Designing, Visualizing and Understanding Deep Neural Networks

Lecture 4: Optimization and Backpropagation

CS 182/282A Spring 2019
John Canny

Last Time: Bias-Variance Tradeoff

Bias at a point x is the expected difference between predictions and the true y.

i.e. $$Bias\left(\hat{f}(x)\right) = \mathrm{E}\left[\hat{f}(x) - f(x)\right] = \bar{f}(x) - f(x)$$

where $\bar{f}(x) = \mathrm{E}\left[\hat{f}(x)\right]$ is the expected prediction at x

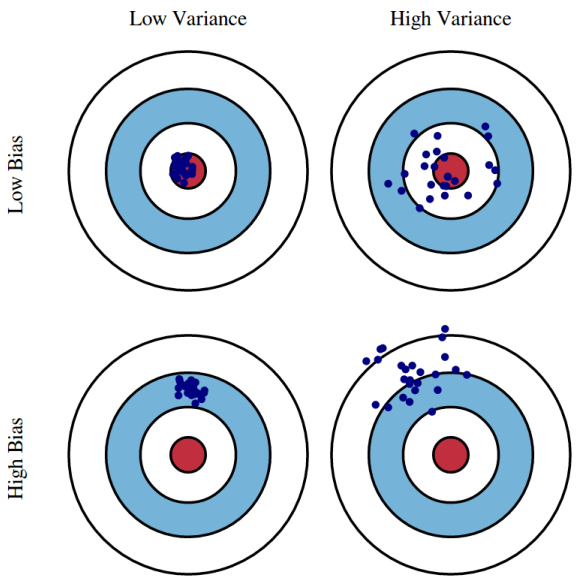
Variance is the variance of the predictions:

$$Variance\left(\hat{f}(x)\right) = \mathrm{E}\left[\left(\hat{f}(x) - \bar{f}(x)\right)^2\right]$$

The linear least-squares error decomposes as:

$$Bias^2 + Variance$$

Tradeoff: Can reduce bias at the expense of higher variance and vice versa.



Last Time: Regularization

L2 regularization is widely used with deep networks (its commonly implemented as "weight decay" in SGD optimizers). For multivariate linear regression, the regularized loss is:

$$L(A) = \sum_{i=1}^{n} (x_i^T A^T - y_i^T)(Ax_i - y_i) + \lambda \sum_{i,j} A_{ij}^2$$

Like other forms of regularization, L2 regularization allows you to trade off bias and variance.

Strong regularization (large λ) ➔ **lower variance** and **higher bias**

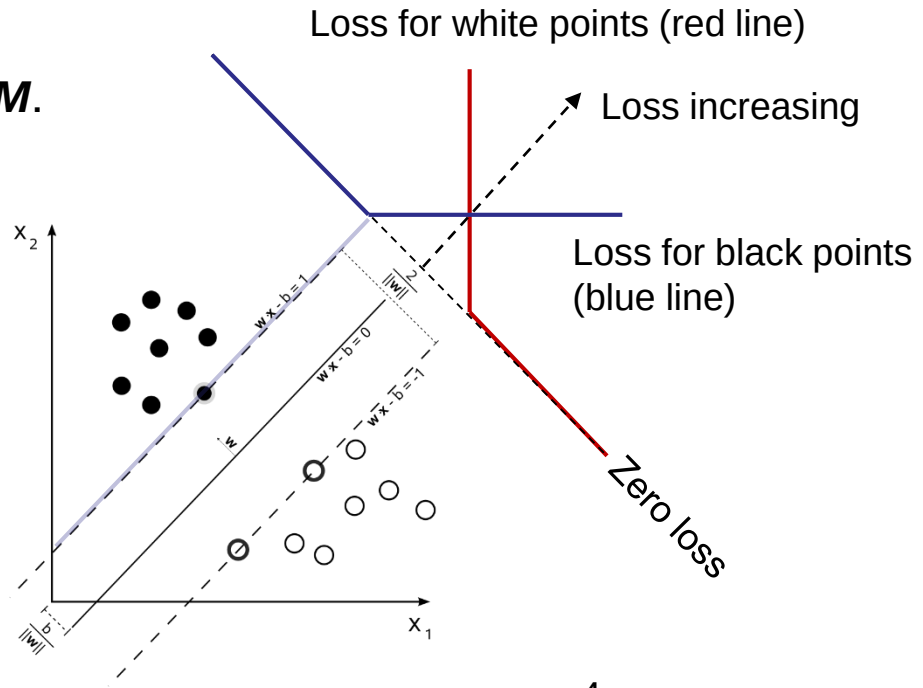
Weak regularization (small λ) ➔ **higher variance** and **lower bias**

Last Time: Hinge Loss and SVMs

Hinge loss penalizes data points that lie inside the margin around the decision boundary

$$L = \max(0, 1 - yf(x))$$

A linear classifier that minimizes hinge loss is called a ***Support Vector Machine or SVM***.



Last Time: Multi-Class Classification

Multiclass SVM loss uses the difference between each class score and the correct class score. It measures how much this difference fails to meet a margin (1 here):

$$L = \max\left(0, 1 - \left(f_y(x) - f_j(x)\right)\right)$$

where $f(x) = Wx$.

Multiclass Logistic regression estimates the class target probability with a softmax function:

$$f_j(x) = \frac{\exp(s_j)}{\exp(s_1) + \exp(s_2) + \cdots + \exp(s_k)}$$

where $s = Wx$.

And minimizes the cross-entropy loss which is $-\log f_y(x)$

Reminder – Assignment 1

Assignment 1 is out, due on Feb 19th, 11pm.

- Uses python + ipython. Please check right away that you have a working installation of python 2.7 and ipython2, and that you can load and execute the assignment notebooks.
- Python virtualenv is your best bet, but there are tricks to doing it on a Mac.
- Don't assume that just because python/ipython used to work on a given machine, that it still does. e.g. I had to uninstall and reinstall pyzmq on a machine that used to work…

- The assignment itself closely tracks the lecture material over the next couple of weeks, so you'll get best value by doing it in stages.

Issue Reporting

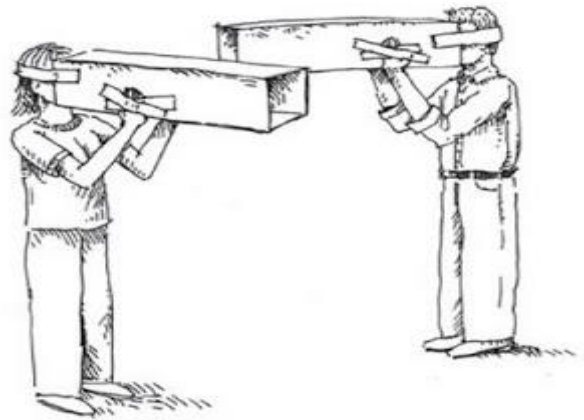
Please use best issue reporting practices for Python or other system issues:

https://bcourses.berkeley.edu/courses/1478831/pages/reporting-an-issue

- Give background info: your system, OS version, software version.
- Show what you saw (complete error message, not summary), and what you expected.
- Exact steps to reproduce, not “I followed these directions <<link>>”.
- Structured issue reporting is required in many companies, and its an important skill to have.
- It will also help you to maintain good relationships with system support staff…

Issue Reporting Principles

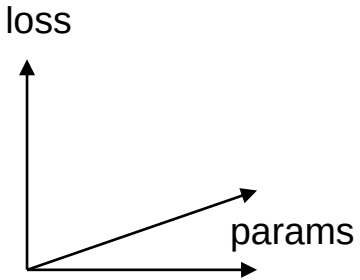
- Complete issue reports reduce the diagnostic cycle by 2-5x.
- Most simple issues are resolved in a single cycle from good reports.
- Don’t rely on “Hashing” i.e. that the error message uniquely specifies the problem – often it doesn’t.
- Avoid “Tunnel Vision” – i.e. don’t make assumptions about the cause of the error if you cant fix it.
  Provide ***all*** the relevant information.

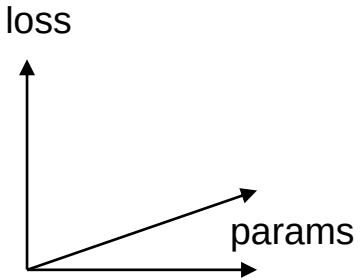


Responding to Others' Issue Reports

- The goal should be to help the poster solve their own problem, not to solve it for them.

This Time: Optimization


loss
params


loss
params

Method 1: Search

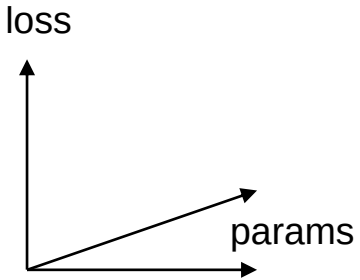
Take several random steps and measure the altitude (loss). Then go to the lowest one.

- Not totally crazy. This is one kind of “gradient-free” optimization. It makes sense if you can’t compute gradients for some reason (e.g. loss is not continuous or differentiable).
- A slightly smarter version takes an average of the random points weighted by altitude (deeper points get higher weight). This in fact approximates the gradient.

See:

Salimans, Ho, Chen, Sidor and Sutskever “Evolution Strategies as a Scalable Alternative to Reinforcement Learning” arXiv 1703.03864, 2017.

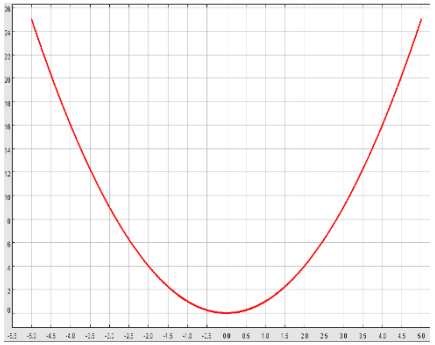
But it’s less efficient than gradient descent when gradients are available.


loss
params

Losses we have seen so far

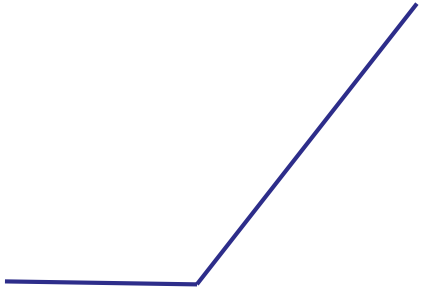
Squared Loss

$L = \left(y_i - f(x_i)\right)^2$



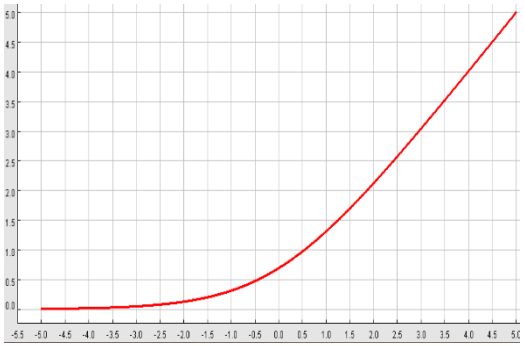
Hinge Loss, $y_i \in \{-1,1\}$

$L = \max(0,1 - y_i f(x_i))$



Cross-entropy loss on logistic function, $y_i \in \{-1,1\}$

$L = \log(1 + \exp(-y_i f(x_i)))$



All three have “well behaved” derivatives. $f(x) = w^T x$ is a linear function of the weights W, so we can differentiate loss with respect to weights.

Gradients Again

When we write $\nabla_W L(W)$, we mean the vector of partial derivatives wrt all coordinates of W:

$$\nabla_W L(W) = \left[\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \ldots, \frac{\partial L}{\partial W_m}\right]^T$$

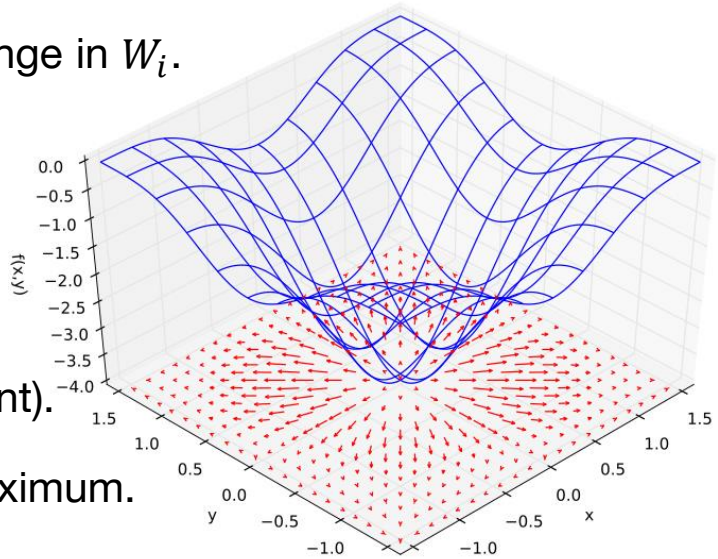
Where $\frac{\partial L}{\partial W_i}$ measures how fast the loss changes vs. change in W_i.

In figure: loss surface is blue, gradient vectors are red:

When $\nabla_W L(W) = 0$, it means all the partials are zero.
i.e. the loss is not changing in any direction.

Thus we are at a local optimum (or at least a saddle point).

Note: arrows point out from a minimum, in toward a maximum.



Gradient Descent

So to reach a minimum of loss, we should follow the negative gradient.

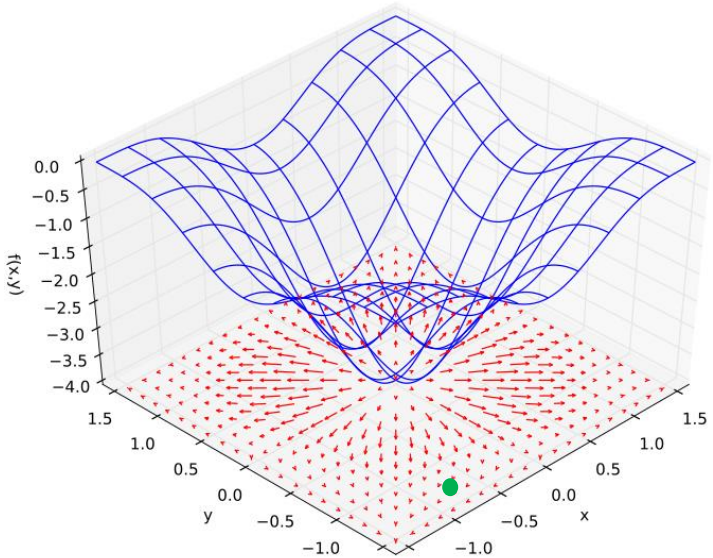
i.e. we should take small steps in direction

$$-\nabla_W L(W)$$

To be more concrete, let W^t denote the weights at step t of gradient descent. Then

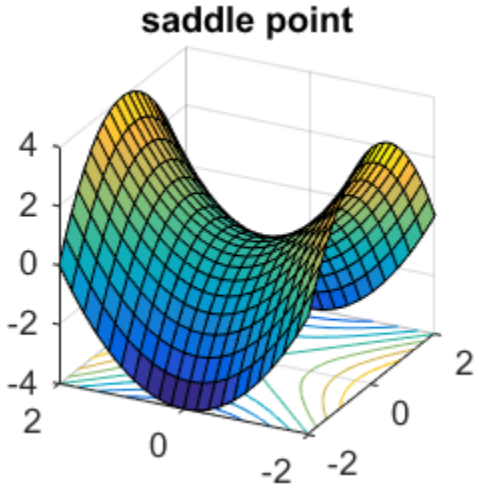
$$W^{t+1} = W^t - \alpha \ \nabla_W L(W)$$

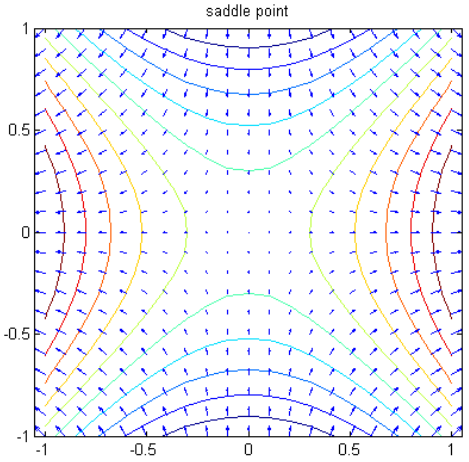
Where α is called the ***learning rate***.



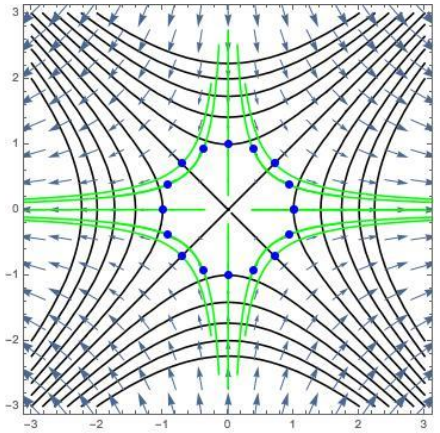
Caveat: Saddle Points

Following the negative gradient should eventually get you to a loss minimum. But it may take a long time. One reason is the presence of saddle points, where the gradient also vanishes:





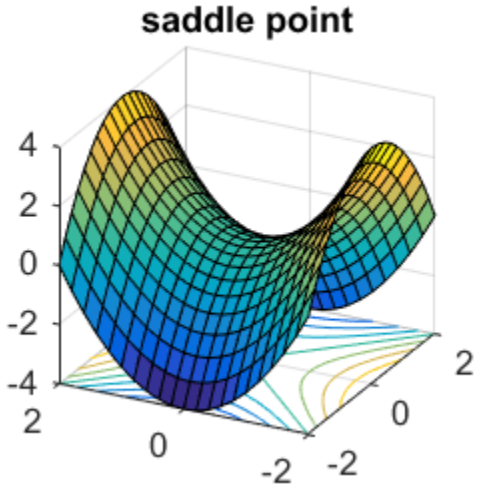
Gradient vectors

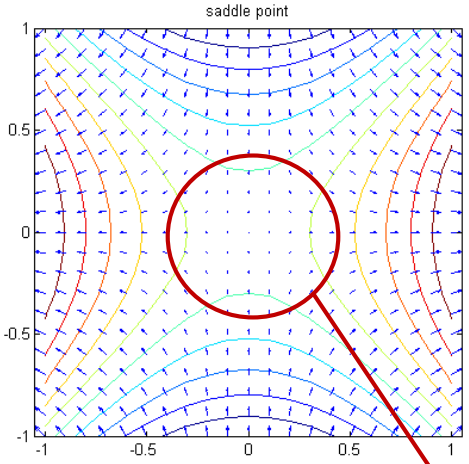


Gradient flows (green)

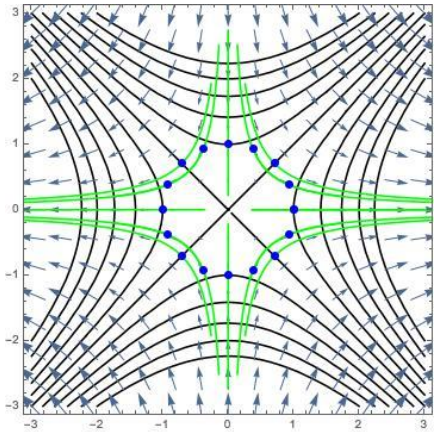
Caveat: Saddle Points

Following the negative gradient should eventually get you to a loss minimum. But it may take a long time. One reason is the presence of saddle points:





Gradient vectors



Gradient flows (green)

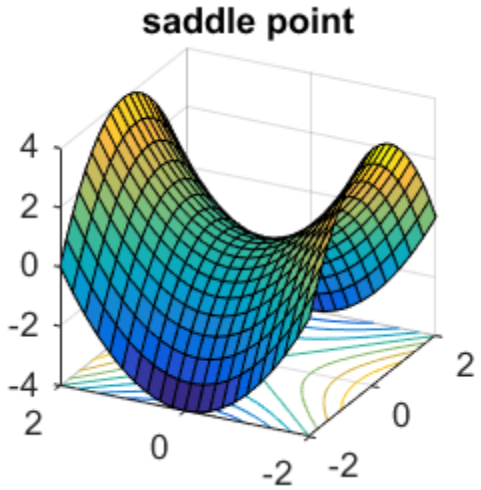
Gradients very small in here
– slow progress

Caveat: Saddle Points

There is a lot of evidence that ***most zeros of the loss gradient*** $\nabla_W L$ for neural networks ***are in fact saddles***.

See e.g.

Dauphin et al. “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization” arXiv 1406.2572, 2014.



Caveat: Efficiency

The loss L is a sum of the losses for all the data items:

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

and

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

so computing the gradient wrt W requires a ***full pass through the dataset***. Getting to the loss minimum may require millions of gradient steps, so this is ***very*** expensive.

Minibatches

Instead of computing a gradient across the entire dataset, we can compute it using a fixed-size subset of data samples called a ***minibatch***. The minibatch size m is typically 32, 64, 128, 256,…

The minibatch is ideally a random sample of size m from the full dataset. In practice it may just be m consecutive samples.

So we compute this gradient (N is our dataset size, m is our minibatch size):

$$g^{(t)} = \frac{1}{m} \sum_{j = i_1,\ldots,i_m \in \{1,\ldots,N\}} \nabla_W L\left(x_j, y_j, W\right)$$

And then (superscripts are iteration numbers):

$$W^{(t+1)} = W^{(t)} - \alpha\, g^{(t)}$$

In this way, we perform N/m updates to the weights for one pass over the dataset.

Stochastic Gradient Descent

This approach is called ***Stochastic Gradient Descent*** or SGD. SGD and its variants are used almost universally in deep network training.

SGD uses $g^{(t)}$, the gradient of a minibatch instead of the true average gradient (call it g) on the full dataset. It should be clear that:

$$g = \mathbb{E}[g^{(t)}]$$

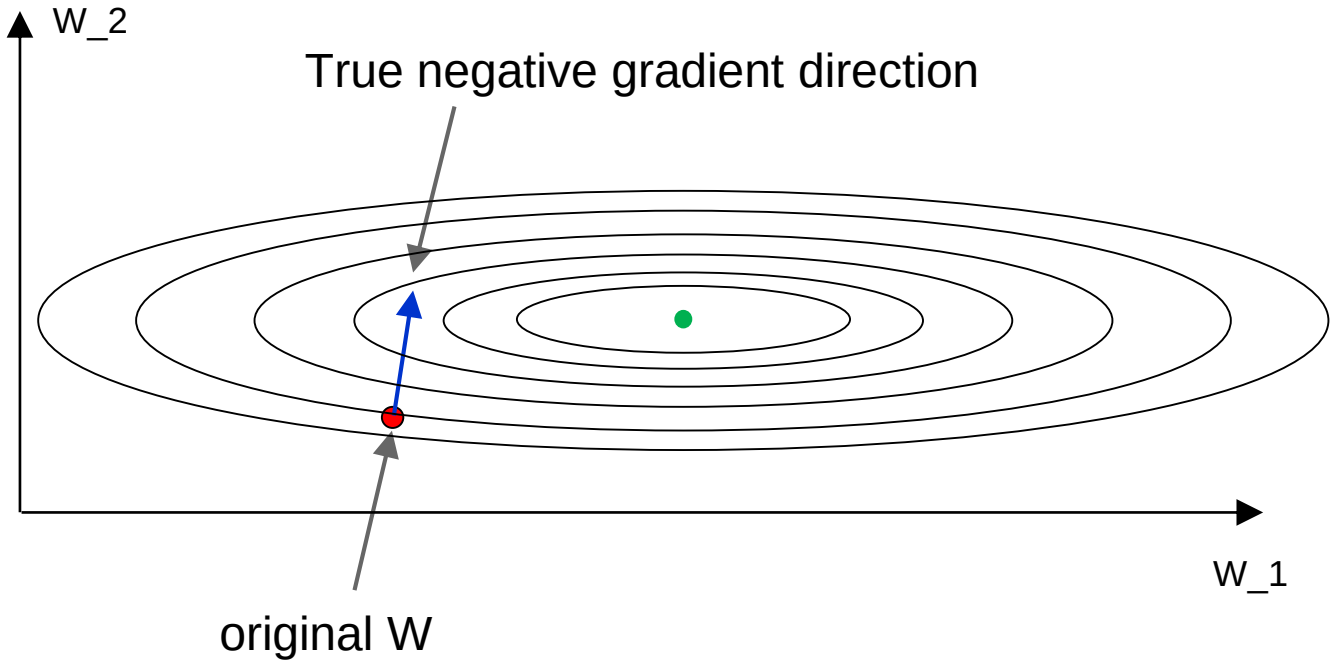
where expectation is over minibatches sampled from the full dataset.

So that $g^{(t)}$ is an unbiased estimate of g. But $g^{(t)}$ will typically have a lot of variance (is noisy) compared to g (if we considered the full dataset as a sample of an infinite dataset…).

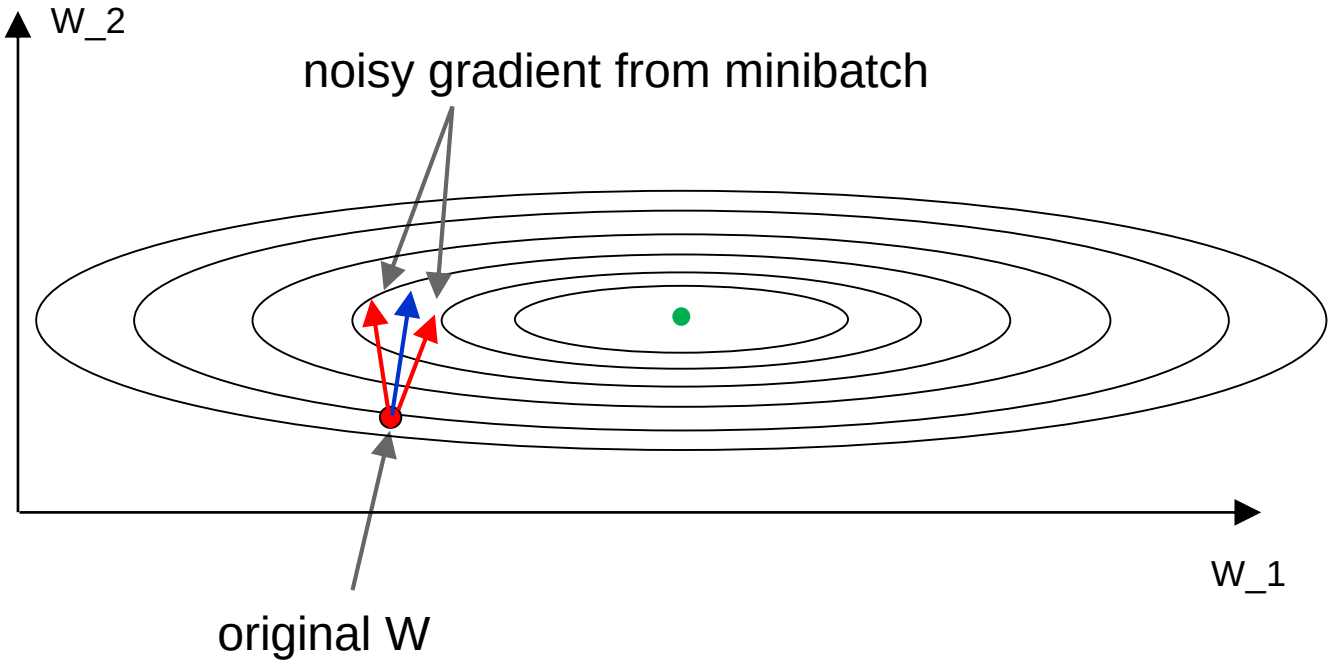
SGD is "stochastic" because it uses $g^{(t)}$, the gradient of a minibatch instead of the true gradient (call it g) on the full dataset.

Minibatch updates

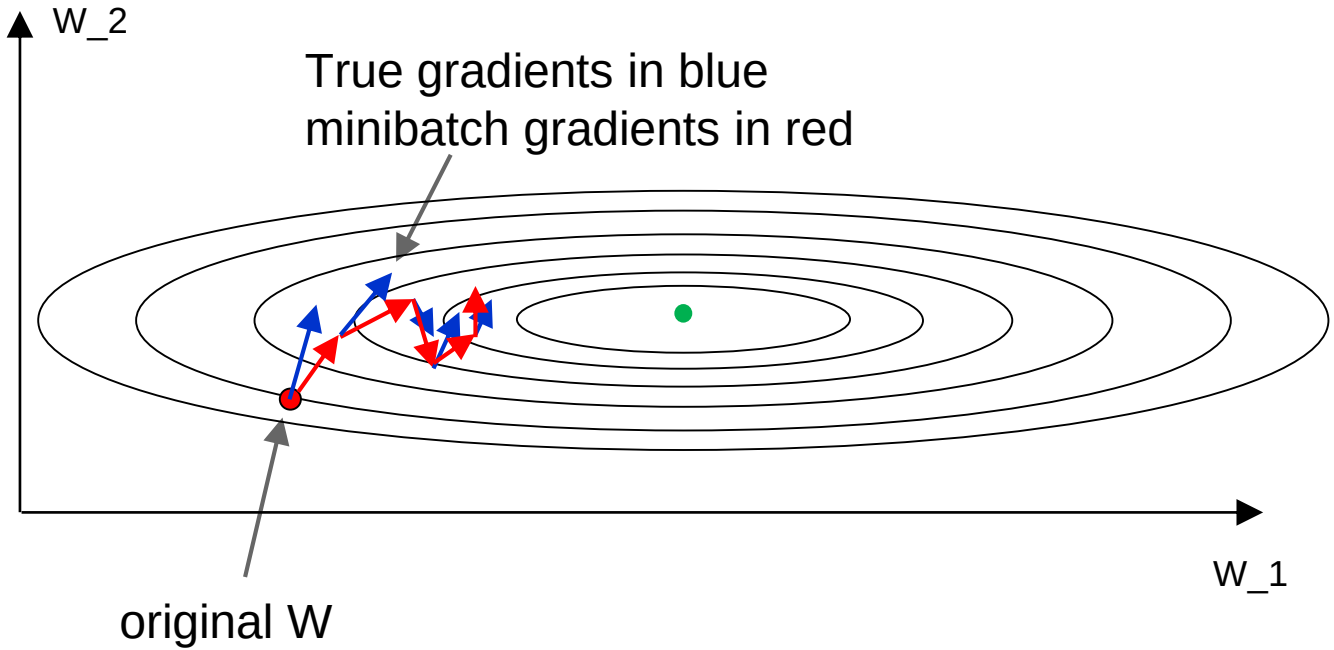
A contour plot represents a function with contours of equal value $f(x) = c$. The gradient of the function is always orthogonal to the contour.



Stochastic Gradient

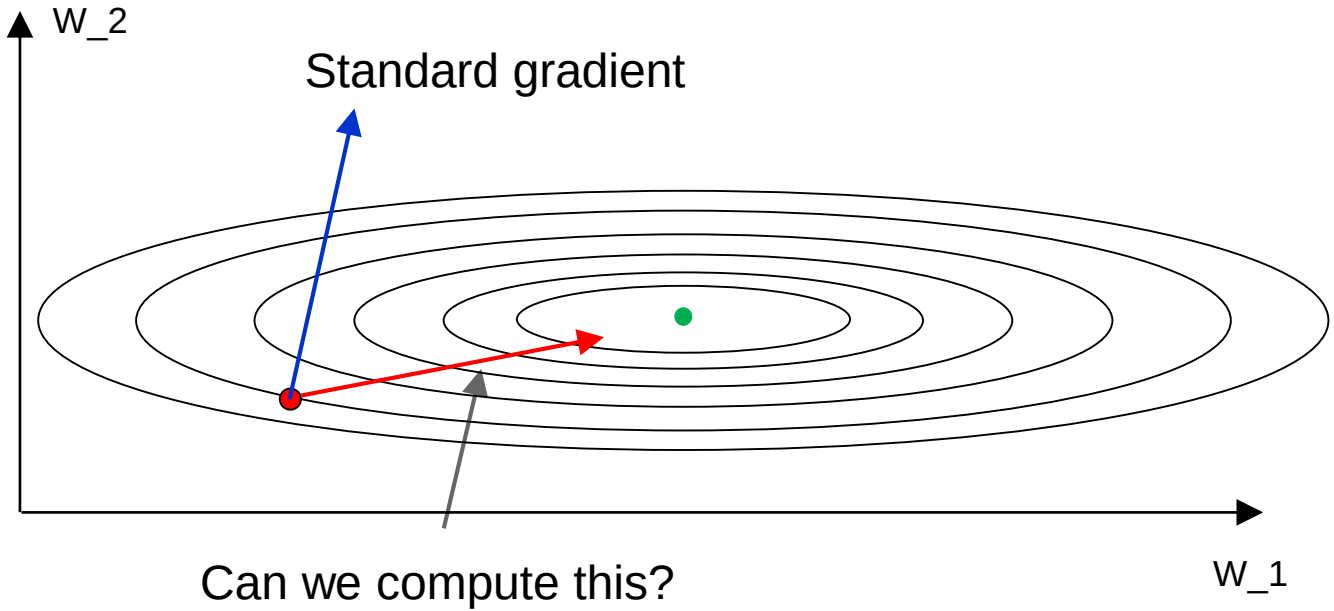


Stochastic Gradient Descent



Gradients are noisy but still make good progress on average

You might be wondering…



Yes: with Newton’s method

Newton's method for zeros of a function

Based on the Taylor Series for $f(x+h)$:

$$f(x+h) = f(x) + hf'(x) + O(h^2)$$

To find a zero of f, assume $f(x+h) = 0$, so

$$h \approx -\frac{f(x)}{f'(x)}$$

And as an iteration:

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$$

Newton's method for optima of a scalar function

For zeros of $f(x)$:

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$$

At a local optima, $f'(x) = 0$, so we use:

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$$

If $f''(x)$ is constant (f is quadratic), then Newton's method finds the optimum in ***one step***.

More generally, Newton's method has ***quadratic (very fast) converge.***

Newton's method for gradient zeros:

To find an optimum of a function $f(x)$ for high-dimensional x, we want zeros of its gradient: $\nabla f(x) = 0$

For zeros of $\nabla f(x)$ with a vector displacement h, Taylor's expansion is:

$$\nabla f(x+h) = \nabla f(x) + h^T H_f(x) + O\left(||h||^2\right)$$

where H_f is the Hessian matrix of second derivatives of f.

We can compute the "best" update by ignoring the $O(\left|\left|h\right|\right|^2)$ term, setting LHS to zero, and solving for h:

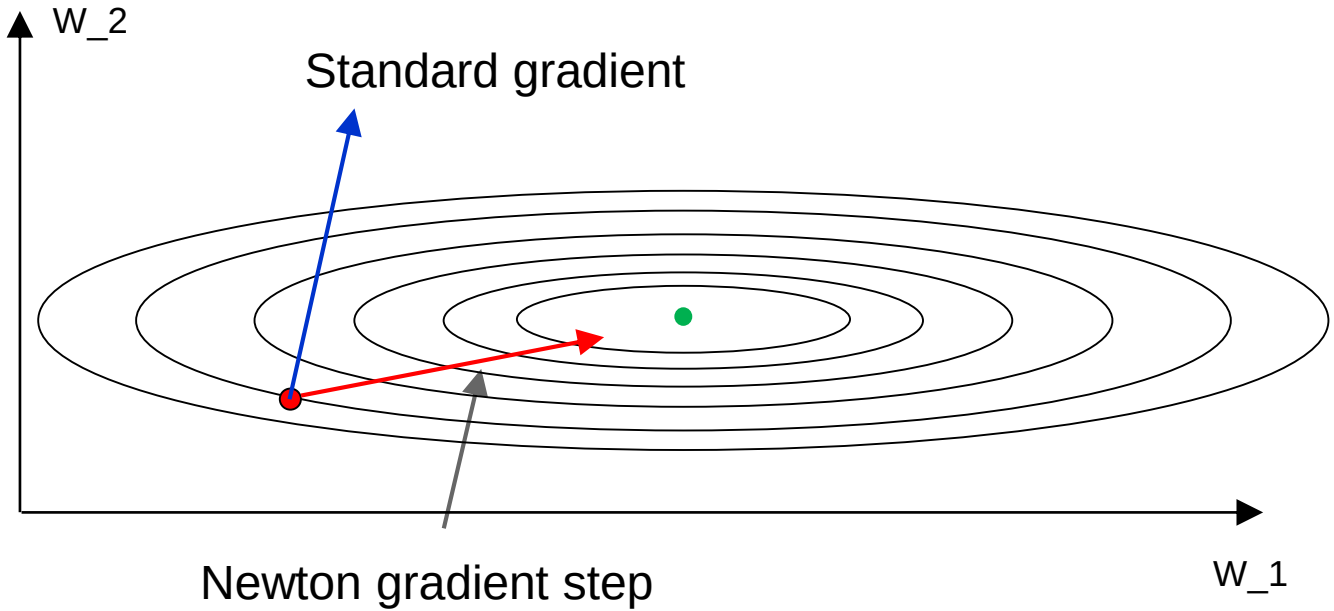
$$x_{t+1} = x_t - H_f(x_t)^{-1}\nabla f(x_t)$$

Aside: Hessian matrices

The Hessian for a function $f(x)$ is the matrix of 2nd order partial derivatives:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \qquad \text{or} \quad H = \nabla_x (\nabla_x f)^T$$

Newton step



Newton's method for gradient zeros

The Newton update is:

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

Converges very fast, but rarely used in Deep Learning.

Why do you think this is?

Newton's method for gradient zeros:

The Newton update is:

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

Converges very fast, but rarely used in Deep Learning. Why?

Too expensive: if x_t has dimension M, the Hessian $H_f(x_t)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Newton's method for gradient zeros:

The Newton update is:

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

Converges very fast, but rarely used in Deep Learning. Why?

Too expensive: if x_t has dimension M, the Hessian $H_f(x_t)$ has dimension M^2 and takes $O(M^3)$ time to invert.

We can address this to some extent with more advanced methods like L-BFGS which uses a K-dimensional approximation: $O(MK^2)$

Newton's method for gradient zeros:

The Newton update is:

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

Converges very fast, but rarely used in Deep Learning. Why?

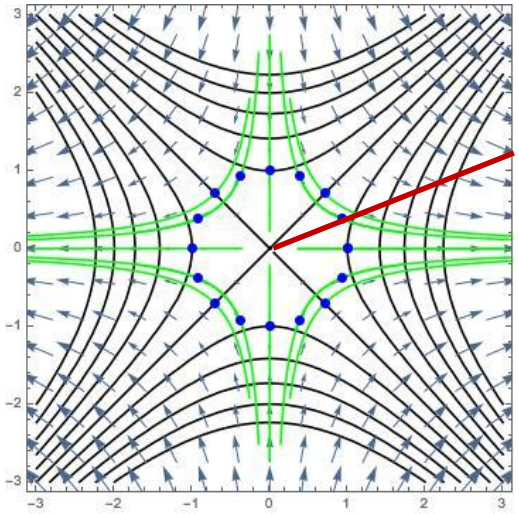
Too expensive: if x_t has dimension M, the Hessian $H_f(x_t)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Too unstable: Because it involves a matrix inverse it can be unstable numerically. Again advanced methods like L-BFGS are more stable.

But there is another ***big*** problem…?

Newton’s method for gradient zeros:

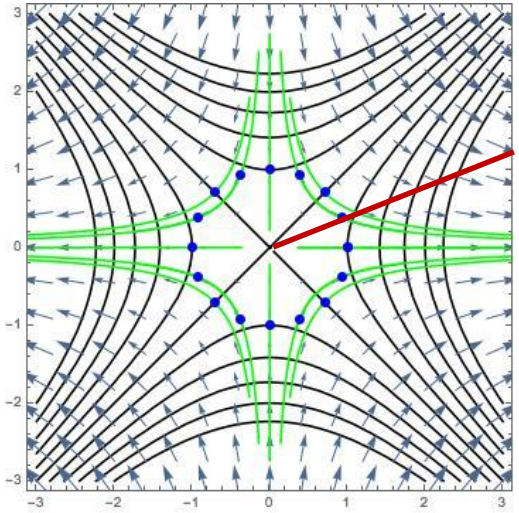
Too clever (gets stuck): The second-order terms in Newton’s method allow it to quickly get to the nearest gradient zero, ***including saddle points***.



Newton gradient flow (red)

Newton’s method for gradient zeros:

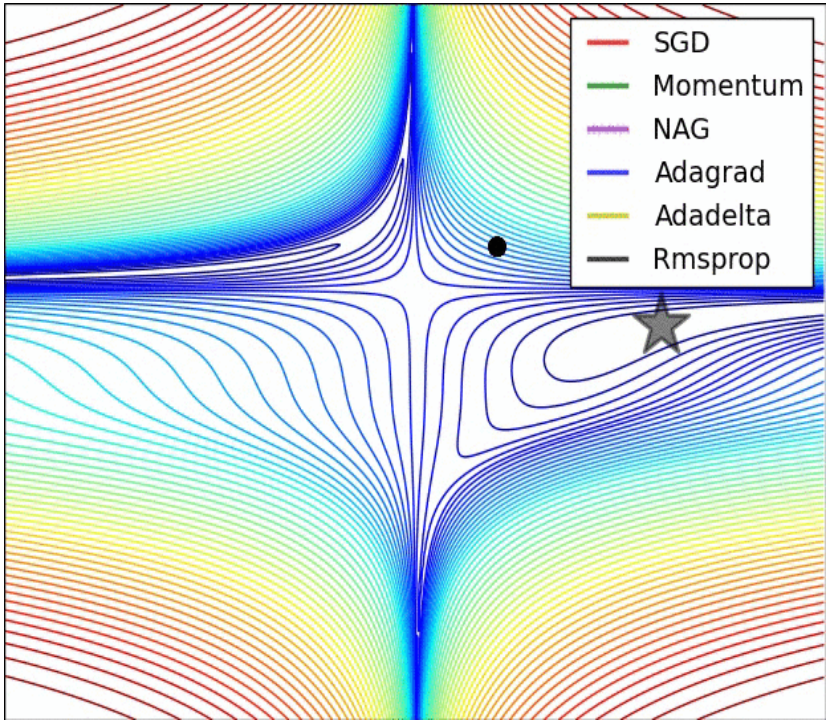
Too clever (gets stuck): The second-order terms in Newton’s method allow it to quickly get to the nearest gradient zero, ***including saddle points***.



Newton gradient flow (red)

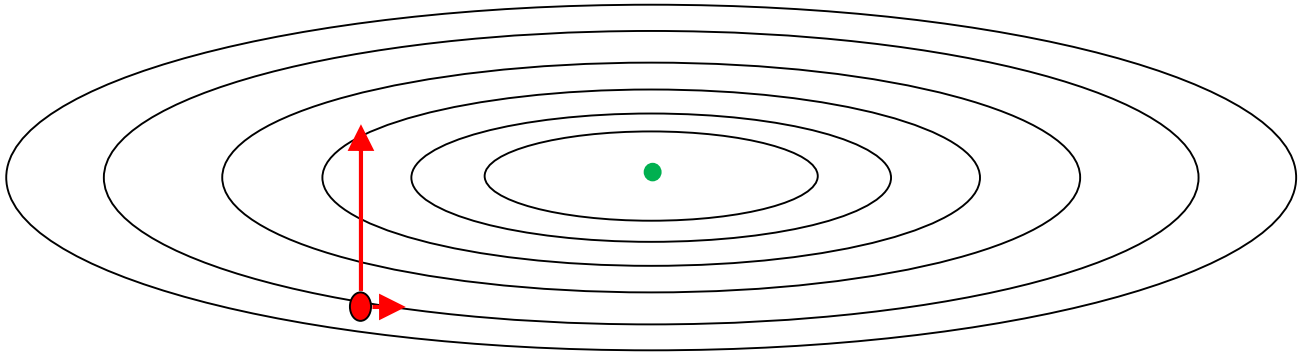
In fact we know that neural loss landscapes have lots of saddle points because people found them with Newton’s method.

The effects of different update formulas



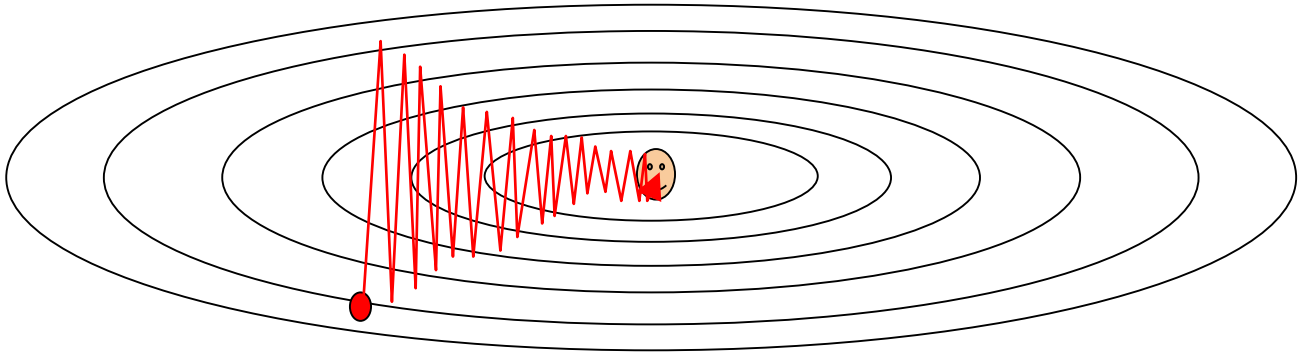
(image credits to Alec Radford)

Another approach to poor gradients:



Q: What is the trajectory along which we converge towards the minimum with SGD?

Another approach to poor gradients:



Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one

SGD with Momentum

“Every body persists in its state of being at rest or of moving uniformly straight forward, except insofar as it is compelled to change its state by force impressed”

– Isaac Newton

The object’s “memory” of its motion state is ***momentum***.

As commonly used in Deep Learning systems, the “momentum” parameter is actually the “momentum decay rate per minibatch”.

(And the physical analog is viscosity)

SGD with Momentum

Momentum update for step i :

$$p^{(t+1)} = \mu p^{(t)} - \alpha g^{(t)}$$

Where $p^{(t)}$ is the momentum, $g^{(t)}$ is the minibatch gradient, $\mu \in [0,1]$ is the “momentum” constant.

The weight update is:

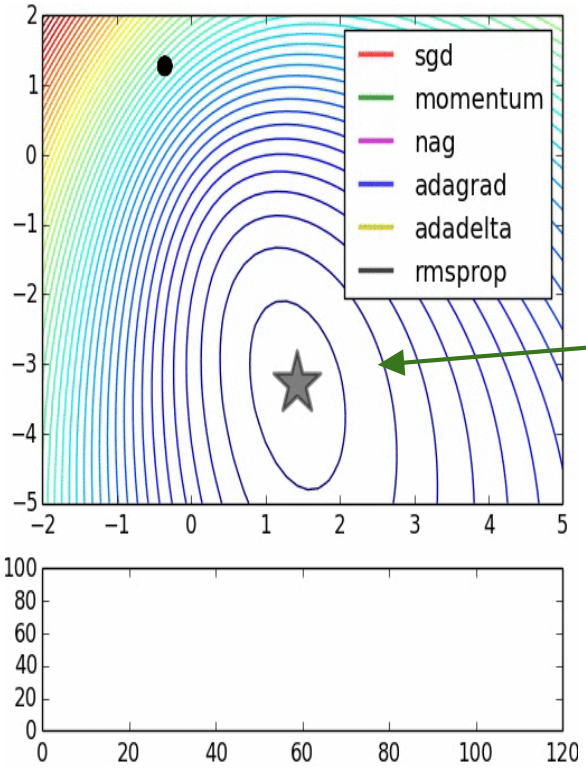
$$W^{(t+1)} = W^{(t)} + p^{(t+1)}$$

Where α is the learning rate.

SGD

vs

Momentum



notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

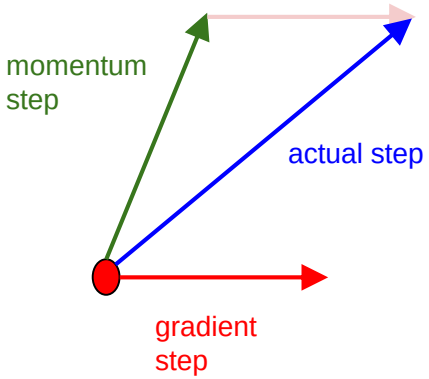
Nesterov Momentum update

Ordinary momentum update:

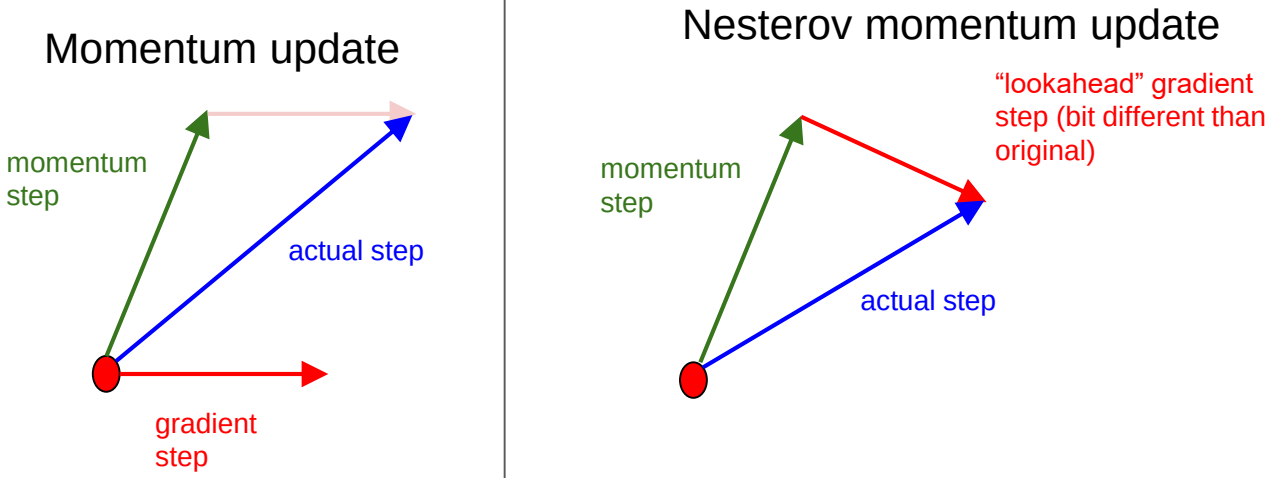
$$p^{(t+1)} = \mu p^{(t)} - \alpha g^{(t)}$$

$$W^{(t+1)} = W^{(t)} + p^{(t+1)}$$

The update as a diagram:

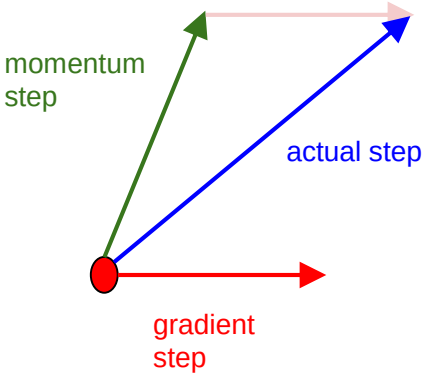


Nesterov Momentum update

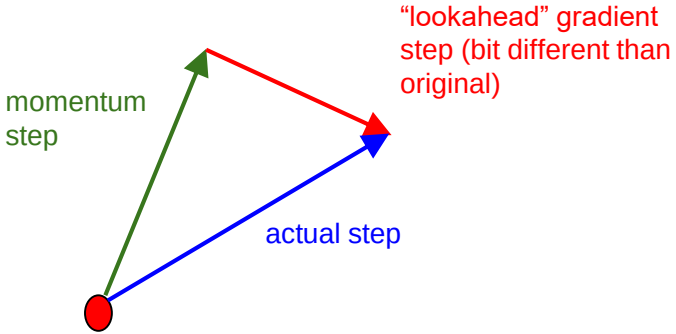


Nesterov Momentum update

Momentum update



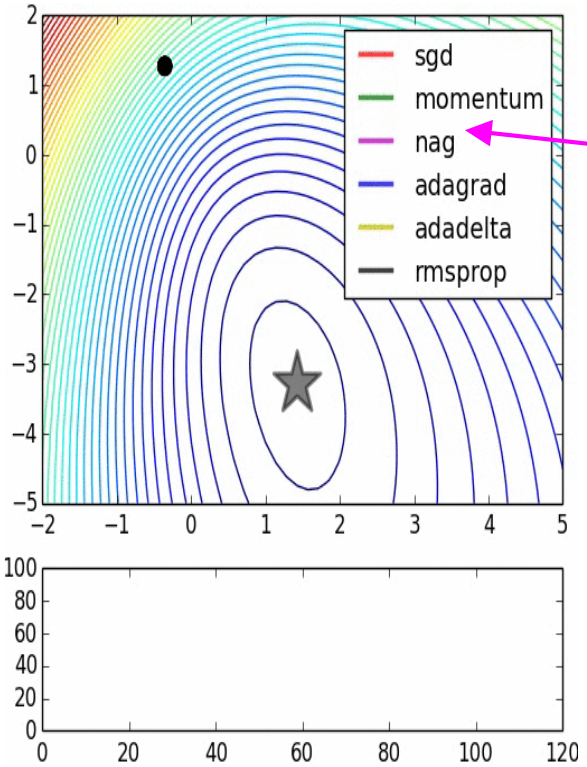
Nesterov momentum update



New term

$$p^{(t+1)} = \mu p^{(t)} - \alpha \nabla_W L(W^{(t)} + \mu p^{(t)})$$

$$W^{(t+1)} = W^{(t)} + p^{(t+1)}$$



nag = Nesterov Accelerated Gradient

RMSprop

[Hinton et al., 2012]

Gradients can vary wildly even though parameters often have the same scale.

RMSprop scales the gradients by the inverse of a moving average, RMS (Root-Mean-Squared) gradient. Define

$$s^{(t)} = \beta s^{(t-1)} + (1-\beta)\left(g^{(t)}\right)^2$$

where $s^{(t)}$ is the (moving average) Mean-Squared Gradient at step t,

$g^{(t)}$ is the normal minibatch gradient at step t,

$\beta \in [0,1]$ is a moving-average decay factor, (close to 1)

$\left(g^{(t)}\right)^2$ is the element-wise square of $g^{(t)}$, so $s^{(t)}$ has same dims as $g^{(t)}$.

RMSprop:

$$W^{(t+1)} = W^{(t)} - \alpha \frac{g^{(t)}}{\sqrt{s^{(t)}}}$$

ADAGRAD

[Duchi et al., 2011]

ADAGRAD is similar to RMSprop, but uses the ***cumulative sum*** of squared gradients.

Define

$$c^{(t)} = \sum_{j=1}^{t} \left(g^{(j)}\right)^2$$

Where $c^{(t)}$ is the ***Cumulative*** Squared Gradient at step t,

ADAGRAD:

$$W^{(t+1)} = W^{(t)} - \alpha \frac{g^{(t)}}{\sqrt{c^{(t)}}}$$

Note: $c^{(t)}$ tends to grow linearly with time t, so ADAGRAD decreases its effective learning rate over time as $1/\sqrt{t}$

RMSprop and ADAGRAD

- Because they normalize gradient magnitudes, both RMSprop and ADAGRAD work very well on datasets with a wide range of gradient magnitudes.

- The most common example is text data. Word frequencies follow a power law: the j^{th} most common word has a relative frequency of $1/j$. So word gradients vary over 4-5 orders of magnitude.

- Using RMSprop/ADAGRAD can accelerate learning simple text models by 2-3 orders of magnitude.

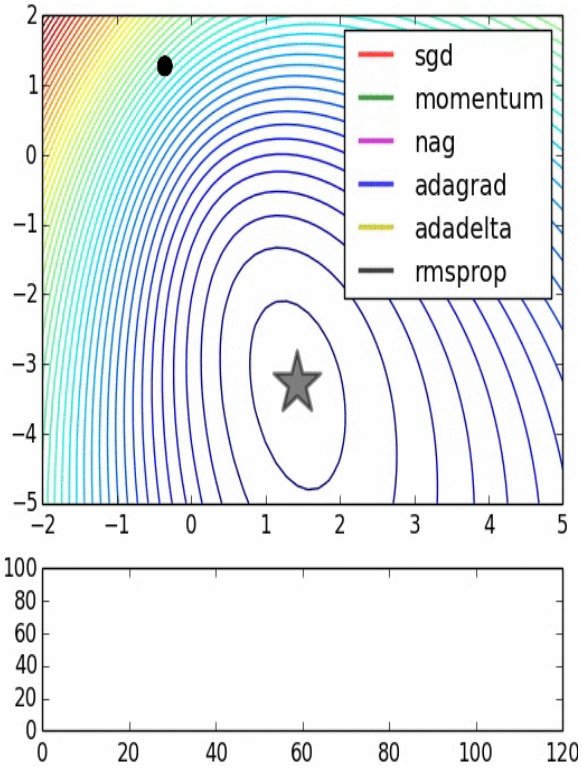
- But less effective with strong feature dependencies.

RMSprop vs. ADAGRAD

- RMSprop is a heuristic method, ADAGRAD has formal bounds on its convergence rate, although only for convex problems.

- The learning rate in RMSprop is fixed across time, and more suitable for long-running training tasks.

- The magnitude of ADAGRAD's sum of squared gradients grows linearly with time, so the learning rate of ADAGRAD decays as $1/\sqrt{T}$ which is quite aggressive. This is very good for short, easy-to-train models but too fast for long-running calculations.



adagrad
rmsprop

Momentum + RMSprop ≈ ADAM

[Kingma and Ba, 2014]

- Compute moving averages of the gradient and squared gradient.
- Treat them as moments and add a small-sample bias correction (next slide):

$$p^{(t)} = \beta_1 p^{(t-1)} + (1-\beta_1) g^{(t)}$$
$$s^{(t)} = \beta_2 s^{(t-1)} + (1-\beta_2)\left(g^{(t)}\right)^2$$

- Then normalize the momentum update (no bias correction):

$$W^{(t+1)} = W^{(t)} - \alpha \frac{p^{(t)}}{\sqrt{s^{(t)}}}$$

- Important practical point: β_1 typically 0.9, β_2 typically much closer to 1, e.g. 0.9999

ADAM Bias Correction

[Kingma and Ba, 2014]

- Moments are initialized to 0 at $t = 0$, so the early moving averages are biased toward zero.

- You can correct this bias (assuming $p^{(t)}$ and $s^{(t)}$ defined as before) like this:

$$p_{corr}^{(t)} = \frac{p^{(t)}}{1 - \beta_1^t}$$

$$s_{corr}^{(t)} = \frac{s^{(t)}}{1 - \beta_2^t}$$

- Then normalize the momentum update:

$$W^{(t+1)} = W^{(t)} - \alpha \frac{p_{corr}^{(t)}}{\sqrt{s_{corr}^{(t)}} + \epsilon}$$

Avoid divide by zero

ADAM Bias Correction

- Caveat: Many users report inconsistent performance with Adam.
- Adam assumes that $s^{(t)}$ and $p^{(t)}$ are *constant* when defining bias.
- Reasonable for $s^{(t)}$ but often false for $p^{(t)}$, since $g^{(t)}$ can oscillate.
- If $g^{(t)}$ oscillates, its true moving average will be smaller (possibly much smaller) than its first value.
- Since β close to 1, e.g. 0.99, the corrections increase $s^{(t)}$ and $p^{(t)}$ by 100x at t=1.

$$s_{corr}^{(t)} = \frac{s^{(t)}}{1-\beta_2^t}$$

Should be OK, since gradient magnitude roughly constant

$$p_{corr}^{(t)} = \frac{p^{(t)}}{1-\beta_1^t}$$

May significantly overestimate the moving average if g not constant

ADAM Bias Correction

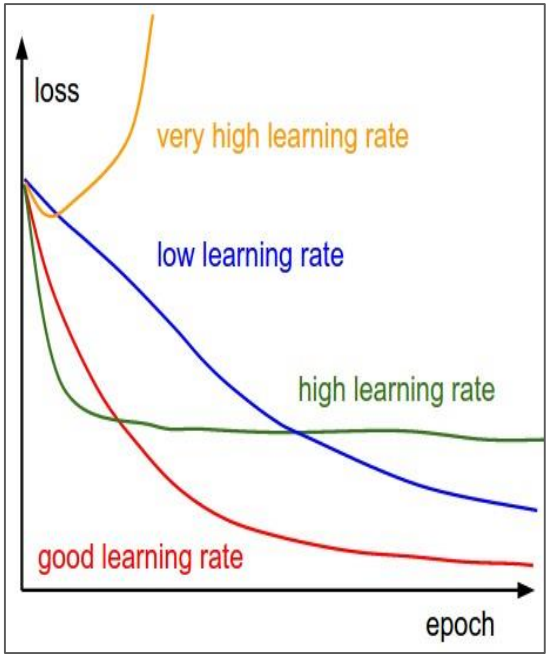
- The ADAM update again:

$$W^{(t+1)} = W^{(t)} - \alpha \frac{p_{corr}^{(t)}}{\sqrt{s_{corr}^{(t)}} + \epsilon}$$

Possibly an over-estimate

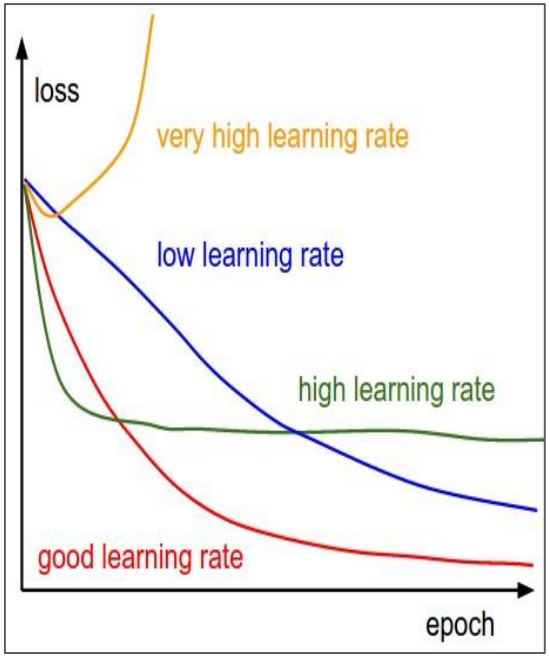
Should be a good estimate

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.



=> Learning rate decay over time!

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

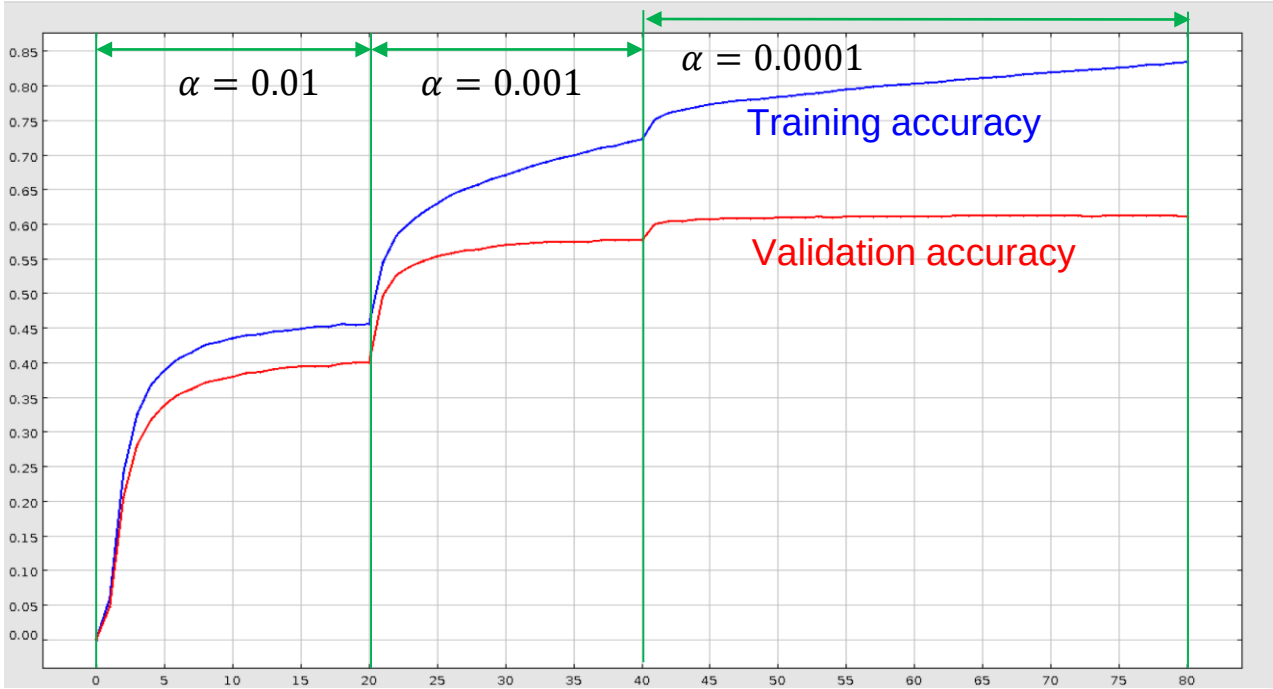
exponential decay: $\alpha = \alpha_0 e^{-kt}$

1/t decay: $\alpha = \alpha_0 / (1 + kt)$

$1/\sqrt{t}$ decay (ADAGRAD): $\alpha = \alpha_0 / \sqrt{1 + kt}$

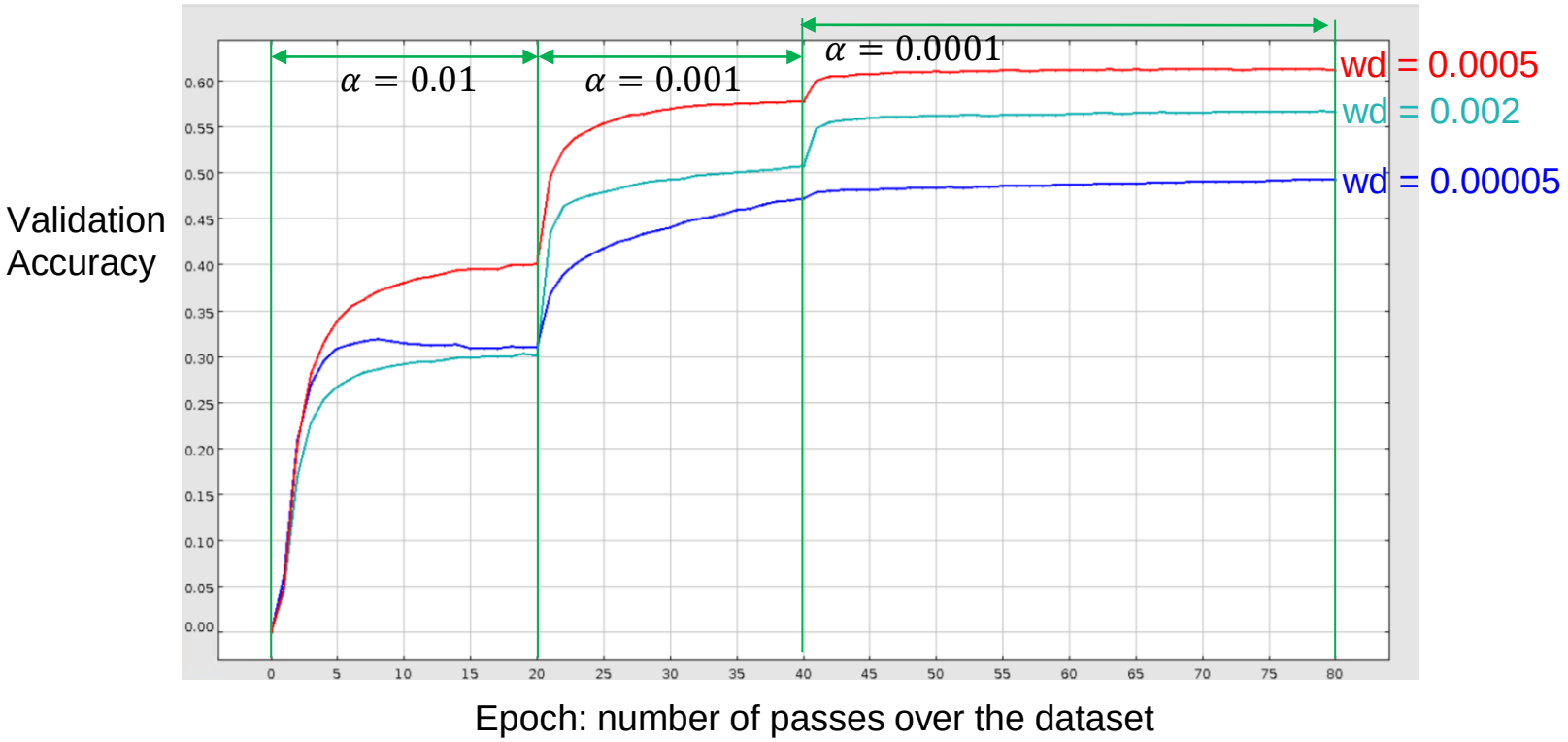
Learning rate schedules

Alexnet trained on ImageNet data. α = learning rate.



Effects of regularization

Alexnet trained on ImageNet data. Validation accuracy shown. wd = weight decay coefficient



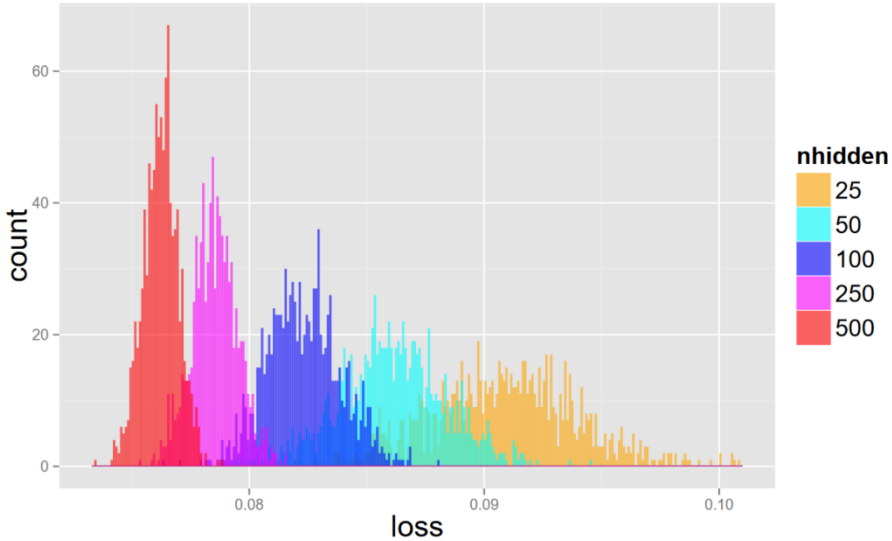
Are Local Minima a Problem?

They happen, but most local minima have similar loss to the global minimum.

e.g. MNIST digit recognition task:
images of 10x10 pixels

Authors built a two-layer network with nhidden units in the middle layer.

1000 networks are trained for each value of nhidden, and their final loss plotted at right.



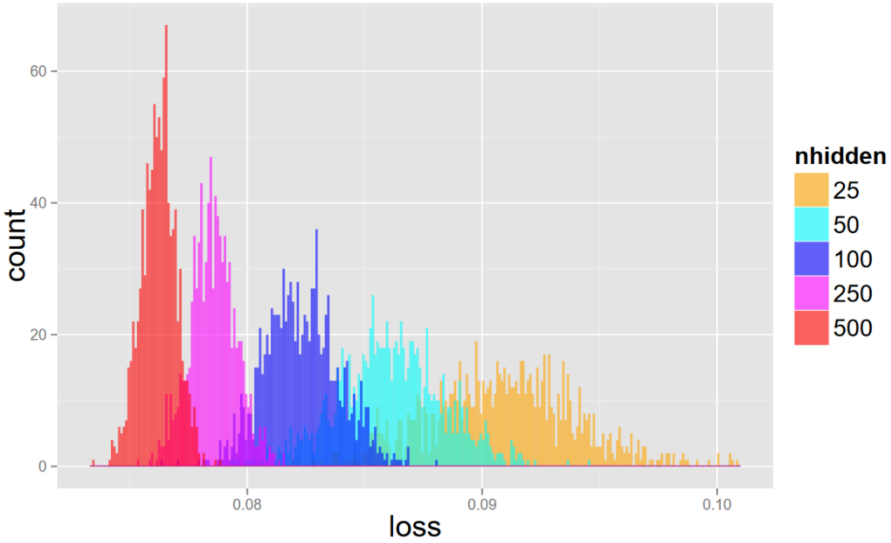
See Choromanska, Henaff, Mathieu, Ben Arous and Lecunn
“The Loss Surface of Multilayer Networks” arXiv 1412.0233, 2014.

Are Local Minima a Problem?

They happen, but most local minima have similar loss to the global minimum.

For a simple image network, as the network gets more complex (more hidden units), there are more local minima clustered even more closely near the global minimum loss.

So its fine, even desirable, to design very complex networks to mitigate the local minima problem.



See Choromanska, Henaff, Mathieu, Ben Arous and Lecunn “The Loss Surface of Multilayer Networks” arXiv 1412.0233, 2014.

Prospectus

- In the early days of deep network optimization, researchers borrowed ideas from convex optimization (e.g. ADAGRAD, Nesterov). These can work very well on certain problems that are,… well,… nearly convex.

- There was concern that existence of multiple local minima might make it hard to find good minima, but Choromanska et al. suggest that it's not a big problem.

- There has also been a lot of concern that saddle points slow down learning. However there are recent positive results:

First-order Methods Almost Always Avoid Saddle Points. Jason D. Lee, Ioannis Panageas, Georgios Piliouras, Max Simchowitz, Michael I. Jordan, and Benjamin Recht.

- As Choromaska et al. suggest "overparametrization" helps the local optima problem.

Gradient Descent Finds Global Minima of Deep Neural Networks Simon S. Du, Jason D. Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai

Optimization Summary

- **Simple Gradient Methods** like SGD can make adequate progress to an optimum when used on minibatches of data.
- **Second-order** methods (quasi-Newton) make much better progress toward a gradient zero, but are more expensive and unstable. They also can't distinguish optima from saddle points, and get trapped in the latter.
- **Momentum:** is another method to produce better effective gradients.
- **ADAGRAD, RMSprop:** diagonally scale the gradient. **ADAM** diagonally scales and applies momentum.
- **Nesterov Momentum:** can improve over vanilla SGD by gradient "look-ahead"
- **Assignment 1** is out, due Feb 19. Please start soon.

Next Topic: Backpropagation

So far we have been using gradient methods to minimize a loss over some parameters.

Our loss is of the form $L(f(x, W), y)$.

where x is an input, y is a target, and W are the parameters.

To compute the gradient of L wrt W, we need the ***chain rule***. If f is single-valued, W a single parameter the chain rule is just:

$$\frac{dL}{dW} = \frac{dL}{df}\frac{df}{dW}$$

If W is a vector of parameters, then we have:

$$\nabla_W L = \frac{dL}{df}\nabla_W f$$

Which is really just the first rule applied to all the partial derivatives wrt elements of W.

The Chain Rule

If f is also vector-valued with k values and W is a vector of m parameters, then we can apply the chain rule parameter-wise and then sum over the contributions:

$$\frac{\partial L}{\partial W_j} = \sum_{i=1}^{k} \frac{\partial L}{\partial f_i} \frac{\partial f_i}{\partial W_j}$$

For $j = 1, \ldots, m$. This can be written as a matrix multiply

$$J_L(W) = J_L(f)\ J_f(W)$$

Where $J_f(W)$ is a Jacobian matrix.

$$J_f(W)_{ij} = \frac{\partial f_i}{\partial W_j}$$

Jacobians

The Jacobian generalizes the gradient of a scalar-valued function f to a k-valued function. Here we think of the function as a neural layer with m inputs and k outputs.

$$J_f(W) = \begin{bmatrix} \frac{\partial f_1}{\partial W_1} & \frac{\partial f_1}{\partial W_2} & \cdots & \frac{\partial f_1}{\partial W_m} \\ \frac{\partial f_2}{\partial W_1} & \frac{\partial f_2}{\partial W_2} & \cdots & \frac{\partial f_2}{\partial W_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial W_1} & \frac{\partial f_k}{\partial W_2} & \cdots & \frac{\partial f_k}{\partial W_m} \end{bmatrix}$$

The Jacobian has dimensions $k \times m$ which is $n_{outputs} \times n_{inputs}$.

N-step Chain Rule

Now suppose we have several vector-valued functions $A(.)$, $B(.)$, $C(.)$, ... composed in a chain (e.g. a deep network):

$$W \to A \to B \to C \to \cdots L$$

Algebraically, that looks like:

$$L(W) = L(\cdots C\left(B\big(A(W)\big)\right)\cdots)$$

Then we just multiply Jacobians (matrix multiply $*$) to get the gradient:

$$J_L(W) = J_L(K) * \cdots J_C(B) * J_B(A) * J_A(W)$$

And $J_L(W) = (\nabla_W L)^T$ which is the gradient we need to minimize loss over W.

Backpropagation

Now all the Jacobians are matrices, and matrix multiply is associative.

$$J_L(W) = J_L(K) * \cdots J_C(B) * J_B(A) * J_A(W)$$

So we could actually evaluate the product of Jacobians in any order, including left-to-right and right-to-left.

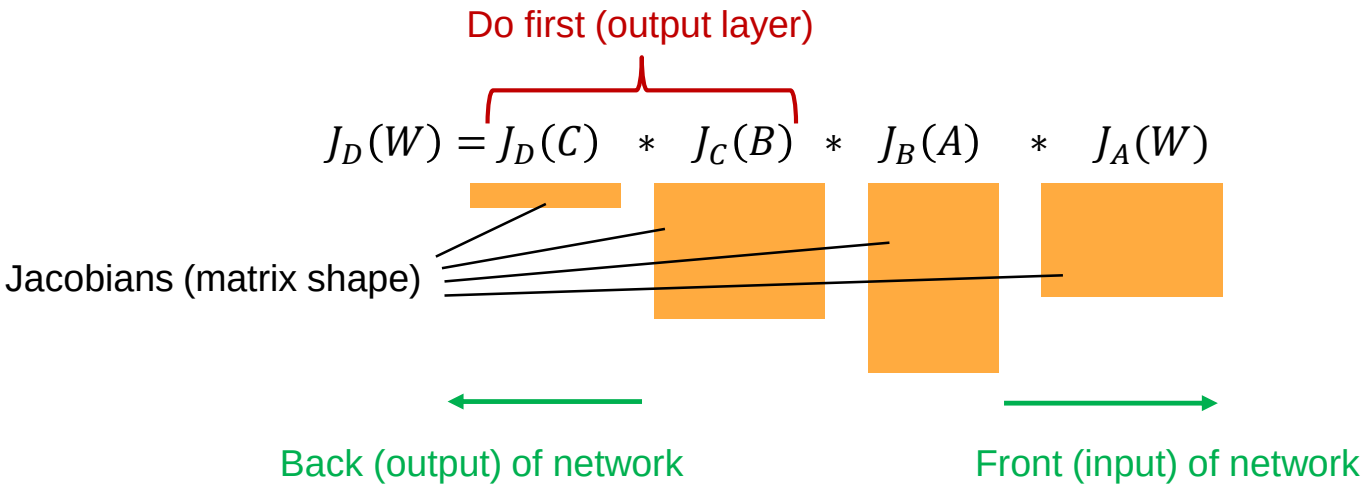
Backpropagation: evaluate the Jacobian product (loss gradient wrt params) left-to-right, i.e. from the output of the neural network toward its input.

Why not some other order?

Backpropagation

Reason 1: Efficiency

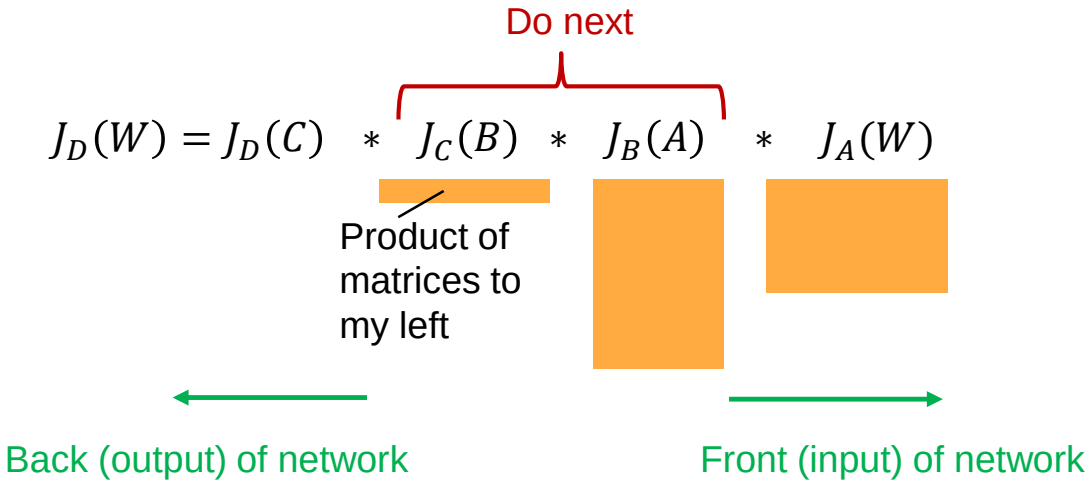
Output Jacobian is always a row vector (because loss is a scalar). Matrix-vector multiply is much less expensive than matrix-matrix multiply.



Backpropagation

Reason 1: Efficiency

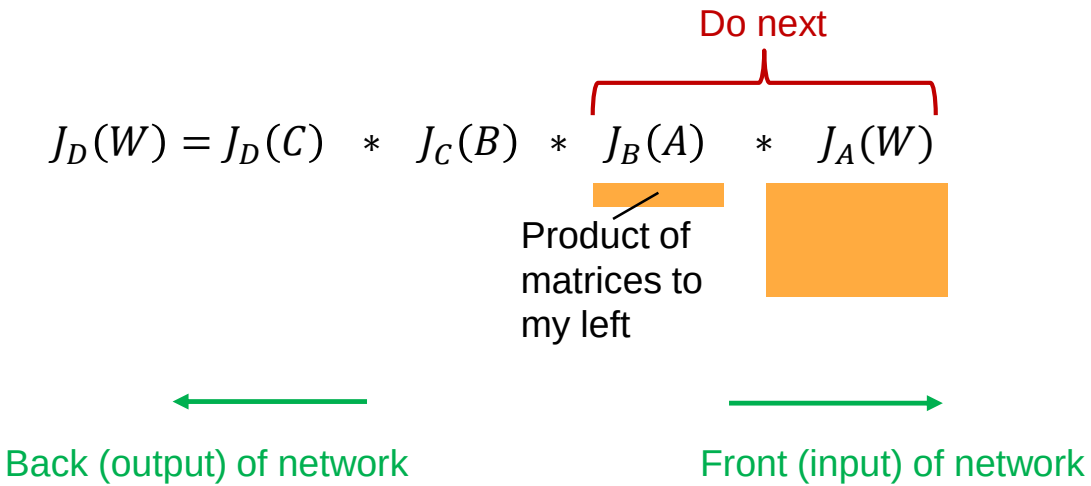
Output Jacobian is always a row vector. Matrix-vector multiply is much less expensive than matrix-matrix multiply.



Backpropagation

Reason 1: Efficiency

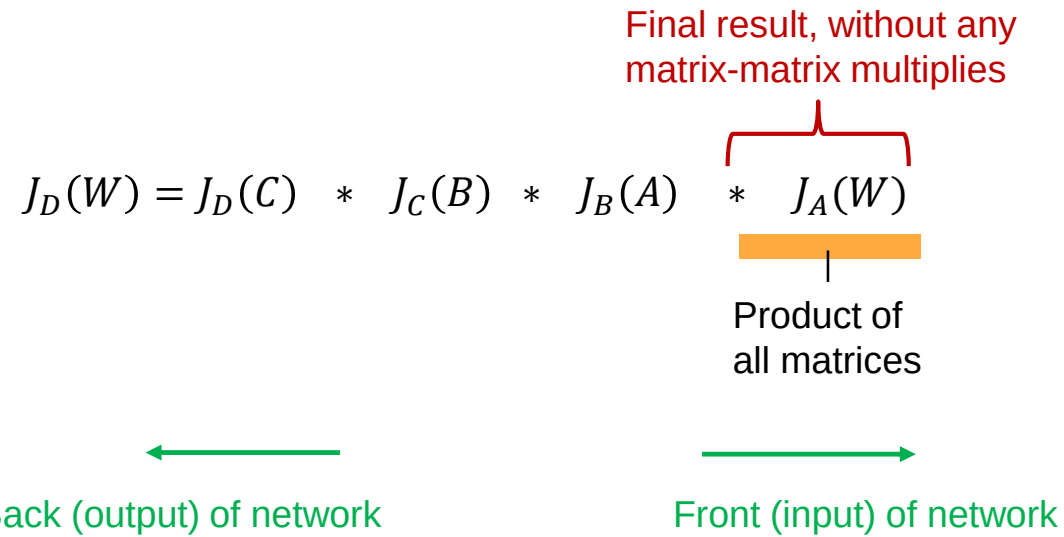
Output Jacobian is always a row vector. Matrix-vector multiply is much less expensive than matrix-matrix multiply.



Backpropagation

Reason 1: Efficiency

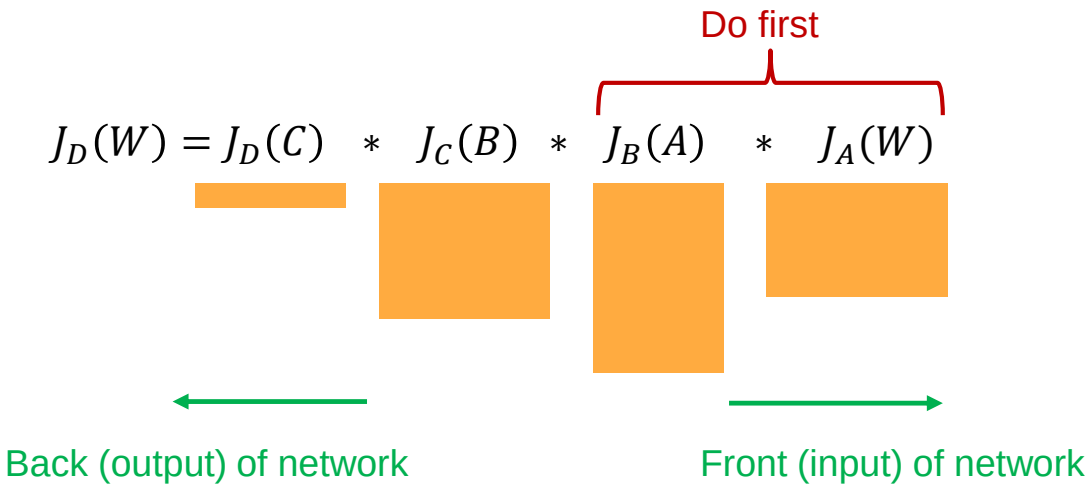
Output Jacobian is always a row vector. Matrix-vector multiply is much less expensive than matrix-matrix multiply.



For-propagation

Reason 1: Efficiency

By comparison, we could invent “for-propagation”:

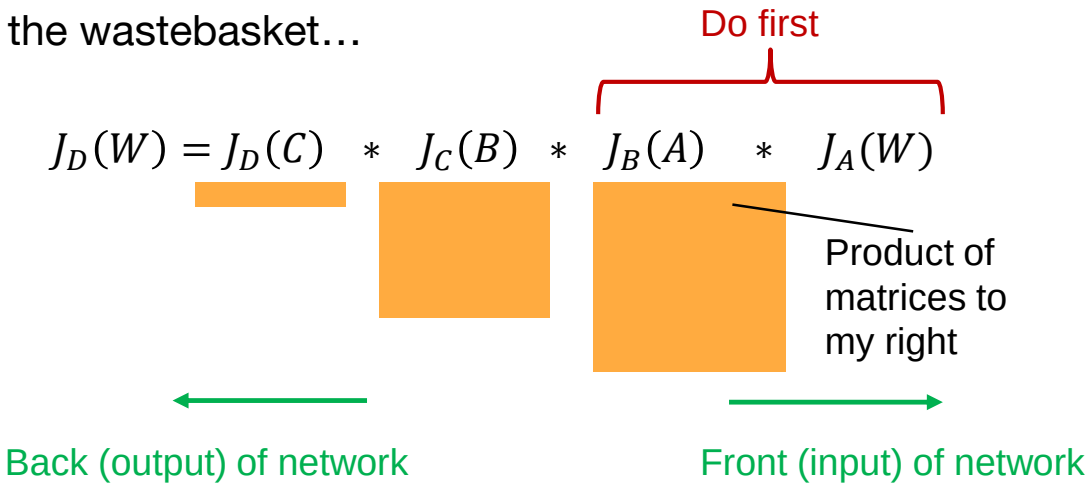


For-propagation

Reason 1: Efficiency

We could invent “for-propagation”: Oops, cost $O(n^3)$ instead of $O(n^2)$ for the first multiply and we still have another matrix-matrix multiply to do.

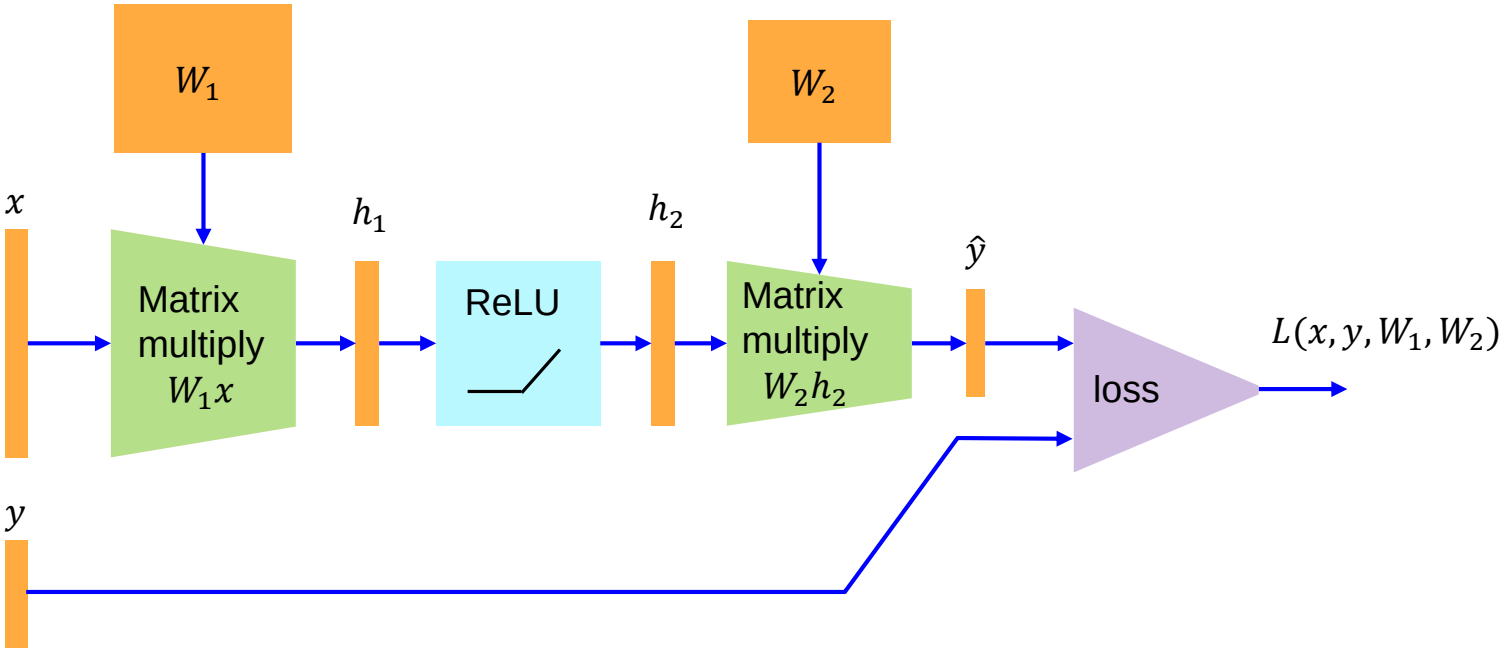
A good invention for the wastebasket…



Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

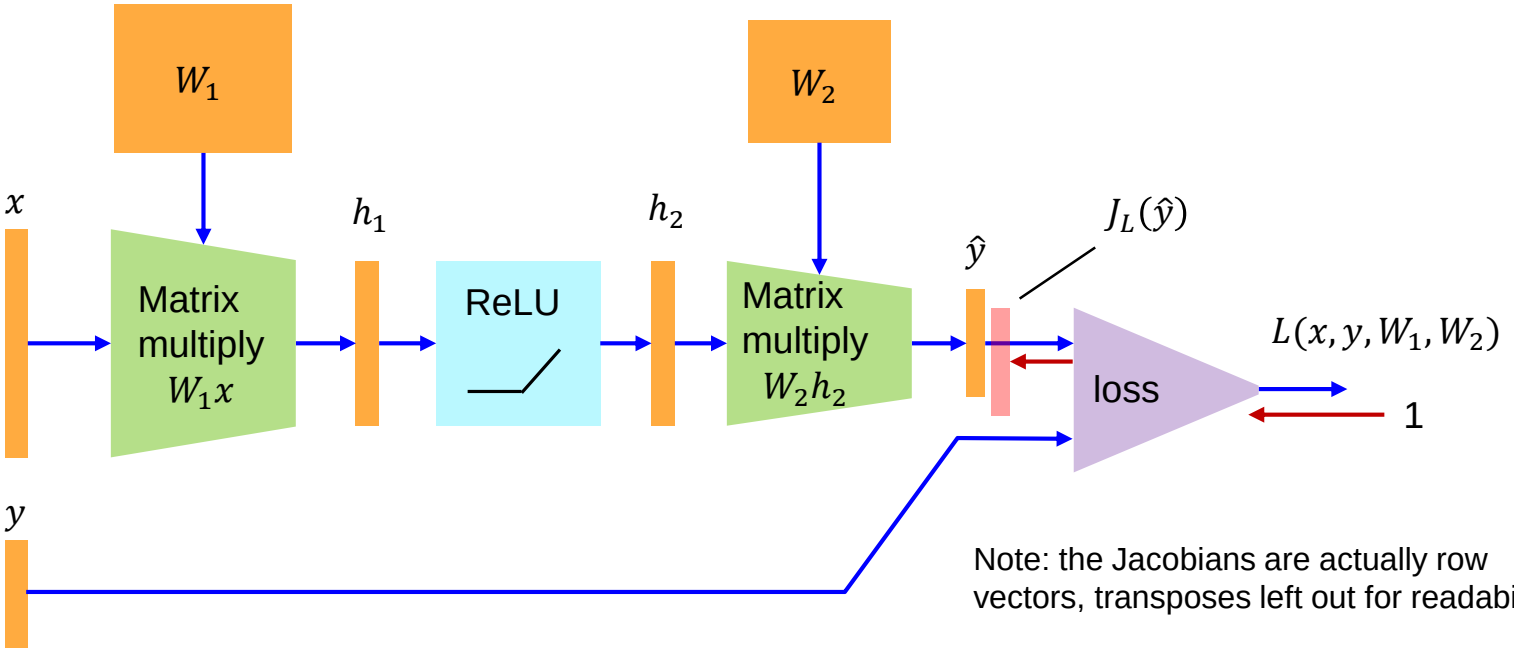
Let's build a real neural network (Tensorflow style):



Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

Let's build a real neural network (Tensorflow style):

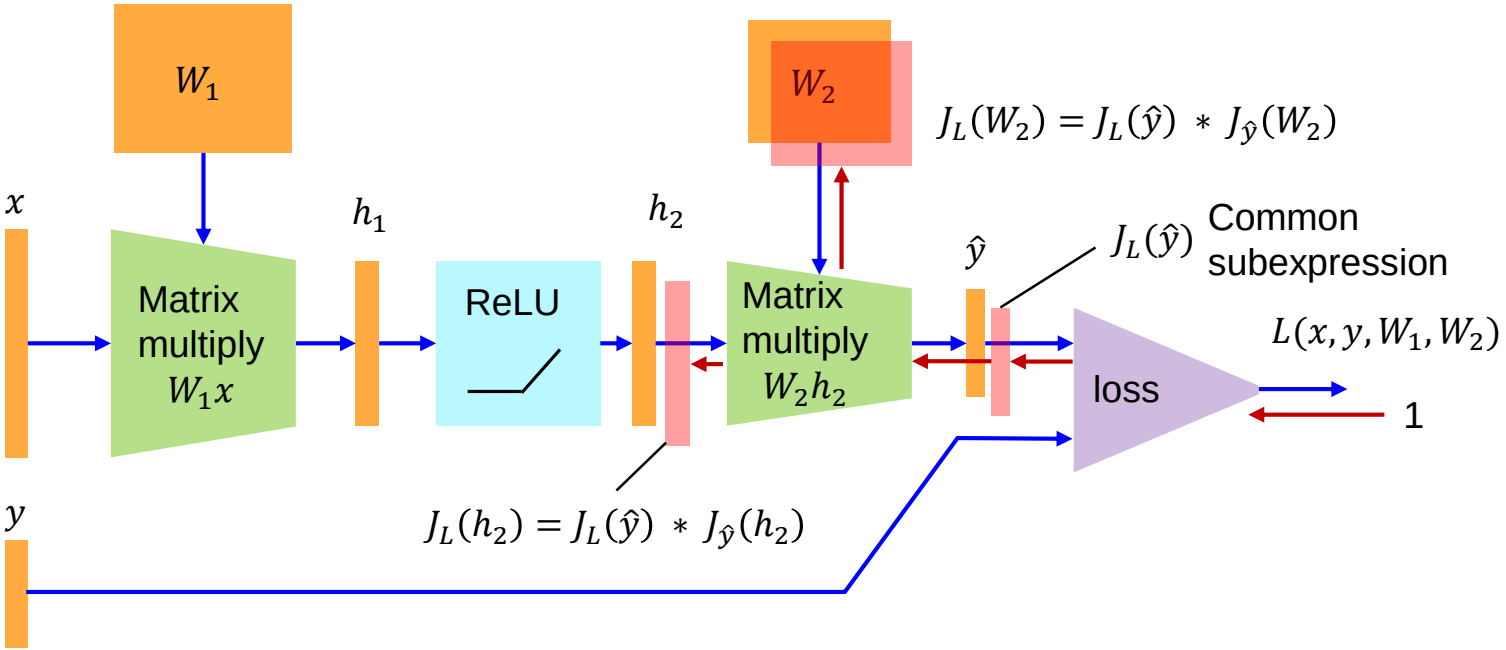


Note: the Jacobians are actually row vectors, transposes left out for readability.

Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

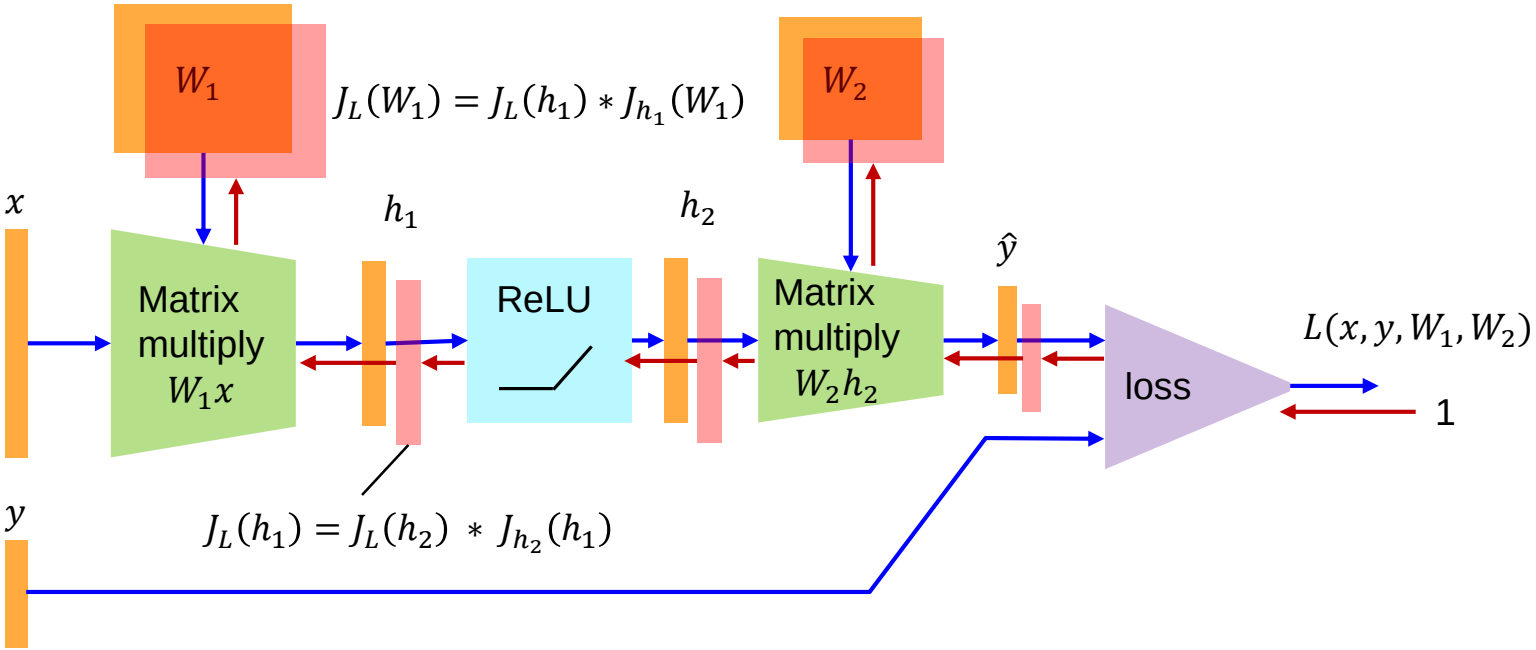
Let's build a real neural network (Tensorflow style):



Backpropagation

Reason 2 to use backpropagation: Common subexpressions.

Let's build a real neural network (Tensorflow style):



Backpropagation

- Compute function values (activations) from the first layer to the last.
- Compute derivatives of the loss wrt other layers from the last layer to the first (backpropagation).
- This only requires matrix-vector multiplies.
- Paths from the loss layer to inner layers are re-used.