      - random instead of grid
        - random gives more try for each of the hp



      - coarse to fine
        - shrink the range of potentials



- Using an appropriate scale to pick hyperparameters
  - A
- Hyperparameters tuning in practice: Pandas vs. Caviar
  - A

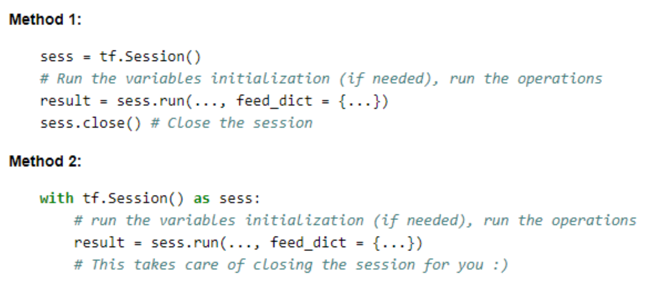
- Batch Normalization
  - Normalizing activations in a network
    - A
  - Fitting Batch Norm into a neural network
    - A
  - Why does Batch Norm work?
    - A
  - Batch Norm at test time
    - A

- Multi-class classification
  - Softmax Regression
    - A
  - Training a softmax classifier
    - A

- Introduction to programming frameworks
  - Deep learning frameworks
    - Why framework
      - to code in scratch is to understand the detail about the algorithm, to know how the benethe goes
      - but for large applications and complex NNs, better to use framework in order to make use of their efficient computation and algorithm.

        - popular frameworks
            - Caffe/Caffe2, CNTK, DL4J, Keras, Lasagne, mxnet, PaddlePaddle, TensorFlow, Theano, Torch
            - to do: research for their pros & cons
        - choosing criteria
            - ease of programming both for development & deployment
            - running speed
            - truly open - open source with good governance
                - not maintained by single company or group
            - objective
                - application: computer vision, NLP, online advertising....
                - language
    - TensorFlow
        - basic structure:
            - **Writing and running programs in TensorFlow has the following steps:**
                - **Create Tensors (variables) that are not yet executed/evaluated.**
                - **Write operations between those Tensors.**
                    - **to construct the computation graph, no computation here.**
                - **Initialize your Tensors.**

        - **Create a Session.**



Method 1:

```
sess = tf.Session()
# Run the variables initialization (if needed), run the operations
result = sess.run(..., feed_dict = {...})
sess.close() # Close the session
```

Method 2:

```
with tf.Session() as sess:
    # run the variables initialization (if needed), run the operations
    result = sess.run(..., feed_dict = {...})
    # This takes care of closing the session for you :)
```

        - **Run the Session. This will run the operations you'd written above.**
            - **only this step executes the computation**

- e.g.
    - w = tf.Variable(value, dtype)
        - specify the training variables
    - `x = tf.placeholder(tf.float32, [3,1])`
        - `specify a variable whose value can be assigned later which is convenient to feed data later e.g. mini_batches`
        - `to feed the data, use feed_dict={x:coefficients, ...}`
    - cost = #function()
        - define the cost function
    - train = tf.train.XXXOptimizer(learning_rate).minimize(cost)
        - *define the optimization algorithm*
    - *init = tf.global_variables_initializer()*
    - *sess = tf.Session()*
    - *sess.run(init)*
        - *all has this*
        - **to start a session, another better way**
            - **with tf.Session() as sess**
                - **sess.run(init)**

- **print(sess.run(train)**

- `sess.run(train, feed_dict={x:coefficients})` #run the optimization once
  - to run more, put in into a for loop
- `print(sess.run(w))`

- basic comments
  - the cost function is the key which build the **computation graph**, e.g. forward prop
  - there's no need to worry about back prob, since TF takes care of it automatically.

- Ex - TF
  - package
    - h5py
    - matplotlib
    - from .. import
  - TensorFlow - basic
    - y = tf.constant(36, name='y')
      - tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)
      - related
        - tf.zeros(shape, dtype=tf.float32, name=None)
        - tf.range(start, limit=None, delta=1, dtype=None, name='range')
        - tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)
    - loss = tf.Variable((y - y_hat)**2, name='loss')
      - meaning

        - A variable maintains state in the graph across calls to `run()`. You add a variable to the graph by constructing an instance of the class `Variable`.
    - implement
        - The `Variable()` constructor requires an initial value for the variable
        - The initial value defines and fixes the type and shape of the variable
        - The value can be changed using one of the assign methods
        - to change the shape of a variable later you have to use an `assign` Op with `validate_shape=False, e.g. w.assign(w + 1.0)`
        - `global_variables()` returns the contents of the graph collection `GraphKeys.GLOBAL_VARIABLES` which automatically collects variables into the graph
- program in TF
    - create tensors #that are not yet executed/evaluated.
    - build operation among tensors
    - initialize tensor

    - create a session
    - run the operations inside the session # evaluate the value of all variables
- placeholder
    - tf.placeholder(dtype, shape=None, name=None)
    - A placeholder is an object whose value you can specify only later. To specify values for a placeholder, you can pass in values by using a "feed dictionary" (feed_dict = {var:x,...} variable).
- Variable vs. placeholder
    - property
        - varable: fixed shape & type; initialization;
        - placeholder: flexible shape & type; no ;
    - function
        - varable: to store state in the graph; initialization;
            - build the computation graph
        - placeholder: to input external data; no ;
            - pass parameter to function
            - pass training data
- tf.one_hot(labels, depth, axis)
    - Conversion: single number to vector. e.g. 3 => [0,0,1,0...]
- tf.ones(shape, dtype = tf.float32, name = None)
    - tf.zeros()

        - tf.ones_like(tensor, dtype = None, name = None, optimize=True)
            - Given a single tensor (`tensor`), this operation returns a tensor of the same type and shape as `tensor` with all elements set to 1.
    - COST
        - tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = Z3, labels = Y))
            - the "`logits`" and "`labels`" of `tf.nn.softmax_cross_entropy_with_logits` are expected to be of shape **(number of examples, num_classes)**.
- numpy
    - b = a.reshape(a.shape[0],-1) #keep the first dim, the rest are flattened to be a vector.
        - a = np.random.randn(3,4,2,2)
        - b.shape = (3,16)
    - train = train.astype(np.float32)
        - **ndarray.astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
        - Copy of the array, cast to a specified type. Return a array with specific type