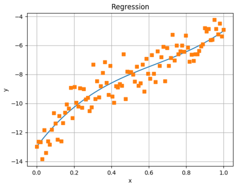
THE UNIVERSITY OF CHICAGO | Research Computing Center

Introduction to Deep Learning with TensorFlow

Igor Yakushin
ivy2@uchicago.edu

August 18, 2017

Supervised learning: regression



- Suppose we are trying to fit N measurements (X_i, Y_i) - **regression**.
- We can search for a solution, for example, in the form of

$$y(x, n, a) = \sum_{j=0}^{j=n} a_j x^j \tag{1}$$

by minimizing loss function with respect to a_j

$$loss(a) = \frac{1}{N} \sum_{i=1}^{i=N} (y(X_i, n, a) - Y_i)^2 \tag{2}$$

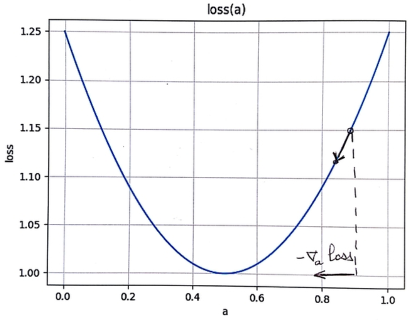
- Learning to generalize from a set of pairs (**features**, **labels**) - **supervised learning**

Supervised learning: gradient descent

- A straightforward way to minimize the **loss function** is to solve

$$\frac{\partial loss}{\partial a_i} = 0. \qquad (3)$$

- It might not be practical:
  - data might not fit into memory
  - the equations might be too complicated to solve analytically



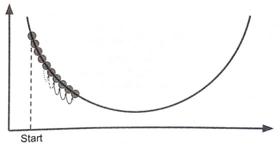
Instead - **(batch) gradient descent method**.

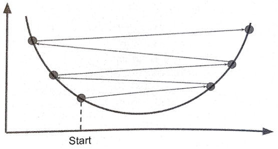
$$\Delta\vec{a} = -\lambda\nabla_a(loss) \qquad (4)$$

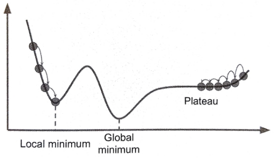
where λ - **learning rate**.

Supervised learning: learning rate

- If learning rate is too small
  - the convergence to minimum might be too slow
  - the algorithm might get stuck at local minimum
- If learning rate is too large, the algorithm might never converge but keep jumping around the minimum
- Learning rate is one of the metaparameters that needs to be determined experimentally. Or perhaps, use some kind of adaptive learning rate.







Supervised learning: gradient descent with momentum

- **Gradient descent with momentum** is a modification of the vanilla gradient descent method, which sometimes might produce better results
- The weight change is determined not only by the current value of gradient but also by the previous change of weights (called inertia).
- The exact mixture of gradient and inertia contribution is determined by yet another metaparameter ν:

$$\Delta\vec{a}(t+1) = -(1-\nu)\lambda\nabla_a(loss) + \nu\Delta\vec{a}(t) \tag{5}$$

- The physics analogy justifying the introduction of the inertia term: a ball rolling down a hill selects where it goes not only based on the gradient at the point where it is but also based on its velocity and mass - momentum.

Supervised learning: stochastic and mini-batch gradient descent

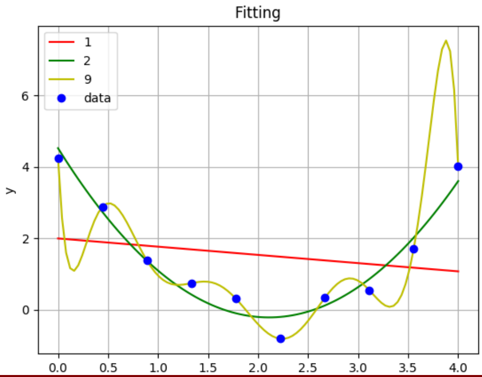
- The main problem with batch gradient descent is that it is using the whole data set to compute the gradients at every step:
  - Data might not fit into memory
  - It is very slow to train the model for big data sets
- In **stochastic gradient descent** a random instance in the training set is chosen at each step
  - Faster
  - Bounces a lot, less likely to settle on the local minimum
  - One can use huge data sets
- **Mini-batch gradient descent** is in between the batch and stochastic: at each step select some batch of samples.
  - It is usually faster then SGD due to hardware optimization of matrix operations: vectorization & multithreading

Supervised learning: inference

- Once one learns good parameters a_i for the model, one can do **inference/prediction**: the resulting model is used to find y for x that was not in the training set

Supervised learning: overfitting

- Red line - **underfitting**
- Yellow curve - **overfitting**, given data, no reason to believe in such complicated fit
- Green curve - just right
- Also real data often has noise and the fit should smooth it out



Supervised learning: regularization

- How can we avoid overfitting?
- We can try to simplify the fitting function and look at the fit but it is not usually practical since typically there are many dimensions. We need a way to do it automatically.
- One way is to introduce the **regularization** term into the loss function that would penalize large values of parameters. For example:

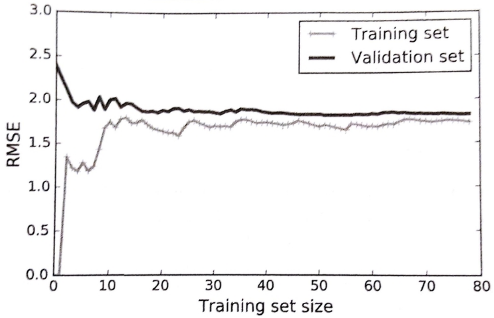
$$loss(a) = \frac{1}{N}\sum_{i=1}^{i=N}(y(X_i, n, a) - Y_i)^2 + \mu \sum_{i=1}^{i=N} a_i^2 \qquad (6)$$

- μ - regularization factor - is yet another metaparameter to tune
- The larger is μ, the simpler the fit would be

Supervised learning: training, testing, validation sets

- How can one judge automatically if overfitting is happening? How can one tune learning rate, penalty factor, other metaparameters?
- To solve these problems the data set is divided into 2-3 parts:
  - **Training set** - the fit parameters a_i are learned on that set
  - **Validation set** - we use it to judge the fit during the training
  - **Testing set** - once the fit is trained well on training set according to tests on validation we make the final judgment on testing set. We can also use it to compare the effects of different metaparameter choices.
- A reasonable split of data, for example: 70% - training set, 10% - testing set, 20% - validation set
- Often validation and testing sets are lumped together

Supervised learning: Learning curves



- For small training sets, the fit is perfect on training set and bad on validation set
- As training set becomes bigger, learning curves approach each other
- If there is a gap between them - poor generalization, possibly overfitting, getting more training samples or simplifying the fit function might help
- If the curves close to each other but at high error - underfitting, consider more complex fitting function, getting more training samples would not help
- We want both curves converging to each other at low error.

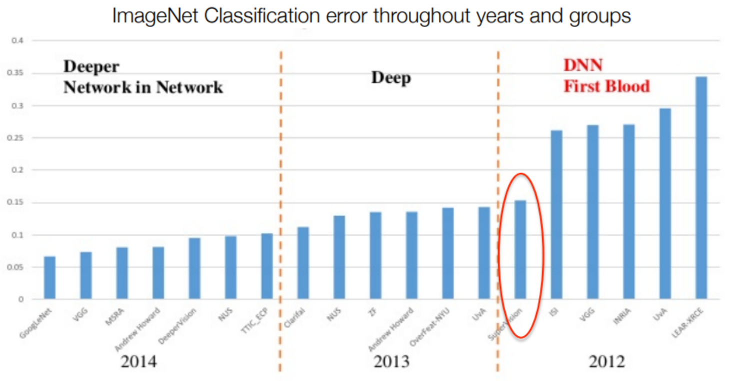
Supervised learning: classification

- Besides regression, another typical problem that can be solved by supervised learning is **classification**:
    - The labels are no longer continuous values but take a few discrete values
        - For example: classify the given pictures into cats and dogs
- Similar approach, as described above, can be used to classification problem

Neural Networks: motivation

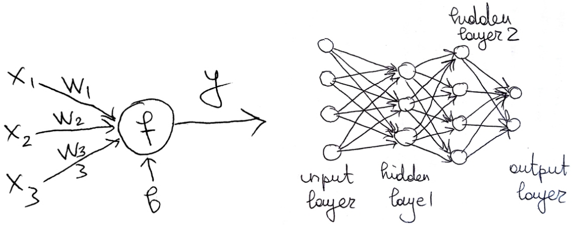
- How does one select the form of the fitting function?
  - Sometimes one has a model and just needs to learn a few parameters.
  - Popular universal fitting functions are **Neural Networks** (NN).
- Any function can be approximated with the desired precision by NN
- NNs were originally inspired by our limited understanding of how the human brain works.
- Until recently the available computational power was not enough to make NNs practical to use. The revolution started in 2012 when a deep convolution network was combined with GPU to beat the best image recognition algorithms by a huge margin.
- NNs were rebranded under the name of **Deep Learning** (DL) since the most useful networks these days have of the order of $10 - 100$ layers.

Neural Networks: motivation



Li Fei-Fei: ImageNet Large Scale Visual Recognition Challenge, 2014 http://image-net.org/

Neural Networks: structure



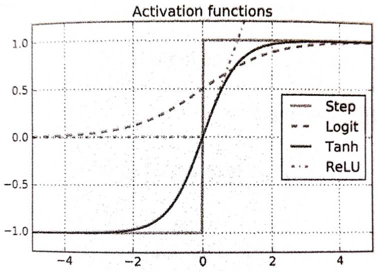
- NN consists of nodes.
- Each node might have many inputs x_i and an output y that can go to many other nodes.
- Inputs are linearly combined with weights W_i and bias b and some non-linear activation function f is applied to produce the output: $y = f(\sum_{i=1}^{i=N} W_i x_i + b)$.
- Nodes are grouped into layers, layers - into NN.
- Training NN means minimizing the loss with respect to weights and biases.

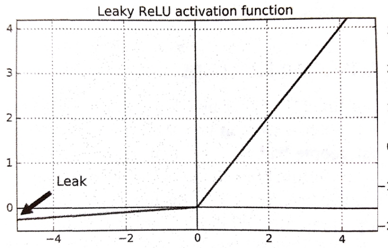
Neural Networks: activation function

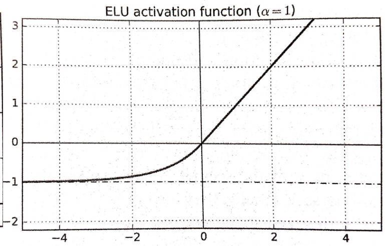
- There are several popular activation functions used in NN's nodes:
  - Originally, in 1940s, step functions were used
  - $\frac{1}{1+e^{-x}}$ - **sigmoid** (also called **logistic function**);
    - used to be very popular in the past;
    - slow training near 0,1
  - $tanh(x)$ - similar to sigmoid
  - $ReLU(x)$ - **rectified linear function**
    - currently very popular
    - often faster training than sigmoid
  - Leaky $ReLU(x)$ - small slope for negative x to avoid zeroes
  - Exponential Linear Unit (ELU) - smooth version of ReLU:

$$ELU_{\alpha}(x) = \begin{cases} \alpha(exp(x) - 1), & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases} \quad (7)$$

Neural Networks: activation function





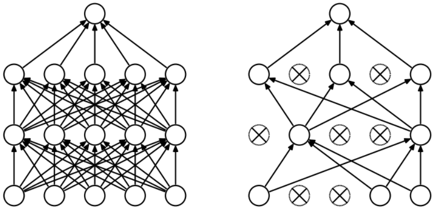


Neural Networks: backpropagation

- The loss function is measured on the output layer
- We need to compute derivatives of this loss with respect to all the weights
- When the network is layered, the **backpropagation** method, derived using chain rule for differentiation, can provide some shortcut in computing those derivatives
- First one computes derivatives with respect to weights on the links connecting the output layer M with the previous layer $M-1$
- Then, there is a recursive formula that relates the derivatives on $(M-2, M-1)$ weights to the derivatives on $(M-1, M)$ weights
- One keeps backpropagating derivatives until $(1, 2)$ links are processed
- When using such high level frameworks as TensorFlow, one does not need to know the details of backpropagation method, just select one of the available optimizers

Neural Networks: Regularization with dropout

- Another regularization method, besides penaly term, specific to neural networks - **dropout**
- During training, at each iteration turn off some nodes with certain probability trying to identify the ones unimportant for the performance
- During inference the full network is used but the weights for each node that was subject to dropout is multiplied by the corresponding dropout probability - effectively combining models
- “It prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently.”



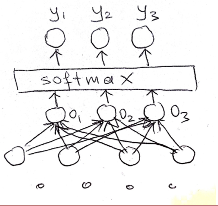
Neural networks: softmax

$$\bigcirc \to \begin{bmatrix}1\\0\end{bmatrix} \qquad \triangle \to \begin{bmatrix}0\\1\end{bmatrix}$$

- Suppose we are using NN to solve classification problem and there are N possible classes.
- Let us use **one-hot encoding**: each class i is represented by δ_{ij}
- As a last layer of N outputs, **softmax** is used:

$$y_i = \frac{e^{o_i}}{\sum_j e^{o_j}} \tag{8}$$

- The last layer gives probabilities for classes



Neural networks: cross entropy

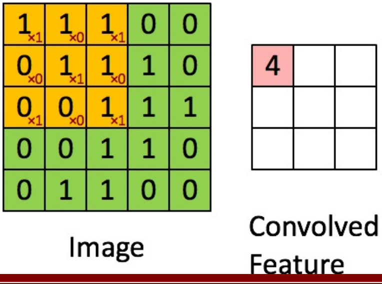
- For classification, with softmax, one usually uses a **cross entropy**, rather than a mean square of errors, as a loss function:

$$loss(a) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{n} Y_j^i log(y(X_j^i, a)) \tag{9}$$

  where i runs over measurements and j runs over outputs
  - For each measurement i only one output j with non-zero label $Y_j = 1$ contributes to the loss
  - If the corresponding probability softmax $y(X_j^i, a)$ is close to 1 - agrees well with the label Y_j - it does not contribute to the loss ($log(1) = 0$)
  - The closer the probability is to 0, the more it disagrees with the label and the more it contributes to the loss

Neural networks: convolution

- Fully connected NNs:
  - each unit of layer M is connected to all the units of layer $M+1$
  - very expensive to train for large input size
  - do not take into account space locality of input in, for example, images
  - do not take into account that features learned on one part of the image are often applicable to other parts
- Possible solution: allow units in layer $M+1$ to connect to only a small contiguous region of units in the in layer M

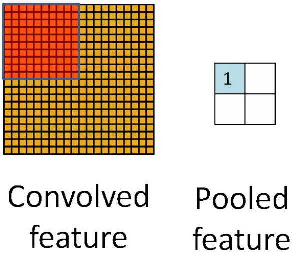


Neural networks: convolution

- The **convolution** filter moves over the layer with a certain stride in x and y directions, typically 1, applying the same filter with trainable weights over and over again
- To preserve the dimensions of the original layer, one might want to pad the boundaries with 0s
- Convolution is inspired by how neurons in the visual cortex have localized receptive fields: they respond only to stimuli in a certain location.
- Typically we want to learn many features of layer M and therefore apply N convolution filters with different weights resulting in N convolutional layers connected to layer M

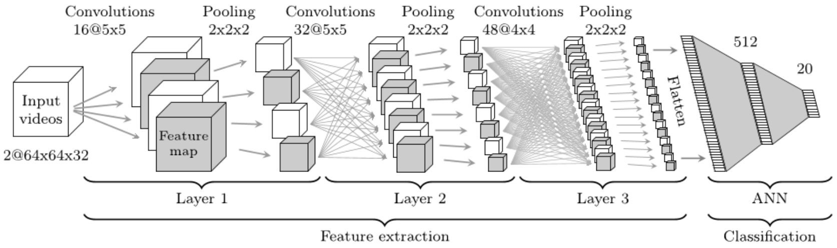
Neural networks: pooling

- After obtaining features using convolution, we would next like to use them for classification
- However, it might be too computationally expensive and some downsampling is desired
- **Pooling**: divide the convoloved feature into disjoint $m \times n$ adjacent regions and compute some statistics on it: min, max, mean, sum, etc
- Use the resulting downsampled data to feed to softmax or other convolution-pooling layers



Neural networks: convolution networks

- **Convolutional Neural Network** (CNN) is comprised of one or more convolutional layers usually followed by pooling, it might be also followed by fully connected layer and softmax
- The architecture of a CNN is designed to take advantage of the 2D structure of an input image
- CNNs are easier to train and have much fewer parameters than fully connected networks with the same number of hidden units.
- Often, for practical problems CNNs are quite deep: that's the origin of the term **Deep Learning**



TensorFlow

- Open source library by Google for numerical computation using data flow graphs
  - Nodes - operations
  - Edges - multidimensional data arrays (tensors)
- Typically one uses python interface but APIs are also available in: C++, Java, Go.
- What makes it efficient and convenient: Python is not used for heavy computations but to construct an expression tree that is executed outside of Python in C++ backend
- Can utilize multiple CPU cores, GPUs, nodes
- Installation is as simple as `'pip install ...'` provided you have sufficiently modern Linux, otherwise - use Singularity container
- One of the most popular frameworks for DL but can also be used for any numeric computations
- Current release version: 1.2.1
- Has tensorboard monitoring and graph visualization GUI and debugger

TensorFlow: how to use on midway

- midway 2, GPU
  - Get into gpu2 partition
    - `sinteractive -p gpu2 --gres=gpu:1 --time=01:00:00`
  - and load one of these pythons:
    - `module load Anaconda2 cuda/8.0`
    - `module load Anaconda3 cuda/8.0`
- midway 2, CPU only
  - `sinteractive -p broadwl --time=01:00:00`
  - `module load python_ucs4/2.7.12`
- midway 1, GPU
  - `sinteractive -p gpu --gres=gpu:1 --time=01:00:00`
  - `module load singularity`
  - `singularity shell /software/src/singularity_images/tensorflow_1.1.0.img`
  - `source /usr/local/nvidia.sh 352.55`
  - `/usr/local/Anaconda2/bin/python` or `/usr/local/Anaconda3/bin/python`

TensorFlow: Lab 1: Hello world!

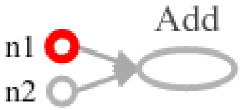
```
import tensorflow as tf

node1 = tf.constant(3.0, dtype=tf.float32, name='n1')
node2 = tf.constant(4.0, name='n2')
node3 = tf.add(node1, node2)
print(node1, node2, node3)
sess = tf.Session()
print(sess.run([node1, node2]))
print("sess.run(node3): ",sess.run(node3))
file_writer = tf.summary.FileWriter('./l1', sess.graph)
file_writer.close()
```

TensorFlow: Lab 1: Hello World! Tensorboard

```
cd
rsync -av /project/rcc/workshops/Introduction_to_Deep_Learning_with_TensorFlow .
sinteractive -p gpu2 --time=01:30:00 --reservation=TensorFlowWorkshop --gres=gpu:1
module load Anaconda2 cuda/8.0
cd Introduction_to_Deep_Learning_with_TensorFlow/lab1
python l1.py
tensorboard --logdir=l1 --host=<ip or hostname> --port=<port number if not default>
```

- You can figure out the host IP with `'ifconfig -a'`
- There is a default port (6006) on which **tensorboard** listens
- Once started, tensorboard will print out URL to point your browser to; if you are on uchicago network or VPN, you can point the browser on your laptop to this IP, otherwise, run the browser on the node itself
- Since there are 4 GPUs per node, there might be conflict with several tensorboards connecting to the same port on the same host, try incrementing it if it happens
- tensorboard is optional: it just helps to understand what's going on

TensorFlow: Lab 1: Hello world!



- In particular, tensorboard can be used to visualize the expression graph
- When you execute `l1.py`, among various housekeeping messages, you would see the outputs from prints:

```
Tensor("n1:0", shape=(), dtype=float32) Tensor("n2:0", shape=(), dtype=float32) \
  Tensor("Add:0", shape=(), dtype=float32)
[3.0, 4.0]
sess.run(node3):  7.0
```

- Nodes are evaluated only in session as you can see from the above output

TensorFlow: Lab 2: placeholders

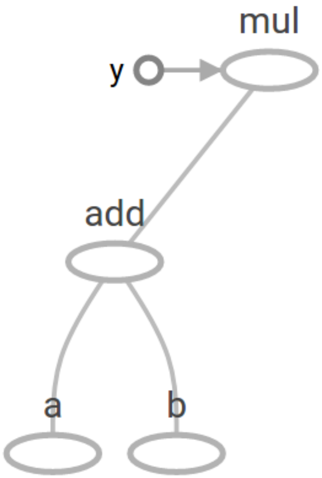
- Instead of using constants, we can use placeholders to be later substituted by any data via dictionary:

```
a = tf.placeholder(tf.float32, name='a')
b = tf.placeholder(tf.float32, name='b')
adder_node = a + b
add_and_triple = adder_node * 3.

sess = tf.Session()
print(sess.run(adder_node, {a: 3, b:4.5}))
print(sess.run(adder_node, {a: [1,3], b: [2, 4]}))
print(sess.run(add_and_triple, {a: 3, b:4.5}))
```

- Output:

```
7.5
[ 3.  7.]
22.5
```

TensorFlow: Lab 2: placeholders



TensorFlow: Lab 3: Variable

- To be able to modify the tensors as the program runs, for example, as a result of learning, *Variable* is used
- Variables might have initial values: one must initialize them at the beginning of the session
- To modify a variable, *assign* is used

TensorFlow: Lab 3: Variable

```
W = tf.Variable([.3], dtype=tf.float32, name='W')
b = tf.Variable([-.3], dtype=tf.float32, name='b')
x = tf.placeholder(tf.float32, name='x')
linear_model = W * x + b
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
print(sess.run(linear_model, {x:[1,2,3,4]}))
y = tf.placeholder(tf.float32, name='y')
squared_deltas = tf.square(linear_model - y)
loss = tf.reduce_sum(squared_deltas)
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))
fixW = tf.assign(W, [-1.])
fixb = tf.assign(b, [1.])
sess.run([fixW, fixb])
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))
```

TensorFlow: Lab 3: Variable

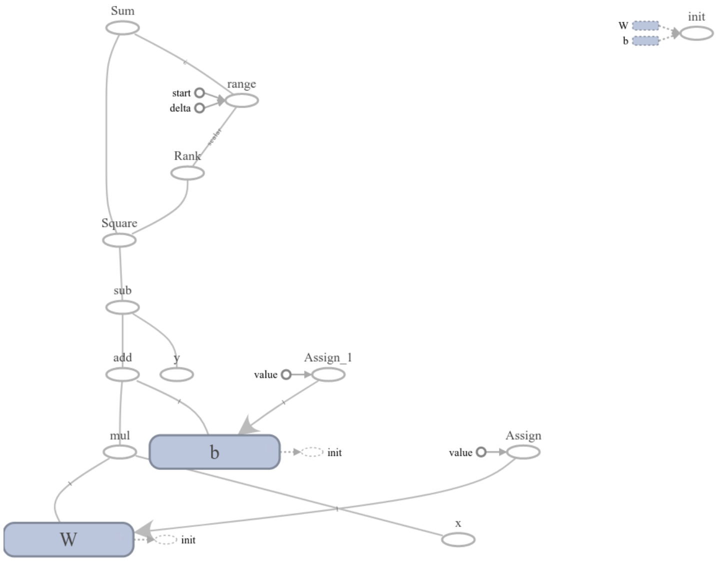
- Output:

```
<tf.Variable 'W:0' shape=(1,) dtype=float32_ref>
<tf.Variable 'b:0' shape=(1,) dtype=float32_ref>
Tensor("add:0", dtype=float32)
[ 0.  0.30000001  0.60000002  0.90000004]
23.66
0.0
```

- The expression graph is getting quite complicated
- One can use scopes to group tensors together, create a hierarchical structure in order to make the graph more readable

```
with tf.variable_scope("conv1"):
        # Variables created here will be named "conv1/weights", "conv1/biases".
        relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
```

TensorFlow: Lab 3: Variable



TensorFlow: Lab 4: Training linear model

- In lab 3 we manually found optimal values for the weights to minimize the loss function
- In this lab we are going to let the model learn the optimal weights
- To do that we use GradientDescentOptimizer with learning rate 0.01 and ask it to minimize the loss function for the given training data:

  ```
  optimizer = tf.train.GradientDescentOptimizer(0.01)
  train = optimizer.minimize(loss)
  ```
- We iterate 1000 times, making one optimization step per iteration.
- We start utilizing tensorboard to monitor the evolution of various variables every 100 iterations

TensorFlow: Lab 4: Training linear model

```
import numpy as np
import tensorflow as tf
W = tf.Variable([.3], dtype=tf.float32, name='W')
b = tf.Variable([-.3], dtype=tf.float32, name='b')
x = tf.placeholder(tf.float32, name='x')
linear_model = W * x + b
y = tf.placeholder(tf.float32, name='y')
loss = tf.reduce_sum(tf.square(linear_model - y))
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

TensorFlow: Lab 4: Training linear model

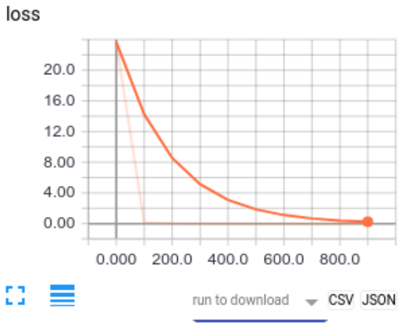
```
file_writer = tf.summary.FileWriter('./l4')
file_writer.add_graph(sess.graph)
tf.summary.scalar('loss', loss)
tf.summary.scalar('W',W[0])
tf.summary.scalar('b',b[0])
merged = tf.summary.merge_all()
for i in range(1000):
      WW,bb,l,m,_ = sess.run([W,b,loss,merged,train],
                             {x:x_train, y:y_train})
      if(i%100==0):
            file_writer.add_summary(m,i)
            print("%s\t%s\t%s\t%s"%(i,WW,bb,l))
curr_W, curr_b, curr_loss = sess.run([W, b, loss],
                             {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
file_writer.close()
```

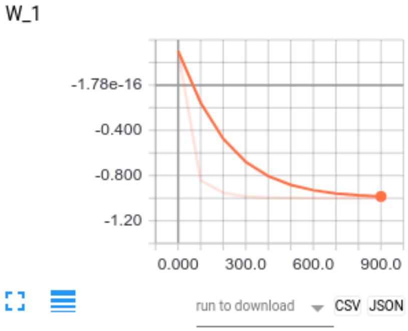
TensorFlow: Lab 4: Training linear model

- Ouput:

```
0       [ 0.30000001]   [-0.30000001]   23.66
100     [-0.84079814]   [ 0.53192717]   0.146364
200     [-0.95227844]   [ 0.85969269]   0.0131513
300     [-0.98569524]   [ 0.95794225]   0.00118168
400     [-0.99571204]   [ 0.98739296]   0.000106178
500     [-0.99871469]   [ 0.99622095]   9.54086e-06
600     [-0.99961472]   [ 0.99886727]   8.57163e-07
700     [-0.99988455]   [ 0.99966055]   7.69487e-08
800     [-0.99996537]   [ 0.99989825]   6.90848e-09
900     [-0.99998957]   [ 0.99996936]   6.24471e-10
W: [-0.9999969] b: [ 0.99999082] loss: 5.69997e-11
```

TensorFlow: Lab 4: Training linear model





TensorFlow: Lab 5: tf.contrib.learn

- TensorFlow's tf.contrib.learn library significantly simplies the training:

```
tf.logging.set_verbosity(tf.logging.INFO)
import numpy as np
features = [tf.contrib.layers.real_valued_column("x", dimension=1)]
estimator = tf.contrib.learn.LinearRegressor(feature_columns=features, model_dir="l5",
                          config=tf.contrib.learn.RunConfig(save_checkpoints_secs=1))
x_train = np.array([1., 2., 3., 4.])
y_train = np.array([0., -1., -2., -3.])
x_eval = np.array([2., 5., 8., 1.])
y_eval = np.array([-1.01, -4.1, -7, 0.])
input_fn = tf.contrib.learn.io.numpy_input_fn({"x":x_train}, y_train,
                                        batch_size=4,num_epochs=1000)
eval_input_fn = tf.contrib.learn.io.numpy_input_fn({"x":x_eval}, y_eval,
                                        batch_size=4, num_epochs=1000)
estimator.fit(input_fn=input_fn, steps=1000)
train_loss = estimator.evaluate(input_fn=input_fn)
eval_loss = estimator.evaluate(input_fn=eval_input_fn)
print("train loss: %r"% train_loss)
print("eval loss: %r"% eval_loss)
```

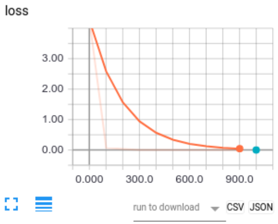
- Notice that here we are using separate training and validation sets

TensorFlow: Lab 5: tf.contrib.learn

- `tf.logging.set_verbosity` generates to stdout:

```
INFO:tensorflow:loss = 4.25, step = 1
INFO:tensorflow:global_step/sec: 819.655
...
INFO:tensorflow:loss = 2.46991e-06, step = 901 (0.148 sec)
INFO:tensorflow:Saving checkpoints for 1000 into t5/model.ckpt.
INFO:tensorflow:Loss for final step: 6.28759e-07.
```

- Also notice that the tensorboard events are generated automatically as well



- Another high level framework that can use either TensorFlow or Theano as backends is Keras.

TensorFlow: Lab 6: Handwriting recognition using fully connected NN

- MNIST data set:
  - handwritten numbers, 28x28 pixels, and the corresponding labels
  - 55,000 entries - training set, 10,000 - test set, 5,000 - validation set
- NN:
  - First layer: flattened 28x28 image - a vector of 784 entries
  - Second layer: 10 outputs that go into softmax
  - One-hot encoding of labels
  - Loss function - cross entropy

TensorFlow: Lab 6: Handwriting recognition using fully connected NN

```
mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(FLAGS.learning_rate).minimize(cross_entropy)
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy',accuracy)
tf.summary.scalar('cross entropy', cross_entropy)
tf.summary.histogram('b',b)
tf.summary.histogram('W',W)
tf.summary.image('W',tf.reshape(W, [1,784,10,1]))
merged = tf.summary.merge_all()
file_writer_test = tf.summary.FileWriter('./mnist1/test', sess.graph)
file_writer_train = tf.summary.FileWriter('./mnist1/train', sess.graph)
```

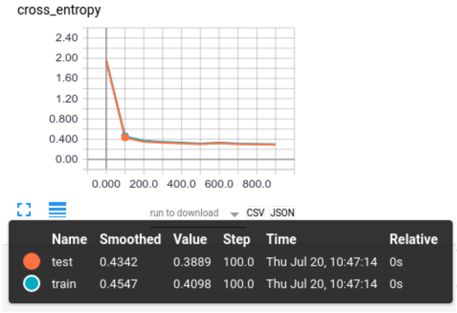
TensorFlow: Lab 6: Handwriting recognition using fully connected NN

```
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
  if(i%100==0):
    summary_test  = sess.run(merged, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
    summary_train = sess.run(merged, feed_dict={x: mnist.train.images, y_: mnist.train.labels})
    file_writer_test.add_summary(summary_test,i)
    file_writer_train.add_summary(summary_train,i)
print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                    y_: mnist.test.labels}))
file_writer_test.close()
file_writer_train.close()
```

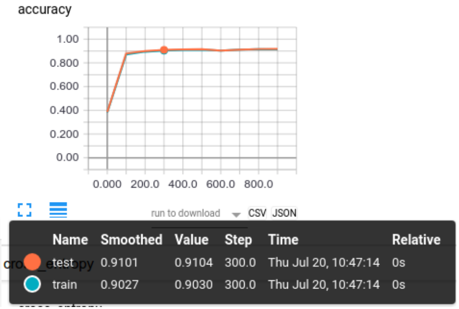
- Notice that we created two file writers so that we can compare variables on training and testing sets. Using separate file writers is also convenient way to compare the effect of changing metaparameters.
- Also we create an image for W.
- Vanilla softmax computation might be numerically unstable, so instead raw unnormalized inputs are used and later averaged over measurements

TensorFlow: Lab 6: Handwriting recognition using fully connected NN

- The achieved accuracy is 92%
- Point tensorboard to the root directory `mnist1` rather than subdirectories to compare data on training and testing sets: `tensorboard --logdir=mnist1`
- Accuracy and entropy on both data sets are very close



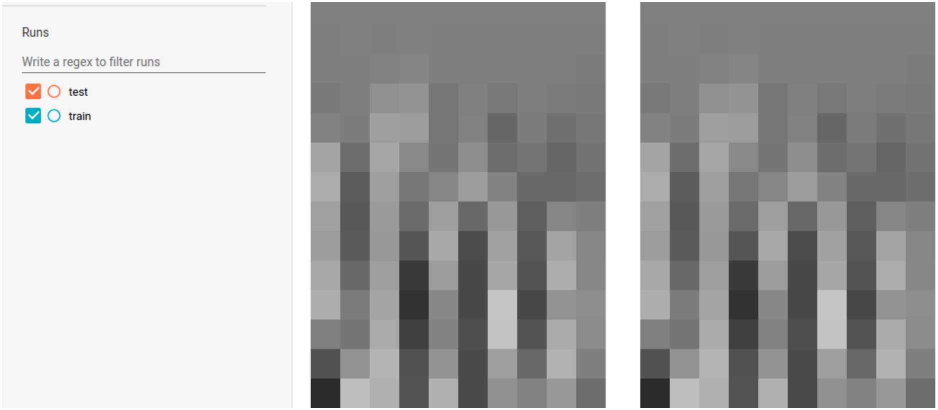
	Name	Smoothed	Value	Step	Time	Relative
●	test	0.4342	0.3889	100.0	Thu Jul 20, 10:47:14	0s
●	train	0.4547	0.4098	100.0	Thu Jul 20, 10:47:14	0s



	Name	Smoothed	Value	Step	Time	Relative
●	test	0.9101	0.9104	300.0	Thu Jul 20, 10:47:14	0s
●	train	0.9027	0.9030	300.0	Thu Jul 20, 10:47:14	0s

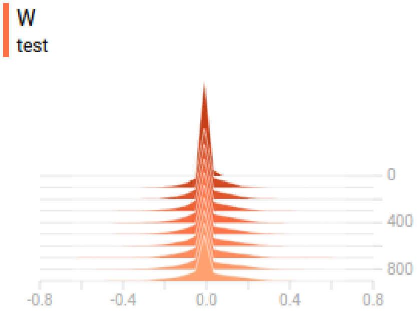
TensorFlow: Lab 6: Handwriting recognition using fully connected NN

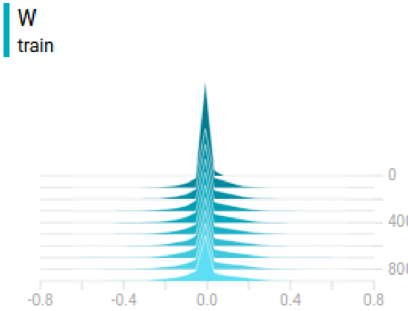
- A part of W matrix on both data sets:



TensorFlow: Lab 6: Handwriting recognition using fully connected NN

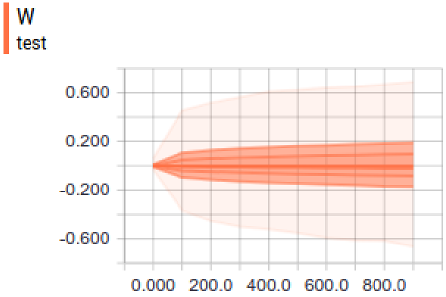
- Histograms:

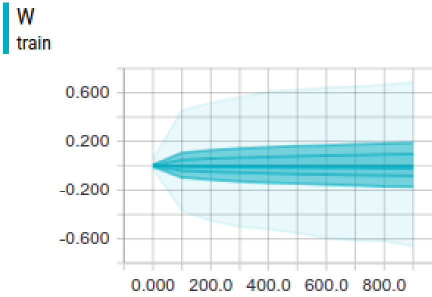




TensorFlow: Lab 6: Handwriting recognition using fully connected NN

- Distributions (generated if we ask for histograms):





TensorFlow: Lab 7: Handwriting recognition using Deep Learning

- Let us solve the same problem but finally use Deep Learning and achieve better accuracy
- The input layer is 2D: 28x28
- The first convolution layer uses 5x5 filters, ReLU activation functions and generates 32 feature maps; since padding is used, each map has the same 28x28 dimensions as input
- The feature maps are downsampled by 2x2 max pool reducing their dimensionality to 14x14
- The second convolution layer again uses 5x5 filters and converts 32 14x14 feature maps into 64
- Another 2x2 max pooling turns it into 64 7x7 feature maps
- They are reshaped into 64x7x7 vector which is fully connected to the next layer with 1024 nodes and ReLU activation function
- Dropout regularization with $p = 0.5$ is used on this layer

TensorFlow: Lab 7: Handwriting recognition using Deep Learning

- Then there is another fully connected layer with 10 nodes that are input to softmax
- The weights W are initialized with truncated normal distribution and the biases b are initially set to 0.1
- AdamOptimizer with the learning rate of $1e-4$ is used
- The training runs significantly longer then in Lab 6.
- The achieved accuracy is 99.3%

TensorFlow: Lab 7: Handwriting recognition using Deep Learning

```
def deepnn(x):
  """deepnn builds the graph for a deep net for classifying digits.

  Args:
    x: an input tensor with the dimensions (N_examples, 784), where 784 is the
    number of pixels in a standard MNIST image.

  Returns:
    A tuple (y, keep_prob). y is a tensor of shape (N_examples, 10), with values
    equal to the logits of classifying the digit into one of 10 classes (the
    digits 0-9). keep_prob is a scalar placeholder for the probability of
    dropout.
  """
  # Reshape to use within a convolutional neural net.
  # Last dimension is for "features" - there is only one here, since images are
  # grayscale -- it would be 3 for an RGB image, 4 for RGBA, etc.
  x_image = tf.reshape(x, [-1, 28, 28, 1])

  W_conv1 = weight_variable([5, 5, 1, 32])
  b_conv1 = bias_variable([32])
  h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

  h_pool1 = max_pool_2x2(h_conv1)
```

TensorFlow: Lab 7: Handwriting recognition using Deep Learning

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
return y_conv, keep_prob
```

TensorFlow: Lab 7: Handwriting recognition using Deep Learning

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def main(_):
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
  x = tf.placeholder(tf.float32, [None, 784])
  y_ = tf.placeholder(tf.float32, [None, 10])
  y_conv, keep_prob = deepnn(x)

  cross_entropy = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
  train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
  correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

TensorFlow: Lab 7: Handwriting recognition using Deep Learning

```
  config = tf.ConfigProto()
  config.gpu_options.per_process_gpu_memory_fraction=0.9

  with tf.Session(config=config) as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(20000):
      batch = mnist.train.next_batch(50)
      if i % 100 == 0:
        train_accuracy = accuracy.eval(feed_dict={
            x: batch[0], y_: batch[1], keep_prob: 1.0})
        print('step %d, training accuracy %g' % (i, train_accuracy))
      train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

    print('test accuracy %g' % accuracy.eval(feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str,
                      default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

TensorFlow: Benchmarking using Lab 7 code

Hardware	Video card	(V)RAM (G)	Time (s)	accuracy
Dell XPS 15	GTX 960M	2	OOM	N/A
gpu2 partition	Tesla K80	12	221	99.3%
gpu2 partition, Singularity	Tesla K80	12	326	99.2%
gpu2 partition, Singularity, dl	Tesla K80	12	258	99.2%
CPP partition	Tesla P100	16	132	99.2%
gpu partition, Singularity	Tesla K40	12	397	99.1%
gpu partition, Singularity	Tesla K20	5	OOM	N/A
gpu partition, Singularity	Tesla M2090	6	6162	99.3%
broadwl CPU, 28 cores	N/A	64	1308	99.2%

TensorFlow: Saving models

```
theta = tf.Variable([-3], name="theta")
init = tf.global_variables_initializer()
# create a Saver node at the end
# of the construction phase:
saver = tf.train.Saver()
with tf.Session() as sess:
   sess.run(init)
   for epoch in range(n_epochs):
      # Save checkpoint with all
      # the variables and graph structure:
      save_path = saver.save(sess, "/tmp/my_model.ckpt")
   sess.run(training_op)
   best_theta = theta.eval()
   # Save final checkpoint
   save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

TensorFlow: Restoring models

```
# restore the graph structure
saver = tf.train.import_meta_graph("/tmp/
                                my_model_final.ckpt.meta")
with tf.Session() as sess:
    #populate the graph values
    saver.restore(sess, "/tmp/my_model_final.ckpt")
```

- To train your own network, it is often useful to start with the pretrained network that does similar task
  - For example, you need to train a network to disginguish dogs and cats.
  - Download a deep convolutional network that classifies cars, freeze most of the layers that are closer to inputs and train only the last few layers
  - The layers closer to input usually pick up more simple features, for example, line orientation, while the layers closer to the output assemble out of those features more complicated ones, for example a certain combination of lines.
- The working example of save/restore is given in Lab 9.

TensorFlow: Running on multiple GPUs and CPU cores

- Install `tensorflow-gpu` - it supports both GPUs and CPUs, while `tensorflow` supports only CPUs.
- `CUDA_VISIBLE_DEVICES=2,3 python program.py`
  - Can be used to run different TF processes on different GPUs
- By default, TF grabs all the VRAM in all available GPUs. To restrict the amount of RAM:

  ```
  config = tf.ConfigProto()
  config.gpu_options.per_process_gpu_memory_fraction = 0.4
  session = tf.Session(config=config)
  ```

  - Can be used to run multiple TF processes on the same GPU
- When runnig on CPU, TF grabs all the cores

TensorFlow: Running on multiple GPUs and CPU cores

Simple placement:

```
config = tf.ConfigProto()
config.log_device_placement = True
with tf.device("/gpu:1"):
    a = tf.Variable(3.0, name = 'a')
    b = tf.constant(4.0, name = 'b')
    c = a + b
with tf.device("/cpu:0"):
    d = tf.Variable(5.0, name='d')
    e = tf.constant(7.0, name='e')
    f = d*e
with tf.device("/gpu:3"):
    h = tf.Variable(15.0, name='h')
    g = tf.constant(97.0, name='g')
    k = h/g
sess = tf.Session(config=config)
init = tf.global_variables_initializer()
sess.run(init)
print(sess.run([a,b,c]))
print(sess.run([f]))
print(sess.run([k]))
```

- see Lab 8.

Dynamic placement:

```
def variables_on_cpu(op):
   if op.type == "Variable":
      return "/cpu:0"
   else:
      return "/gpu:0"

with tf.device(variables_on_cpu):
   a = tf.Variable(3.0)
   b = tf.constant(4.0)
   c = a * b
```

Soft vs hard placement:

- Not all kernels can run on all devices:
    - no GPU kernel for integer variables
    - exception - hard placement

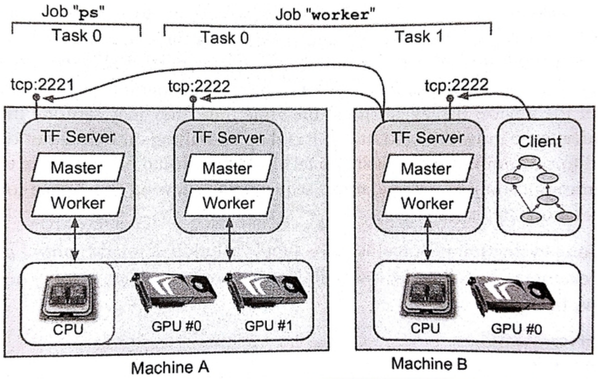
```
config.allow_soft_placement = True
```

- fall back on CPU

TensorFlow: Running on multiple nodes

To run a graph across multiple servers - define a **cluster**

- Cluster is composed of TF servers - **tasks**
- Each task belongs to a **job** - a named group of tasks that have a common role



TensorFlow: Running on multiple nodes

```
cluster_spec = tf.train.ClusterSpec({
  "ps": [
     "machine-a.example.com:2221", #/job:ps/task:0
   ],
  "worker": [
     "machine-a.example.com:2222", #/job:worker/task:0
     "machine-b.example.com:2222", #/job:worker/task:1
   ]})
```

- To start a TensorFlow server, you must create a `Server` object, passing it the cluster specification and its own job name and task number:

```
server = tf.train.Server(cluster_spec, job_name="worker",
                         task_index=0)
server.join()
```

TensorFlow: Running on multiple nodes

- Once all the tasks are up and running (doing nothing yet), you can open a session on any of the servers and use that session as a regular local session

```
a = tf.constant(1.0)
b = a + 2
c = a * 3
with tf.Session("grpc://machine-b.example.com:2222") as sess:
  print(c.eval())
```

- You can use `tf.device` to pin operations on any device in a cluster:

```
with tf.device("/job:ps/task:0/cpu:0")
   a = tf.constant(1.0)
with tf.device("/job:worker/task:0/gpu:1")
   b = a + 2
c = a + b
```

TensorFlow: Running on multiple nodes

- Another possibility to do distributed programming in TF is to use queues
  - Queue graphs to execute by one task,
  - Populate them with data by the second task,
  - Dequeue and execute by the third task, etc.
- TF can share the state of variables across tasks
- One can impose constraints on the order in which operations are executed
- Therefore, TF can be used to make a fast (C/C++ performance) general purpose parallel program in Python that can use multiple CPU cores, GPUs and be distributed across the nodes. The question is: does TF have kernels for your particular problem.

TensorFlow: miscellaneous tips

- If you get OOM (Out of memory) error:
  - From a different terminal ssh to the same node where you run interactive session and use `'nvidia-smi -l'` to monitor memory consumption as your program runs. It also tells you how much memory each GPU card has, how many GPU cards are in the host
  - You can also try to run on CPU (possibly on bigmem2 node) and monitor memory consumption with `'top'` again from a different terminal session. You might have to request more RAM with '-mem' option to sinteractive since by default you would only get 2G
- `'nvidia-smi -l'` would also give you an idea how heavily GPU is utilized, `~100%` is ideal. Otherwise, your program might be spending most of the time moving data between CPU and GPU

Some Deep Learning applications

- Image recognition
- Object Classification in Photographs
- Automatic Machine Translation
- Image Caption Generation
- Voice recognition
- Fraud detection
- Transferring style from famous paintings
- Sentiment analysis
- Motion detection
- Restore colors in B&W photos and videos
- Game playing using reinforcement deep-learning
- Self-driving cars
- Music composition

Other software for DL

- **Theano** - Python API, converts expression tree into C program and compiles it into native CPU or GPU code on the fly, grand-daddy of deep-learning frameworks, similar to TensorFlow, popular in academia
- **Keras** - high level framework to construct and train NNs using either TensorFlow or Theano as a backend, Python API
- **Torch** - Lua API, used by Facebook
- **Caffe**, **Caffe2** - Python API, used by Facebook
- **CNTK** is Microsoft's open-source deep-learning framework, Python API
- **MxNet** is a machine-learning framework with APIs is languages such as R, Python and Julia which has been adopted by Amazon Web Services
- **Intel Deep Learning Training Tool** (formerly known as Intel Deep Learning SDK): web GUI to Intel optimized TensorFlow, Theano, Caffe... **Intel Nervana Neon** - Python API
- Mathematica, Matlab, ...

Hardware for DL

- Couple years ago I attended NVIDIA conference. At least 30% of the conference was about DL since these days it generates a lot of business for NVIDIA.
  - NVIDIA Tesla V100, P100 cards
  - NVIDIA DGX - appliance with 8 latest and greatest V100, NVlink, containers
  - Power9 from IBM - several times (5x?) faster connection between CPU and GPU then for a typical Intel CPU (except for DGX?)
- Intel is also building special chips for DL: Lake Crest (due this year), Knights Crest, Knights Mill ...
- Google recently released TPU (Tensor Processing Unit): special purpose chip used just for predictions, fast, low power, the training is still done on GPUs

Conclusion

- In this talk we have introduced concepts from Supervised Learning, Neural Networks, Deep Learning and and showed how to use TensorFlow
- Other DL-related topics where there was a tremendous progress recently and that we had no time to discuss:
    - Recurrent Neural Networks - time series prediction, text and music composition...
    - Autoencoders - dimensionality reduction, language translation, unsupervised pretraining of DL classification networks, ...
    - Reinforcement learning - game-playing, self-driving cars...

TensorFlow: References

- "Building Machine Learning Projects with TensorFlow" by Rodolfo Bonnin
- "Hands-On Machine Learning with Scikit-Learn & TensorFlow" by Aurelien Geron
- `https://www.tensorflow.org/` - original TensorFlow documentation, tutorials, software
- `http://ufldl.stanford.edu/tutorial/` - DL tutorial from Stanford
- `http://www.image-net.org/` - ImageNet
- `http://yann.lecun.com/exdb/mnist/` - MNIST data
- `http://playground.tensorflow.org` - configure and play with neural network online
- `http://scs.ryerson.ca/~aharley/vis/conv/` - visualization of convolutional network to recognize hand-written digits
- `https://github.com/tensorflow/models/tree/master/slim` - example of pretrained networks