UNT
UNIVERSITY OF NORTH TEXAS

CSCE 5218 & 4930
Deep Learning

Neural Network Training

Slides adapted from Adriana Kovashka

Plan for this lecture

- Tricks of the trade
  - Preprocessing, initialization, normalization
  - Dealing with limited data
- Convergence of gradient descent
  - How long will it take?
  - Will it work at all?
- Different optimization strategies
  - Alternatives to SGD
  - Learning rates
  - Choosing hyperparameters
- How to do the computation
  - Computation graphs
  - Vector notation (Jacobians)

Convergence of training

Successful training

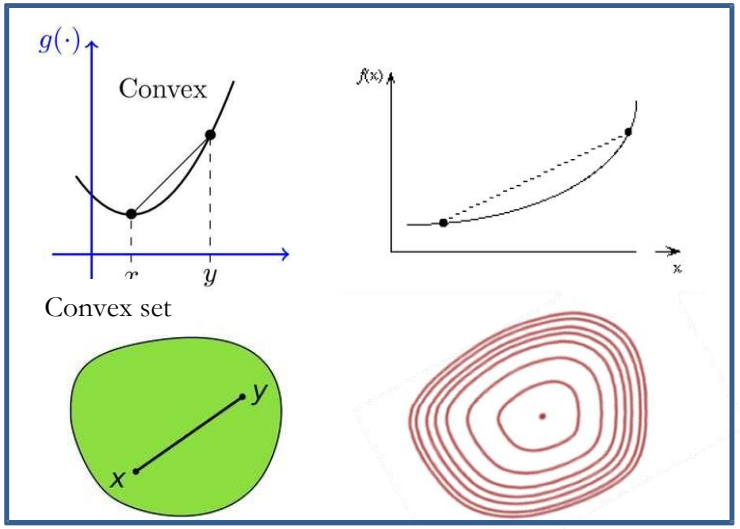
- We want training to converge (stop) at a reasonable place
- Stopping is not guaranteed – e.g. imagine taking larger and larger steps…
- Stopping in a good place is not guaranteed

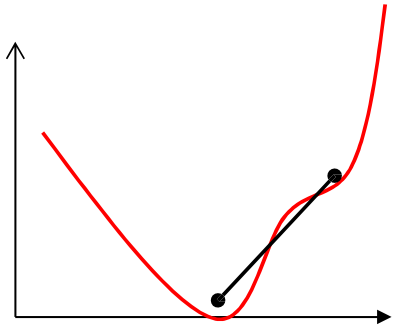
Loss surfaces

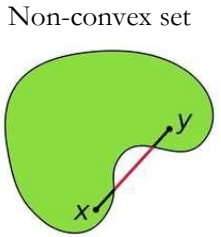
- Usually Loss(W) is not convex, so there are many local minima
- However, in deep networks, these minima are reasonably similar – not true in small networks
- What are desirable properties of the loss surface?

Convexity

- A surface is "convex" if it continuously curves upward
  - We can connect any two points above the surface without intersecting it
  - Many mathematical definitions that are equivalent
- Caveat: Neural net loss surface generally not convex



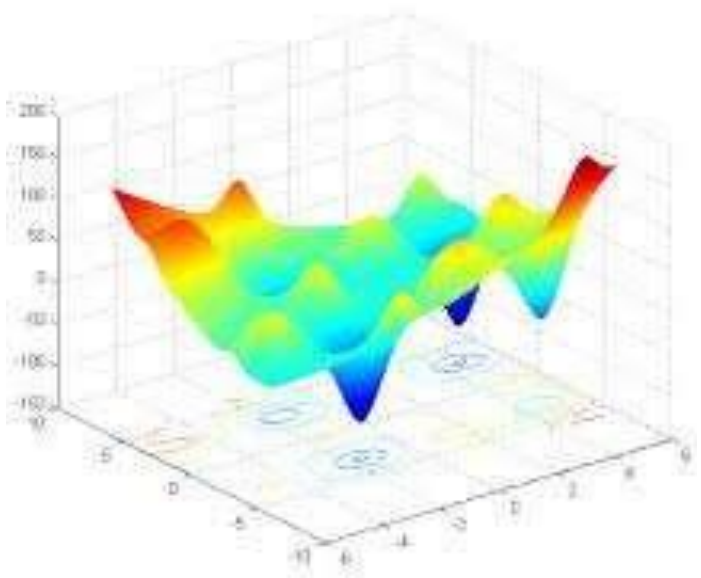




Adapted from Bhiksha Raj

The loss surface

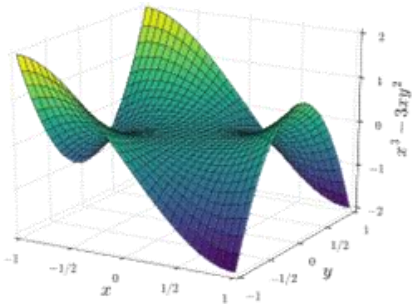
- Gradient descent makes the assumption that loss/objective has a single global optimum
- What about local optima?

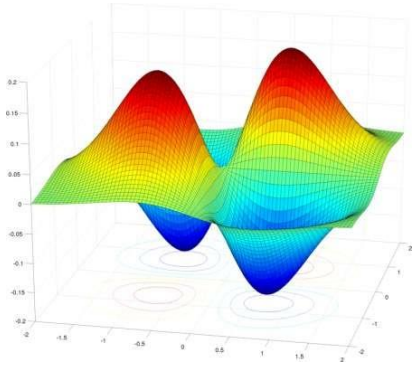


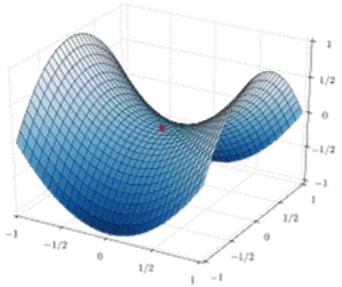
Adapted from Bhiksha Raj

The loss surface

- Popular hypothesis:
  - Most local minima are equivalent
    - And close to global minimum
  - This is not true for small networks
  - In large networks, saddle points are far more common than local minima
    - Frequency exponential in network size
- Saddle point: A point where:
  - The slope is zero
  - The surface increases in some directions, but decreases in others
    - Some of the Eigenvalues of the Hessian are positive; others are negative
  - Gradient descent algs often get "stuck" in saddle points







Conditions for convergence

- So far we have assumed training arrives at a local minimum

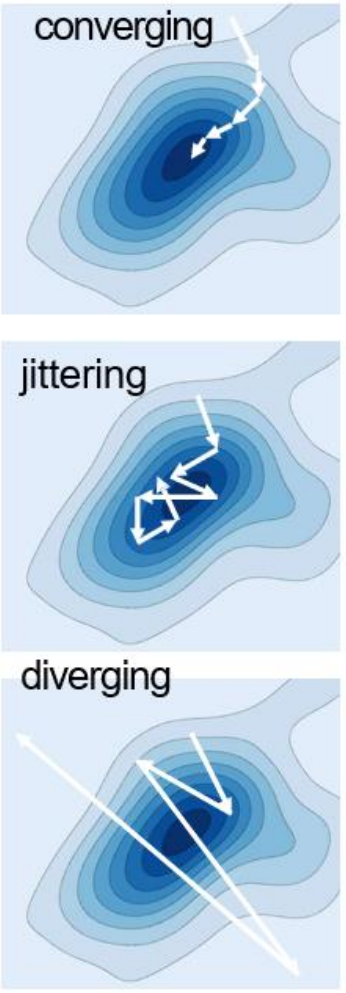
- Does it always converge?
- How long does it take?

- Hard to analyze for a neural network, but we can look at the problem through the lens of convex optimization

Adapted from Bhiksha Raj

Convergence and convergence rate

- An iterative algorithm is said to *converge* to a solution if the value updates arrive at a fixed point
  - Where the gradient is 0 and further updates do not change the estimate
- The algorithm may not converge
  - It may jitter around the local minimum
  - It may even diverge
- Conditions for convergence?



Convergence and convergence rate

- Convergence rate: how fast iterations arrive at the solution
- Generally quantified as:

$$R = \frac{\left|f\left(x^{(k+1)}\right) - f(x^*)\right|}{\left|f\left(x^{(k)}\right) - f(x^*)\right|}$$

— $x^{(k+1)}$ is the k-th iteration

— x^* is the optimal value of x

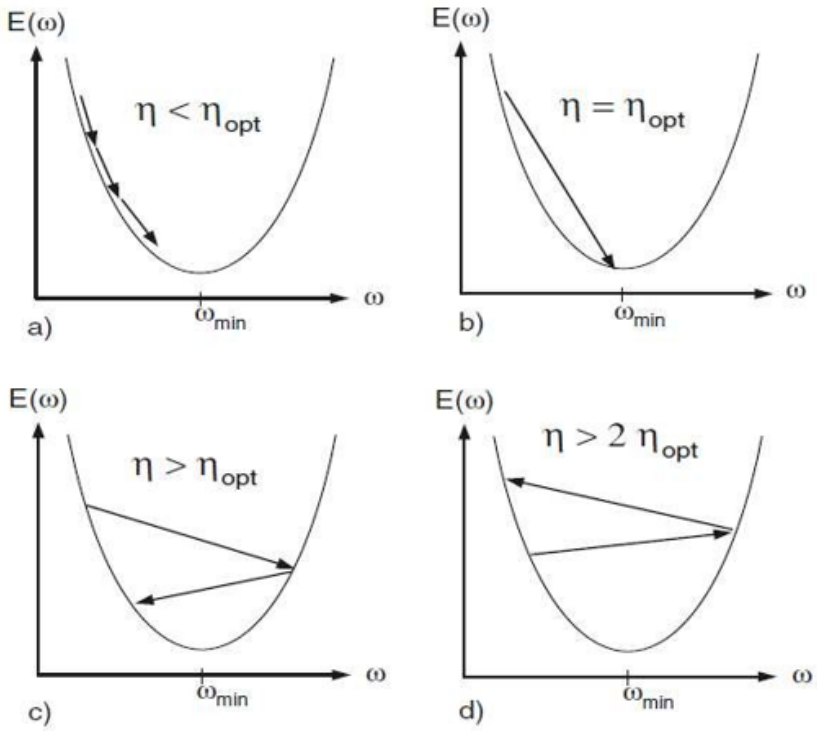
- If R is a constant (or upper-bounded): convergence is linear

Adapted from Bhiksha Raj

With non-optimal step size

$$w^{(k+1)} = w^{(k)} - \eta \frac{dE\left(w^{(k)}\right)}{dw}$$

Gradient descent with fixed step size η to estimate scalar parameter w



- For $\eta < \eta_{opt}$ the algorithm will converge monotonically
- For $2\eta_{opt} > \eta > \eta_{opt}$ we have oscillating convergence
- For $\eta > 2\eta_{opt}$ we get divergence

Optimization strategies

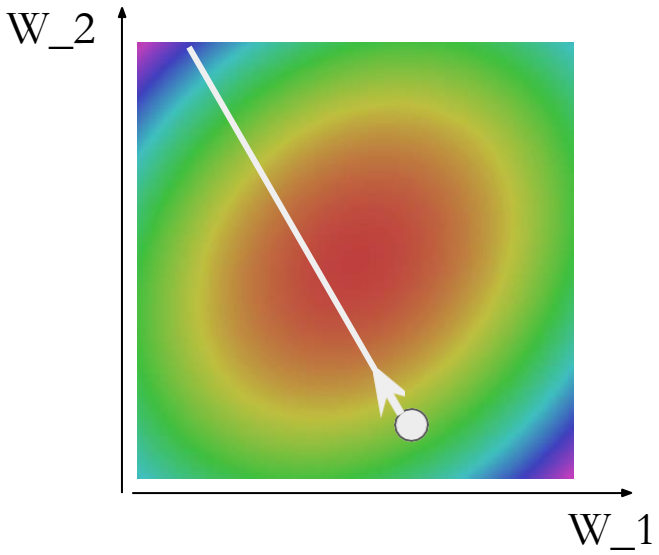
Getting to the minimum

- Gradient descent is just one strategy, but has several problems
- What other "steps" can we take?
- How far in the direction of decreasing gradient do we go? With what speed/acceleration?
- What about overshooting minima?

Optimization

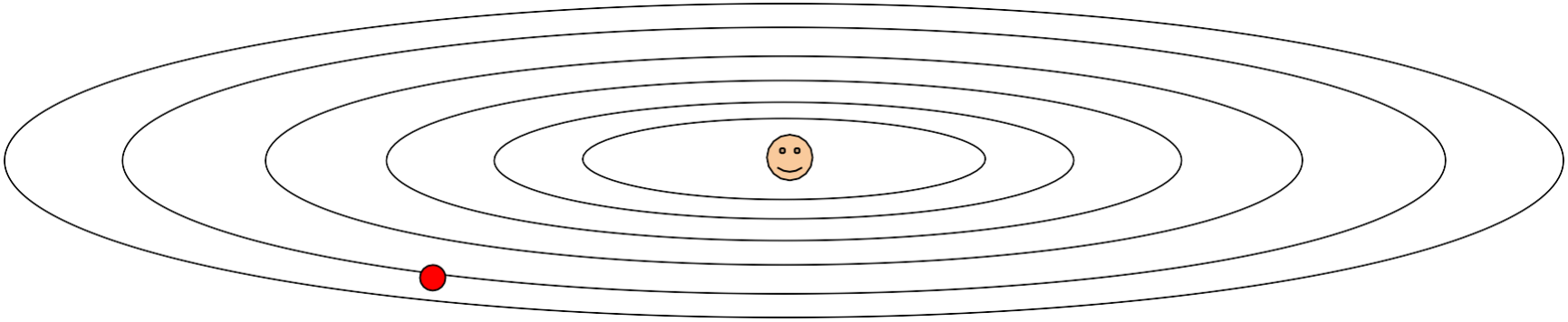
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do?

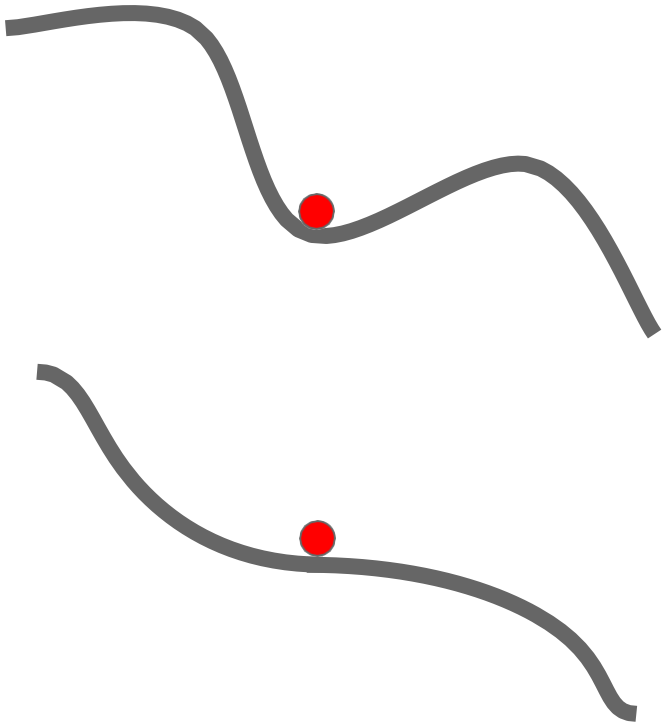


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck

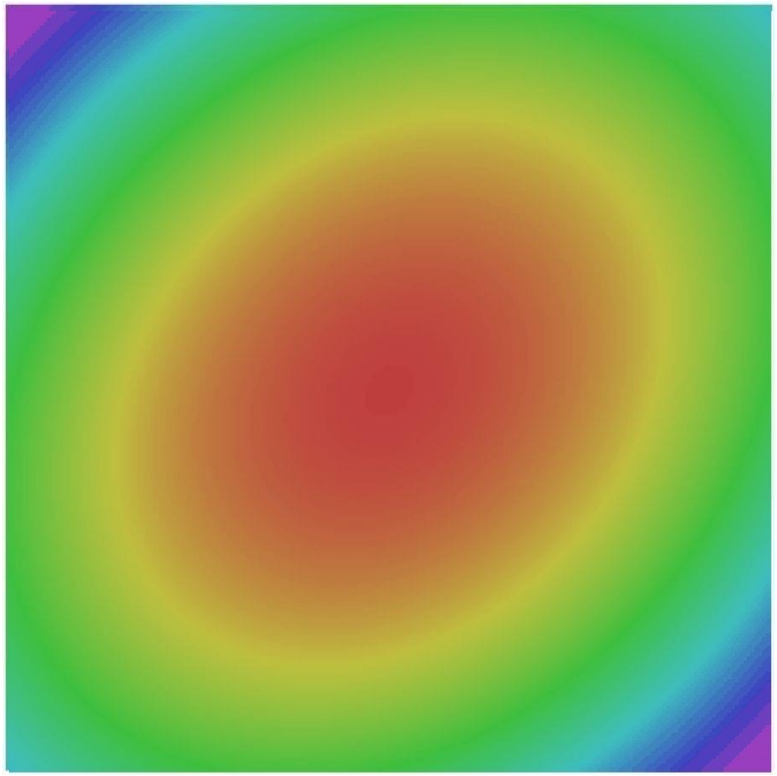


Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



SGD + Momentum

SGD

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

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

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

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

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

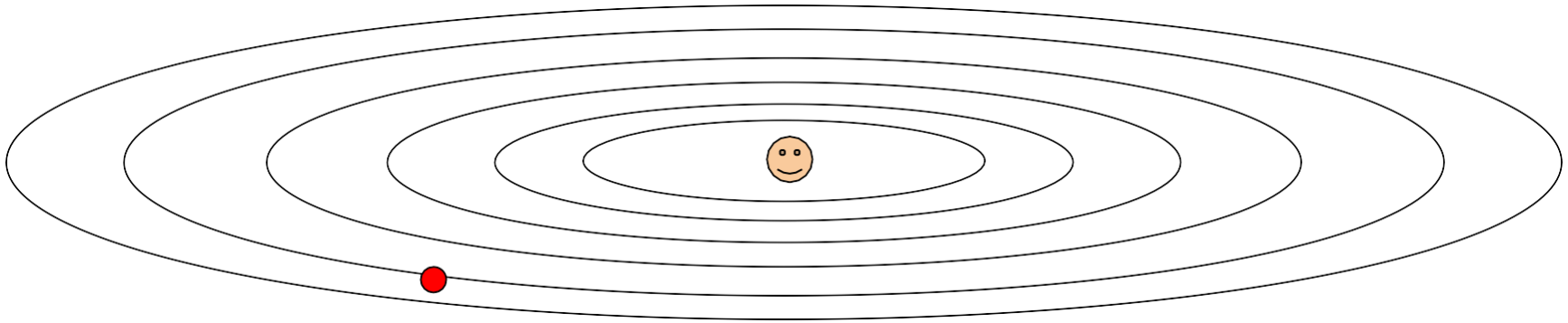
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates" or
"adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

AdaGrad

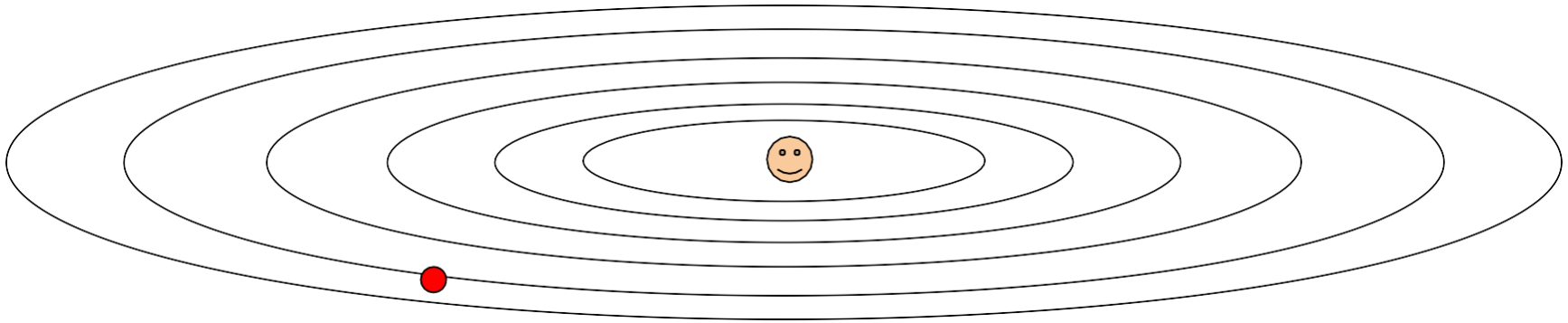
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

Progress along "steep" directions is damped;
progress along "flat" directions is accelerated

RMSProp (Root Mean Squared Propagation)

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

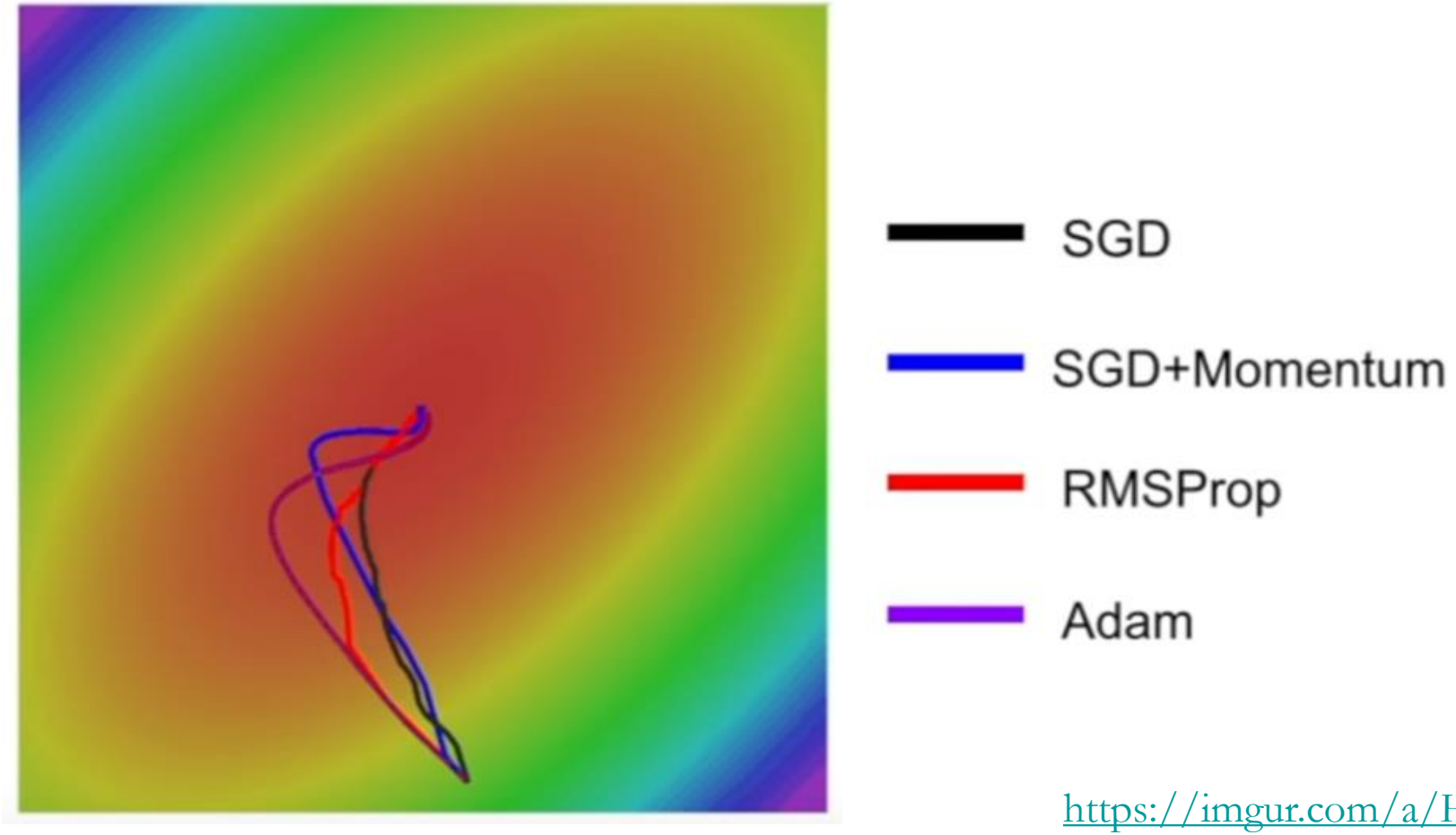
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Optimizers comparison



https://imgur.com/a/Hqolp

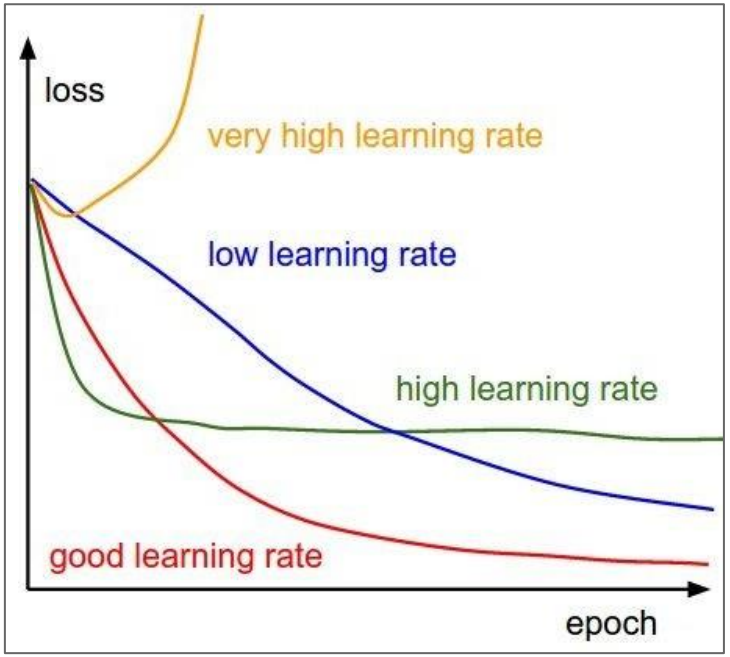
Visualization

https://www.deeplearning.ai/ai-notes/optimization/

(bottom of page)

"In this visualization, you can compare optimizers applied to different cost functions and initialization. For a given cost landscape (**1**) and initialization (**2**), you can choose optimizers, their learning rate and decay (**3**). Then, press the play button to see the optimization process (**4**). There's no explicit model, but you can assume that finding the cost function's minimum is equivalent to finding the best model for your task."

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

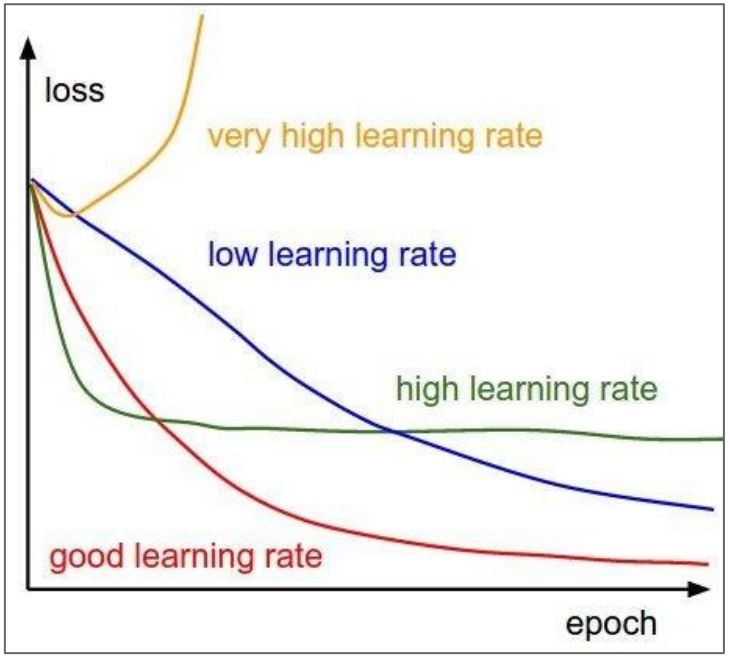
exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

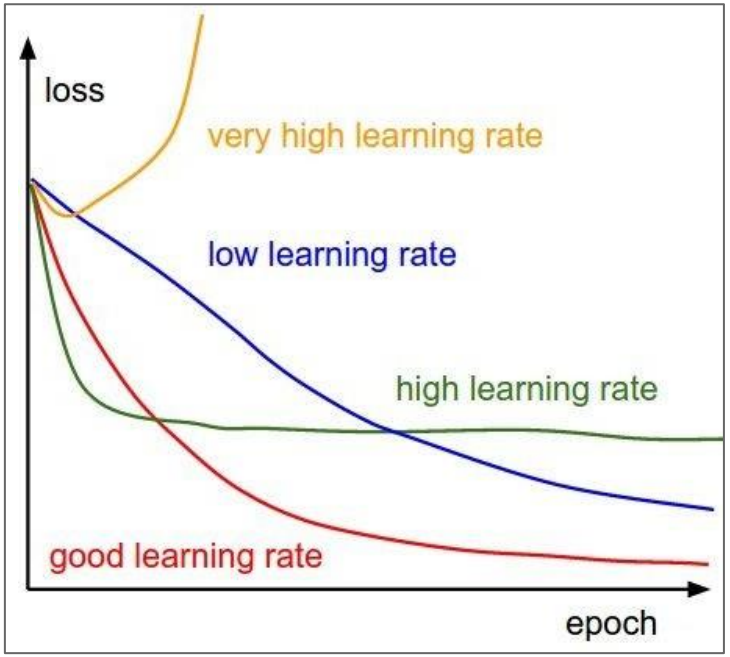
$$\alpha = \alpha_0 / (1 + kt)$$

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

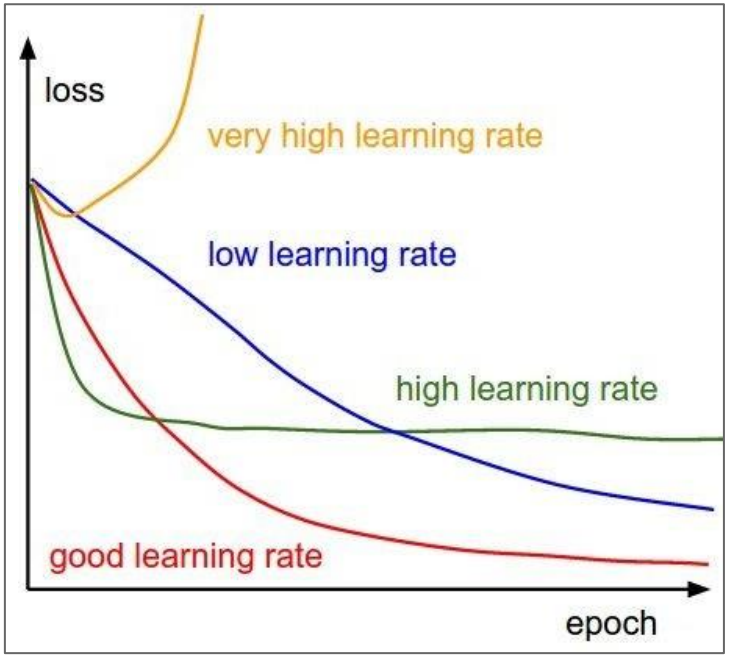
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

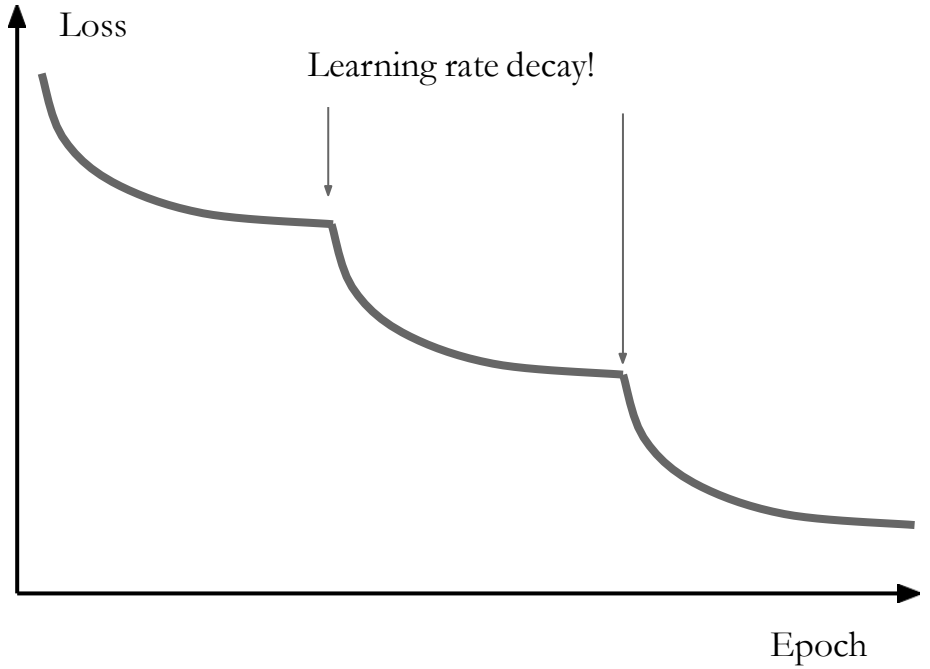


Q: Which one of these learning rates is best to use?

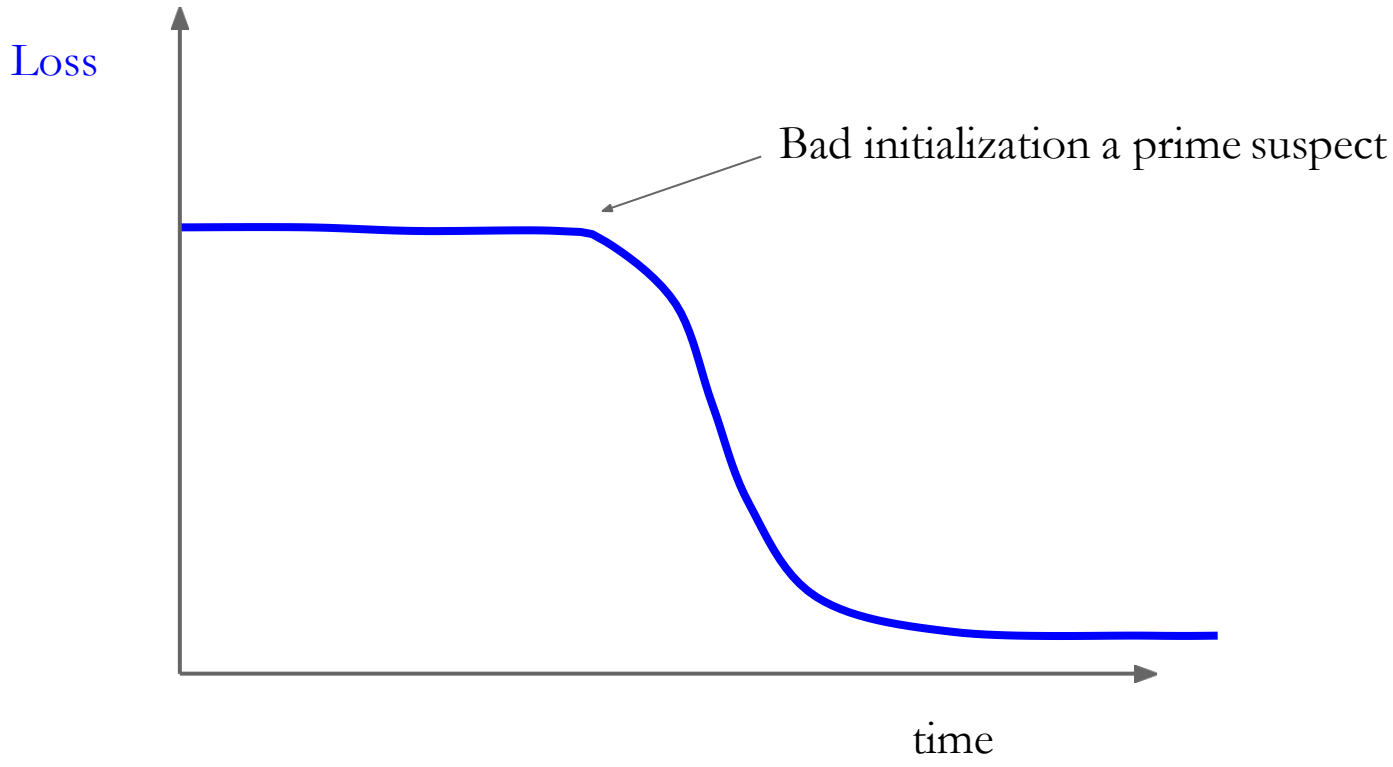
A: All of them! Start with large learning rate and decay over time

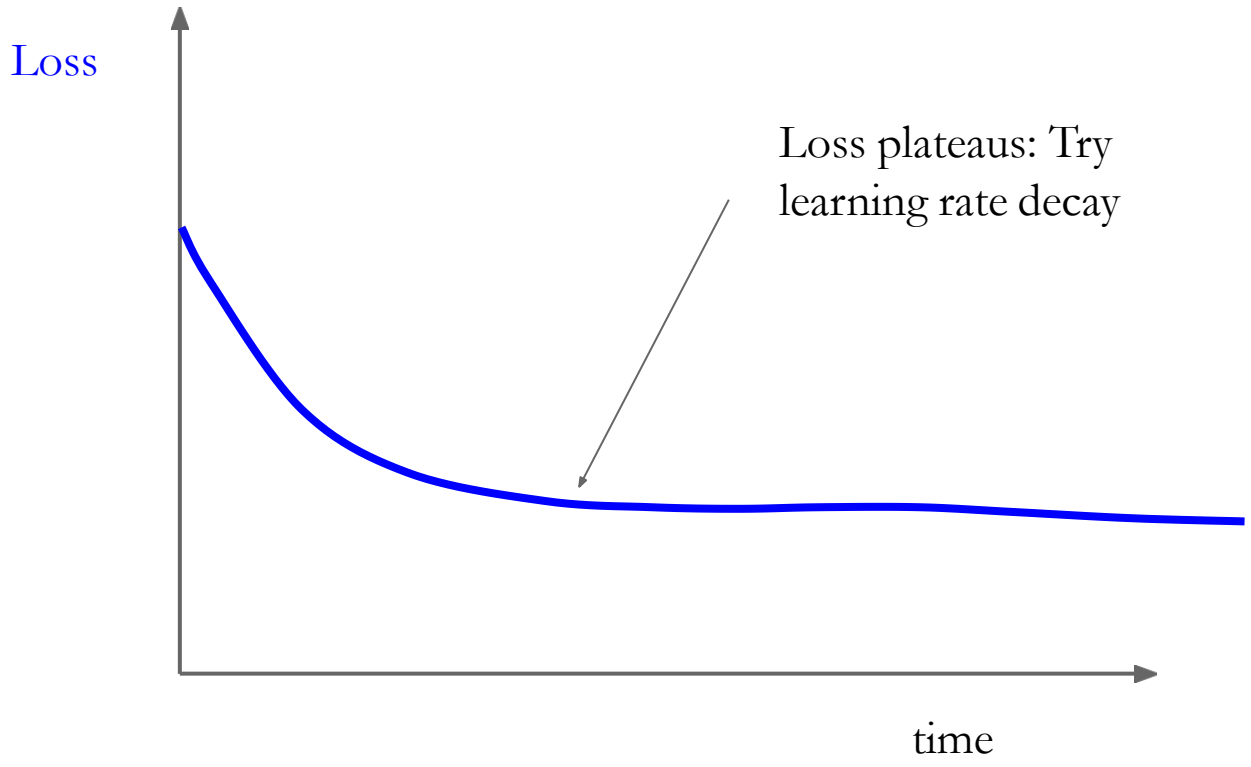
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

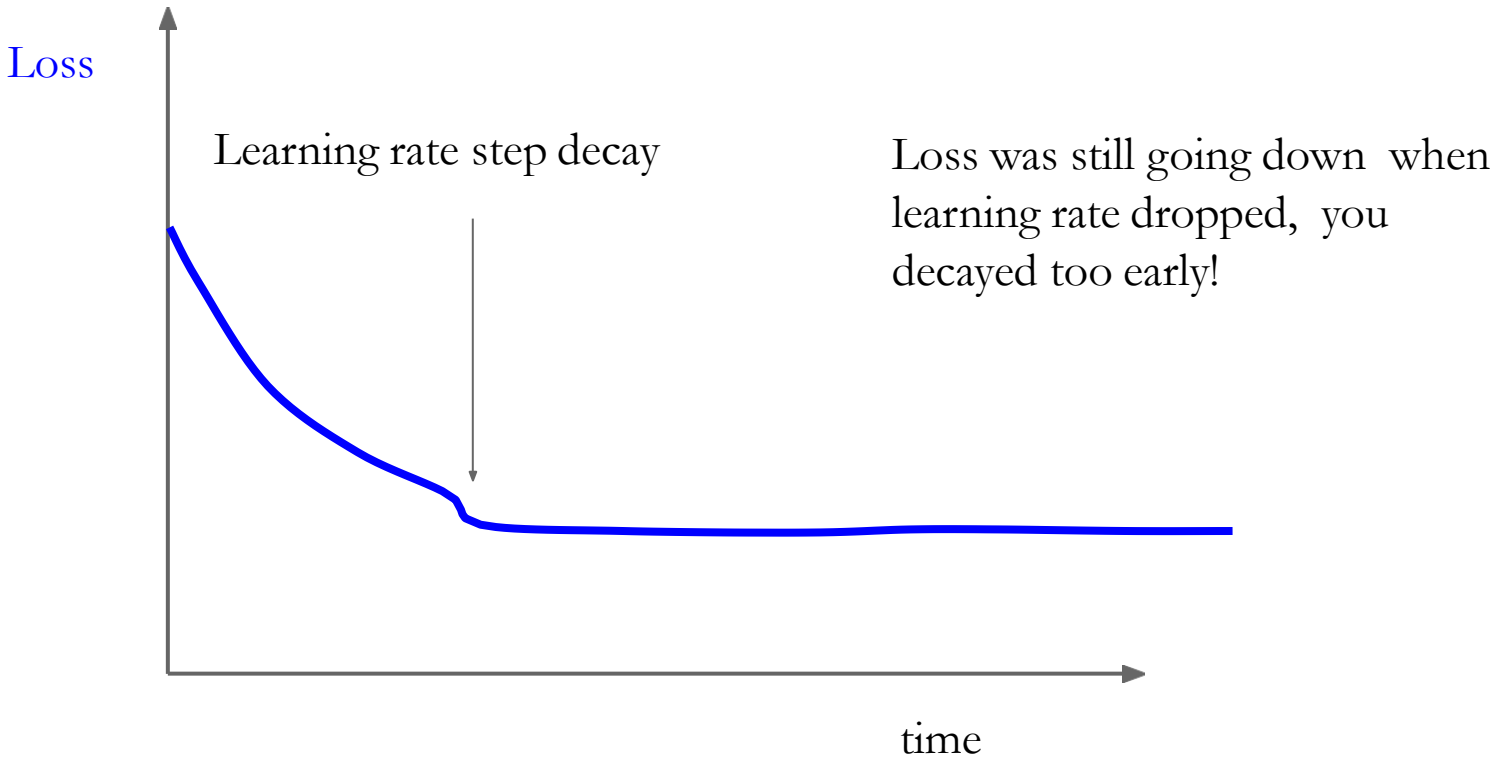




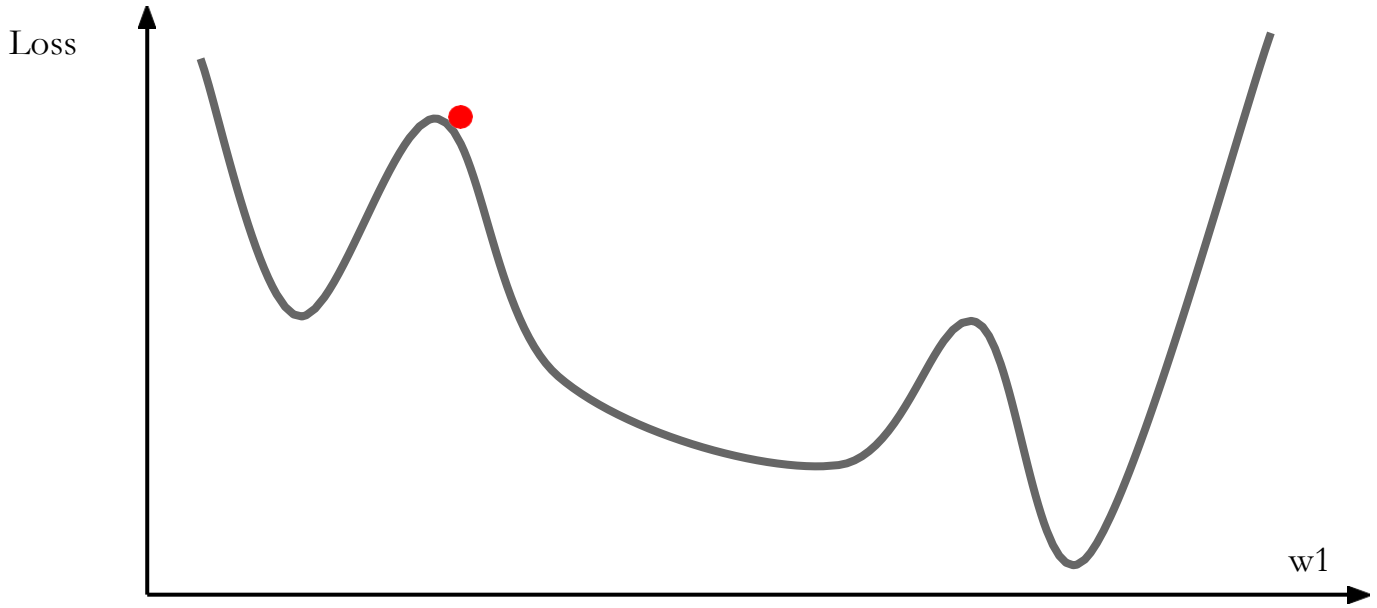
Also see https://openreview.net/pdf?id=r1eOnh4YPB


Loss
Bad initialization a prime suspect
time


Loss
Loss plateaus: Try learning rate decay
time

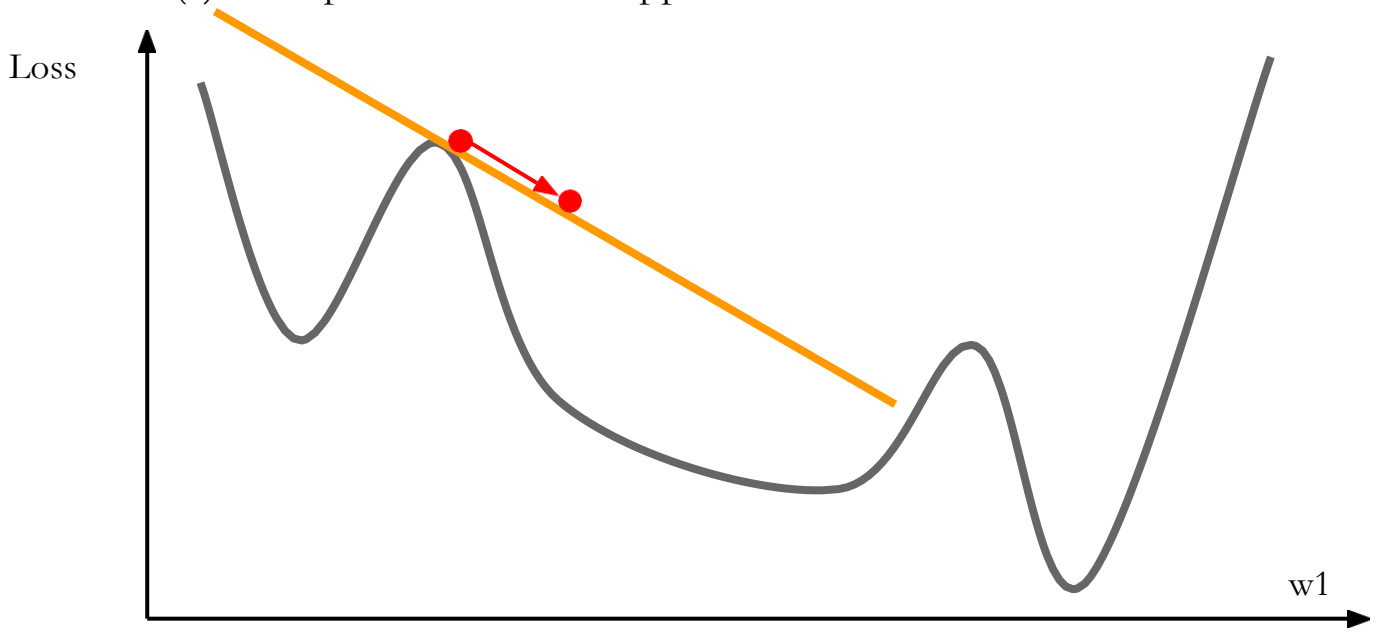

Loss
Learning rate step decay
Loss was still going down when learning rate dropped, you decayed too early!
time

First-Order Optimization



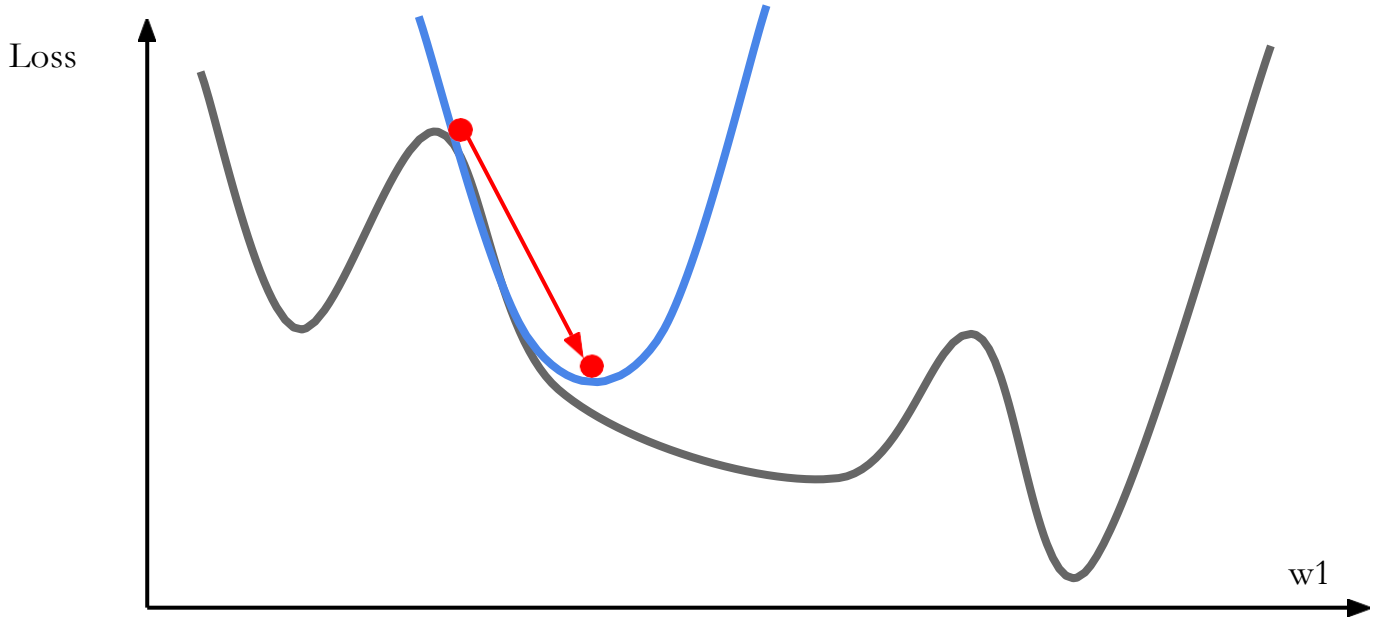
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Second-Order Optimization

$$H(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 \ell}{\partial w_1^2} & \frac{\partial^2 \ell}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 \ell}{\partial w_1 \partial w_n} \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial^2 \ell}{\partial w_n \partial w_1} & \cdots & \cdots & \frac{\partial^2 \ell}{\partial w_n^2} \end{pmatrix},$$

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q: Why is this bad for deep learning?

Partial solution: Quasi-Newton methods (e.g. **BGFS**) approximate inverse Hessian

Adapted from Fei-Fei Li, Justin Johnson, Serena Yeung http://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote07.html

Choosing Hyperparameters

Step 1: Check initial loss

Without weight decay (regularization), sanity check loss at initialization
e.g. log(C) for softmax with C classes

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization
Loss explodes to Inf or NaN? LR too high, bad initialization

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4
Good weight decay to try: 1e-4, 1e-5, 0

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid search, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer

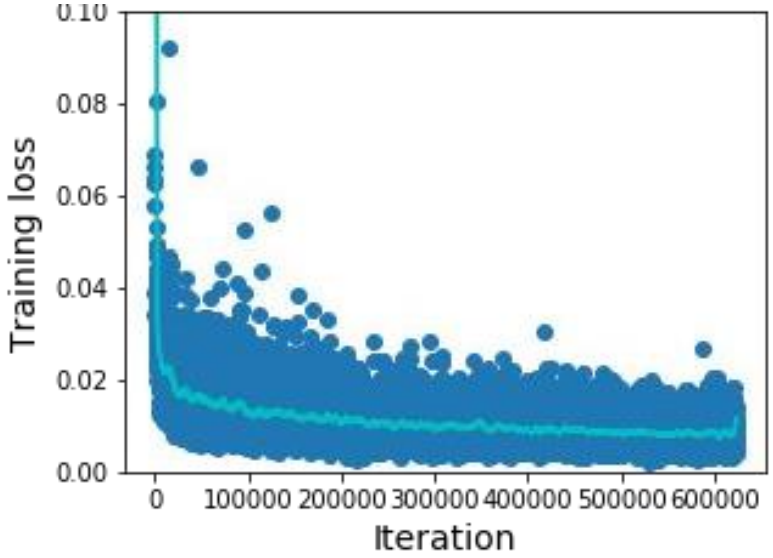
Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves

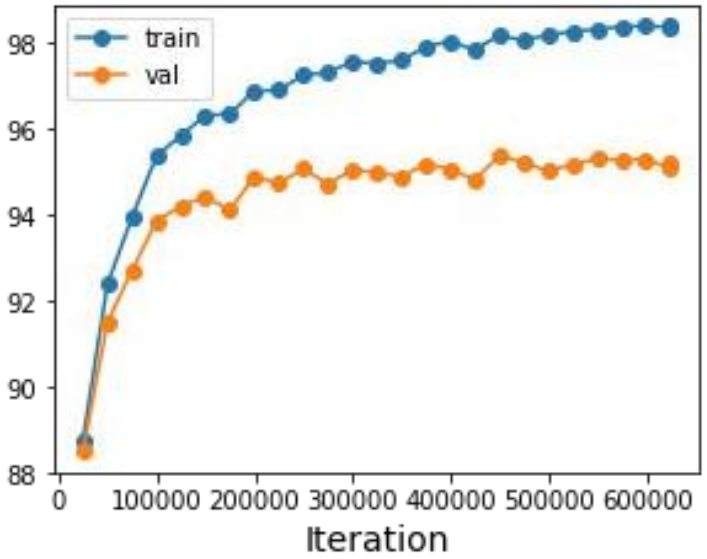
Look at learning curves!

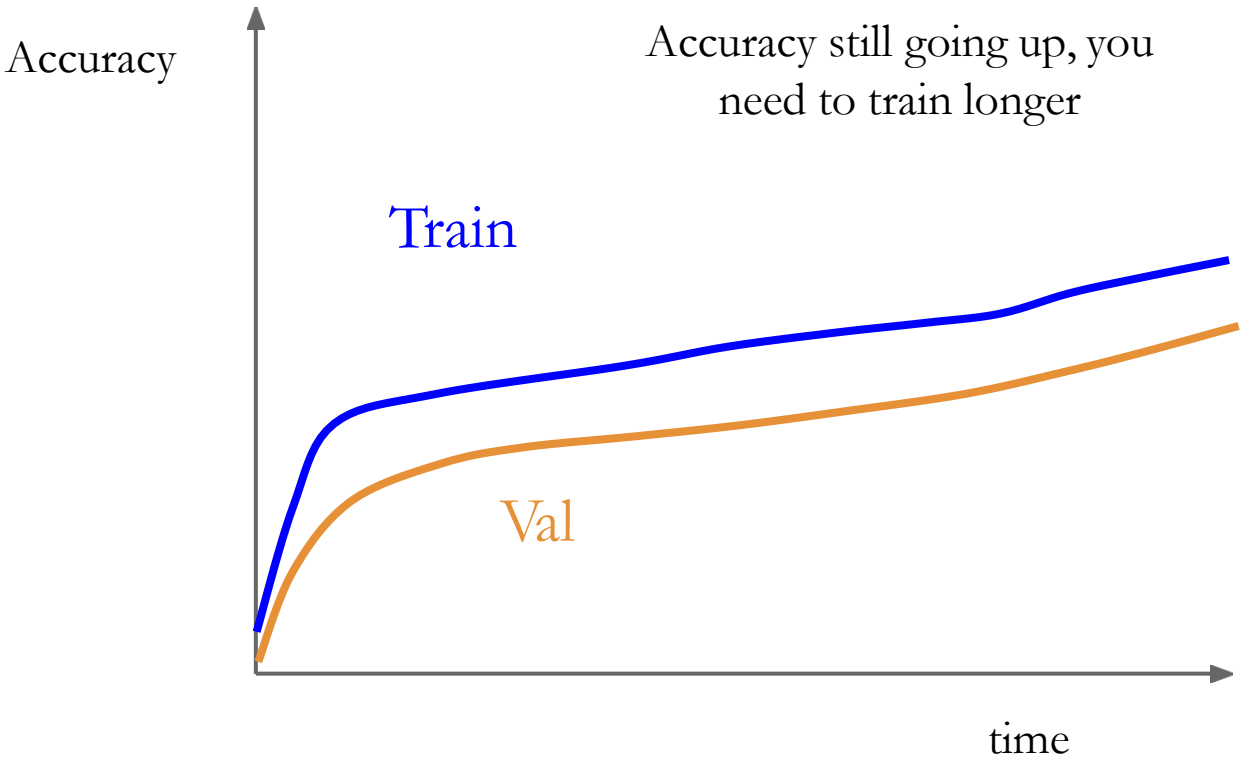
Training Loss

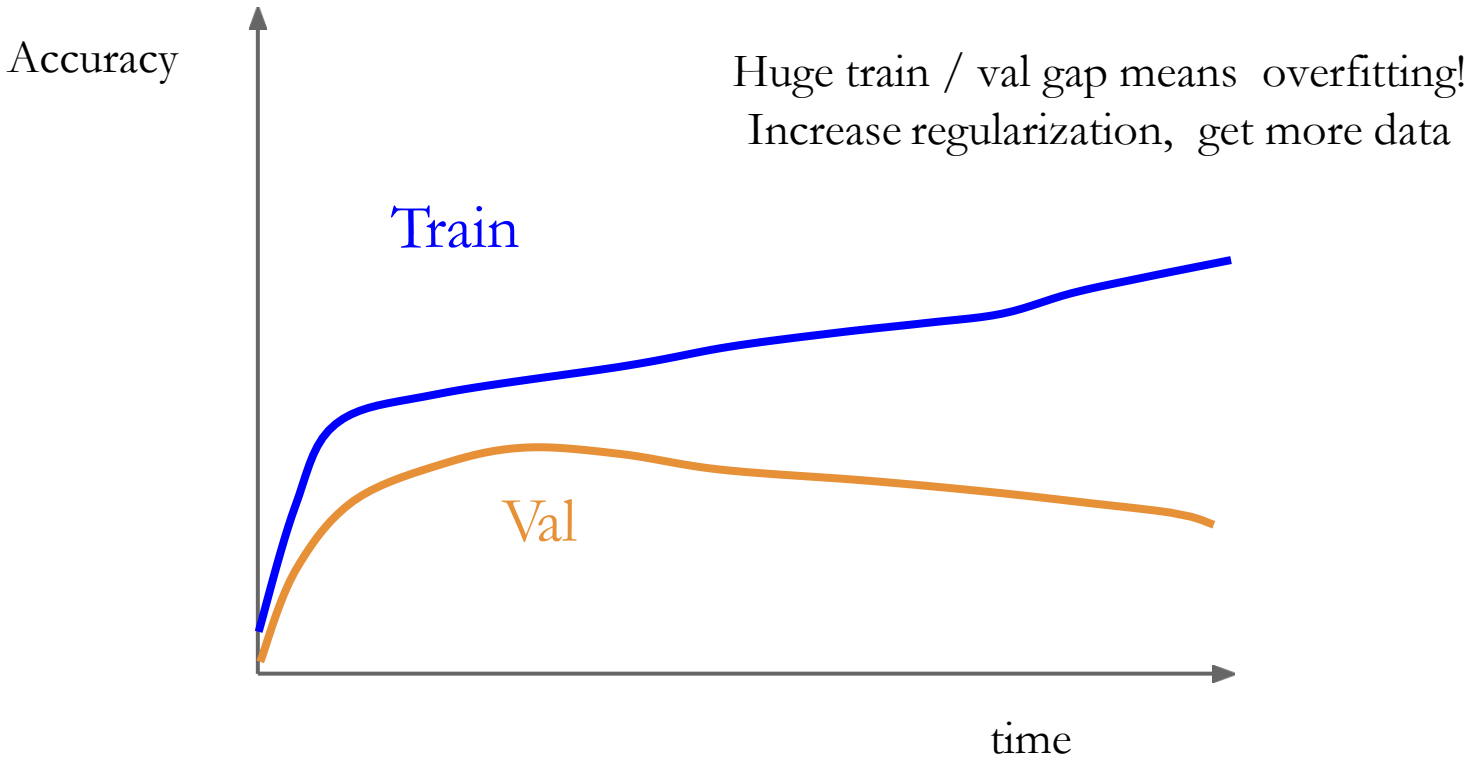


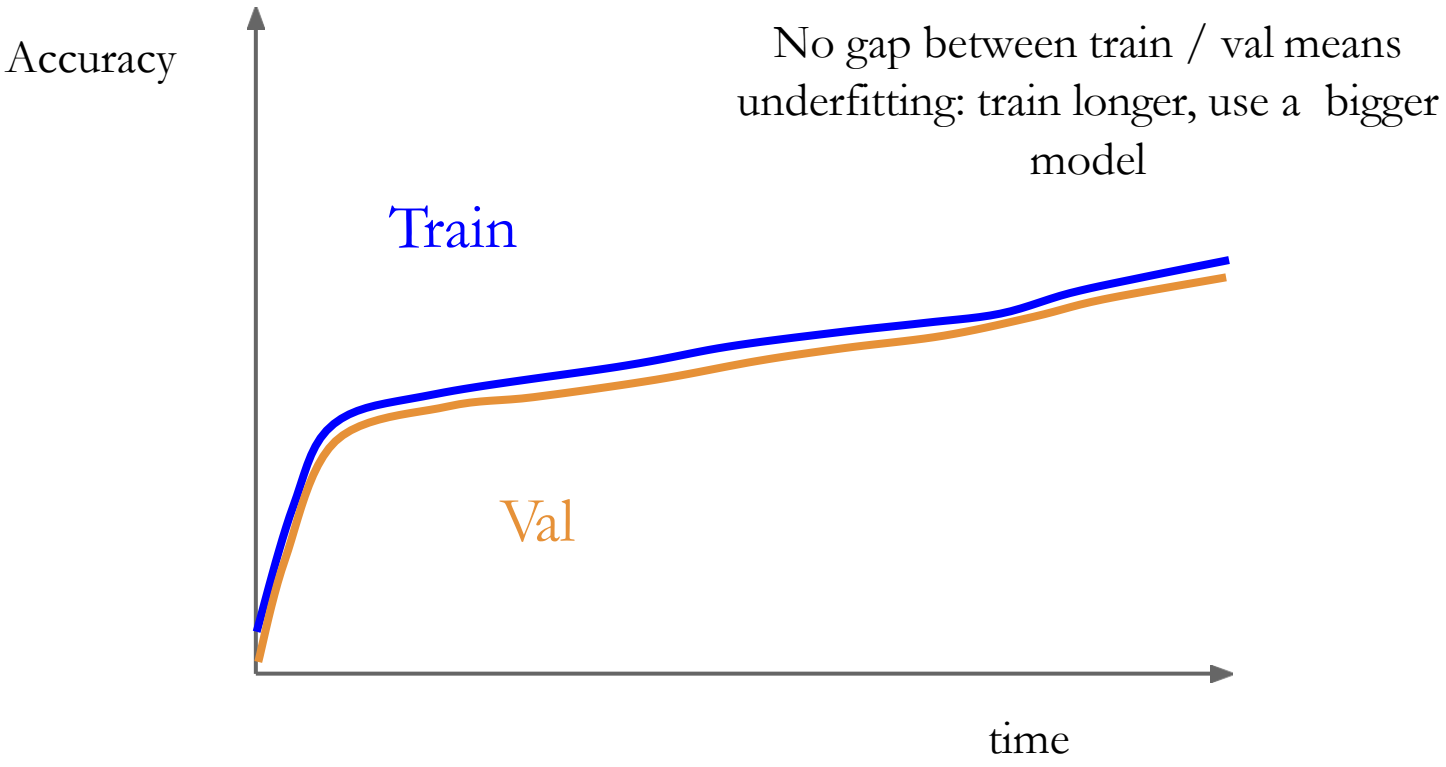
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

Train / Val Accuracy

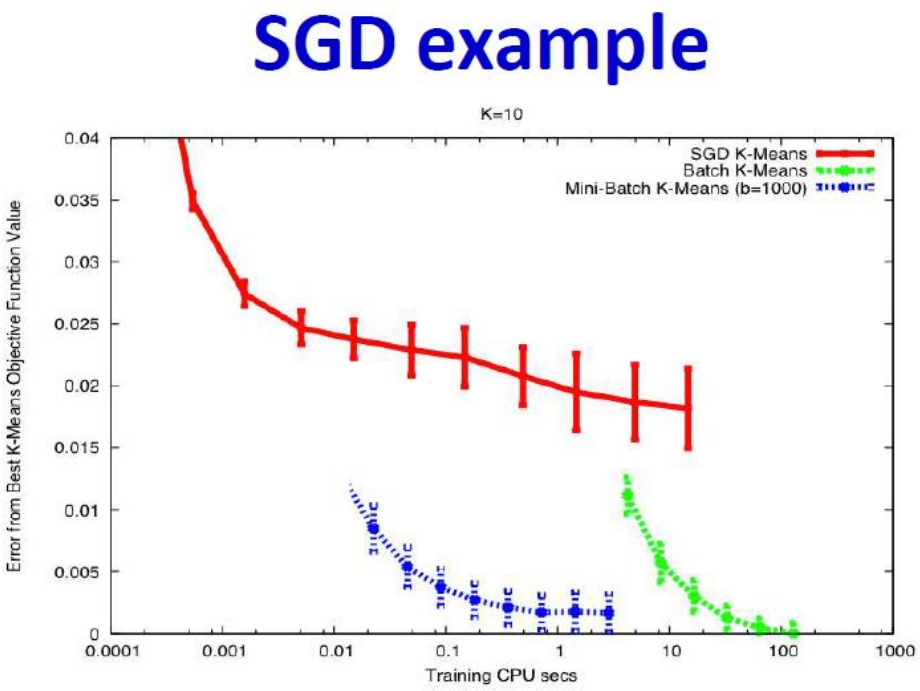



Accuracy
Accuracy still going up, you need to train longer
Train
Val
time


Accuracy
Huge train / val gap means overfitting!
Increase regularization, get more data
Train
Val
time


Accuracy
No gap between train / val means underfitting: train longer, use a bigger model
Train
Val
time

Mini batch size



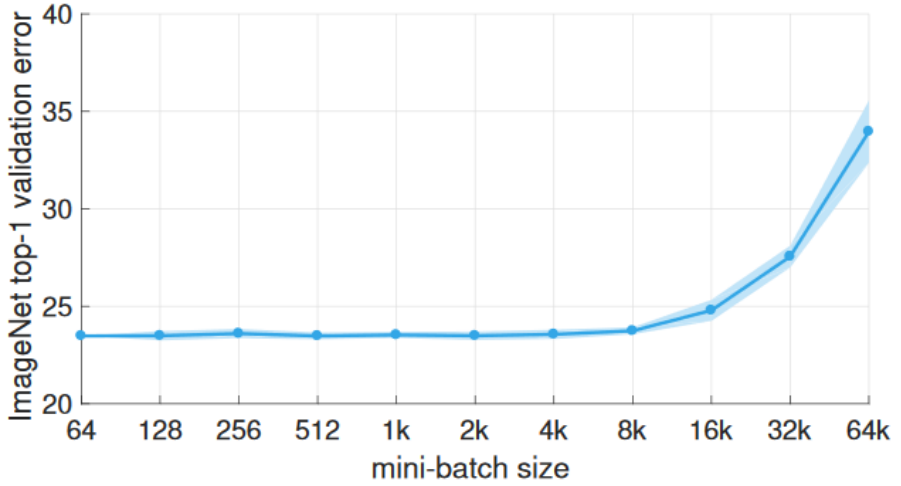


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch sizes we set the learning rate as a *linear* function of the minibatch size and apply a simple warmup phase for the first few epochs of training. All other hyper-parameters are kept fixed. Using this simple approach, accuracy of our models is invariant to minibatch size (up to an 8k minibatch size). Our techniques enable a linear reduction in training time with ~90% efficiency as we scale to large minibatch sizes, allowing us to train an accurate 8k minibatch ResNet-50 model in 1 hour on 256 GPUs.

Bhiksha Raj; Goyal et al. "Accurate, Large Minibatch SGD:Training ImageNet in 1 Hour"

Model Ensembles

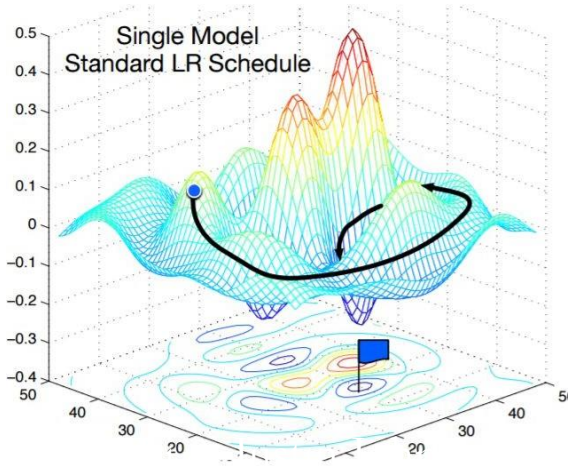
1. Train multiple independent models
2. At test time average their results

   (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

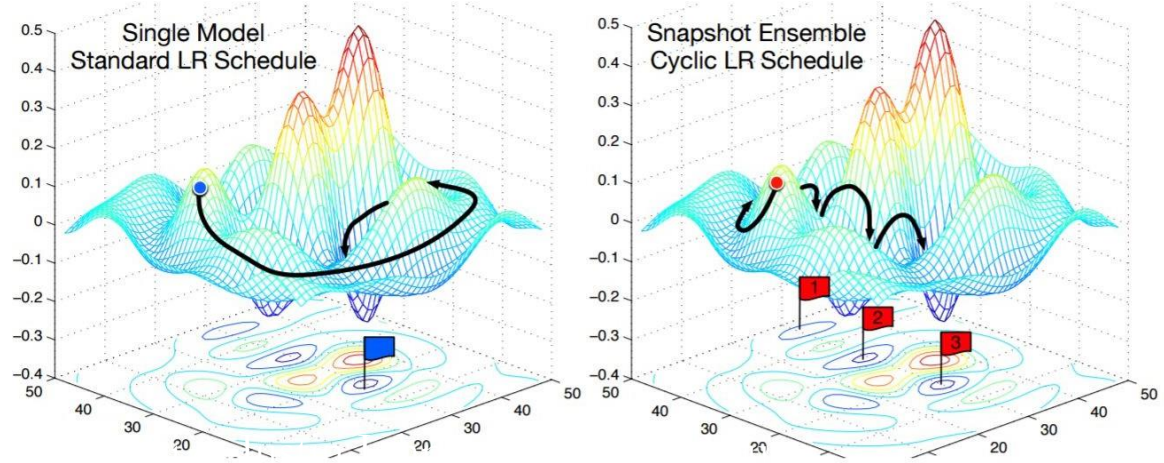
Instead of training independent models, use multiple snapshots of a single model during training!

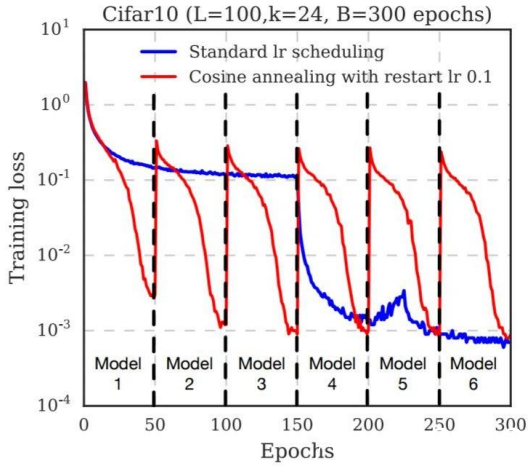


Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016 Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!





Cyclic learning rate schedules can make this work even better!

Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016 Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing hyperparameters
  - Model ensembles