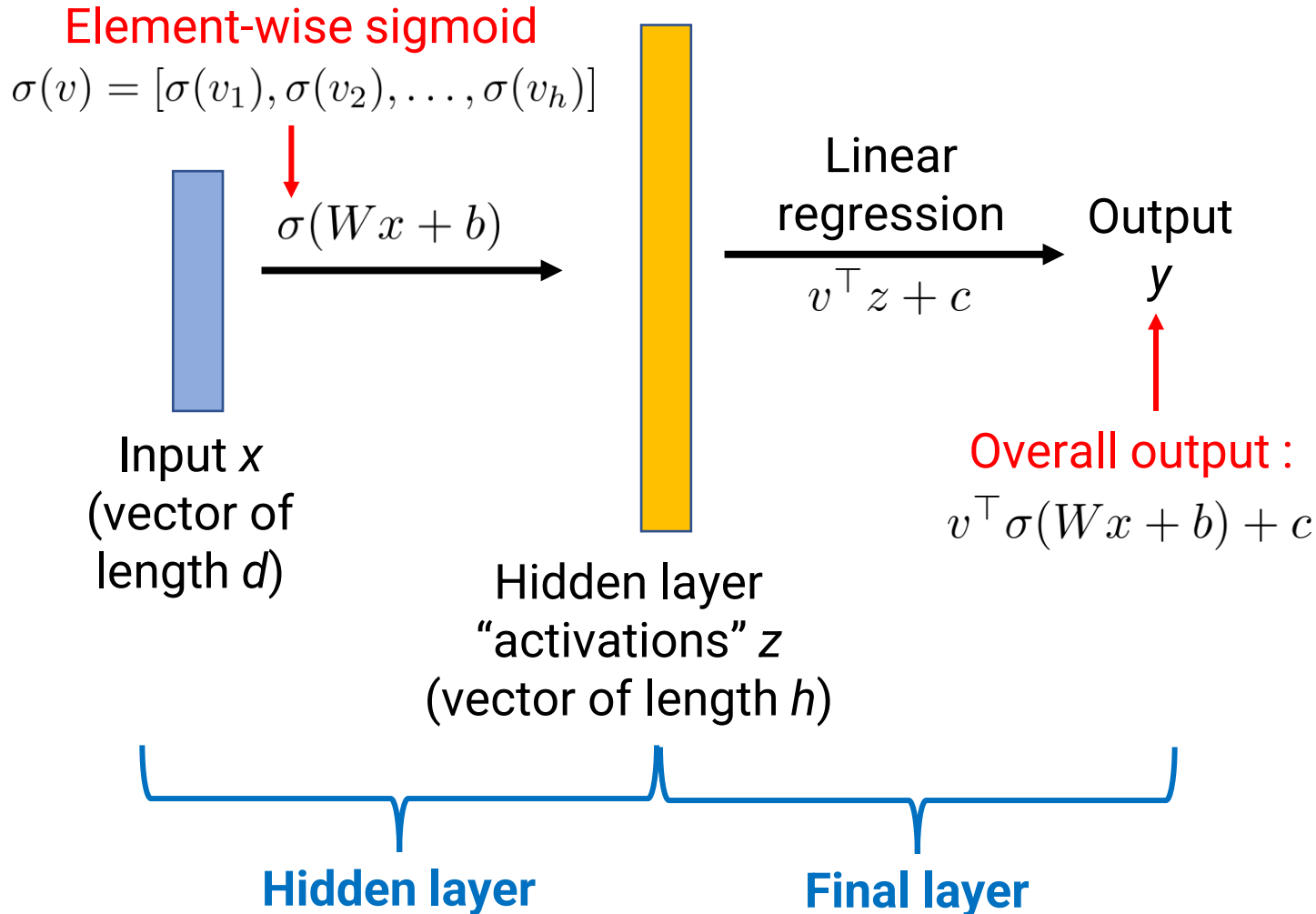


Neural Networks III: Optimization, Regularization

Robin Jia
USC CSCI 467, Spring 2025
February 20, 2025

Review: Neural Networks (2-layer MLP)



- Hidden layer = A bunch of logistic regression classifiers
 - Parameters: w_j and b_j for each classifier, for each $j=1, \dots, h$
 - Equivalently: matrix W ($h \times d$) and vector b (length h)
 - h = number of neurons in hidden layer (“hidden nodes”)
 - Produces “activations” = learned feature vector
- Final layer = linear model
 - For regression: linear model with weight vector v and bias c
- Parameters of model are $\theta = (W, b, v, c)$

Review: Training Neural Networks

Linear Regression

- Model's output is

$$g(x) = w^\top x + b$$

- (Unregularized) loss function is

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

Regression w/ Neural Networks

- Model's output is

$$g(x) = v^\top \sigma(Wx + b) + c$$

- **Use same loss function**, in terms of g !

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

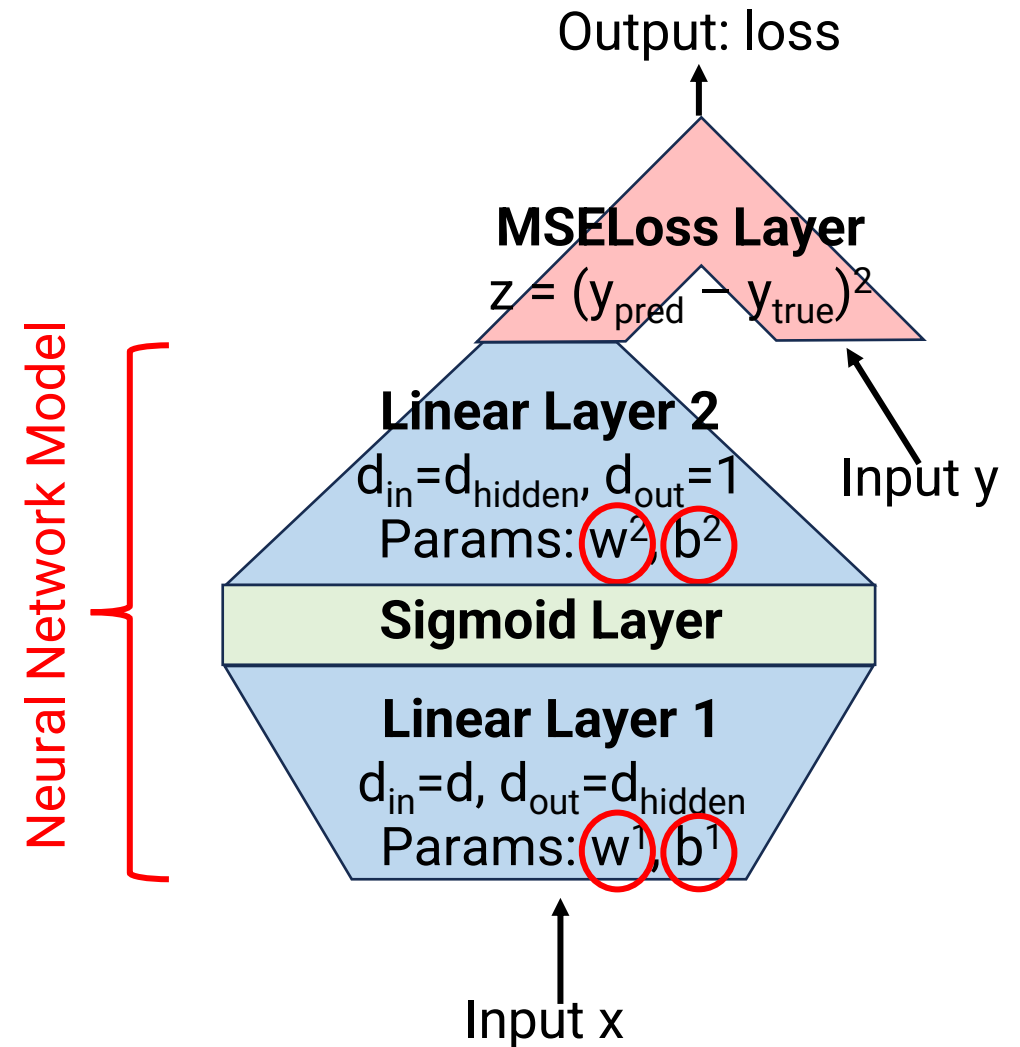
Training objective for both types of models:

$$\frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, g(x^{(i)})), \text{ where } \ell(y, u) = (y - u)^2$$

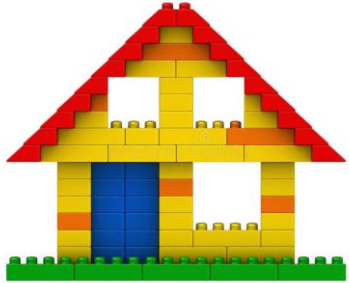
Also applies for
logistic regression,
softmax regression, etc.

Review: Neural Building Blocks

- Can build many different neural architectures from same set of building blocks
- To train any model, first build the computation graph that computes the loss
- With **backpropagation**, gradient of loss w.r.t. parameters can be computed automatically!
- Update all parameters with gradient descent update rule



Neural Network Hyperparameters (so far)



- Architecture
 - How many “neurons” (i.e., how big is hidden layer)?
 - How many layers?
 - Which activation function (sigmoid, tanh, ReLU)?
- Optimization [coming today]
- Regularization [coming today]

**How to choose? Big grid search or random search,
optimize for development set**

Today's Plan

- Optimization (i.e., Training)
 - Stochastic gradient descent
 - Random initialization
 - Learning rate schedules
 - Momentum & Adam
- Regularization
 - Early stopping
 - Dropout

Stochastic gradient descent

General loss function: $\frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, g(x^{(i)}))$

Model's output, depends on parameters θ

Gradient Descent

$$\theta \leftarrow \theta - \eta \cdot \underbrace{\frac{1}{n} \sum_{i=1}^n \nabla_{\theta} \ell(y^{(i)}, g(x^{(i)}))}_{\text{Average of per-example gradients}}$$

- Disadvantage: 1 update is $O(n)$ time
 - What if dataset is very large?
- Idea: Approximate with sample mean

Stochastic Gradient Descent

- Sample a *batch* B of examples from the training dataset
- Do the update

$$\theta \leftarrow \theta - \eta \cdot \underbrace{\frac{1}{|B|} \sum_{(x,y) \in B} \nabla_{\theta} \ell(y, g(x))}_{\text{Sample mean within batch}}$$

Sample mean within batch

Stochastic gradient descent

In practice, partition training set into batches:

for $t = 1, \dots, T$: Each t (i.e., each pass over the dataset) is called one “epoch”

Randomly partition training examples into batches $B_1, \dots, B_k$

for $i = 1, \dots, k$:

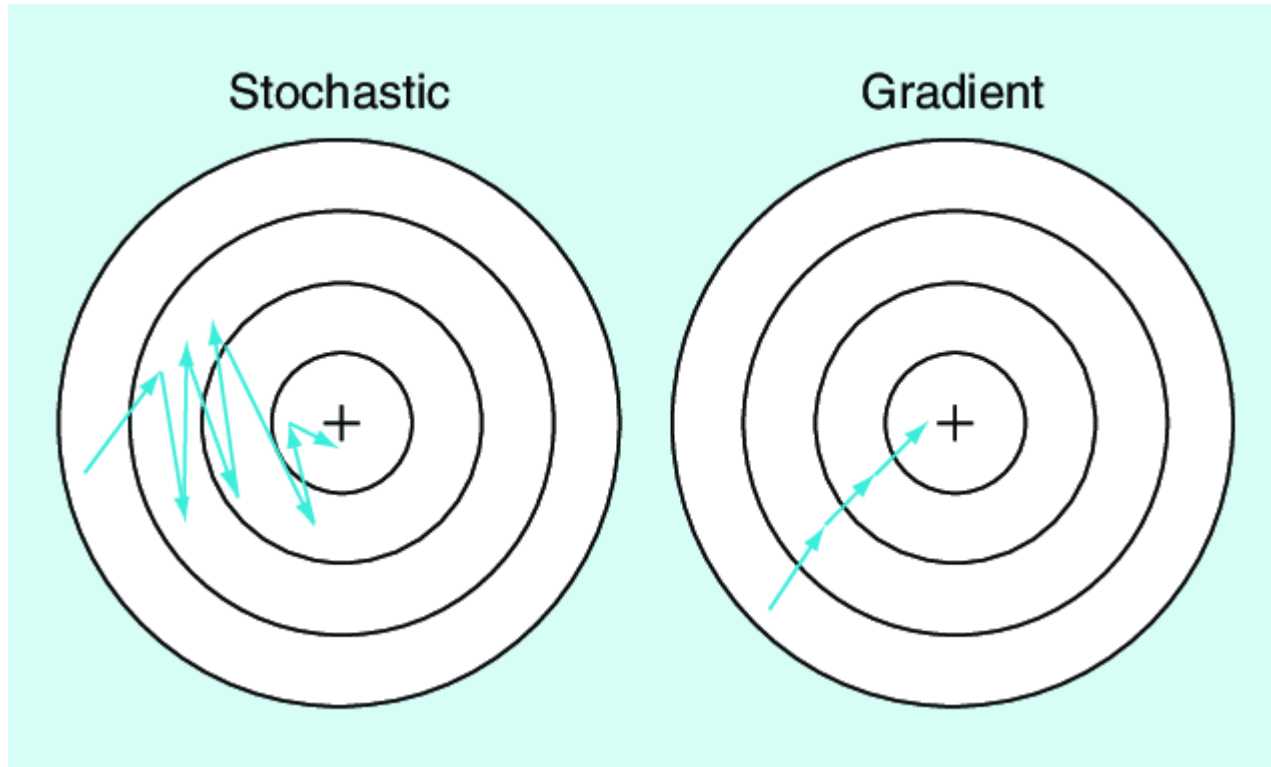
$$\theta \leftarrow \theta - \eta \cdot \frac{1}{|B_i|} \sum_{(x,y) \in B_i} \nabla_{\theta} \ell(y, g(x))$$

Update based on sample mean within current batch

How many batches? Desired “batch size” (# examples/batch) is another hyperparameter to tune

- Larger batch size = more accurate gradient, but slower
- Smaller batch size = faster, but may wander in suboptimal directions
- SGD is most useful when training data is large, computing full gradient is expensive
 - Can be used with any model

Stochastic gradient descent



- SGD: Each parameter update is only “approximately” going towards the minimum
- But given enough time, you’ll end up in (almost) the same place
 - Plus each step is much faster!

Training Objective Comparison

Linear Regression

- Model's output is

$$g(x) = w^\top x + b$$

- (Unregularized) loss function is

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

**Convex loss function
when $g(x)$ is linear**

Regression w/ Neural Networks

- Model's output is

$$g(x) = v^\top \sigma(Wx + b) + c$$

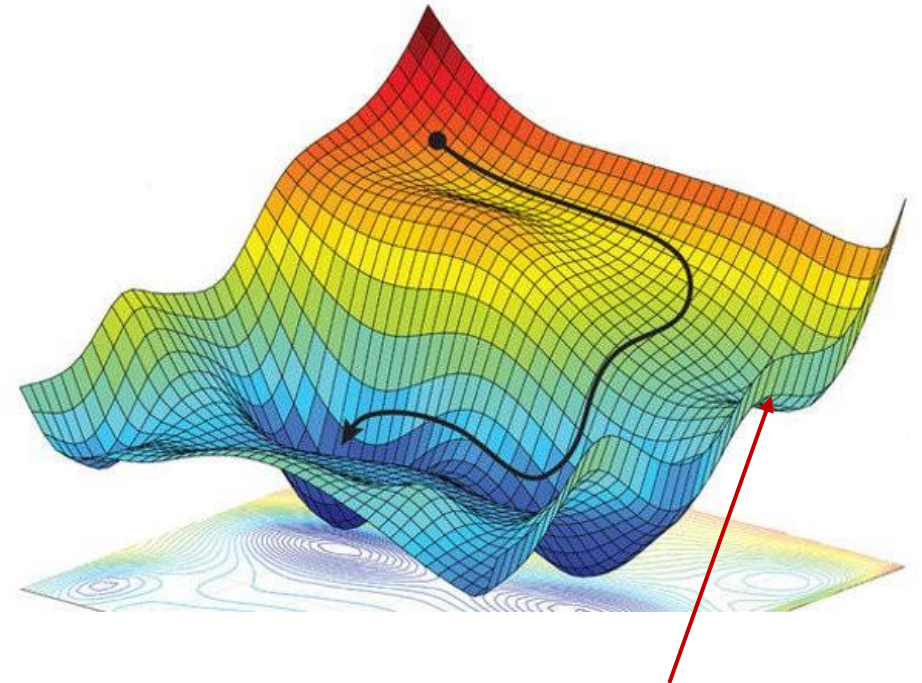
- **Use same loss function**, in terms of g !

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

**Non-convex loss function
when $g(x)$ is neural network**

Non-convexity During Training

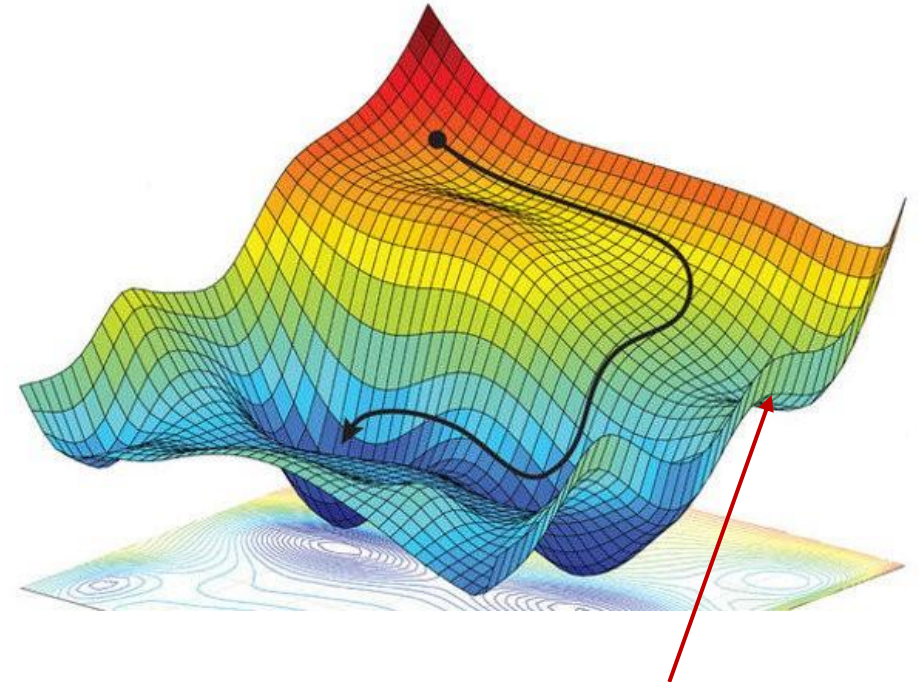
- Linear models were **convex**
 - All local optima are global optima
 - All reasonable optimization methods will find a global optimum
- Neural Networks are **non-convex**
 - Very hard to find global optimum
 - Different optimization techniques will converge to **different local optima**, some of which are much better than others
 - **Choice of optimization method really matters!**



Local minimum
Gradient descent can get stuck here!

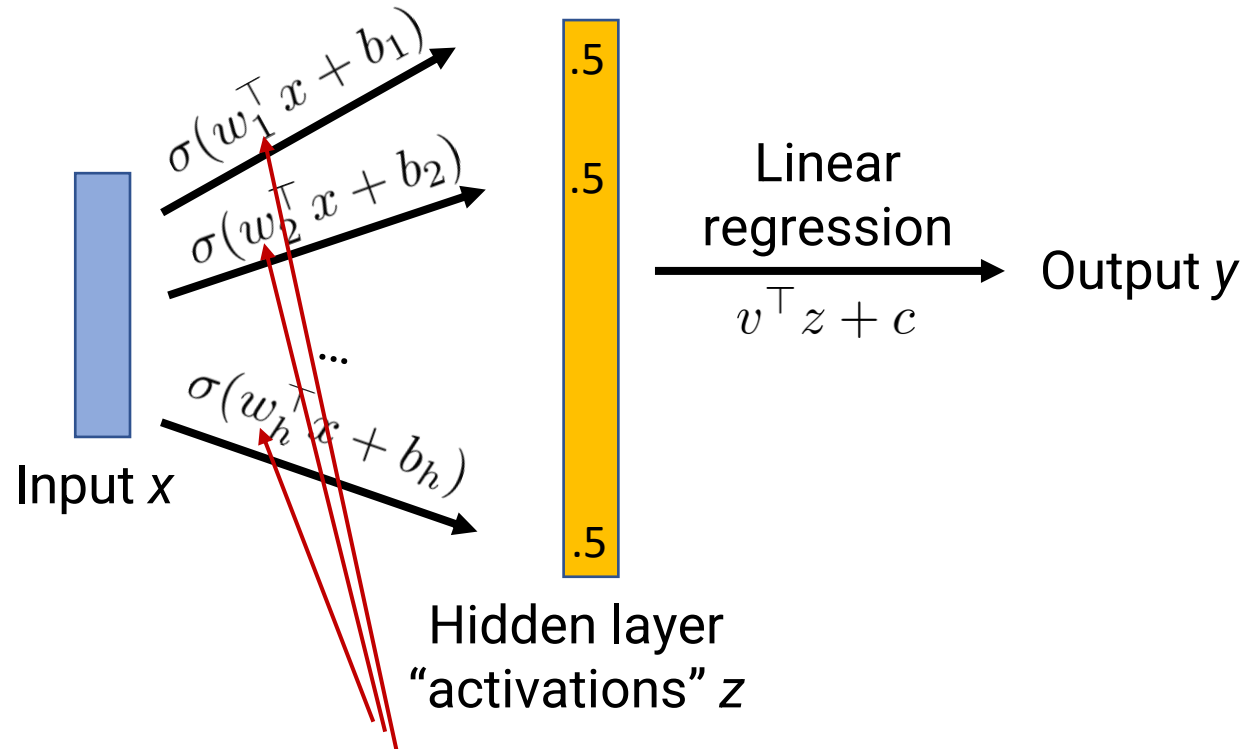
Initialization

- For convex problems (e.g. logistic regression), initialization doesn't matter much for final result
 - We just initialize parameters to all 0's
- For neural networks, initialization matters a lot!
 - Optimization problem is non-convex
 - Where you start determines what parameters you learn



Local minimum
If you initialize here,
you get stuck here!

The problem with all-0's initialization

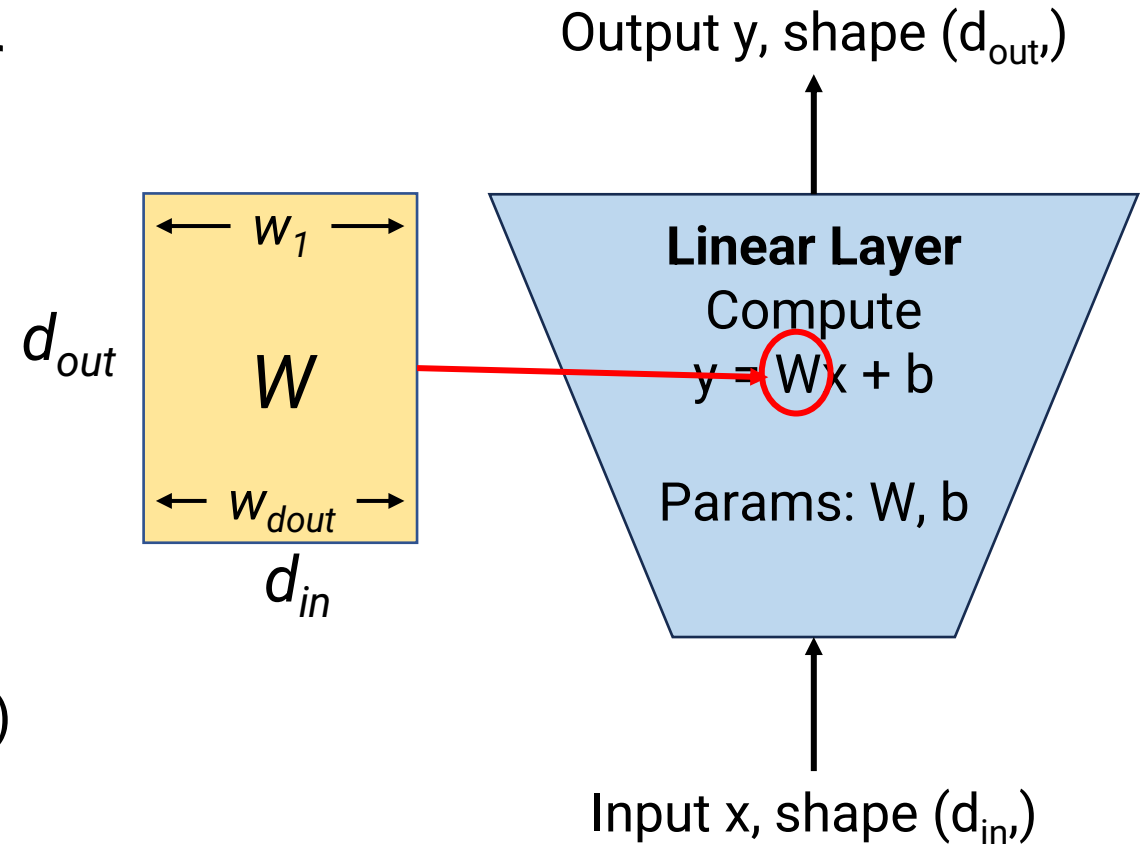


If every w_j starts as 0 vector,
gradient update to each w_j will be the same

- What if we initialize with all 0's?
- Problem: Symmetry
 - All hidden units start out the same, so gradients will be the same for each
 - Thus, all hidden units will stay the same!
- **We must initialize in a way that breaks the symmetry**

How to initialize neural networks?

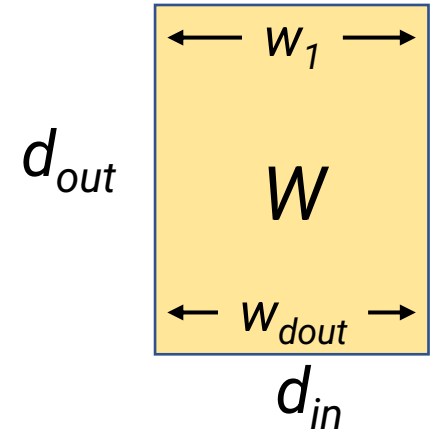
- Solution: Initialize every parameter to be a small random number
- How small? Depends on “fan-in” d_{in} (# input features) and “fan-out” d_{out} (# output features)
 - Suppose input x_j 's have variance γ^2 and w_{ij} 's have variance σ^2
 - For each row w_i of W :
$$\text{Variance}[w_i^\top x] = d_{in} \sigma^2 \gamma^2.$$
(Sum of d_{in} things, each with var. $\sigma^2 \gamma^2$)
 - This gets bigger as d_{in} gets bigger
 - So: choose σ^2 proportional to $1/d_{in}$



How to initialize neural networks

- As usual, you have many options...

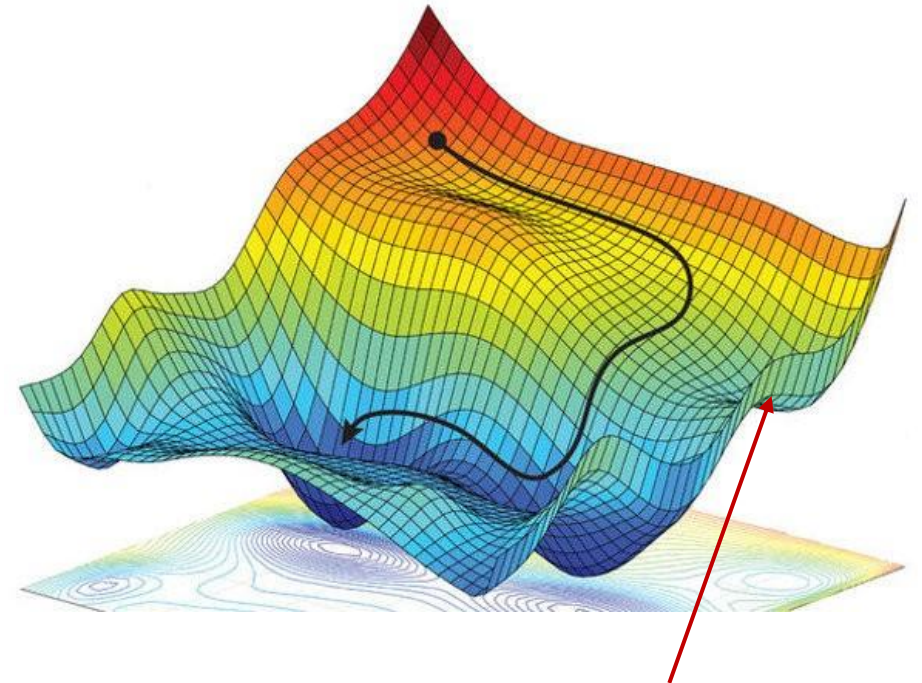
- He initialization: $\text{Normal}\left(0, \frac{2}{d_{in}}\right)$ (mean 0, variance $2/d_{in}$)
- Xavier initialization: $\text{Normal}\left(0, \frac{2}{d_{in} + d_{out}}\right)$ Also divides by d_{out}
(But usually d_{in} and d_{out} are similar size)
- Pytorch default: $\text{Uniform}\left(-\frac{1}{\sqrt{d_{in}}}, \frac{1}{\sqrt{d_{in}}}\right)$ Uniform avoids large outliers
Note: Variance is proportional to $1/d_{in}$



- Usually you don't tune these as hyperparameters, just use defaults

Importance of Learning Rate

- For convex problems (e.g. logistic regression), **learning rate** doesn't change final result very much
 - All reasonable values converge to the same final answer
- For neural networks, learning rate matters a lot!



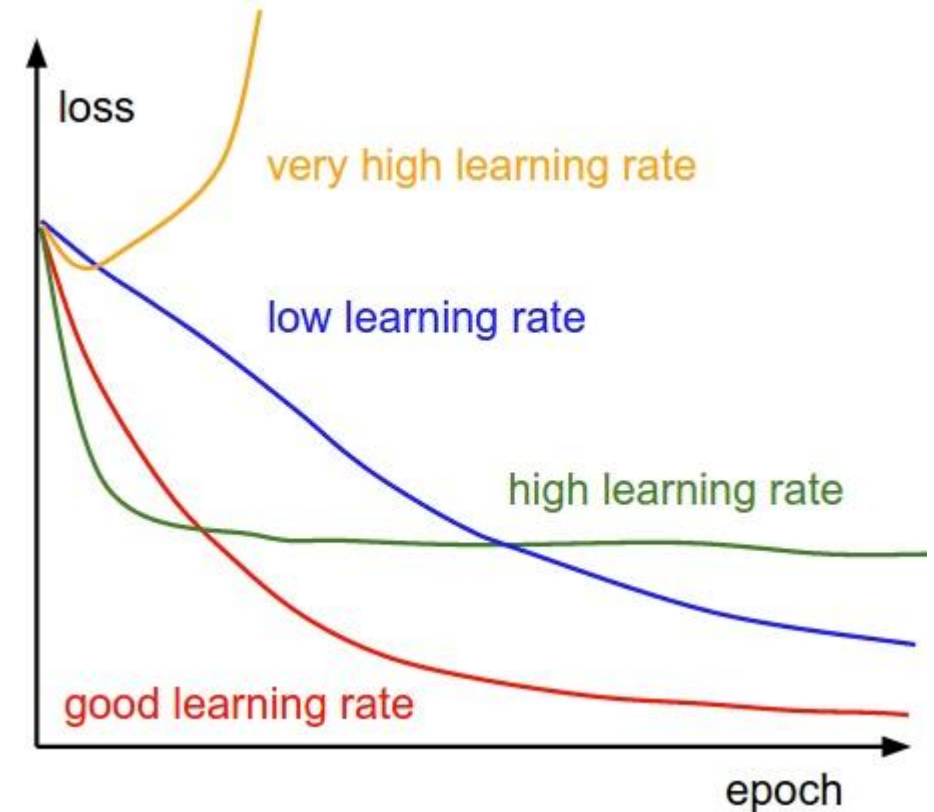
Local minimum

Get stuck here if learning rate
is too small

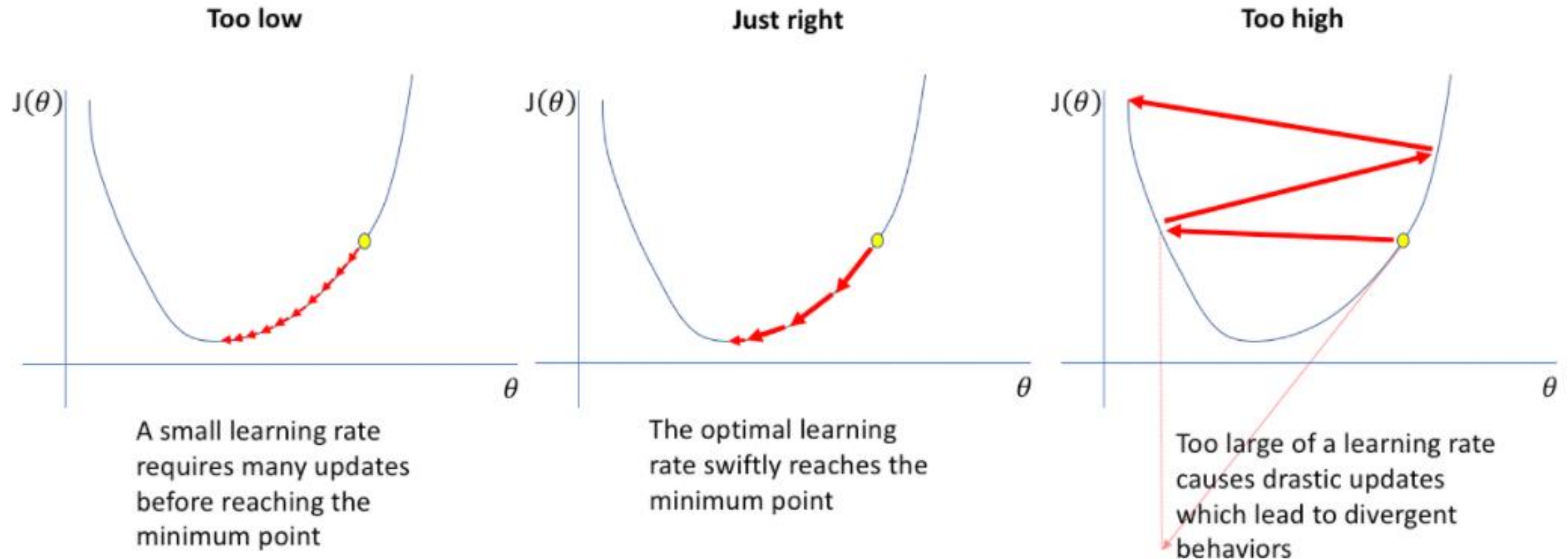
Importance of Learning Rate

$$\theta \leftarrow \theta - \boxed{\eta} \cdot \frac{1}{|B|} \sum_{(x,y) \in B} \nabla_{\theta} \ell(y, g(x))$$

- Too small: Can't take big enough steps, don't converge fast enough, can get stuck in "flat" regions
- Too large: Steps are too large, too erratic and doesn't converge
- Need to carefully tune learning rate

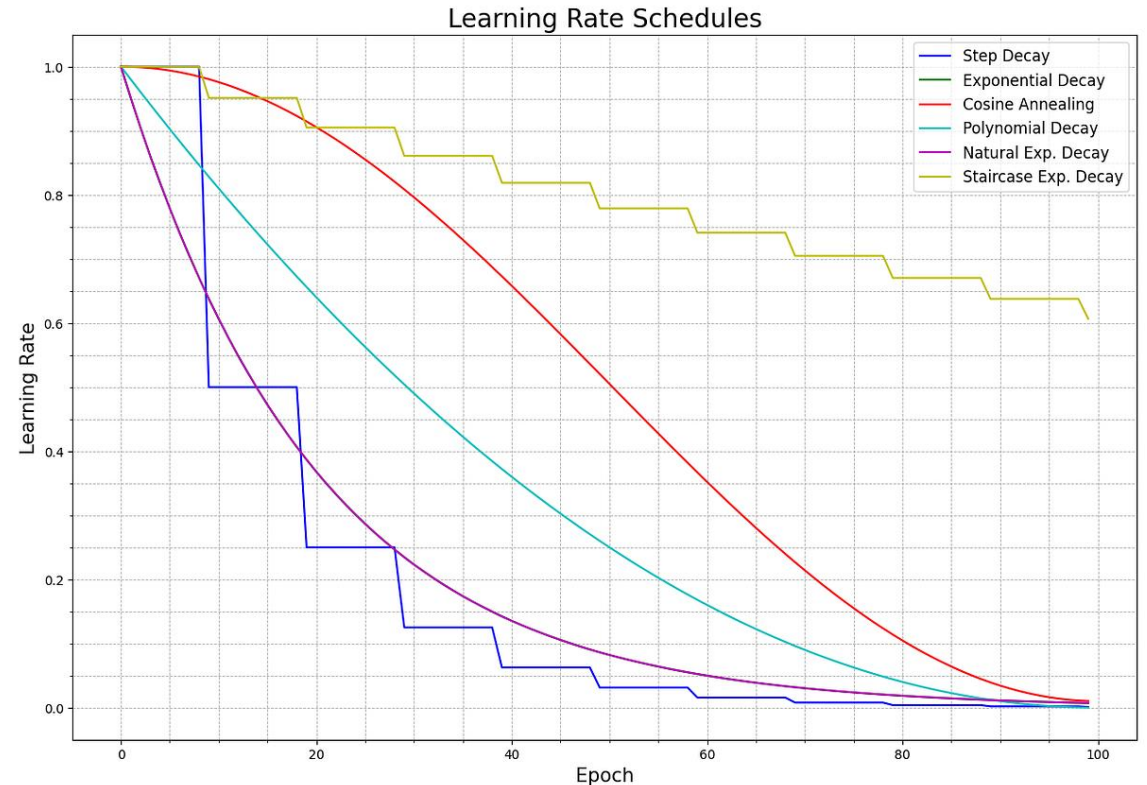


Importance of Learning Rate



Learning Rate Schedules

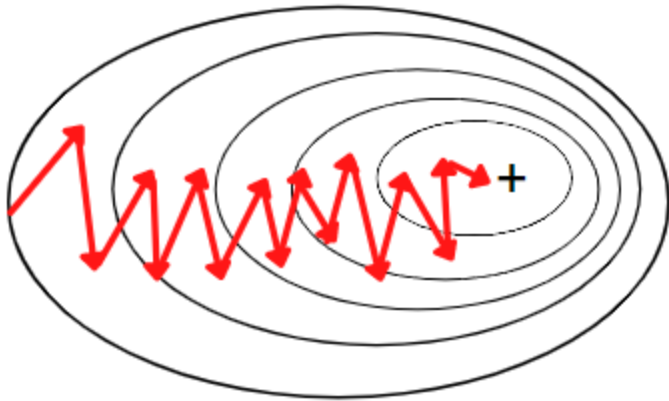
- Early on: We're far from the optimum, want to take large steps
- Later on: We're close to the optimum, take small steps so we don't "overshoot"
- Solution: Start with large learning rate, make it smaller over time ("decay")



Challenges for SGD

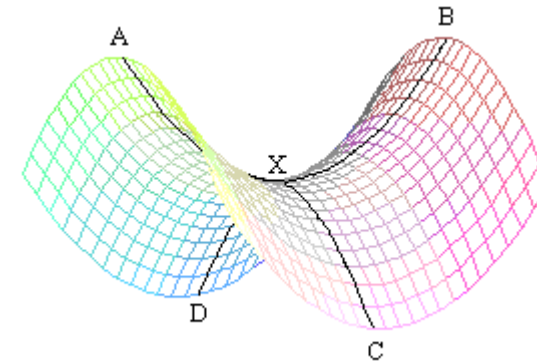
Problem #1: Ill-Conditioned loss

- Surface is very curved/steep in some directions, but shallow in others
- High learning rate: Zig-zag along steep direction
- Low learning rate: Move slowly along shallow direction



Problem #2: Saddle Points

- Saddle Point = Area that is locally flat, but neither minimum nor maximum
- SGD can get stuck here because gradient is close to 0 $\rightarrow$ take small steps



Idea #1: Momentum

SGD

$$w = w - lr * grad$$

SGD + Momentum

Momentum builds in direction of gradient

$$v = \underbrace{\text{beta1} * v}_{\text{Keep momentum, with some "friction" beta1}} + \underbrace{(1-\text{beta1}) * grad}_{\text{Apply a "force" in direction of gradient}}$$

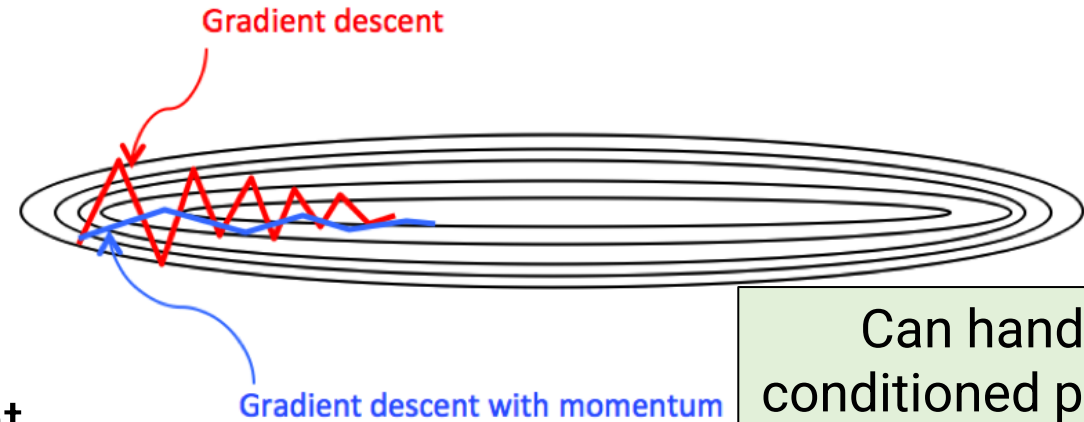
↑
"velocity" or
"momentum"

Keep momentum,
with some "friction" beta1

Apply a "force"
in direction of gradient

$$w = w - lr * v$$

↑
Parameters w are "position", they move
in the direction of momentum



Can handle ill-
conditioned problems!
Velocities cancel out
in vertical direction,
but build up in
horizontal direction

Idea #2: Per-Parameter Learning Rates

SGD

$$w = w - lr * grad$$

RMSProp: Normalize gradient for each parameter

$$s = \text{beta2} * s + (1 - \text{beta2}) * \text{grad}^{**2}$$

↑
Tracks how big grad usually
is for each parameter

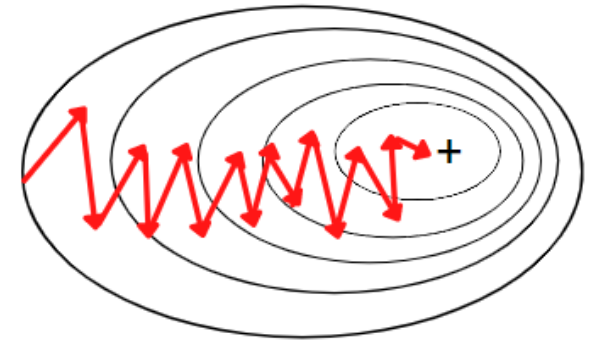
Weighted average of
elementwise squares

Elementwise square
Measures size of grad for each parameter

$$w = w - lr * grad / (\text{sqrt}(s) + \text{eps})$$

Elementwise sqrt

Avoid divide by 0



For each parameter, divide gradient by the size of its gradient on average
Result: parameters with large average gradient (steeper directions)
get smaller effective learning rate

Adam = Momentum + Per-Parameter LR

Momentum: Build momentum in direction of gradient

```
v = beta1 * v + (1 - beta1) * grad # Add momentum
w = w - lr * v # Move in direction of momentum
```

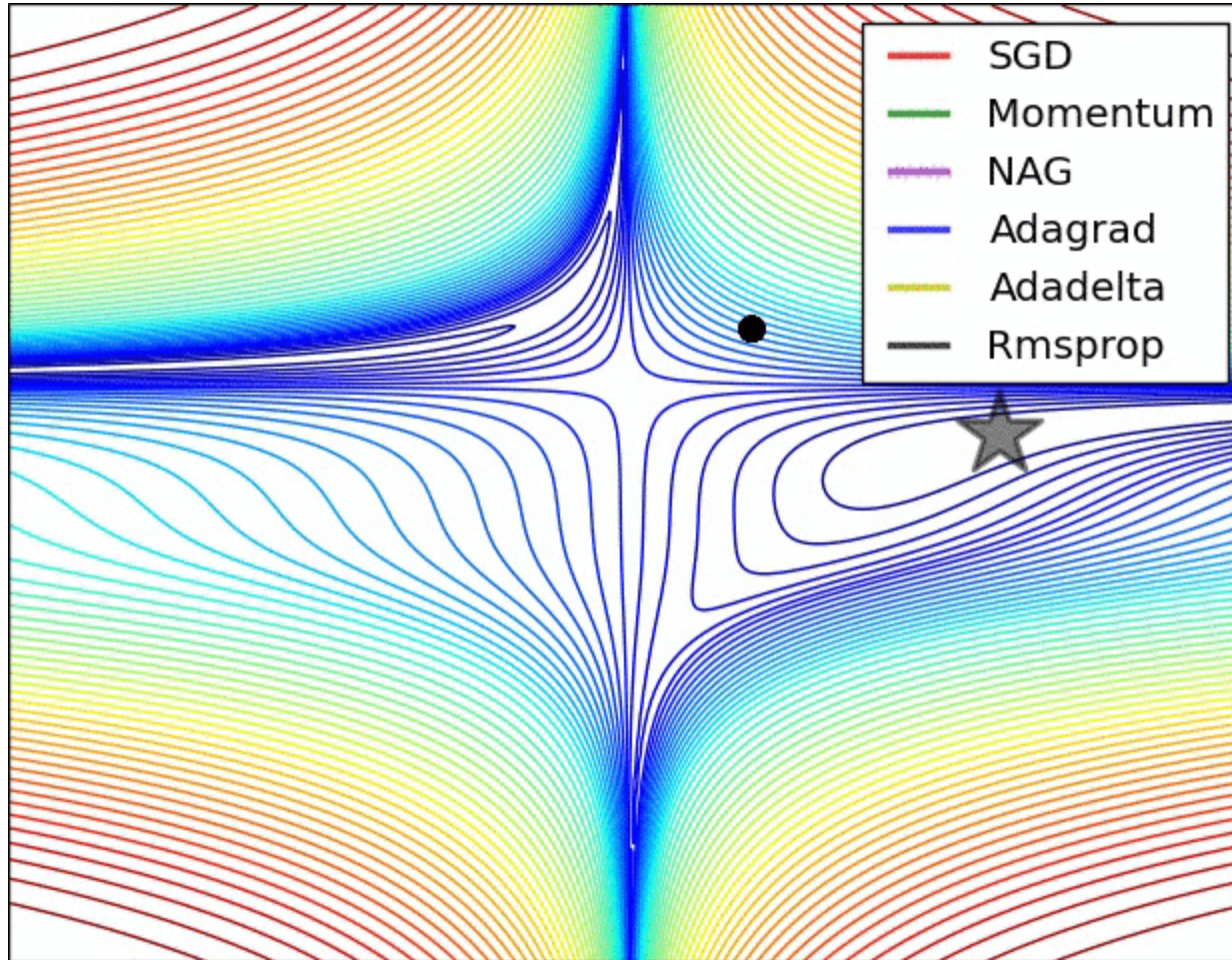
RMSProp: Adapt learning rate for each parameter separately

```
s = beta2 * s + (1-beta2) * grad**2 # Per-parameter scale
w = w - lr * grad / (sqrt(s) + eps) # Divide by scale
```

Adam: Combine Momentum and RMSProp, commonly used!

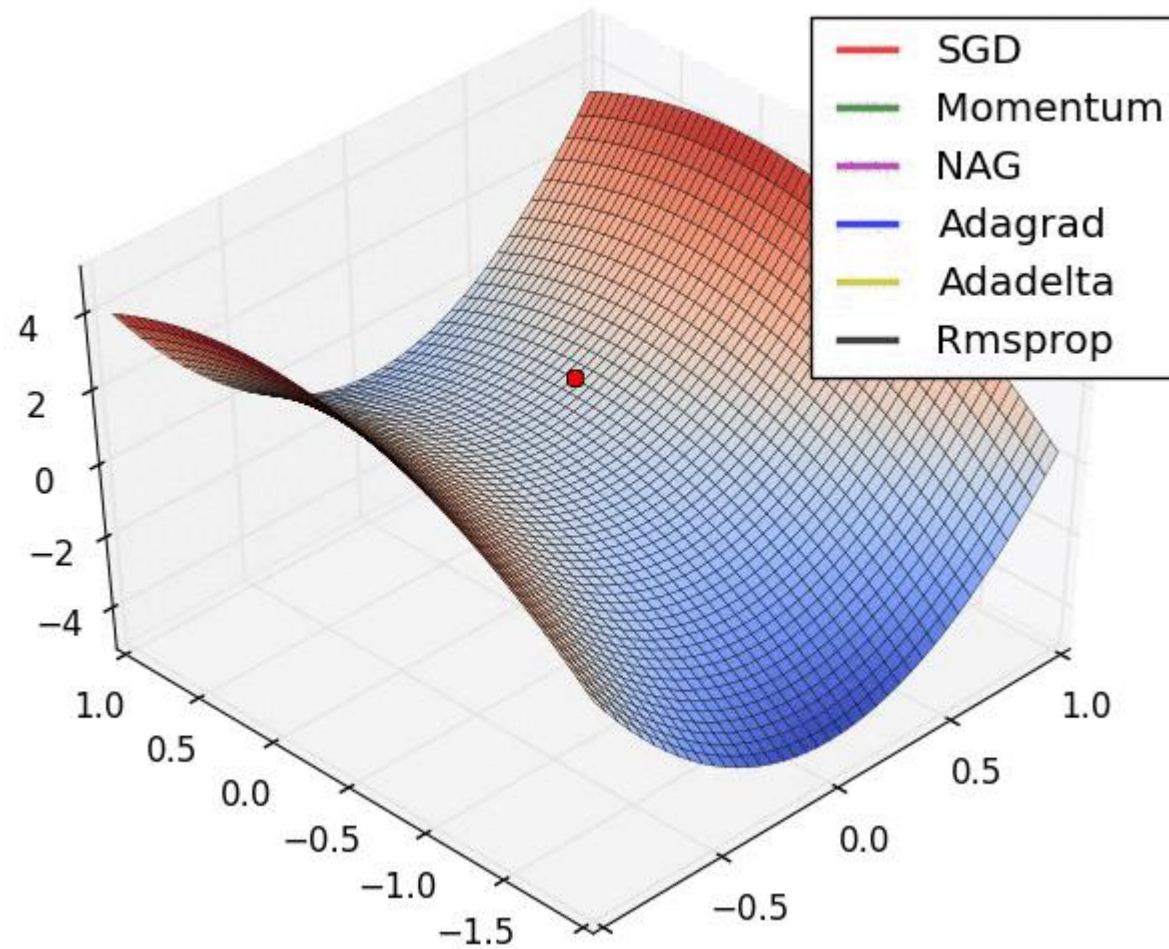
```
v = beta1 * v + (1 - beta1) * grad # Momentum
s = beta2 * s + (1-beta2) * grad**2 # RMSProp
w = w - lr * v / (sqrt(s) + eps) # Mix both update rules
```

Comparison of SGD Variants



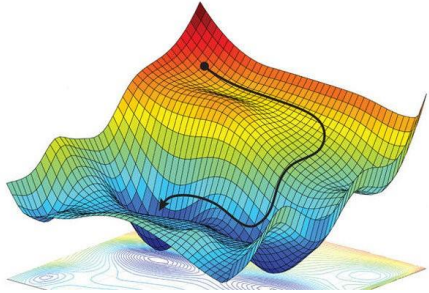
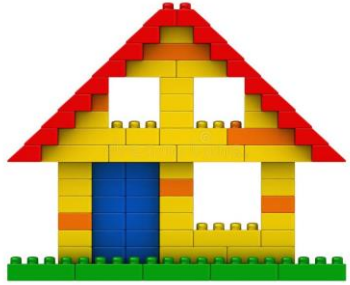
- SGD: Converges slowly
- Momentum: Moves much faster, although can overshoot
- Adagrad, RMSProp: Converge quickly here

Comparison of SGD Variants



- One problem for non-convex optimization:
Saddle points
 - Function is locally flat but is neither local minimum nor local maximum
 - Need to “escape” these to find a minimum
- SGD gets stuck, other methods can escape

Neural Network Hyperparameters



- Architecture
 - How many “neurons” (i.e., how big is hidden layer)?
 - How many layers?
 - Which activation function (sigmoid, tanh, ReLU)?
- Optimization
 - Learning rate (initial & decay)
 - Batch size
 - Optimizer (momentum? Adam?)
- Regularization [Next!]

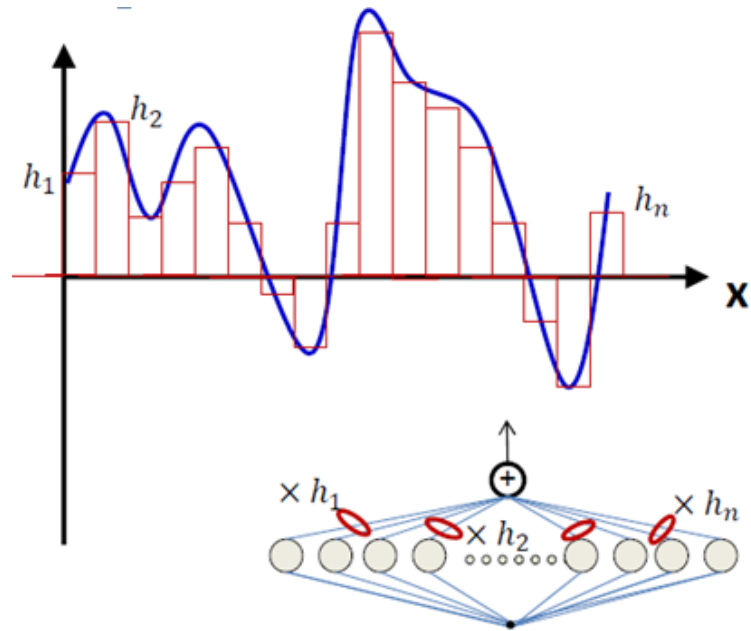
Announcements

- HW2 released, due Thursday, March 6
- Midterm exam Thursday, March 13
 - In-class, 80 minutes
 - We have both DMC 100 (this room) and SOS B4 (section room) reserved
 - Last Name A-K: Go to DMC 100
 - Last Name L-Z: Go to SOS B4
 - Allowed one double-sided 8.5x11 sheet of notes
 - Contact me ASAP about OSAS accommodations
- Section Friday: Pytorch

Today's Plan

- Optimization (i.e., Training)
 - Stochastic gradient descent
 - Random initialization
 - Learning rate schedules
 - Momentum & Adam
- Regularization
 - Early stopping
 - Dropout

Regularization & Neural Networks



- Recall: Neural networks are universal approximators
- This means they are prone to overfitting!
 - Low bias, high variance
- How to avoid overfitting too badly?

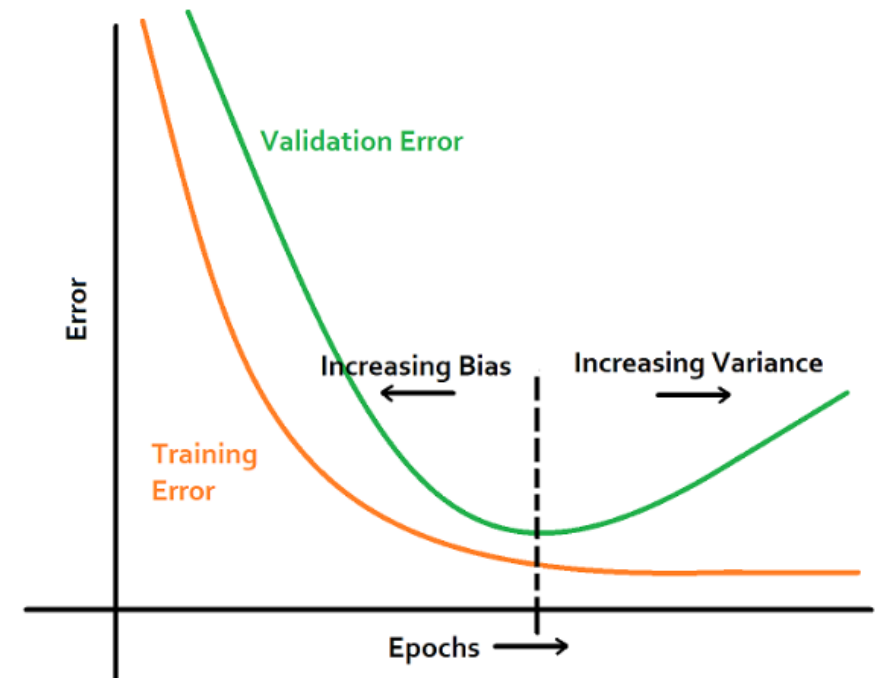
Weight decay (AKA L2 Regularization)

- L2 regularization is a valid strategy!
 - Add an L2 penalty for every parameter in the model
- Often called “weight decay” when used with neural networks
 - Because every gradient step, you add the update

$$\theta \leftarrow \theta - \eta \cdot \lambda \cdot \theta \quad \text{Weights literally “decay” by factor of } (1 - \eta\lambda)$$

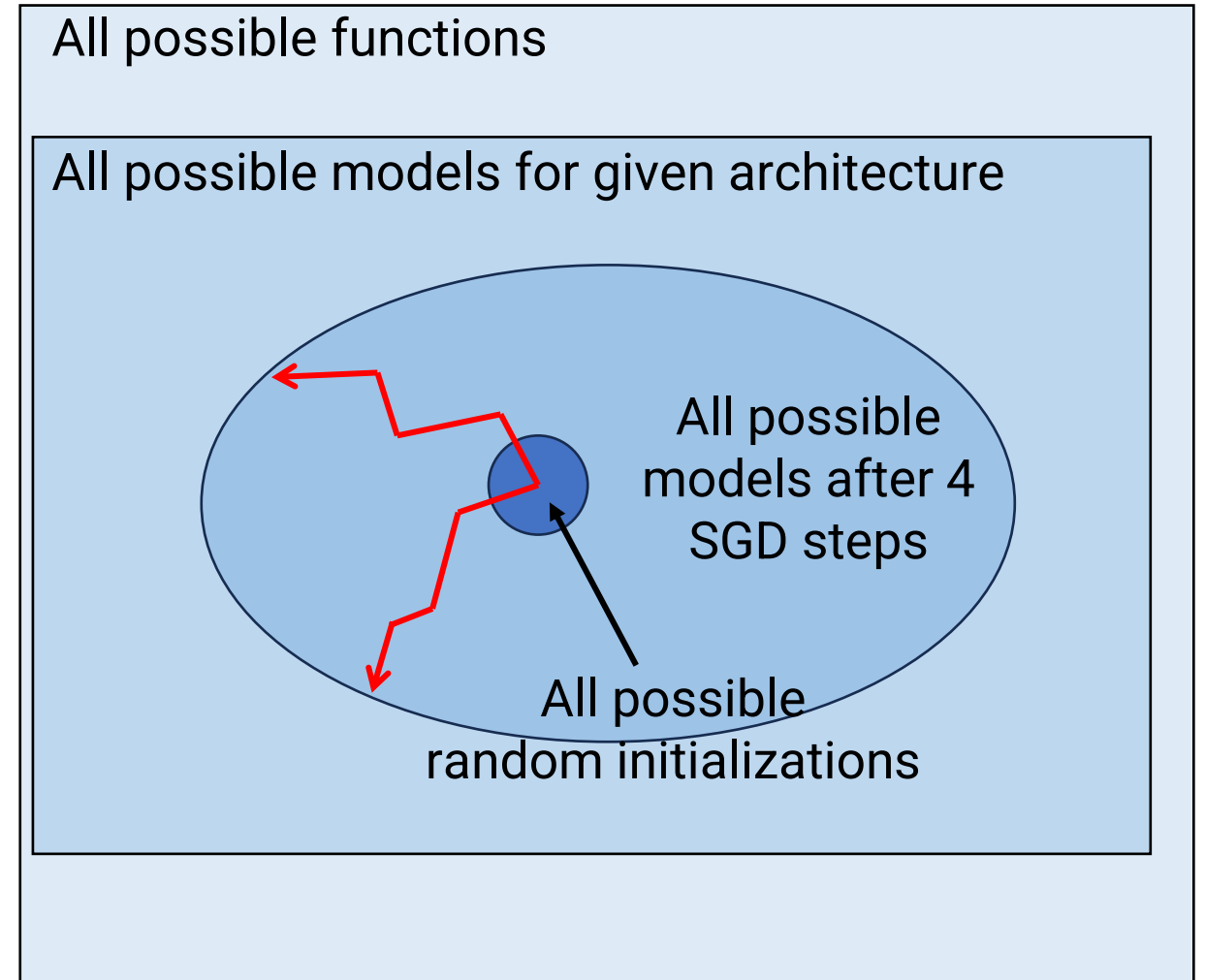
Early stopping

- Prevent overfitting by stopping training before you overfit too much
 - Every so often during training, save “checkpoint” of model parameters and evaluate development loss
 - Remember which checkpoint had best development loss
 - If development loss keeps going up, stop training
- Can be used for any model, but especially common for neural networks
 - For linear models, also common to train all the way to convergence

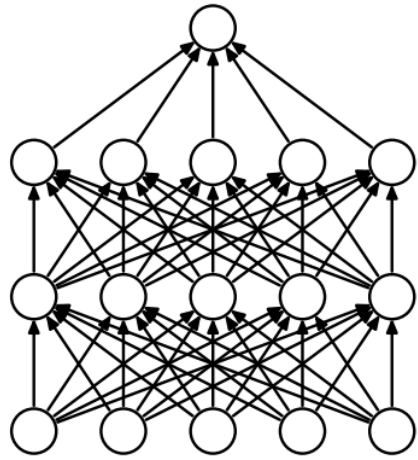


Why Does Early Stopping Apply Regularization?

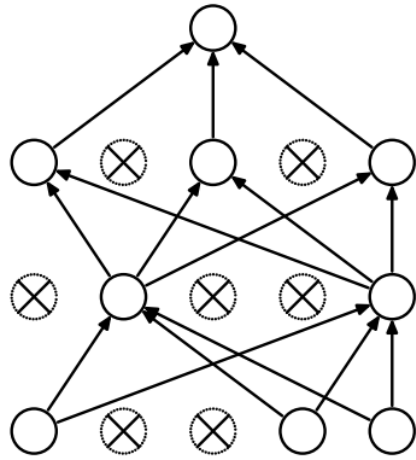
- Set of “Models that can be learned after T steps” is smaller than all possible models
 - Parameters start as small random numbers
 - Early stopping prevents parameters from changing too much from this initialization
- Thus, model family is restricted!
 - Similar to L2 regularization



Dropout



(a) Standard Neural Net



(b) After applying dropout.

- During Training: Randomly “drop out” some neurons by setting their value to 0
 - Drop each out with probability p
 - To compensate, scale the other neurons up by $1/p$
- During testing, don't do dropout

Dropout prevents “Co-adapted” features

A problem with sexual reproduction

- Fitness depends on genes working well together. But sexual reproduction breaks up sets of co-adapted genes.
 - This is a puzzle in the theory of evolution.
- A recent paper by Livnat, Papadimitriou and Feldman (PNAS 2008) claims that breaking up complex co-adaptations is actually a good thing even though it may be bad in the short term.
 - It may help optimization in the long run.
 - It may make organisms more robust to changes in the environment. *We show this is a big effect.*



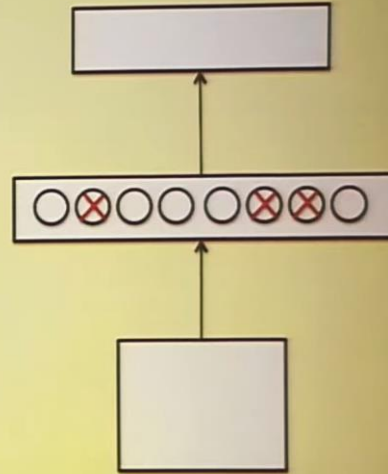
Geoffrey Hinton
(Turing Award winner,
One of the “Godfathers” of
deep learning)

- Without dropout, two neurons could compute features that are only useful in tandem
 - E.g., A and B individually are bad predictors, but A+B is useful predictor
 - A and B are “co-adapted”
- Dropout disincentivizes this—each neuron individually should be useful

Dropout as an Ensemble

Dropout: An efficient way to average many large neural nets.

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
 - All architectures share weights.



But what do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
 - This exactly computes the geometric mean of the predictions of all 2^H models.

During training, each neuron is on 1/2 the time and its value is 2x, so **on average** its output is the same as during testing

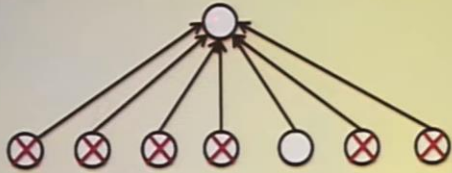


- “Ensemble” = average of multiple models’ predictions
- Usually better than using a single model
- Dropout ensembles over all 2^h different ways to dropout the h hidden neurons

Dropout as a Generalization of Naïve Bayes

A familiar example of dropout

- Do logistic regression, but for each training case, dropout all but one of the inputs.



- At test time, use all of the inputs.
 - Its better to divide the learned weights by the number of features, but if we just want the best class its unnecessary.
- This is called “Naïve Bayes”.
 - Why keep just one input?

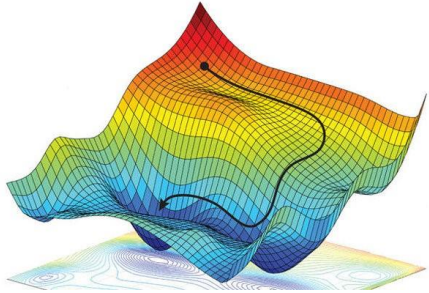
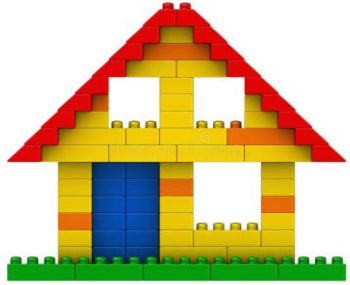
- What about dropout at the input layer?
 - If we dropout all but one feature x_j , we are just estimating $P(y|x_j)$, which is closely related to $P(x_j|y)$:

$$P(x_j | y) = \frac{P(x_j)P(y | x_j)}{P(y)}$$

- At test time, we use all features: same as multiplying all $P(x_j|y)$'s
 - This is Naïve Bayes!
- Thus, Dropout at the input layer *generalizes* Naïve Bayes



Neural Network Hyperparameters



- Architecture
 - How many “neurons” (i.e., how big is hidden layer)?
 - How many layers?
 - Which activation function (sigmoid, tanh, ReLU)?
- Optimization
 - Learning rate (initial & decay)
 - Batch size
 - Optimizer (momentum? Adam?)
- Regularization
 - Weight decay
 - Early stopping
 - Dropout

Conclusion

- Optimization: Neural networks are non-convex, so choice of optimization strategy really matters!
- Regularization: Neural networks are very good at overfitting, need to counterbalance this
- Lots of hyperparameters!