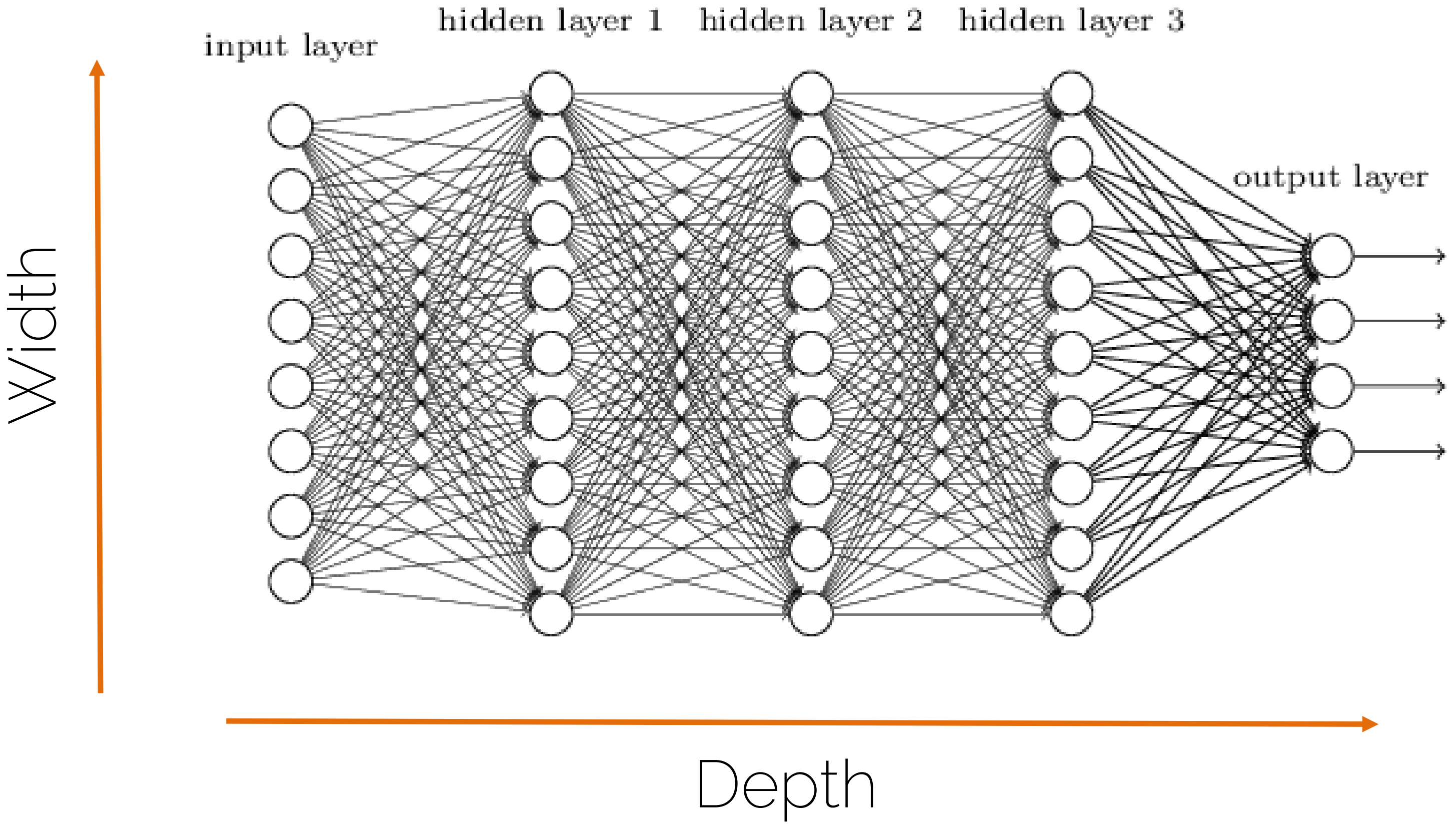
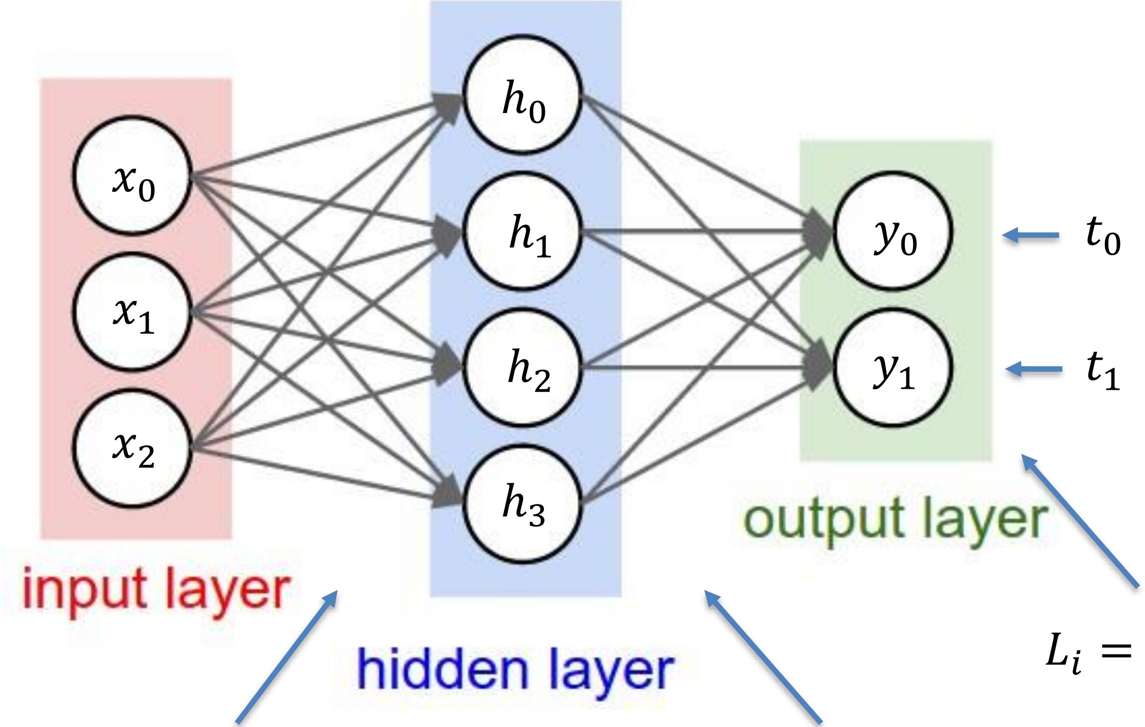
Lecture 5 recap

Neural Network



Gradient Descent for Neural Networks



$$L_i = (y_i - t_i)^2$$

$$\nabla_{w,b} f_{\{x,t\}}(w) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \cdots \\ \cdots \\ \frac{\partial f}{\partial w_{l,m,n}} \\ \cdots \\ \cdots \\ \frac{\partial f}{\partial b_{l,m}} \end{bmatrix}$$

$$h_j = A(b_{0,j} + \sum_k x_k w_{0,j,k})$$

$$y_i = A(b_{1,i} + \sum_j h_j w_{1,i,j})$$

Just simple: $A(x) = \max(0, x)$

Stochastic Gradient Descent (SGD)

$$\theta^{k+1} = \theta^{k} - \alpha \nabla_\theta L(\theta^{k}, x_{\{1..m\}}, y_{\{1..m\}})$$

$$\nabla_\theta L = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L_i$$

k now refers to k-th iteration

m training samples in the current batch

Gradient for the k-th batch

Note the terminology: iteration vs epoch

Gradient Descent with Momentum

$$v^{k+1} = \beta \cdot v^k + \nabla_\theta L(\theta^k)$$

accumulation rate
('friction', momentum)

velocity

Gradient of current minibatch

$$\theta^{k+1} = \theta^k - \alpha \cdot v^{k+1}$$

model

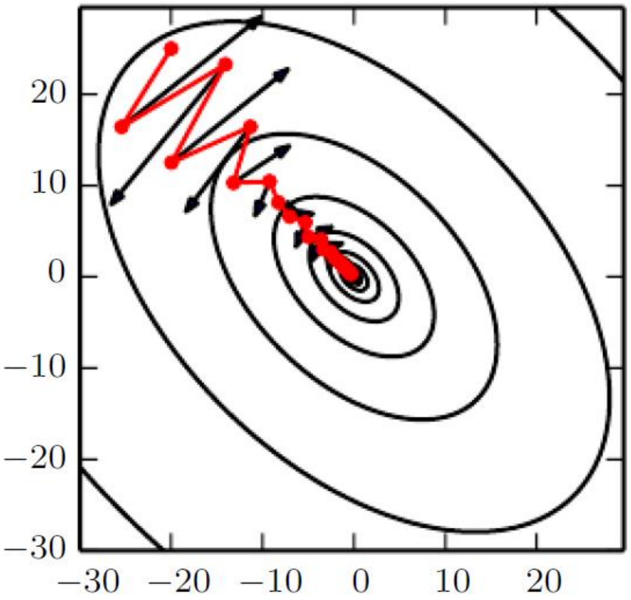
learning rate

velocity

Exponentially-weighted average of gradient

Important: velocity $\boldsymbol{v}^k$ is vector-valued!

Gradient Descent with Momentum



Step will be largest when a sequence of gradients all point to the same direction

Hyperparameters are α, β
β is often set to 0.9

$$\theta^{k+1} = \theta^k - \alpha \cdot v^{k+1}$$

Fig. credit: I. Goodfellow

RMSProp

$$s^{k+1} = \beta \cdot s^k + (1-\beta)\boxed{[\nabla_\theta L \circ \nabla_\theta L]}$$

Element-wise multiplication

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \epsilon}$$

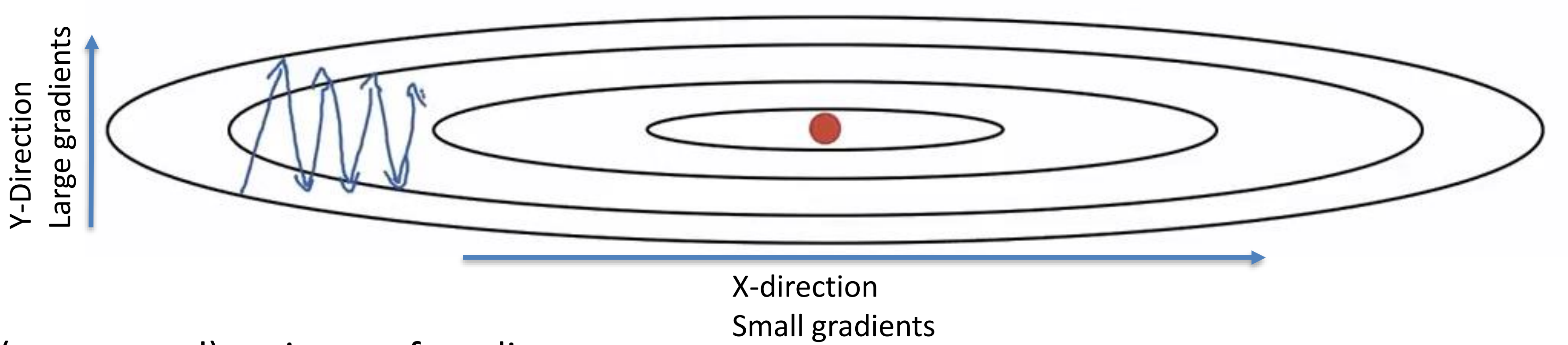
Hyperparameters: α, β, ϵ

Needs tuning!

Often 0.9

Typically 10^{-8}

RMSProp



(uncentered) variance of gradients
-> second momentum

$$\boxed{s^{k+1} = \beta \cdot s^{k} + (1-\beta)[\nabla_\theta L \circ \nabla_\theta L]}$$

$$\theta^{k+1} = \theta^{k} - \alpha \cdot \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \epsilon}$$

We're dividing by square gradients:
- Division in Y-Direction will be large
- Division in X-Direction will be small

Can increase learning rate!

Fig. credit: A. Ng

Adaptive Moment Estimation (Adam)

Combines Momentum and RMSProp

$$\boxed{m^{k+1} = \beta_1 \cdot m^k + (1-\beta_1)\nabla_\theta L\left(\theta^k\right)}$$

First momentum: mean of gradients

$$\boxed{v^{k+1} = \beta_2 \cdot v^k + (1-\beta_2)[\nabla_\theta L\left(\theta^k\right) \circ \nabla_\theta L\left(\theta^k\right)]}$$

Second momentum: variance of gradients

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{m^{k+1}}{\sqrt{v^{k+1}}+\epsilon}$$

Adam

Combines Momentum and RMSProp

$$m^{k+1} = \beta_1 \cdot m^k + (1 - \beta_1)\nabla_\theta L\left(\theta^k\right)$$

$$v^{k+1} = \beta_2 \cdot v^k + (1 - \beta_2)[\nabla_\theta L\left(\theta^k\right) \circ \nabla_\theta L\left(\theta^k\right)]$$

m^{k+1} and v^{k+1} are initialized with zero
-> bias towards zero

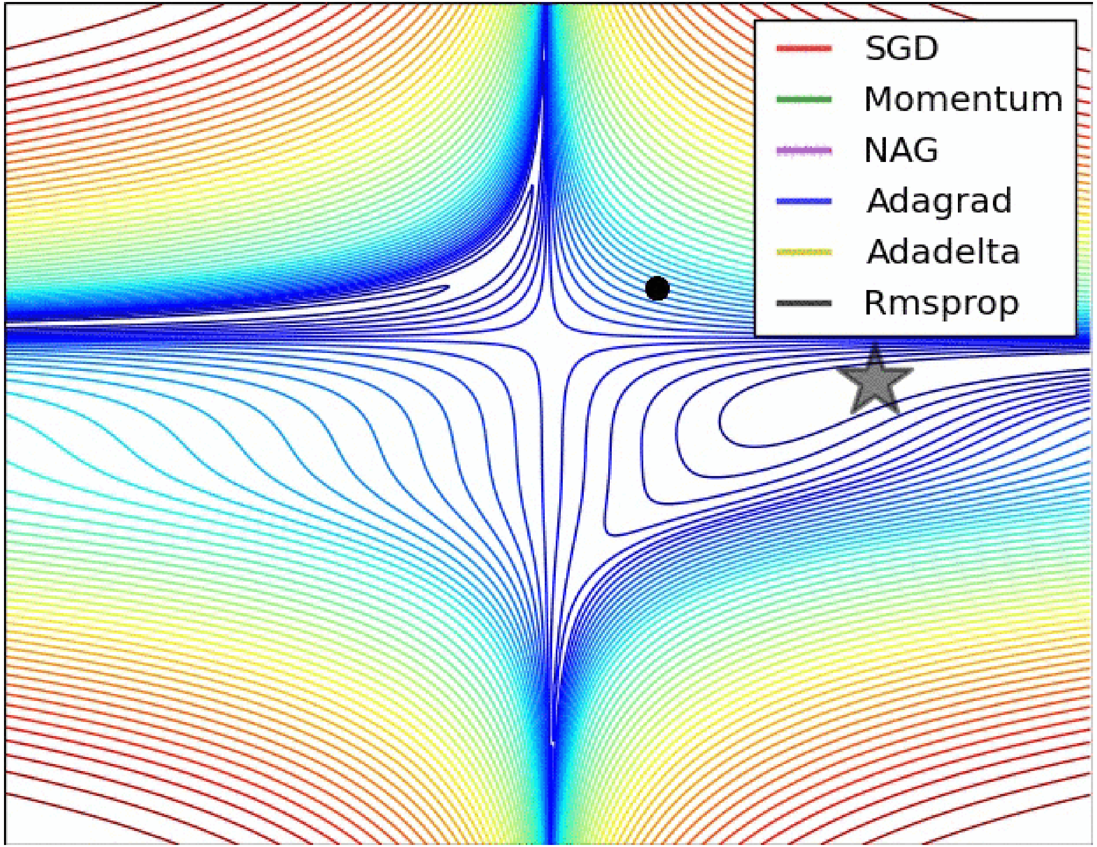
Typically, bias-corrected moment updates

$$\hat{m}^{k+1} = \frac{m^k}{1 - \beta_1}$$

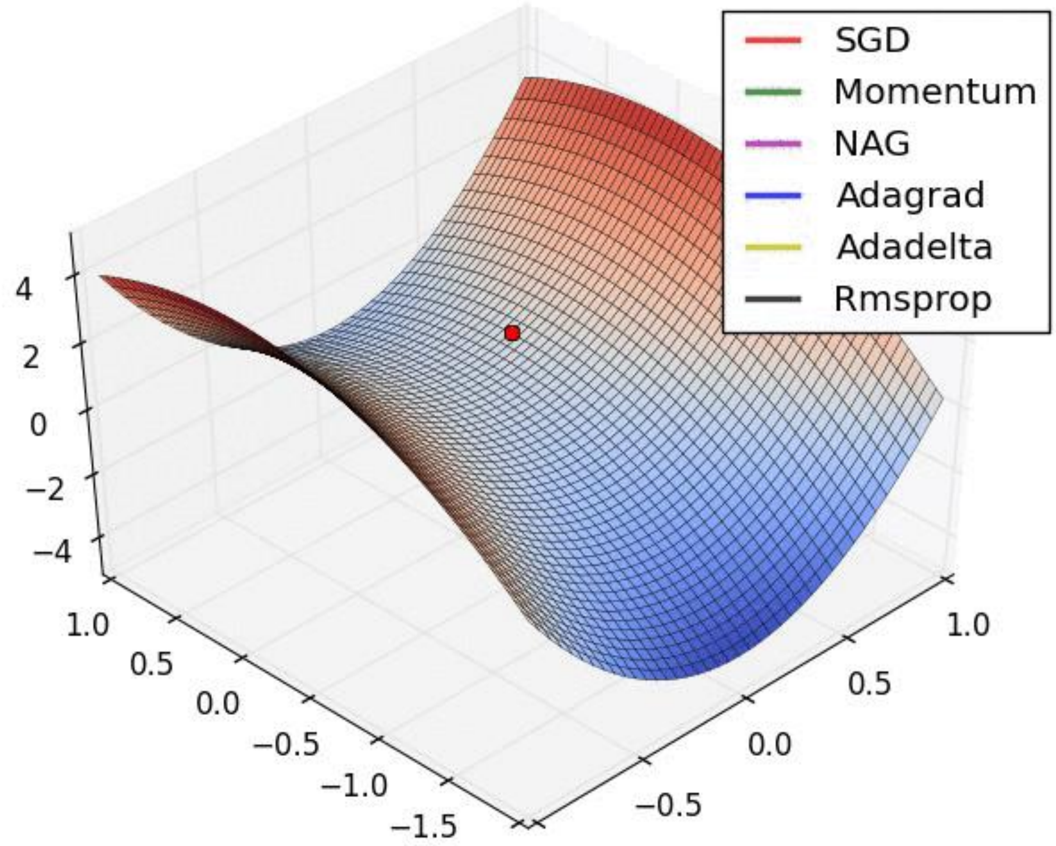
$$\hat{v}^{k+1} = \frac{v^k}{1 - \beta_2}$$

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \epsilon}$$

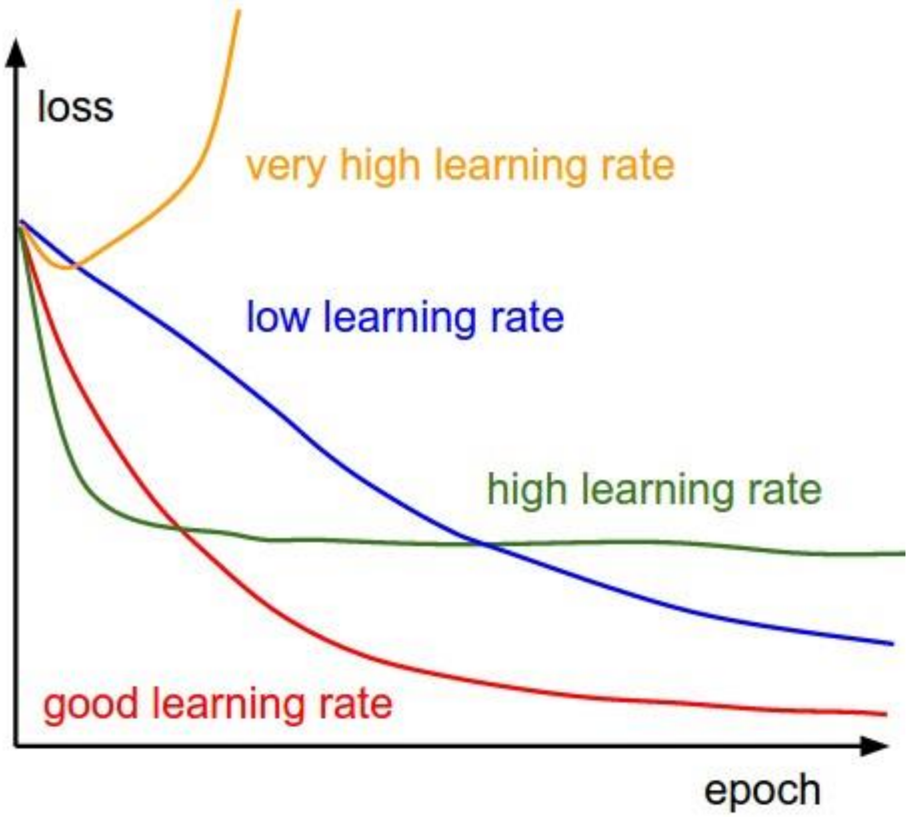
Convergence



Convergence



Importance of Learning Rate



Jacobian and Hessian

- Derivative $\mathbf{f}:\mathbb{R}\rightarrow\mathbb{R}$ $\quad \dfrac{df(x)}{dx}$

- Gradient $\mathbf{f}:\mathbb{R}^m\rightarrow\mathbb{R}$ $\quad \nabla_{\mathbf{x}}f(\mathbf{x})$ $\quad \left(\dfrac{df(x)}{dx_1},\dfrac{df(x)}{dx_2}\right)$

- Jacobian $\mathbf{f}:\mathbb{R}^m\rightarrow\mathbb{R}^n$ $\quad \mathbf{J}\in\mathbb{R}^{n\times m}$

- Hessian $\mathbf{f}:\mathbb{R}^m\rightarrow\mathbb{R}$ $\quad \mathbf{H}\in\mathbb{R}^{m\times m}$

SECOND DERIVATIVE

Newton's method

- Approximate our function by a second-order Taylor series expansion

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

First derivative

Second derivative (curvature)

https://en.wikipedia.org/wiki/Taylor_series

Newton's method

- Differentiate and equate to zero

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

Update step

We got rid of the learning rate!

SGD $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k, \mathbf{x}^i, \mathbf{y}^i)$

Newton's method

- Differentiate and equate to zero

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

Update step

Parameters of a network (millions)	Number of elements in the Hessian	Computational complexity of 'inversion' per iteration
k	k^2	$\mathcal{O}(k^3)$

Newton's method

- SGD (green)
- Newton's method exploits the curvature to take a more direct route

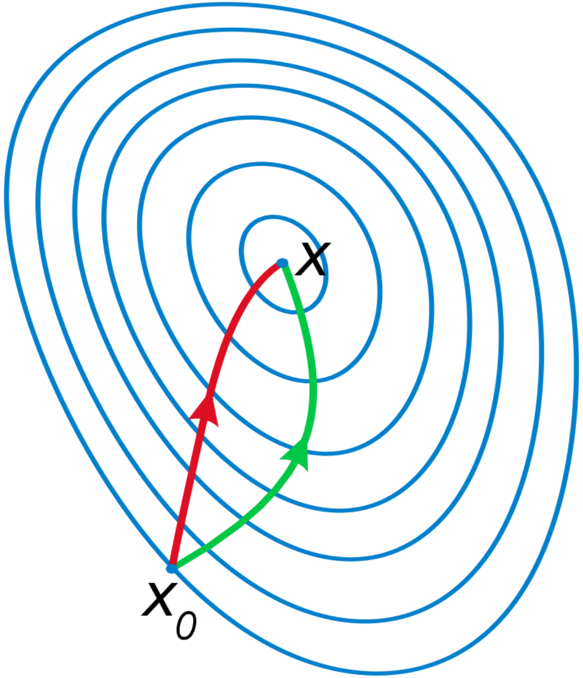


Image from Wikipedia

Newton's method

$$J(\boldsymbol{\theta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

Can you apply Newton's method for linear regression? What do you get as a result?

BFGS and L-BFGS

- Broyden-Fletcher-Goldfarb-Shanno algorithm
- Belongs to the family of quasi-Newton methods
- Have an approximation of the inverse of the Hessian

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

- BFGS $\mathcal{O}(n^2)$
- Limited memory: L-BFGS $\mathcal{O}(n)$

Gauss-Newton

- $x_{k+1} = x_k - H_f(x_k)^{-1} \nabla f(x_k)$
  - 'true' 2nd derivatives are often hard to obtain (e.g., numerics)
  - $H_f \approx 2 J_F^T J_F$
- Gauss-Newton (GN):

$$x_{k+1} = x_k - [2 J_F(x_k)^T J_F(x_k)]^{-1} \nabla f(x_k)$$

- Solve linear system (again, inverting a matrix is unstable):

$$2\big(J_F(x_k)^T J_F(x_k)\big)\underbrace{(x_k - x_{k+1})}_{\text{Solve for delta vector}} = \nabla f(x_k)$$

Levenberg

- Levenberg
  - "damped" version of Gauss-Newton:
  - $(J_F(x_k)^T J_F(x_k) + \lambda \cdot I) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$

Tikhonov regularization

  - The damping factor λ is adjusted in each iteration ensuring:
  - $f(x_k) > f(x_{k+1})$
    - if inequation is not fulfilled increase λ
    - →Trust region

- →"Interpolation" between Gauss-Newton (small λ) and Gradient Descent (large λ)

Levenberg-Marquardt

- Levenberg-Marquardt (LM)

$$(J_F(x_k)^T J_F(x_k) + \lambda \cdot diag(J_F(x_k)^T J_F(x_k))) \cdot (x_k - x_{k+1})$$
$$= \nabla f(x_k)$$

  – Instead of a plain Gradient Descent for large λ, scale each component of the gradient according to the curvature.
    - Avoids slow convergence in components with a small gradient

Which, what and when?

- Standard: **Adam**

- Fallback option: **SGD with momentum**

- **Newton, L-BFGS, GN, LM** only if you can do full batch updates (doesn't work well for minibatches!!)

This practically never happens for DL
Theoretically, it would be nice though due to fast convergence

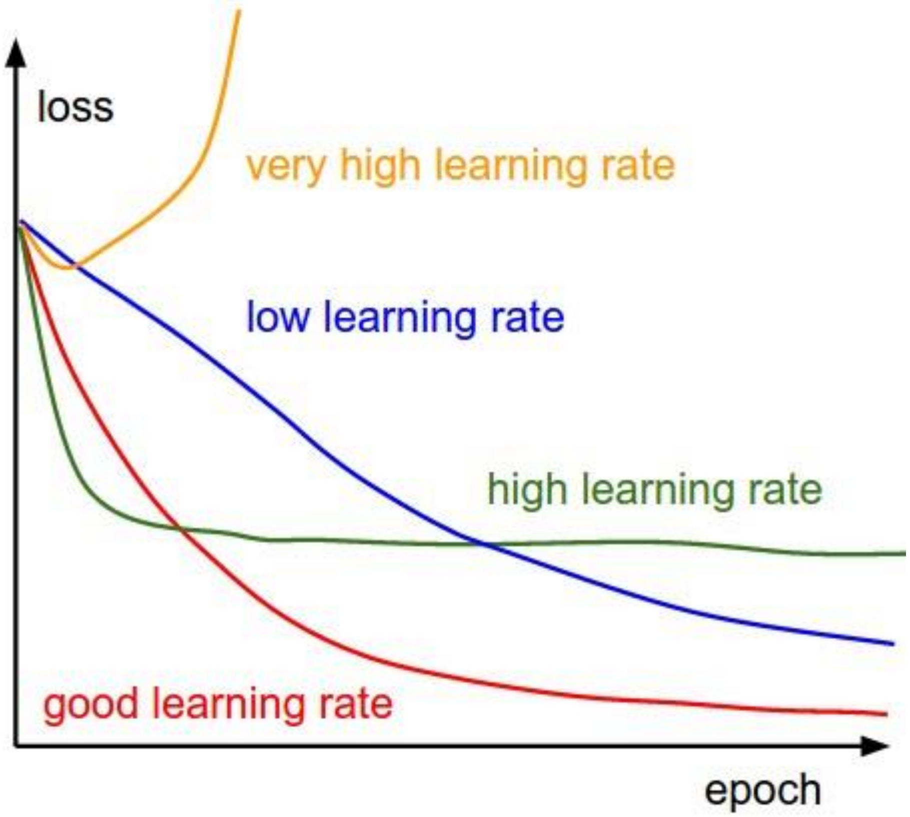
General Optimization

- Linear Systems (Ax = b)
  - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, etc.
- Non-linear (gradient-based)
  - Newton, Gauss-Newton, LM, (L)BFGS <- second order
  - Gradient Descent, SGD <- first order
- Others:
  - Genetic algorithms, MCMC, Metropolis-Hastings, etc.
  - Constrained and convex solvers (Langrage, ADMM, Primal-Dual, etc.)

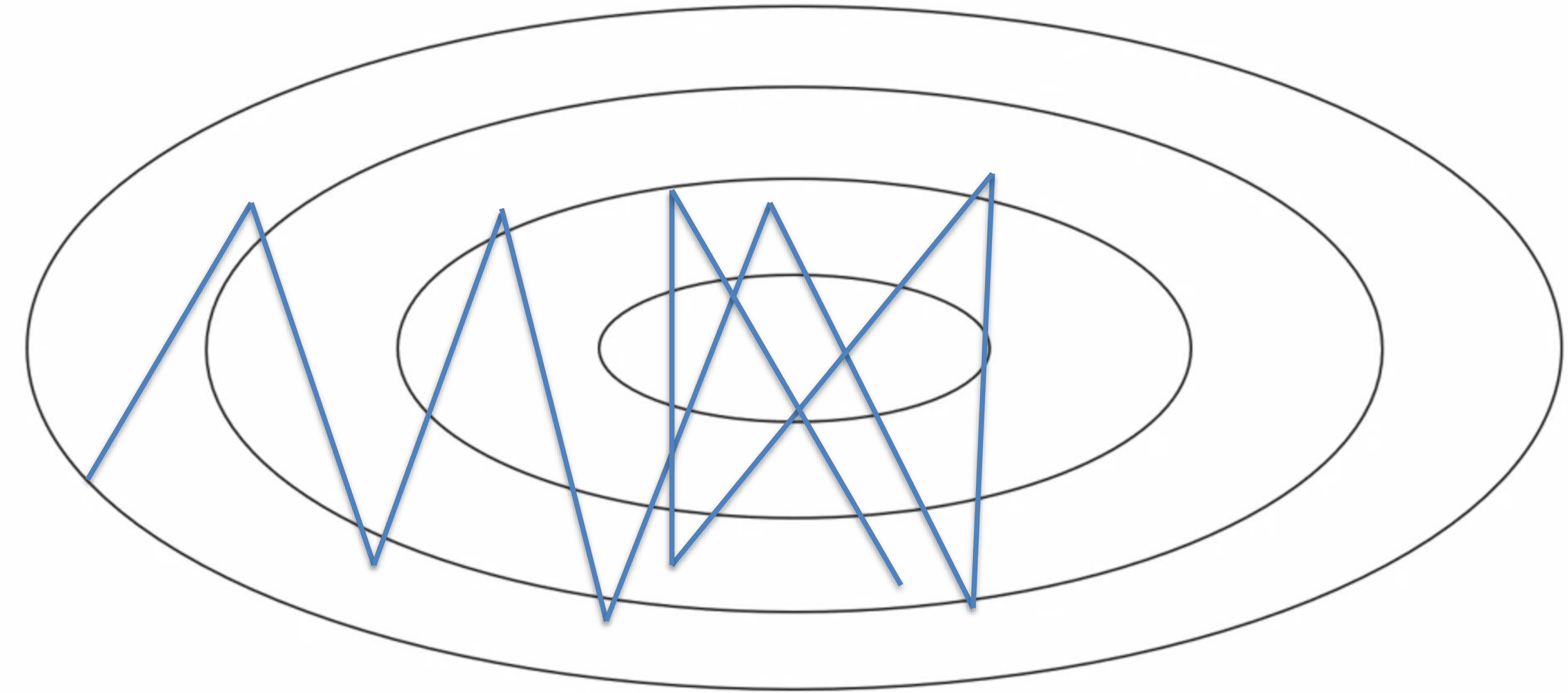
Please Remember!

- Think about your problem and optimization at hand
- SGD is specifically designed for minibatch
- When you can, use 2nd order method -> it's just faster
- GD or SGD is **not** a way to solve a linear system!

Importance of Learning Rate



Learning Rate



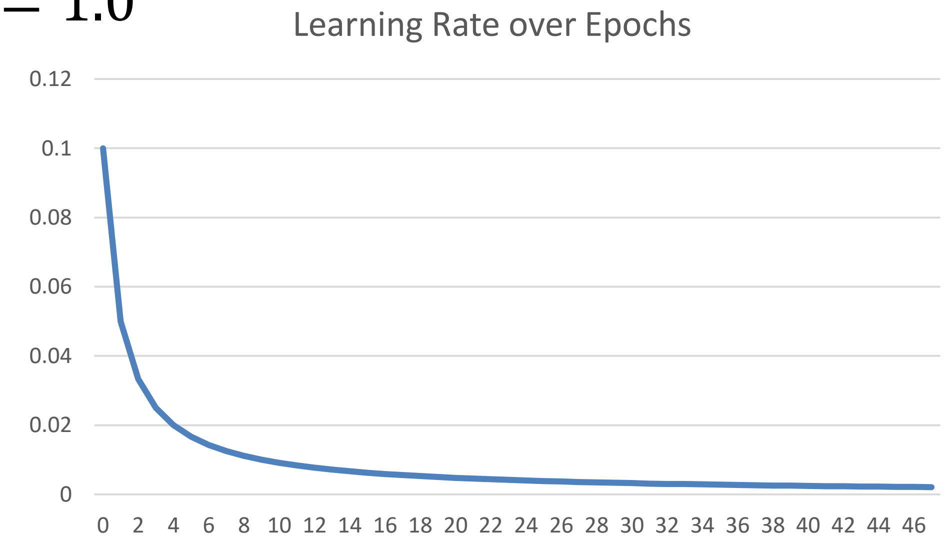
Need high learning rate when far away

Need low learning rate when close

Learning Rate Decay

- $\alpha = \frac{1}{1+decayrate \cdot epoch} \cdot \alpha_0$
  - E.g., $\alpha_0 = 0.1$, $decayrate = 1.0$
  - > Epoch 0: 0.1
  - > Epoch 1: 0.05
  - > Epoch 2: 0.033
  - > Epoch 3: 0.025

…



Learning Rate Decay

Many options:

- Step decay $\boldsymbol{\alpha = \alpha - t \cdot \alpha}$ (only every n steps)
  - T is decay rate (often 0.5)

- Exponential decay $\boldsymbol{\alpha = t^{epoch} \cdot \alpha_0}$
  - t is decay rate (t < 1.0)

- $\boldsymbol{\alpha = \frac{t}{\sqrt{epoch}} \cdot a_0}$
  - t is decay rate

- Etc.

Training Schedule

Manually specify learning rate for entire training process

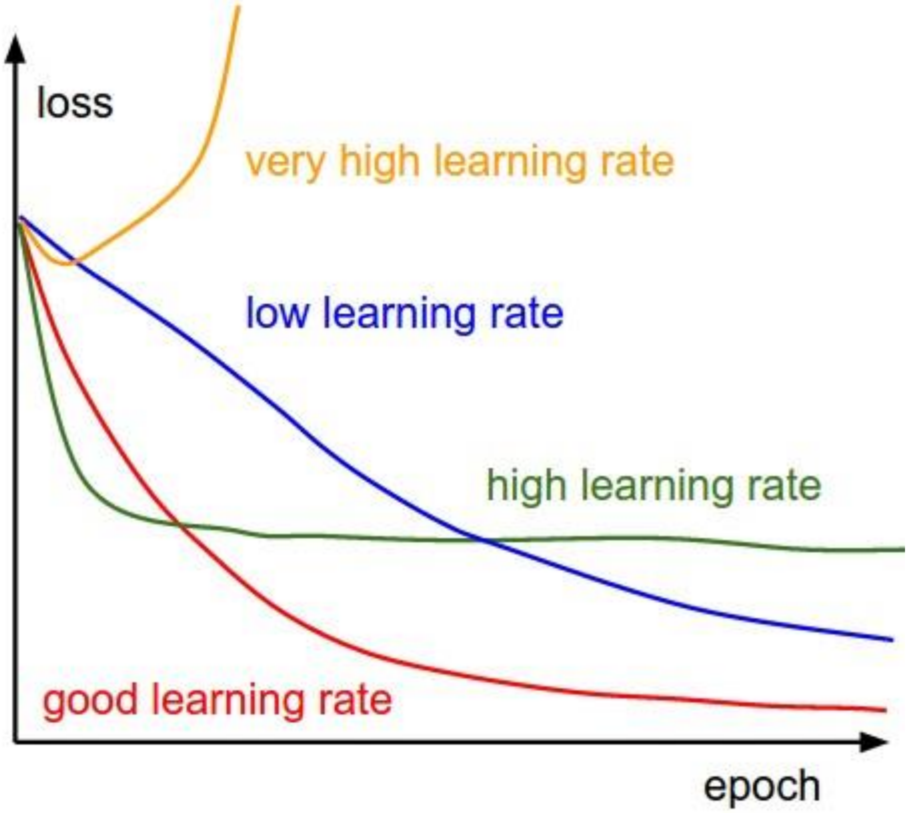
- Manually set learning rate every n-epochs
- How?
  - Trial and error (the hard way)
  - Some experience (only generalizes to some degree)

Consider: #epochs, training set size, network size, etc.

Learning Rate: Implications

- What if too high?

- What if too low?



Training

- Given ground dataset with ground lables
  - $\{x_i, y_i\}$
    - For instance $\boldsymbol{x_i}$-th training image, with label $\boldsymbol{y_i}$
    - Often $\mathbf{dim}(\boldsymbol{x}) \gg \mathbf{dim}(\boldsymbol{y})$ (e.g., for classification)
    - $\boldsymbol{i}$ is often in the 100-thousands or millions
  - Take network $\boldsymbol{f}$ and its parameters $\boldsymbol{w}, \boldsymbol{b}$

  - Use SGD (or variation) to find optimal parameters $\boldsymbol{w}, \boldsymbol{b}$
    - Gradients from backprop

Learning

- Learning means generalization to unknown dataset
  - (so far no 'real' learning)
  - I.e., train on known dataset -> test with optimized parameters on unknown dataset

- Basically, we hope that based on the train set, the optimized parameters will give similar results on different data (i.e., test data)

Learning

- Training set (‘*train*’):
  - Use for training your neural network
- Validation set (‘*val*’):
  - Hyperparameter optimization
  - Check generalization progress
- Test set (‘*test*’):
  - Only for the very end
  - NEVER TOUCH DURING DEVELOPMENT OR TRAINING

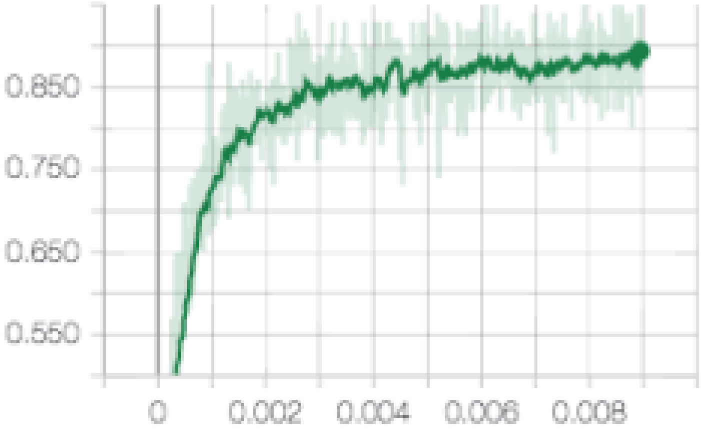
Learning

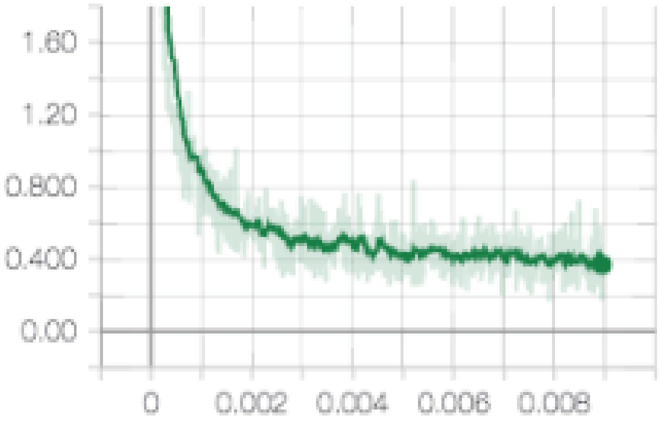
- Typical splits
  - Train (60%), Val (20%), Test (20%)
  - Train (80%), Val (10%), Test (10%)

- During training:
  - Train error comes from average mini-batch error
  - Typically take subset of validation every n iterations

Learning

- Training graph
  - Accuracy
  - Loss

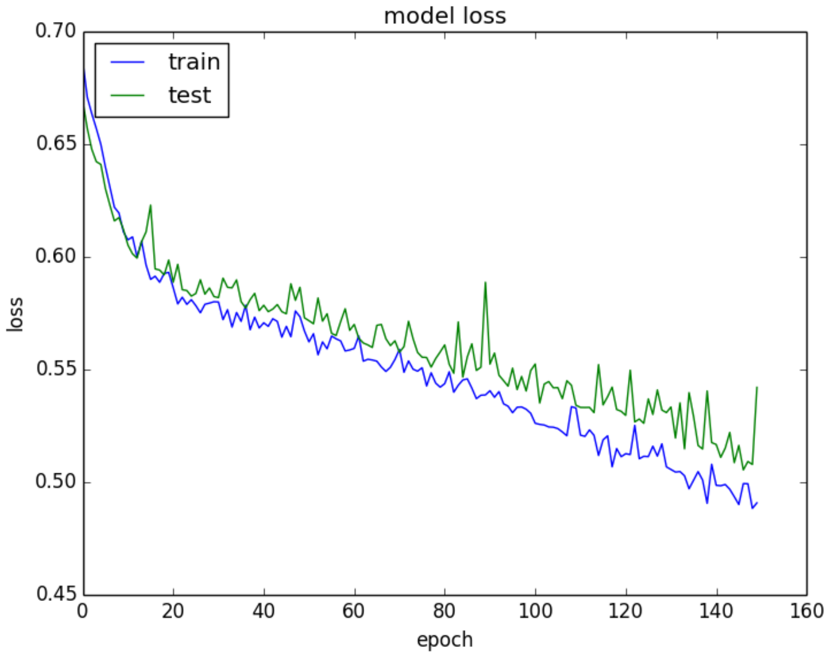




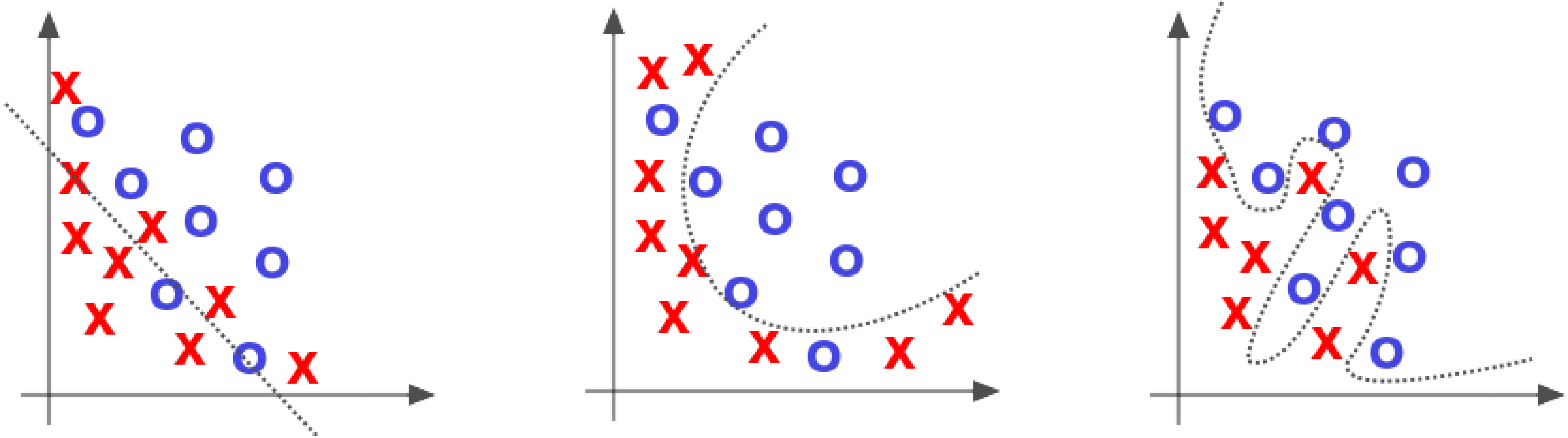
(EMA smoothing)

Learning

- Validation graph



Over- and Underfitting



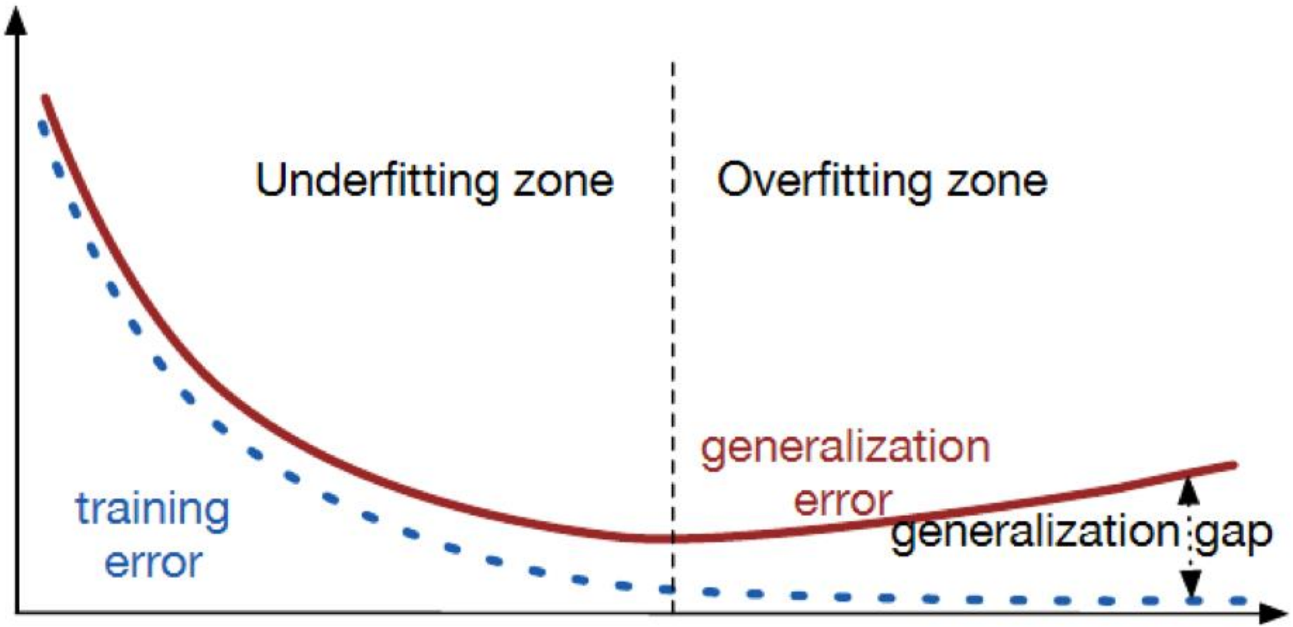
Underfitted

Appropriate

Overfitted

Figure extracted from Deep Learning by Adam Gibson, Josh Patterson, O'Reily Media Inc., 2017

Over- and Underfitting



Source: http://srdas.github.io/DLBook/ImprovingModelGeneralization.html

Hyperparameters

- Network architecture (e.g., num layers, #weights)
- Number of iterations
- Learning rate(s) (i.e., solver parameters, decay, etc.)
- Regularization (more later next lecture)
- Batch size
- …
- Overall: learning setup + optimization = hyerparameter

Hyperparameter Tuning

- Methods:
  - Manual search: most common ☺
  - Grid search (structured, for ‘real’ applications)

    Define ranges for all parameters spaces and select points (usually pseudo-uniformly distributed). Iterate over all possible configurations
  - Random search:

    Like grid search but one picks points at random in the predefined ranges

Simple Grid Search Example

```
learning_rates = [1e-2, 1e-3, 1e-4, 1e-5]
regularization_strengths = [1e2, 1e3, 1e4, 1e5]
num_iters = [500, 1000, 1500]
best_val = 0

for learning_rate in learning_rates:
  for reg in regularization_strengths:
    for iterations in num_iters:
      model = train_model(learning_rate, reg., iterations)
      validation_accuracy = evaluate(model)
      if validation_accuracy > best_val:
        best_val = validation_accuracy
        best_model = model
```

Cross Validation

- Example: k=5

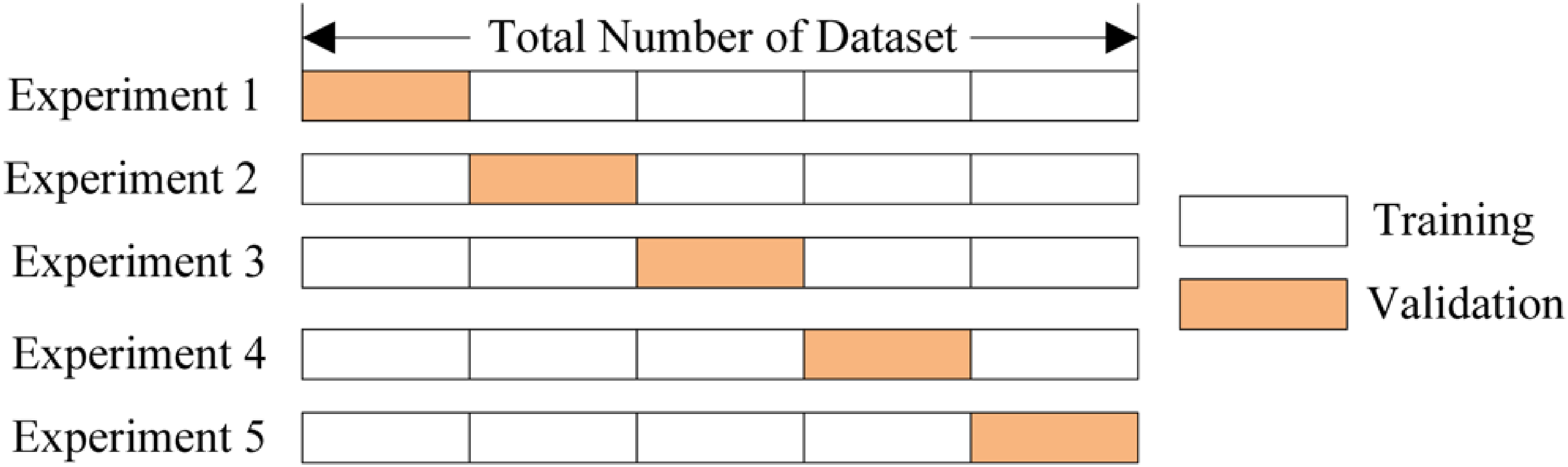


Figure extracted from cs231n

Cross Validation

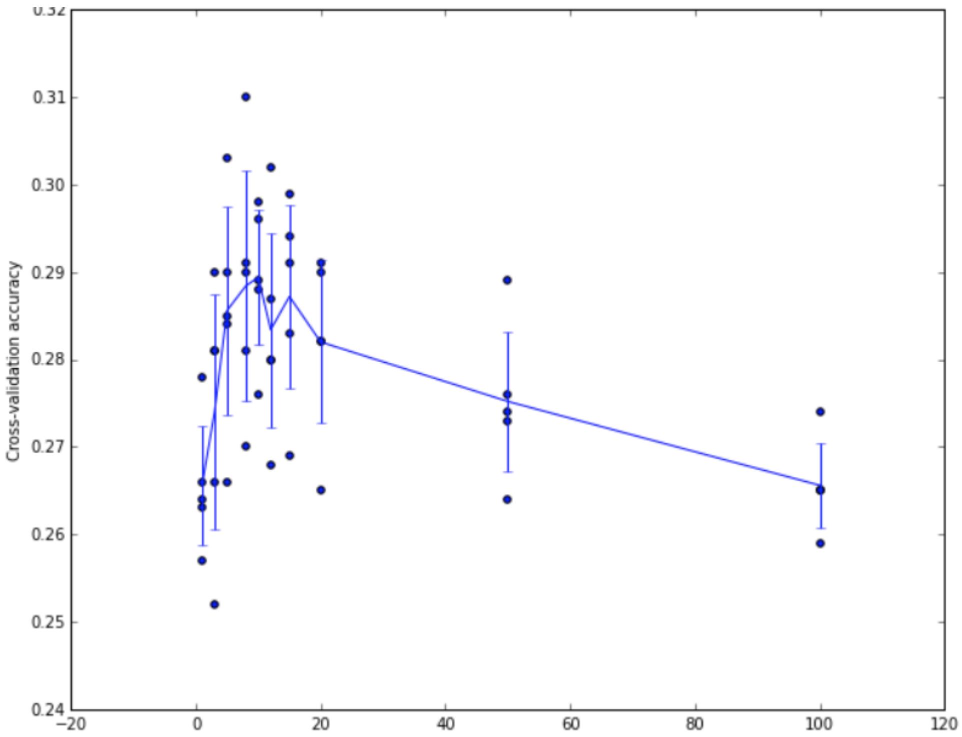
- Used when data set is extremely small and/or our method of choice has low training times

- Partition data into k subsets, train on k-1 and evaluate performance on the remaining subset

- To reduce variability: perform on different partitions and average results

Cross Validation

Results for k=5



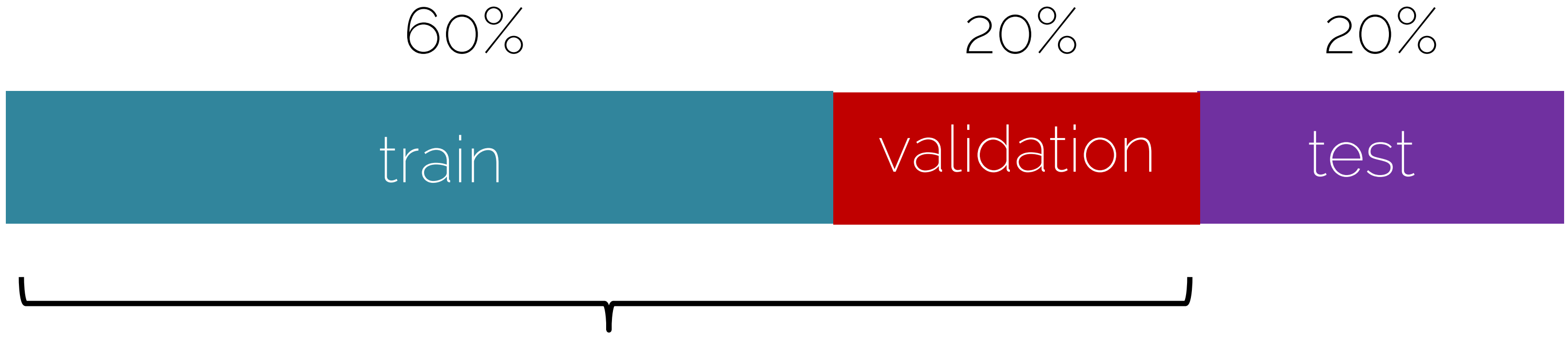
Hyperparmeter value

Figure extracted from cs231n

Basic recipe for machine learning

Basic recipe for machine learning

- Split your data



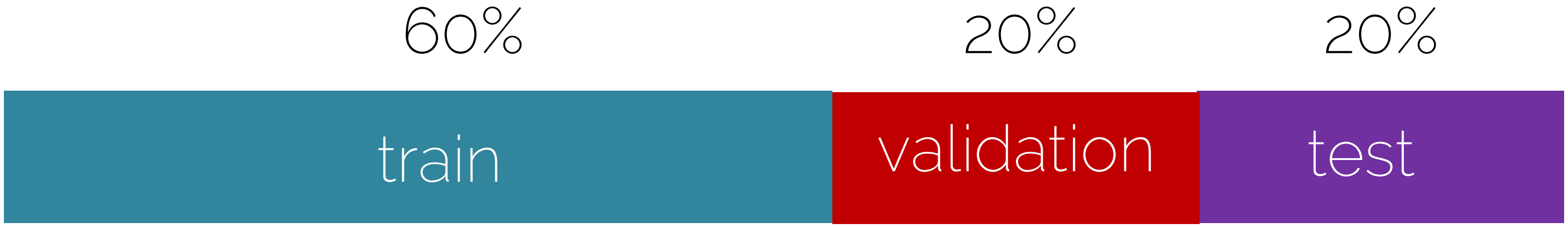
60% 20% 20%

train validation test

Find your hyperparameters

Basic recipe for machine learning

- Split your data


60% 20% 20%

train validation test

Human level error 1%

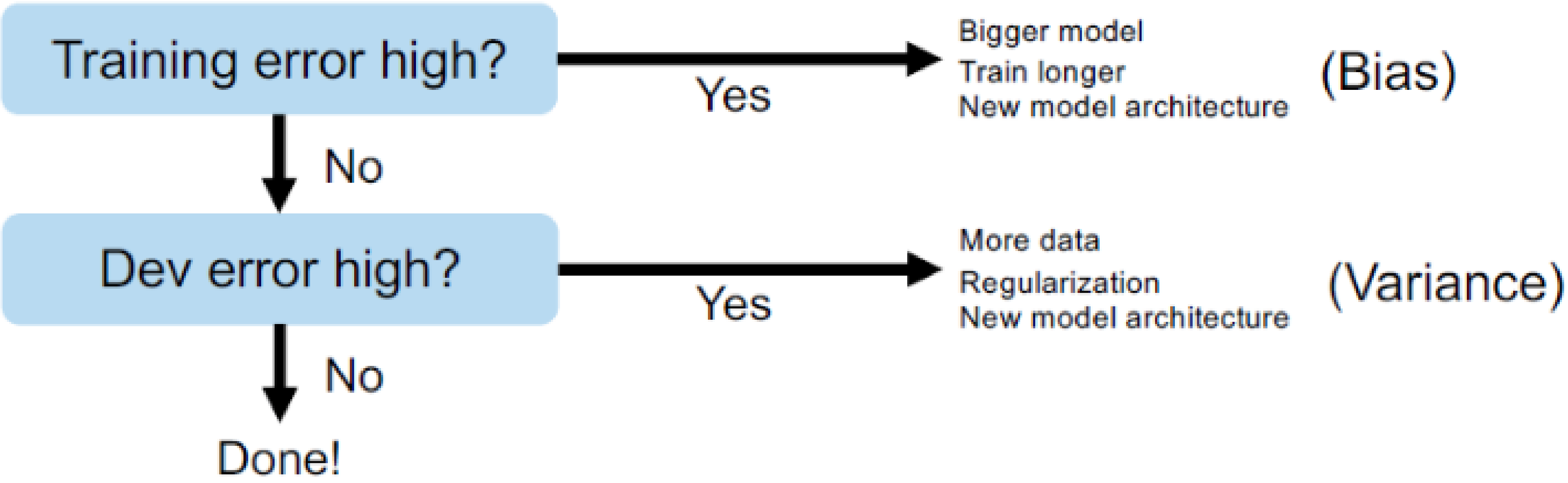
Training set error 5%

Val/test set error 8%

Bias (or underfitting)

Variance (overfitting)

Basic recipe for machine learning



Training error high?

Yes

Bigger model
Train longer
New model architecture

(Bias)

No

Dev error high?

Yes

More data
Regularization
New model architecture

(Variance)

No

Done!

More on

Credits: A. Ng

Next lecture

- Monday: Deadline Ex1!
- Next Tuesday:
  - Discussion solution exercise and presentation exercise 2
- Next lecture on Dec 6th:
  - Training Neural Networks

See you next week!