Chapter5

April 27, 2019

D2L Textbook Solution

```
In [1]: 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 numpy as np

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

0.0.1 Chapter 5

5.1.6. Exercises

1. **What kind of error message will you get when calling an init method whose parent class not in the init function of the parent class?**

   InitializationError, i.e. cannot initialize the related parameters (the weights).

2. **What kinds of problems will occur if you remove the asscalar function in the FancyMLP class?**

   Returns a scalar whose value is copied from the resulted array.

3. **What kinds of problems will occur if you change self.net defined by the Sequential instance in the NestMLP class to self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]?**

   If change *nn.Sequential()* to the above list, then we cannot add additional network to NestMLP, since the function of Sequential is the concatenations of layers and blocks. Following code will give you the error.

```
In [8]: class NestMLP_exercise(nn.Block):
            def __init__(self, **kwargs):
                super(NestMLP_exercise, self).__init__(**kwargs)
                self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]
                self.net.add(nn.Dense(64, activation='relu'),
```

```
                         nn.Dense(32, activation='relu'))
            self.dense = nn.Dense(16, activation='relu')

        def forward(self, x):
            return self.dense(self.net(x))

net = nn.Sequential()
net.add(NestMLP_exercise(), nn.Dense(20))
net.initialize()
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-8-ac890d95d2b6> in <module>
 11
 12 net = nn.Sequential()
---> 13 net.add(NestMLP_exercise(), nn.Dense(20))
 14 net.initialize()


<ipython-input-8-ac890d95d2b6> in __init__(self, **kwargs)
  3         super(NestMLP_exercise, self).__init__(**kwargs)
  4         self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu
----> 5         self.net.add(nn.Dense(64, activation='relu'),
  6                      nn.Dense(32, activation='relu'))
  7         self.dense = nn.Dense(16, activation='relu')


AttributeError: 'list' object has no attribute 'add'
```

4. **Implement a block that takes two blocks as an argument, say net1 and net2 and returns the concatenated output of both networks in the forward pass (this is also called a parallel block).**

```
In [18]: class ParallelMLP(nn.Block):
             def __init__(self, **kwargs):
                 super(ParallelMLP, self).__init__(**kwargs)
                 self.input_net1 = nn.Sequential()
                 for item1 in net1:
                     self.input_net1.add(item1)
                 self.input_net1.add(nn.Dense(1))

                 self.input_net2 = nn.Sequential()
                 for item2 in net2:
                     self.input_net2.add(item2)
```

```
            self.input_net2.add(nn.Dense(1))

        def forward(self, x):
            out1 = self.input_net1(x) ## shape of out1 : (batch_size, 1)
            out2 = self.input_net2(x) ## shape of out1 : (batch_size, 1)
            out = mx.nd.concat(out1, out2, dim=-1) ## shape of out : (batch_size, 2)
            return out


    x = nd.random.uniform(shape=(2, 20))
    net1 = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]
    net2 = [nn.Dense(64, activation='relu')]
    parallel = ParallelMLP()
    parallel.initialize()
    parallel(x)
```

Out[18]:

```
[[ 0.00052053 -0.03974626]
 [-0.00133751 -0.01168625]]
<NDArray 2x2 @cpu(0)>
```

5. **Assume that you want to concatenate multiple instances of the same network. Implement a factory function that generates multiple instances of the same block and build a larger network from it.**

In [30]:

```
out = mx.nd.zeros(shape=(2,3))
out
```

Out[30]:

```
[[0. 0. 0.]
 [0. 0. 0.]]
<NDArray 2x3 @cpu(0)>
```

In [33]:

```
class LargeNetwork(nn.Block):
    def __init__(self, net, **kwargs):
        super(LargeNetwork, self).__init__(**kwargs)
        self.input_net = nn.Sequential()
        for item in net:
            self.input_net.add(item)
        self.input_net.add(nn.Dense(1))
        self.large_net = {}

    def __init_large_net(self, length):
        for i in range(length):
            self.large_net[i] = self.input_net

    def forward(self, input_list):
        '''
        input_list is a list of input instance of same shape
```

```
            '''
            out = mx.nd.zeros(shape=(len(input_list), len(input_list[0]), 1)) ## 1 is the

            ## initial large net if it does not exist
            if len(self.large_net.keys()) == 0:
                self.__init_large_net(len(input_list))

            for j, instance in enumerate(input_list):
                net = self.large_net[j]
                out[j,:] = net(instance) ## shape of out1 : (batch_size, 1)
            return mx.nd.concat(out, dim=-1)


    x = nd.random.uniform(shape=(2, 20))
    y = nd.random.uniform(shape=(2, 20))
    z = nd.random.uniform(shape=(2, 20))

    net = [nn.Dense(64, activation='relu'), nn.Dense(10, activation='relu')]
    large_network = LargeNetwork(net)
    large_network.initialize()
    large_network([x,y,z])
```

Out[33]:

```
[[[-0.00043408]
  [ 0.00071604]]

 [[-0.00023887]
  [ 0.00123192]]

 [[ 0.00160318]
  [ 0.00164159]]]
<NDArray 3x2x1 @cpu(0)>
```

5.2.5. Exercises

1. **Use the FancyMLP definition of the previous section and access the parameters of the various layers.**

In [34]:

```
class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        # Random weight parameters created with the get_constant are not
        # iterated during training (i.e. constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', nd.random.uniform(shape=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, x):
```

```
            x = self.dense(x)
            # Use the constant parameters created, as well as the relu and dot
            # functions of NDArray
            x = nd.relu(nd.dot(x, self.rand_weight.data()) + 1)
            # Reuse the fully connected layer. This is equivalent to sharing
            # parameters with two fully connected layers
            x = self.dense(x)
            # Here in Control flow, we need to call asscalar to return the scalar
            # for comparison
            while x.norm().asscalar() > 1:
                x /= 2
            if x.norm().asscalar() < 0.8:
                x *= 10
            return x.sum()

    x = nd.random.uniform(shape=(2, 20))
    net = FancyMLP()
    net.initialize()
    net(x)
    print(net.collect_params())
```

```
fancymlp0_ (
  Constant fancymlp0_rand_weight (shape=(20, 20), dtype=<class 'numpy.float32'>)
  Parameter dense76_weight (shape=(20, 20), dtype=float32)
  Parameter dense76_bias (shape=(20,), dtype=float32)
)
```

2. **Look at the MXNet documentation and explore different initializers.**

   Constant, Normal, Xavier, Orthogonal, MSRAPrelu, etc. http://mxnet.incubator.apache.org/api/python/optimization/optimization.html#mxnet.initializer.Mixed

3. **Try accessing the model parameters after net.initialize() and before net(x) to observe the shape of the model parameters. What changes? Why?**

   The shape of layer changed. In the below example, for eg, "dense77_weight" was of shape=(20, 0) before net(x), but became shape=(20, 20) after input x. Since dim==-1 of x is 20.

```
In [40]: print(x.shape, "\n")
         net3 = FancyMLP()
         net3.initialize()
         print(net3.collect_params())
         net3(x)
         print(net3.collect_params())
```

```
(2, 20)

fancymlp4_ (
```

```
  Constant fancymlp4_rand_weight (shape=(20, 20), dtype=<class 'numpy.float32'>)
  Parameter dense80_weight (shape=(20, 0), dtype=float32)
  Parameter dense80_bias (shape=(20,), dtype=float32)
)
fancymlp4_ (
  Constant fancymlp4_rand_weight (shape=(20, 20), dtype=<class 'numpy.float32'>)
  Parameter dense80_weight (shape=(20, 20), dtype=float32)
  Parameter dense80_bias (shape=(20,), dtype=float32)
)
```

4. **Construct a multilayer perceptron containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.**

In [59]:
```
## ??????
x = nd.random.uniform(shape=(2, 20))
y = nd.random.uniform(shape=(2, 1))

net4 = nn.Sequential()
shared = nn.Dense(8, activation='relu')
shared_reuse = nn.Dense(8, activation='relu', params=shared.params)
net4.add(nn.Dense(8, activation='relu'),
        shared,
        shared_reuse,
        nn.Dense(1))
net4.initialize()
loss = gloss.SoftmaxCrossEntropyLoss()


with mx.autograd.record():
    y_hat = net4(x)
    print(net4.collect_params())
    l = loss(y_hat, y).sum()
l.backward()
print(shared.weight.grad())
print(shared_reuse.weight.grad())
```

```
sequential50_ (
  Parameter dense139_weight (shape=(8, 20), dtype=float32)
  Parameter dense139_bias (shape=(8,), dtype=float32)
  Parameter dense137_weight (shape=(8, 8), dtype=float32)
  Parameter dense137_bias (shape=(8,), dtype=float32)
  Parameter dense140_weight (shape=(1, 8), dtype=float32)
  Parameter dense140_bias (shape=(1,), dtype=float32)
)

[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
```

```
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
<NDArray 8x8 @cpu(0)>

[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
<NDArray 8x8 @cpu(0)>
```

5. **Why is sharing parameters a good idea?**

   Sharing parameters can save memory in general and have specific benefits for the following:

   For CNN in image recognition, sharing parameters gives the network the ability to look for a given feature everywhere in the image, rather than in just a certain area.

   For RNN, it shares parameters across time steps of the sequence, so it can generalize well to examples of different sequence length.

   For autoencoder, encoder and decoder share parameters. In a single layer autoencoder with linear activation, sharing weights forces orthogonality among different hidden layer of weight matrix.

5.3.5. Exercises

1. **What happens if you specify only parts of the input dimensions. Do you still get immediate initialization?**

   No. Initialization will occur right before the forward pass.

2. **What happens if you specify mismatching dimensions?**

   Error like the following will occur:

   ```
   Check failed: from.shape() == to->shape() operands shape mismatchfrom.shape = (126,) to.sh
   ```

3. **What would you need to do if you have input of varying dimensionality? Hint - look at parameter tying.**

   See below printed results for details. Notice that the seiral number xxx of the **shared : sequentialxxx** is always the same, while that of **unique : sequentialxxx** is different for different dimensionality inputs.

```
In [96]: def get_net535_init():
             net535 = nn.Sequential()
             net535.add(nn.Dense(16, activation='relu'), nn.Dense(8))
             net535.initialize()  ## reinit one specific layer
             return(net535)

         def train_535(net535_unique, net535_shared, X, y, test_iter=None, loss=gloss.L2Loss()
                       num_epochs=2, batch_size=32, params=None, lr=0.05):
             for epoch in range(num_epochs):
                 print("***************** epoch {} *****************".format(epoch))
                 with autograd.record():
                     try:
                         mid = net535_unique(X)
                         print("unique : ", net535_unique.collect_params())

                     except:
                         ## if existing network shape does not match, reinitialize a network
                         net535_unique_new = get_net535_init()
                         net535_unique = net535_unique_new
                         mid = net535_unique(X)
                         print("unique : ", net535_unique.collect_params())

                     ## the shared network does not need to be reinit as the shape is consista
                     y_hat = net535_shared(mid)
                     print("shared : ", net535_shared.collect_params())


         ## Generate dataset
         batch_size = 10
         features1 = nd.random.uniform(shape=(batch_size, 3))
         print(features1.shape)
         features2 = nd.random.uniform(shape=(batch_size, 6))
         print(features2.shape)

         y = nd.random.uniform(shape=(batch_size, 1))
         net535_unique_ex = get_net535_init()
         net535_shared_ex = get_net535_init()
         print("\n^^^^^^^^^^^^^^^^^^^^^ features2 ^^^^^^^^^^^^^^^^^^^^^".format(features))
         train_535(net535_unique_ex, net535_shared_ex, features1, y)
         print("\n^^^^^^^^^^^^^^^^^^^^^ features2 ^^^^^^^^^^^^^^^^^^^^^".format(features))
         train_535(net535_unique_ex, net535_shared_ex, features2, y)

         # train_535(net=net535, train_iter=data_iter)
```

```
^^^^^^^^^^^^^^^^^^^^^ features2 ^^^^^^^^^^^^^^^^^^^^^
***************** epoch 0 *****************
unique :  sequential117_ (
```

```
  Parameter dense261_weight (shape=(16, 3), dtype=float32)
  Parameter dense261_bias (shape=(16,), dtype=float32)
  Parameter dense262_weight (shape=(8, 16), dtype=float32)
  Parameter dense262_bias (shape=(8,), dtype=float32)
)
shared :  sequential118_ (
  Parameter dense263_weight (shape=(16, 8), dtype=float32)
  Parameter dense263_bias (shape=(16,), dtype=float32)
  Parameter dense264_weight (shape=(8, 16), dtype=float32)
  Parameter dense264_bias (shape=(8,), dtype=float32)
)
****************** epoch 1 ******************
unique :  sequential117_ (
  Parameter dense261_weight (shape=(16, 3), dtype=float32)
  Parameter dense261_bias (shape=(16,), dtype=float32)
  Parameter dense262_weight (shape=(8, 16), dtype=float32)
  Parameter dense262_bias (shape=(8,), dtype=float32)
)
shared :  sequential118_ (
  Parameter dense263_weight (shape=(16, 8), dtype=float32)
  Parameter dense263_bias (shape=(16,), dtype=float32)
  Parameter dense264_weight (shape=(8, 16), dtype=float32)
  Parameter dense264_bias (shape=(8,), dtype=float32)
)

^^^^^^^^^^^^^^^^^^^^^^ features2 ^^^^^^^^^^^^^^^^^^^^^^
****************** epoch 0 ******************
unique :  sequential119_ (
  Parameter dense265_weight (shape=(16, 6), dtype=float32)
  Parameter dense265_bias (shape=(16,), dtype=float32)
  Parameter dense266_weight (shape=(8, 16), dtype=float32)
  Parameter dense266_bias (shape=(8,), dtype=float32)
)
shared :  sequential118_ (
  Parameter dense263_weight (shape=(16, 8), dtype=float32)
  Parameter dense263_bias (shape=(16,), dtype=float32)
  Parameter dense264_weight (shape=(8, 16), dtype=float32)
  Parameter dense264_bias (shape=(8,), dtype=float32)
)
****************** epoch 1 ******************
unique :  sequential119_ (
  Parameter dense265_weight (shape=(16, 6), dtype=float32)
  Parameter dense265_bias (shape=(16,), dtype=float32)
  Parameter dense266_weight (shape=(8, 16), dtype=float32)
  Parameter dense266_bias (shape=(8,), dtype=float32)
)
shared :  sequential118_ (
  Parameter dense263_weight (shape=(16, 8), dtype=float32)
```

```
  Parameter dense263_bias (shape=(16,), dtype=float32)
  Parameter dense264_weight (shape=(8, 16), dtype=float32)
  Parameter dense264_bias (shape=(8,), dtype=float32)
)
```

5.4.4. Exercises

1. **Design a layer that learns an affine transform of the data, i.e. it removes the mean and learns an additive parameter instead.**

```
In [98]: class CenteredLayer(nn.Block):
             def __init__(self, **kwargs):
                 super(CenteredLayer, self).__init__(**kwargs)

             def forward(self, x):
                 return x - x.mean()

         layer = CenteredLayer()
         layer(nd.array([1, 2, 3, 4, 5]))
```

```
Out[98]:
         [-2. -1.  0.  1.  2.]
         <NDArray 5 @cpu(0)>
```

2. **Design a layer that takes an input and computes a tensor reduction, i.e. it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.**

```
In [126]: class TensorReductionLayer(nn.Block):
              def __init__(self, k, x_shape, **kwargs):
                  super(TensorReductionLayer, self).__init__(**kwargs)
                  self.weight = self.params.get('weight', shape=(x_shape, k, x_shape))

              def forward(self, x):
                  mid = nd.dot(self.weight.data(), x)
                  print(mid.shape)
                  out = nd.dot(x.T, mid)
                  return out.reshape(k)

          ## sample random x with given length
          x_length = 5
          x = nd.random.uniform(shape=(x_length, 1))

          k = 3  ## k can be any integer
          TRlayer = TensorReductionLayer(k, x_length)
          TRlayer.initialize()
          TRlayer(x)
```

```
(5, 3, 1)
```

```
Out[126]:
          [-0.01547093 -0.00399414 -0.01535948]
          <NDArray 3 @cpu(0)>
```

3. **Design a layer that returns the leading half of the Fourier coefficients of the data. Hint - look up the fft function in MXNet.**

```
In [ ]: ## TODO: Run on GPU

        class FourierLayer(nn.Block):
            def __init__(self, k, x_shape, **kwargs):
                super(FourierLayer, self).__init__(**kwargs)
                self.weight = self.params.get('weight', shape=(x_shape, k, x_shape))

            def forward(self, x):
                mid = nd.dot(self.weight.data(), x)
                print(mid.shape)
                out = nd.dot(x.T, mid)
                return out.reshape(k)

        ## sample random x with given shape
        data = np.random.normal(0,1,(3,4))
        out = mx.contrib.ndarray.fft(data = mx.nd.array(data,ctx = mx.gpu(0)))
        out
```

5.5.4. Exercises

1. **Even if there is no need to deploy trained models to a different device, what are the practical benefits of storing model parameters?**

   a. Saving intermediate results (checkpointing) to ensure that we don't lose several days worth of computation when running a long training process.
   b. To load a pretrained model for fine tuning.

2. **Assume that we want to reuse only parts of a network to be incorporated into a network of a different architecture. How would you go about using, say the first two layers from a previous network in a new network.**

```
In [136]: from mxnet.gluon import model_zoo
          alexnet = model_zoo.vision.alexnet(pretrained=True)
          print(alexnet.collect_params('alexnet.*_conv0.*'))
          print(alexnet.collect_params('alexnet.*_conv1.*'))
```

```
alexnet6_ (
  Parameter alexnet6_conv0_weight (shape=(64, 3, 11, 11), dtype=<class 'numpy.float32'>)
  Parameter alexnet6_conv0_bias (shape=(64,), dtype=<class 'numpy.float32'>)
)
alexnet6_ (
  Parameter alexnet6_conv1_weight (shape=(192, 64, 5, 5), dtype=<class 'numpy.float32'>)
```

```
  Parameter alexnet6_conv1_bias (shape=(192,), dtype=<class 'numpy.float32'>)
)
```

3. **How would you go about saving network architecture and parameters? What restrictions would you impose on the architecture?**

   In order to reload a trained model, we need to generate the architecture in code and then load the parameters from disk.

```
In [ ]: class MLP(nn.Block):
            def __init__(self, **kwargs):
                super(MLP, self).__init__(**kwargs)
                self.hidden = nn.Dense(256, activation='relu')
                self.output = nn.Dense(10)

            def forward(self, x):
                return self.output(self.hidden(x))

        ## train a network and save
        net = MLP()
        net.initialize()
        x = nd.random.uniform(shape=(2, 20))
        y = net(x)
        clone.save_parameters('mlp.params')

        ## define the architecture and then reload parameters
        clone = MLP()
        clone.load_parameters('mlp.params')
```

5.6.5. Exercises

1. **Try a larger computation task, such as the multiplication of large matrices, and see the difference in speed between the CPU and GPU. What about a task with a small amount of calculations?**

```
In [ ]: ## TODO: Run on GPU

        s = 4096

        A = nd.random.normal(shape=(s, s))
        B = nd.random.normal(shape=(s, s))
        tic = time.time()
        C = nd.dot(A, B)
        C.wait_to_read()
        print("On CPU : Matrix by matrix: " + str(time.time() - tic) + " seconds")


        A1 = A.copyto(mx.gpu(1))
```

```
B1 = B.copyto(mx.gpu(1))
tic = time.time()
C = nd.dot(A, B)
C.wait_to_read()
print("On GPU : Matrix by matrix: " + str(time.time() - tic) + " seconds")
```

2. **How should we read and write model parameters on the GPU?**

   Use `net.load_parameters(file_name, ctx=ctx)` to read model parameters, and `net.save_parameters(file_name)` to save model parameters.

3. **Measure the time it takes to compute 1000 matrix-matrix multiplications of 100Œ100 matrices and log the matrix norm $tr(MM)$ one result at a time vs. keeping a log on the GPU and transferring only the final result.**

In [ ]:
```
## TODO: Run on GPU
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


tic = time.time()
for j in range(1000):
    M = nd.random.normal(shape=(100, 100))
    C = nd.sum(nd.diag(nd.dot(M, M.T)))
    C.wait_to_read()
print("Read one-by-one on GPU : " + str(time.time() - tic) + " seconds")



tic = time.time()
D = nd.zeros(shape=(1000,))
for j in range(1000):
    M = nd.random.normal(shape=(100, 100))
    D[i] = nd.sum(nd.diag(nd.dot(M, M.T)))

D.wait_to_read()
print("Read all at once on GPU : " + str(time.time() - tic) + " seconds")
```

4. **Measure how much time it takes to perform two matrix-matrix multiplications on two GPUs at the same time vs. in sequence on one GPU (hint - you should see almost linear scaling).**

In [151]:
```
## TODO: Run on GPU

  s = 4096
  tic = time.time()
  for j in range(2):
      M = nd.random.normal(shape=(s, s))
      C = nd.dot(M, M.T)
```

```
    C.shape
#     C.wait_to_read()
print("Two matrix-matrix multiplications in sequence on GPU : " + str(time.time() - t



tic = time.time()
M = nd.random.normal(shape=(2, s, s))
N = nd.random.normal(shape=(s, s))
D = nd.dot(M, N)
D.shape
print("Two matrix-matrix multiplications at the same time on GPU : " + str(time.time
```

Out[151]: (4096, 4096)

Out[151]: (4096, 4096)

```
Two matrix-matrix multiplications in sequence on GPU : 0.009020805358886719 seconds
```

Out[151]: (2, 4096, 4096)

```
Two matrix-matrix multiplications at the same time on GPU : 0.004949331283569336 seconds
```