LAB1

January 20, 2025

1 Lab 1

This very first lab will introduce to some PyTorch basics including Tensors, Loss, and Autograd. Hopefully, we will start to get familar with PyTorch. In the end, we will implement a linear regression model from sratch with some synthetic data.

Table of Contents: - Tensors - Loss - Autograd - Assignment

Some contents of this lab are adapted from Dive into Deep Learning and Official PyTorch Tutorials.

[ ]:
```
import os
import torch
import numpy as np
import matplotlib.pyplot as plt
import random
```

[ ]:
```
# set seed
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
os.environ["PYTHONHASHSEED"] = str(seed)
```

1.1 1. Tensors

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators. Tensors are also optimized for automatic differentiation (we'll see more about that later in the Autograd section).

1.1.1 1.1 Initializing a Tensor

Tensors can be initialized in various ways. Take a look at the following examples:

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

[ ]:
```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
x_data
```

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa).

[ ]:
```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
x_np
```

From another tensor:

The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden.

[ ]:
```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of␣
↪x_data
print(f"Random Tensor: \n {x_rand} \n")
```

With random or constant values:

`shape` is a tuple of tensor dimensions. In the functions below, it determines the dimensionality of the output tensor.

[ ]:
```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

1.1.2 1.2 Attributes of a Tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

[ ]:
```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

1.1.3 1.3 Operations on Tensors

Over 100 tensor operations, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling and more are comprehensively described here.

Let us try out some of the operations from the list. They are pretty similar to the NumPy API.

Standard numpy-like indexing and slicing:

[ ]:
```
tensor = torch.arange(12).reshape(3, 4).float()
print(tensor)
print('First row: ',tensor[0])
print('First column: ', tensor[:, 0])
print('Last column:', tensor[:, -1])
```

Joining tensors

You can use `torch.cat` to concatenate a sequence of tensors along a given dimension. See also `torch.stack`, another tensor joining op that is subtly different from `torch.cat`.

[ ]:
```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

Arithmetic operations

[ ]:
```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will␣
↪have the same value
y1 = tensor @ tensor.T
print(y1)
y2 = tensor.matmul(tensor.T)
print(y2)
y3 = torch.rand_like(tensor)
torch.matmul(tensor, tensor.T, out=y3)
print(y3)
```

[ ]:
```
# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
print(z1)
z2 = tensor.mul(tensor)
print(z2)
z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
print(z3)
```

Single-element tensors

If you have a one-element tensor, for example by aggregating all values of a tensor into one value, you can convert it to a Python numerical value using `item()`:

[ ]:
```
agg = tensor.sum()
agg_item = agg.item()
```

```
print(agg_item, type(agg_item))
```

1.1.4 1.4 GPU Acceleration

If we have NVIDIA GPU(s), we can accelerate computation once we move Tensors onto GPU. Let's compare how much GPU can accelerate especially matrix operations. We will do a matrix-matrix multiplication between two 5k-by-5k matrices on both CPU and GPU.

Unfortunately, Coursera does not have a GPU environment. But feel free to try the following snippets on a GPU machine. Ideally, with GPU acceleration, matrix multiplication will be much faster.

[ ]:
```
mat = torch.rand(5000, 5000)
mat
```

[ ]:
```
%%time
torch.mm(mat.t(), mat)
```

[ ]:
```
%%time
if torch.cuda.is_available():
    mat = mat.cuda()
    torch.mm(mat.t(), mat)
else:
    print('GPU is not available!')
```

1.1.5 Exercise 1 [10 points]

Implement the Sigmoid function on your own.

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Note that you should not use existing PyTorch implementation.

Hint: try `torch.exp()`.

[ ]:
```
def sigmoid(x):
    # your code here
    raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert torch.allclose(sigmoid(torch.tensor([1.2])), torch.tensor([0.7685]),␣
↪rtol=1e-2)
assert torch.allclose(sigmoid(torch.tensor([0, 1.5])), torch.tensor([0.5000, 0.
↪8176]), rtol=1e-2)
```

1.1.6 Exercise 2 [10 points]

Implement a Softmax function on your own.

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}$$

Note that you should not use existing PyTorch implementation.

Hint: try `torch.exp()` and `torch.sum()`.

[ ]:
```
def softmax(X):
    # your code here
    raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

X = torch.tensor([[0.2288, 0.4111, 0.0385], [0.6233, 0.0364, 0.1999]])
assert torch.allclose(softmax(X), torch.tensor([[0.3304, 0.3965, 0.2731], [0.
↪4523, 0.2515, 0.2962]]), rtol=1e-2)
```

1.1.7 Exercise 3 [10 points]

Implement a linear layer.

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b},$$

where $\mathbf{X}$ is the input feature, $\mathbf{O}$ is the output feature, $\mathbf{W}$ and $\mathbf{b}$ are the weight parameters.

Hint: try `torch.matmul()`.

[ ]:
```
def linear(X, W, b):
    # your code here
    raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

X = torch.Tensor([[0.1, 0.2, 0.3]])
W = torch.Tensor([[0.1, 0.2, 0.3]]).T
b = torch.Tensor([-0.5])
assert torch.allclose(linear(X, W, b), torch.Tensor([[-0.3600]]), rtol=1e-2)
```

1.2 2. Loss

When presented with some training data, our untrained network is likely not to give the correct answer. Loss function measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training. To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value.

Common loss functions include `nn.MSELoss` (Mean Square Error) for regression tasks, and `nn.NLLLoss` (Negative Log Likelihood) for classification. `nn.CrossEntropyLoss` combines `nn.LogSoftmax` and `nn.NLLLoss`. `nn.BCELoss` is specially designed for binary classification.

Mean Square Error

The most popular loss function in regression problems is the squared error. When our prediction for an example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the squared error is given by:

$$l^{(i)} = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2.$$

To measure the quality of a model on the entire dataset of n examples, we simply average (or equivalently, sum) the losses on the training set.

$$L = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}.$$

Let us see how to implement this.

[ ]:
```
def squared_loss(y_hat, y):
    return ((y_hat - y.reshape(y_hat.shape)) ** 2 / 2).mean()
```

[ ]:
```
y_hat = torch.tensor([0.5, 0.8, 0.2])
y = torch.tensor([1.0, 1.0, 0.0])
squared_loss(y_hat, y)
```

1.2.1 Exercise 4 [10 points]

Implement the cross-entropy loss function on your own.

$$l^{(i)} = -\sum_{j=1}^{q} y_j^{(i)} \log \hat{y}_j^{(i)},$$

$$L = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}.$$

where $\mathbf{y}^{(i)}$ is a one-hot vector of length q, the sum over all its coordinates j vanishes for all but one term.

Note that you should not use existing PyTorch implementation.

Hint: try `torch.log()`.

[ ]:
```
def cross_entropy(y_hat, y):
    # your code here
    raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

y = torch.tensor([[1, 0, 0], [0, 0, 1]])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
assert torch.allclose(cross_entropy(y_hat, y), torch.tensor([1.4979]),␣
 ↪rtol=1e-2)
```

1.3 3. Autograd

When training the model, we want to find parameters (denoted as Θ) that minimize the total loss across all training examples:

$$\Theta = \underset{\Theta}{\operatorname{argmin}} \; L(\Theta).$$

To do this, we will iteratively reduce the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called gradient descent. The most naive application of gradient descent consists of taking the derivative of the loss function. Let us see how to do this.

As a toy example, say that we are interested in differentiating the function $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to the column vector $\mathbf{x}$. To start, let us create the variable x and assign it an initial value.

[ ]:
```
x = torch.arange(4.0)
x
```

[ ]:
```
x.requires_grad_(True)  # Same as `x = torch.arange(4.0, requires_grad=True)`
x.grad  # The default value is None
```

Now let us calculate y.

[ ]:
```
y = 2 * torch.dot(x, x)
y
```

Since $\mathbf{x}$ is a vector of length 4, an inner product of $\mathbf{x}$ and $\mathbf{x}$ is performed, yielding the scalar output that we assign to y. Next, we can automatically calculate the gradient of y with respect to each component of $\mathbf{x}$ by calling the function for backpropagation and printing the gradient.

[ ]:
```
y.backward()
x.grad
```

The gradient of the function $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$. Let us quickly verify that our desired gradient was calculated correctly.

[ ]:
```
x.grad == 4 * x
```

1.3.1 Exercise 5 [10 points]

Let $f(x) = \sin(x)$. Plot $f(x)$ and and $\frac{df(x)}{dx}$, where the latter is computed without exploiting that $f'(x) = \cos(x)$.

[ ]:
```
x = np.linspace(-np.pi, np.pi, 100)
x = torch.tensor(x, requires_grad=True)
y = torch.sin(x)
# your code here
raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert torch.allclose(x.grad[10].float(), torch.Tensor([-0.8053]), rtol=1e-2)
assert torch.allclose(x.grad[50].float(), torch.Tensor([0.9995]), rtol=1e-2)
```

1.4 Assignment [50 points]

Now, you understand the basics of PyTorch. Let us implement an entire method from scratch, including the data pipeline, the linear model, the loss function, and the gradient descent optimizer. While modern deep learning frameworks can automate nearly all of this work, implementing things from scratch is the only way to make sure that you really know what you are doing. Moreover, when it comes time to customize models, defining our own layers or loss functions, understanding how things work under the hood will prove handy. In this section, we will rely only on tensors and auto differentiation. Afterwards, we will introduce a more concise implementation, taking advantage of bells and whistles of deep learning frameworks.

To keep things simple, we will construct an artificial dataset according to a linear model with additive noise. Our task will be to recover this model's parameters using the finite set of examples contained in our dataset. We will keep the data low-dimensional so we can visualize it easily. In the following code snippet, we generate a dataset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution. Thus our synthetic dataset will be a matrix $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

The true parameters generating our dataset will be $\mathbf{w} = [2, -3.4]^\top$ and $b = 4.2$, and our synthetic labels will be assigned according to the following linear model with the noise term ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon.$$

[ ]:
```
def synthetic_data(w, b, num_examples):
    """Generate y = Xw + b + noise."""
    X = torch.normal(0, 1, (num_examples, len(w)))
```

```
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))

true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

[ ]:
```
print('features:', features[0], '\nlabel:', labels[0])
```

1.4.1 Reading the Dataset

Recall that training models consists of making multiple passes over the dataset, grabbing one minibatch of examples at a time, and using them to update our model. Since this process is so fundamental to training machine learning algorithms, it is worth defining a utility function to shuffle the dataset and access it in minibatches.

In the following code, we define the `data_iter` function to demonstrate one possible implementation of this functionality. The function takes a batch size, a matrix of features, and a vector of labels, yielding minibatches of the size batch_size. Each minibatch consists of a tuple of features and labels.

[ ]:
```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(indices[i:min(i + batch_size,␣
↪num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

In general, note that we want to use reasonably sized minibatches to take advantage of the GPU hardware, which excels at parallelizing operations. Because each example can be fed through our models in parallel and the gradient of the loss function for each example can also be taken in parallel, GPUs allow us to process hundreds of examples in scarcely more time than it might take to process just a single example.

To build some intuition, let us read and print the first small batch of data examples. The shape of the features in each minibatch tells us both the minibatch size and the number of input features. Likewise, our minibatch of labels will have a shape given by batch_size.

[ ]:
```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

As we run the iteration, we obtain distinct minibatches successively until the entire dataset has

been exhausted (try this). While the iteration implemented above is good for didactic purposes, it is inefficient in ways that might get us in trouble on real problems. For example, it requires that we load all the data in memory and that we perform lots of random memory access. The built-in iterators implemented in a deep learning framework are considerably more efficient and they can deal with both data stored in files and data fed via data streams.

1.4.2 Initializing Model Parameters [10 points]

Before we can begin optimizing our model's parameters by minibatch stochastic gradient descent, we need to have some parameters in the first place. In the following code, we initialize weights by sampling random numbers from a normal distribution with mean 0 and a standard deviation of 0.01, and setting the bias to 0.

[ ]:
```
w = None # size (2,1)
b = None # size (1,)
# your code here
raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert type(w) is torch.Tensor
assert type(b) is torch.Tensor
assert w.requires_grad
assert b.requires_grad
```

1.4.3 Defining the Model [10 points]

Next, we must define our model, relating its inputs and parameters to its outputs. Recall that to calculate the output of the linear model, we simply take the matrix-vector dot product of the input features $\mathbf{x}$ and the model weights $\mathbf{w}$, and add the offset b to each example.

This is exactly the same as what we implemented in Exercise 3.

[ ]:
```
def linear(X, W, b):
    # your code here
    raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

X_ = torch.Tensor([[0.1, 0.2, 0.3]])
W_ = torch.Tensor([[0.1, 0.2, 0.3]]).T
b_ = torch.Tensor([-0.5])
assert torch.allclose(linear(X_, W_, b_), torch.Tensor([[-0.3600]]), rtol=1e-2)
```

1.4.4 Defining the Loss Function [10 points]

Since updating our model requires taking the gradient of our loss function, we ought to define the loss function first. Here we will use the squared loss function (remember use 1/2 * (a-b)^2).

[ ]:
```
def squared_loss(y_hat, y):
    # your code here
    raise NotImplementedError
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

y = torch.tensor([2, 0.4, 1.1, 2.1])
y_hat = torch.tensor([[2.1, 0.3, 1.2, 2.5]])
assert torch.allclose(squared_loss(y_hat, y), torch.tensor([0.0238]), rtol=1e-2)
```

1.4.5 Defining the Optimization Algorithm [10 points]

Though linear regression has a closed-form solution, we will take this opportunity to introduce your first working example of minibatch stochastic gradient descent.

At each step, using one minibatch randomly drawn from our dataset, we will estimate the gradient of the loss with respect to our parameters. Next, we will update our parameters in the direction that may reduce the loss. The following code applies the minibatch stochastic gradient descent update, given a set of parameters, a learning rate, and a batch size. The size of the update step is determined by the learning rate lr. Because our loss is calculated as a sum over the minibatch of examples, we normalize our step size by the batch size (batch_size), so that the magnitude of a typical step size does not depend heavily on our choice of the batch size.

[ ]:
```
def sgd(params, lr, batch_size):
    """  Minibatch stochastic gradient descent """
    with torch.no_grad():
        for param in params:
            """
            TODO: perform a step gradient descent
                  param = param - lr * grad
            """
            # your code here
            raise NotImplementedError
            param.grad.zero_()
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

m = torch.ones(1, requires_grad=True)
n = 2 * m
n.backward()
```

```
sgd([m], lr=0.1, batch_size=1)
assert torch.allclose(m, torch.tensor([0.8000]), rtol=1e-2)
```

1.4.6 Training [10 points]

Now that we have all of the parts in place, we are ready to implement the main training loop. It is crucial that you understand this code because you will see nearly identical training loops over and over again throughout your career in deep learning.

In each iteration, we will grab a minibatch of training examples, and pass them through our model to obtain a set of predictions. After calculating the loss, we initiate the backwards pass through the network, storing the gradients with respect to each parameter. Finally, we will call the optimization algorithm sgd to update the model parameters.

In summary, we will execute the following loop:

[ ]:
```
lr = 0.03
num_epochs = 20
net = linear
loss = squared_loss

for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Compute gradient on `l` with respect to [`w`, `b`]
        l.backward()
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

[ ]:
```
print(f'error in estimating w: {true_w - w.reshape(true_w.shape)}')
print(f'error in estimating b: {true_b - b}')
```

[ ]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert (true_w - w.reshape(true_w.shape)).abs().mean() < 0.05
assert (true_b - b).abs() < 0.05
```