hw4 5525

Kexin Feng 5165462 fengx463@umn.edu

December 2018

1

1.1 (a)

The question asks about a one-d parameter θ to optimize, but the following algorithm works for a more general case, a vector parameter $\vec{\theta}$, a component of which is θ^i. From here, a component of a vector $\vec{V}$ is denoted as V^i

Algorithm 1 Adagrad

1: Initialize parameters $r_0 = 0$, $\delta > 0$, η_0. Initialize $\vec{\theta}_0$.
2: **for** Iteration t = 1...T **do**
3: Calculate the batch gradient at θ_t,

$$\vec{g_t} = \nabla_{\vec{\theta}} \frac{1}{m} \sum_{n=1}^{m} L(f(x_n|\vec{\theta_t}), y_n)$$

4: For each component, $r_t^i = r_{t-1}^i + (g_t^i)^2$
5: For each component, $\theta_t^i = \theta_{t-1}^i - \frac{\eta_0}{\delta + \sqrt{r_t^i}} g_t^i$
6: **end for**
7: Output $\vec{\theta}$

The SGD uses global scaling of learning rate across all the dimensions, which could lead to slow convergence, if the landscape changes differently on different dimensions. Adagrad uses different learning rates for different dimensions. What's more, Adagrad can adapt the step size based on the sum of the historical gradients, as is shown in line 4, 5 in algo.1, such that if historical gradient has large variance, the step is small; if historical gradient has small variance, meaning it's flat, then the step size is big. It's good for large-scale problems.

The limitation is that if the original gradient is large, the learning rate will be very slow, too slow to reach the fast-change region. Also since, r^i keeps summing up, the learning rate monotonically decreases, which could lead to early training termination.

1.2 (b)

One layer of the model is a mapping f from input to output. So the output and the gradient at layer k is calculated below,

$$f^{(k)} = f_k \circ f_{k-1} \circ \cdots f_1 \tag{1}$$
$$f'_{(k)} = f'_k \circ f'_{k-1} \circ \cdots f'_1 \tag{2}$$

So as it goes from the input to deeper layers, this chain is long. And if most of them are below 1 or bigger than 1, then the gradient at the k-th layer will diminish or explode, respectively.

Such problem is seen in deeper layers. Because as explained above, at deeper layers, k is big, the chain is long, so the gradient diminish or explode there.

2

2.1 Question (b)

To tackle this problem, I break it to several steps. First of all, this problem consists of several important elements. First, this looks like a regression problem, given a datum predicting for $\hat{y} = f(x|\theta)$. Next, since there is a non-random pattern of the missing features, it's something that should affect the prediction and should be learned. Learning this non-random probability is like a two-classification problem, meaning the prediction is whether in missing class or in keep class. We have the following notation. $\vec{p} = p_j$ is the probability of x_j being not missing and q is probability for y not missing. Finally, we also need to deal with incomplete data problem.

$$\vec{p} = p_j, \ j = 1, 2, \ldots, D \tag{3}$$
$$q = q \tag{4}$$

2.1.1 Not-missing probability $\vec{p}, q$

Having missing data is very much like applying dropouts, except now applying on the input layer and output layer, instead of hidden layer. Also the keep-probability of dropouts is hyper parameter, fixed by user. But here the not-missing probability $\vec{p}, q$ is learned based on data Z, z. Here and below, D is the dimension of feature space, N is the number of data points, j denotes a certain feature, n denotes a certain data point.

A simple and straight forward way to deal with this problem is to treat $\vec{p}, q$ independent on the input X, y, therefore the optimization of $\vec{p}, q$ is decoupled from the optimization of other parameters like weight w. Thus the loss function is just the cross entropy,

$$L = \left[\prod_{nj} p_j^{Z_{nj}} (1 - p_j)^{1 - Z_{nj}} \right] q^{z_n} (1 - q)^{1 - z_n} \tag{5}$$

the optimization of which leads to the following estimation:

$$p_j = \frac{\sum_n Z_{nj}}{N} \tag{6}$$
$$q = \frac{\sum_n z_n}{N} \tag{7}$$

Next we can just apply these learned keep-probability on the dropouts of the input layer and the output layer. So that we can just focus on the regression problem with incomplete data next.

2.1.2 EM-like algorithm to deal with regression with incomplete data

The loss function without missing data is

$$L = \frac{1}{N} \sum_n |f(\vec{x}_n|\theta) - y_n|^2 \tag{8}$$
$$= L(\theta|\vec{x}, y) \tag{9}$$

Here θ is the parameter to optimize, which can include w or p_j, q.

The loss function with missing data is

$$L(\theta|\vec{x}_n, \vec{\xi}_n, y_n, \eta_n) \overset{e.g.}{=\!=} \frac{1}{N} \sum_n |f((x_1, \xi_2, x_3, \ldots, x_D)_n, \eta_n|\theta)|^2 \tag{10}$$

where ξ, η means the data is missing and n has been summed over.

An effective way to deal with missing data optimization and to predict for the missing data is EM algorithm. The idea is, instead of optimizing the original loss function, to first obtain a distribution of the missing data, take the expectation of the loss function, optimize it, and then go back to update the distribution of the missing data based on the optimized parameter. As

this iteration goes on, it should converge to the real optimized parameters. K-means clustering algorithm is based on the same principle, and the following algorithm is a little similar to K-means.

$$\mathbb{E}[L(\theta_t|\vec{x}_n, y_n, \vec{\xi}_n, \eta_n)|\theta_{t-1}, \vec{x}_n, y_n] \tag{11}$$

$$= \int L(\theta_t|\vec{x}_n, y_n, \vec{\xi}_n, \eta) \prod \mathrm{d}\xi \mathrm{d}\eta D(\xi, \eta|\theta_t) \tag{12}$$

where, $D(\xi, \eta)$ is the distribution function of the missing data.

To obtain the distribution of each missing data, we first need to find the candidate value (the domain) of it. We take only one component ξ_i of $\vec{\xi}$ as an example WLOG. For ξ, we can hist-count the existing values of this feature across all the data, and put them into bins. So we've discretized the candidate values of ξ_i and obtained $h(\xi)$ which is the amount of the value. And this function $h(\xi)$ does not change after this. As initialization, the probability for each value of ξ to appear, denoted as $\psi(\xi)$ is set to be uniform. Then the distribution of ξ is

$$D(\xi) = h(\xi)\psi(\xi) \tag{13}$$

with a difference of normalization factor. To generalize it to all feature dimensions,

$$D(\vec{\xi}, \eta) = h(\vec{\xi}, \eta)\psi(\vec{\xi}, \eta) \tag{14}$$

Then we can run the optimization algorithm on the loss function (12), and find the optimal parameter θ_t. To, based on this parameter, go back to update the distribution, we do the following. We can set the updated parameter probability to be proportional to the value's contribution to the loss function, that is

$$\psi(\vec{\xi}, \eta) \propto \frac{1}{L(\theta_t|\vec{x}, y, \vec{\xi}, \eta)} \tag{15}$$

So

$$D(\vec{\xi}, \eta|\theta_t) = h(\vec{\xi}, \eta)\psi(\vec{\xi}, \eta|\theta_t) \tag{16}$$

is updated, and can be used for the next iteration of expectation and maximization. This update comes from the idea that when the loss is smaller, the corresponding value for xi should be more likely to appear.

Finally, we obtained the optimal parameter θ^* and the optimal distribution of $\vec{\xi}, \eta$, which is

$$D(\vec{\xi}, \eta|\theta^*) = h(\vec{\xi}, \eta)\psi(\vec{\xi}, \eta|\theta^*) \tag{17}$$

Algorithm 2 A sketch of the EM-like algorithm

1: Histcount all the feature value across the data to obtain $D(\vec{\xi}, \eta)$.
2: Initialize $\psi(\vec{\xi}\eta)$ to be uniform distribution.
3: **for** t = 1, 2, ... T **do**
4: Calculate the expectation of the objective function on the distribution function

$$D(\vec{\xi}, \eta|\theta_t) = h(\vec{\xi}, \eta)\psi(\vec{\xi}, \eta|\theta_t)$$

5: Run the optimization algorithm to get the optimal θ_t.
6: Update the distribution function:

$$\psi(\vec{\xi}, \eta) \propto \frac{1}{L(\theta_t|\vec{x}, y, \vec{\xi}, \eta)} \tag{18}$$

7: **end for**

Note that line 4 can be computationally expensive. An alternative way is, instead of calculating the expectation of the loss function, to calculate the expectation of ξ, η and plug it into the loss function, which amounts to exchanging the order of calculating the function and calculating the expectation. Which, intuitively speaking, should work, too.

2.1.3 Combine the two

Now basically the problem is partly solved, if you use Z, z to train p_j, q and then treat them as the dropout keep-probability on the output and input layer, and use the above EM algorithm to obtain the optimal parameter, which decides the regression function, and to obtain the distribution of the missing data.

But a remaining problem is that the estimation of p_j, q is solely dependent of Z, z, decoupled from X, y, which is somewhat over simplified, because, from real situation, we know that a certain value being missing is not totally irrelevant from what this value is. So in a more complicated case, p_j, q should be dependent on D={X, y} as well,

$$p_j(x), q(x)$$

, which makes p and q regression function of x as well. So how to train p, q and y=f(x|w) all together?

I bought the idea from LSTM, the gated Recurrent Neural Network, and instead of using drop out on input and output layer, we can use the gates to control how much the information is kept and transferred.

This is a self designed, so there is no Tensorflow code for this architecture. But I've drawn this architecture below in fig.1. This architecture can be put in a cell. It takes a vector $\vec{x}$ in (in the picture there is 5 hidden units), and outputs regression function $f(x|\theta)$ and the keep-probability $p_j(x|\theta), q(x|\theta)$. Here θ represents all the variable parameters in this cell that can tune to optimize the loss function. The training data set is X, target used for p, q are Z and z, the target for $f(x|\theta)$ is y. The way to deal with missing X and y is the same as the EM algorithm above. The difference is that now we are dealing with p, q and $f(x|\theta)$ together, so the objective function consists of two terms,

$$E = \left[\sum_{nj} Z_{nj} \log p_j(x_n) + (1 - Z_{nj}) \log(1 - p_j(x_n))\right] + z_n \log q + (1 - z_n) \log(1 - q(x_n)) + \frac{1}{N} \sum_n |f(\vec{x}_n|\theta) - y_n|^2 \tag{19}$$

They are optimized together. But to update the distribution function in EM algorithm, still only the second term is used.

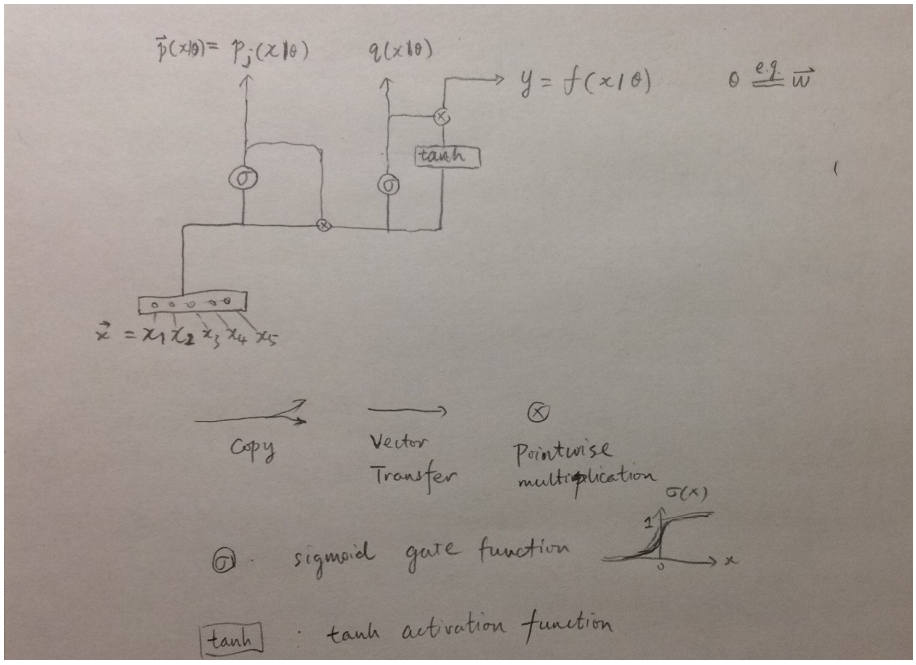

Figure 1: Architecture

Again, finally, we obtained the optimal parameter θ^*, which decides $y = f(x|\theta)$ and the optimal distribution of $\vec{\xi}, \eta$, which is equation (17). The missing data can be learned by taking expectation of the distribution function.

Note that here we are using the same $\vec{\xi}, \eta$ across the whole data set. To learn the missing value in each individual data, the updating of distribution function can be changed to be proportional to the inverse of individual loss function contribution $L_n(\theta_t|\vec{x}, y, \vec{\xi}, \eta) \overset{e.g.}{=\!=} |f((x_1, \xi_2, x_3, \ldots, x_D|\theta) - \eta_n|^2$

2.2 (c)

This problem is very much like an encoder-decoder problem. So I started with a generic encoder-decoder code, which is cited from [1], and try to build on it the solution to our problem.

```
#parameters
epoch = 2
batch_size = 100
n_hidden = 28
seq_len = 28
n_classes = 10

#place holders
inputs = tf.placeholder(tf.float, (None, seq_len), 'inputs')
outputs = tf.placeholder(tf.float, (None, None), 'outputs')
targets = tf.placeholder(tf.float, (None, None), 'targets')
labels = tf.placeholder(tf.float32, [None, n_classes])

weights_out = tf.Variable(tf.truncated_normal([n_hidden, n_classes], stddev=0.1))
biases_out =  tf.Variable(tf.constant(0.1, shape=[n_classes]))

with tf.variable_scope("encoding") as encoding_scope:
    lstm_enc = tf.contrib.rnn.BasicLSTMCell(n_hidden)
    enc_outputs, last_state = tf.nn.dynamic_rnn(lstm_enc, inputs=inputs )

with tf.variable_scope("decoding") as decoding_scope:
    lstm_dec = tf.contrib.rnn.BasicLSTMCell(n_hidden)
    dec_outputs, _ = tf.nn.dynamic_rnn(lstm_dec, inputs=outputs, initial_state=last_st

logits = tf.contrib.layers.fully_connected(dec_outputs, num_outputs=256)
logits_labels = tf.add(tf.matmul(enc_outputs[:], weights_out), biases_out)


with tf.name_scope("optimization"):
    loss1 = tf.contrib.seq2seq_loss(logits, targets, tf.ones([batch_size, seq_len]))
    loss_label = tf.Variable(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2
    for idx in range(1,len(loss_label)):
        loss_label += tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits
    optimizer = tf.train.AdamOptimizer(1e-3).minimize(loss1 + loss_label)

# training:
sess = tf.Session()
sess.run(tf.global_variables_initializer())
for e in range(epoch):
    for ind, (source, target) in enumerate(batch_data(x_train, y_train, batch_size)):
        _, batch_loss, batch_logits = sess.run([optimizer, loss, logits],
                        feed_dict= {inputs: source,
                                    outputs: target[:, :-1],
                                    targets: target[:, 1:]})
        accuracy = np.mean(batch_logits.argmax(axis=-1) == target[:, 1:])

# testing:
(source, target) = next(batch_data(x_test, y_test, batch_size))

# predict for sequence
```

```
dec_input = np.zeros((len(source), 1)) + char2numY['<Start>']
for i in range(y_seq_length):
    batch_logits = sess.run(logits,
            feed_dict={inputs: source,
                       outputs: dec_input})
    prediction = batch_logits[:, -1].argmax(axis=-1)
    dec_input = np.hstack([dec_input, prediction[:, None]])
print('Accuracy for sequence is ', np.mean(dec_input))

# predict for label
logits_label = sess.run(logits_labels, feed_dict={inputs: source})
prediction_labels = logits_label.argmax(axis=-1)
print('The predicted label: ', prediction_labels, ' corresponding to the pictures in s
```

For our problem, given an sequence mnist pictures, the task is to learn both their labels and the future picture sequence. The latter can be solved by a generic encoder-decoder model, and the loss function is seq2seq_loss. which is shown in figure 2. To predict the label of each image, note that in the encoder LSTM, the outputs are unoccupied. So we can use those outputs as prediction for the label of the image, since each element of sequence is an image vector of 784 dimension. This is a classification problem and the loss function is crossentropy. Therefore for a sequence of images, the total loss function is

$$L = seq2seq\_loss + \sum_{i=0}^{seq\_len} crossentropy(logits\_label[i], label[i]) \tag{20}$$

which is the same as line 30 - 34 in the code above 2.2. Then we can optimize this total loss function to get the optimal parameters. The testing is done in line 47-63. The prediction for sequence is done in the for loop, the dec_input is updated by horizontally stack the new element gerated from the model.

The training of this new model should work, because now the model has not just the information of the target sequence to learn, but also the target labels to learn. These target labels offers additional constrains to the parameter space, so the model has less degrees of freedom to change and easier to train.

2.3 Question(a)

This is a sequence learning and prediction problem. So an encoder-decoder model can be used, which is the same as question(c), above. However the difference in this problem is that the output sequence, which is structure database query, must strictly obey the grammar. So the encoder-decoder problem can be modified in the following way.

First the candidates of frames of the query needs to be ready, such that what the machine needs to do is just choose from these query frames and fill in a sequence of the blanks, based on the training parameters. The query_id of the correct frame can be used as target compared against the last output of the encoder LSTM. Then the target sequence is compared with the output sequence from decoder.

Therefore the loss function consists of two parts,

$$L = seq2seq\_loss + crossentropy(query\_id\_prediction, query\_id) \tag{21}$$

The code has been shown below. The only difference from the code for question (c) is in line 26-34, where only the last output is taken to predict for the query frame id and calculate the loss.

```
#parameters
epoch = 2
batch_size = 100
n_hidden = 28
seq_len = 28
n_classes = 10

```

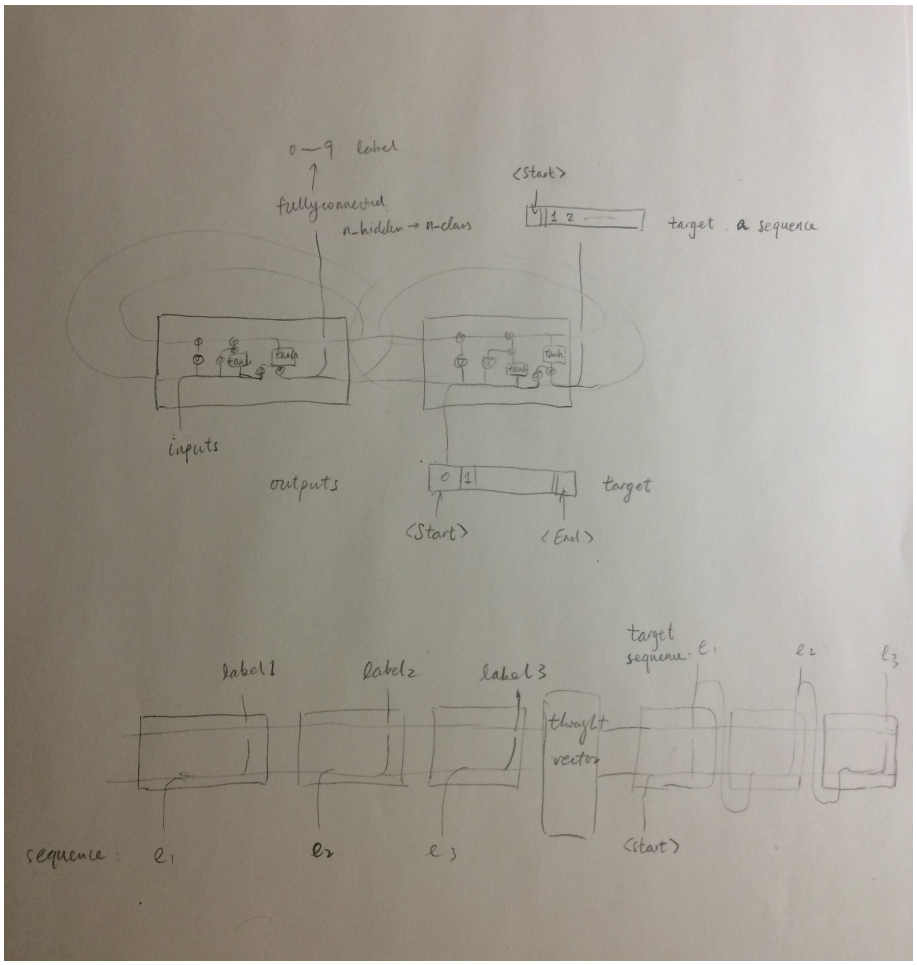

Figure 2: Caption

```
#place holders
inputs = tf.placeholder(tf.float , (None,  seq_len), 'inputs')
outputs = tf.placeholder(tf.float , (None, None), 'outputs')
targets = tf.placeholder(tf.float , (None, None), 'targets')
labels = tf.placeholder(tf.float32 , [None, n_classes])

weights_out = tf.Variable(tf.truncated_normal([n_hidden, n_classes], stddev=0.1))
biases_out =  tf.Variable(tf.constant(0.1, shape=[n_classes]))

with tf.variable_scope("encoding") as encoding_scope:
    lstm_enc = tf.contrib.rnn.BasicLSTMCell(n_hidden)
    enc_outputs, last_state = tf.nn.dynamic_rnn(lstm_enc, inputs=inputs )

with tf.variable_scope("decoding") as decoding_scope:
    lstm_dec = tf.contrib.rnn.BasicLSTMCell(n_hidden)
    dec_outputs, _ = tf.nn.dynamic_rnn(lstm_dec, inputs=outputs, initial_state=last_st

logits = tf.contrib.layers.fully_connected(dec_outputs, num_outputs=256)
logits_labels = tf.add(tf.matmul(enc_outputs[-1], weights_out), biases_out)


with tf.name_scope("optimization"):
    loss1 = tf.contrib.seq2seq_loss(logits, targets, tf.ones([batch_size, seq_len]))
    loss_label = tf.Variable(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2
    # for idx in range(1,len(loss_label)):
    #     loss_label += tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logi
    optimizer = tf.train.AdamOptimizer(1e-3).minimize(loss1 + loss_label)

# training:
sess = tf.Session()
sess.run(tf.global_variables_initializer())
for e in range(epoch):
    for ind, (source, target) in enumerate(batch_data(x_train, y_train, batch_size)):
        _, batch_loss, batch_logits = sess.run([optimizer, loss, logits],
                        feed_dict= {inputs: source,
                                    outputs: target[:, :-1],
                                    targets: target[:, 1:]})
        accuracy = np.mean(batch_logits.argmax(axis=-1) == target[:, 1:])

# testing:
(source, target) = next(batch_data(x_test, y_test, batch_size))

# predict for sequence
dec_input = np.zeros((len(source), 1)) + char2numY['<Start>']
for i in range(y_seq_length):
    batch_logits = sess.run(logits,
            feed_dict={inputs: source,
                       outputs: dec_input})
    prediction = batch_logits[:, -1].argmax(axis=-1)
    dec_input = np.hstack([dec_input, prediction[:, None]])
print('Accuracy for sequence is ', np.mean(dec_input))

# predict for label
logits_label = sess.run(logits_labels, feed_dict={inputs: source})
prediction_labels = logits_label.argmax(axis=-1)
print('The predicted label: ', prediction_labels, ' corresponding to the pictures in s
```

References

[1] https://www.youtube.com/watch?v=_Sm0q_FckM8&ab_channel=TheMathStudent