import torch
from torch import nn
ce = nn.CrossEntropyLoss(reduction='sum')
x_i = torch.tensor([1.0, 2.0, 3.0])
w = torch.tensor([[4.0, 5.0, 6.0], [7.0, 8.0, 9.0]], requires_grad=True)
y_i = torch.tensor(1)
loss = ce(w @ x_i, y_i)
print(loss)

For the code above, \( w \) would predict 1 given \( x_i \), regardless of what \( y_i \) is. The variable \( y_i \) is the label we assign to \( x_i \). If the prediction of \( w \) agrees with \( y_i \) (when they both are 1), then the loss should be low, so it’s no surprise that the loss is close to 0.

Because the dot product for class 0 is 32 and the dot product for class 1 is 50, the gap is so large that \( \exp(32 - 50) \) is a tiny number. As a consequence, \( 1/(1 + \exp(32 - 50)) \) is very close to 1, and \( -\log(1) \) is 0.

import torch
from torch import nn
w = nn.Linear(3, 2)
print(list(w.parameters()))

The above code gives different values every time you run it, because PyTorch initializes a new set of parameters every time. Initialization plays a big role in training deep networks. Sometimes different layer types even have different initialization strategies. When you run the above code, you will see that \( \text{nn.Linear} \) includes a weight matrix and a bias vector. Technically, it really should be an affine layer rather than linear.

import torch
from torch import nn
from torch import optim
ce = nn.CrossEntropyLoss(reduction='sum')
w = nn.Linear(3, 2)
opt = optim.SGD(w.parameters(), lr=0.1)

for i in range(20):
    opt.zero_grad()

    # fake data
x_i = torch.rand(3)
y_i = torch.tensor(i % 2)

loss = ce(w(x_i), y_i)
print(loss)
loss.backward()

opt.step()

The step size is 0.1 as specified in lr. In PyTorch and the deep learning community, the step size is often called the learning rate.

The line `zero_grad()` is critical as it clears any gradient on the graph. If you forget to `zero_grad()`, the gradients accumulate and you'd be optimizing towards the wrong direction.