Linear Regression Implementation

- Goal: implement the entire method from scratch, including the data pipeline, the model, the loss function, and the minibatch stochastic gradient descent optimizer.
    - Deep learning frameworks can automate nearly all of this work!
- For training we need the following:
    - Create and read dataset
    - Create linear regression model ($y = \mathbf{w}^T\mathbf{x} + b$)
    - Initialize parameters $(\mathbf{w}, b)$
    - Define loss function (squared loss)
- Then, we execute the following loop:
    - Repeat until done
        - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
        - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta\mathbf{g}$

Concise Implementation of Linear Regression

- Deep Learning algorithms are implemented using frameworks like PyTorch
- So far from PyTorch we relied only on (i) tensors for data storage and linear algebra; and (ii) auto differentiation for calculating gradients.
- In practice, data iterators, loss functions, optimizers, and neural network layers are provided as well.
- Here, we will see how to implement the linear regression model concisely by using the high-level API of PyTorch.

Generating the Dataset

- We construct an artificial dataset according to a linear model with additive noise.
- Goal: to recover the model's parameters using the dataset.
- Dataset contains 1000 examples, each consisting of 2 features sampled from a Gaussian distribution. Dataset is represented by a matrix $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.
- True parameters (generating our dataset): $\mathbf{w} = [2, -3.4]^\top$ and $b = 4.2$.
- Synthetic labels: according to the following linear model with noise term ϵ:

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

- ϵ captures potential measurement errors on the features and labels.
- Standard assumptions: ϵ is zero-mean Gaussian. We set its std to 0.01.

In [2]:

```python
# Generating the Dataset
def synthetic_data(w, b, num_examples):
    """Generate y = Xw + b + noise."""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.mm(X, w) + b
    y += torch.normal(0, 0.01, y.size())
    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)
```

Reading the Dataset

- We need 2 things: (a) A way to access the dataset and (b) A way to iterate through it.
- For (a), PyTorch has an abstract Dataset class. A Dataset can be anything that has a `__len__` function (called by Python's standard `len` function) and a `__getitem__` function as a way of indexing into it.
- PyTorch's TensorDataset is a Dataset wrapper for tensors. By defining a length and way of indexing, this also gives us a way to iterate, index, and slice along the first dimension of a tensor.
    - In general we will have to implement our own Dataset class, extending `torch.utils.data.Dataset`
- For (b), we use `torch.utils.data.DataLoader`. In addition to iterate through the dataset, this also provides built-in functionality for: 1. Batching the data, 2. Shuffling the data and 3. **Load the data in parallel using multiprocessing workers.**

In [3]:

```
dataset = data.TensorDataset(features, labels) #TensorDataset object
print(dataset[0]) #First example in our dataset
batch_size = 10
data_iter =  data.DataLoader(dataset, batch_size, shuffle=True) #DataLoader object
```

```
(tensor([1.3314, 0.0222]), tensor([6.7981]))
```

```python
class myDataset(torch.utils.data.Dataset):
    """My dataset."""

    def __init__(self, csv_file, root_dir):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
        """
        self.labels = pd.read_csv(csv_file)
        self.root_dir = root_dir

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        # read image
        img_name = os.path.join(self.root_dir, self.labels.iloc[idx, 0])
        image = io.imread(img_name)
        # read labels
        labels = self.labels.iloc[idx, 1:]

        sample = {'image': image, 'labels': labels}
        return sample
```

Reading the Dataset

- `data_iter` is iterable object: we can use `iter` to construct a Python iterator and use `next` to obtain the items from it.

```
data_iterator = iter(data_iter)
```

```
next(data_iterator)
```

```
[tensor([[-0.2104, -2.9304],
         [-0.0618, -0.1305],
         [-0.7488, -1.3081],
         [ 1.5559,  0.3034],
         [ 0.3386,  1.1037],
         [-0.9248, -0.3067],
         [-0.7349,  0.8960],
         [ 0.7369, -0.6641],
         [-3.0672,  0.8497],
         [-0.8393,  0.7160]]),
 tensor([[13.7376],
         [ 4.5224],
         [ 7.1611],
         [ 6.2903],
         [ 1.1231],
         [ 3.4113],
         [-0.3085],
         [ 7.9220],
         [-4.8186],
         [ 0.0899]])]
```

Defining the Model

- When we implement linear regression from scratch, we must define our model parameters explicitly and code up the calculations to produce output using basic linear algebra operations.
- But once models get more complex, and for standard operations, we can use PyTorch's predefined layers.
- For Linear Regression we need a single Linear (or Fully-Connected) layer: each of its inputs is connected to each of its outputs by means of a matrix-vector multiplication.
- In PyTorch, the FC layer is defined in the `Linear` class. Note that we pass two arguments into `nn.Linear`.

In [5]:
```
num_of_inp, num_of_out = 2, 1 # input, output feature dimension
net = nn.Linear(num_of_inp, num_of_out)
```

```
def linreg(X, w, b):
    """The linear regression model: y=Xw+b
       X: [num_of_examples, num_of_feat]
       w: [num_of_feat, 1]
       b: scalar
       y: [num_of_examples, 1]"""
    return torch.mm(X, w) + b
```

Initializing Model Parameters

- Deep learning frameworks have a predefined way to initialize the parameters.
- We can access the parameters directly to specify the initial values:
    - We use the `weight.data` and `bias.data` methods to access the parameters.
    - Finally use the inplace methods `normal_` and `fill_` to overwrite parameter values.

In [6]:

```
net.weight.data.normal_(0, 0.01); # each weight sampled from a Gaussian with mean 0 and std 0.01.
net.bias.data.fill_(0); # bias initialized to 0.
```

Defining the Loss Function

- The `MSELoss` class computes the mean squared error, also known as squared L_2 norm.
- By default it returns the average loss over examples.

In [7]:
```
loss = nn.MSELoss()
```

```
def squared_loss(y_hat, y):
    """Squared loss."""
    return (y_hat - y.reshape(y_hat.size())) ** 2 / 2
```

Defining the Optimization Algorithm

- Minibatch SGD is a standard tool for optimizing neural networks and thus PyTorch supports it
  - A number of variations of this algorithm can be found in the `optim` module.
- To initialize an `SGD` optimizer we need:
  - The parameters to optimize (obtainable from our net via `net.parameters()`),
  - A dictionary of hyperparameters required.
    - For now we just require to set the `lr`

In [8]:
```
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)
```

Training

- The training loop is similar to the one from implementing everything from scratch:
    - For each epoch, we will make a complete pass over the dataset, iteratively grabbing one minibatch of inputs and ground-truth labels.
    - For each minibatch:
        - Generate predictions by calling `net(X)` and calculate the loss `l` (the forward pass).
        - Calculate gradients by running the backpropagation.
        - Update the model parameters by invoking our optimizer.
- Compute the loss after each epoch and print it to monitor progress.

In [9]:

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        y_hat = net(X)
        #print(y.size(), y_hat.size())
        l = loss(y_hat, y)
        optimizer.zero_grad()
        l.backward()
        optimizer.step()
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')
```

```
epoch 1, loss 0.000403
epoch 2, loss 0.000104
epoch 3, loss 0.000104
```

Evaluation

- Below, we compare the model parameters learned by training on finite data and the actual parameters that generated our dataset.

In [10]:
```
w = net.weight.data
print('error in estimating w:', true_w - w.reshape(true_w.shape))
b = net.bias.data
print('error in estimating b:', true_b - b)
```

```
error in estimating w: tensor([[-0.0006],
        [-0.0005]])
error in estimating b: tensor([7.0572e-05])
```

Summary

- Using PyTorch's high-level API, we can implement models much more concisely.
- In PyTorch, the `data` module provides tools for data processing, the `nn` module defines a large number of neural network layers and common loss functions.
- Initialize the parameters by replacing their values with methods ending with `_`.