

پیش گزارش آزمایش دوم (پیاده سازی پرسپترون)

امیرحسین احمدی آشتیانی ۱۹۹۲۳۵۰۱

محمد رضا امیری ۹۹۲۶۰۴۰

محمد مهدی نوروزی ۹۹۲۳۰۸۵

(1) با توجه به رابطه و فرمول پرسپترون، این واحد میتواند یکتابع خطی روی ورودی فیت کند یا دو کلاس را به صورت خطی از هم جدا کند. اما با اعمال یکتابع غیرخطی برخروجی پرسپترون و پشت هم قرار دادن پرسپترون میتوان رفتار غیرخطی را نیز یاد گرفت.

(2) گرادیان چگونه در پایتورچ محاسبه میشود (عملیات پس انتشار خطای؟)

پاسخ این است که از graph comutational graph برای هندل کردن گرادیان و آپدیت وزن ها استفاده میشود. اینکه چگونه باید از graph برای محاسبه گرادیان نسبت به یک پارامتر استفاده کرد، در آزمایش به صورت مفصل توضیح داده شده است. همچین آزمایش هم با همین روش انجام شده است. لذا در این قسمت بیشتر به پایتورچ می پردازیم.

سه مفهوم مهم در پایتورچ وجود دارد: Autograd، tensor و Module. یک تنسور همانند آرایه numpy است با این تفاوت که روی GPU هم اجرا میشود. Autograd هم برای ساختن گراف محاسبات استفاده میشود. هم یک لایه شبکه عصبی است. در این قسمت به Autograd خواهیم پرداخت. در ادامه مثال هایی را بررسی خواهیم کرد.

اگر بخواهیم به صورت معمولی گرادیان را محاسبه کنیم:

PyTorch: Tensors

Backward pass: manually compute gradients

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
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):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

اگر بخواهیم از Autograd استفاده کنیم. در تصاویر زیر توضیحاتی بیان شده است:

PyTorch: Autograd

We will not want gradients (of loss) with respect to data

Do want gradients with respect to weights

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Forward pass looks exactly the same as before, but we don't need to track intermediate values - PyTorch keeps track of them for us in the graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Computes gradients with respect to all inputs that have `requires_grad=True`!

```
import torch

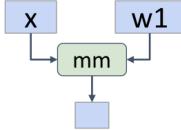
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



Every operation on a tensor with `requires_grad=True` will add to the computational graph, and the resulting tensors will also have `requires_grad=True`

```
import torch

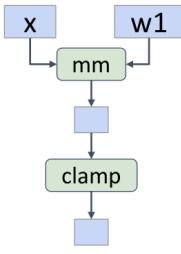
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



Every operation on a tensor with `requires_grad=True` will add to the computational graph, and the resulting tensors will also have `requires_grad=True`

```
import torch

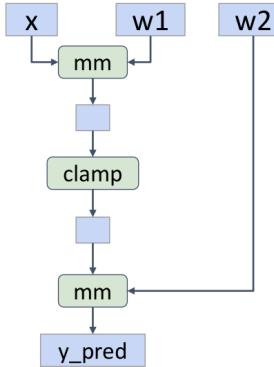
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



```
import torch

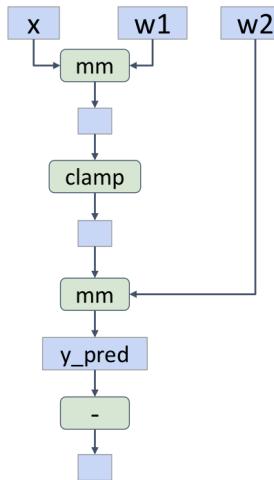
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



```
import torch

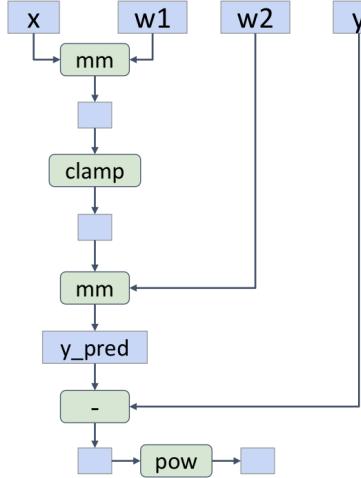
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



```
import torch

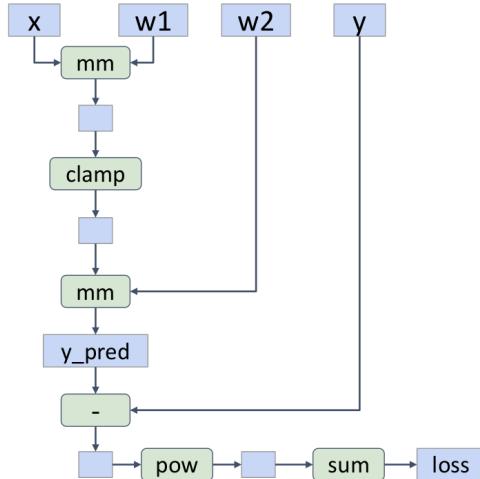
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



```
import torch

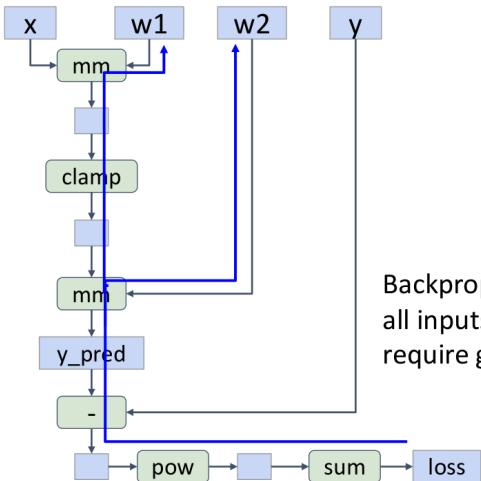
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd



```

import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()

```

PyTorch: Autograd



```

import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()

```

PyTorch: Autograd

x w1 w2 y

After backward finishes, gradients are **accumulated** into w1.grad and w2.grad and the graph is destroyed

Set gradients to zero – forgetting this is a common bug!

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

x w1 w2 y

After backward finishes, gradients are **accumulated** into w1.grad and w2.grad and the graph is destroyed

Tell PyTorch not to build a graph for these operations

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

همانطور که در تصاویر بالا توضیح داده شد، گراف محاسبات به صورت خودکار محاسبه شده و برای آپدیت وزن ها مورد استفاده قرار میگیرد.
گرادیان چگونه در تنسورفلو انجام میشود؟

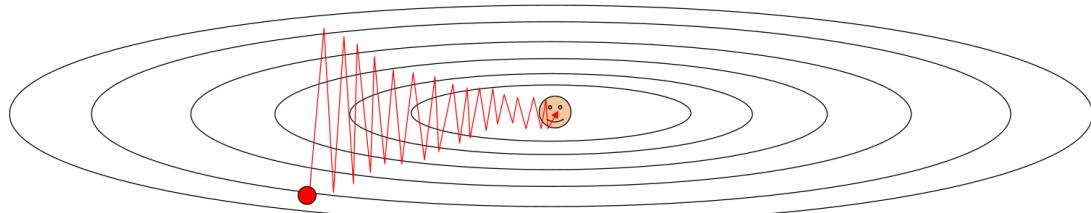
همانند چیزی که در پایتون ج انجام میشود، در تنسورفلو نیز وجود دارد. به عبارت دیگر، تنسورفلو نیز از

استفاده میکند و آنرا به صورت داینامیک آپدیت کرده و گرادیان هارا

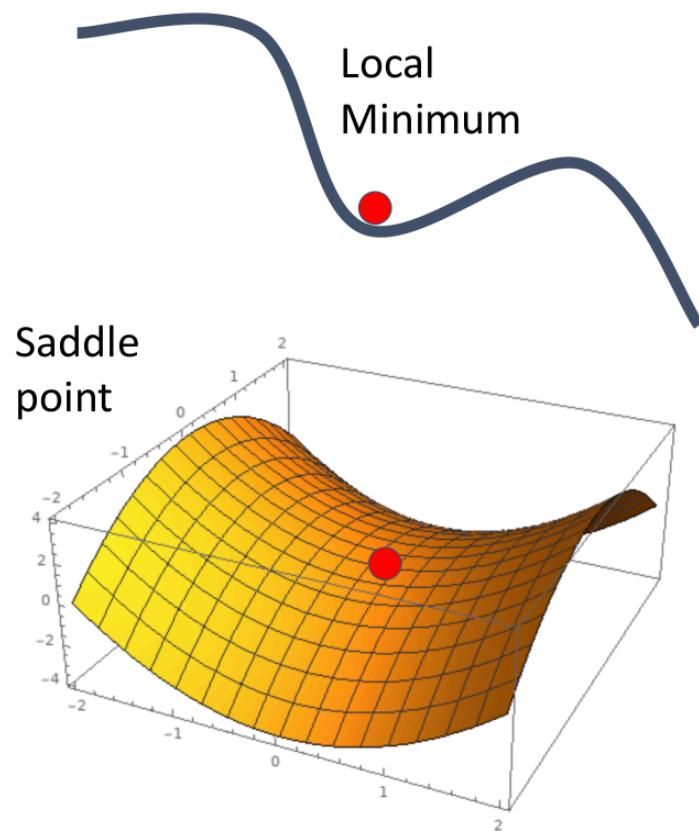
محاسبه میکند. این عملیات توسط tf.GradientTape انجام میشود.

(4) نقض آپتیمایزر هار در شبکه عصبی چیست؟

ایده استفاده از آپتیمایزر ها زمانی مطرح شد که هنگام استفاده از SGD با مشکلات زیر روبرو شدند:
 اول اینکه: اگر loss در یک جهت بسیار سریع تغییر کند ولی در جهت دیگر آرام چه میشود؟ گردایان کاهشی چطور خواهد شد؟ وزن ها در یک جهت بسیار سریع حرکت آپدیت میشود ولی در جهت دیگر بسیار کند خواهد بود. تصویر زیر گویای این مشکل است:



در این صورت خیلی دیر به نقطه آپتیمم میرسیم.
 دوم اینکه: اگر تابعی که قصد بهینه سازی آنرا داریم، بهینه محلی داشته باشد چه؟ گردایان صفر خواهد شد و وزن ها دیگر آپدیت نخواهند شد.



سوم اینکه: در SGD گرادیان ما از دسته کوچکی از داده ها انتخاب شده و وزن هارا آپدیت میکند.
لذا ممکن است نویزی باشد.

اینجا بود که راه حلی را برای آپدیت کردن وزن ها پیشنهاد کردند. برای مثال در + sgd + momentum از یک شهود و تجربه قبلی برای آپدیت کردن وزن ها استفاده میکند که باعث میشود مشکل دوم حل شود.

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

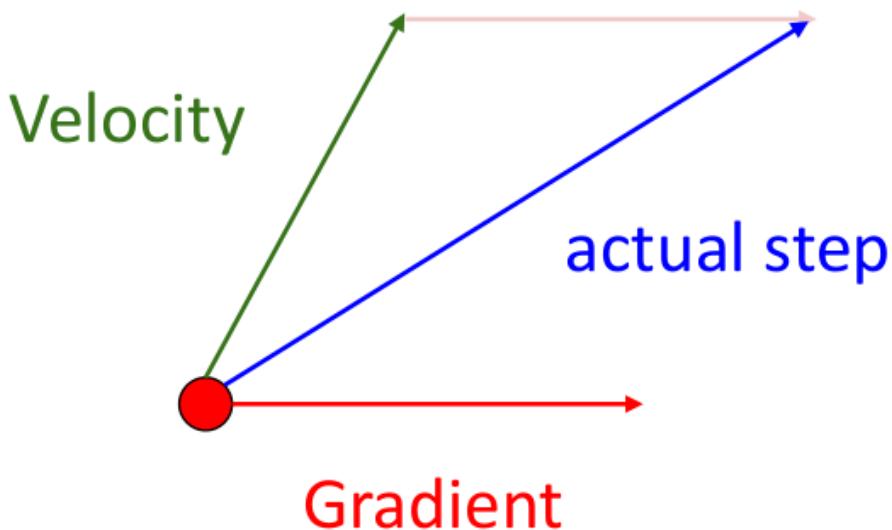
SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

Momentum update:



همانطور که در شکل بالا مشهود است، فقط از گرادیان برای آپدیت کردن استفاده نمیشود، بلکه از سرعت هم استفاده میشود که برایند آنها مسیر بهتری را پیشنهاد میدهد. اما رایج ترین آپتیمالایزر Adam است.

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```