CS 480/680
Introduction to Machine Learning

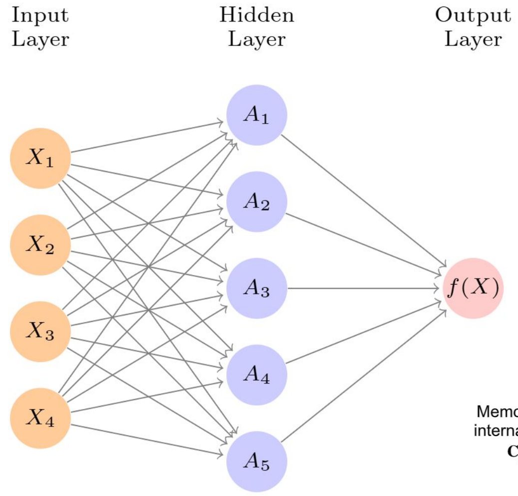
Lecture 17
Advanced Optimization

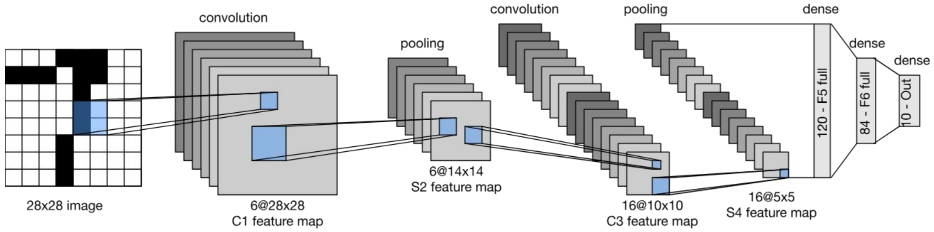
Kathryn Simone

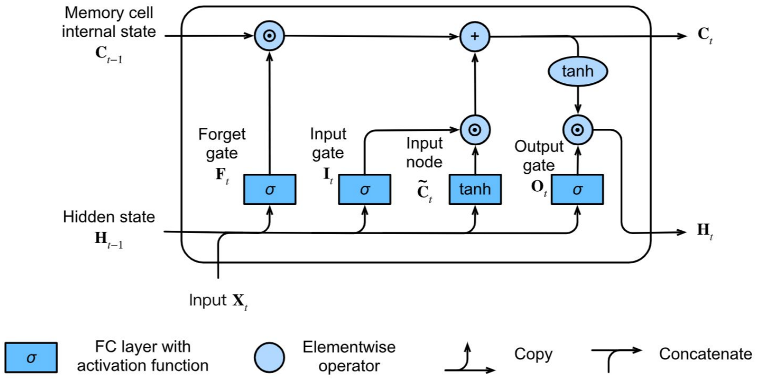
14 November 2024

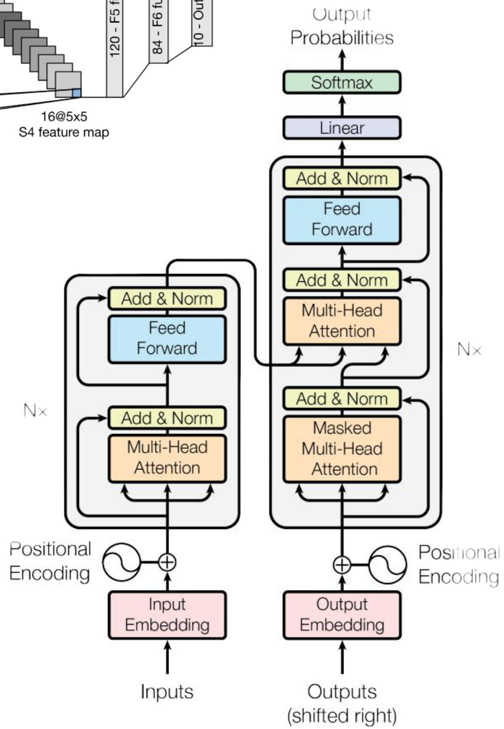
UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Specialized NN architectures for diverse tasks

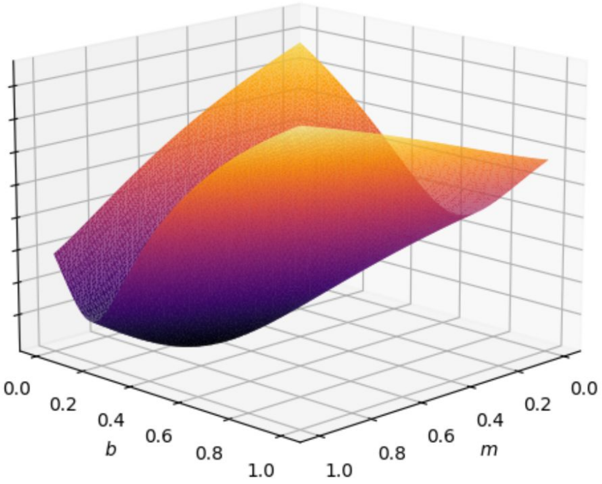




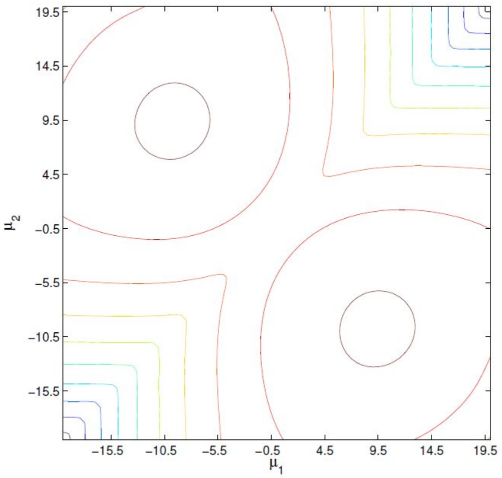




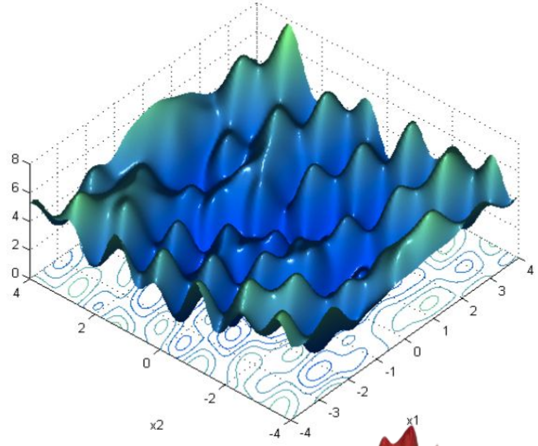
Challenge 1: Non-Convex Objectives

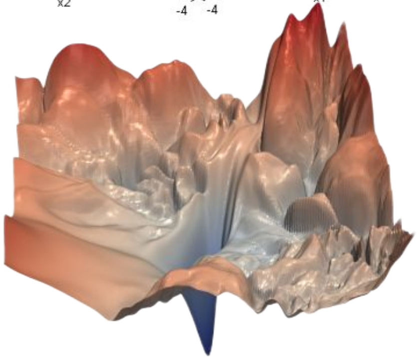


$$L = \|Aw - z\|_2^2$$



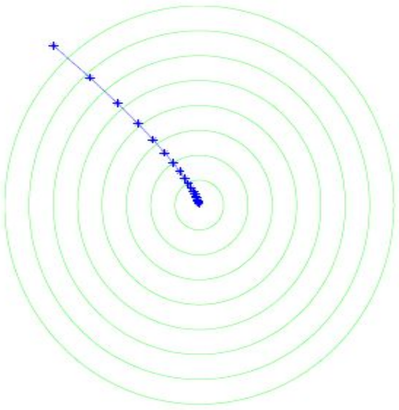
$$\log \mathcal{L}(\pi, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2 \mid \boldsymbol{X}) = \sum_{i=1}^{n} \log \left[(1-\pi)\mathcal{N}_{\mu_1,\sigma_1^2}(x) + \pi \mathcal{N}_{\mu_2,\sigma_2^2}(x) \right]$$

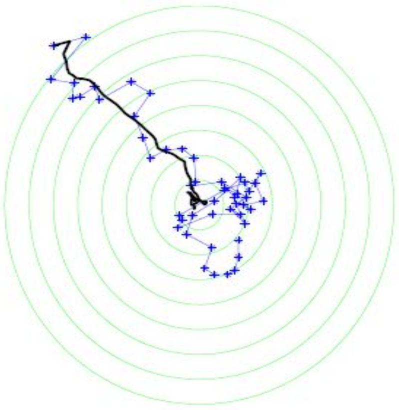


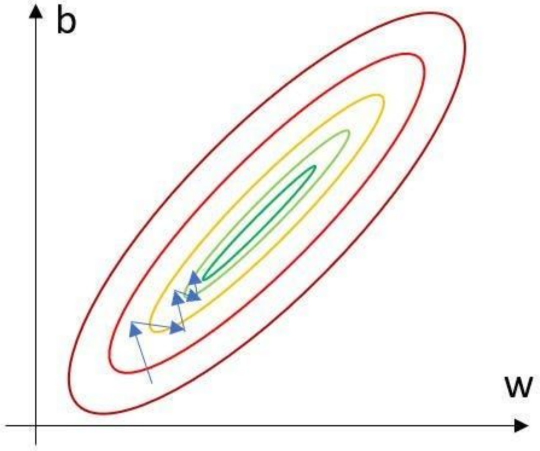


Left: Lecture 2; Middle: Lecture 12; Right, top: Lecture 13; Right, bottom: Lecture 2

Challenge 2: Ill-Conditioned Loss Landscapes





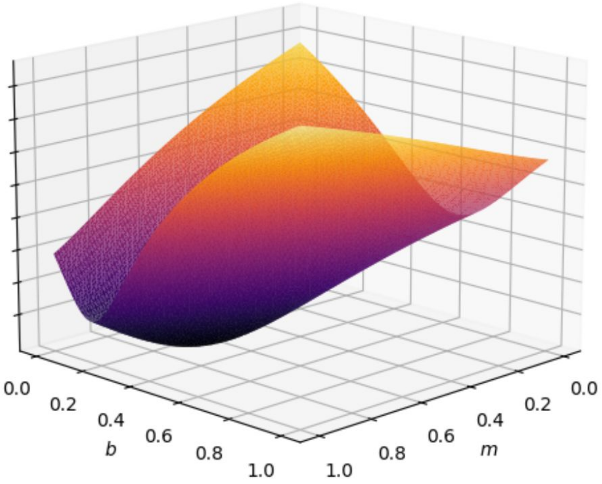


$$\nabla l_w(x, y) = \frac{1}{n}\sum_{i=1}^{n} \nabla_w l_{w,t-1}(x_i, y_i)$$

$$\nabla l_w(x, y) = \frac{1}{m}\sum_{i=1}^{n} \nabla_w l_{w,t-1}(x_i, y_i)$$

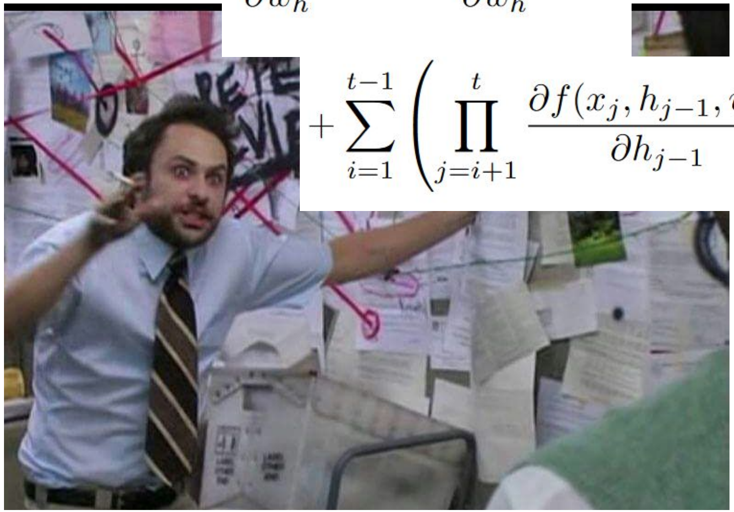
Left & Middle: Lecture 5; Right: Lecture 13

Challenge 3: Complicated gradients

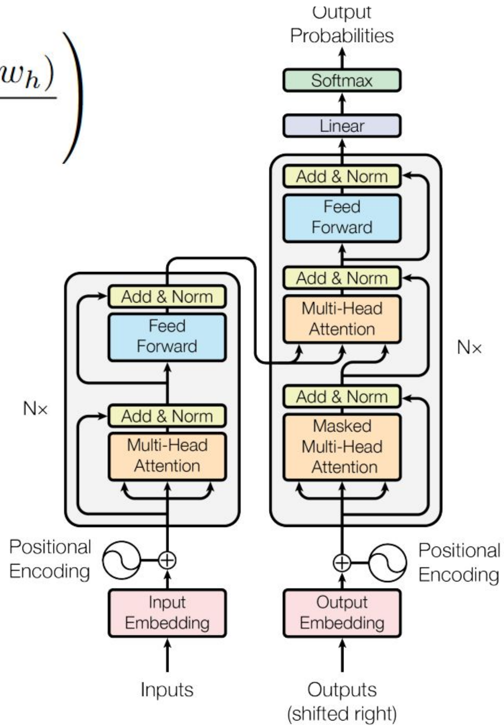


$$\nabla_w L = 2A^T Aw - 2A^T z$$

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}} \right)$$



$$\nabla L = ????$$



Left: Lecture 5; Middle: Lecture 15; Right: Lecture 16

Key questions

I. What are some parameter initialization strategies for SGD?

II. Can we navigate the loss landscape more efficiently?

III. How can we compute gradients reliably?

Key questions

I. **What are some parameter initialization strategies for SGD?**

II. Can we navigate the loss landscape more efficiently?

III. How can we compute gradients reliably?

Symmetry must be broken at initialization

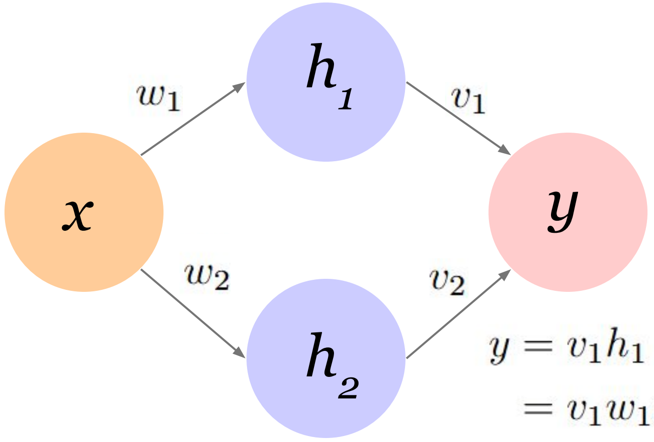
Suppose we initialize:

$$w_1 = w_2 = a, \text{ and}$$
$$v_1 = v_2 = b$$

At the first update, we compute:

$$\frac{\partial y}{\partial w_1} = v_1 x = bx$$
$$\frac{\partial y}{\partial w_2} = v_2 x = bx$$

$$h_1 = w_1 x$$



$$y = v_1 h_1 + v_2 h_2$$
$$= v_1 w_1 x + v_2 w_2 x$$

$$h_2 = w_2 x$$

Scale of weight initialization matters

Avoid exploding gradients
Avoid chaos in RNNs
Avoid saturation for some activations

Stronger symmetry-breaking
Avoid vanishing gradients

Small weights

Large weights

To break symmetry we start with small random weights. Variants on the learning procedure have been discovered independently by David Parker (personal communication) and by Yann Le Cun[3].

Rumelhart, Hinton, and Williams, 1986

Initialization to preserve the activation and gradient variances between layers

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j$$

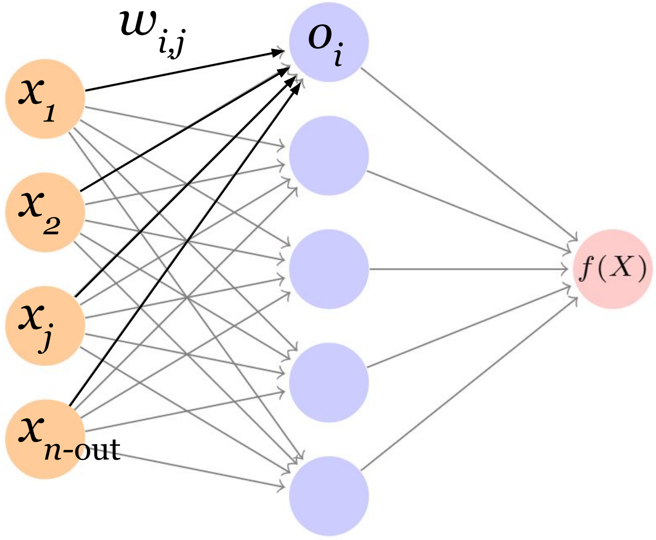
$$E[o_i] = 0$$

$$\begin{aligned}
\text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2
\end{aligned}$$

$$n_{\text{in}}\sigma^2 = 1 \qquad n_{\text{out}}\sigma^2 = 1$$

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1$$

$$\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

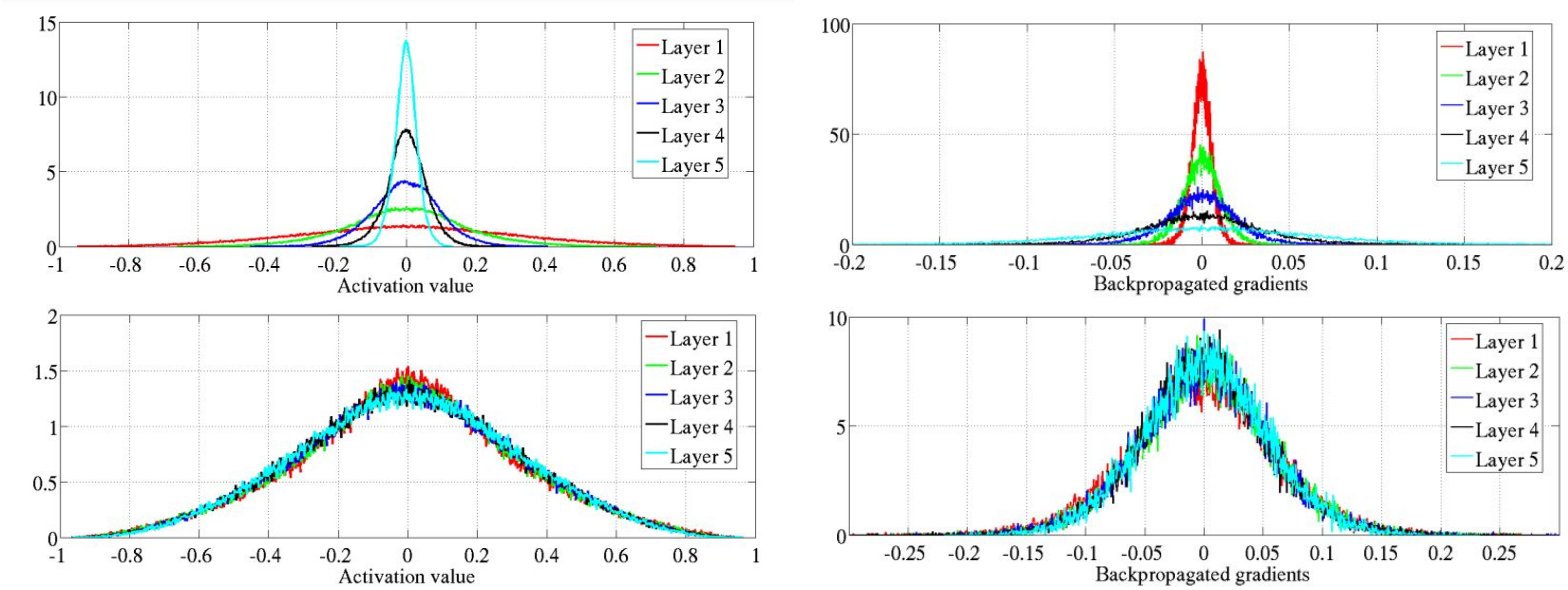


$$w_{i,j} \sim \mathcal{N}\left(0, \sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

$$w_{i,j} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}} \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

Where we have used $\text{Var}[X \sim \mathcal{U}(-a, a)] = \dfrac{a^2}{3}$

Xavier initialization



Glorot and Bengio, 2010

Other weight initializations

- Orthogonal initialization
  - Initialize weights using random orthogonal matrices
  - Scaling factors for specific activation funtions
- Sparse initialization (Martens, 2010)
  - Scaling heuristics can cause initial weights to become small when layers become large
  - Initialize each unit to have k non-zero weights
  - Aims to keep the preactivation independent of the number of inputs
  - Limitation: Introduces bias that must be “corrected” by the gradient descent

Bias initialization

Biases on hidden units typically initialized to zero

Bias on the output

- Can select to reflect the marginal statistics in the training dataset
- E.g. imbalanced dataset on a classification problem

Key questions

I. What are some parameter initialization strategies for SGD?

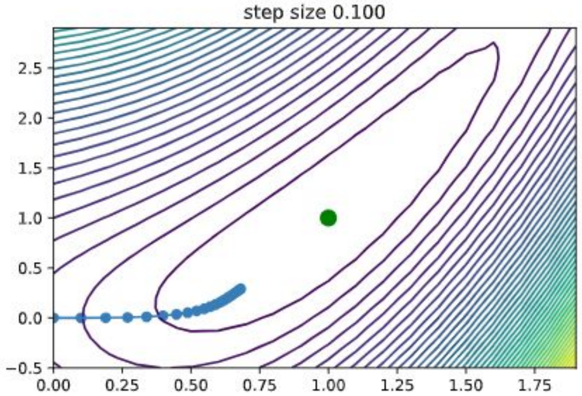
II. **Can we navigate the loss landscape more efficiently?**

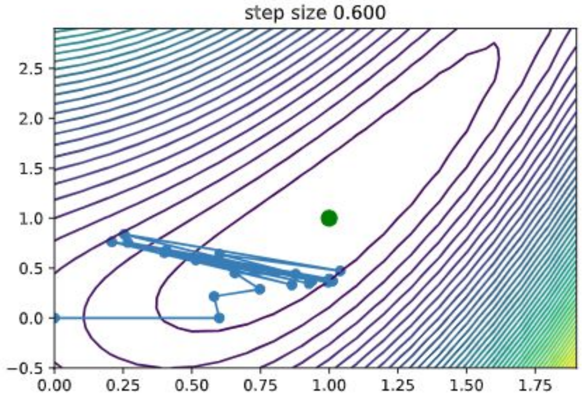
III. How can we compute gradients reliably?

Challenges with (stochastic) gradient descent

In stochastic gradient descent, we estimate the gradient of the loss with respect to the parameters using a minibatch of m samples, and update the parameter vector w in the opposite direction, scaled by step size η:

$$\theta_t = \theta_{t-1} - \eta \nabla \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta l_{\theta,t-1}(x_i, y_i)$$





Probabilistic Machine Learning, Section 8.2

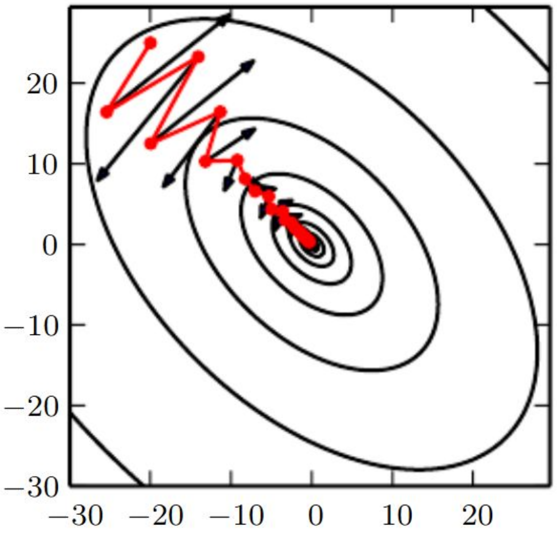
Momentum:
Keep moving in the direction you have tended to in the past

Let $g_{t-1} = \nabla \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta l_{\theta,t-1}(x_i, y_i)$. Then the parameter update for SGD with momentum is

$$\theta_t = \theta_{t-1} + v_t$$
$$v_t = \alpha v_{t-1} - \epsilon g_{t-1}$$

Where v_t is the "velocity" of a point particle at position θ_t, and $\alpha \in (0,1)$ is a hyperparameter that controls the contribution of previous gradient steps as compared to the most recent gradient step.

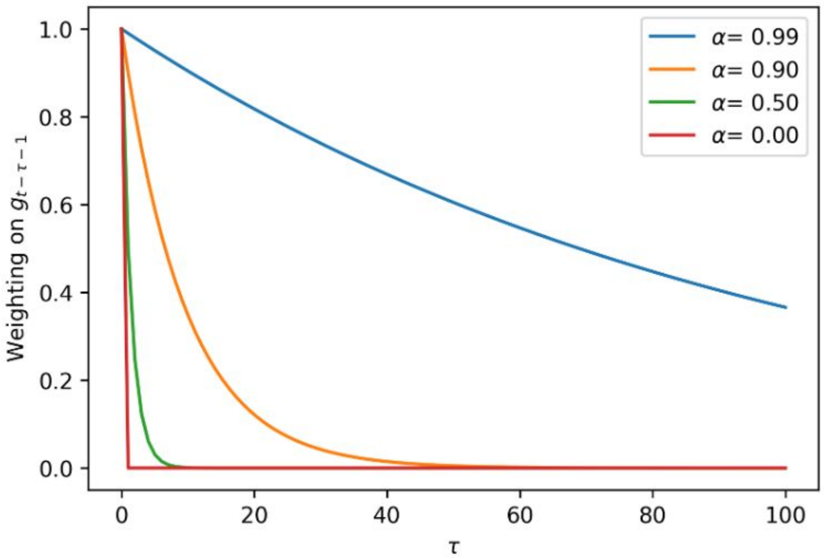
$$\Delta\theta_t = \alpha v_{t-1} - \epsilon g_{t-1}$$



Deep Learning, Section 8.3

Momentum accumulates an exponentially-weighted average over gradient steps

$$
\begin{aligned}
v_t &= \alpha v_{t-1} - \epsilon g_{t-1} \\
&= \alpha(\alpha v_{t-2} - \epsilon g_{t-2}) - \epsilon g_{t-1} \\
&= \alpha(\alpha(\alpha v_{t-3} - \epsilon g_{t-3})) - \epsilon g_{t-2}) - \epsilon g_{t-1} \\
&= -\epsilon(g_{t-1} + \alpha g_{t-2} + \alpha^2 g_{t-3}) + \alpha^3 v_{t-3} \\
&\vdots \\
&= -\epsilon \sum_{\tau=0}^{t-1} \alpha^\tau g_{t-\tau-1}
\end{aligned}
$$



Special cases:

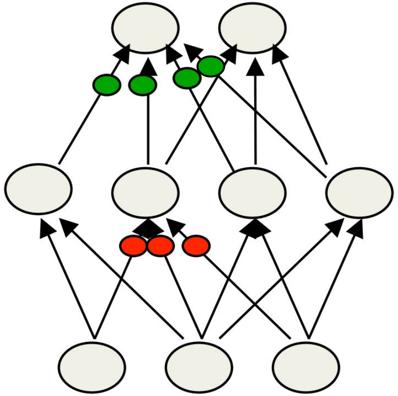
$$\alpha = 0$$

$$\implies \theta_t = \theta_{t-1} - \epsilon g_{t-1}$$

$$g_{t-1} = g \;\forall t$$

$$\implies ||\Delta\theta_t|| = \frac{\epsilon||g||}{1-\alpha}$$

A separate, adaptive learning rate for each parameter: Adagrad



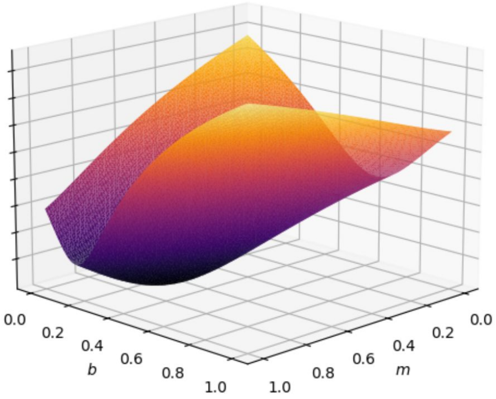
$$\theta_{t+1,j} = \theta_{t,j} - \eta_t \frac{1}{\sqrt{s_{t,j} + \epsilon}} g_{t,j}$$

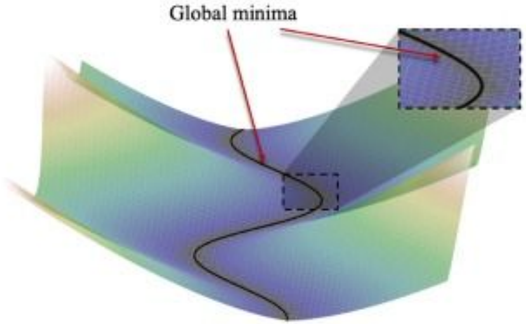
Where $j = 1 : p$ indexes the dimensions of the parameter vector, and

$$s_{t,j} = \sum_{i=1}^{t} g_{i,j}^2$$

is the sum of squared gradients, and ϵ avoids division by zero.

$$\Delta\boldsymbol{\theta_t} = -\eta \frac{1}{\sqrt{\boldsymbol{s_t} + \epsilon}} \boldsymbol{g_t}$$





https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

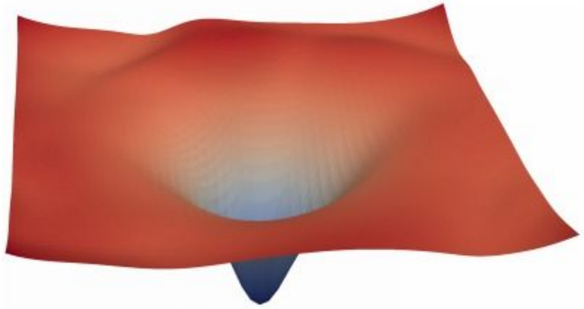
RMSProp:
Discard gradient information from the extreme past

The RMSProp update is also given by

$$\Delta \boldsymbol{\theta_t} = -\eta \frac{1}{\sqrt{\boldsymbol{s_t} + \epsilon}} \boldsymbol{g_t}.$$

However it uses an exponentially-weighted moving average of past squared gradients:

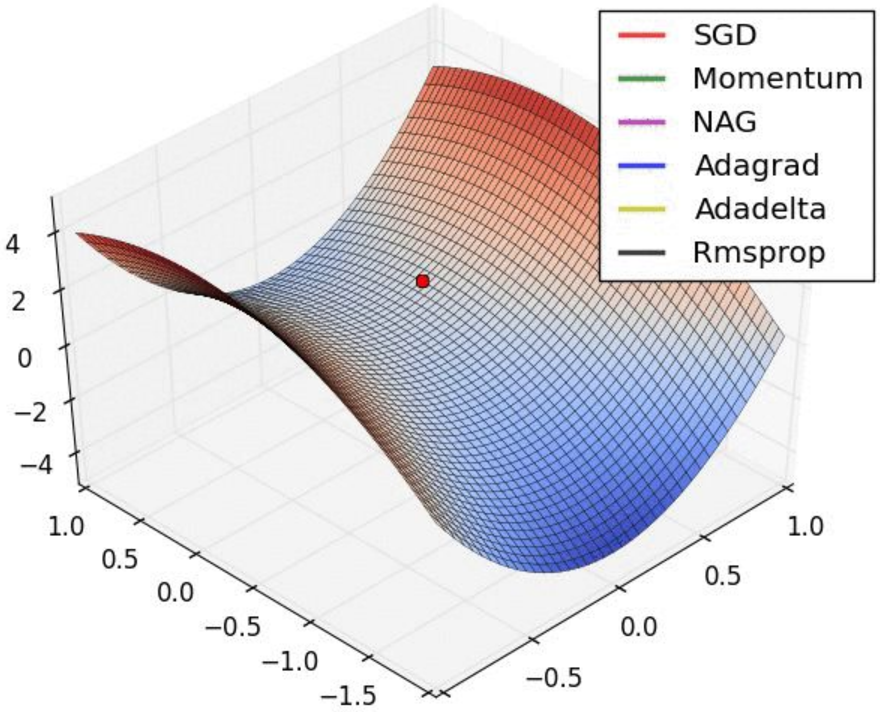
$$s_{t+1,j} = \beta s_{t,j} + (1-\beta) g_{t,j}^2$$



Where $\beta \in (0,1)$ is a hyperparameter.
When $\beta \approx 0.9$

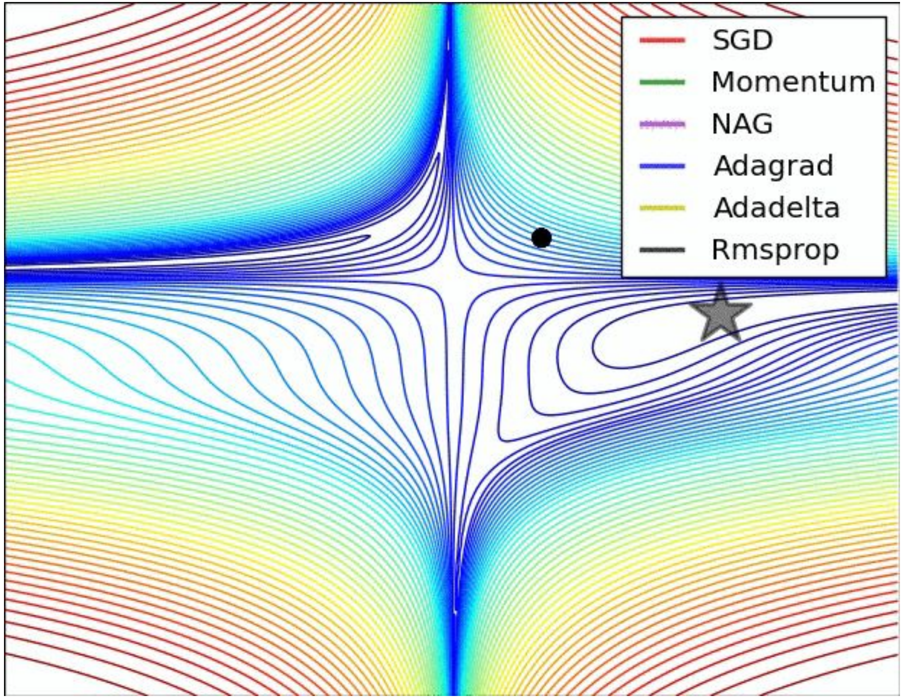
$$\sqrt{s_{t,j}} \approx \sqrt{\frac{1}{t} \sum_{\tau=1}^{t} g_{\tau,d}^2}$$
$$\approx \mathrm{RMS}(g_{1:t,j})$$

Comparing optimizers on non-convex functions



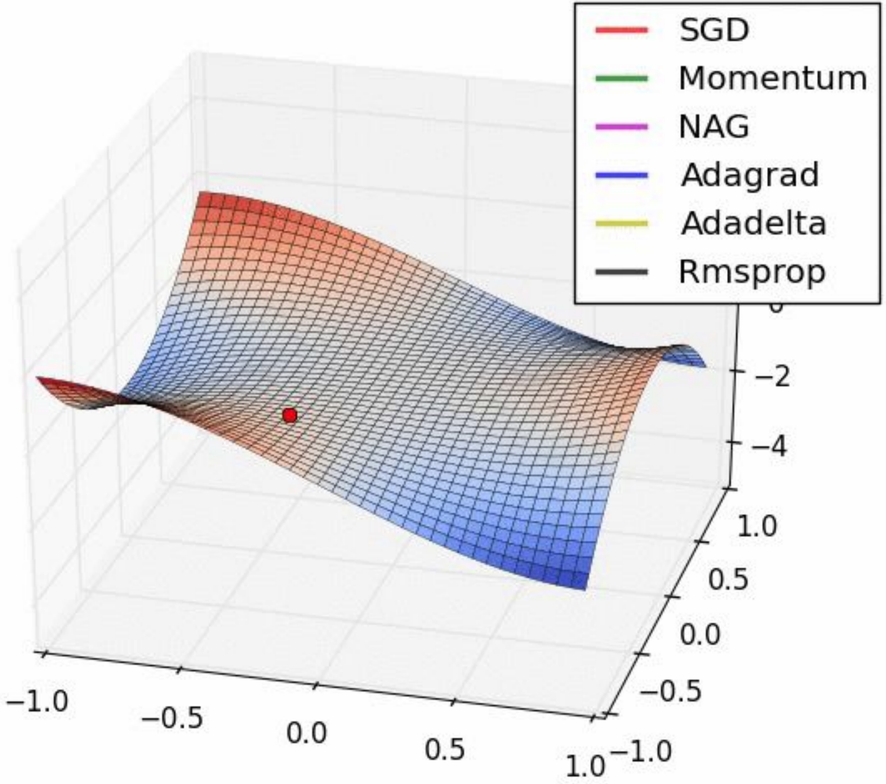
Original animations from Alec Radford; https://imgur.com/a/visualizing-optimization-algos-Hqolp

Comparing optimizers on non-convex functions



Original animations from Alec Radford; https://imgur.com/a/visualizing-optimization-algos-Hqolp

Comparing optimizers on non-convex functions



Original animations from Alec Radford; https://imgur.com/a/visualizing-optimization-algos-Hqolp

Adam: Adaptive moment estimation

Estimate first and second moments:

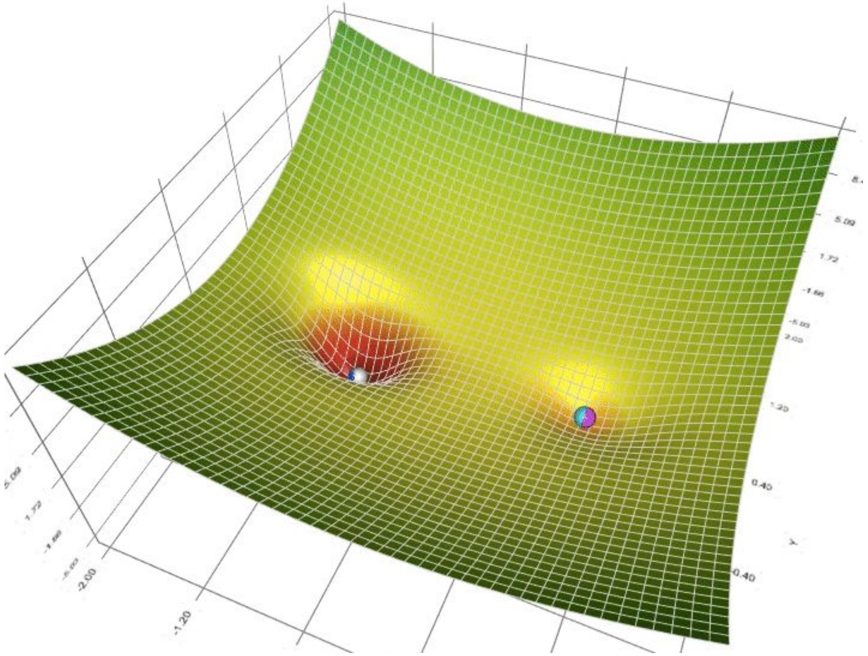
$$\boldsymbol{m_t} = \beta_1 \boldsymbol{m_{t-1}} + (1-\beta_1)\boldsymbol{g_t}$$
$$\boldsymbol{s_t} = \beta_2 \boldsymbol{s_{t-1}} + (1-\beta_2)\boldsymbol{g_t}^2$$

Bias correction:

$$\boldsymbol{\hat{m}_t} = \boldsymbol{m_t}/(1-\beta_1^t)$$
$$\boldsymbol{\hat{s}_t} = \boldsymbol{s_t}/(1-\beta_2^t)$$

$$\Delta\boldsymbol{\theta_t} = -\eta \frac{1}{\sqrt{\boldsymbol{\hat{s}_t}} + \epsilon}\boldsymbol{\hat{m}_t}$$

Comparing optimizers on non-convex functions



Gradient descent
Momentum
Adagrad
RMSProp
Adam

https://github.com/lilipads/gradient_descent_viz

Optimizer	Hyperparameters and some values
(Stochastic) Gradient Descent with Momentum	Momentum parameter α = 0.5, 0.9, 0.99 Learning rate ϵ
AdaGrad	Global learning rate η Constant for numerical stability $\epsilon = 10^{-7}$
RMSProp	Global learning rate η Decay rate β = 0.9 Constant for numerical stability $\epsilon = 10^{-7}$
Adam	Global learning rate η $\beta_1 = 0.9$, $\beta_2 = 0.999$, Constant for numerical stability $\epsilon = 10^{-6}$

Key questions

I. What are some parameter initialization strategies for SGD?

II. Can we navigate the loss landscape more efficiently?

III. How can we compute gradients reliably?

Automatic Differentiation (Autograd)

Hand-calculating derivatives is tedious and prone to errors.
Most modern deep learning libraries have built-in automatic differentiation to handle that for us.

`>> Jupyter Notebook Demo`

Aside: Talk from Yann LeCun at NeurIPS 2007: https://videolectures.net/videos/eml07_lecun_wia
At around 30 mins, advocates for automatic differentiation and explains how it works
Somewhat implies that others' inability to reproduce his results are because they don't use autograd

Now that we're at the end of the lecture, you should be able to…

- ★ Defend the need for **symmetry-breaking** in a neural network.
- ★ Recall three weight **initialization strategies** for neural networks.
- ★ Use appropriate **initialization strategies** to break symmetry prior to optimization.
- ★ Defend the need for **momentum- and/or adaptive learning** rate based optimization with reference to the limitations of **stochastic gradient descent** and the **conditioning of the loss-landscape**.
- ★ Write and interpret **gradient update equations** for **major momentum- and/or adaptive learning rate** based optimization techniques (**Momentum, Adagrad, RMSProp, Adam**).
- ★ Identify the **strengths and limitations** of different optimizers.
- ★ Use **autodifferentiation** to optimize a PyTorch model.