



0.4.0

## BEGINNER TUTORIALS

[Deep Learning with PyTorch: A 60 Minute Blitz](#)[PyTorch for former Torch users](#)

## Learning PyTorch with Examples

[Tensors](#)[Autograd](#)[nn module](#)

## Examples

[Tensors](#)[Autograd](#)[nn module](#)[PyTorch: nn](#)[PyTorch: optim](#)[PyTorch: Custom nn Modules](#)[PyTorch: Control Flow + Weight Sharing](#)

## Transfer Learning tutorial

[Data Loading and Processing Tutorial](#)[Deep Learning for NLP with Pytorch](#)

## INTERMEDIATE TUTORIALS

[Classifying Names with a Character-Level RNN](#)[Generating Names with a Character-Level RNN](#)[Translation with a Sequence to Sequence Network and Attention](#)

## PyTorch: optim

A fully-connected ReLU network with one hidden layer, trained to predict  $y$  from  $x$  by minimizing squared Euclidean distance.

This implementation uses the `nn` package from PyTorch to build the network.

Rather than manually updating the weights of the model as we have been doing, we use the `optim` package to define an `Optimizer` that will update the weights for us. The `optim` package defines many optimization algorithms that are commonly used for deep learning, including SGD+momentum, RMSProp, Adam, etc.

```
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(size_average=False)

# 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 variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    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()
```

Total running time of the script: ( 0 minutes 0.000 seconds)

[Download Python source  
code: two\\_layer\\_net\\_optim.py](#)[Download Jupyter notebook:  
two\\_layer\\_net\\_optim.ipynb](#)

Gallery generated by Sphinx-Gallery

[Previous](#)[Next](#)