DSCI 565: OPTIMIZATION ALGORITHMS

This content is protected and may not be shared, uploaded, or distributed.

Ke-Thia Yao

Lecture 17: 2025-10-29

Sparse Features and Learning Rates

- In natural language, some words occur much more often than others, e.g., the words *learning vs preconditioning*
- A global learning rate schedule for all words may cause the rate to decrease too slowly for some words and too fast for other words
- How to keep track of which feature is frequent, and which is rare?
- Adagrad proposes maintaining a summary of past gradient variances for all parameters

Adagrad

- Use variable $\mathbf{s}_t$ to accumulate past gradient variance
- $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ computes the loss wrt to sample $(\mathbf{x}_t, y_t)$ and weight $\mathbf{w}$
- $\mathbf{g}_t$ is the gradient of the loss
- Initially $\mathbf{s}_0 = \mathbf{0}$
- $\mathbf{w}_t$ is the weight at time t
- For some fixed η, and fixed $\epsilon > 0$ to prevent division by zero

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$$

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.$$

Adagrad Rationale

- Why $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$?
  - Similar to analysis of Newton's method we want to precondition with the diagonal of the Hessian matrix
  - For deep learning even computing the diagonal of second derivatives is difficult due to memory constraints
  - But the diagonal elements does depend on the gradients
- Why $-\frac{\eta}{\sqrt{(\mathbf{s}_t+\epsilon)}}$?
  - For stochastic gradient descent, under convexity assumption the optimal rate schedule is $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$
  - See section 12.4.3
  - Here instead of the number of steps T, we use $\mathbf{s}_t$

RMSProp

$S_t = \gamma S_{t-1} + (1-\gamma) g_t^2$

- For Adagrad the state $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$ can grow without bound
- This can cause adaptive learning rate $\frac{\eta}{\sqrt{(\mathbf{s}_t+\epsilon)}}$ to decay too quickly
- RMSProp proposes using leaky averaging γ

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2,$$

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.$$

- For some fixed η, and for small $\epsilon > 0$, typically $\epsilon = 10^{-6}$

Adadelta

- Enhances Adagrad by
  - Using leaky average for state variable $\mathbf{s}_t$, similar to RMSProp
  - Decoupling per-coordinate scaling with learning rate adjustment
    - Which also removes the need for a global learning rate η
    - PyTorch implementation still has learning rate but defaults to $\eta = 1$

Adadelta

- Uses two state variables
  - Variable $\mathbf{s}_t$ for maintaining the variance of the gradient (with leaky average)

$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2.$$

  - Variable Δx_t to store the variance of the parameter (with leaky average)
    - Removing the need to manually select a global rate η (no need)

modified gradient

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}'_t. \qquad \mathbf{g}'_t = \frac{\sqrt{\Delta \mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t \qquad \Delta \mathbf{x}_t = \rho \Delta \mathbf{x}_{t-1} + (1-\rho)\mathbf{g}'^2_t.$$

    - With $\Delta \mathbf{x}_0 = 0$, and small $\epsilon > 0$

Adam

- Adam combines all techniques we have seen
  - Minibatch stochastic gradient descent
  - Momentum
  - Per-coordinate scaling (Adagrad, RMSProp)
  - Decoupling per-coordinate scaling from learning rate adjustment (Adadelta)

Adam

momentum + variance of the gradient

- Uses two state variables with leaky averages
  - $\mathbf{v}_t$ for the momentum
  - $\mathbf{s}_t$ for the variance of the gradient
  - Initialize $\mathbf{v}_0 = \mathbf{s}_0 = 0$

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1)\mathbf{g}_t,$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2.$$

- Typically, $\beta_1 = 0.9$ and $\beta_2 = 0.999$
  - Variance estimate moves much more slowly than momentum
- At startup biased toward smaller values, rescale:

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}$$

- Update equation

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t \qquad \mathbf{g}'_t = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$$

AdamW

- Adam with decoupled weight decay

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta \left(\frac{\hat{\mathbf{v}}_t}{\sqrt{\hat{s}_t} + \epsilon} + \lambda \mathbf{x}_{t-1} \right)$$

  where $\lambda \in \mathbb{R}$

- Currently, the most popular for training big models
- Loshchilov and Hutter, 2019

Learning Rate Scheduling

- So far focused on algorithms for updating weight vectors rather than the rate at which they are updated
- Aspect of rate scheduling
  - Magnitude
  - Decay
  - Initialization

Toy Problem

- Convolutional network with cross entropy loss on FashionMNIST data

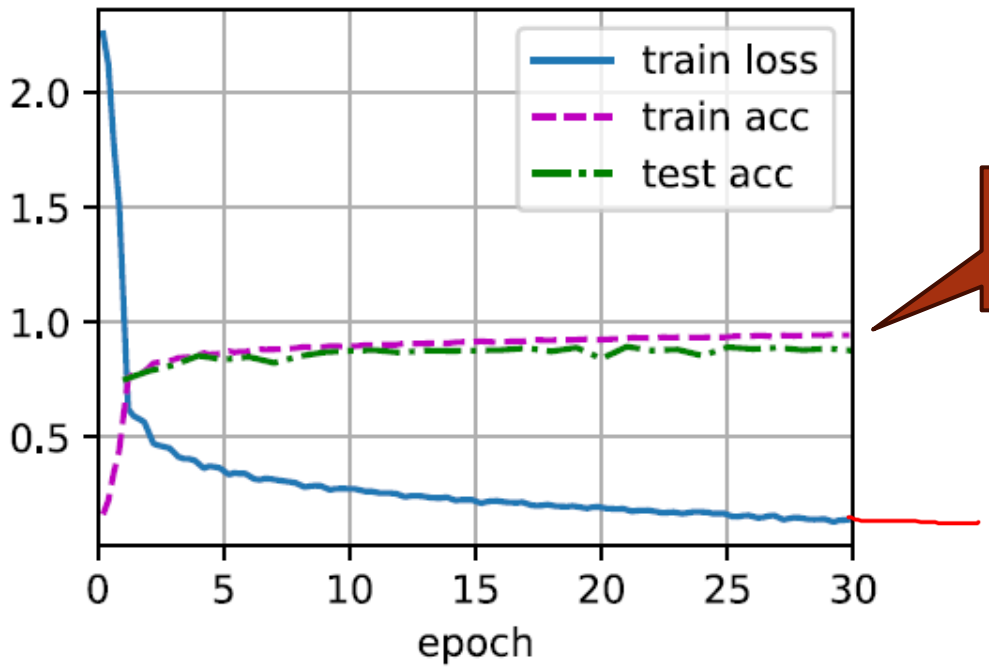
```
def net_fn():
    model = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 *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

    return model

loss = nn.CrossEntropyLoss()
```

Baseline with No Schedule


```
lr, num_epochs = 0.3, 30
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

Constant learning rate

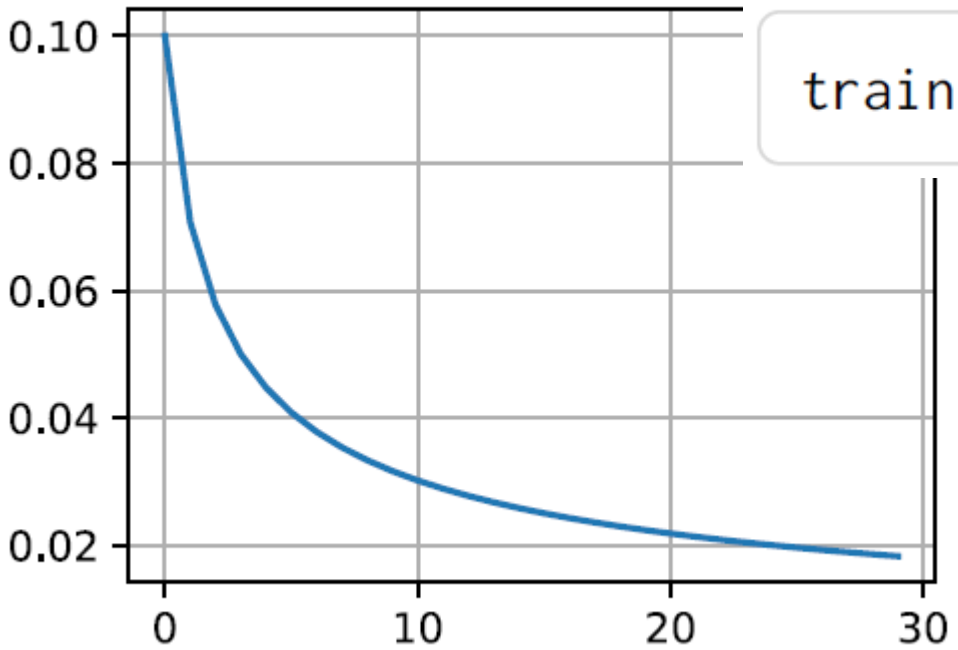
```
train loss 0.145, train acc 0.944, test acc 0.877
```



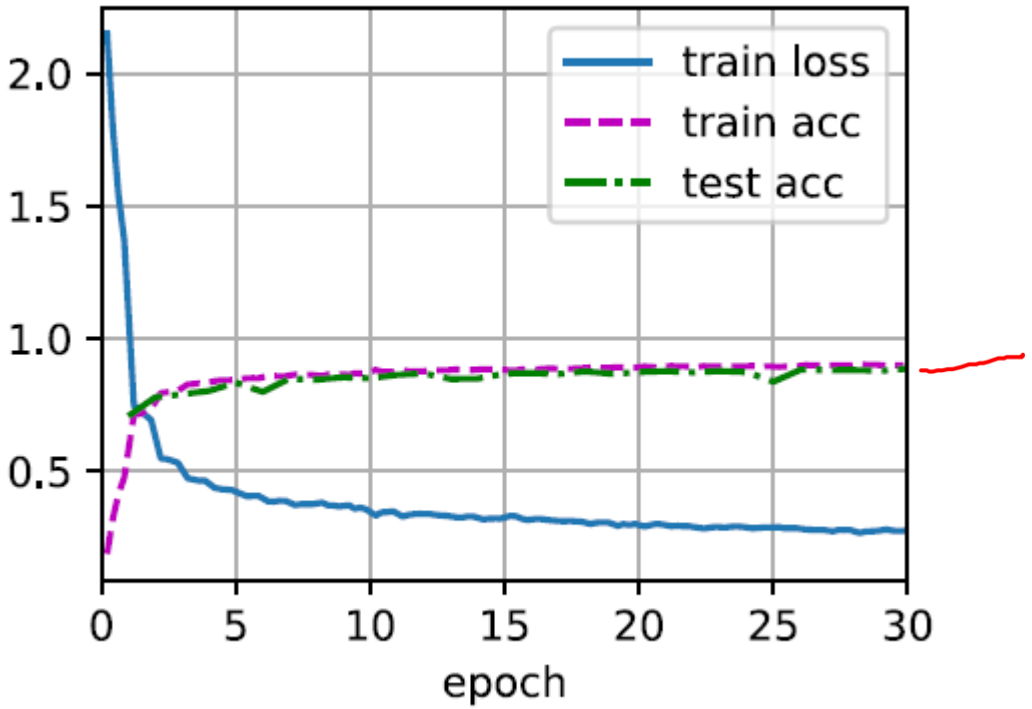
Gap between train and test

Square Root Scheduler

- $\eta = \eta_0(t+1)^{-\frac{1}{2}}$ with $\eta_0 = 0.1$




```
train loss 0.273, train acc 0.900, test acc 0.886
```

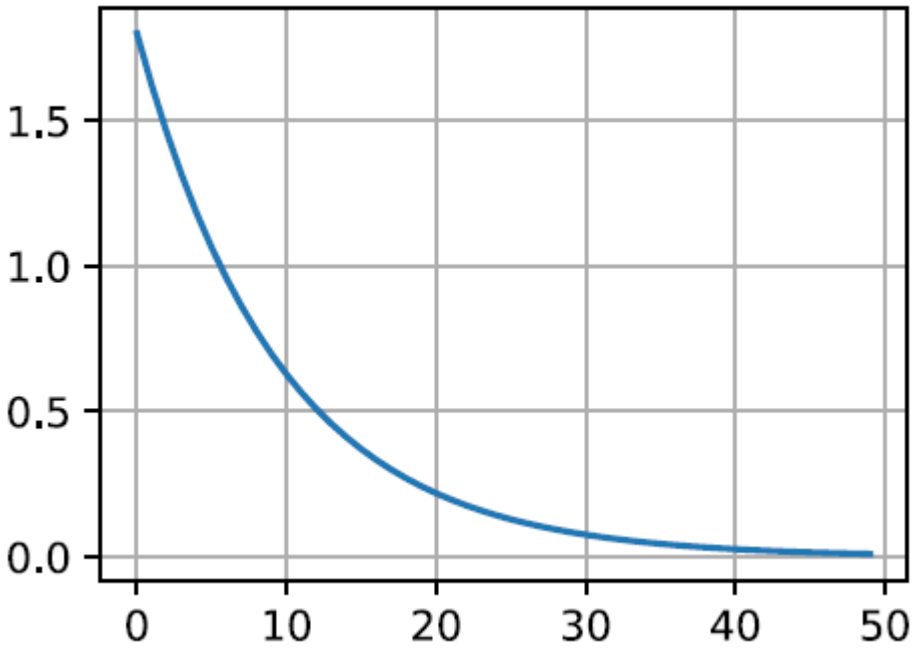


- Good for convex problems
- Slower decrease for nonconvex

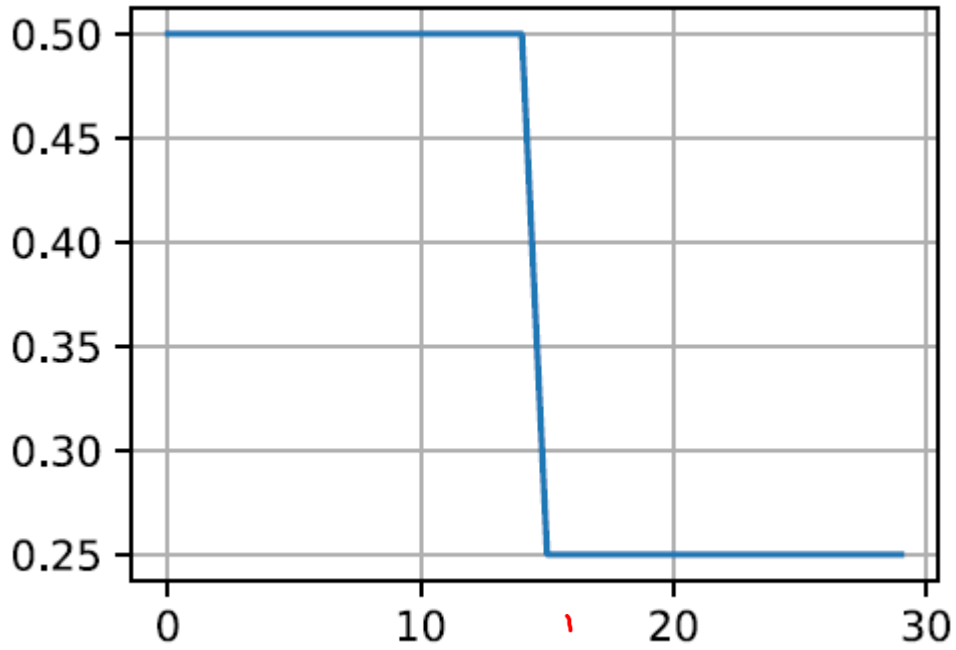
Factor Scheduler

- Factor scheduler

  $\eta_{t+1} \leftarrow \alpha\, \eta_t$ for $\alpha \in (0,1)$

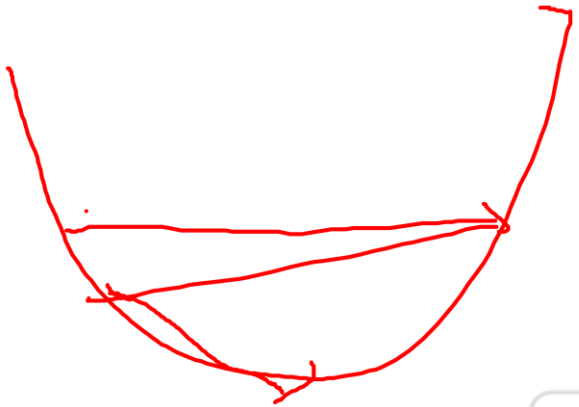


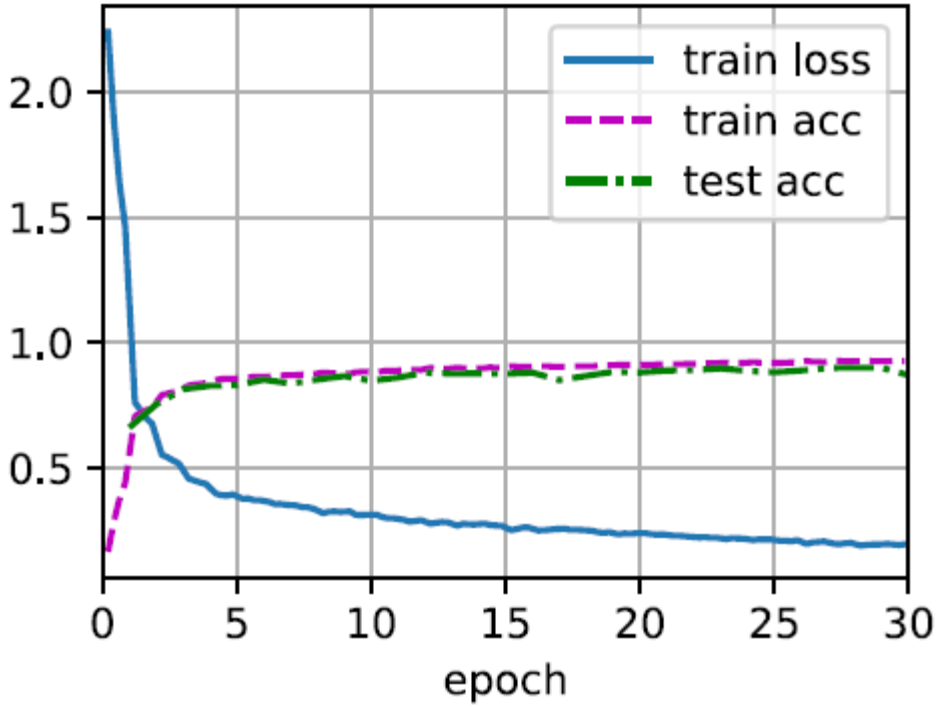
- Multifactor scheduler
  - Apply factor scheduler update only at certain time steps $s = \{15, 30, 45\}$



Multifactor Scheduler

- Intuition for piecewise constant learning rate
  - Optimize until near local optima
  - Then decrease rate to get even closer



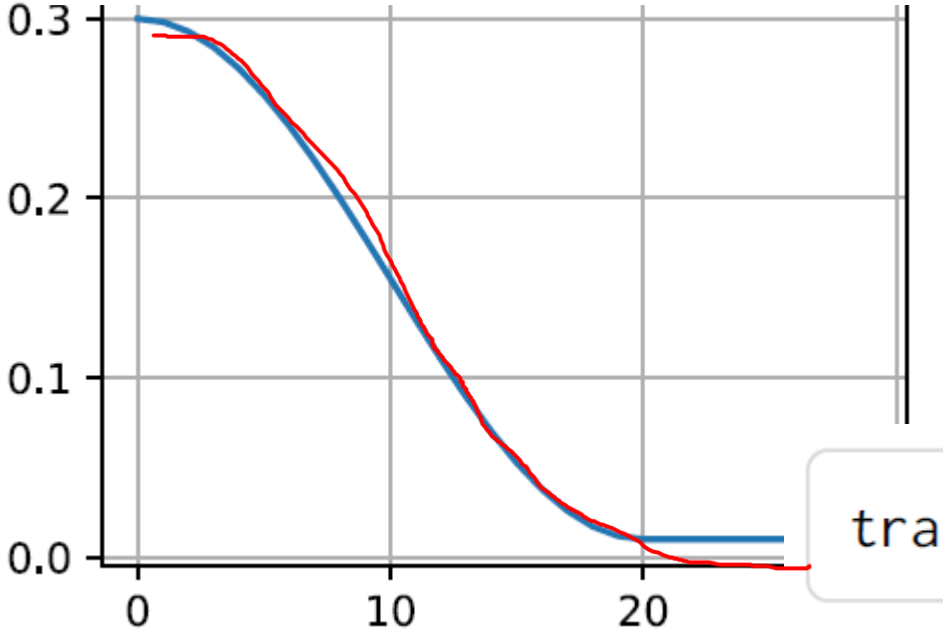



```
train loss 0.194, train acc 0.927, test acc 0.869
```

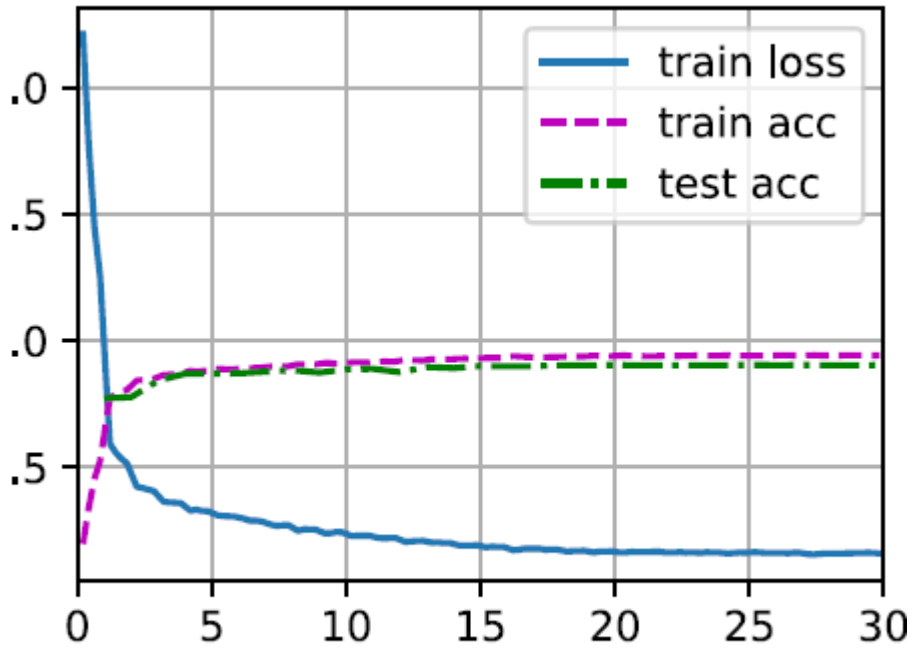
Cosine Scheduler

- Larger rates in the beginning
- Very small rates to refine solution

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



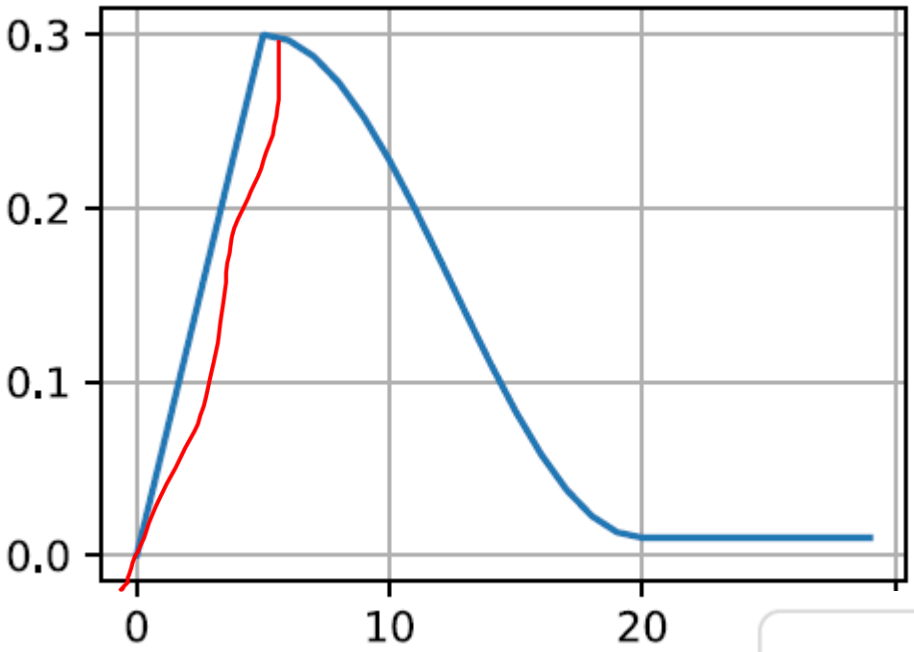
- Popular for computer vision

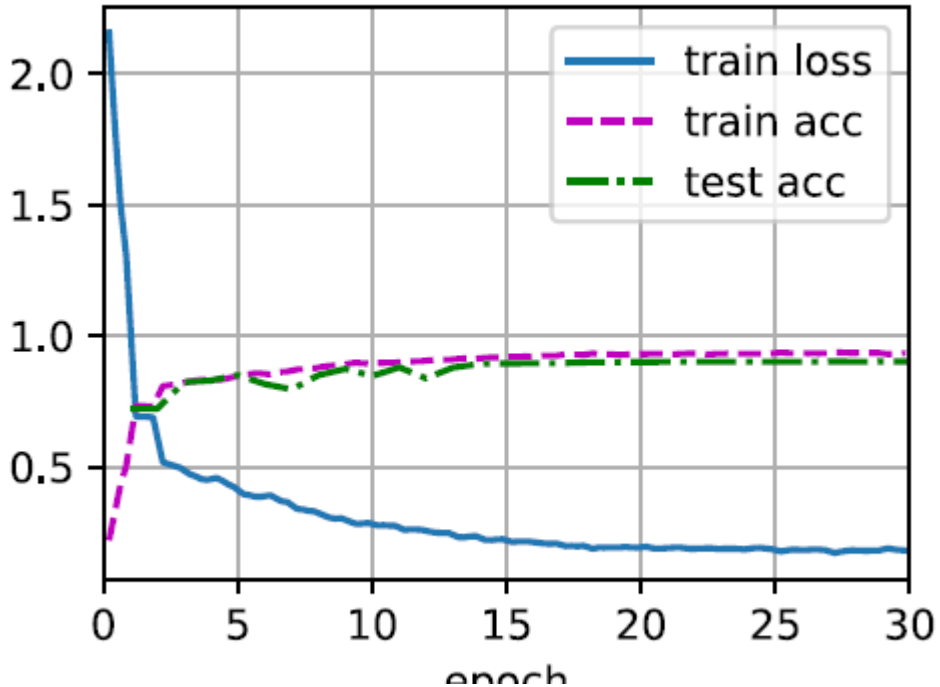



```
train loss 0.159, train acc 0.942, test acc 0.904
```

Warmup

- Sometimes large rates at startup cause optimization to diverge
- Start with small rates then switch to cosine






```
train loss 0.181, train acc 0.934, test acc 0.901
```