HWK4

February 27, 2025

CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2025

Homework 4 - PyTorch

Due Sunday, March 2, at 11:59 PM

Do not redistribute without the instructor's written permission.

```
[36]: import torch
      import torch.nn as nn
      from torch.nn.functional import mse_loss
```

0.1 Part 1: Overview of built-in functions

PyTorch contains many pre-defined functions for tensor operations beyond simple numerical ones. Most functions which operate on a single tensor also have an analogous verison when called from the tensor.

See PyTorch's tensor class reference for a more, but the examples below are the ones relevant to this assignment.

```
[37]: # construct an example tensor
      x = torch.tensor([-1.5330,  0.4530, -0.7361, -0.3403,  2.4078])
      x
```

```
[37]: tensor([-1.5330,  0.4530, -0.7361, -0.3403,  2.4078])
```

The `sigmoid` activation function, typically represented by the symbol σ, is defined as:

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

When applied to a tensor, the function is applied per-element to produce a new tensor with the same shape.

```
[38]: # sigmoid is predefined for you within PyTorch and can be applied to any tensor
      torch.sigmoid(x)
```

```
[38]: tensor([0.1776, 0.6114, 0.3239, 0.4157, 0.9174])
```

[39]:
```
# sigmoid can also be called using this alternate notation
x.sigmoid()
```

[39]:
```
tensor([0.1776, 0.6114, 0.3239, 0.4157, 0.9174])
```

The `relu` activation function is defined as:

$$\text{relu}(x) = \max(0, x)$$

Like `sigmoid`, it is applied to tensors per-element

[40]:
```
# positive values remain, while negative values become zero
torch.relu(x)
```

[40]:
```
tensor([0.0000, 0.4530, 0.0000, 0.0000, 2.4078])
```

[41]:
```
# relu can also be called using the alternate notation
x.relu()
```

[41]:
```
tensor([0.0000, 0.4530, 0.0000, 0.0000, 2.4078])
```

0.2 Part 2: Implementing a neural network by hand

Now that you are an expert in PyTorch tensors, we can start creating and using a neural network. Consider this (tiny) neural network:

Recall that an artificial neural network is composed of layers of connections between nodes. Each node performs a weighted aggregation of outputs from the previous layer and applies a nonlinearity to produce its output. See here for a refresher of neural networks terminology.

Our example network has 3 input nodes, a hidden layer with 4 nodes, and 2 output nodes. The network has two layers, one being the transformation from input to the hidden layer, and the other the transformation from the hidden layer to output. We will use ReLU activation as the nonlinearity for the first layer and sigmoid as the nonlinearity for the second layer (see formalization in exercise 1).

We will work on a synthetic dataset `(X, Y)`, drawn from the distribution defined in the function `generate_data`. Understanding the exact nature of this distribution is not necessary: you only need to know that it defines some deterministic function mapping from 3-dimensional vectors to 2-dimensional vectors. Notably, the output is binary (either 0, or 1), hence why we use sigmoid activation for the last layer (output). Note that this is not a classification task; there's simply two output features, both of which can be either 0 or 1.

[42]:
```
def generate_data(N):
    X = torch.randn(N, 3)
    Y = (torch.stack([
        (X[:, 0] + X[:, 1]) * (X[:, 2] > 0) + (X[:, 0] - X[:, 1]) * (X[:, 2] <␣
 ↪0),
        X.norm(dim=1)
    ]).t().abs() > 1.33).float()
    return X, Y
```

[43]:
```
# fixed seed for reproducible results
torch.manual_seed(0)

X, Y = generate_data(20)
```

[44]:
```
X
```

[44]:
```
tensor([[-1.1258, -1.1524, -0.2506],
        [-0.4339,  0.8487,  0.6920],
        [-0.3160, -2.1152,  0.3223],
        [-1.2633,  0.3500,  0.3081],
        [ 0.1198,  1.2377,  1.1168],
        [-0.2473, -1.3527, -1.6959],
        [ 0.5667,  0.7935,  0.5988],
        [-1.5551, -0.3414,  1.8530],
        [ 0.7502, -0.5855, -0.1734],
        [ 0.1835,  1.3894,  1.5863],
        [ 0.9463, -0.8437, -0.6136],
        [ 0.0316, -0.4927,  0.2484],
        [ 0.4397,  0.1124,  0.6408],
        [ 0.4412, -0.1023,  0.7924],
        [-0.2897,  0.0525,  0.5943],
        [ 1.5419,  0.5073, -0.5910],
        [-0.5692,  0.9200,  1.1108],
        [ 1.2899, -1.4959, -0.1938],
        [ 0.4455,  1.3253, -1.6293],
        [-0.5497, -0.4798, -0.4997]])
```

[45]:
```
Y
```

[45]:
```
tensor([[0., 1.],
        [0., 0.],
        [1., 1.],
        [0., 1.],
        [1., 1.],
        [0., 1.],
        [1., 0.],
        [1., 1.],
        [1., 0.],
        [1., 1.],
        [1., 1.],
        [0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 1.],
        [0., 1.],
        [1., 1.],
        [0., 1.],
        [0., 0.]])
```

In order to implement the weighted aggregation for each node, we can represent the weights of a layer as a matrix, and add a vector as the bias. Initially, these matrices are randomly initialized and subsequently optimized such that the network's performance improves.

Parameters for the above toy network are defined below. In this case, we have: - a linear layer from 3 to 4 nodes (3x4 matrix) - bias for the 4 hidden nodes (1x4 matrix) - a linear layer from 4 to 2 nodes (4x2 matrix) - bias for the 2 output nodes (1x2 matrix)

Take a moment to make sure it's clear why the parameters are defined as such.

[46]:
```
# initialize random values for NN weights and biases
W1 = torch.randn(3, 4) * 0.2
B1 = torch.randn(1, 4) * 0.2
W2 = torch.randn(4, 2) * 0.2
B2 = torch.randn(1, 2) * 0.2
```

0.2.1 Exercise 1 (20 pts)

Manually perform a forward pass on the toy neural network

First, we calculate the pre-activation values z_1 for the hidden layer, using the input x directly. The hidden node values h are computed by applying the `relu` function, which is defined as an element-wise $\mathrm{relu}(x) := \max(0, x)$. We use this to compute the pre-activation values z_2 of the output layer. The output node values are computed with the `sigmoid` activation function.

The following equations are the steps needed to calculate the forward pass for the neural network given some input x, with $\times$ representing matrix multiplication.

$$\begin{aligned} z_1 &= x \times W_1 + B_1 \\ h &= \mathrm{relu}(z_1) \\ z_2 &= h \times W_2 + B_2 \\ \hat{y} &= \sigma(z_2) \end{aligned}$$

Look near the end of the tensors tutorial for examples of how to use PyTorch's implementations of `relu` and `sigmoid`, as well as how to do matrix multiplication using `@`.

[47]:
```
import torch.nn.functional as F
```

[48]:
```
def forward(x):
    # Compute pre-activation values for hidden layer
    z1 = x @ W1 + B1
    # Apply ReLU activation function to get hidden layer values
    h = torch.relu(z1)
    # Compute pre-activation values for output layer
    z2 = h @ W2 + B2
    # Apply sigmoid activation function to get output values
    y_hat = torch.sigmoid(z2)

    return y_hat
```

[49]:

```
# the network's output should have the same shape as Y, but totally wrong values
y_hat = forward(X)
y_hat
```

[49]:

```
tensor([[0.5900, 0.6397],
        [0.6077, 0.6478],
        [0.6165, 0.6393],
        [0.6050, 0.6484],
        [0.6133, 0.6459],
        [0.5693, 0.6405],
        [0.6051, 0.6452],
        [0.6458, 0.6463],
        [0.6025, 0.6453],
        [0.6236, 0.6454],
        [0.5940, 0.6452],
        [0.6130, 0.6469],
        [0.6151, 0.6455],
        [0.6219, 0.6453],
        [0.6153, 0.6475],
        [0.5856, 0.6438],
        [0.6178, 0.6478],
        [0.6134, 0.6439],
        [0.5864, 0.6476],
        [0.5897, 0.6459]])
```

Finally, we will use mean squared error (mse) as the numerical representation of how poorly the neural network performed on the task–i.e., the *loss function*. Mean squared error is defined as the mean squared L2 norm between the true labels Y and the predicted labels $\hat{y}$, i.e. $\sum_{i=1}^{|y|}(\hat{y}_i - Y_i)^2$, or equivalently mean($||\hat{y} - Y||_2^2$). Since the vector size n is fixed, MSE is always proportional to taking the sum instead of mean:

$$\mathrm{L} = ||\hat{y} - Y||_2^2$$

We will be using this definition of L for easier computation. In PyTorch, this implies using mean squared error with `reduction = 'sum'`.

[50]:

```
# loss is a numerical representation of how "poorly" the NN did the task
loss = ((y_hat - Y)**2).sum()
```

```
loss
```

[50]:
```
tensor(10.3925)
```

[51]:
```
# there's also a built-in function to do this
mse_loss(y_hat, Y, reduction='sum')
```

[51]:
```
tensor(10.3925)
```

0.2.2 Exercise 2 (30 pts)

Convert the provided math into code that manually performs the backward pass on the toy neural network. Modify the next code cell for this.

Reference exercise 1 for forward pass and loss calculation.

Note the derivatives of the activation functions:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$
$$\text{relu}'(x) = H(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

`relu` isn't differentiable at $x = 0$, but we can hardcode a value for the $x = 0$ derivative for practical purposes.

If we break it apart like shown below, we can easily compute gradients by repeatedly using the chain rule.

$$\begin{aligned}
\frac{\partial \mathrm{L}}{\partial \hat{y}} &= 2(\hat{y} - Y) \\
\frac{\partial \mathrm{L}}{\partial z_2} &= \frac{\partial \mathrm{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} = \frac{\partial \mathrm{L}}{\partial \hat{y}} \cdot \frac{\partial \sigma(z_2)}{\partial z_2} \\
&= \frac{\partial \mathrm{L}}{\partial \hat{y}} \cdot \sigma(z_2)(1 - \sigma(z_2)) \\
&= \frac{\partial \mathrm{L}}{\partial \hat{y}} \cdot \hat{y}(1 - \hat{y}) \\
\frac{\partial \mathrm{L}}{\partial h} &= \frac{\partial \mathrm{L}}{\partial z_2} \frac{\partial z_2}{\partial h} = \frac{\partial \mathrm{L}}{\partial z_2} W_2 \\
\frac{\partial \mathrm{L}}{\partial z_1} &= \frac{\partial \mathrm{L}}{\partial h} \frac{\partial h}{\partial z_1} = \frac{\partial \mathrm{L}}{\partial h} H(z_1)
\end{aligned}$$

$H(z_1)$ is easiest computed with `(z1 >= 0).float()` in PyTorch, or something similar.

Computations for weight and bias gradients are provided. Bias gradients are just aggregated pre-activation node value gradients; weight gradients are an outer product of the node outputs with pre-activation node value gradients, aggregated over the batch.

[52]:

```
def backprop(x):
    """
    Forward pass
    """
    z1 = x @ W1 + B1
    h = torch.relu(z1)
    z2 = h @ W2 + B2
    y_hat = torch.sigmoid(z2)

    """
    Backward pass
    """
    # Compute gradient of loss with respect to y_hat: 2 * (y_hat - Y)
    y_hat_grad = 2 * (y_hat - Y)

    # Compute gradient with respect to z2
    # dL/dz2 = dL/dy_hat * dy_hat/dz2 = dL/dy_hat * y_hat * (1 - y_hat)
    z2_grad = y_hat_grad * y_hat * (1 - y_hat)

    # Compute gradient with respect to h
    # dL/dh = dL/dz2 * dz2/dh = dL/dz2 * W2
    h_grad = z2_grad @ W2.t()

    # Compute gradient with respect to z1
    # dL/dz1 = dL/dh * dh/dz1 = dL/dh * relu'(z1) = dL/dh * (z1 >= 0).float()
    z1_grad = h_grad * (z1 >= 0).float()

    # calculate parameter gradients
    B2_grad = z2_grad.sum(0)
    W2_grad = (h[:,:,None] * z2_grad[:,None,:]).sum(0)
    B1_grad = z1_grad.sum(0)
    W1_grad = (X[:,:,None] * z1_grad[:,None,:]).sum(0)

    # output tuple of gradients for all parameters
    return W1_grad, B1_grad, W2_grad, B2_grad
```

[53]:

```
backprop(X)
```

[53]:

```
(tensor([[ 0.0024,  0.1060, -0.3577,  0.0399],
         [-0.0191,  0.0088,  0.1067, -0.0391],
         [-0.0850,  0.3183,  0.0782,  0.0288]]),
 tensor([-0.1472, -0.1702,  0.3098, -0.1245]),
 tensor([[ 0.0102,  0.0283],
         [ 0.0862, -0.1289],
         [ 0.2990,  0.1101],
         [ 1.0258,  0.2250]]),
 tensor([1.9934, 0.4087]))
```

[54]:

```
# take note of the loss we have before optimization
y_hat = forward(X)
loss = ((y_hat - Y)**2).sum()
loss
```

[54]:

```
tensor(10.3925)
```

After implementing backprop, gradient descent is easy: just take small steps in the opposite direction. This corresponds to multiplying some small constant by the gradient and subtracting that from the parameters. In math form that looks like:

$$w_{t+1} = w_t - \alpha \nabla f(w_t)$$

This small constant is called the **learning rate**, for which the symbol α is usually used.

The next cell demonstrates a single step of gradient descent. Try running the cell repeatedly, and watch as the loss reduces with each step.

[55]:

```
# perform one step of gradient descent
lr = 0.01

with torch.no_grad():
    W1_grad, B1_grad, W2_grad, B2_grad = backprop(X)
    W1 -= W1_grad * lr
    B1 -= B1_grad * lr
    W2 -= W2_grad * lr
    B2 -= B2_grad * lr

# forward pass again, notice the change in loss
y_hat = forward(X)
loss = ((y_hat - Y)**2).sum()
loss
```

[55]:

```
tensor(10.3353)
```

0.2.3 Exercise 3 (10 pts)

Could loss ever increase when you run the above cell?

Try running the above cell multiple times and notice how the loss keeps decreasing. However, is the loss guaranteed to not increase? Answer below and explain why.

Hint: What if the learning rate were really large? Would that be a meaningful difference compared to this case?

markdown

Yes, the loss could increase when running the above cell under certain conditions. While with the current learning rate of 0.01, the loss is consistently decreasing, this is not guaranteed in general.

The reason is that gradient descent works by taking steps in the direction of steepest descent, but the size of those steps is determined by the learning rate. If the learning rate is too large, we might "overshoot" the minimum and end up with higher loss.

Consider what happens with a very large learning rate: 1. The gradients tell us which direction to move to reduce the loss 2. But if we move too far in that direction (due to a large learning rate), we could end up on the other side of the minimum 3. This could result in a higher loss than before the update

This becomes particularly problematic in areas with high curvature, where the direction of steepest descent changes rapidly. A learning rate that's appropriate for one region of the loss landscape might be too large for another region.

Additionally, since our computation of gradients is based on local information, it doesn't account for how the loss function behaves far away from the current point. This means that taking large steps based on local gradients can lead us into regions with higher loss values.

0.3 Part 3: PyTorch is a deep learning framework

While that was very fun and intellectually stimulating, we can actually make PyTorch do all of that math for us instead. Pytorch is well-known for its automatic differentiation feature. We can call the `backward()` method to ask PyTorch to calculate the gradients, which are then stored in the `grad` attribute for tensors which are marked as needing gradients.

[56]:

```python
# x is just an example tensor
# `requires_grad_` tells PyTorch to store gradients for the tensor
x = torch.tensor([2., 3., 4.]).requires_grad_(True)

# `x.grad` is currently empty, because we haven't called `backward()` on anything
print(x.grad)
```

```
None
```

[57]:

```python
y = (x**2 + x).sum()
print(y)
y.backward()
print(x.grad)
```

```
tensor(38., grad_fn=<SumBackward0>)
tensor([5., 7., 9.])
```

Let's run backprop again to see what happens. Feel free to run the next cell repeatedly and watch as the stored gradients keep increasing.

[58]:

```python
# Notice that this is the same calculation, but the gradients increase!
# This shows that `.backward()` adds to the gradients without resetting them
y = (x**2 + x).sum()
print(y)
y.backward()
print(x.grad)
```

```
tensor(38., grad_fn=<SumBackward0>)
tensor([10., 14., 18.])
```

We can see that the `x.grad` is updated to be the sum of the gradients calculated so far. When we run backprop in a neural network, we sum up all the gradients for a particular neuron before making an update. This is exactly what is happening here! This is also the reason why we need to run `zero_grad()` in every training iteration (more on this later). Otherwise our gradients would keep building up from one training iteration to the other, which would cause our updates to be wrong.

0.3.1 Using Autograd on our toy neural network

Applying the Autograd feature to our prior code, we can skip doing all the math for backprop. Just by doing the forward pass, we get the backward pass completely for free :)

[59]:
```
# `requires_grad_` tells pytorch to save gradients for those tensors
W1.requires_grad_(True)
B1.requires_grad_(True)
W2.requires_grad_(True)
B2.requires_grad_(True)

# remove the gradients so running this cell repeatedly doesn't break anything
W1.grad = None
B1.grad = None
W2.grad = None
B2.grad = None

y_hat = forward(X)

# this time we use `mse_loss` built into PyTorch, as it is, in fact, a deep␣
 ↪learning framework
loss = mse_loss(y_hat, Y, reduction='sum')
loss.backward()  # this is the line which actually calculates gradients!
loss
```

[59]:
```
tensor(10.3353, grad_fn=<MseLossBackward0>)
```

[60]:
```
# W1.grad now contains the gradients of W1 without having to manually calculate␣
 ↪them!
print('PyTorch Autograd:')
print(W1.grad)

# If we *do* manually calculate it, we _should_ find them to be equal
with torch.no_grad():
    W1_grad, B1_grad, W2_grad, B2_grad = backprop(X)

print('Manually calculated gradient:')
print(W1_grad)
```

```
PyTorch Autograd:
tensor([[-0.0357,  0.1023, -0.3548,  0.0483],
        [-0.0276,  0.0040,  0.1077, -0.0460],
        [-0.0397,  0.3204,  0.0709,  0.0368]])
Manually calculated gradient:
tensor([[-0.0357,  0.1023, -0.3548,  0.0483],
        [-0.0276,  0.0040,  0.1077, -0.0460],
        [-0.0397,  0.3204,  0.0709,  0.0368]])
```

If your `backprop` code matches the provided math, you should see your calculated gradients being the same as the automatic gradients. Run the next cell and make sure the differences are close to zero.

Under the hood, PyTorch uses faster/better versions of what we implemented and tends to get *slightly* different answers. If you're not satisfied with this level of similarity, feel free to switch everything to float64, and should find that it matches to around 15 decimal places.

[61]:

```
print(f'W1 grad distance:{(W1_grad - W1.grad).norm().item():.10f}')
print(f'B1 grad distance:{(B1_grad - B1.grad).norm().item():.10f}')
print(f'W2 grad distance:{(W2_grad - W2.grad).norm().item():.10f}')
print(f'B2 grad distance:{(B2_grad - B2.grad).norm().item():.10f}')
```

```
W1 grad distance:0.0000000493
B1 grad distance:0.0000000333
W2 grad distance:0.0000000817
B2 grad distance:0.0000001229
```

0.3.2 Neural network modules

However, listing out all parameters without structure and defining the operations manually would be a mess with a more complicated architecture, especially when it comes to handling updates. PyTorch, being a deep learning framework, provides structure and utilities to help us handle every part of that intelligently.

We use the `nn.Module` base class to organize our code. This allows PyTorch to keep track of all parameters, their gradients, and apply updates once we use an optimizer. PyTorch allows `nn.Modules` within other `nn.Modules`, which allows effective code reuse and organization. Notably, many common layers are already defined as `nn.Modules` within `nn`.

Here's an example PyTorch neural network, annotated with comments explaining parts:

```
class Model(nn.Module):  # replace "Model" with what you want to call the network class
    def __init__(self):  # constructor defined as `__init__`. You can also provide args.
        super().__init__()  # call the super class's constructor for PyTorch to properly regis
        # store the model's parameters in fields
        self.conv1 = nn.Conv2d(1, 20, 5)  # nn.Conv2d is one of PyTorch's many built-in layers
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):  # the `forward` method is how you run the model, and `__call__` get
        # this is where you should implement the forward pass
        x = self.conv1(x).relu()
        return self.conv2(x).relu()
```

For more information/examples, see the following: - nn.Module Documentation - The "Creating Models" section of the PyTorch Quickstart Tutorial

[62]:

```
# Here's another example network defined using `nn.Module`, and
# using `nn.Sequential` internally to organize the strucure better

class ExampleNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.main = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10),
        )

    def forward(self, x):
        return self.main(x)

# We can instantiate an instance of this example network with the constructor
model = ExampleNetwork()
# `nn.Module` exposes a simple display function just by `print`ing the model
print(model)
# Run the model on data just by calling the model
model(torch.randn(1, 28, 28))
```

```
ExampleNetwork(
  (main): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=128, bias=True)
    (2): ReLU()
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): ReLU()
    (5): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

[62]:

```
tensor([[-0.0176,  0.1386,  0.1281, -0.2005, -0.1357,  0.0697, -0.0694,  0.0506,
          0.2302,  0.0141]], grad_fn=<AddmmBackward0>)
```

0.3.3 Exercise 4 (20 pts)

Implement the same toy neural network we hand-coded backpropagation for, but this time using PyTorch's neural network modules.

Note that we also start using `mse_loss` back as the default `reduction='mean'`.

- `nn.Sequential` – A sequential container of modules.
- `nn.Linear` – Applies a linear transformation to the incoming data: $y = xA^T + b$
- `nn.ReLU` – Applies the rectified linear unit function element-wise: $\text{ReLU}(x) = \max(0, x)$
- `nn.Sigmoid` – Applies the element-wise function: $\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1+\exp(-x)}$

[63]:
```
class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Define the structure of our toy neural network using PyTorch modules
        self.layer1 = nn.Linear(3, 4)  # Input dim 3, hidden dim 4
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(4, 2)  # Hidden dim 4, output dim 2
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Implement the forward pass using the layers defined above
        z1 = self.layer1(x)
        h = self.relu(z1)
        z2 = self.layer2(h)
        y_hat = self.sigmoid(z2)
        return y_hat

model = ToyModel()
model
```

[63]:
```
ToyModel(
  (layer1): Linear(in_features=3, out_features=4, bias=True)
  (relu): ReLU()
  (layer2): Linear(in_features=4, out_features=2, bias=True)
  (sigmoid): Sigmoid()
)
```

[64]:
```
y_hat = model(X)
loss = mse_loss(y_hat, Y)
loss
```

[64]:
```
tensor(0.2555, grad_fn=<MseLossBackward0>)
```

0.3.4 Exercise 5 (30 pts)

Implement an optimization loop using the following PyTorch components. Optimize the model to get MSE loss below 0.01 on the toy data.

Note that in a real task, we loop through the data, running the model on a small part of it each iteration. A real task should also always involve a separation between train and test data. However, this is a toy minimal example to play with the syntax and tools available in PyTorch; although this exercise is similar to a typical training loop, it's more apt to call it just optimizing to overfit some data. This network does not have the expressive power necessary to model the true distribution of the data.

- Select a suitable optimizer
  - torch.optim.SGD is what you manully implemented above, but you will likely find it necessary to either configure parameters such as momentum, or use an alternate optimizer such as torch.optim.Adam. We suggest experimenting with this to find a configuration which effectively reduces loss.
  - Figure out what parameters are required when defining it. Make sure you define the optimizer outside of the loop, as the optimizer should not be redefined each time.
- Use a loop to repeatedly run the optimization. You may have to loop for many steps for it to properly converge, depending on the settings you select for your optimizer.
  - Our solution runs it for 10,000 steps for good measure, which takes around 10 seconds on CPU. It is possible to have it converge with far fewer steps.
- optimizer.zero_grad() will zero all the gradients on parameters given to the optimizer
- Run the model on the data X to produce predicted values for Y
- Calculate the loss using mse_loss
- Call .backward() on the loss to populate the parameters with gradients
- optimizer.step() performs a single optimization step (parameter update based on gradients)
- See "taking an optimization step" in torch.optim documentation to see an example.
- You may also find the "Optimizing the Model Parameters" section of the Pytorch Quickstart Tutorial helpful, though it implements "real" training code, so we omit the data iteration.

This network just *barely* has the expressive ability necessary to fit this amount of data, and you will have to play with the optimizer configuration a fair bit before you get a low enough loss.

[65]:

```
model = ToyModel()

# Choose Adam optimizer which typically converges faster than SGD for this type␣
 ↪of problem
optimizer = torch.optim.Adam(model.parameters(), lr=0.02)

# Run optimization for many steps
for step in range(10000):
    # Zero gradients at the start of each iteration
    optimizer.zero_grad()

    # Forward pass: compute predictions
    y_hat = model(X)

    # Compute loss
    loss = mse_loss(y_hat, Y)

    # Backward pass: compute gradients
    loss.backward()

    # Update weights
    optimizer.step()

    # Print progress every 1000 steps
    if step % 1000 == 0:
        print(f"Step {step}, Loss: {loss.item():.4f}")

print(f"Final loss: {loss.item():.6f}")
```

```
Step 0, Loss: 0.2571
Step 1000, Loss: 0.0257
Step 2000, Loss: 0.0252
Step 3000, Loss: 0.0251
Step 4000, Loss: 0.0250
Step 5000, Loss: 0.0250
Step 6000, Loss: 0.0250
Step 7000, Loss: 0.0250
Step 8000, Loss: 0.0250
Step 9000, Loss: 0.0250
Final loss: 0.025001
```

```
[66]: # Exercise 5: Don't modify this cell, but make sure to run it.
      y_hat = model(X)
      loss = mse_loss(y_hat, Y)
      loss
```

```
[66]: tensor(0.0250, grad_fn=<MseLossBackward0>)
```

The following is not graded.

Notice that if we try different data from the same distribution, it does terribly. Why is this? What should we do differently to have it generalize better to the true distribution of the data? Feel free to add another cell below this with a modified optimization loop which trains on the distribution by calling `generate_data` each iteration instead of just overfitting on a single batch.

```
[67]: X2, Y2 = generate_data(100)
      y_hat = model(X2)
      loss = mse_loss(y_hat, Y2)
      loss
```

```
[67]: tensor(0.2728, grad_fn=<MseLossBackward0>)
```