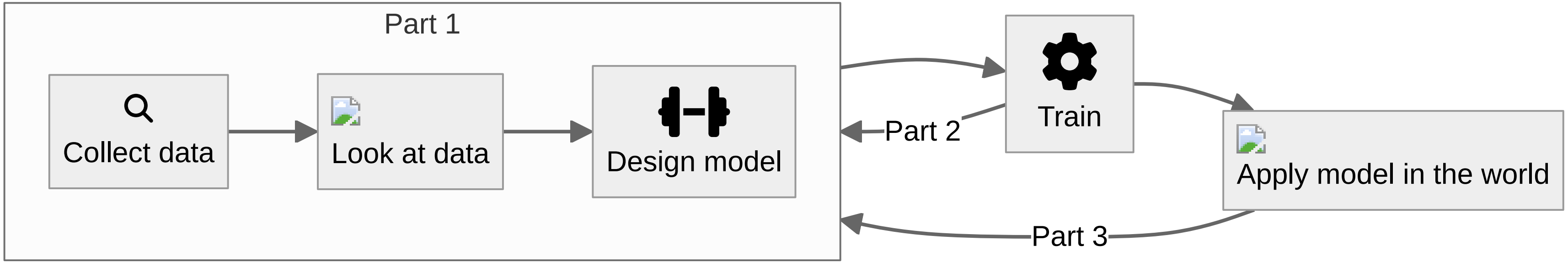
Advanced Training

Recap: Developing a Model



Recap: Stochastic Gradient Descent

Default Optimizer

✓ Works well in most cases

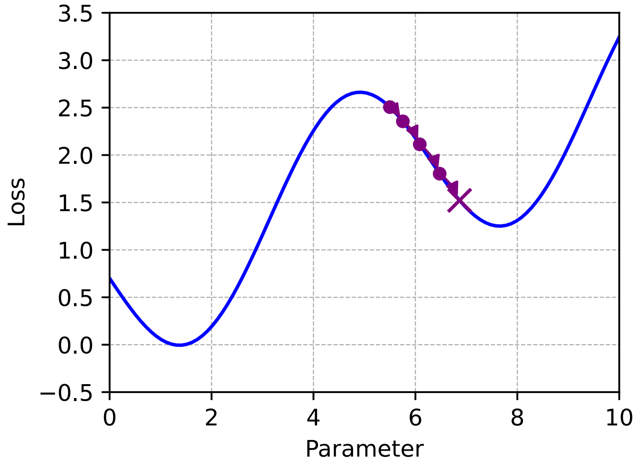
✗ Need to tune learning rate

Stochastic Gradient Descent (with Momentum)

```
m = 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = J + momentum * m
    θ = θ - ϵ * m.mT
```

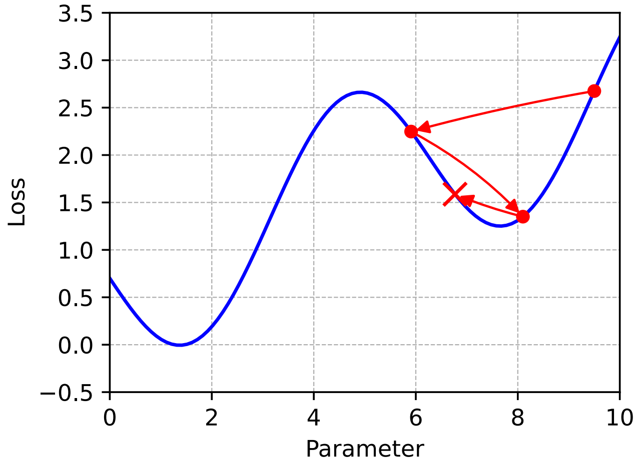
Recap: Learning Rate Magnitude

Learning Rate Too Low



- Slow training

Learning Rate Too High

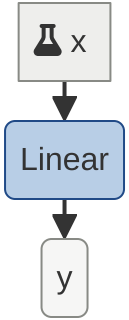


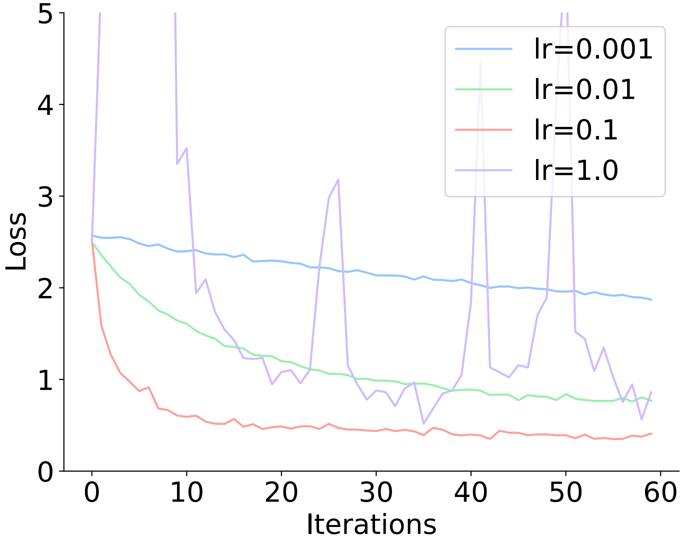
- Convergence
- Extreme case - NaNs!

Example: Learning Rates for Linear Network

Digit Classification With MNIST

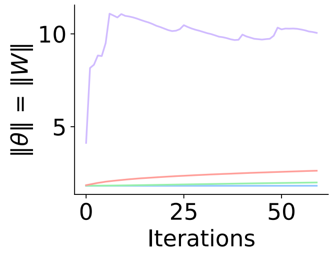
- Loss spikes when learning rate is too high
- Updates are too large!



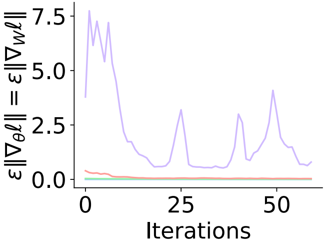


Example: Learning Rates for Linear Network

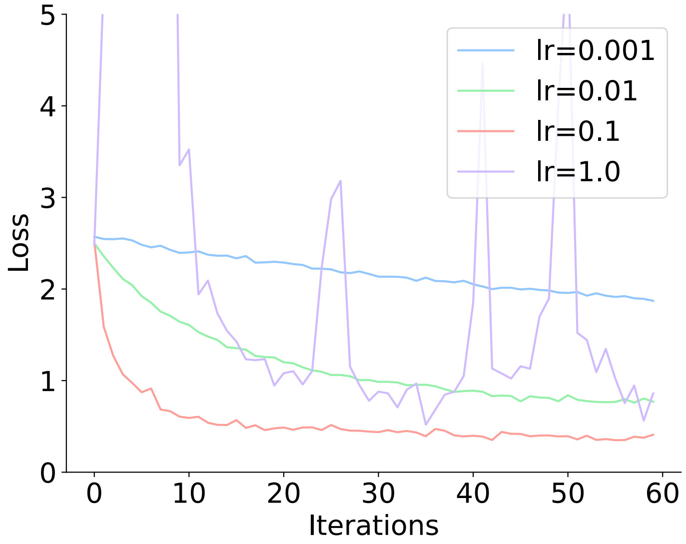
Why Does the Loss Spike?



weights are much larger



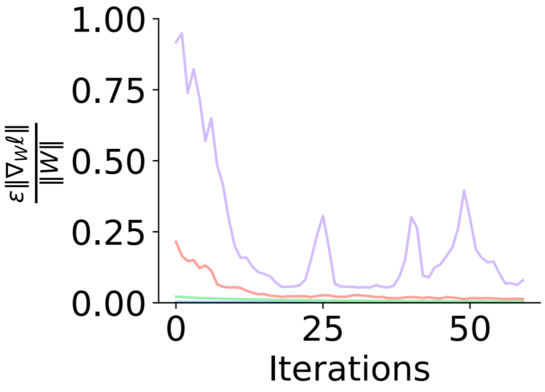
steps are much larger

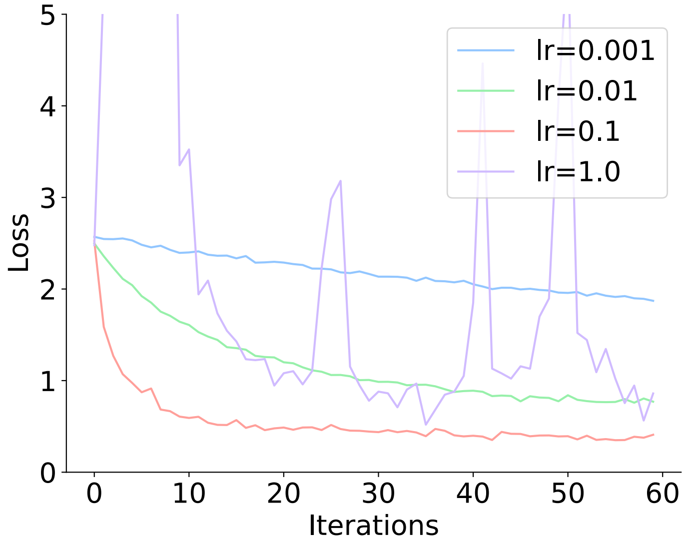


Example: Learning Rates for Linear Network

Why Does the Loss Spike?

Step size is as large as the weights!

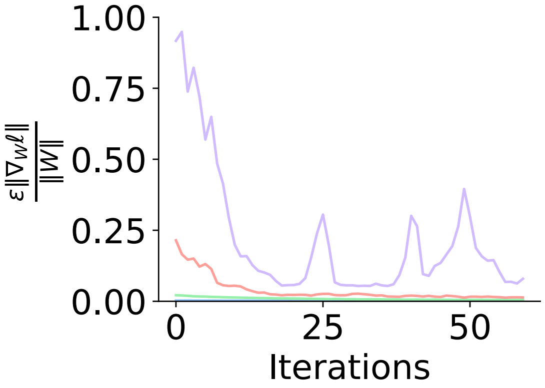


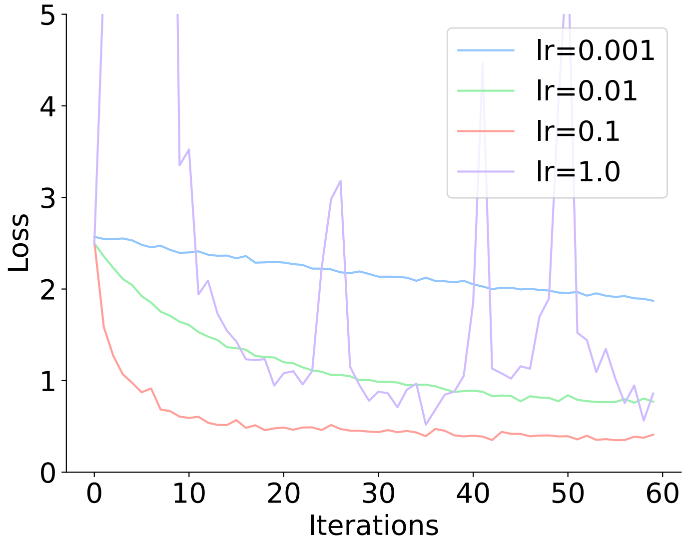


Example: Learning Rates for Linear Network

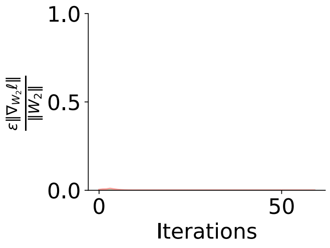
Solution: largest learning rate without this behavior

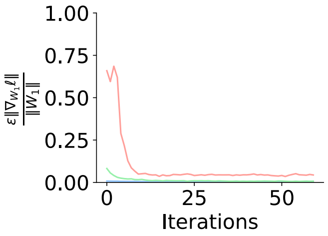
In this case `lr=0.1`

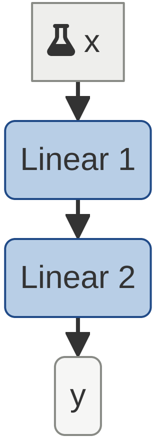




Example: Learning Rates for Deep Network







`lr=0.1` works fine

`lr=0.01` works OK

`lr=0.1` is too large

`lr=0.01` works fine

Solution: minimum learning rate without any spiking

In this case `lr=0.01`

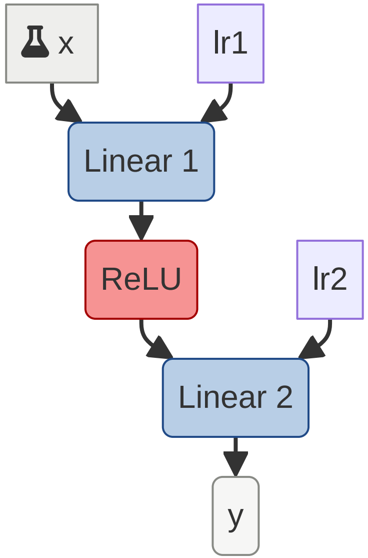
Can We Do Better?

Idea: Different learning rate per layer

- Each layer gets to learn faster

How do we set these learning rates?

- By hand



Can We Do Better?

Idea: Different learning rate per layer

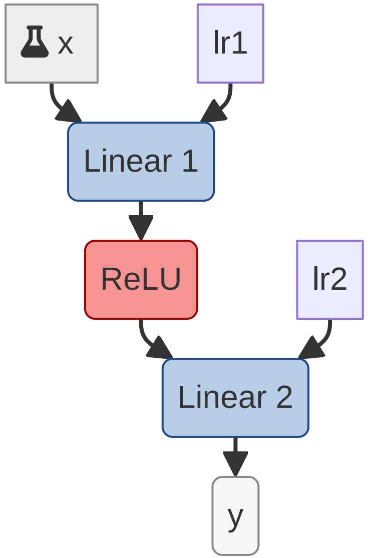
Before

```
optim = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum)
```

After

```
optim1 = torch.optim.SGD(model.layer1.parameters(), lr=lr1, momentum=momentum)
optim2 = torch.optim.SGD(model.layer2.parameters(), lr=lr2, momentum=momentum)
optim3 = torch.optim.SGD(model.layer3.parameters(), lr=lr3, momentum=momentum)
...
```

Problem: many more hyperparameters



RProp

Scale gradients by $\frac{1}{||\nabla_\theta \ell||}$

- uses only the sign of the gradient

✓ All updates are now the same norm

✓ Automatically sets different learning rates

✗ Different norms across batches

RProp [1]

```
m = 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = J / J.norm() + momentum * m
    θ = θ - ϵ * m.mT
```

1. G. Hinton "Lecture 6.4: A separate, adaptive learning rate for each connection" Neural networks for ML 2012 ↩️

RProp to RMSProp

RProp [1]

```
m = 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = J / J.norm() + momentum * m
    θ = θ - ϵ * m.mT
```

RMSProp [2]

```
m, v = 0, 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    v = β_2 * v + (1-β_2) * J.square()
    m = J / v.sqrt() + momentum * m
    θ = θ - ϵ * m.mT
```

1. G. Hinton "Lecture 6.4: A separate, adaptive learning rate for each connection" Neural networks for ML 2012 ↩️
2. T. Tieleman, G. Hinton "Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude." Neural networks for ML 2012 ↩️

RMSProp

Compute v, a running average of $||\nabla_\theta \ell||^2$

Scale gradients by $\frac{1}{\sqrt{v}}$

✓ Autotunes learning rate using prior gradients

✓ Plays well with mini-batches

✗ Not effectively using momentum

RMSProp [1]

```
m, v = 0, 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    v = β_2 * v + (1-β_2) * J.square()
    m = J / v.sqrt() + momentum * m
    θ = θ - ϵ * m.mT
```

1. T. Tieleman, G. Hinton "Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude." Neural networks for ML 2012 ↩

RMSProp to Adam V0

RMSProp [1]

```
m, v = 0, 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    v = β_2 * v + (1-β_2) * J.square()
    m = J / v.sqrt() + momentum * m
    θ = θ - ϵ * m.mT
```

Adam v0 [2]

```
m, v = 0, 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    v = β_2 * v + (1-β_2) * J.square()
    m = J + momentum * m
    b = m / v.sqrt()
    θ = θ - ϵ * b.mT
```


1. T. Tieleman, G. Hinton "Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude." Neural networks for ML 2012 ↩️
2. Kingma et al. Adam: a Method for Stochastic Optimization. ICLR 2015 ↩️

Adam V0

Compute v, a running average of $||\nabla_\theta \ell||^2$

Scale gradients **with momentum** by $\frac{1}{\sqrt{v}}$

✓ Momentum is effectively used

✗ `v` is small at the beginning of training (`β_2=0.999`)

Adam v0 [^2]

```
m, v = 0, 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    v = β_2 * v + (1-β_2) * J.square()
    m = J + momentum * m
    b = m / v.sqrt()
    θ = θ - ϵ * b.mT
```

Adam V0 to Adam

Adam v0

```
m, v = 0, 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    v = β_2 * v + (1-β_2) * J.square()
    m = J + momentum * m
    b = m / v.sqrt()
    θ = θ - ϵ * b.mT
```

Adam [1]

```
m, v, t = 0, 0, 1
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = (1-β_1) * J + β_1 * m
    v = β_2 * v + (1-β_2) * J.square()
    m = m / (1 - β_1^t)
    v = v / (1 - β_2^t)
    b = m / v.sqrt()
    θ = θ - ϵ * b.mT
    t += 1
```

1. Kingma et al. Adam: a Method for Stochastic Optimization. ICLR 2015 ↩

Adam

Bias Correction

- Divide by $1 - \beta_1^t$ and $1 - \beta_2^t$

✓ Training starts with properly scaled terms

✗ Mathematically not correct*

*Fixed by `amsgrad=True` variant

Adam [1]:

```
m, v, t = 0, 0, 1
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = (1-β_1) * J + β_1 * m
    v = β_2 * v + (1-β_2) * J.square()
    m = m / (1 - β_1^t)
    v = v / (1 - β_2^t)
    b = m / v.sqrt()
    θ = θ - ϵ * b.mT
    t += 1
```

1. Kingma et al. Adam: a Method for Stochastic Optimization. ICLR 2015 ↩

Adam to AdamW

Adam [1]:

```
m, v, t = 0, 0, 1
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = (1-β_1) * J + β_1 * m
    v = β_2 * v + (1-β_2) * J.square()
    m = m / (1 - β_1^t)
    v = v / (1 - β_2^t)
    b = m / v.sqrt()
    θ = θ - ϵ * b.mT
    t += 1
```

AdamW [2]:

```
m, v, t = 0, 0, 1
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = (1-β_1) * J + β_1 * m
    v = β_2 * v + (1-β_2) * J.square()
    m = m / (1 - β_1^t)
    v = v / (1 - β_2^t)
    b = m / v.sqrt()
    θ = θ - ϵ * (b.mT + decay * θ)
    t += 1
```


1. Kingma, D. P., & Ba, J. L. Adam: a Method for Stochastic Optimization. ICLR 2015 ↩️
2. Loshchilov, I., & Hutter, F. Decoupled weight decay regularization. ICLR 2019 ↩️

AdamW

Weight Regularization Done Separately

- more on this later

✓ Works well with momentum

✓ Different learning rate for each parameter

✓ Compatible with mini-batches

✗ Memory intensive

AdamW [1]:

```
m, v, t = 0, 0, 1
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    m = (1-β_1) * J + β_1 * m
    v = β_2 * v + (1-β_2) * J.square()
    m = m / (1 - β_1^t)
    v = v / (1 - β_2^t)
    b = m / v.sqrt()
    θ = θ - ϵ * (b.mT + decay * θ)
    t += 1
```

1. Loshchilov, I., & Hutter, F. Decoupled weight decay regularization. ICLR 2019 ↩

Lion

Symbolic Programming Search

- Automatically* improved optimizer

Directly uses sign of the momentum

✓ Memory efficient (does not store `v`)

✗ Sensitive to batch size

Lion [1]

```
m = 0
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    b = (1-β_1) * J + β_1 * m
    b = sign(b)
    θ = θ - ϵ * (b.mT + decay * θ)

    m = (1-β_2) * J + β_2 * m
```

1. Chen X., Liang C., Huang D., Real E., Wang K., Liu Y., Pham H., Dong X., Luong T., Hsieh C.-J., Lu Y., Le Q.V. Symbolic discovery of optimization algorithms. ↩

Can We Do Better?

~~Idea: Different Learning Rate Per Layer~~

~~Before~~

```
optim = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum)
```

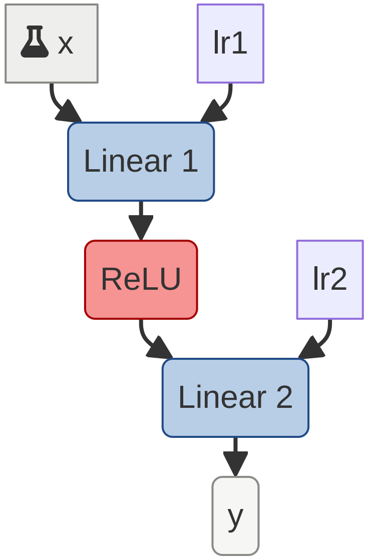
~~After~~

```
optim1 = torch.optim.SGD(model.layer1.parameters(), lr=lr1, momentum=momentum)
optim2 = torch.optim.SGD(model.layer2.parameters(), lr=lr2, momentum=momentum)
optim3 = torch.optim.SGD(model.layer3.parameters(), lr=lr3, momentum=momentum)
...
```

~~Problem: many more hyperparameters~~

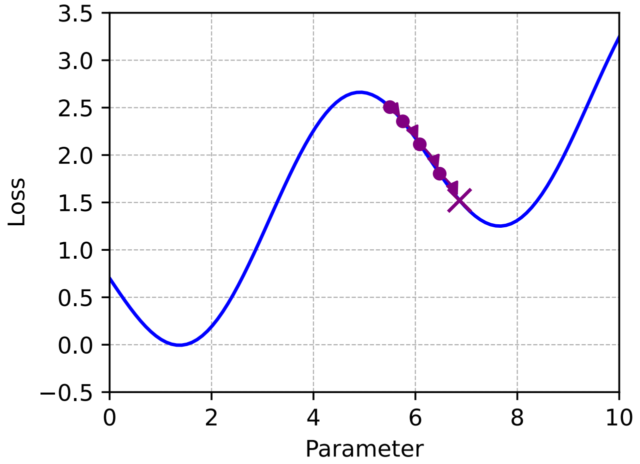
Use AdamW or LION

```
optim = torch.optim.AdamW(model.parameters(), lr=lr)
```



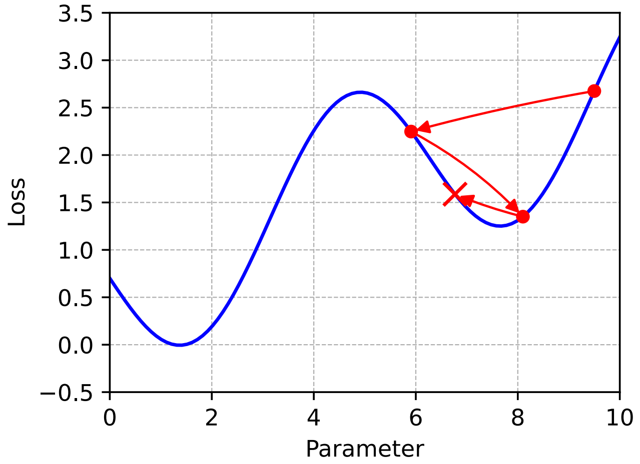
Recap: Learning Rate Magnitude Matters

Learning Rate Too Low



- Slow training

Learning Rate Too High

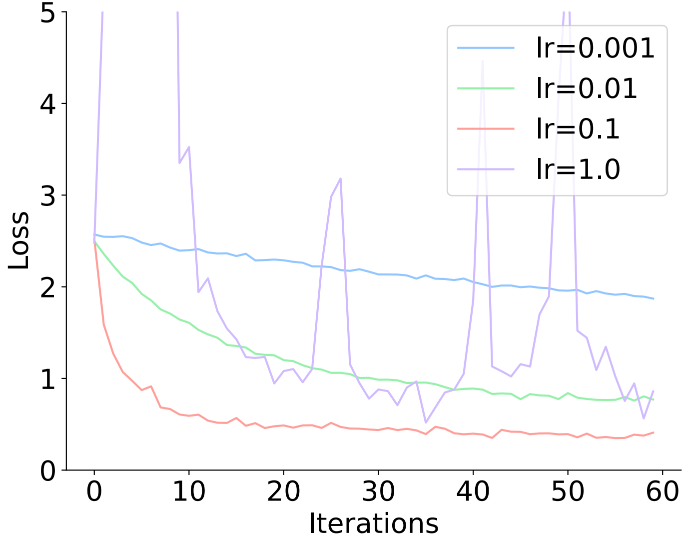


- Convergence
- Extreme case - NaNs!

Recap: What Learning Rate to Use?

Rule of thumb: largest learning rate that trains

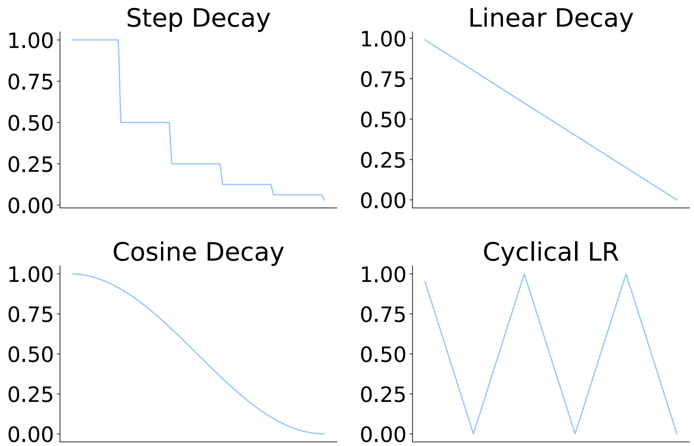
- Train for a few epochs
- Measure validation accuracy



Learning Rate Schedules

A fixed learning rate will eventually stop making progress...

Idea: change learning rate over time



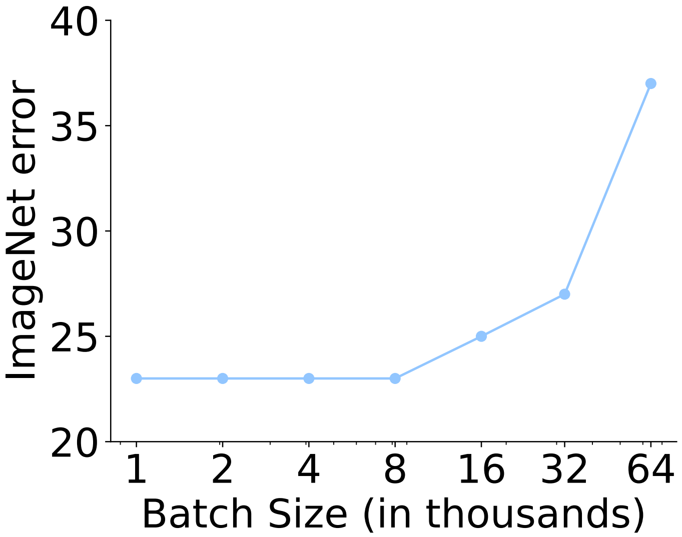
Learning Rate vs. Batch Size

Bigger batch sizes $\rightarrow$ faster convergence

Linear Scaling Rule [1]:

When the minibatch size is multiplied by k,

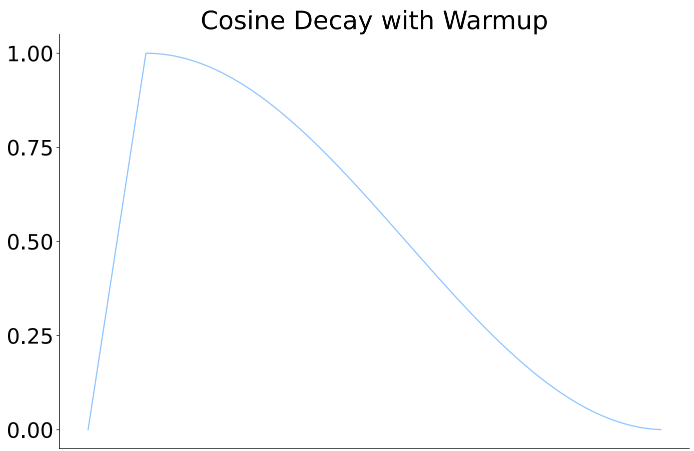
multiply the learning rate by k (fixing # epochs)

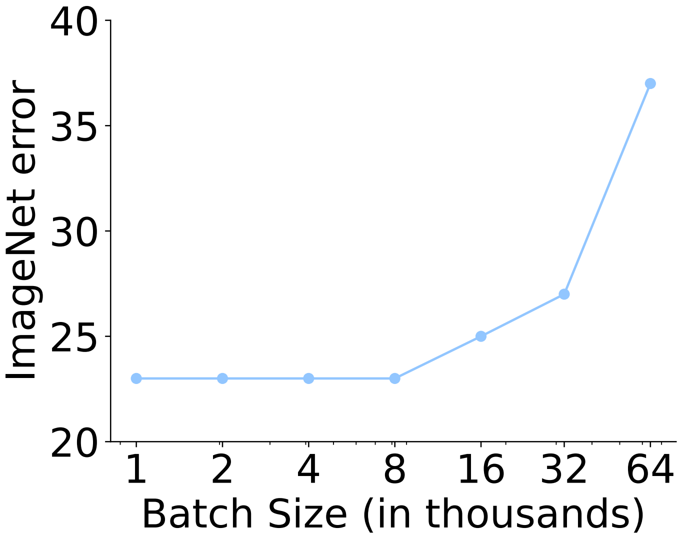


1. Priya Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017 ↩

Learning Rate Warmup

Network updates rapidly at the beginning of training [1]:





1. Priya Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017 ↩

Advanced Training - TL;DR

Optimizers adaptively scale learning rates

Use AdamW as default, LION for memory-expensive applications

Pick LR close to the largest LR that trains

Cosine schedule with warmup works well