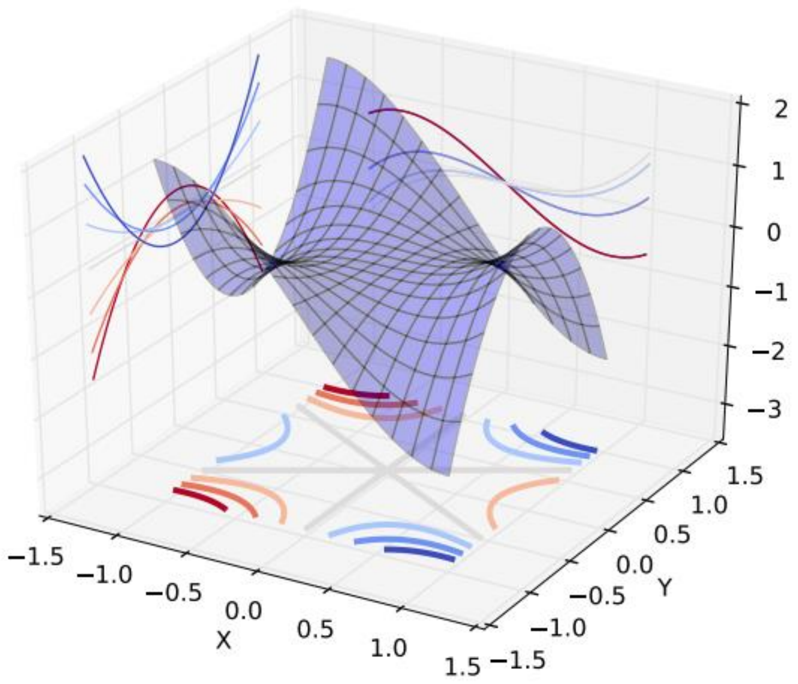
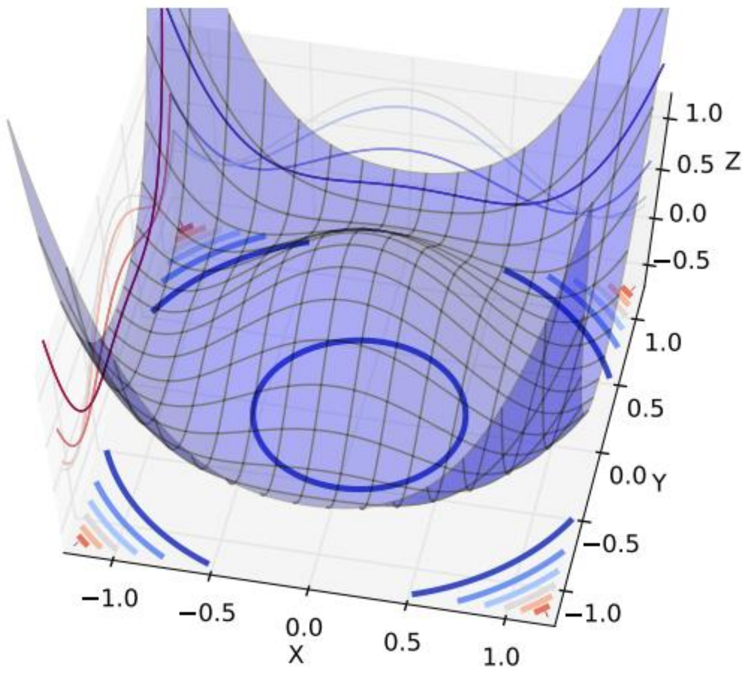
Deep Neural Networks Optimization





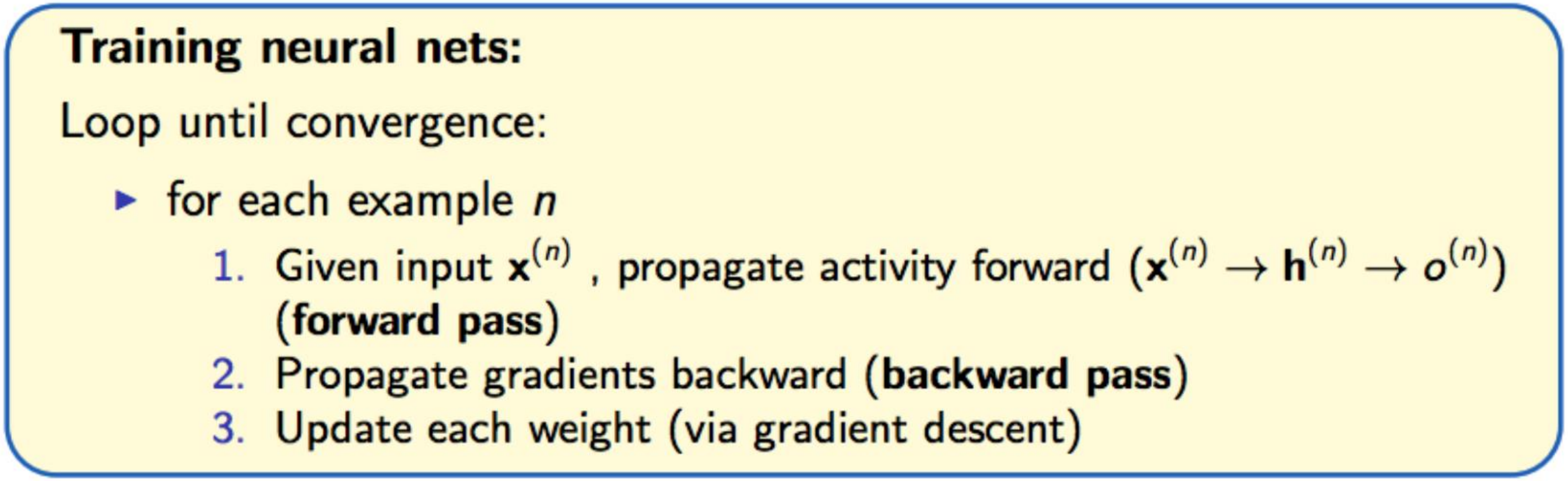
http://arxiv.org/pdf/1406.2572.pdf

Slides from Geoffrey Hinton

CSC411/2515: Machine Learning and Data Mining, Winter 2018

Michael Guerzhoy and Lisa Zhang

Gradient Descent for Neural Network



Training neural nets:

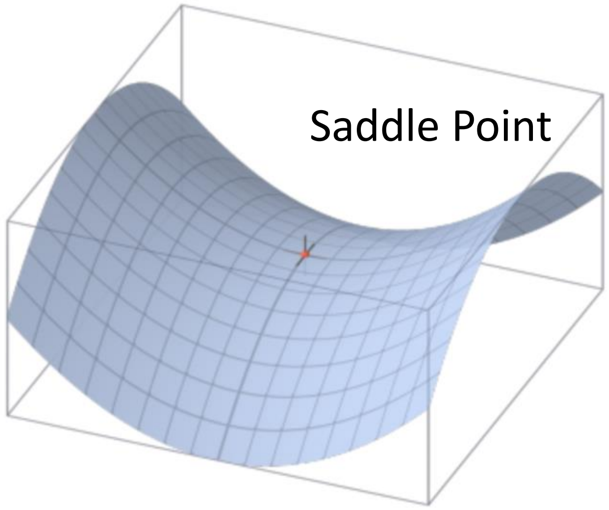
Loop until convergence:

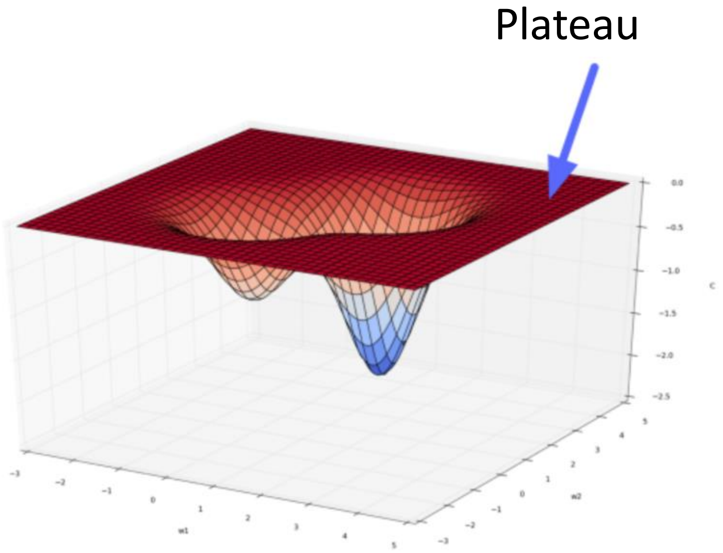
- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

- Use backpropagation to efficiently compute $\frac{\partial C}{\partial W}$ for each layer

Why is training neural networks so hard?

- Hard to optimize:
  - Not convex
  - Local minima, saddle point, plateau,…
  - Lots of parameters -- overfitting
  - Can take a long time to train





Why is training neural networks so hard?

- Architecture choices:
  - How many layers? How many units in each layer?
  - What activation function to use?
- Choice of optimizer:
  - We talked about gradient descent, but there are techniques that improves upon gradient descent
- Regularization?
- Initialization?

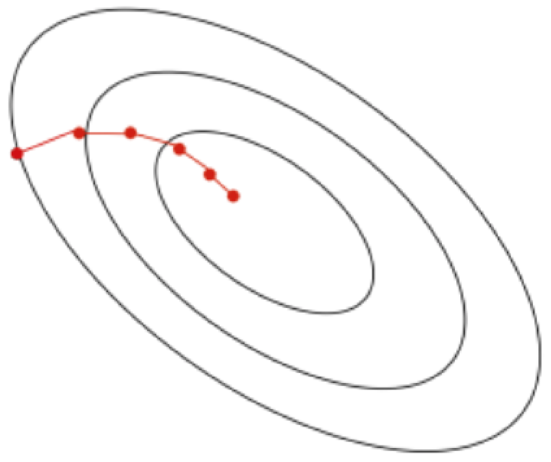
Contents:

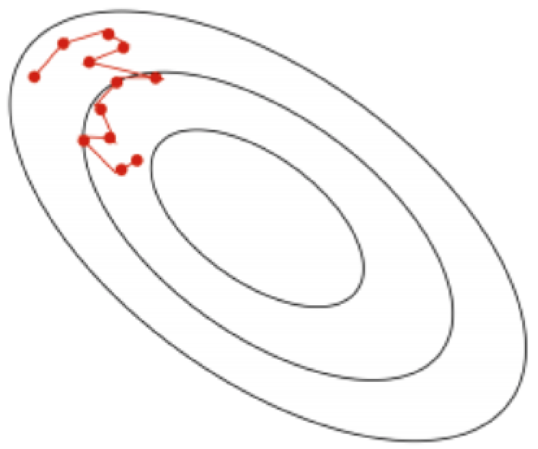
- Optimizers:
  - SGD, Momentum, adaptive learning rates
- Regularization:
  - L1 & L2
  - Dropout
- Activation Functions:
  - sigmoid, tanh, ReLU

Gradient Descent

- Compute the cost $C(\boldsymbol{o}, \boldsymbol{y})$ and gradients $\frac{\partial C}{\partial \boldsymbol{W}}$ using the <u>entire training set</u>
  - If we have large amounts of data, this is very inefficient
  - “Batch” gradient descent
- Stochastic Gradient Descent:
  - Sample a mini<u>batch</u> of data from the training set
  - Use the batch to estimate $\frac{\partial C}{\partial \boldsymbol{W}}$

Stochastic Gradient Descent





Stochastic Gradient Descent (SGD)

- Terminology:
    - **minibatch**: sample of training data used to estimate $\frac{\partial C}{\partial \boldsymbol{W}}$ for a parameter update
    - **batch size**: number of data points in a batch
    - **iteration**: one parameter update $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial C}{\partial \boldsymbol{W}}$
    - **epoch**: one pass of the full training set
- SGD typically implemented by randomly shuffling the training data into batches at the start of each epoch
- Different epochs would have different batches

Choosing minibatch size

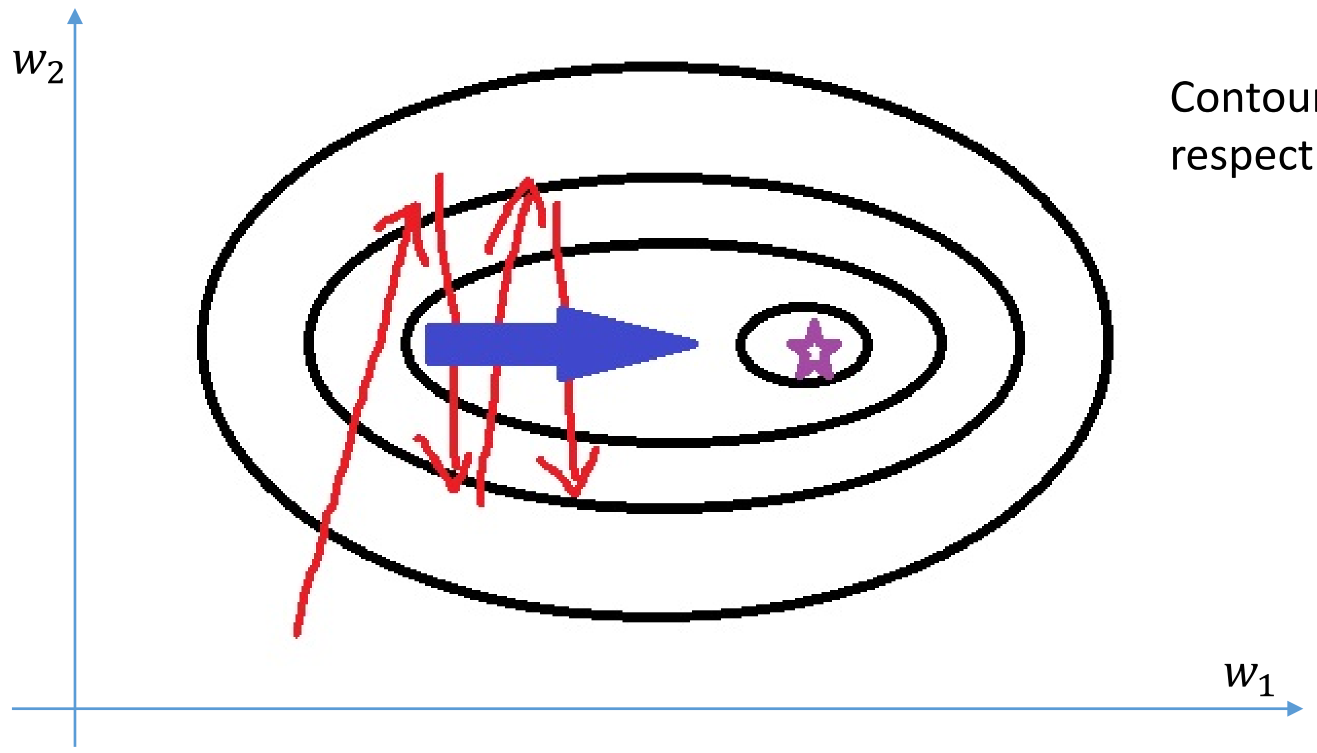
- If too large:
  - Training will be slow
  - If batch size = size of training set, then we recover (non-stochastic) gradient descent
- If too small:
  - Gradients will be very noisy
  - If batch size = 1, then we perform a parameter update for every item in the training set
- Typical minibatch sizes:
  - 8, 16, 32, 64, 128

Choosing learning rate

- If too large:
    - Can hinder convergence
- If too small:
    - Slow convergence
- Typically use a learning rate schedule:
    - Reduce learning rate according to a pre-defined schedule
        - E.g. cut learning rate by half every 10 epochs
    - Reduce learning rate when objective stops improving
        - E.g. when validation accuracy stops improving

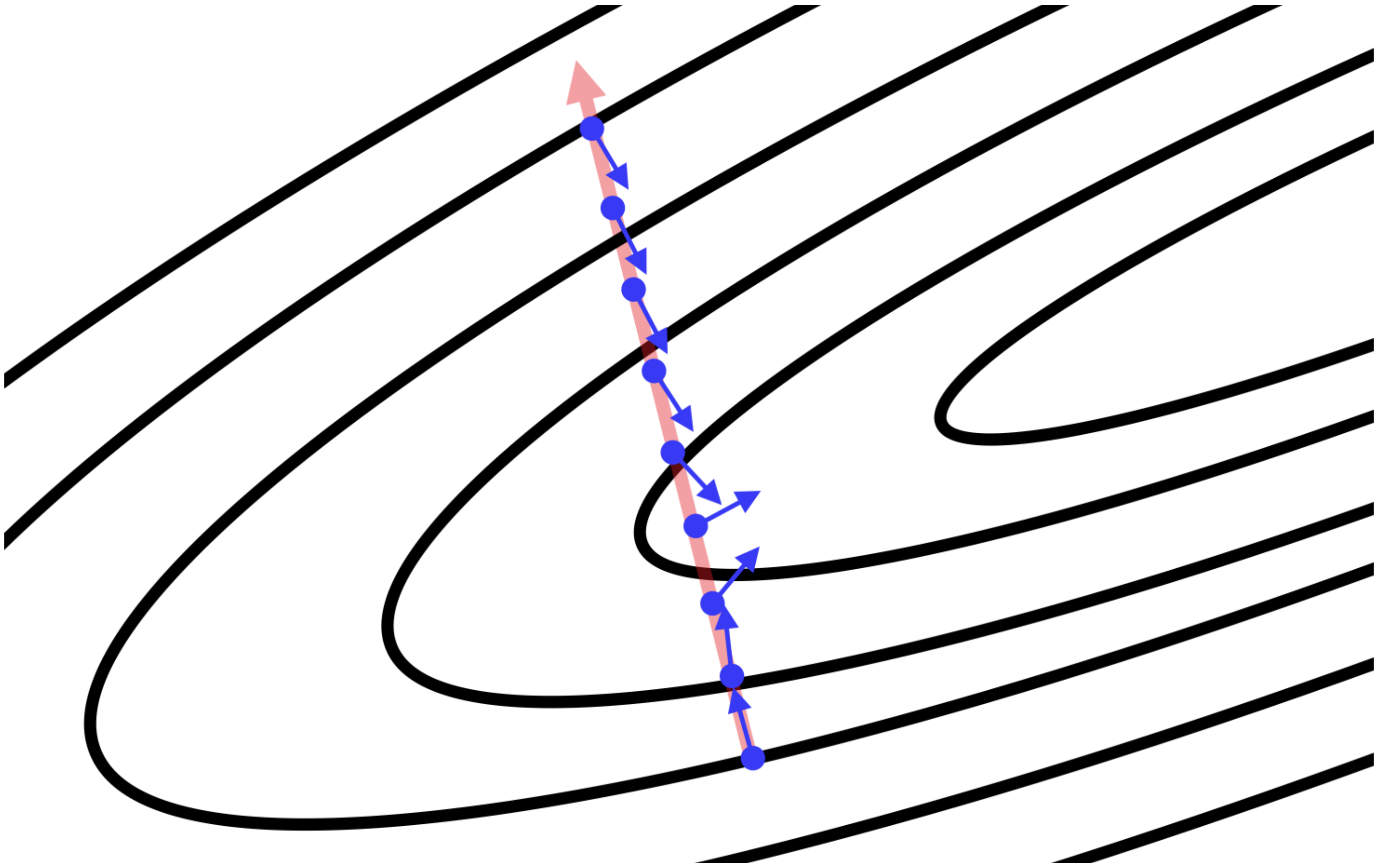
Weakness of (vanilla) Gradient Descent

- SGD has trouble navigating areas where surface curves more steeply in one direction



Contour plot of cost with respect to two weights.

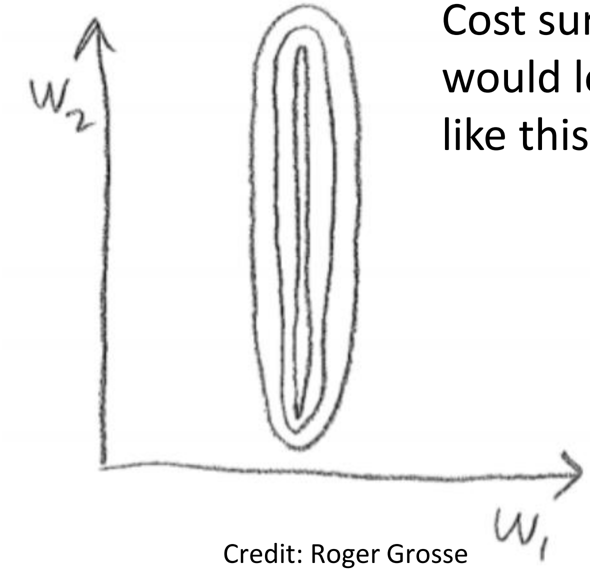
Weakness of (vanilla) Gradient Descent



Why do cost functions have this shape?

- Suppose we have the following dataset for linear regression:

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮



Cost surface would look like this.

Credit: Roger Grosse

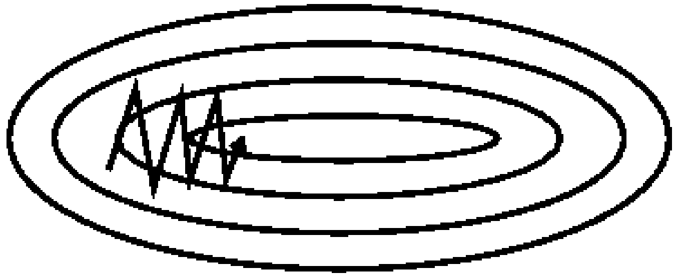
- Need to move w_1 by just a little bit, but need to move w_2 by a lot

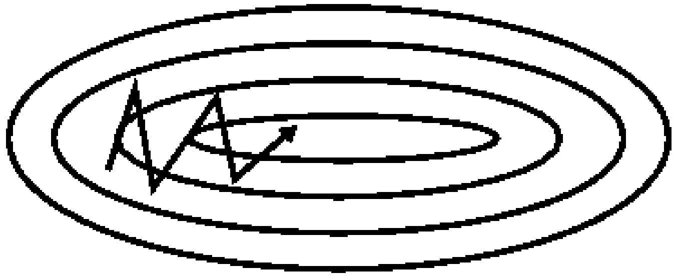
How to avoid the ravine problem

- For the input layer, we can center input
  - For example, to zero mean and unit variance
  - $x^{(i)} \leftarrow \frac{x-\bar{x}}{SD(x)}, sd(x) = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \bar{x})^2}$
- What about hidden units?
  - Difficult to avoid ravines in general

Momentum

Idea: if gradient consistently points toward one direction, go faster in that direction. (like intertia in physics)


No momentum


Momentum

Update rule for momentum

Parameter update rule for momentum:

$$\boldsymbol{v} \leftarrow \gamma \boldsymbol{v} + \alpha \frac{\partial C}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{v}$$

Where

γ is the momentum term (set to 0.9, 0.99, …)

α is the learning rate, like gradient descent

Why momentum works

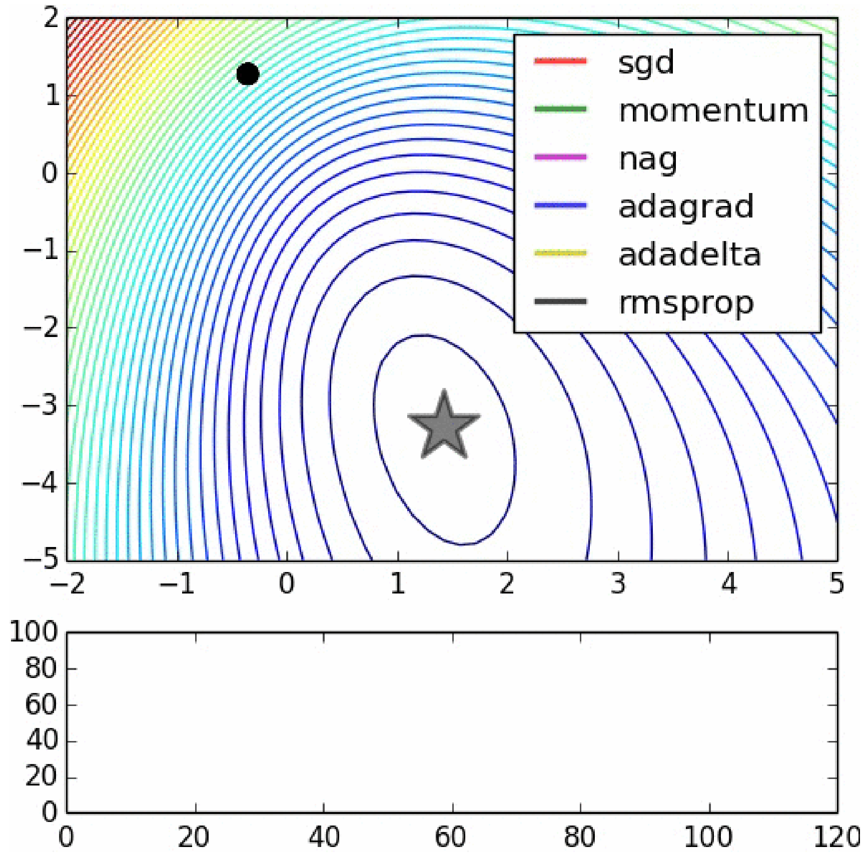
- Gradients of high curvature directions cancels out, so momentum dampens oscillation

- Gradient of low curvature directions accumulates, so momentum improves speed
- Momentum can help a lot, almost never hurts

Other Optimizers

- Why use same learning rate for all the weights?
- Adapt the learning rate to parameters
- This is the idea behind:
  - Adagrad
  - Adadelta
  - RMSProp
  - Adam
- Which is best? Depends on problem.

Optimizer comparison demo



- Alec Radford’s animations:
  http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Initialization

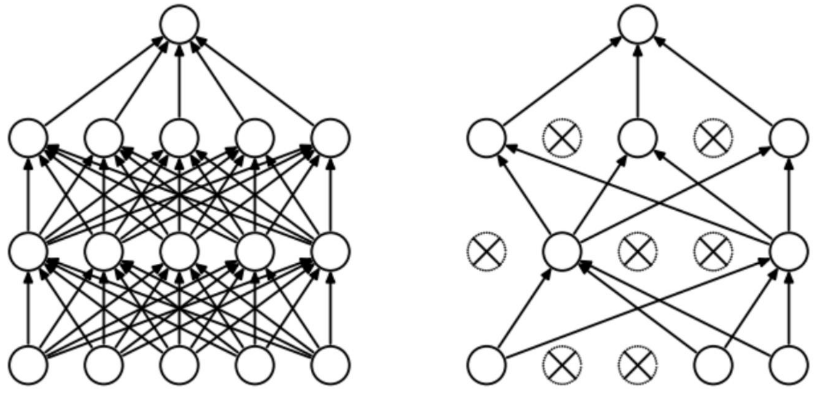
- How do we initialize weights?
- What if we initialize all to 0? (or a constant c?)
  - All neurons will stay the same!
- Standard approach:
  - Sample from a Gaussian
  - How to choose variance?
    - Xavier initialization: $\sigma^2 = \frac{2}{n_{in}+n_{out}}$
    - He initialization: $\sigma^2 = \frac{2}{n_{in}}$

Regularization

- Use L1 and L2 regularization as in regression
- Combination of L1 and L2 regularization also possible
- Dropout

Dropout

- During training, hidden units are set to 0 with probability $(1 - p)$

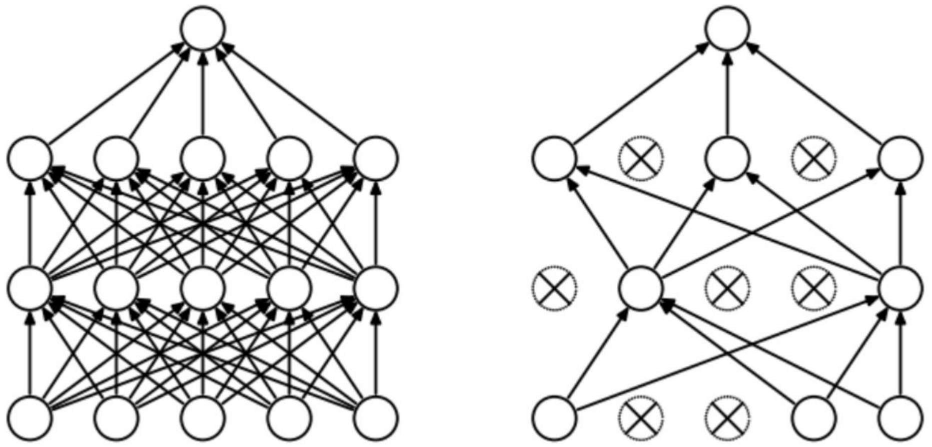


(a) Standard Neural Net (b) After applying dropout.

- When computing test outputs, scale all activations by the factor of p
    - Keeps the scale of the output consistent, and gives the right output in expectation

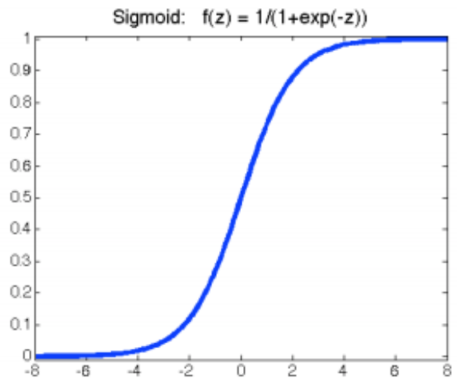
Why does dropout prevent overfitting?

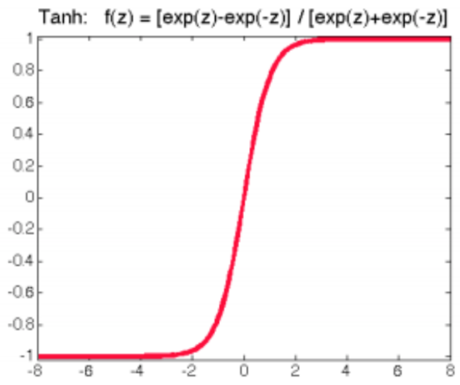
- Prevents dependence between units
  - Each unit must be “useful” independent of other units
  - We cannot learn a network that depends on complex activation patterns

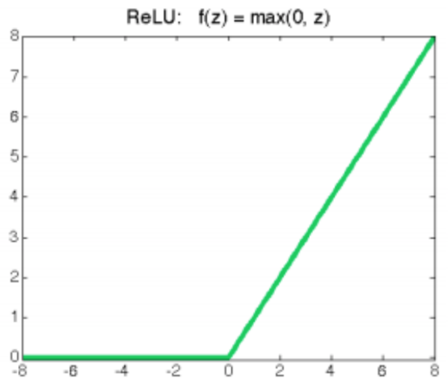

(a) Standard Neural Net (b) After applying dropout.

Activation functions

- Choices:
  - sigmoid, tanh, ReLU, …
- Recommendation:
  - don’t use sigmoid
  - Try ReLU first, and then tanh

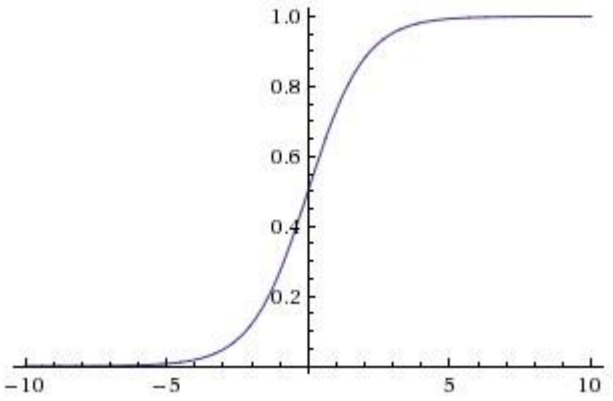






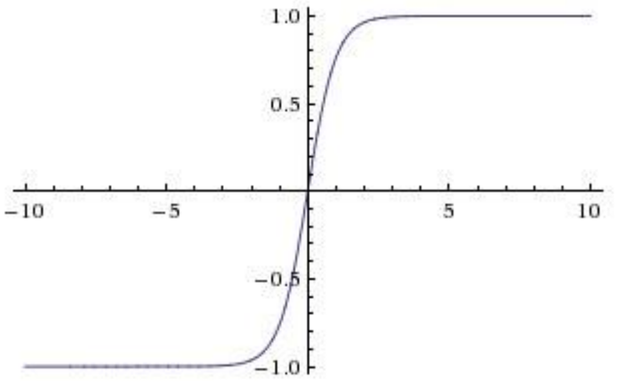
Sigmoid

- $\sigma(t) = 1/(1 + \exp(-t))$
- Disadvantages:
  - $\sigma'(t)$ is very small for t outside of t $\in [-5, 5]$
    - If that happens, the neuron "dies": the weights below the neuron won´t change, and so the value of the neuron remains fixed (since any change to the weights is multiplied by $\sigma'(t)$
  - $\sigma(t)$ is always positive
    - *All* the weights will either move in the positive direction or the negative direction during a given step of gradient descent



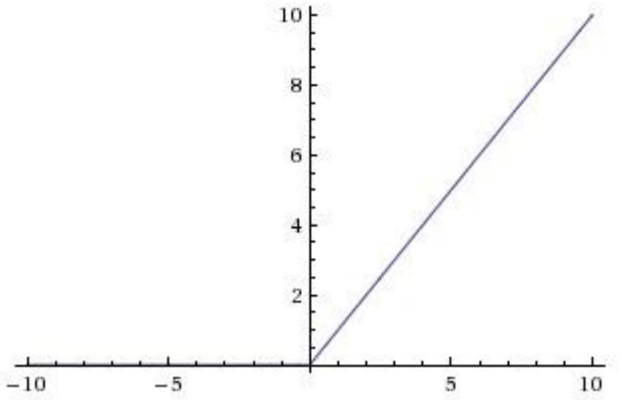
Tanh

- $\tanh(t) = \frac{1-\exp(-2t)}{1+\exp(-2t)}$
- (=$2\sigma(2t) - 1$)
- Not always positive
  - No problem with all the weights having to move in the same direction
  - Advantage over the sigmoid



ReLU

- Rectified Linear Unit
- $f(t) = \max(0, t)$
- Works well if you're careful – better than others (but needs care!)
- Cheap to compute
- Dies if t is too small
  - No gradient at all!

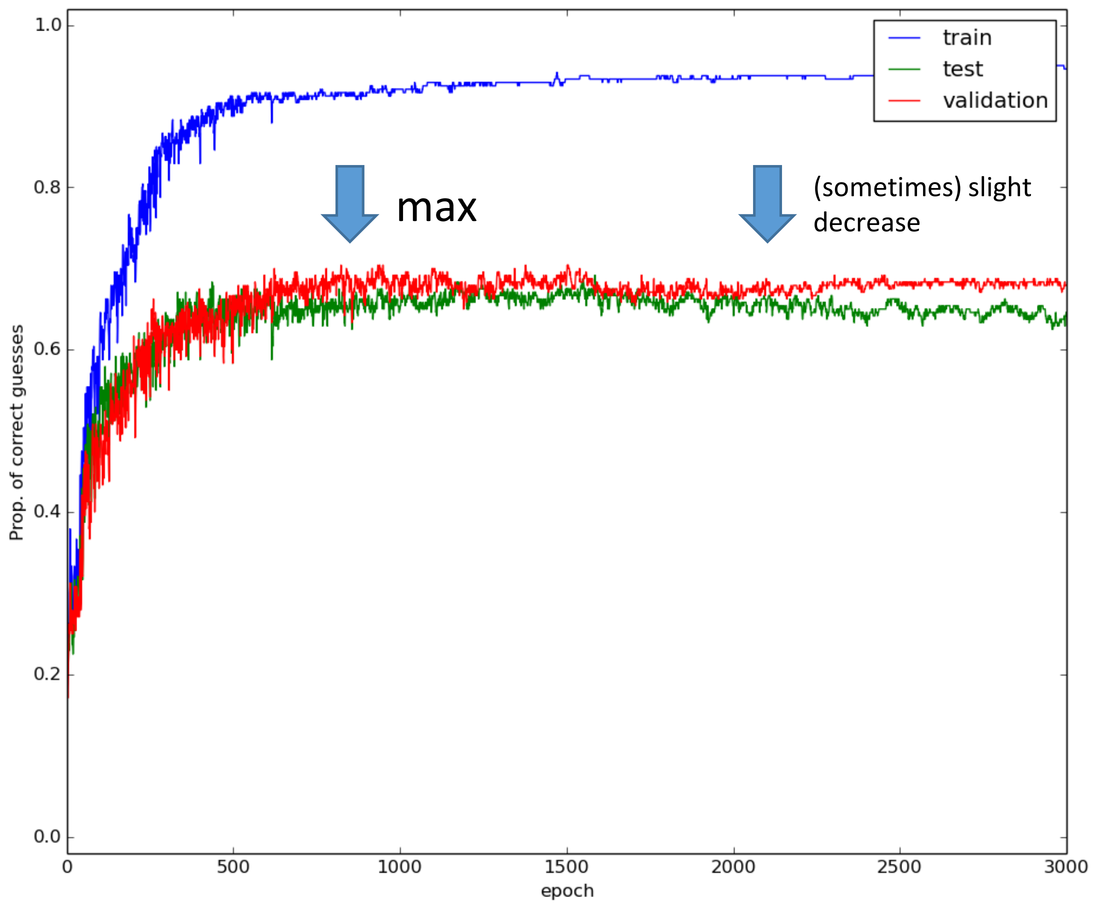


Learning Curves

- Split the data into a training set, validation set, and test set.
- Minimize the cost function on the training set, measure performance on all sets
- Plot the performance on the three sets vs. the number of optimization iterations performed
  - Optimization iteration i:

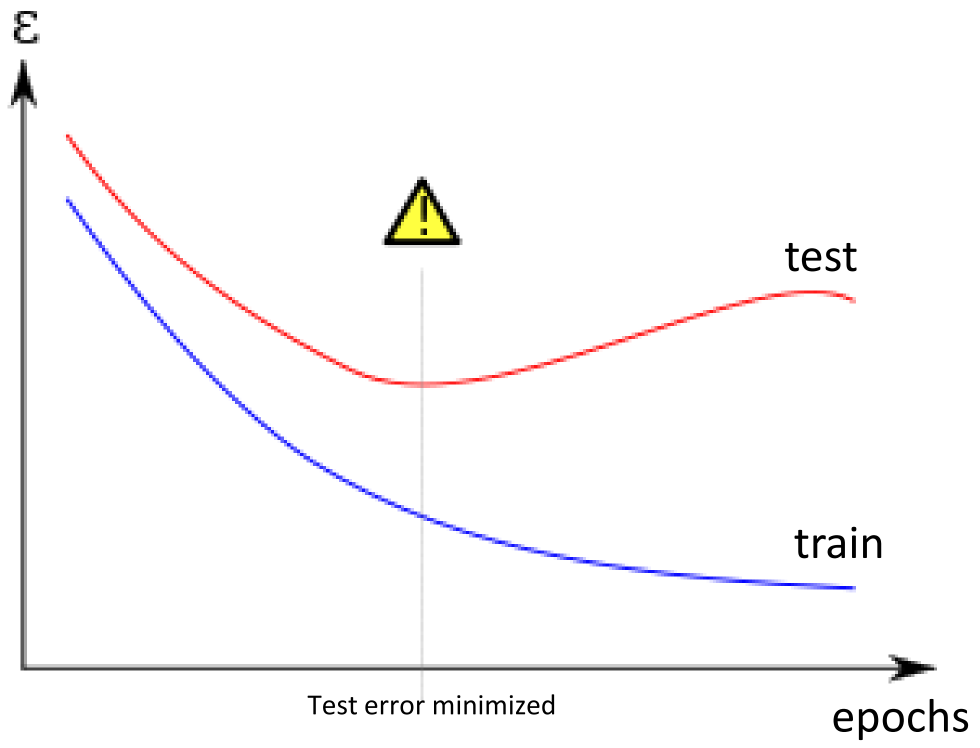
$$\theta_{i+1} \leftarrow \theta_i - \alpha \nabla cost(\theta, x_{train}, y_{train})$$

“Typical” Learning Curves



300-unit hidden layer. 6 people, 80 examples each. Best test performance: 68%

Wikipedia version



(Basically a fairytale: the moral of the story is true, but things rarely look this nice)

Learning Curves

- The training performance always increases
  - (Assuming the performance is closely enough related to the cost we're optimizing – we sometimes also plot the cost directly)
- The test and validation curve should be the same, up to sampling error (i.e., variation due to the fact that the sets are small and sometimes things work better on one of them by chance)
- The training and validation performance sometimes initially increases and then decreases as we minimize the cost function

Overfitting and Learning Curves

- If performance decreases on the validation set but increases on the training set, that is an indication that we are overfitting
  - Predicting outputs based on the peculiarities of the training set rather on the genuine patterns in the data