Advanced Training of Neural Networks

https://shala2020.github.io/

Learning Objectives

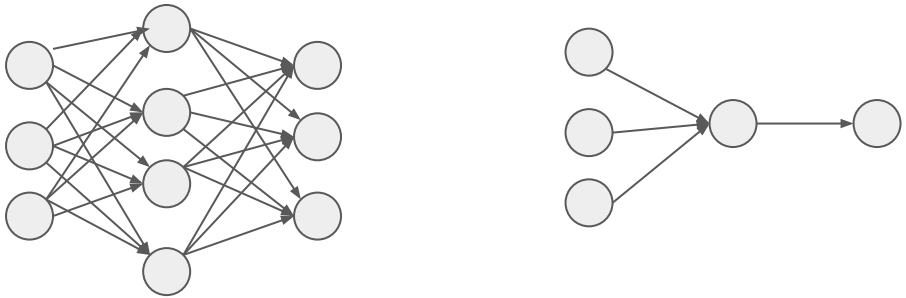
- Write two different ways to initialize weights
- List issues with vanilla gradient descent
- Add momentum to gradient descent
- Use ADAM optimizer for faster training
- Use dropout for better training
- Use batch normalization for better training

Contents

- **Weight initialization**
- Making GD faster
- Helping deep NNs train better

Weight Initialization

- Before we start training the network with gradient descent, we need to initialize the weights
- Initializing all the weights to the same value(eg 0) is not a good choice because all the neurons of a given layer will behave in exactly the same way!



For the forward pass, $X^{(l)} = W \cdot Y^{(l-1)}$
If all the weights in W are same, then all dimensions of $X^{(l)}$ are also same

$$\begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \end{bmatrix} = \begin{bmatrix} a & \cdots & a & \cdots \\ \vdots & & & \\ a & \cdots & a & \cdots \\ \vdots & & & \vdots \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_j \\ \vdots \end{bmatrix}$$

For the backward pass, we had derived

$$dW^{(l)} = \nabla_{X^{(l)}} L^T \cdot {Y^{(l-1)}}^T$$

Since all dimensions of $X^{(l)}$ are same, the gradient $\nabla_{X^{(l)}} L^T$ will also have the same values and all rows of $dW^{(l)}$ will be same! So each neuron will have the same update rule for gradient descent and all neurons will continue to behave in the same way

$$dW = \begin{bmatrix} b \\ \vdots \\ b \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} y_1 & \cdots & y_j & \cdots \end{bmatrix} = \begin{bmatrix} by_1 & \cdots & by_j & \cdots \\ \vdots & & & \\ by_1 & \cdots & by_j & \cdots \\ \vdots & & & \vdots \end{bmatrix}$$

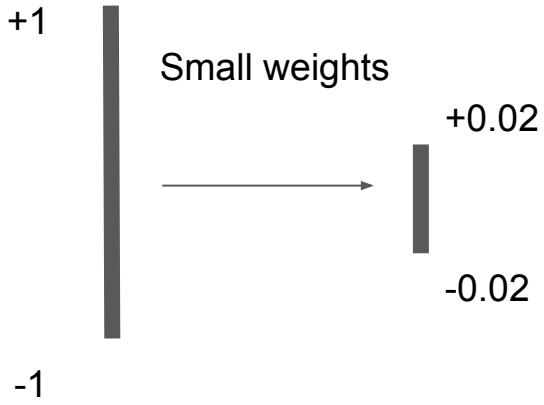
$$W - dW = \begin{bmatrix} a - by_1 & \cdots & a - by_j & \cdots \\ \vdots & & & \\ a - by_1 & \cdots & a - by_j & \cdots \\ \vdots & & & \vdots \end{bmatrix}$$

Random Initialization

- To prevent neurons of a layer from behaving identically, we can ‘break the symmetry’ by randomly initializing weights (eg sampling from a Gaussian distribution)
- This method works for small networks, but we encounter problems with deeper networks.

Random Initialization

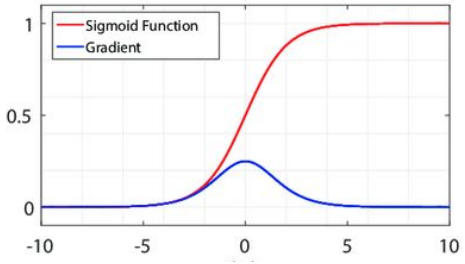
- Let’s say the weights initialized are too small. The output of sigmoid or tanh is bounded between -1 and +1, so multiplying these by small weights will push the inputs to the next layer close to 0 ; and in deeper networks activations of neurons in deeper layers will become very close to 0 (with small variance)

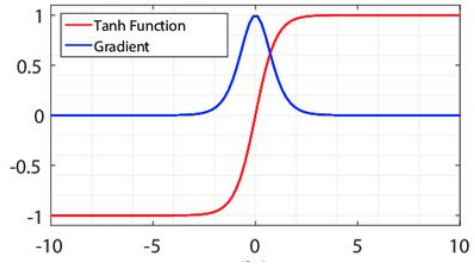


Eventually the activations become close to 0. For the backward pass we calculate dW by multiplying dX with Y ; and dY by multiplying dX with W. Since both W and Y are small, both gradients are also small and the weights won’t get updated properly in the descent step

Random Initialization

- If the weights are too large, then the variance of neurons increases rapidly as we move deeper and we get stuck in the saturation regions of the activation functions where the gradient is very small

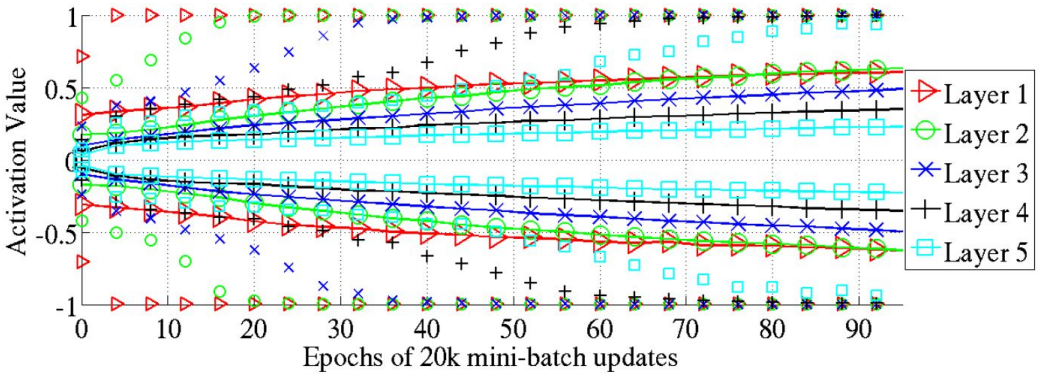




Due to very small gradients, we again face problems in training the neural network

Problem: Saturation of activations

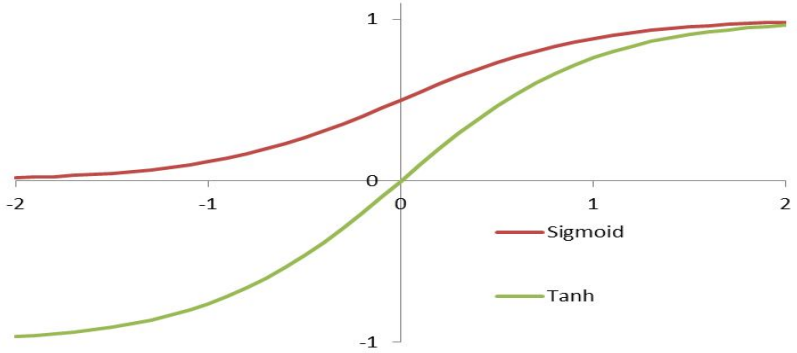
- Activations saturate for the lower layers first, then upper layers

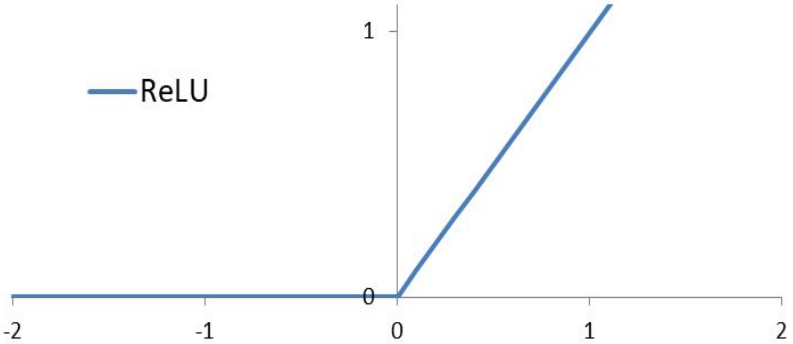


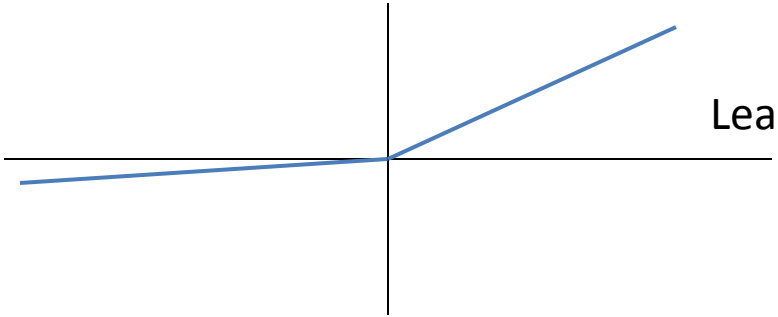
- Markers are 98th percentile, lines are std. dev.
- Activation is *tanh*

Source: "Understanding the difficulty of training deep feedforward neural networks," Xavier Glorot and Yoshua Bengio, AISTATS'10

Activation function solve some of the vanishing gradient problem







Leaky ReLU

Solution: Better initialization

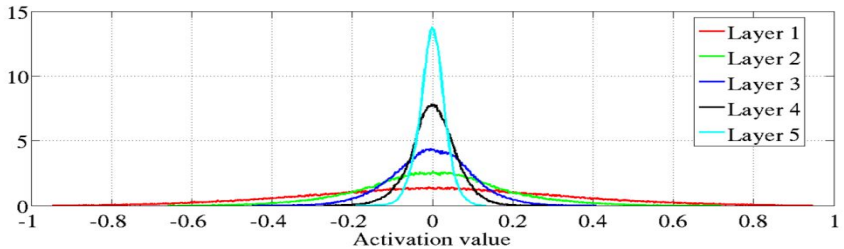


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

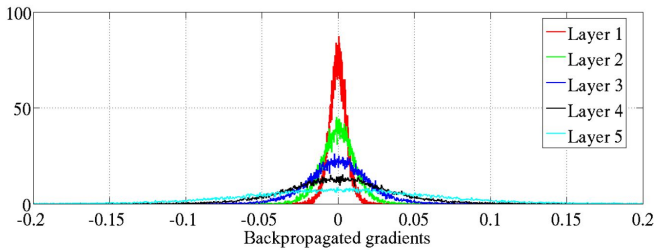


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Source: "Understanding the difficulty of training deep feedforward neural networks," Xavier Glorot and Yoshua Bengio, AISTATS'10

Conditions for keeping it together

- If
  - $z_{i+1} = f(a_i)$ is the layer output
  - $a_i = z_i W_i + b_i$ is the input to the nonlinear function
- For all pairs of layers i and j
  - For forward prop: $Var[z_i] = Var[z_j]$
  - For backprop: $Var[\partial L/\partial a_i] = Var[\partial L/\partial a_j]$
- That is, for all i, assuming linear activation
  - $n_i \, Var[W_i] = 1$
  - $n_{i+1} \, Var[W_i] = 1$
- For tanh activation, its derivate also plays a role
  - Therefore,

$$W \sim U\Big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\Big]$$

Source: "Understanding the difficulty of training deep feedforward neural networks," Xavier Glorot and Yoshua Bengio, AISTATS'10

Xavier initialization

- We know that diminishing and exploding variance with each passing layer is linked with very small gradients. Xavier initialization is a technique which aims at keeping the variance of the outputs of hidden layers equal to that of its input and hence constant throughout.
- The weights for each layer are initialized by sampling from a Gaussian distribution with mean 0 and variance equal to 1/(n) where n is the number of inputs to the layer
- Xavier initialization works well with the assumption that we are in the approximately linear region of the activation function

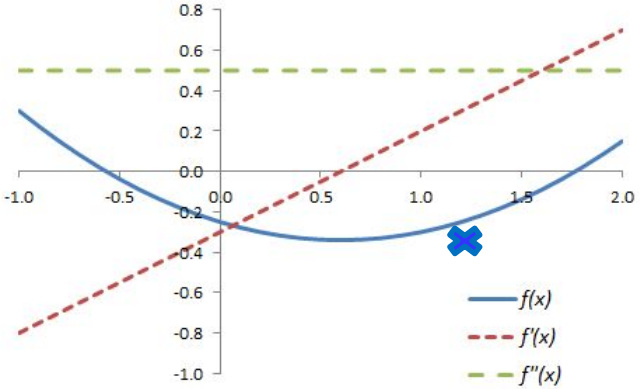
He Initialization

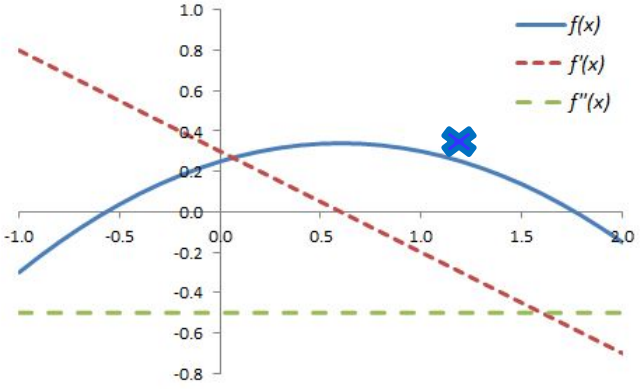
- Xavier initialization does not work well with the ReLU activation function. The intuition is that when the weights are sampled from the Gaussian distribution, approximately half of the outputs will be positive and half negative. Since ReLU kills off the negative part, the next layer effectively sees only half of the previous layers outputs and to keep the variances same, we should take the variance = 2/n where n is the number of outputs of the previous layer.

Contents

- Weight initialization
- **Making GD faster**
- Helping deep NNs train better

Derivative of a function of a scalar

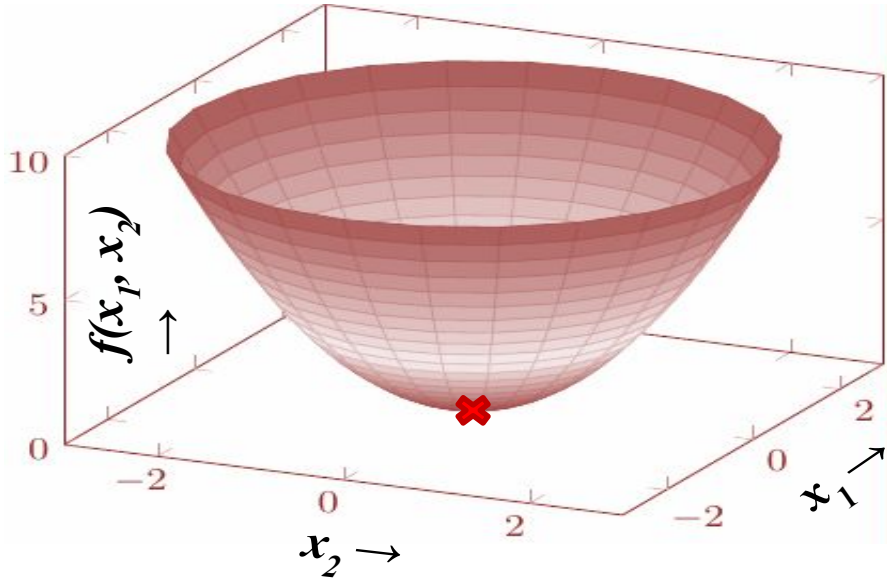




E.g. $f(x) = ax^2 + bx + c, \quad f'(x) = 2ax + b, \quad f''(x) = 2a$

- Derivative $f'(x) = \frac{d\,f(x)}{d\,x}$ is the rate of change of $f(x)$ with x
- It is zero when then function is flat (horizontal), such as at the minimum or maximum of $f(x)$
- It is positive when $f(x)$ is sloping up, and negative when $f(x)$ is sloping down
- To move towards the maxima, taking a small step in a direction of the derivative

Gradient of a function of a vector

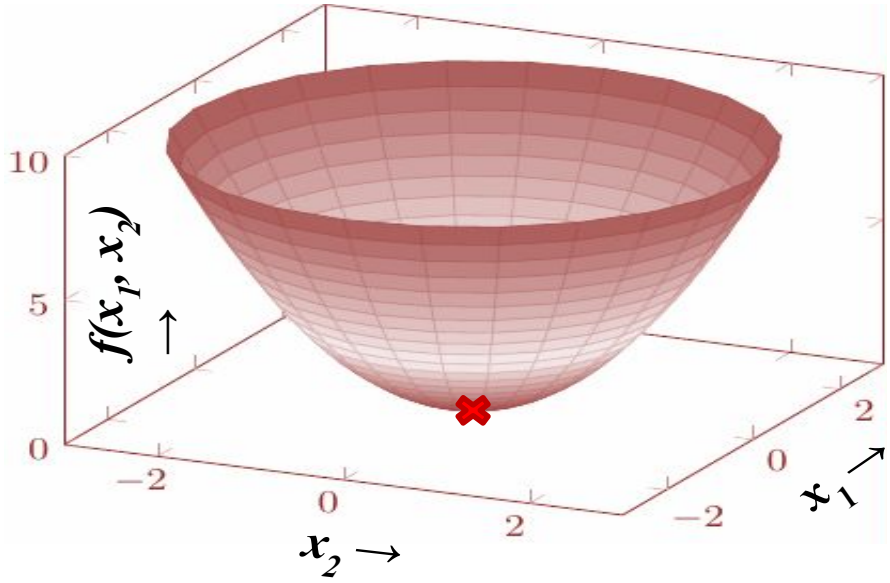


- Derivative with respect to each dimension, holding other dimensions constant
- $\nabla f(\boldsymbol{x}) = \nabla f(x_1, x_2) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$
- At a minima or a maxima the gradient is a zero vector

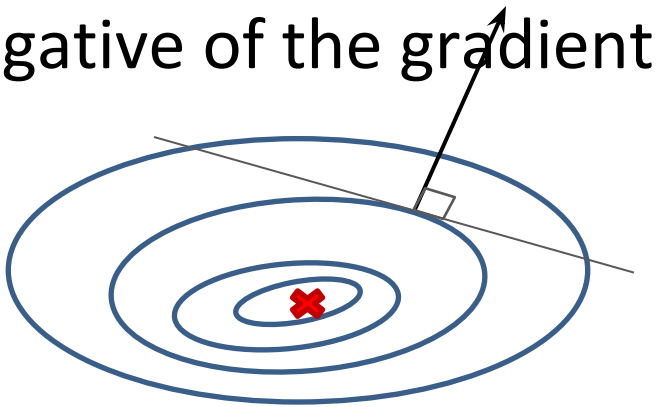
  The function is flat in every direction
- At a minima or a maxima the gradient is a zero vector

Original image source unknown

Gradient of a function of a vector



- Gradient gives a direction for moving towards the minima
- Take a small step towards negative of the gradient

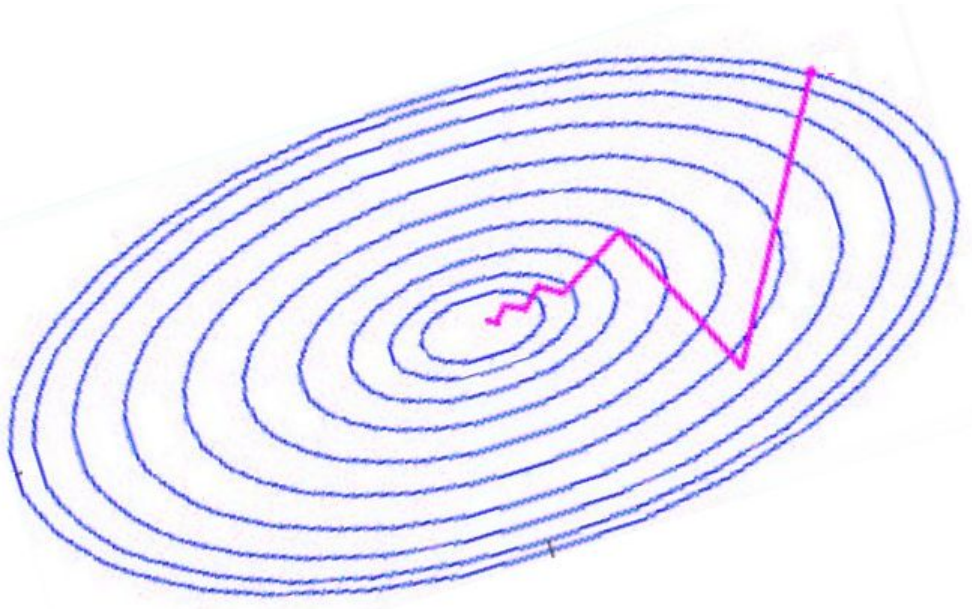


Original image source unknown

Example of gradient

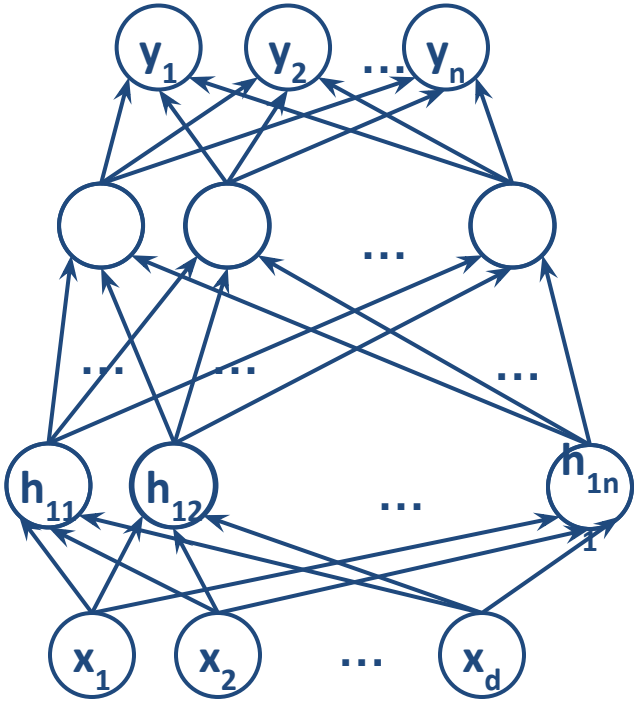
- Let $f(\boldsymbol{x}) = f(x_1, x_2) = 5{x_1}^2 + 3{x_2}^2$
- Then $\nabla f(\boldsymbol{x}) = \nabla f(x_1, x_2) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 10x_1 \\ 6x_2 \end{bmatrix}$
- At a location (2,1) a step in $\begin{bmatrix} 20 \\ 6 \end{bmatrix}$ or $\begin{bmatrix} 0.958 \\ 0.287 \end{bmatrix}$ direction will lead to maximal increase in the function

This story is unfolding in multiple dimensions



Original image source unknown

Backpropagation



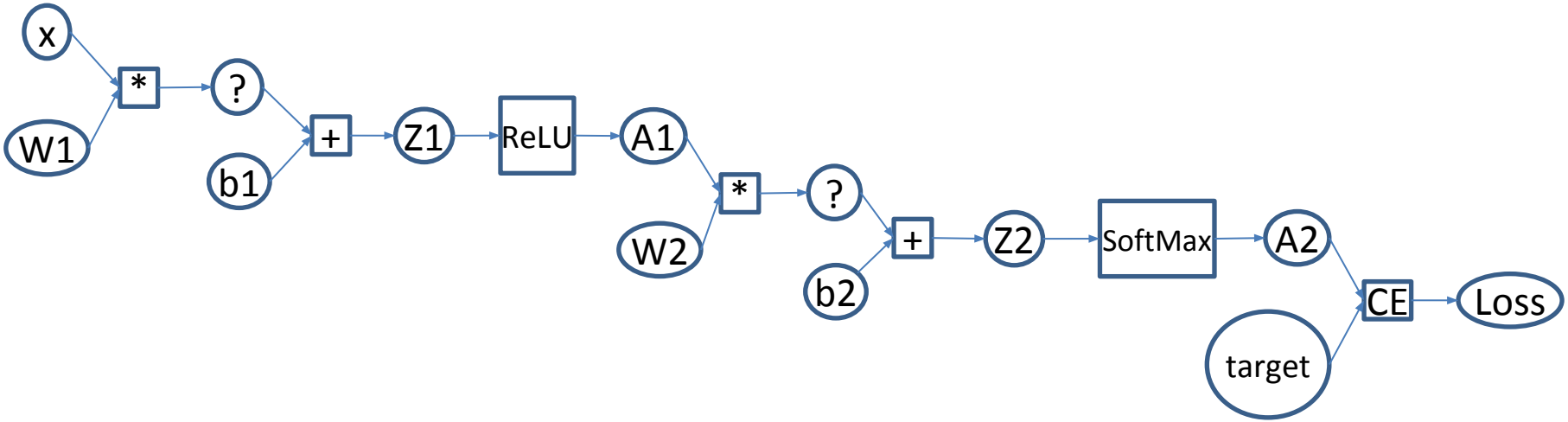
- Backpropagation is an efficient method to do gradient descent
- It saves the gradient w.r.t. the upper layer output to compute the gradient w.r.t. the weights immediately below
- It is linked to the chain rule of derivatives
- All intermediary functions must be differentiable, including the activation functions

Chain rule of differentiation

- Very handy for complicated functions
  - Especially functions of functions
  - E.g. NN outputs are functions of previous layers
- For example: Let $f(x) = g\big(h(x)\big)$
  - Let $y = h(x), z = g(y) = g\big(h(x)\big)$
- Then $f'(x) = \frac{d\,z}{d\,x} = \frac{d\,z}{d\,y}\frac{d\,y}{d\,x} = g'(y)h'(x)$
- For example: $\frac{d\,\sin(x^2)}{d\,x} = 2x\cos(x^2)$

Backpropagation makes use of chain rule of derivatives

- Chain rule: $\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$



Vector valued functions and Jacobians

- We often deal with functions that give multiple outputs
- Let $\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, x_3) \\ f_2(x_1, x_2, x_3) \end{bmatrix}$
- Thinking in terms of vector of functions can make the representation less cumbersome and computations more efficient
- Then the Jacobian is
- $\boldsymbol{J}(\boldsymbol{f}) = \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial x_1} & \frac{\partial \boldsymbol{f}}{\partial x_2} & \frac{\partial \boldsymbol{f}}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix}$

Jacobian of each layer

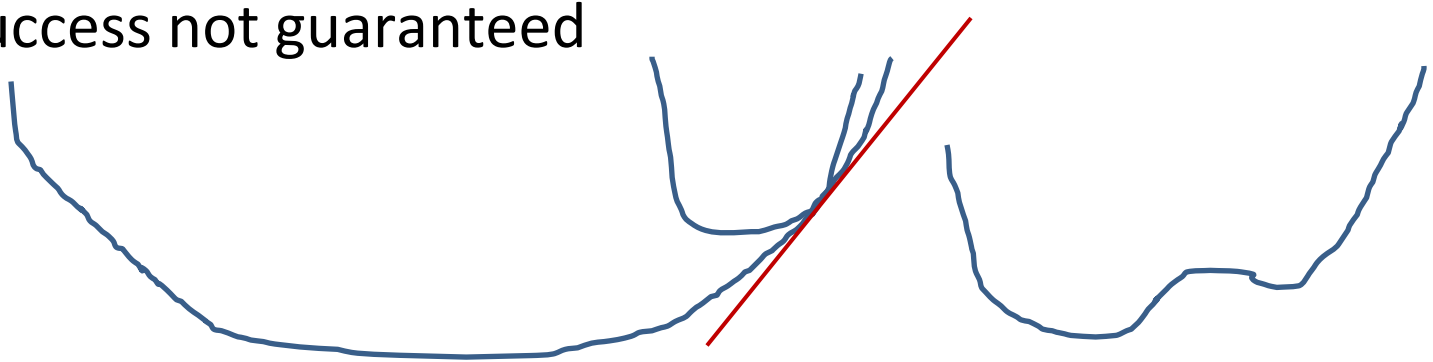
- Compute the derivatives of a higher layer's output with respect to those of the lower layer

- What if we scale all the weights by a factor *R?*

- What happens a few layers down?

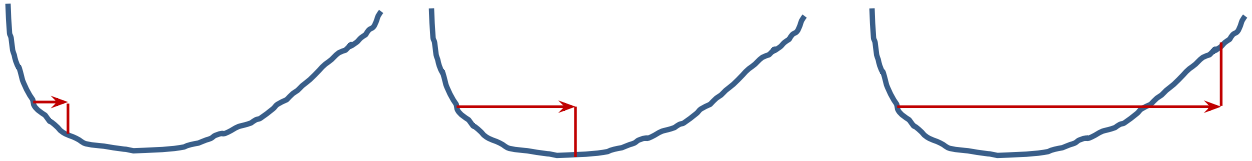
Role of step size and learning rate

- Tale of two loss functions
  - Same value, and
  - Same gradient (first derivative), but
  - Different Hessian (second derivative)
  - Different step sizes needed
- Success not guaranteed



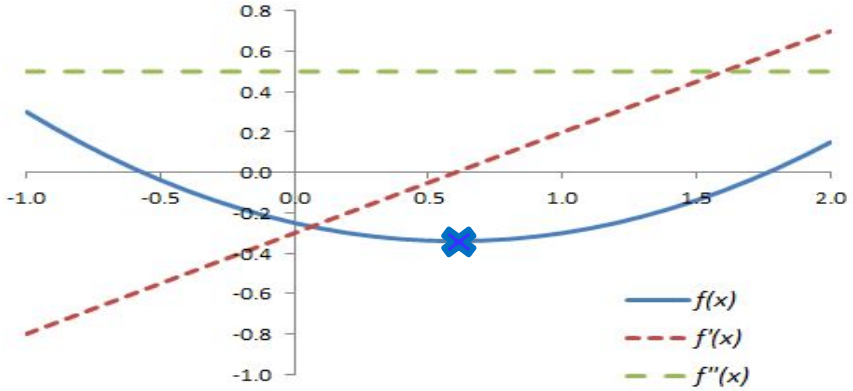
The perfect step size is impossible to guess

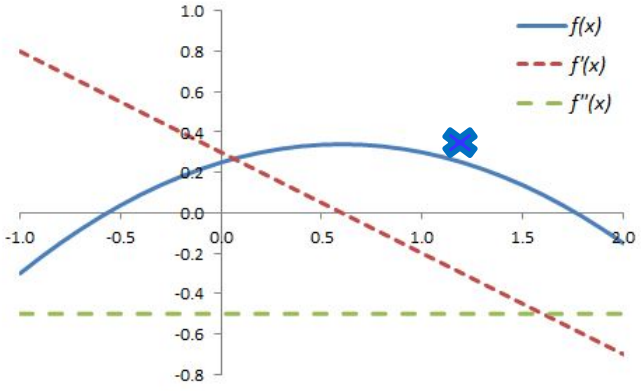
- Goldilocks finds the perfect balance only in a fairy tale



- The step size is decided by learning rate η and the gradient

Double derivative

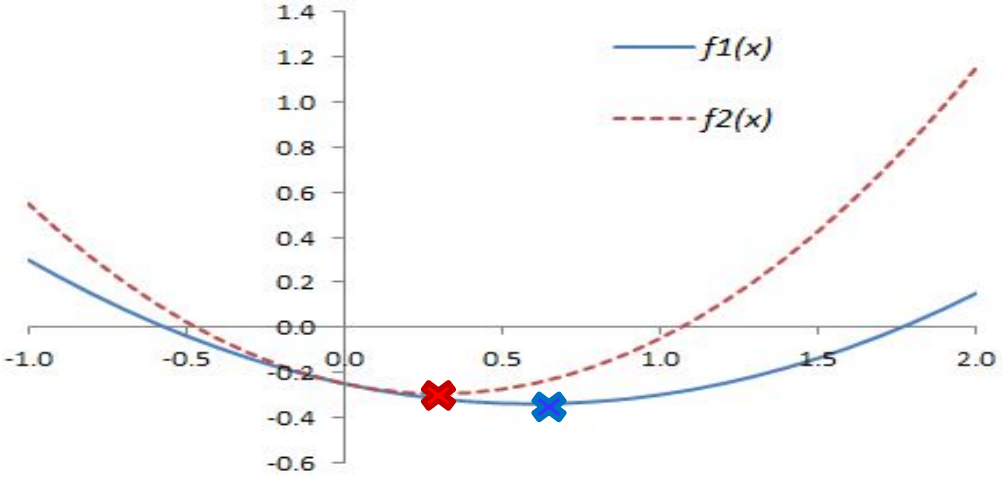




E.g. $f(x) = ax^2 + bx + c, \quad f'(x) = 2ax + b, \quad f''(x) = 2a$

- Double derivative $f''(x) = \frac{d^2 f(x)}{d x^2}$ is the derivative of derivative of $f(x)$
- Double derivative is positive for convex functions (have a single minima), and negative for concave functions (have a single maxima)

Double derivative



$$f(x) = ax^2 + bx + c,$$
$$f'(x) = 2ax + b,$$
$$f''(x) = 2a$$

- Double derivative tells how far the minima might be from a given point.
- From $x = 0$ the minima is closer for the red dashed curve than for the blue solid curve, because the former has a larger second derivative (its slope reverses faster)

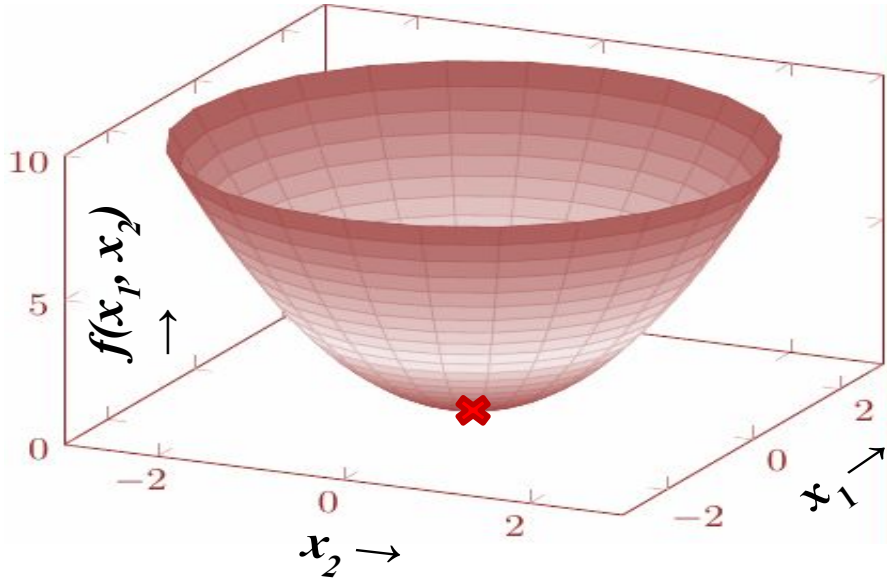
Perfect step size for a paraboloid

- Let $f(x) = ax^2 + bx + c$
- Assuming $a < 0$
- Minima is at: $x^* = -\frac{b}{2a}$
- For any x the perfect step would be:

$$-\frac{b}{2a} - x = -\frac{2ax+b}{2a} = -\frac{f'(x)}{f''(x)}$$

- So, the perfect learning rate is: $\eta^* = \frac{1}{f''(x)}$
- In multiple dimensions, $\boldsymbol{x} \leftarrow \boldsymbol{x} - H\big(f(\boldsymbol{x})\big)^{-1}\nabla(f(\boldsymbol{x}))$
- Practically, we do not want to compute the inverse of a Hessian matrix, so we approximate Hessian inverse

Hessian of a function of a vector



- Double derivative with respect to a pair of dimensions forms the Hessian matrix:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \, \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \, \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \, \partial x_1} & \frac{\partial^2 f}{\partial x_n \, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- If all eigenvalues of a Hessian matrix are positive, then the function is convex

Original image source unknown

Example of Hessian

- Let $f(\boldsymbol{x}) = f(x_1, x_2) = 5{x_1}^2 + 3{x_2}^2 + 4x_1x_2$
- Then

$$\nabla f(\boldsymbol{x}) = \nabla f(x_1, x_2) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 10x_1 + 4x_2 \\ 6x_2 + 4x_1 \end{bmatrix}$$

- And, $H(f(\boldsymbol{x})) = \begin{bmatrix} \frac{\partial^2 f}{\partial {x_1}^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial {x_2}^2} \end{bmatrix} = \begin{bmatrix} 10 & 4 \\ 4 & 6 \end{bmatrix}$

Saddle points, Hessian and long local furrows

- Some variables may have reached a local minima while others have not
- Some weights may have almost zero gradient
- At least some eigenvalues may not be negative

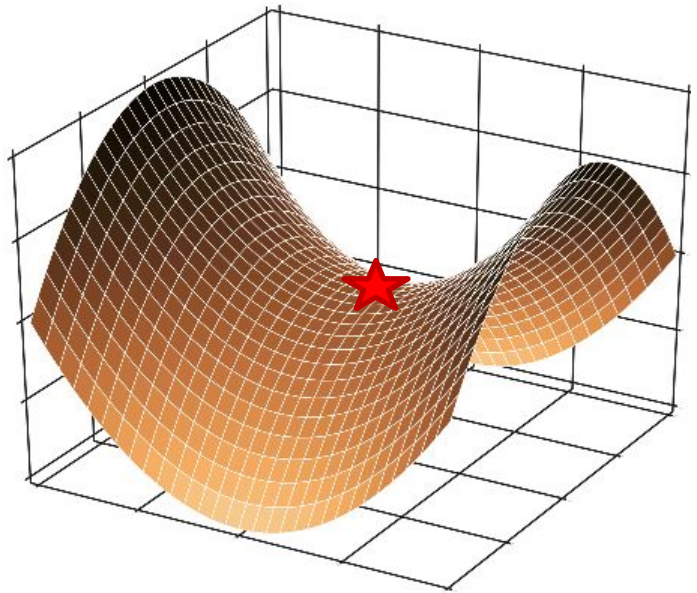
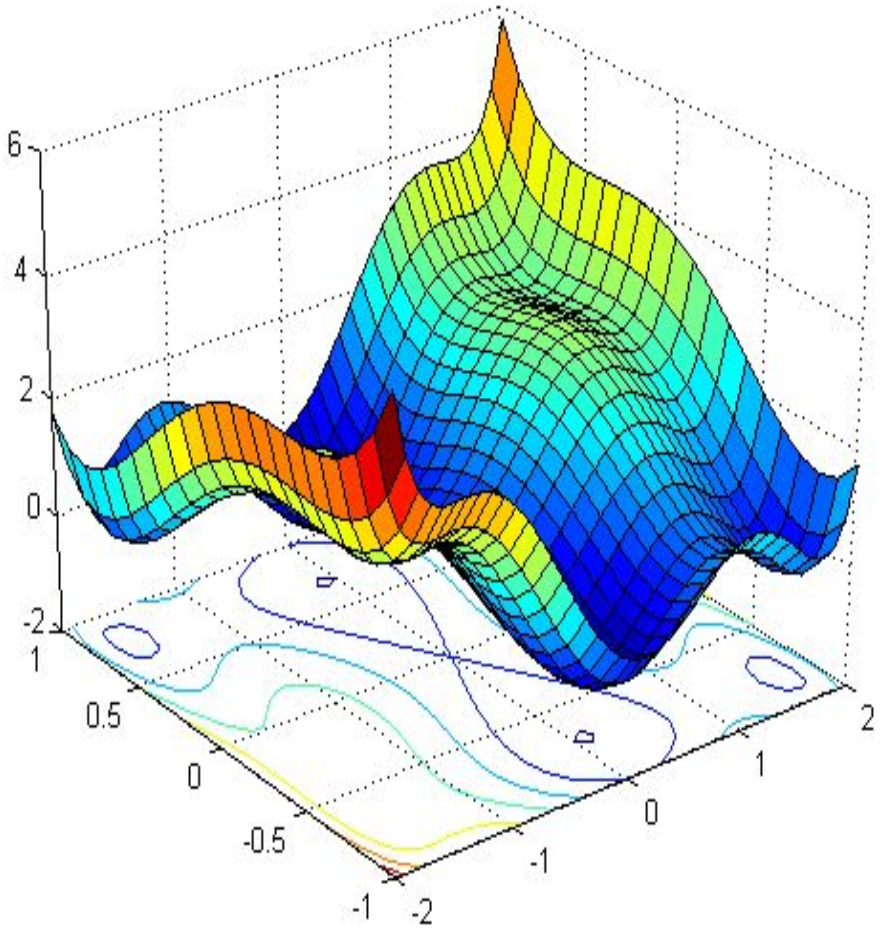


Image source: Wikipedia

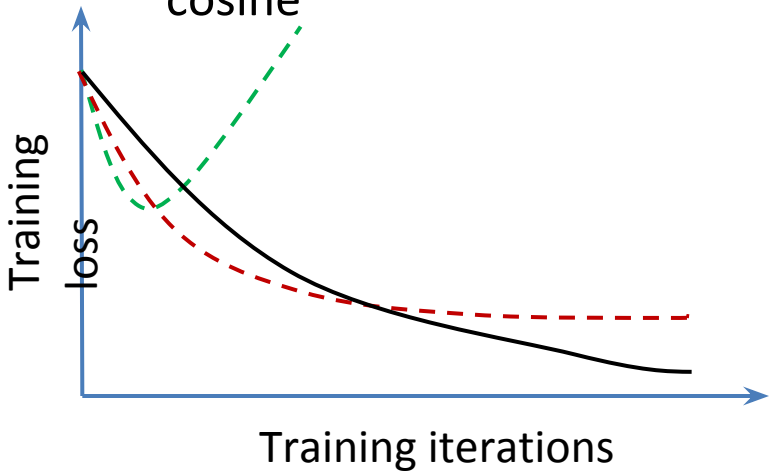
Complicated loss functions

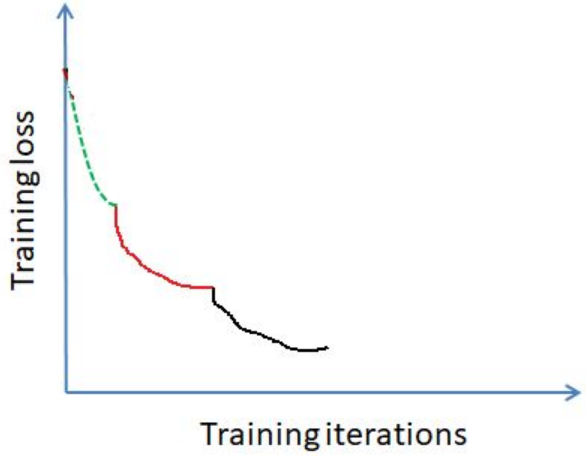


Original image source unknown

Learning rate scheduling

- High learning rates explore faster earlier
  - But, they can lead to divergence or high final loss
- Low learning rates fine-tune better later
  - But, they can be very slow to converge
- LR scheduling combines advantages of both
  - Lots of schedules possible: linear, exponential, square-root, step-wise, cosine

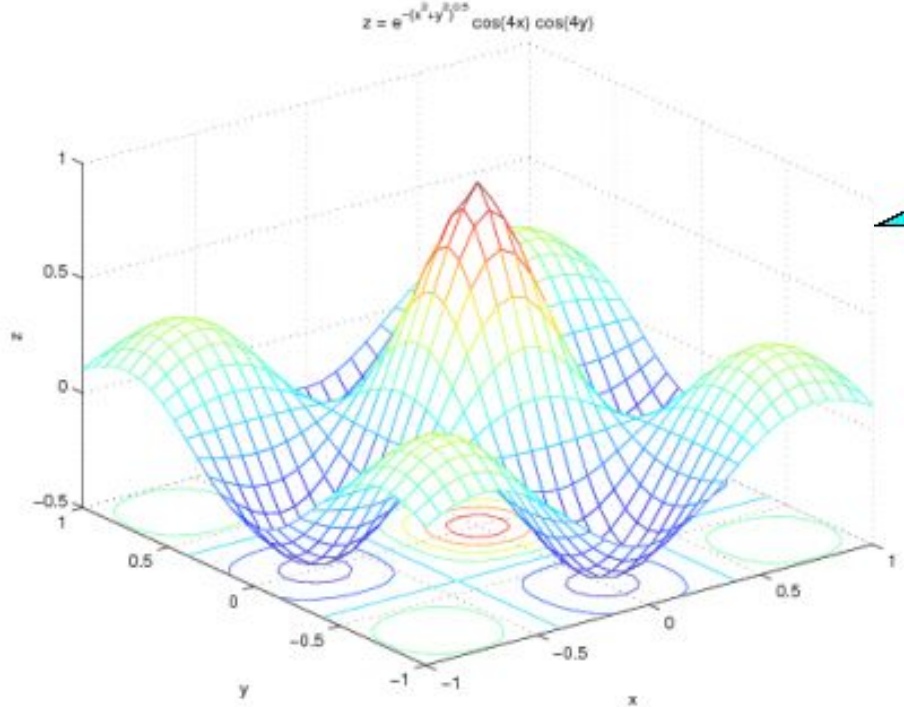




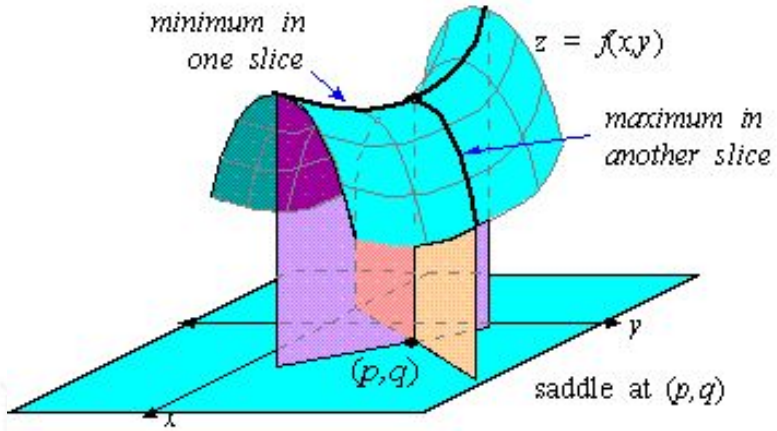
Why do we not get stuck in bad local minima?

- Local minima are close to global minima in terms of errors
- Saddle points are much more likely at higher portions of the error surface (in high-dimensional weight space)
- SGD (and other techniques) allow you to escape the saddle points

Error surfaces and saddle points

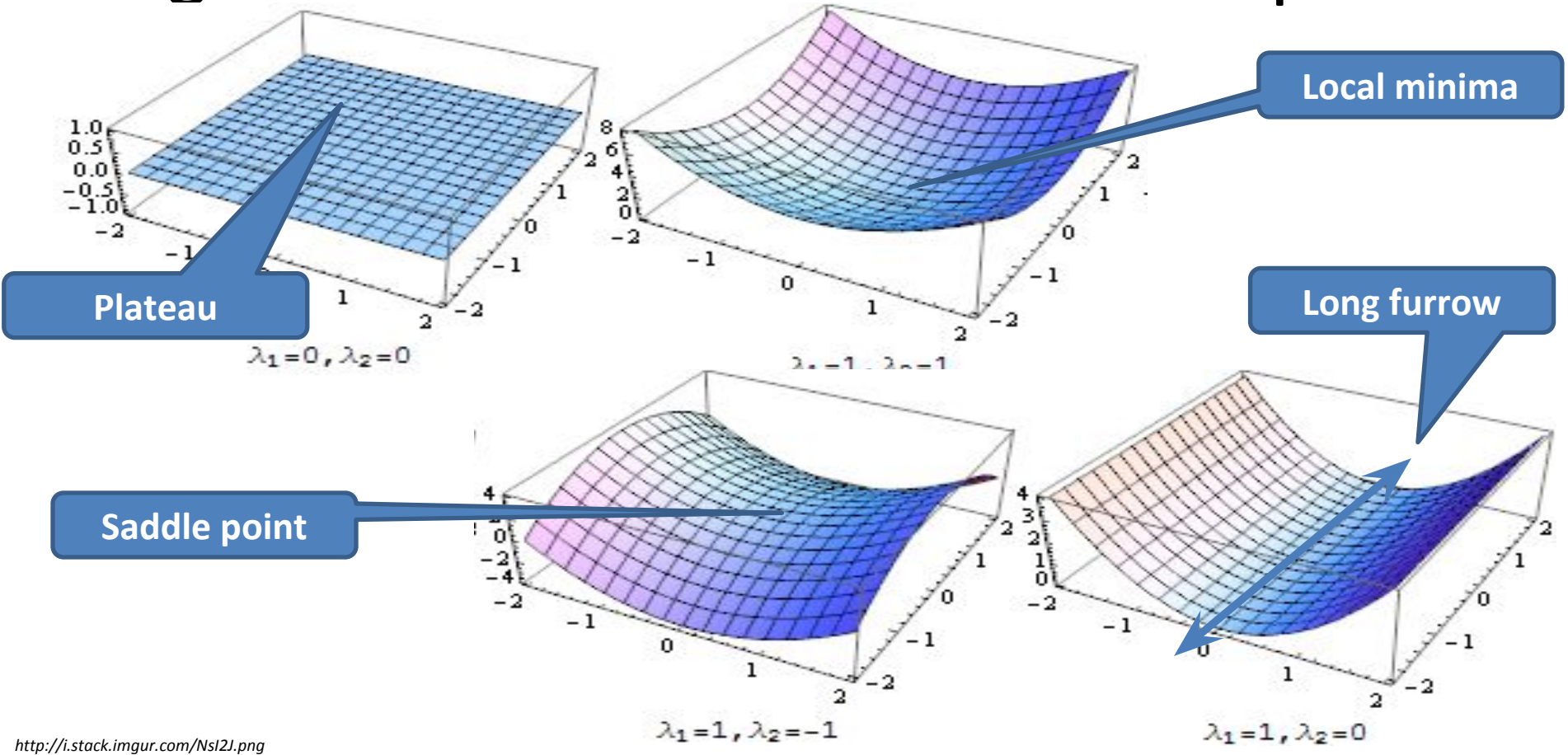


http://pundit.pratt.duke.edu/piki/images/thumb/0/0a/SurfExp04.png/400px-SurfExp04.png



http://math.etsu.edu/multicalc/prealpha/Chap2/Chap2-8/10-6-53.gif

Eigenvalues of Hessian at critical points



http://i.stack.imgur.com/NsI2J.png

GD vs. Newton's method

- Gradient descent is based on first-order approx.

$$f(\theta^* + \Delta\theta) = f(\theta^*) + \nabla f^T \Delta\theta$$

$$\Delta\theta = -\eta \nabla f$$

- Newton's method is based on second order

$$f(\theta^* + \Delta\theta) = f(\theta^*) + \nabla f^T \Delta\theta + \frac{1}{2}\Delta\theta^T H \Delta\theta$$

$$\Delta\theta = -H^{-1} \nabla f$$

"Identifying and attacking the saddle pointproblem in high-dimensional non-convex optimization" Dauphin et al., NIPS'15

Disadvantages of 2nd order methods

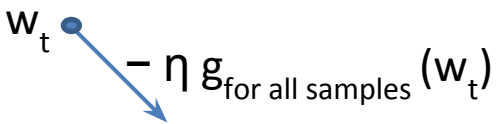
- Updates require $O(d^3)$ or at least $O(d^2)$
- May not work well for non-convex surfaces
- Get attracted to saddle points (how?)
- Not very good for batch-updates

Noise can be added in other ways to escape saddle points

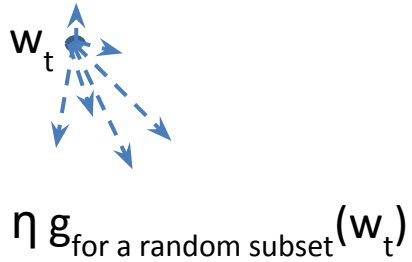
- Random mini-batches / SGD
- Add noise to the gradient or the update
- Add noise to the input

GD vs. SGD

- GD:
  - $w_{t+1} = w_t - \eta\, g_{\text{for all samples}}(w_t)$



- SGD momentum:
  - $w_{t+1} = w_t - \eta\, g_{\text{for a random subset}}(w_t)$

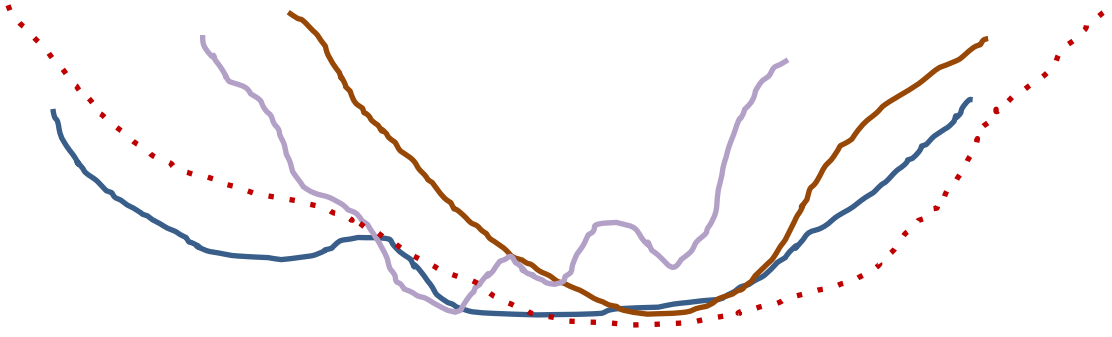


Compare GD with SGD

- GD requires more computations per update
- SGD is more noisy

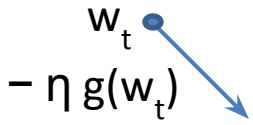
SGD helps by changing the loss surface

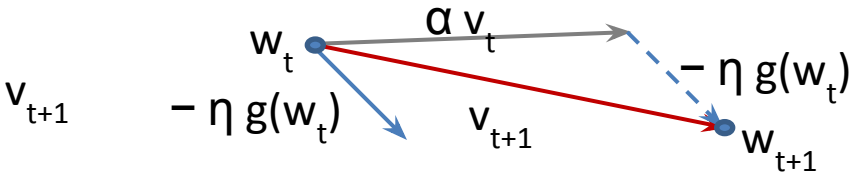
- Different mini-batches (or samples) have their own loss surfaces
- The loss surface of the entire training sample (dotted) may be different
- Local minima of one loss surface may not be local minima of another one
- This helps us escape local minima using stochastic or batch gradient descent
- Mini-batch size depends on computational resource utilization

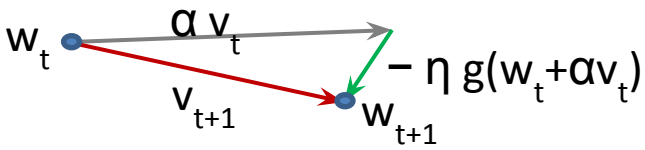


Classical and Nesterov Momentum

- GD:
  - $w_{t+1} = w_t - \eta\, g(w_t)$
- Classical momentum:
  - $v_{t+1} = \alpha\, v_t - \eta\, g(w_t)$;
  - $w_{t+1} = w_t + v_{t+1}$
- Nesterov momentum
  - $v_{t+1} = \alpha\, v_t - \eta\, g(w_t + \alpha v_t)$;
  - $w_{t+1} = w_t + v_{t+1}$
  - Better course-correction for bad velocity







Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Sutskever et al. "On the importance of initialization and momentum in deep learning", ICML 2013

Momentum

- Momentum optimizer adds an extra term to the update rule. This term is the weighted average of all previous gradients, which can be thought of as an average velocity of the previous steps. In Image 3 we see that taking an average of previous velocities cancels out the zig-zag motion in the vertical direction

$$v_{t+1} = \rho v_t + \nabla_W J$$
$$W = W - v_{t+1}$$

ρ is also called the dampening factor

Image 2: SGD without momentum

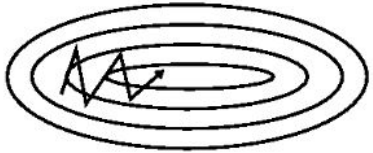
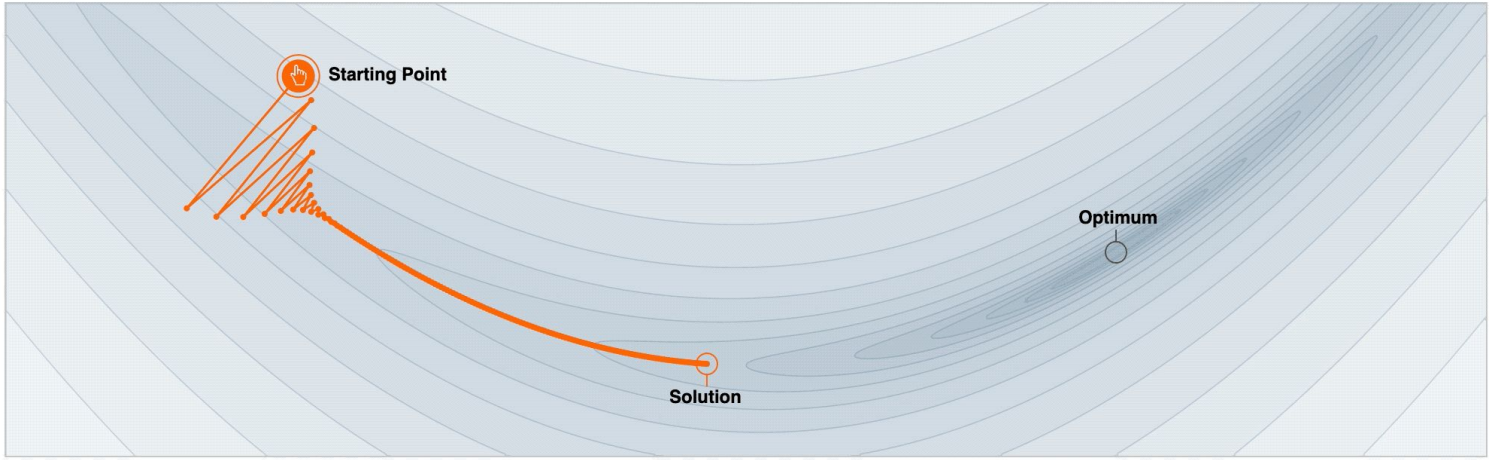


Image 3: SGD with momentum

This velocity term also helps go past saddle points where the gradient is 0

Image 2,3 source : https://ruder.io/optimizing-gradient-descent/index.html#momentum

Momentum



GIF source : https://mlfromscratch.com/optimizers-explained/#/

AdaGrad, RMSProp, AdaDelta

- Scales the gradient by a running norm of all the previous gradients
- Per dimension:

$$w_{t+1} = w_t - \eta \frac{g(w_t)}{\sqrt{\sum_{i=1}^{t} g(w_t)^2 + \varepsilon}}$$

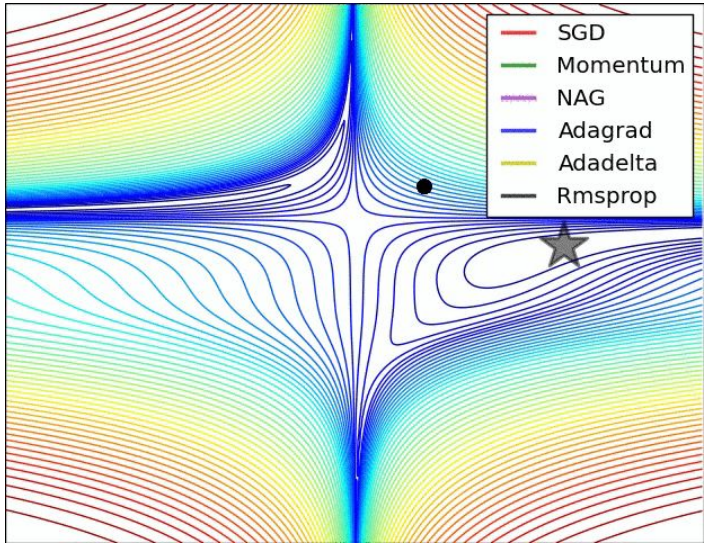
- Automatically reduces learning rate with t
- Parameters with small gradients speed up
- RMSProp and AdaDelta use a forgetting factor in grad squared so that the updates do not become too small

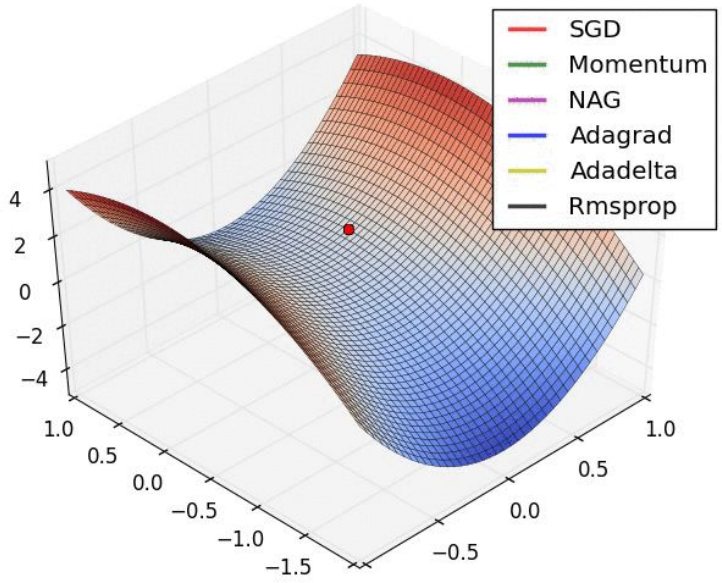
Adam optimizer combines AdaGrad and momentum

- Initialize
  - $m_0 = 0$
  - $v_0 = 0$
- Loop over t
  - $g_t = \nabla_w f_t(w_{t-1})$ — Get gradient
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ — Update first moment (biased)
  - $v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2$ — Update second moment (biased)
  - $\hat{m}_t = m_t / (1 - {\beta_1}^t)$ — Correct bias in first moment
  - $\hat{v}_t = v_t / (1 - {\beta_2}^t)$ — Correct bias in second moment
  - $w_t = w_{t-1} - \alpha \hat{m}_t / \sqrt{\hat{v}_t + \varepsilon}$ — Update parameters

"ADAM: A method for stochastic optimization" Kingma and Ba, ICLR'15

Visualizing optimizers





Source: http://ruder.io/optimizing-gradient-descent/index.html

Contents

- Weight initialization
- Making GD faster
- **Helping deep NNs train better**

Is achieving global minima important?

- Global minima for the training data may not be the global minima for the validation or test data

- Local minimas are often good enough

"The Loss Surfaces of Multilayer Networks" Choromanska et al. JMLR'15

Under certain assumptions, theoretically also they are of high quality

- Results:
  - Lowest critical values of the random loss form a band
  - Probability of minima outside that band diminishes exponentially with the size of the network
  - Empirical verification
- Assumptions:
  - Fully-connected feed-forward neural network
  - Variable independence
  - Redundancy in network parametrization
  - Uniformity

"The Loss Surfaces of Multilayer Networks" Choromanska et al. JMLR'15

Empirically, most minima are of high quality

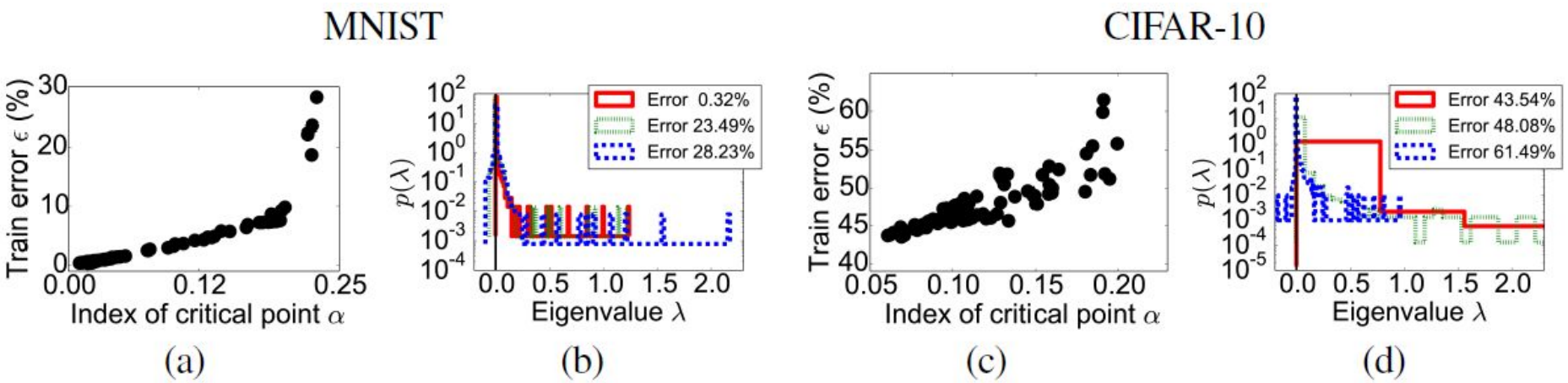
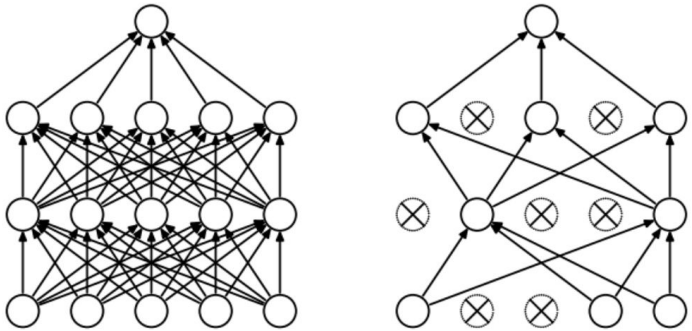


Figure 1: (a) and (c) show how critical points are distributed in the ϵ–α plane. Note that they concentrate along a monotonically increasing curve. (b) and (d) plot the distributions of eigenvalues of the Hessian at three different critical points. Note that the y axes are in logarithmic scale. The vertical lines in (b) and (d) depict the position of 0.

"Identifying and attacking the saddle point problem in high-dimensional non-convex optimization" Dauphin et al., NIPS'15

Dropout Regularization

- Dropout is a regularisation technique used while training neural networks. Like weight decay, dropout tends to make the network less dependent on some particular neurons.
- It is implemented by randomly removing (by specifying a probability of dropping) some neurons during each training step. This random removal prevents the network from relying on some particular neurons.



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

Source : Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research. 2014 Jan 1;15(1):1929-58.

Batch Normalization

- Why normalize an activation?
  - Without normalization, some gradients would be very small or very large
  - Already, input normalization and ZCA seem helpful
- Normalization has to be a part of the network
  - We cannot just normalize the activations without worrying about backprop; biases can blow up
  - First normalize each activation with respect to its mini-batch; this is a differentiable transform
  - Then, let the network learn a scale and shift parameter for that activation
  - How would this work for convolution? Consider each location as a part of the mini-batch

Source: “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” Ioffe and Szegedy, 2015

Batch Normalization

- Without normalization, we can face the vanishing or exploding gradients problem.
- Input normalization can only help the initial layers ; we also need normalization in the hidden layers.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

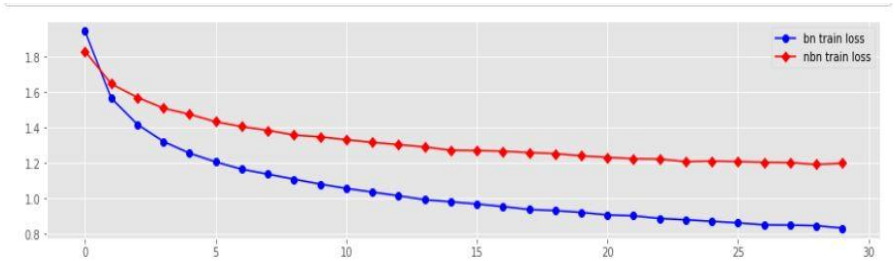
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

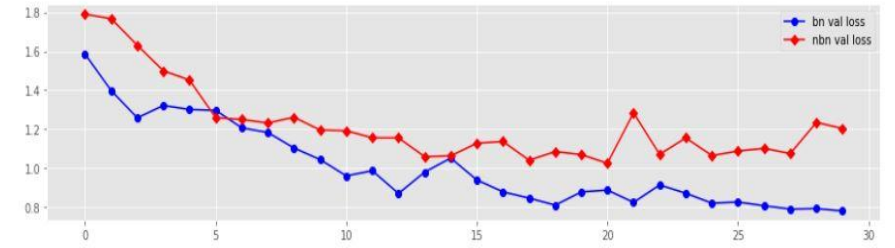
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

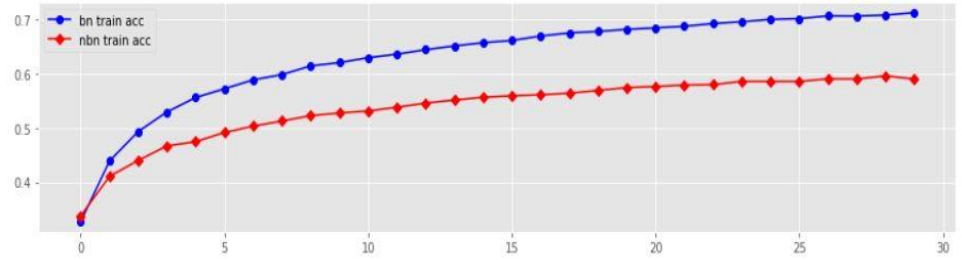
Normalization is a differentiable transform!
Even after normalization, the biases can blow up during backprop so we let the network learn appropriate scaling of the normalized outputs

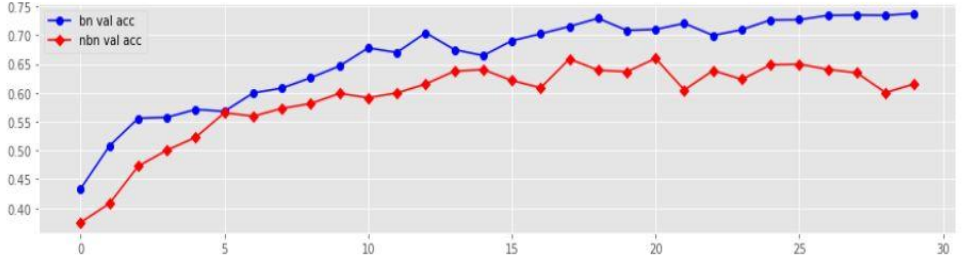
Source: “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” Ioffe and Szegedy, 2015

Effect of Batch Normalization









The blue curve represents a neural network with batch normalization and the red one for a neural network with the same architecture except for batch norm. Batch norm gives us better results with much faster training!