


 [jcjohnson](#) / [pytorch-examples](#) Public

Simple examples to introduce PyTorch

 MIT License 4.2k stars  912 forks Star Watch ▼[Code](#)[Issues](#) 6[Pull requests](#) 2[Actions](#)[Projects](#)[Wiki](#)[Security](#)[Insights](#) master ▼

...



jcjohnson ...

on Jul 2, 2019

[View code](#)

This repository introduces the fundamental concepts of [PyTorch](#) through self-contained examples.

At its core, PyTorch provides two main features:

- An n-dimensional Tensor, similar to numpy but can run on GPUs
- Automatic differentiation for building and training neural networks

We will use a fully-connected ReLU network as our running example. The network will have a single hidden layer, and will be trained with gradient descent to fit random data by minimizing the Euclidean distance between the network output and the true output.

NOTE: These examples have been update for PyTorch 0.4, which made several major changes to the core PyTorch API. Most notably, prior to 0.4 Tensors had to be wrapped in Variable objects to use autograd; this functionality has now been added directly to Tensors, and Variables are now deprecated.

Table of Contents

- [Warm-up: numpy](#)
- [PyTorch: Tensors](#)
- [PyTorch: Autograd](#)
- [PyTorch: Defining new autograd functions](#)

- [TensorFlow: Static Graphs](#)
- [PyTorch: nn](#)
- [PyTorch: optim](#)
- [PyTorch: Custom nn Modules](#)
- [PyTorch: Control Flow and Weight Sharing](#)

Warm-up: numpy

Before introducing PyTorch, we will first implement the network using numpy.

Numpy provides an n-dimensional array object, and many functions for manipulating these arrays. Numpy is a generic framework for scientific computing; it does not know anything about computation graphs, or deep learning, or gradients. However we can easily use numpy to fit a two-layer network to random data by manually implementing the forward and backward passes through the network using numpy operations:

```
# Code in file tensor/two_layer_net_numpy.py
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
```

```
grad_w1 = x.T.dot(grad_h)

# Update weights
w1 -= learning_rate * grad_w1
w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Numpy is a great framework, but it cannot utilize GPUs to accelerate its numerical computations. For modern deep neural networks, GPUs often provide speedups of [50x or greater](#), so unfortunately numpy won't be enough for modern deep learning.

Here we introduce the most fundamental PyTorch concept: the **Tensor**. A PyTorch Tensor is conceptually identical to a numpy array: a Tensor is an n-dimensional array, and PyTorch provides many functions for operating on these Tensors. Any computation you might want to perform with numpy can also be accomplished with PyTorch Tensors; you should think of them as a generic tool for scientific computing.

However unlike numpy, PyTorch Tensors can utilize GPUs to accelerate their numeric computations. To run a PyTorch Tensor on GPU, you use the `device` argument when constructing a Tensor to place the Tensor on a GPU.

Here we use PyTorch Tensors to fit a two-layer network to random data. Like the numpy example above we manually implement the forward and backward passes through the network, using operations on PyTorch Tensors:

```
# Code in file tensor/two_layer_net_tensor.py
import torch

device = torch.device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# Randomly initialize weights
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
```

```

h_relu = h.clamp(min=0)
y_pred = h_relu.mm(w2)

# Compute and print loss; loss is a scalar, and is stored in a PyTorch Tensor
# of shape (); we can get its value as a Python number with loss.item().
loss = (y_pred - y).pow(2).sum()
print(t, loss.item())

# Backprop to compute gradients of w1 and w2 with respect to loss
grad_y_pred = 2.0 * (y_pred - y)
grad_w2 = h_relu.t().mm(grad_y_pred)
grad_h_relu = grad_y_pred.mm(w2.t())
grad_h = grad_h_relu.clone()
grad_h[h < 0] = 0
grad_w1 = x.t().mm(grad_h)

# Update weights using gradient descent
w1 -= learning_rate * grad_w1
w2 -= learning_rate * grad_w2

```

PyTorch: Autograd

In the above examples, we had to manually implement both the forward and backward passes of our neural network. Manually implementing the backward pass is not a big deal for a small two-layer network, but can quickly get very hairy for large complex networks.

Thankfully, we can use [automatic differentiation](#) to automate the computation of backward passes in neural networks. The **autograd** package in PyTorch provides exactly this functionality. When using autograd, the forward pass of your network will define a **computational graph**; nodes in the graph will be Tensors, and edges will be functions that produce output Tensors from input Tensors. Backpropagating through this graph then allows you to easily compute gradients.

This sounds complicated, it's pretty simple to use in practice. If we want to compute gradients with respect to some Tensor, then we set `requires_grad=True` when constructing that Tensor. Any PyTorch operations on that Tensor will cause a computational graph to be constructed, allowing us to later perform backpropagation through the graph. If `x` is a Tensor with `requires_grad=True`, then after backpropagation `x.grad` will be another Tensor holding the gradient of `x` with respect to some scalar value.

Sometimes you may wish to prevent PyTorch from building computational graphs when performing certain operations on Tensors with `requires_grad=True`; for example we usually don't want to backpropagate through the weight update steps when training a neural network. In such scenarios we can use the `torch.no_grad()` context manager to prevent the construction of a computational graph.

Here we use PyTorch Tensors and autograd to implement our two-layer network; now we no longer need to manually implement the backward pass through the network:

```
# Code in file autograd/two_layer_net_autograd.py
import torch

device = torch.device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# Create random Tensors for weights; setting requires_grad=True means that we
# want to compute gradients for these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y using operations on Tensors. Since w1 and
    # w2 have requires_grad=True, operations involving these Tensors will cause
    # PyTorch to build a computational graph, allowing automatic computation of
    # gradients. Since we are no longer implementing the backward pass by hand we
    # don't need to keep references to intermediate values.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # Compute and print loss. Loss is a Tensor of shape (), and loss.item()
    # is a Python number giving its value.
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call w1.grad and w2.grad will be Tensors holding the gradient
    # of the loss with respect to w1 and w2 respectively.
    loss.backward()

    # Update weights using gradient descent. For this step we just want to mutate
    # the values of w1 and w2 in-place; we don't want to build up a computational
    # graph for the update steps, so we use the torch.no_grad() context manager
    # to prevent PyTorch from building a computational graph for the updates
    with torch.no_grad():
```

⋮ README.md

```
    # Manually zero the gradients after running the backward pass
```

```
w1.grad.zero_()
w2.grad.zero_()
```

PyTorch: Defining new autograd functions

Under the hood, each primitive autograd operator is really two functions that operate on Tensors. The **forward** function computes output Tensors from input Tensors. The **backward** function receives the gradient of the output Tensors with respect to some scalar value, and computes the gradient of the input Tensors with respect to that same scalar value.

In PyTorch we can easily define our own autograd operator by defining a subclass of `torch.autograd.Function` and implementing the `forward` and `backward` functions. We can then use our new autograd operator by constructing an instance and calling it like a function, passing Tensors containing input data.

In this example we define our own custom autograd function for performing the ReLU nonlinearity, and use it to implement our two-layer network:

```
# Code in file autograd/two_layer_net_custom_function.py
import torch

class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """
    @staticmethod
    def forward(ctx, x):
        """
        In the forward pass we receive a context object and a Tensor containing the
        input; we must return a Tensor containing the output, and we can use the
        context object to cache objects for use in the backward pass.
        """
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive the context object and a Tensor containing
        the gradient of the loss with respect to the output produced during the
        forward pass. We can retrieve cached data from the context object, and must
        compute and return the gradient of the loss with respect to the input to the
        forward function.
        """
        x, = ctx.saved_tensors
        grad_x = grad_output.clone()
```

```

grad_x[x < 0] = 0
return grad_x

device = torch.device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and output
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# Create random Tensors for weights.
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y using operations on Tensors; we call our
    # custom ReLU implementation using the MyReLU.apply function
    y_pred = MyReLU.apply(x.mm(w1)).mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())

    # Use autograd to compute the backward pass.
    loss.backward()

    with torch.no_grad():
        # Update weights using gradient descent
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

    # Manually zero the gradients after running the backward pass
    w1.grad.zero_()
    w2.grad.zero_()

```

TensorFlow: Static Graphs

PyTorch autograd looks a lot like TensorFlow: in both frameworks we define a computational graph, and use automatic differentiation to compute gradients. The biggest difference between the two is that TensorFlow's computational graphs are **static** and PyTorch uses **dynamic** computational graphs.

In TensorFlow, we define the computational graph once and then execute the same graph over and over again, possibly feeding different input data to the graph. In PyTorch, each forward pass defines a new computational graph.

Static graphs are nice because you can optimize the graph up front; for example a framework might decide to fuse some graph operations for efficiency, or to come up with a strategy for distributing the graph across many GPUs or many machines. If you are reusing the same graph over and over, then this potentially costly up-front optimization can be amortized as the same graph is rerun over and over.

One aspect where static and dynamic graphs differ is control flow. For some models we may wish to perform different computation for each data point; for example a recurrent network might be unrolled for different numbers of time steps for each data point; this unrolling can be implemented as a loop. With a static graph the loop construct needs to be a part of the graph; for this reason TensorFlow provides operators such as `tf.scan` for embedding loops into the graph. With dynamic graphs the situation is simpler: since we build graphs on-the-fly for each example, we can use normal imperative flow control to perform computation that differs for each input.

To contrast with the PyTorch autograd example above, here we use TensorFlow to fit a simple two-layer net:

```
# Code in file autograd/tf_two_layer_net.py
import tensorflow as tf
import numpy as np

# First we set up the computational graph:

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create placeholders for the input and target data; these will be filled
# with real data when we execute the graph.
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

# Create Variables for the weights and initialize them with random data.
# A TensorFlow Variable persists its value across executions of the graph.
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

# Forward pass: Compute the predicted y using operations on TensorFlow Tensors.
# Note that this code does not actually perform any numeric operations; it
# merely sets up the computational graph that we will later execute.
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)
```



```

# Compute loss using operations on TensorFlow Tensors
loss = tf.reduce_sum((y - y_pred) ** 2.0)

# Compute gradient of the loss with respect to w1 and w2.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

# Update the weights using gradient descent. To actually update the weights
# we need to evaluate new_w1 and new_w2 when executing the graph. Note that
# in TensorFlow the the act of updating the value of the weights is part of
# the computational graph; in PyTorch this happens outside the computational
# graph.
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

# Now we have built our computational graph, so we enter a TensorFlow session to
# actually execute the graph.
with tf.Session() as sess:
    # Run the graph once to initialize the Variables w1 and w2.
    sess.run(tf.global_variables_initializer())

    # Create numpy arrays holding the actual data for the inputs x and targets y
    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        # Execute the graph many times. Each time it executes we want to bind
        # x_value to x and y_value to y, specified with the feed_dict argument.
        # Each time we execute the graph we want to compute the values for loss,
        # new_w1, and new_w2; the values of these Tensors are returned as numpy
        # arrays.
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})

        print(loss_value)

```

PyTorch: nn

Computational graphs and autograd are a very powerful paradigm for defining complex operators and automatically taking derivatives; however for large neural networks raw autograd can be a bit too low-level.

When building neural networks we frequently think of arranging the computation into **layers**, some of which have **learnable parameters** which will be optimized during learning.

In TensorFlow, packages like [Keras](#), [TensorFlow-Slim](#), and [TFLearn](#) provide higher-level abstractions over raw computational graphs that are useful for building neural networks.

In PyTorch, the `nn` package serves this same purpose. The `nn` package defines a set of **Modules**, which are roughly equivalent to neural network layers. A Module receives input Tensors and computes output Tensors, but may also hold internal state such as Tensors containing learnable parameters. The `nn` package also defines a set of useful loss functions that are commonly used when training neural networks.

In this example we use the `nn` package to implement our two-layer network:

```
# Code in file nn/two_layer_net_nn.py
import torch

device = torch.device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
# After constructing the model we use the .to() method to move it to the
# desired device.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
).to(device)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function. Setting
# reduction='sum' means that we are computing the *sum* of squared errors rather
# than the mean; this is for consistency with the examples above where we
# manually compute the loss, but in practice it is more common to use mean
# squared error as a loss by setting reduction='elementwise_mean'.
loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-4
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(x)

    # Compute and print loss. We pass Tensors containing the predicted and true
```

```

# values of y, and the loss function returns a Tensor containing the loss.
loss = loss_fn(y_pred, y)
print(t, loss.item())

# Zero the gradients before running the backward pass.
model.zero_grad()

# Backward pass: compute gradient of the loss with respect to all the learnable
# parameters of the model. Internally, the parameters of each Module are stored
# in Tensors with requires_grad=True, so this call will compute gradients for
# all learnable parameters in the model.
loss.backward()

# Update the weights using gradient descent. Each parameter is a Tensor, so
# we can access its data and gradients like we did before.
with torch.no_grad():
    for param in model.parameters():
        param.data -= learning_rate * param.grad

```

PyTorch: optim

Up to this point we have updated the weights of our models by manually mutating Tensors holding learnable parameters. This is not a huge burden for simple optimization algorithms like stochastic gradient descent, but in practice we often train neural networks using more sophisticated optimizers like AdaGrad, RMSProp, Adam, etc.

The `optim` package in PyTorch abstracts the idea of an optimization algorithm and provides implementations of commonly used optimization algorithms.

In this example we will use the `nn` package to define our model as before, but we will optimize the model using the Adam algorithm provided by the `optim` package:

```

# Code in file nn/two_layer_net_optim.py
import torch

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

```

```

loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use Adam; the optim package contains many other
# optimization algorithms. The first argument to the Adam constructor tells the
# optimizer which Tensors it should update.
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the Tensors it will update (which are the learnable weights
    # of the model)
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its parameters
    optimizer.step()

```

PyTorch: Custom nn Modules

Sometimes you will want to specify models that are more complex than a sequence of existing Modules; for these cases you can define your own Modules by subclassing `nn.Module` and defining a `forward` which receives input Tensors and produces output Tensors using other modules or other autograd operations on Tensors.

In this example we implement our two-layer network as a custom Module subclass:

```

# Code in file nn/two_layer_net_module.py
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

```

```

def forward(self, x):
    """
    In the forward function we accept a Tensor of input data and we must return
    a Tensor of output data. We can use Modules defined in the constructor as
    well as arbitrary (differentiable) operations on Tensors.
    """
    h_relu = self.linear1(x).clamp(min=0)
    y_pred = self.linear2(h_relu)
    return y_pred

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Construct our model by instantiating the class defined above.
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
loss_fn = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

PyTorch: Control Flow + Weight Sharing

As an example of dynamic graphs and weight sharing, we implement a very strange model: a fully-connected ReLU network that on each forward pass chooses a random number between 1 and 4 and uses that many hidden layers, reusing the same weights multiple times to compute the innermost hidden layers.

For this model can use normal Python flow control to implement the loop, and we can implement weight sharing among the innermost layers by simply reusing the same Module multiple times when defining the forward pass.

We can easily implement this model as a Module subclass:

```
# Code in file nn/dynamic_net.py
import random
import torch

class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we construct three nn.Linear instances that we will use
        in the forward pass.
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 0, 1, 2, or 3
        and reuse the middle_linear Module that many times to compute hidden layer
        representations.

        Since each forward pass builds a dynamic computation graph, we can use normal
        Python control-flow operators like loops or conditional statements when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same Module many
        times when defining a computational graph. This is a big improvement from Lua
        Torch, where each Module could be used only once.
        """
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. Training this strange model with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(reduction='sum')
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Releases

No releases published

Packages

No packages published

Contributors 5



Languages

● Python 100.0%