module_5_practice_2_assignment

April 12, 2020

1 Module 5 Lesson 4 Practice Assessment

This lesson covered how to write a training loop in MXNet with the Gluon API and how to write a validation loop to evaluate a trained network. This practice assessment will allow you to become more familiar with these concepts.

[1]:
```
from mxnet import nd, gluon, init, autograd, metric
from mxnet.gluon import nn
from mxnet.gluon.data.vision import datasets, transforms
from pathlib import Path
import os
```

[2]:
```
M5_DATA = Path(os.getenv('DATA_DIR', '../../data'), 'module_5')
M5_IMAGES = Path(M5_DATA, 'images')
M5_MODELS = Path(M5_DATA, 'models')
```

1.1 Prepare Dataset

First, let's prepare the dataset we'll use for the training exercise. We will take the popular MNIST dataset but convert the labels so that it becomes a binary classification problem. To do this, we will simply set the label of all digits greater than 0 to 1. This means that we now have two labels. 0 for digit images that correspond to the handwritten 0 and 1 for all other digits in the MNIST dataset.

[3]:
```
train_data = datasets.MNIST(train=True, root=M5_IMAGES)
train_data._label[train_data._label > 0] = 1

val_data = datasets.MNIST(train=False, root=M5_IMAGES)
val_data._label[val_data._label > 0] = 1

batch_size = 128
train_data = gluon.data.DataLoader(train_data.transform_first(transforms.
 ↪ToTensor()),
                                  batch_size = batch_size,
                                   shuffle=True)
val_data = gluon.data.DataLoader(val_data.transform_first(transforms.
 ↪ToTensor()),
                                  batch_size = batch_size,
                                  shuffle=False)
```

1.2 Question 1

Now you will get some practice defining a network that solve the binary classification task. Fill in the function to define a network based on the LeNet architecture from the lesson but instead of 10 output layers for the original MNIST data, you should have 2 output layers for the modified task. The function should also return the loss function for the classification problem.

[4]:
```
def get_network():
    """
    Should create the LeNet 10 network but with 2 output units instead of 10␣
↪and return a classification loss function

    :return: the network and the loss function
    :rtype: (gluon.Block, gluon.Block)
    """

    net = None
    loss_fn = None

    net = nn.Sequential()
    with net.name_scope():
        net.add(
            nn.Conv2D(channels=6, kernel_size = 5, activation='relu'),
            nn.MaxPool2D(pool_size=2, strides = 2),
            nn.Conv2D(channels=16,kernel_size = 3, activation='relu'),
            nn.MaxPool2D(pool_size=2, strides = 2),
            nn.Flatten(),
            nn.Dense(120, activation='relu'),
            nn.Dense(84, activation='relu'),
            nn.Dense(2)
        )

    net.initialize(init=init.Xavier())

    # choose and set loss_fn for a classification task
    loss_fn = gluon.loss.SoftmaxCrossEntropyLoss()

    return net, loss_fn
```

[5]:
```
n, loss_fn = get_network()

assert isinstance(n[0], nn.Conv2D)
assert isinstance(n[2], nn.Conv2D)
assert isinstance(n[1], nn.MaxPool2D)
assert isinstance(n[3], nn.MaxPool2D)

for l in n[-3:]:
    assert isinstance(l, nn.Dense)
```

If you implemented the above function correctly, the code cell below should print a 7 layer neural network similar to LeNet but with the final layer being a 2 unit fully-connected or Dense Layer. It should also print a description of the loss function.

[6]:
```
print(get_network())
```

```
(Sequential(
  (0): Conv2D(None -> 6, kernel_size=(5, 5), stride=(1, 1), Activation(relu))
  (1): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False,
global_pool=False, pool_type=max, layout=NCHW)
  (2): Conv2D(None -> 16, kernel_size=(3, 3), stride=(1, 1), Activation(relu))
  (3): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False,
global_pool=False, pool_type=max, layout=NCHW)
  (4): Flatten
  (5): Dense(None -> 120, Activation(relu))
  (6): Dense(None -> 84, Activation(relu))
  (7): Dense(None -> 2, linear)
), SoftmaxCrossEntropyLoss(batch_axis=0, w=None))
```

1.3 Question 2

In the function definition below write the training loop so you can use the function to train the network you defined earlier. As training progress, print out the current loss and training accuracy metric after every epoch. The loss function is passed in to the function as an argument but you can create a metric accumulator for accuracy using `mxnet.metric.Accuracy`. For model parameter updates you will need to construct a trainer. Use the following values for the optimization hyperparameters.

- optimizer - `sgd`
- learning_rate - `0.1`

[13]:
```
from time import time

def train(net, loss_fn, train_data, epochs, batch_size):
    """
    Should take an initialized network and train that network using data from␣
 ↪the data loader.

    :param network: initialized gluon network to be trained
    :type network: gluon.Block
    :param loss_fn: the loss function
    :type loss_fn: gluon.Block

    :param train_data: the training DataLoader provides batches for data for␣
↪every iteration
    :type train_data: gluon.data.DataLoader

    :param epochs: number of epochs to train the DataLoader
    :type epochs: int

    :param batch_size: batch size for the DataLoader.
    :type batch_size: int

    :return: tuple of trained network and the final training accuracy
    :rtype: (gluon.Block, float)
    """
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate':0.1})
    train_acc = metric.Accuracy()

    for epoch in range(epochs):
        train_loss = 0.
        tic = time()

        for data, label in train_data:
            with autograd.record():
                output = net(data)
                loss = loss_fn(output, label)
            loss.backward()

            trainer.step(batch_size)

            train_loss += loss.mean().asscalar()
            train_acc.update(label, output)

        print("Epoch [%d], Loss: %.3f, Acc: %3f" %(
            epoch, train_loss/len(train_data), train_acc.get()[1]))

    net.save_parameters("trained_net.params")

    return net, train_acc.get()[1]
```

We're running through the dataset twice (i.e. 2 epochs) in the next cell, so this could take a few minutes.

```
[14]: n, ta = train(*get_network(), train_data, 2, batch_size)
      assert ta >= .98
```

```
Epoch [0], Loss: 0.054, Acc: 0.981750
Epoch [1], Loss: 0.016, Acc: 0.988200
```

If you have implemented the function above correctly, the code below should print a line after every epoch and you should the loss go down and the accuracy go up. After 5 epochs you should have accuracies that are well past .99 (but this might take a few minutes to run).

[19]:
```
net, ta = train(*get_network(), train_data, 5, batch_size)
```

```
Epoch [0], Loss: 0.047, Acc: 0.984267
Epoch [1], Loss: 0.014, Acc: 0.989933
Epoch [2], Loss: 0.010, Acc: 0.992178
Epoch [3], Loss: 0.007, Acc: 0.993550
Epoch [4], Loss: 0.006, Acc: 0.994430
```

1.4 Question 3

Now fill in the function below to evaluate the model on the validation dataset. The function should only do one pass through the validation dataset and should report back the model accuracy.

[20]:
```
def evaluate(network, dataloader):
    """
    Should compute the accuracy of the network on the validation set.

    :param network: initialized gluon network to be trained
    :type network: gluon.Block

    :param dataloader: the validation DataLoader provides batches for data for␣
↪every iteration
    :type dataloader: gluon.data.DataLoader

    :return: validation accuracy
    :rtype: float
    """
    preds = []
    valid_acc = metric.Accuracy()
    network.load_parameters("trained_net.params")

    for data, label in dataloader:
        output = network(data)
        valid_acc.update(label, output)

    return valid_acc.get()[1]
```

[21]:
```
assert evaluate(net, val_data) > .98
```

```
Validation accuracy: 0.996
```

If you implemented the function above correctly, then the cell below should print a validation accuracy of around 0.99 as well.

```
[22]: print("Validation Acc: %.3f "%(evaluate(net, val_data)))
```

```
Validation accuracy: 0.996
Validation Acc: 0.996
```

```
[ ]:
```