به نام خدا

درس یادگیری عمیق

تمرین سوم

غزل زمانی نژاد

401722244

1. الف) به طور کلی بهینه سازها به دو دسته تقسیم می شوند:

Accelerated stochastic gradient descent,

Adaptive learning rate methods

دسته اول (متدهای SGD) از نرخ آموزش ثابت استفاده می کنند در حالیکه دسته دوم برای هر یک از پارامترها یک نرخ آموزش جداگانه دارد. الگوریتم های adaptive معمولا در فاز آموزش سریعتر همگرا می شوند ولی تعمیم دهی ضعیفی دارند.

در این مقاله سعی شده به ۳ هدف همزمان دست یابند که می توان به عنوان مزایای آن اشاره کرد: همگرایی سریع مثل الگوریتم های adaptive، تعمیم دهی مناسب مثل متدهای SGD و ثبات در آموزش در مدل های پیشرفته همچون GANها. برای این کار سعی شده stepsize بر اساس belief در مورد جهت فعلی گرادیان تنظیم شود. در صورتی که مشاهده گرادیان تا حد زیادی از پیش بینی منحرف شود، نسبت به مشاهدات فعلی بی اعتماد هستیم و یک قدم کوچک برمی داریم ولی اگر گرادیان مشاهده شده نزدیک به پیش بینی باشد، به آن اعتماد می کنیم و قدم بزرگی برمی داریم.

ب) Adabelief در مقایسه با Adam پارامتر اضافی ندارد. در زیر pseudo-code هر دو الگوریتم آورده شده است و تفاوت پیاده سازی بین این دو با رنگ آبی مشخص شده است. علامت گذاری ها به شرح زیر است:

گرادیان مشاهده شده در گام t را با g_t

میانگین متحرک نمایی را با m_t

میانگین متحرک نمایی g_t^2 را با v_t

$(g_t - m_t)^2$ را با s_t

نمایش می دهیم.

در الگوریتم آدام، m_t تقسیم بر $\sqrt{v_t}$ می شود در حالیکه در Adabelief، تقسیم بر $\sqrt{s_t}$ می شود. به طور شهودی، $1/\sqrt{st}$ را به عنوان belief در مشاهده در نظر می گیریم. مشاهده m_t به عنوان

پیش‌بینی گرادیان، اگر g_t بسیار از m_t منحرف شود، باور ضعیفی در g_t داریم و گام کوچکی برمی داریم.

ولی اگر g_t به m_t نزدیک باشد، باور قوی ای در g_t داریم و گام بزرگی برمی داریم.

Notations By the convention in [8], we use the following notations:

- $f(\theta) \in \mathbb{R}, \theta \in \mathbb{R}^d$: f is the loss function to minimize, θ is the parameter in $\mathbb{R}^d$
- $\prod_{\mathcal{F},M}(y) = \text{argmin}_{x \in \mathcal{F}} ||M^{1/2}(x-y)||$: projection of y onto a convex feasible set $\mathcal{F}$
- g_t: the gradient and step t
- m_t: exponential moving average (EMA) of g_t
- v_t, s_t: v_t is the EMA of g_t^2, s_t is the EMA of $(g_t - m_t)^2$
- α, ϵ: α is the learning rate, default is 10^{-3}; ϵ is a small number, typically set as 10^{-8}
- β_1, β_2: smoothing parameters, typical values are $\beta_1 = 0.9, \beta_2 = 0.999$
- β_{1t}, β_{2t} are the momentum for m_t and v_t respectively at step t, and typically set as constant (e.g. $\beta_{1t} = \beta_1, \beta_{2t} = \beta_2, \forall t \in \{1, 2, ...T\}$

Algorithm 1: Adam Optimizer

Initialize θ_0, $m_0 \leftarrow 0$, $v_0 \leftarrow 0$, $t \leftarrow 0$
While θ_t not converged
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
Bias Correction
$\widehat{m_t} \leftarrow \frac{m_t}{1-\beta_1^t}$, $\widehat{v_t} \leftarrow \frac{v_t}{1-\beta_2^t}$
Update
$\theta_t \leftarrow \prod_{\mathcal{F}, \sqrt{\widehat{v_t}}} \left(\theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \right)$

Algorithm 2: AdaBelief Optimizer

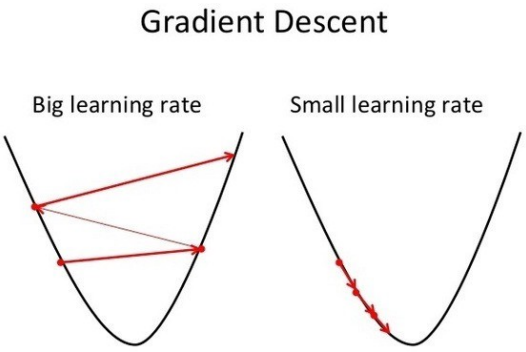
Initialize θ_0, $m_0 \leftarrow 0$, $s_0 \leftarrow 0$, $t \leftarrow 0$
While θ_t not converged
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
$s_t \leftarrow \beta_2 s_{t-1} + (1-\beta_2)(g_t - m_t)^2 + \epsilon$
Bias Correction
$\widehat{m_t} \leftarrow \frac{m_t}{1-\beta_1^t}$, $\widehat{s_t} \leftarrow \frac{s_t}{1-\beta_2^t}$
Update
$\theta_t \leftarrow \prod_{\mathcal{F}, \sqrt{\widehat{s_t}}} \left(\theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{s_t}} + \epsilon} \right)$

2. الف) اگر مقدار نرخ آموزش بسیار بزرگ باشد، باعث می شود حین آموزش از نقطه minima بپرد و loss به کندی کاهش یابد. باید در نرخ آموزش دقت باشیم که مقدار مناسبی را انتخاب کنیم. اگر مقدار آن کوچک باشد، loss به کندی کاهش می یابد. اگر مقدار آن بزرگ باشد، می تواند باعث شود که مدل خیلی سریع به یک نقطه غیربهینه همگرا شود. در صورتی که مقدار آن مناسب باشد می تواند به نقطه بهینه همگرا شود.

ب) باید نرخ آموزش را کوچکتر از مقدار فعلی انتخاب کنیم. همچنین می توانیم از learning rate scheduler استفاده کنیم تا در طول آموزش نرخ یادگیری را کنترل کنیم. روش دیگر، استفاده از الگوریتم

های بهینه سازی adaptive همچون Adam است زیرا در این گونه الگوریتم ها هر یک از پارامترها نرخ یادگیری جدا دارند.

ت) اگر از نرخ یادگیری کوچک و ثابت استفاده کنیم، زمان زیادی برای همگرایی نیاز دارد. اگر از نرخ یادگیری بزرگ ثابت استفاده کنیم، حول نقطه مینیمم پرش خواهیم داشت. به همین دلیل است استفاده از روش های learning rate decay ایده خوبی است و مقدار نرخ یادگیری را به خوبی کنترل می کند.



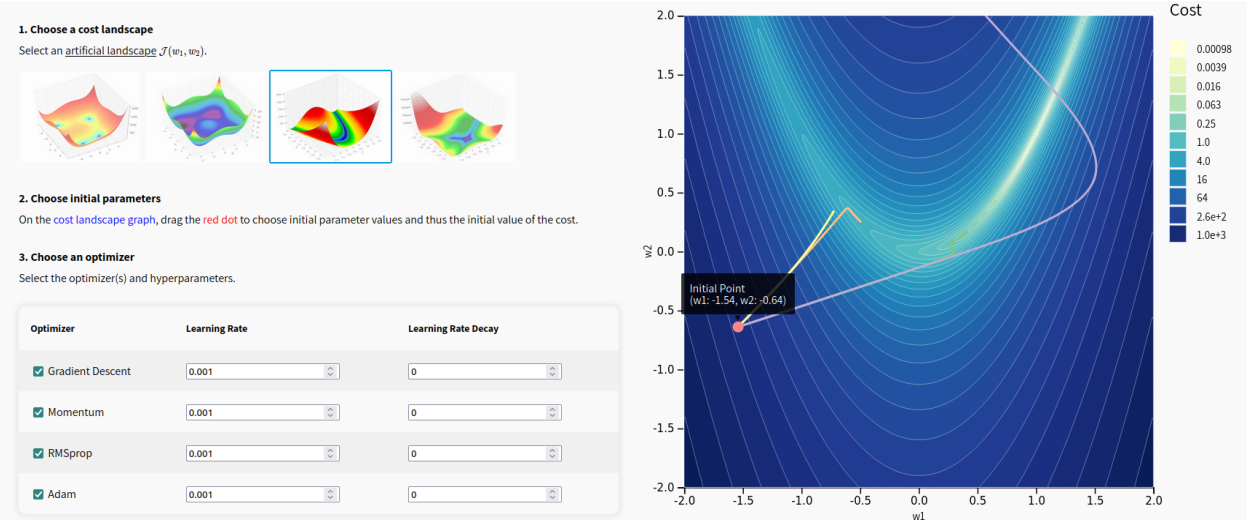
ث) روش اول: می توانیم از روش های learning rate decay برای تنظیم میزان نرخ یادگیری در حین آموزش استفاده کنیم. در ابتدای آموزش نرخ یادگیری بزرگتر است تا ناحیه بیشتری را بررسی کند ولی با استفاده از الگوریتم های مختلف به تدریج این میزان کاهش می یابد.

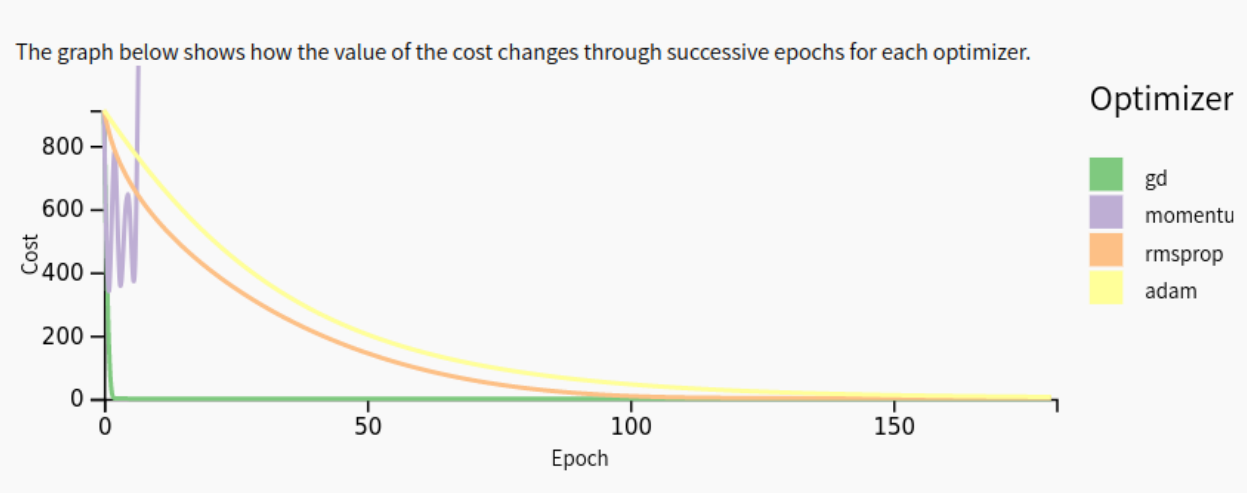
روش دوم: می توانیم از SGD + momentum استفاده کنیم. یعنی یک پارامتر سرعت محاسبه می کنیم که به سرعت قبل هم بستگی دارد. در این روش در نقطه زینی متوقف نمی شود چون از قبل سرعت دارد و اگر نقطه مینیمم محلی خیلی عمیق نباشد از آن هم عبور می کند.

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W, X, y)$$
$$W = W - V_t$$

از روش Nesterov هم می توانیم استفاده کنیم که گرادیان در نقطه بعدی را حساب می کند و جلوی overshootهای بزرگ را می گیرد.

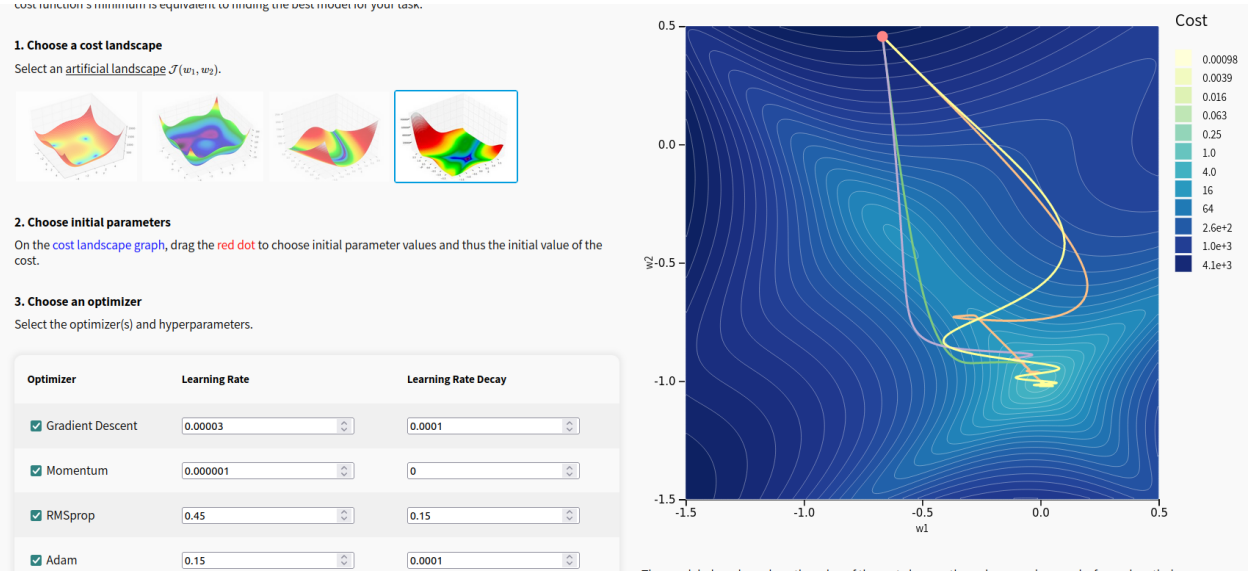
3. الف) مطابق نمودارها، gd سریعتر از سایر بهینه سازها توانسته همگرا شود زیرا در gradient descent، در هر ایپاک تمامی داده ها را مشاهده کرده و مطابق همه داده ها آپدیت کردن پارامترها انجام می شود. در این مثال، Momentum عملکرد بدی داشته و دچار overshoot شده است و بدلیل زیاد بودن پارامتر v، از نقطه مینیمم گذر کرده است. دو نمودار RMSProp و Adam نیز توانسته اند همگرا شوند اما سرعت همگرایی شان از gd کمتر بوده.

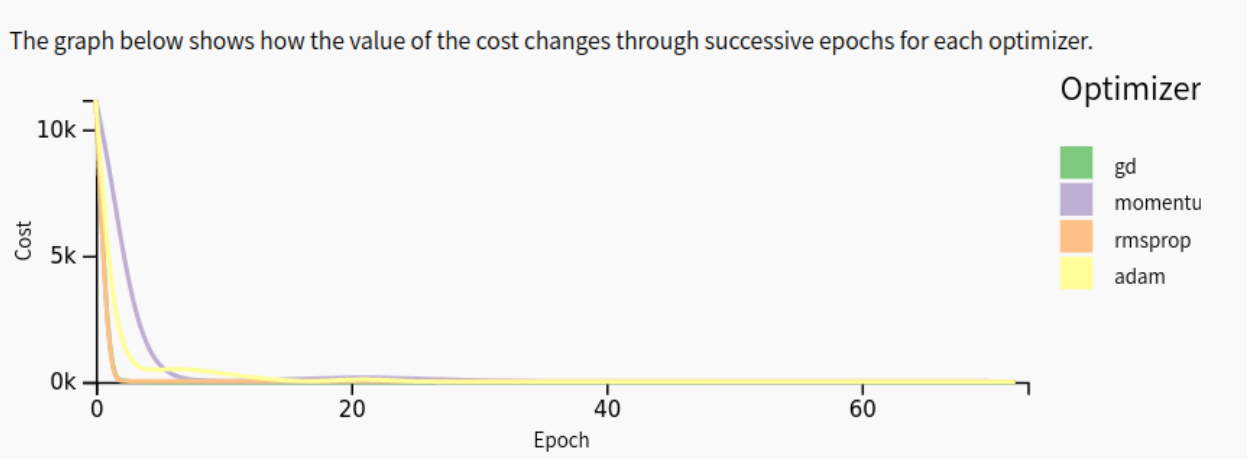




ب) بهترین مقادیر نرخ یادگیری و decay در تصویر زیر قابل مشاهده است (نمودار مربوط به gd و rmsprop روی یکدیگر قرار گرفته اند).در اثر آزمایش کردن مقادیر مختلف به این مقدارها رسیده ایم. اگر نرخ یادگیری بزرگ باشد از نقطه مینیمم گذر می کند و اگر کوچک باشد به تعداد ایپاک های بیشتری نیاز

دارد تا همگرا شود. همچنین اگر مقدار decay بزرگ باشد، نرخ یادگیری زیاد کوچک می شود و از سرعت آن کاسته می شود. اگر مقدار decay کوچک باشد، نرخ یادگیری به آرامی کم می شود و زمان زیادی طول می کشد تا همگرا شود.





4. روش های گوناگونی برای مقابله با catastrophic forgetting وجود دارد که در اینجا به چند مورد اشاره می کنیم:

متدهای replay: در این روش نمونه های آموزشی قبلی در یک حافظه ذخیره می شوند. زمانی که نمونه آموزشی جدید اضافه می شود، داده های جدید و قدیم با هم ترکیب می شوند و برای fine-tune کردن شبکه از آن استفاده می شود. در این روش اینکه چه داده هایی ذخیره شوند و به چه میزان، اهمیت دارد.

روش elastic weight consolidation: در این روش،با اضافه شدن نمونه های آموزشی جدید، وزن هایی که در گذشته اهمیت داشته اند باید به گونه ای انتشار یابند و بدین صورت مانع فراموش شدن داده های قبلی شویم.

روش های parameter isolation: در این روش پارامترهای مربوط به تسک مشخص را ایزوله می کنیم و این می تواند حداکثر ثبات در زیرمجموعه ای از تسک های گذشته را تضمین کند.

5. .

5) $\hat{y} = w_1 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + b$

SGD + momentum: $\begin{cases} v_{t+1} = \rho v_t + \nabla f(x_t) \\ x_{t+1} = x_t - \alpha v_{t+1} \end{cases}$

first batch

forward: $x_1 = 1, x_2 = -1 \rightarrow \hat{y} = 2$

$x_1 = 2, x_2 = 0 \rightarrow \hat{y} = 5$ $\rightarrow$ loss = 64

$$\frac{\partial loss}{\partial w_1} = \frac{\partial loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} = \frac{1}{2} \sum_i -2(y_i - \hat{y}_i) x_{1i}^2 = -\sum_i (y_i - \hat{y}_i) x_{1i}^2$$

$$\frac{\partial loss}{\partial w_2} = \frac{\partial loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} = -\sum_i (y_{0i} - \hat{y}_{0i}) x_{2i}^2$$

$$\frac{\partial loss}{\partial w_3} = \frac{\partial loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_3} = -\sum_i (y_{0i} - \hat{y}_{0i}) x_{1i} x_{2i}$$

$$\frac{\partial loss}{\partial b} = \frac{\partial loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = -\sum_i (y_i - \hat{y}_i) \cdot 1$$

برای انجام محاسبات از کد استفاده شده است.

$$\frac{\partial loss}{\partial w_1} = -40, \quad \frac{\partial loss}{\partial w_2} = -8, \quad \frac{\partial loss}{\partial w_3} = 8, \quad \frac{\partial loss}{\partial b} = -16$$

update: $w_1 = 5, \; w_2 = -0.2, \; w_3 = -1.8, \; b = 2.6$

second batch

forward: $x_1 = 0, x_2 = 2 \rightarrow \hat{y} = 1.8$

$x_1 = -1, x_2 = 1 \rightarrow \hat{y} = 9.2$ $\rightarrow$ loss = 55.84

$\frac{\partial loss}{\partial w_1} = 5.2, \frac{\partial loss}{\partial w_2} = -31.6, \frac{\partial loss}{\partial w_3} = -5.2, \frac{\partial loss}{\partial b} = -4$

update: $w_1 = 3.08, w_2 = 3.68, w_3 = -2, b = 4.44$

```python
w1 = 1
w2 = -1
w3 = -1
b = 1

forward = lambda x1, x2: w1*x1**2 + w2*x2**2 + w3*x1*x2 + b
```

```python
import numpy as np

def mse_loss(y, y_hat):
    return np.average((y - y_hat)**2)
```

```python
def grad_w1(y, y_hat, x1):
    return -np.sum((y-y_hat) * x1**2)

def grad_w2(y, y_hat, x2):
    return -np.sum((y-y_hat) * x2**2)

def grad_w3(y, y_hat, x1, x2):
    return -np.sum((y-y_hat) * x1 * x2)

def grad_b(y, y_hat):
    return -np.sum(y-y_hat)
```

```python
v_t = 0

def update(grad, x_t, v_t, rho=0.9, lr=0.1):
    v_t_1 = rho * v_t + grad
    x_t_1 =  x_t - lr * v_t_1
    return (v_t_1, x_t_1)
```

batch 1

[2]
```
print(forward(1, -1))
print(forward(2, 0))
```

```
2
5
```

[7]
```
y_b1 = np.array([10, 13])
y_hat_b1 = np.array([2, 5])
print(mse_loss(y_b1, y_hat_b1))
```

```
64.0
```

```
x1 = np.array([1, 2])
x2 = np.array([-1, 0])

print(grad_w1(y_b1, y_hat_b1, x1))
print(grad_w2(y_b1, y_hat_b1, x2))
print(grad_w3(y_b1, y_hat_b1, x1, x2))
print(grad_b(y_b1, y_hat_b1))
```

```
-40
-8
8
-16
```

```
v_t_w1, w1 = update(-40, 1, v_t)
v_t_w2, w2 = update(-8, -1, v_t)
v_t_w3, w3 = update(8, -1, v_t)
v_t_b, b = update(-16, 1, v_t)

print(w1)
print(w2)
print(w3)
print(b)
```

```
5.0
-0.19999999999999996
-1.8
2.6
```

6. الف) اگر مجموعه داده‌ها بسیار متمایز باشد، مدل ممکن است دچار overfit زودهنگام شود. اگر داده‌های بر خورده شامل مجموعه‌ای از مشاهدات مرتبط با ویژگی‌های قوی باشد، آموزش اولیه مدل می‌تواند به شدت به سمت آن ویژگی‌ها (یا بدتر از آن، به سمت ویژگی‌های اتفاقی که اصلاً به موضوع مرتبط نیستند) منحرف شود. به جهت جلوگیری از این اتفاق می توانیم نرخ یادگیری را در ابتدا مقدار کوچکی قرار دهیم اما در این صورت سرعت همگرایی کم خواهد شد. warmup روشی برای کاهش اثر برتری نمونه های آموزشی اولیه است. بدون آن، ممکن است لازم باشد در چندین ایپاک اضافی مدل را آموزش دهیم تا به همگرایی مورد نظر برسیم. این روش به شبکه کمک می‌کند تا به آرامی با داده‌ها سازگار شود. همچنین به دلیل اینکه وزن های مدل در ابتدای آموزش به صورت تصادفی مقداردهی می شوند، مدل در ابتدای آموزش با شبکه مد نظر ما بسیار متفاوت است. پس آموزش صحیح مدل در ایپاک های ابتدایی اهمیت دارد. استفاده از warmup میتواند به ثبات شبکه و جلوگیری از overfit کمک کند. Generalization مدل نیز بهتر خواهد شد. علاوه بر آن، می تواند باعث بهبود دقت مدل و همگرایی سریعتر شود.

ب) warmup مرحله ای در آغاز آموزش شبکه عصبی است که در آن با نرخ یادگیری بسیار کمتر از نرخ یادگیری "اولیه" شروع می کنیم و سپس آن را در چند ایپاک افزایش می دهیم تا به آن نرخ یادگیری "اولیه" برسیم. این کار از overfit زودهنگام پیشگیری می کند.

Learning rate decay: اگر در ابتدا نرخ یادگیری بالا (به دنبال decay آهسته) باشد، عملکرد شبکه عصبی را بهبود می بخشد. نرخ یادگیری پایین باعث می‌شود شبکه الگوهایی که زیاد تکرار می شوند مانند نویز را قبل از سایر الگوها بیاموزد.

می توانیم در ایپاک های ابتدایی آموزش با استفاده از warmup رفته رفته نرخ یادگیری را بالا ببریم و پس از آن با استفاده از روش های decay، آن را کاهش دهیم تا به عملکرد مطلوب برسیم.

ت) ابتدا یک مدل CNN تعریف می کنیم. تابع ضرر را کراس آنتروپی قرار می دهیم و داده MNIST را لود می کنیم.

```python
def net_fn():
    model = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
        nn.Linear(120, 84), nn.ReLU(),
        nn.Linear(84, 10))

    return model

loss = nn.CrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

سپس با استفاده از تابع train، مدل را آموزش می دهیم. در اینجا از scheduler نیز استفاده شده است.

```python
# The code is almost identical to `d2l.train_ch6` defined in the
# lenet section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
          scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

        if scheduler:
            if scheduler.__module__ == lr_scheduler.__name__:
                # Using PyTorch In-Built scheduler
                scheduler.step()
            else:
                # Using custom defined scheduler
                for param_group in trainer.param_groups:
                    param_group['lr'] = scheduler(epoch)

    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

سپس از cosine scheduler استفاده می کنیم. فرمول آن مطابق زیر است:

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2}(1 + \cos(\pi t/T))$$

پیاده سازی آن:

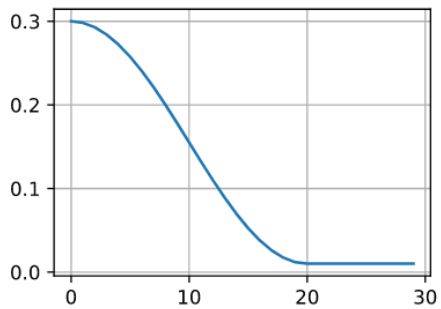
```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
                 warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr

scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```

از warmup خطی در پیاده سازی scheduler استفاده شده است. مقدار warmup_steps تعداد ایپاک هایی که در آن warmup انجام می شود را تعیین می کند که در این مثال به دلیل اینکه این مقدار ۰ است، warmup نداریم و فقط scheduler داریم. یعنی نرخ یادگیری با مقدار base_lr = 0.3 شروع شده و به تدریج کاهش می یابد.

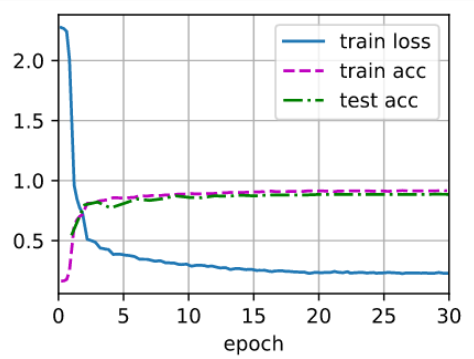


سپس مدل را آموزش می دهیم:

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

خروجی به صورت زیر است:

```
train loss 0.229, train acc 0.916, test acc 0.886
```



در حالت بعدی از scheduler به همراه warmup استفاده می کنیم.

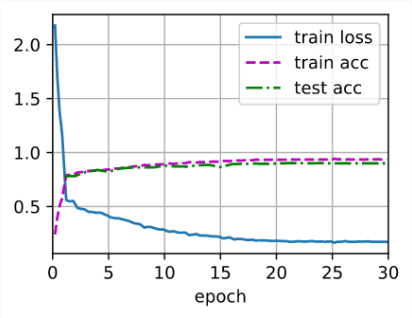
```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```

و مدل را آموزش می دهیم:

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

خروجی آموزش به صورت زیر است:

```
train loss 0.173, train acc 0.936, test acc 0.902
```



در مقایسه، میزان دقت آموزش مدلی که نه از scheduler و نه از warmup استفاده کرده است بالاتر است اما معیار سنجش، دقت آموزش نیست. در این حالت دقت آزمون کمتر از ۲ حالت دیگر است. دقت آزمون در حالتی که هم از scheduler و هم از warmup استفاده کرده است ۰.۹ و دقت آزمون با تنها استفاده از scheduler، به ۰.۸۸ می رسد. در حالتی که از warmup استفاده کرده ایم میزان کم شدن loss بیشتر است و سریعتر همگرا می شود و تعمیم دهی بهتری دارد.

منابع

https://stackoverflow.com/questions/62690725/small-learning-rate-vs-big-learning-rate

https://arxiv.org/pdf/1909.08383.pdf

https://stackoverflow.com/questions/55933867/what-does-learning-rate-warm-up-mean

https://www.mdpi.com/2079-9292/10/16/2029

https://www.reddit.com/r/MachineLearning/comments/es9qv7/d_warmup_vs_initially_high_learning_rate/