Chapter3_answer

April 24, 2019

Chapter 3 of D2L Textbook

```
In [55]: import d2l
         import mxnet as mx
         from mxnet import autograd, gluon, init, nd
         from mxnet.gluon import loss as gloss, data as gdata, nn

         import time
         import random
         import numpy as np

         from IPython.core.interactiveshell import InteractiveShell
         InteractiveShell.ast_node_interactivity = "all"
```

3.1.5. Exercises

1. **Assume that we have some data $x_1, x_n \in \mathbb{R}$. Our goal is to find a constant such that $\sum_i (x_i - b)^2$ is minimized.**

   a.**Find the optimal closed form solution.**

   b.**What does this mean in terms of the Normal distribution?**

$$\frac{\partial}{\partial b}\sum_i (x_i - b)^2 = -2\sum_i (x_i - b) = 0 \text{ => } \hat{b} = \bar{x} = mean(x_i)$$

$$\frac{\partial}{\partial b}\sum_i (x_i - b)^2 = 2 > 0 \text{ => } \hat{b} \text{ is a minimum.}$$

   This equals to the mean (μ) of the normal distribution.

2. **Assume that we want to solve the optimization problem for linear regression with quadratic loss explicitly in closed form. To keep things simple, you can omit the bias from the problem.**

   a. **Rewrite the problem in matrix and vector notation (hint - treat all the data as a single matrix).**

$$L(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} - y^{(i)}\right)^2 . = ||\mathbf{X}\mathbf{w} - \mathbf{y}||^2$$

b. **Compute the gradient of the optimization problem with respect to .**

$$\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}) = 2X^T(X\mathbf{w} - \mathbf{y})$$

c. **Find the closed form solution by solving a matrix equation.**

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T \mathbf{y}$$

d. **When might this be better than using stochastic gradient descent (i.e. the incremental optimization approach that we discussed above)? When will this break (hint - what happens for high-dimensional , what if many observations are very similar)?.**

When the features are linear independent.

For high demensional x, if some of the features are of high correlations, then $(X^T X)^{-1}$ may not existed.

3. **Assume that the noise model governing the additive noise is the exponential distribution. That is,** $p(\epsilon) = \frac{1}{2} exp(|\epsilon|)$**.**

a.**Write out the negative log-likelihood of the data under the model** $logp(Y|X)$**.**

$$logp(Y|X) = -log(\frac{1}{2} exp(-|X\mathbf{w} - Y|)) = |X\mathbf{w} - Y| - log(\frac{1}{2})$$

b. **Can you find a closed form solution?**

$$\mathbf{w} = argmin_{\mathbf{w}} |X\mathbf{w} - Y|$$

c. **Suggest a stochastic gradient descent algorithm to solve this problem. What could possibly go wrong (hint - what happens near the stationary point as we keep on updating the parameters). Can you fix this?**

4. **Compare the runtime of the two methods of adding two vectors using other packages (such as NumPy) or other programming languages (such as MATLAB).**

3.2.9. Exercises

1. **What would happen if we were to initialize the weights =0 . Would the algorithm still work?**

   It doesn't work. Each neuron at a layer will have exactly same gradient at each update, hence all the neurons will learn the same thing.

2. **Assume that you're Georg Simon Ohm trying to come up with a model between voltage and current. Can you use autograd to learn the parameters of your model.**

```
In [53]: lr = 0.1  # Learning rate
         num_epochs = 10  # Number of iterations
         batch_size = 10
         num_examples = 1000

         ## construct voltage, current & resistance
         resistance = 10 # nd.random.normal(scale=0.01, shape=(num_examples, 1))
         current = nd.random.normal(scale=1, shape=(num_examples, 1))
         voltage = resistance * current
         print(voltage.shape)
         voltage += nd.random.normal(scale=0.01, shape=(num_examples, 1))

         ## initialize parameters
         loss = gloss.L2Loss()
         net = nn.Sequential()
         net.add(nn.Dense(1))
         net.initialize(init.Normal(sigma=0.1))
         trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd', {'learning_rate': lr
         trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd', {'learning_rate': lr})

         for epoch in range(num_epochs):
             b = nd.zeros(shape=(1))
             w = nd.random.normal(scale=0.01, shape=(1))
             for X, y in data_iter(batch_size, current, voltage):
                 with autograd.record():
                     l = loss(net(X), y)  # Minibatch loss
         #           import ipdb
         #           ipdb.set_trace()
                 l.backward()  # Compute gradient on l with respect to [w,b]
                 trainer_w.step(batch_size)
                 trainer_b.step(batch_size)
             train_l = loss(linreg(voltage, w, b), current)
             print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

```
(1000, 1)
epoch 1, loss 0.477460
epoch 2, loss 0.597413
epoch 3, loss 0.385625
epoch 4, loss 0.644568
epoch 5, loss 0.315570
epoch 6, loss 0.455128
epoch 7, loss 0.450721
epoch 8, loss 0.501864
epoch 9, loss 0.325878
epoch 10, loss 0.545352
```

3. **Can you use Planck's Law to determine the temperature of an object using spectral energy density.**

??????

4. **What are the problems you might encounter if you wanted to extend autograd to second derivatives? How would you fix them?**

   The second order gradient (Jacobian matrix) will have n^2 more parameters for a layer with n neuron weights.

5. **Why is the reshape function needed in the squared_loss function?**

   Since the prediction tensor can have arbitrary shape, while its size(the product of all the dimension times together) is equal to true label "y", hence we just need to reshape true label "y" as same as the shape of prediction y.

6. **Experiment using different learning rates to find out how fast the loss function value drops.**

   A good start choose of lr is 0.1. Sometimes 1 is too high and the model cannot converge, but 0.01 is too small and the model converge slowly.

7. **If the number of examples cannot be divided by the batch size, what happens to the data_iter function's behavior?**

   It will take the last few examples which are less than a batchsize to train.

```
In [50]: def data_iter(batch_size, features, labels):
             num_examples = len(features)
             indices = list(range(num_examples))
             random.shuffle(indices)
             for i in range(0, num_examples, batch_size):
                 j = nd.array(indices[i: min(i + batch_size, num_examples)])
                 yield features.take(j), labels.take(j)
```

3.3.9. Exercises

1. **If we replace l = loss(output, y) with l = loss(output, y).mean(), we need to change trainer.step(batch_size) to trainer.step(1) accordingly. Why?**

   From the documentation of `Gluon.Trainer.step`, gradient will be normalized by 1/batch_size, which is the same as take a mean with batch_size 1.

   See details: https://mxnet.apache.org/api/python/gluon/gluon.html#mxnet.gluon.Trainer

2. **Review the MXNet documentation to see what loss functions and initialization methods are provided in the modules gluon.loss and init. Replace the loss by Huber's loss.**

   L1Loss, L2Loss, SigmoidBinaryCrossEntropyLoss, SoftmaxCrossEntropyLoss, KLDivLoss, etc. https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.Loss

3. **How do you access the gradient of dense.weight?**

   `with autograd.record():` records computation history on the fly to calculate gradients later.

   To compute gradient with respect to an NDArray x, first call `x.attach_grad()` to allocate space for the gradient. Then, start a with `autograd.record():` block, and do some computation. Finally, call `backward()` on the result. See details https://mxnet.apache.org/api/python/autograd/autograd.html