Introduction to Deep Learning with Tensorflow

Yunhao (Robin) Tang
Department of IEOR, Columbia University

February 12, 2018

Summary

What we will cover...

- Simple Example: Linear Regression with Tensorflow
- Intro to Deep Learning
- Deep Learning with Tensorflow
- Beyond

If time permits, we will also cover

- OpenAI Gym
- Implement basic DQN

Info about Tensorflow

- TF is an open source project formally supported by Google.
- Programming interface: Python, C++, C... Python is most well developed and documented.
- Latest version: 1.5. Commonly used version: 1.0. (depend on other dependencies)
- Can be installed via pip. Best supported on mac os and linux.

Simple Example: Linear Regression

- Linear Regression: N data points $\{x_i, y_i\}_{i=1}^N$, $x \in \mathbb{R}^k$, $y \in \mathbb{R}$
- **Specify Architecture:** Consider linear function for regression with parameter: slope $\theta \in \mathbb{R}$ and bias $\theta_0 \in \mathbb{R}$. The prediction is

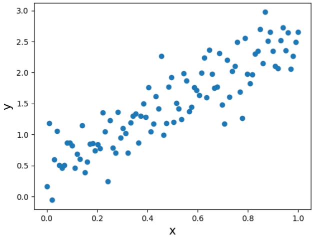
$$\hat{y}_i = \theta^T x_i + \theta_0$$

- **Define Loss:** Loss function to minimize $J = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
- **Gradient Descent:** $\theta \leftarrow \theta - \alpha \nabla_\theta J, \theta_0 \leftarrow \theta_0 - \alpha \nabla_{\theta_0} J$

Linear Regression: Data

```
# true parameters
true_theta = 2.
true_theta_0 = .5

# generate data
x = np.linspace(0,1,100)
y = true_theta * x + true_theta_0 + np.random.randn(x.size) * .3
```

(a) True Parameters



(b) Data

Linear Regression with Tensorflow

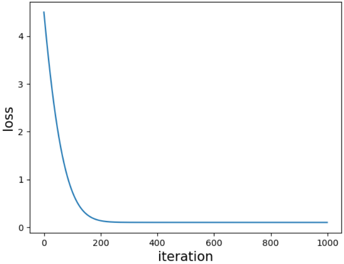
Let us just focus on defining **architecture**...

```
# specify placeholders
X = tf.placeholder(tf.float32,[None])
Y = tf.placeholder(tf.float32,[None])

# specify models
theta = tf.Variable(tf.truncated_normal([1]))
theta_0 = tf.Variable(tf.truncated_normal([1]))
Y_hat = tf.multiply(X,theta) + theta_0

# loss
loss = tf.reduce_mean(tf.square(Y_hat - Y))
```

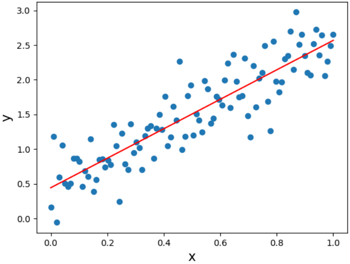
(c) Linear regression tf code

Linear Regression with Tensorflow

We launch the training...



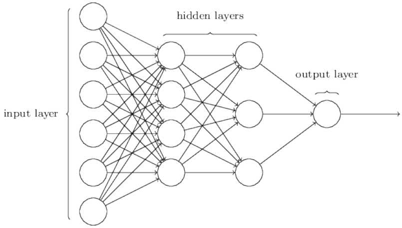
(d) loss



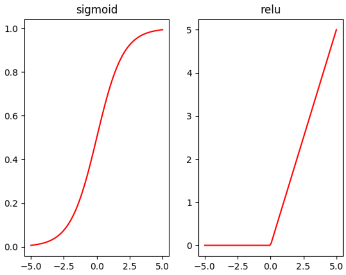
(e) fitted line

Intro to DL

- Consider data sets with complex relation $\{x_i, y_i\}_{i=1}^N$, linear regression will not work.
- **MLP (Multi Layer Perceptron):** stack multiple layers of linear transformations and nonlinear functions.
  - input $x \in \mathbb{R}^6$, first layer linear transformation $W_1 x + b_1 \in \mathbb{R}^4$, then apply nonlinear function $h_1 = \sigma(W_1 x + b_1) \in \mathbb{R}^4$. Nonlinear function applies elementwise.
  - second layer has W_2, b_2, transform as $h_2 = \sigma(W_2 h_2 + b_2) \in \mathbb{R}^3$
  - final output $\hat{y} = W_3 \sigma(W_2 \sigma(W_1 x + b_1) + b_2) + b_3 \in \mathbb{R}$



(f) mlp



(g) nonlinear functions

Intro to DL

- **Architecture:** define a complex input to output relation with stacked layers of linear transformations and nonlinear functions (activations).
  - Architectures are adapted to data structure at hand.
  - Input prior knowledge into modeling through architecture design.
  - MLP, CNN (image), RNN (sequence data, audio, language...
- Parameters: weights W_1, W_2, W_3 and bias b_1, b_2, b_3.
  - Linear regression: slope θ and bias θ_0

DL algorithm

- **Specify Architecture:** how many layers, how many hidden unit per layer? $\hat{y}_i = f_\theta(x_i)$ with generic parameters θ (weights and biases).
- **Define Loss:** Loss function to minimize $J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- **Gradient Descent:** $\theta \leftarrow \theta - \alpha\nabla_\theta J, \theta_0 \leftarrow \theta_0 - \alpha\nabla_{\theta_0} J$

Algorithm 1 Generic DL regression

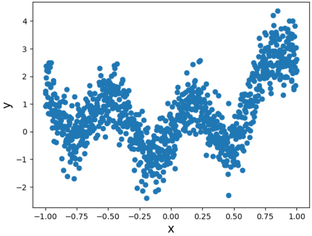
Input: model architecture, data $\{x_i, y_i\}$, learning rate α
Initialize: Model parameters $\theta = \{W_i, b_i\}$
for t=1,2,3...T **do**
    Compute prediction $\hat{y}_i = f_\theta(x_i)$
    Compute loss $J = \frac{1}{N}\sum_i(\hat{y}_i - y_i)^2$
    Gradient update $\theta \leftarrow \theta - \alpha\nabla_\theta J$
end for

DL Regression: Data

Generate data using $y = x^3 + x^2 + \sin(10x) + \textbf{noise}$. Linear regression will fail.

```
# generate data
x = np.linspace(-1,1,1000)
y = x**3 + x**2 * 2 + np.sin(10 * x) + np.random.randn(x.size) * .6
```

(h) data generation



(i) data plots

DL Regression with Tensorflow

Specify the **Architecture, Loss** using Tensorflow syntax just like linear regression.

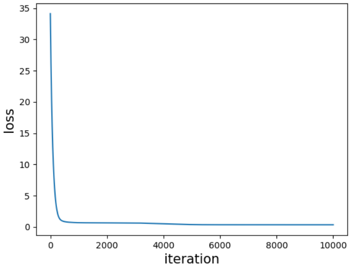
```
# specify placeholders
X = tf.placeholder(tf.float32,[None,1])
Y = tf.placeholder(tf.float32,[None,1])

# specify models
W1 = tf.Variable(tf.truncated_normal([1,10]))  # layer 1
b1 = tf.Variable(tf.truncated_normal([10]))
W2 = tf.Variable(tf.truncated_normal([10,10]))  # layer 2
b2 = tf.Variable(tf.truncated_normal([10]))
W3 = tf.Variable(tf.truncated_normal([10,1]))  # layer 3
b3 = tf.Variable(tf.truncated_normal([1]))

h1 = tf.nn.relu(tf.matmul(X,W1) + b1)
h2 = tf.nn.relu(tf.matmul(h1,W2) + b2)
Y_hat = tf.matmul(h2,W3) + b3

# loss
loss = tf.reduce_mean(tf.square(Y_hat - Y))
```

(j) Linear regression tf code

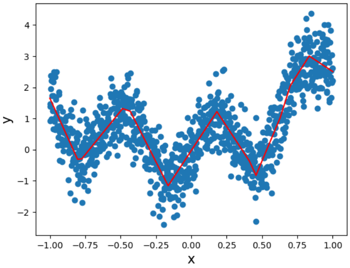
- train the model just like linear regression

DL Regression with Tensorflow

We launch the training...



(k) loss



(l) fitted line

Batch training

- When data set is large, can only compute gradients on mini-batches.
- Sample a batch of data $\rightarrow$ Compute gradient on the batch $\rightarrow$ Update parameters.
- **Batch size:** too small: high variance in data stream, unstable training; too large: too costly to compute gradients.

Auto-differentiation

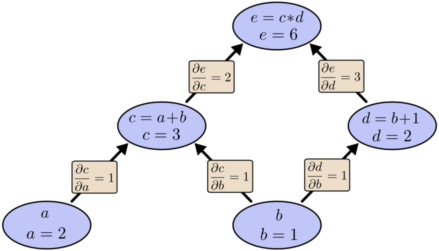
- Taking gradients in linear model is easy.
- Question: how to take gradients for MLP? $\nabla_{W_1} J$ is not straightforward as in linear regression.

$$J = \frac{1}{N}\sum_i (y_i - \hat{y}_i)^2, \hat{y}_i = W_3\sigma(W_2\sigma(W_1 x_i + b_1) + b_2) + b_3$$

Auto-differentiation

- Tensorflow entails automatic differentiation (autodiff), i.e. automatic gradient computation. Users specify forward computation $y = f_\theta(x)$, the program will internally specify a way to compute $\nabla_\theta y$. Not symbolic computation, not finite difference
- Example: start with a, b, compute e: $c = a + b, d = b + 1, e = c \cdot d$. Consider

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial a} + \frac{\partial e}{\partial c}\frac{\partial c}{\partial a}$$



(m) computation graph

DL with Tensorflow

- High level idea: computation graph. Forward computation specifies local operations that connect locally related variables. Local gradients can be computed.
- forward computation $\rightarrow$ local forward ops $\rightarrow$ local gradient $\rightarrow$ long range gradients.
- Just like error back-propagating from output to inputs and params, **back-propagation**.
- Useful link: http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial

DL with Tensorflow

What we have covered from the code...

- **Placeholder:** tf objects that specify users' inputs into the graph, as literally placeholders for data.
- **Variable:** tf objects that represent parameters that can be updated through autodiff. Embedded inside the graph.
- All expressions computed from primitive variables and placeholders are vertices in the graph.

DL with Tensorflow

What we have not yet covered from the code... how to launch the training? how to use the gradients computed by Tensorflow to update parameters?

- **TF expression:** python objects that define how to update the graph (variables in the graph) in a single step.
- **Optimizer:** python objects that can specify gradient updates of variables.
- **Session:** python objects that launch the training.
- And many more...

Launch the training

- Define **loss** and **optimizers**: which specifies how to update model parameters during training.
- Define **session** to launch the training
- Initialize variables
- Feed data into the computation graph

```
# define loss
loss = tf.reduce_mean(tf.square(Y_hat - Y))

# use optimizers to define update operations
optimizer = tf.train.AdamOptimizer(1e-2)
opt = optimizer.minimize(loss)

# initialize sessiont to run update
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())  # initialize all variables
    ldict = []  # a list that records loss
    for _ in range(1000):
        l,_ = sess.run([loss,opt],feed_dict={X:x,Y:y})  # one step gradient update
        ldict.append(l)  # record loss
```

(n) launch the training

Training: whole landscape

```
# ========
# define models
# ========
# specify placeholders
X = tf.placeholder(tf.float32,[None,1])
Y = tf.placeholder(tf.float32,[None,1])

# specify models
W1 = tf.Variable(tf.truncated_normal([1,10]))  # layer 1
b1 = tf.Variable(tf.truncated_normal([10]))
W2 = tf.Variable(tf.truncated_normal([10,10]))  # layer 2
b2 = tf.Variable(tf.truncated_normal([10]))
W3 = tf.Variable(tf.truncated_normal([10,1]))  # layer 3
b3 = tf.Variable(tf.truncated_normal([1]))

h1 = tf.nn.relu(tf.matmul(X,W1) + b1)
h2 = tf.nn.relu(tf.matmul(h1,W2) + b2)
Y_hat = tf.matmul(h2,W3) + b3

# loss
loss = tf.reduce_mean(tf.square(Y_hat - Y))

# gradients
optimizer = tf.train.AdamOptimizer(1e-3)
opt = optimizer.minimize(loss)

# ========
# launch training
# ========
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    ldict = []
    x = np.expand_dims(x,1)
    y = np.expand_dims(y,1)
    for _ in range(10000):
        l,_ = sess.run([loss,opt],feed_dict={X:x,Y:y})
        ldict.append(l)
```

(o) launch the training

Beyond Regression

- **Classification:** regression on probability (image classification)
- **Structured Prediction:** regression on high dimensional objects (autoencoders)
- **Reinforcement Learning:** DQN, policy gradients...
  - DQN is an analogy to regression
  - PG is an analogy to classification

Hyper-parameters

Get a DL model to work involves a lot of hyper-parameters.

- **Optimization:** optimization subroutines that update parameters with gradients.
  - beyond simple sgd, there are rmsprop, adam, adagrad... adam is popular.
  - learning rate
- **Architectures:** number of layers, hidden units per layers
- **Nonlinear function:** sigmoid, relu, tanh...
- **Initialization:** initialization of variables W, b.
  - arbitrary initializations will fail.
  - xavier initialization, truncated normal...
- Need hand tuning or cross validation to select good hyper-parameters.
- Other topics: batch-normalization, dropout, stochastic layers...

Beyond Tensorflow

- Other autodiff softwares: Pytorch/Chainer/Theano/Caffe
  - pytorch and chainer allow for dynamic graph building
  - others use static graph building
  - strengths/weaknesses: gpu, distributed computing, model building flexibility, debug...
- High level interface: Keras
  - use tf and theano as backend
  - specify model architecture more easily
- High level interface: stick with Tensorflow
  - tensorflow.contrib

Beyond Tensorflow

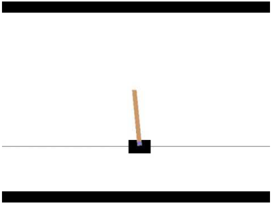
```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None,10])

# build layer
W1 = tf.Variable(tf.truncated_normal([3,10]))
b1 = tf.Variable(tf.truncated_normal([3]))

# compute output
y = tf.nn.relu(tf.matmul(X,W1) + b1)

# ======
# high level definition
# ======
import tensorflow.contrib as tc

# z is the new output
z = tf.layers.dense(x, 3, # 3 is the output dimension
    kernel_initializer=tf.random_uniform_initializer(minval=-3e-3, maxval=3e-3)
```

(p) low level vs. high level

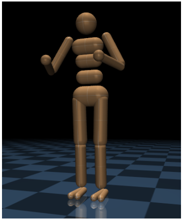
Additional Resources

- TF official website: https://www.tensorflow.org/
- TF Basic tutorial: https://www.tensorflow.org/tutorials/
- Keras doc: https://keras.io/
- TF tutorials: https://github.com/aymericdamien/TensorFlow-Examples

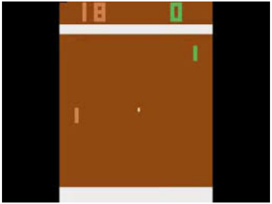
OpenAI Gym - very brief intro



(q) classic control



(r) humanoid



(s) atari game - pong

OpenAI Gym - very brief intro

- Gym is a testbed for RL algorithms. Doc: https://github.com/openai/gym
- **make an environment:** env = gym.make("CartPole-v0")
- **initialize:** obs = env.reset()
  - first **obs**, we take the first **action** based on this
- **take a step:** obs, reward, done, info = env.step(action)
  - based on **obs**, we make choices on which **action** to take
  - **reward** is the reward collected during this one step transition
  - **done** is True or False, to indicate if the episode terminates
  - **info** additional info about the env, normally empty dictionary
- **display state:** env.render(). Only display current state, need to render in a loop to display consecutive states.

Basic DQN [Mnih, 2013]

- MDP with state action space S, A
- Instant reward r
- Policy $\pi : S \mapsto A$
- Maximize cumulative reward $E_\pi[\sum_{t=0}^{\infty} r_t \gamma^t]$
- Action value function $Q^\pi(s, a)$ for state s, action a, under policy π.
- Bellman error
$$E[(Q^\pi(s_t, a_t) - \max_a E[r_t + \gamma Q^\pi(s_{t+1}, a))^2]$$
- Bellman error is zero iff optimal policy $Q^*(s, a)$

Basic DQN: Conventional Q learning

- Start with any Q vector $Q^{(0)}$
- Contractive operator T defined as

$$TQ(s_t, a_t) =_{def} \max_a E[r_t + \gamma Q(s_{t+1}, a)]$$

- Apply contractive operator

$$Q^{(t+1)} \leftarrow TQ^{(t)}$$

- WIll converge under mild conditions. Converge to the fixed point

$$Q = TQ$$

which gives zero bellman error.

Basic DQN

- Approximate using neural net $Q_\theta(s,a) \approx Q^\pi(s,a)$ with parameter θ.
- **Discrete action space** $|A| < \infty$: input state s, output $|A|$ values, the ith value represents the approximate action-value function for the ith action.
- **Continuous action space** $|A| = \infty$: input state s and action a, output one value which represents $Q^\pi(s,a)$. Not our focus here.
- Minimize Bellman error

$$E_\pi[(Q_\theta(s_i,a_i) - r_i - \max_a Q_\theta(s_i',a))^2]$$

- Sample based, given tuples $\{(s_i,a_i,r_i,s_i')\}_{i=1}^N$

$$\min_\theta \sum_{i=1}^N \frac{1}{N}(Q_\theta(s_i,a_i) - r_i - \max_a Q_\theta(s_i',a))^2$$

Basic DQN

- Instablity in optimizing

$$\min_{\theta} \sum_{i=1}^{N} \frac{1}{N} (Q_{\theta}(s_i, a_i) - r_i - \max_{a} Q_{\theta}(s_i', a))^2$$

- Two techniques to alleviate instability: **experience replay** and **target network**.
- **Experience Replay:** store experience tuple $\{s_i, a_i, r_i, s_i'\}$ into replay buffer B, when training, sample batches of experience $\{s_i, a_i, r_i, s_i'\}_{i=1}^{b}$ from B and update parameters using SGD
- **Target Network:** the training target $r_i + \max_a Q_{\theta}(s_i', a)$ is non-stationary. Make it stationary by introducing a slowly updated target net θ^- and compute target as $r_i + \max_a Q_{\theta^-}(s_i', a)$

Basic DQN

- Naive exploration $\epsilon-$greedy
  - in state s, with prob ϵ take action randomly; with prob $1-\epsilon$ take greedy action
$$\arg\max_a Q_\theta(s,a)$$
- More advanced exploration: noisy net, parameter space noise, bayesian updates...
- Learning rate: constant $\alpha = .001$ for example, not RM scheme.

Basic DQN

Algorithm 1 DQN

INPUT: target network update period τ, total number of episodes E, initial time steps before update init, learning rate α, exploration prob ϵ, batchsize for training N

INITIALIZE: DQN principal network $Q_\theta(s,a)$ with parameters θ, target network $Q_{\theta^-}(s,a)$ with parameters θ^-, time steps counter *counter* $\leftarrow 0$, empty buffer $R \leftarrow \{\}$

for $e = 1, 2, 3 \ldots E$ **do**

    while episode not terminated **do**

        Execute actions

        $counter \leftarrow counter + 1$

        Given state s_t, for prob ϵ, take action uniformly random; otherwise, take action by being greedy $a_t \leftarrow \arg\max_a Q_\theta(s_t, a)$

        Save experience tuple $\{s_t, a_t, r_t, s_{t+1}\}$ to buffer R

        Training θ by gradients

        Sample N tuples $\{s_i, a_i, r_i, s_i'\}$ from replay buffer R (uniformly)

        Compute target $d_j = r_j + \max_{a'} Q_{\theta_j^-}(s_j', a')$ for $1 \leq j \leq N$

        Compute empirical loss

$$L = \frac{1}{N} \sum_{j=1}^{N} (Q_\theta(s_j, a_j) - d_j)^2$$

        Update $\theta \leftarrow \theta - \alpha \nabla_\theta L$

        Update target network θ^-

        if *counter* mod $\tau = 0$ **then**

            Update target parameter $\theta^- \leftarrow \theta$

(t) dqn pseudocode

End

Thanks!