Notes de cours CADL - session-2

Thibaut Marmey

November 2, 2018

- The basic components of a neural network
- How to use gradient descent to optimize parameters of a neural network
- How to create a neural network for performing regression

Contents

1 Introduction

1.1 Generalities

- Use data and gradient descent to teach the network what the values of the parameters of the network should be.
- Idea of machine learning : letting the machine learn from the data.
- We are interested in letting the computer figure out what representations it needs in order to better describe the data and some objective we've defined.

1.2 Gradient Descent

- Operations of the network are meant to transform the input data into something meaningful that we want the network to learn about.
- Parameters of the NW are random so output is random as well.
- If we need specific output, we can use "Gradient Descent" : way to optimize set of parameters.

1.3 Defining Cost

- Mesure of the "error"
- Exemple : recognize apple or orange. Random network spit ou random 0s or 1s for apples and oranges. We can define :
  - if the network predicts a 0 for an orange, then the error is 0. If the network predicts a 1 for an orange, then the error is 1.
  - And vice-versa for apples. If it spits out a 1 for an apple, then the error is 0. If it spits out a 0 for an apple, then the error is 1.
- Defining error in terms of our parameters :

$$error = network(image) - true\_label \tag{1}$$

$$network(image) = predicted\_label \tag{2}$$

$$E = f(X) - y \tag{3}$$

1.4 Minimizing Error

- Feed the network many images (100 for e.g) to see what the network is doing on average.
- Changing network's parameters can have effect on the error.
- The error provides a "training signal" or a measure of the "loss" or our network.
- Assumptions in assuming our funtion is continuous and differentiable.
- Gradiant descent in a nutshell : "Error", "Cost", "Loss", or "Training Signal"

1.5 Backpropagation

- The gradient is just saying, how does the error changes at the current set of parameters.
- To figure out what is the gradian we use backpropagation. Whatever differences that output has with the output we wanted it to have, gets *backpropagated* to every single param in our network.
- Backprop is an effective way to find the gradient. Uses the *chain rule* to find the gradian of the error.
- $y = mx + b$ linear function. The slope or gradian is m.
- The process described :

$$\theta = \theta - \eta * \nabla_\theta * J(\theta) \tag{4}$$

  - θ : parameters
  - ∇ : gradient, with repect to parameters θ, ∇_θ
  - J : error
  - η : learning rate describes how far along this gradient we should travel, typically value between 0.01 to 0.00001

1.6 Local Minima/Optima and learning Rate

- Dilemma : find a local or global minima.
- The NW may have million of parameters, so the problem becomes more and more difficult.
- Wize choice of the learning rate :
  - To small we dont get any better cost where we started
  - To big the cost goes up and down

2 Creating a Neural Network

2.1 Exemple with sine wave

- Input X output y
- Here the input is values in an interval instead of images like before.
- The exemple is to create sine wave with uniform noise and create a neural network that is able to discover the sine wave.

```
# Create data : sine wave with random noise in the interval
n_obs = 1000
xs = np.linscape(-n, n, n_obs)
ys = np.sin(xs) + np.random.uniform(-0.5,0.5,n_obs)
plt.scatter(xs, ys)
```

- Train the NW to give any value on the x axis and have the value it should be on the y axis. (fundamental idea of regression : predicting some continuous output value given some continuous input value)

2.2 Defining cost

- Use of placeholder to define input and output values. Those variables will filled at the computation of the graph.

```
X = tf.placeholder(tf.float32, name='X')
Y = tf.placeholder(tf.float32, name='Y')
```

- Create session and define the parameters center and close to 0 with *tf.random_normal(nb_values, stddev=).eval()*

```
sess = tf.InteractiveSession()
n = tf.random_normal([1000], stddev=0.1).eval()
```

- Create **variables** using *tf.Variable*, it does need an initial value or we can call an initializer.
- Define two *tf.Variable* for weight and bias

```
W = tf.Variable(tf.random_normal([1], dtype=tf.float32, stddev=0.1),
    name='weight')
B = tf.Variable(tf.constant([0], dtype=tf.float32), name='bias')
# Scale input placholder X
Y_pred = X * W + B
```

- Use gradient descent to learn what the best value of W and b.
- Before that we have to know how to measure what the *best* mean for what we try to do.
- Let's define the absolute *distance(val1, val2)* from the predicted value to the assumed sine wave value.

```
cost = distance(Y_pred, tf.sin(X))
```

- We calculate the mean of the cost we have for every observation

```
cost = tf.reduce_mean(distance(Y_pred, tf.sin(X))
```

2.3 Training Parameters

- Use tensorflow optimizer

```
optimizer =
    tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost)
```

- Run the optimizer

```
with tf.Session() as sess:
  #initilization of all the tf.Variable
  sess.run(tf.global_varables_initializer())
  prev_training_cost = 0.0
  for it_i in range(n_iterations):
    sess.run(optimizer, feed_dict={X:xs, Y:ys})
    training_cost = sess.run(cost, feed_dict={X:xs}) #...,session=sess)

    # each 10 iterations
    if it_i % 10 == 0:
      ys_pred = Y_pred.eval(feed_dict={X:xs})
      # print stuff like training_cost and plots

    if np.abs(prev_training_cost - training_cost < 0.000001):
      break # if local minima found

    prev_trianing_cost = training_cost
```

- The output doesn't look like a sine wave at all. Actually it's only a line.
- Training vs Testing : Have to learn more about the different between training and testing networks.

2.4 Stochastic and Mini Batch Gradient Descent

- Tricks to find the best local minima : using *mini-batches* of size *batch_size*.

```
idxs = np.arrange(100) # it will be changed to make it random
batch_size = 10
n_batches = idxs // batch_size
```

- Look some random subset of the datase because neural networks love order and would use it to its advantage. But order is irrelevant to our problem.

```
idxs = np.random.permutation(idxs)
```

- Generalise the entire dataset. We modify the code which runs the optomizer to include the mini-batches program.

```
idxs = np.random.permutation(range(len(xs)))
n_batches = len(idxs) // batch_size
for batch_i in range(n_batches):
  idxs_i = idxs[batch_i * batch_size: (batch_i+1) * batch_size]
  sess.run(optimizer, feed_dict={X:xs[idxs_i], Y:ys[idxs_i]})
training_cost = sess.run(cost, feed_dict={X:xs, Y:ys})
```

We get a better result : the line has a curve but it doesn't look like a sine wave.

- This method is :
  - Mini-batches : small pieces of data where we perform gradient descent
  - Stochastic : the order of the data is dandomized
- Use a function of training

```
def train(X, Y, Y_pred, n_iterations=100, batch_size=200,
    learning_rate=0.02):
```

- To get better result we can have a bigger set of parameters.
- We are going to multiply our input by 100 values, creating an "inner layer" of 100 neurons.
  - Define *tf.Variables* : Weights (multiplication) and biais (addition)

```
n_neurons = 100
W = tf.Variable(tf.random_normal([1,n_neurons], stddev=0.1))
b = tf.Variable(tf.constant(0), dtype=tf.float32, shape=[n_neurons])
```

  - Operation with matrix and add every neuron's output

```
h = tf.matmul(tf.expand_dims(X, 1), W) + b)
Y_pred = tf.reduce_sum(h, 1)
```

  - Retrain with new Y_pred

```
train(X, Y, Y_pred)
```

- It takes longer to compute and the result is not better. Our function is still linear but the cost is going up and down : good sign = we can reduce the learning rate
- Input's Representation : important to consider the kind of input we are working on. We don't treat the types of data in the same way like :
  - sound using discrete fourier transform
  - text using word histograms

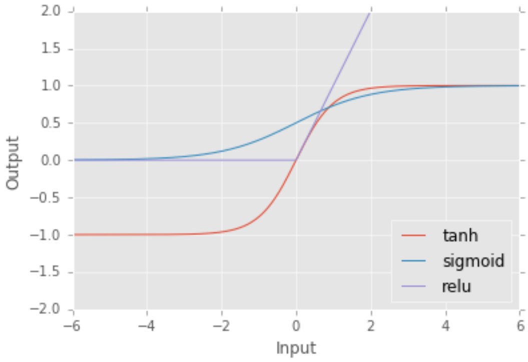
2.5 Over vs. Underfitting

- To approximate curve function we use polynomial function. We will try to learn the influence of each degree of this function

```
Y_pred = tf.Variable(tf.random_normal([1]), name='bias')
for pow_i in range(1,4):
  W = tf.Variable(
      tf.random_normal([1], stddev=0.1), name='weight_%d' % pow_i)
  Y_pred = tf.add(tf.multiply(tf.pow(X, pow_i), W), Y_pred)
```

2.6 Introducing Nonlinearities / Activation Function

- Use of non-linear functions also called activation functions. In every complex DL algorithm there are series of linear, followed by nonlinear operations.
- There are 3 functions used the most : *tanh(), sigmoid(), relu()*



- We modify the matrix multiplication by adding the non-linearity

```
h = tf.nn.tanh(tf.multiply(tf.expland_dims(X,1), W) + b, name='h')
```

- **Fully-connected network** : every neuron is multiplied by every single input value. We multiply our input by a matrix, add a bias, and then apply a non-linearity.

2.7 Going Deeper

- Give useful name within *scopes*. Otherwise the names of the operations are not helpful.

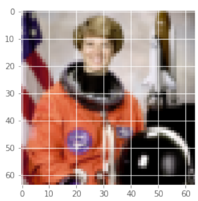
```
W = tf.get_variable( name='W', shape=[n_input, n_output],
    initializer=tf.random_normal_initializer(mean=0.0, stddev=0.1))
```

- This new initializer will create new random value when *sess.run(tf.global_variables_initializer())* is called
- Possible to visualize the network using **Tensorboard**.

3 Image Inpainting

3.1 Description

- This NW will try to demonstrate how the NW we've created bedfore can realize more complicated tasks on the following image :



- We will teach the NW to go from the location on an image frame to a particular color.
- Given any position in an image, the NW will need to learn what color to paint.
- Start
  - Declare input and output

```
# positions in the image
xs = []
# coresponding colors
ys = []
```

  - Storage

```
# for loop on the image
xs.append([row_i, col_i])
ys.append(img[row_i,col_i])
xs = np.array(xs)
ys = np.array(ys)
```

  - Normalization

```
xs = (xs - np.mean(xs)) / np.std(xs)
```

- Goint to use regression to predict the value of a pixel biven its (row, col) position.
- Place input and output in placeholder in which we specify the partially their shape

```
X = tf.placeholder(tf.float32, shape=[None,2], name='X')
Y = tf.placeholder(tf.float32, shape=[None,3], name='Y')
```

3.2 Building the Network

- We are going to create a multi-layers NW.
- We take the *linear()* function created previously.

```
def linear(X, n_input, n_output, activation=None, scope=None):
    with tf.variable_scope(scope or "linear"):
        W = tf.get_variable(
            name='W',
            shape=[n_input, n_output],
            initializer=tf.random_normal_initializer(mean=0.0,
                stddev=0.1))
        b = tf.get_variable(
            name='b',
            shape=[n_output],
            initializer=tf.constant_initializer())
        h = tf.matmul(X, W) + b
        if activation is not None:
```

```
        h = activation(h)
    return h
```

- We use for loop to create the NW

```
n_neurons = [2, 64, 64, 64, 64, 64, 64, 3]
current_input =X
for layer_i in range(1, len(n_neurons)):
   current_input = linear(
      X = current_input, # returned value of the function
      n_input = n_neurons[layer_i - 1],
      n_output = n_neurons[layer_i],
      activation = tf.nn.relu if (layer_i+1) < len(n_neurons) else None,
      scope = 'layer_'+str(layer_i))
Y_pred = current_input
```

3.3 Training

- Define the cost

```
cost = tf.reduce_mean(tf.reduce_sum(distance(Y_pred, Y), 1))
```

- Use another opitmizer *AdamOptimizer* (in general better than *GradientDescentOptimizer*).

```
optimizer = tf.train.AdamOpitmizer(0.001).minimize(cost)
```

- Ready to launch the process
  - Start the session

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

  - Initialize variables (W and b) in the graph so we can use them

    ```
    sess.run(tf.global_variables_initializer())
    prev_training_cost = 0.0
    ```

  - Loop for over number of iterations :

    ```
    for it_i in range(n_iterations):
    ```

    * Permutation of the input (randomness used for the optimizer)

      ```
      idxs = np.random.permutation(range(len(xs)))
      ```

    * Computation of the number of mini-batches

      ```
      n_batches = len(idxs) // batch_size
      ```

* Loop for over the number of mini-batches

```
for batch_i in range(n_batches):
```

  - Take the subinterval of input of the current mini-batch

```
idxs_i = idxs[batch_i*batch_size: (batch_i+1)*batch_size]
```

  - Run the optimizer by feeding input X and output Y to calculate the parameters W and b

```
sess.run(optimizer, feed_dict={X:xs, Y:ys})
```

* Compute the cost to visualize the progression

```
training_cost = sess.run(cost, feed_dict={X:xs, Y:ys})
print(it_i, training_cost)
```

– Visualize the result by plotting

```
ys_pred = Y_pred.eval(feed_dict={X:xs}, session=sess)
img = np.clip(ys_pred.reshape(img.shape), 0, 255).astype(np.uint8)
```

- The entire code

```
n_iterations = 500
batch_size = 50
with tf.Session() as sess:
    # Here we tell tensorflow that we want to initialize all
    # the variables in the graph so we can use them
    # This will set W and b to their initial random normal value.
    sess.run(tf.global_variables_initializer())

    # We now run a loop over epochs
    prev_training_cost = 0.0
    for it_i in range(n_iterations):
        idxs = np.random.permutation(range(len(xs)))
        n_batches = len(idxs) // batch_size
        for batch_i in range(n_batches):
            idxs_i = idxs[batch_i * batch_size: (batch_i + 1) *
                batch_size]
            sess.run(optimizer, feed_dict={X: xs, Y: ys})

        training_cost = sess.run(cost, feed_dict={X: xs, Y: ys})
        print(it_i, training_cost)

        if (it_i + 1) % 20 == 0:
            ys_pred = Y_pred.eval(feed_dict={X: xs}, session=sess)
            fig, ax = plt.subplots(1, 1)
            img = np.clip(ys_pred.reshape(img.shape), 0,
                255).astype(np.uint8)
            plt.imshow(img)
```

```
        plt.show()
```

4 Homework

4.1 Goals

- Learn how to create Neural Network
- Learn to use a NN to paint a image
- Apply creative thinking to the inputs, outputs and definition of a NW

4.2 Part One : Fully Connected Network

- Create the operations for connecting an input to a NW, defined by a *tf.placeholder*, to a series of fully connected, or linear, layers using the formula :

$$\mathbf{H} = \phi(\mathbf{XW} + \mathbf{b}) \tag{5}$$

  - **H** : output layer representing the "hidden" activations of a network
  - ϕ : linearity
  - **X** : input to that layer
  - **W** : layer's weight matirx
  - **b** : layer's bias

- The part **XW** is the most compicated part of the equation scaling and rotating our input.
- By stacking a lot of "linear" + "nonlinear" operations in a series, we can create a **deep neural network**.
- Choising nonlinearities : trial and error. Depends on the normalization scheme : the expected output.
- Keep in mind the functions *relu*, *sigmoïd* and *tanh* for their properties expecially for the final output layer of the NW

4.2.1 Code

- Create a placeholder for the input **X**

```
X = tf.placeholder(dtype=tf.float32, shape=[None, 2], name='X')
```

- Create the paramaters **W** and **b**

```
W = tf.get_variable(dtype=tf.float32, shape=[2,20],
    intializer=tf.random_normal_initializer, name='W')
b = tf.get_variable(dtype=tf.float32, shape=[20],
    intializer=tf.constant_initializer, name='b')
```

- Matrix multiplication **XW** and addition with **b** : XW + b

```
h = tf.matmul(X, W)
h = tf.nn.bias_add(h,b)
```

- Nonlinearity *relu*

```
h = tf.nn.relu(h, name='relu')
```

- New "linear" function using *tf.get_scope*. If there is already a variable created with the same name, TF will raise an exception. Consider 3 solutions :
  * In an interactive console

  ```
  tf.reset_default_graphe()
  ```

  * Typo error creating another layer with the same name
  * Should use context manager when creating graphs and running sessions

  ```
  g = tf.Graph()
  with tf.Session(graph=g) as sess:
     Y_pred, W = linerar(X, 2, 3, activation=tf.nn.relu)
  ```

4.3 Part Two : Image Painting Network

- Load an image

```
img = plt.imshow(dirname)
```

- Collect location of pixel and the related (R,G,B) colors using for loops. Convert the lists of input and output to arrays *np.array()*
- Normalize the input *xs*

```
xs = (xs - np.mean(xs)) / np.std(xs)
```

- Create *placeholder* for the input and the true output

```
# first reset the graph
tf.reset_default_graph()
X = tf.placeholder(dtype=tf.float32, shape=[None,2], name='X')
Y = tf.placeholder(dtype=tf.float32, shape=[None,3], name='Y')
```

- Create 8 layers of neurons : {2, 20, 20, 20, 20, 20, 20, 3}
  - 2 is the input layer

$$H_1 = \phi(XW_1 + b_1) \tag{6}$$

  - All the 20 are the hidden layers

$$H_i = \phi(H_{i-1}W_i + b_i) \tag{7}$$

- 3 is the output layer

$$Y_{pred} = \phi(H_6 W_7 + b_7) \tag{8}$$

4.3.1 Cost function

- Represente how much error there is with our NW and we will use gradient descent and backpropagation.
- Definition of the error

$$cost(Y, \hat{Y}) = \frac{1}{B} \sum_{b=0}^{B} E_b \quad with, E = \sum_{c=0}^{C} (Y_b - \hat{Y}_b)^2 \tag{9}$$