Topics:

- Optimization

CS 4644-DL / 7643-A
ZSOLT KIRA

- **Assignment 1 – Due today!!!**
  - **DO NOT SEARCH FOR CODE!!!!**

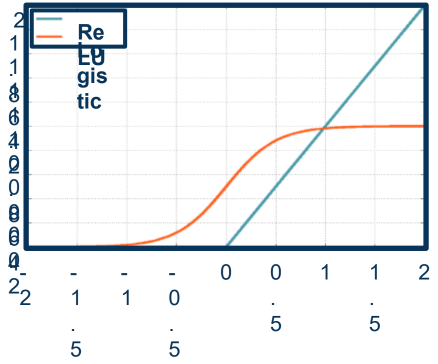
- **Assignment 2**
  - Implement convolutional neural networks

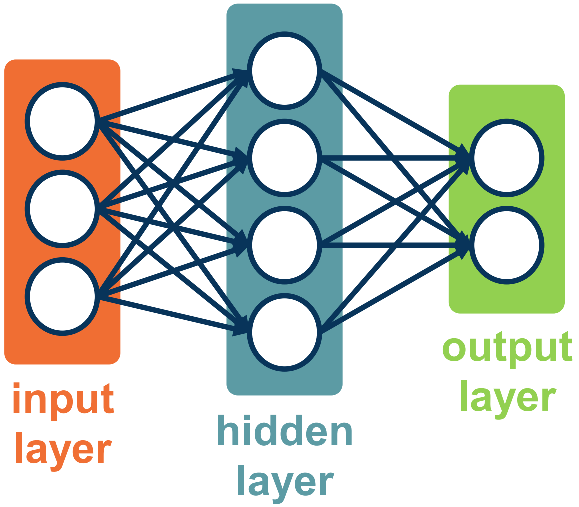
- **Quiz on Feb 11th**
  - Practice Quiz out today

- **Project Proposal:** Out and due **Feb 14th**

- **Meta OH**: We will have OH with Meta researchers (joint w/ OMSCS)
  - **OMSCS** Lessons (videos) linked as dropbox
  - **Full schedule and discussions on** https://ai-learning.org/

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\ w_{31} & w_{32} & \cdots & w_{3m} & b_3 \end{bmatrix}}_{\boldsymbol{W}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{bmatrix}}_{\boldsymbol{x}}$$





- Gradient Descent
- Compute gradients via chain rule
    - Backpropagation
    - Computation Graph + Automatic Differentiation

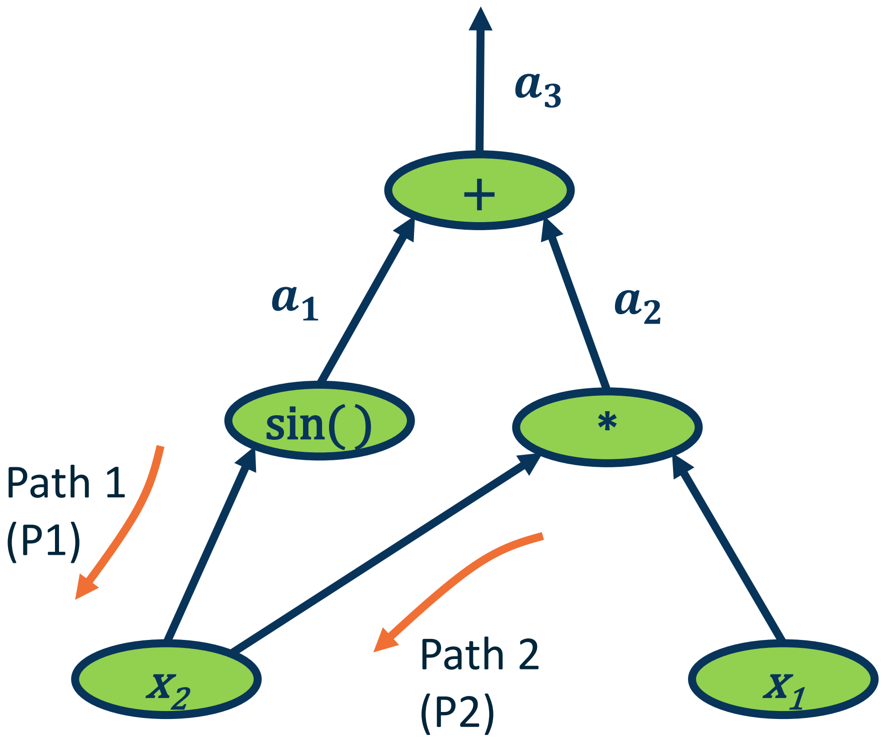
$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

So Far

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Example

$$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$$



$$\overline{a_3} = \frac{\partial f}{\partial a_3} = 1$$

$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3}\,1 = \overline{a_3}$$

$$\overline{a_2} = \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial a_2} = \overline{a_3}$$

$$\overline{x_2^{P1}} = \frac{\partial f}{\partial a_1}\frac{\partial a_1}{\partial x_2} = \overline{a_1}\ \cos(x_2)$$

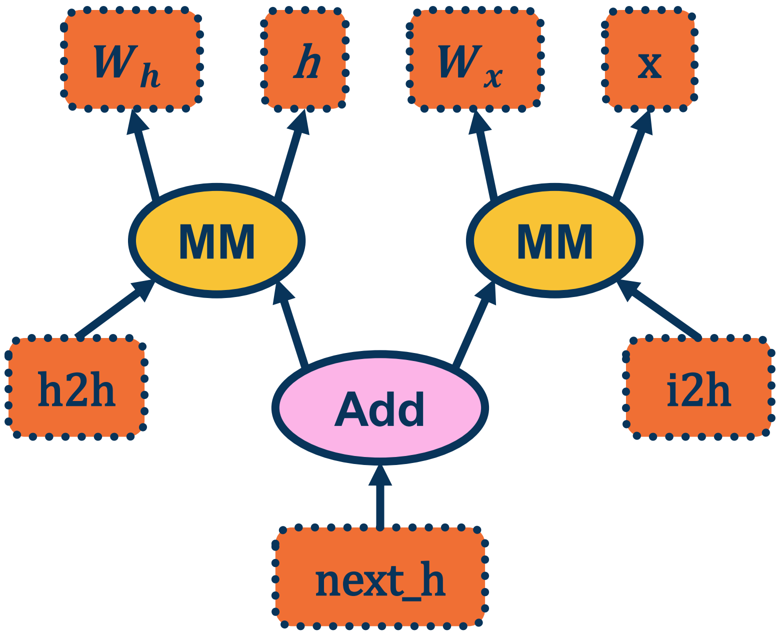
$$\overline{x_2^{P2}} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2}\frac{\partial (x1x2)}{\partial x_2} = \overline{a_2}x_1$$

Gradients from multiple paths **summed**

$$\overline{x_1} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_1} = \overline{a_2}x_2$$

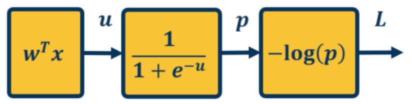
A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



(Note above)

Different Views of Equivalent Ideas



$\bar{L} = 1$

$\bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p}$

where $p = \sigma(w^T x)$ and $\sigma(x) = \frac{1}{1+e^{-x}}$

$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u} = \bar{p}\,\sigma(1-\sigma)$

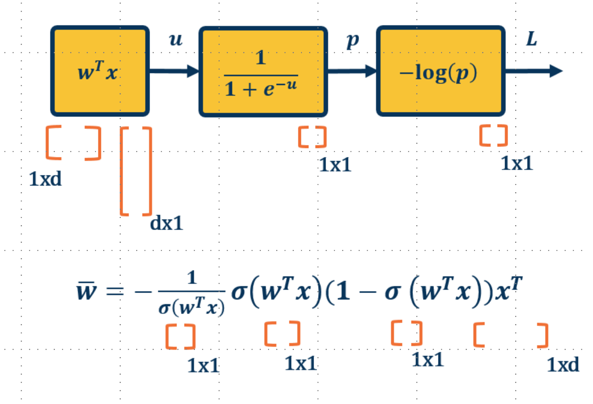
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

We can do this in a combined way to see all terms together:

$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

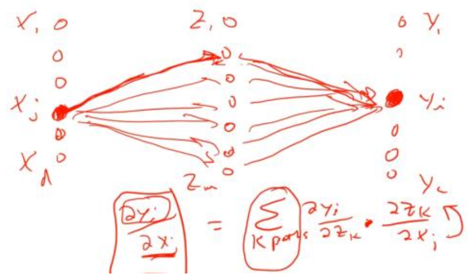
This effectively shows gradient flow along path from L to w

Computation Graph / Global View of Chain Rule



$$\bar{w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$

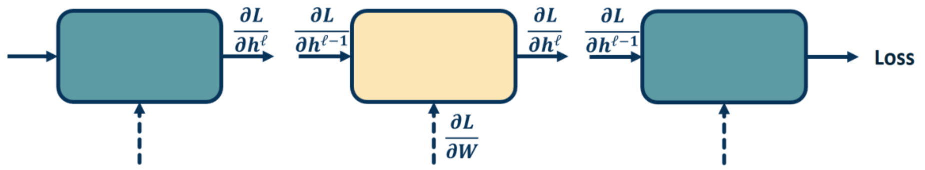
Computational / Tensor View



x_1
x_i
x_n
z_1
z_m
y_1
y_i
y_c
$\frac{\partial y_i}{\partial x_i} = \sum_{K \text{ parents}} \frac{\partial y_i}{\partial z_K} \cdot \frac{\partial z_K}{\partial x_i}$

Graph View

- We want to to compute: $\left\{\frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W}\right\}$

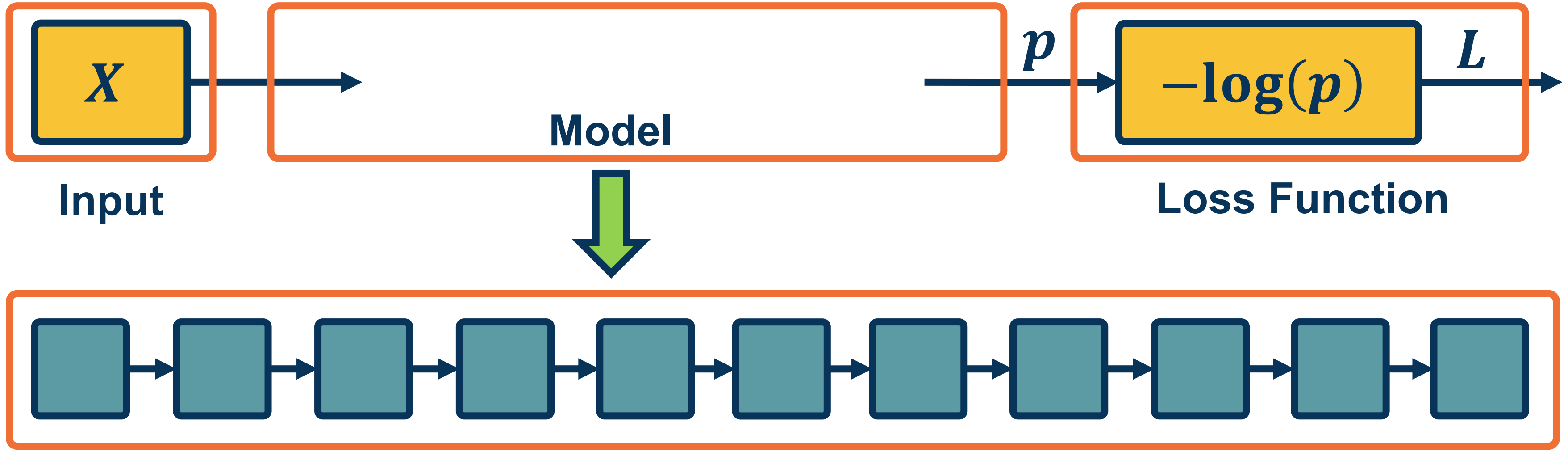


$\frac{\partial L}{\partial h^{\ell}}$
$\frac{\partial L}{\partial h^{\ell-1}}$
$\frac{\partial L}{\partial h^{\ell}}$
$\frac{\partial L}{\partial h^{\ell-1}}$
$\frac{\partial L}{\partial W}$
Loss

Backpropagation View (Recursive Algorithm)

Backpropagation, and automatic differentiation, allows us to optimize **any** function composed of differentiable blocks

- **No need to modify** the learning algorithm!
- The complexity of the function is only limited by **computation and memory**

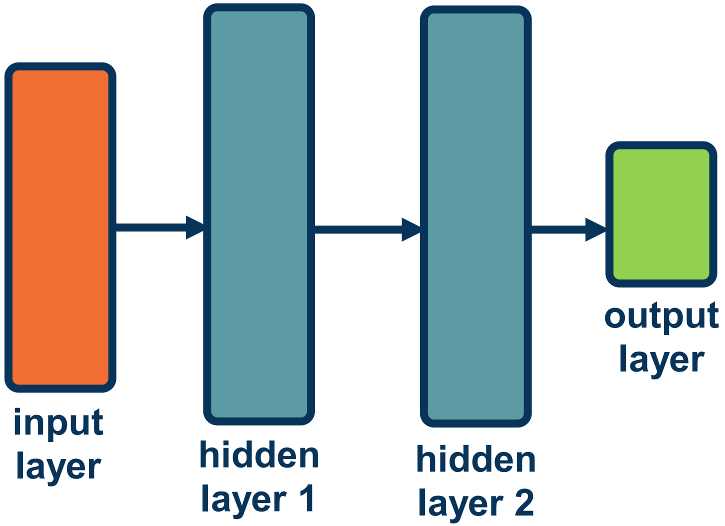


The Power of Deep Learning

A network with two or more hidden layers is often considered a **deep** model

Depth is important:

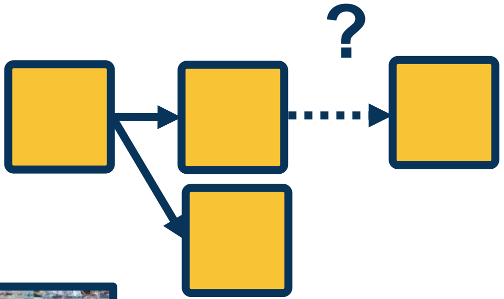
- Structure the model to represent an inherently compositional world
- Theoretical evidence that it leads to parameter efficiency
- Gentle dimensionality reduction (if done right)



Importance of Depth

There are still many design decisions that must be made:

- **Architecture**
- **Data Considerations**
- **Training and Optimization**
- **Machine Learning Considerations**



Machine Learning Considerations

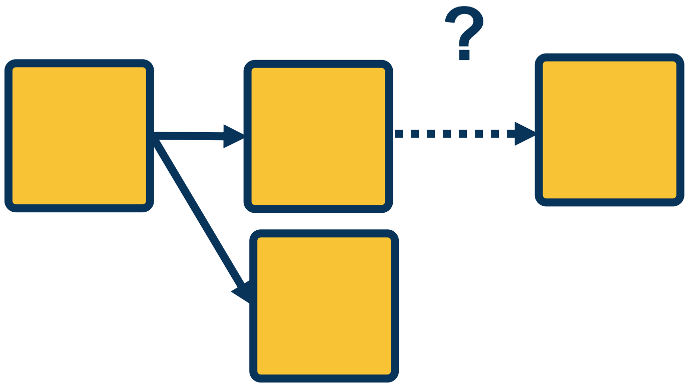
The practice of machine learning is complex: For your particular application you have to **trade off** all of the considerations together

- Trade-off between **model capacity** (e.g. measured by # of parameters) and **amount of data**
- Adding **appropriate biases** based on knowledge of the domain

Architectural Considerations

Determining what modules to use, and how to connect them is part of the **architectural design**

- Guided by the **type of data used** and its **characteristics**
  - Understanding your data is always the first step!
- **Lots of data types (modalities)** already have good architectures
  - Start with what others have discovered!
- **The flow of gradients** is one of the key principles to use when analyzing layers



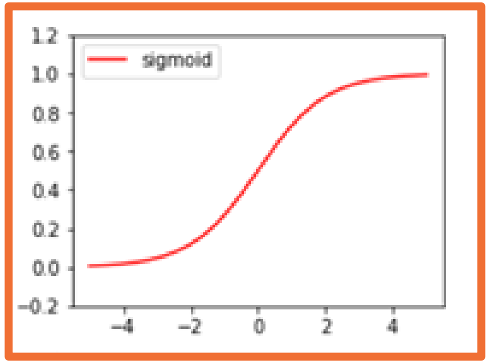
Designing the Architecture

Linear and Non-Linear Modules

- **Combination** of linear and non-linear layers
- Combination of **only** linear layers has same representational power as one linear layer
- **Non-linear layers** are crucial
  - Composition of non-linear layers **enables complex transformations of the data**

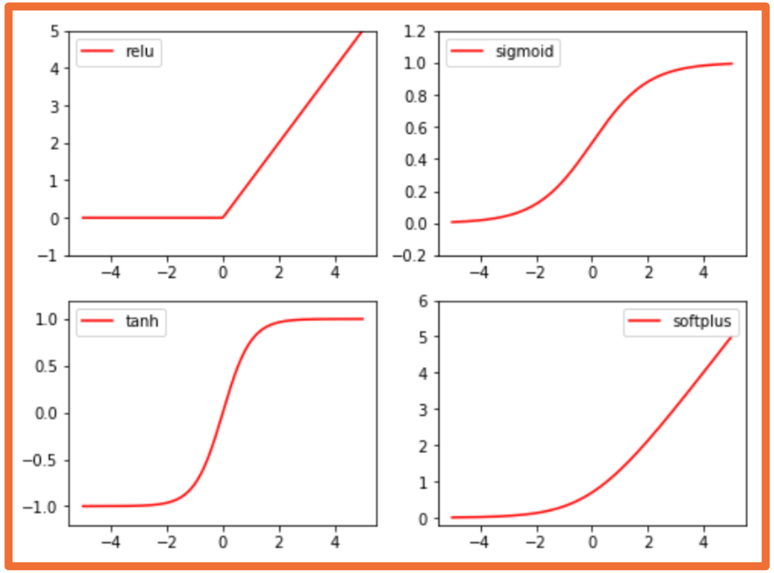
$$w_1^T(w_2^T(w_3^T x)) = w_4^T \mathbf{x}$$

$$\begin{tikzcd} \boxed{w^T x} \arrow[r, "u"] & \boxed{\frac{1}{1+e^{-u}}} \arrow[r, "p"] & \boxed{-\log(p)} \arrow[r, "L"] & {} \end{tikzcd}$$



Several aspects that we can **analyze**:

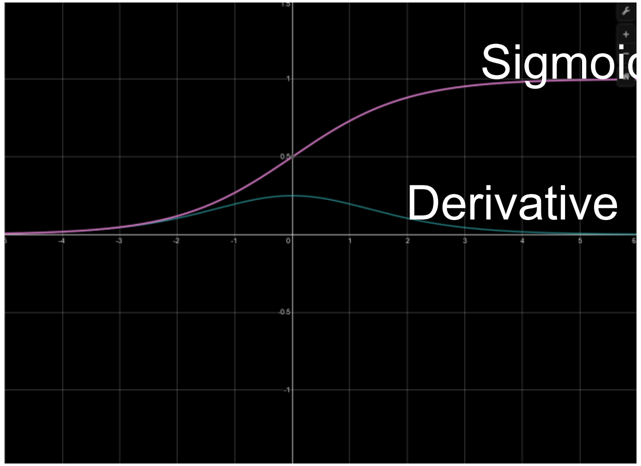
- Min/Max
- Correspondence between input & output statistics
- **Gradients**
  - At initialization (e.g. small values)
  - At extremes
- Computational complexity



Analysis of Non-Linear Function

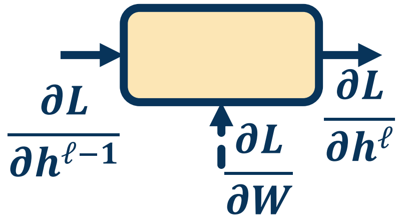
Sigmoid Function

- **Min:** 0, **Max:** 1
- Output **always positive**
- Saturates at **both ends**
- **Gradients**
  - Vanishes at both end
  - Always positive
- **Computation: Exponential term**



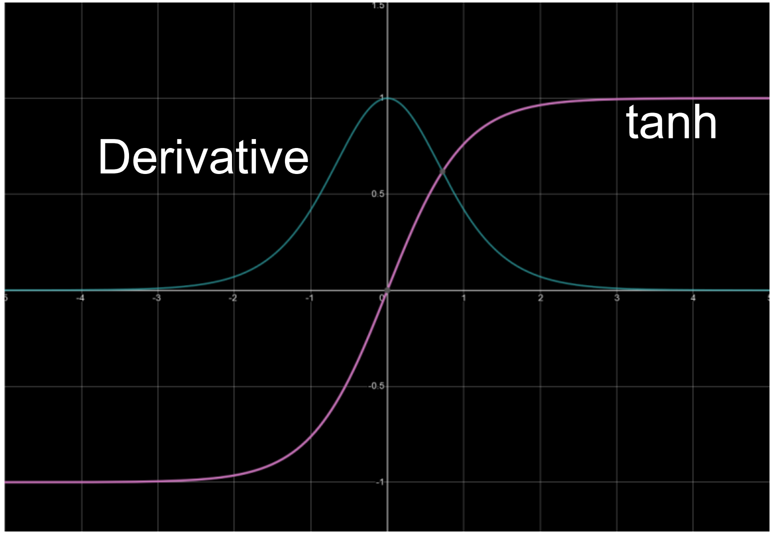
$$h^{\ell} = \sigma\left(h^{\ell-1}\right)$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$


$\frac{\partial L}{\partial h^{\ell-1}}$
$\frac{\partial L}{\partial W}$
$\frac{\partial L}{\partial h^{\ell}}$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$$

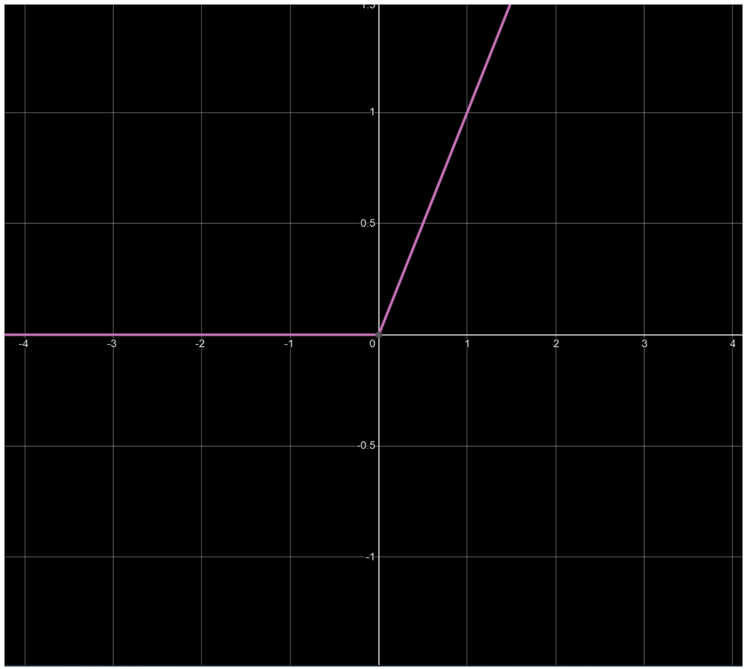
- **Min:** -1, **Max:** 1
  - **Centered**
- Saturates at **both ends**
- **Gradients**
  - Vanishes at both end
  - Always positive
- **Still somewhat computationally heavy**



$$h^{\ell} = tanh(h^{\ell-1})$$

Tanh Function

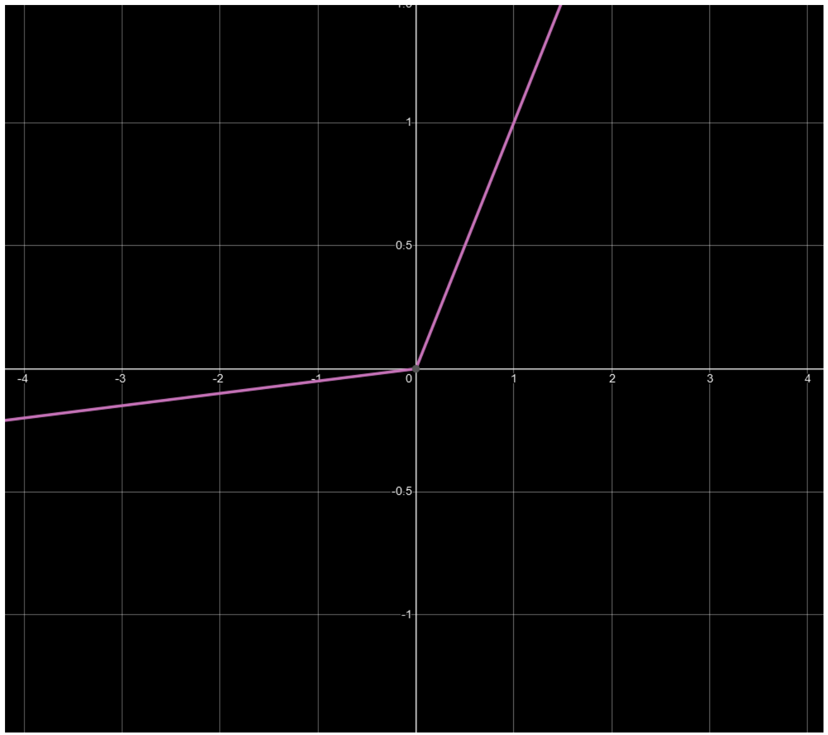
- **Min:** 0, **Max:** Infinity
- Output always **positive**
- **No saturation** on positive end!
- **Gradients**
  - $\mathbf{0 \text{ if } x \leq 0}$ (dead ReLU)
  - Constant otherwise (does not vanish)
- **Cheap to compute (max)**



$$h^{\ell} = max(0, h^{\ell-1})$$

Rectified Linear Unit

- **Min:** -Infinity, **Max:** Infinity
- **Learnable parameter!**
- **No saturation**
- **Gradients**
  - No dead neuron
- **Still cheap to compute**

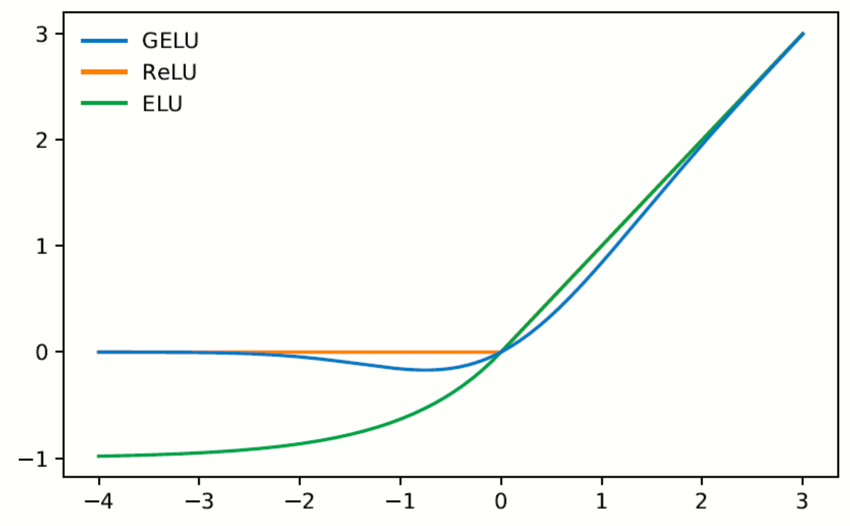


$$h^{\ell} = max(\alpha h^{\ell-1}, h^{\ell-1})$$

Leaky ReLU

- **Activation functions is still area of research!**
  - Though many don't catch on

- **In Transformer architectures, other activations such as GeLU is common**



From "Gaussian Error Linear Units (GELUs)", Hendrycks & Gimpel

Variations: ELU, GeLU, etc.

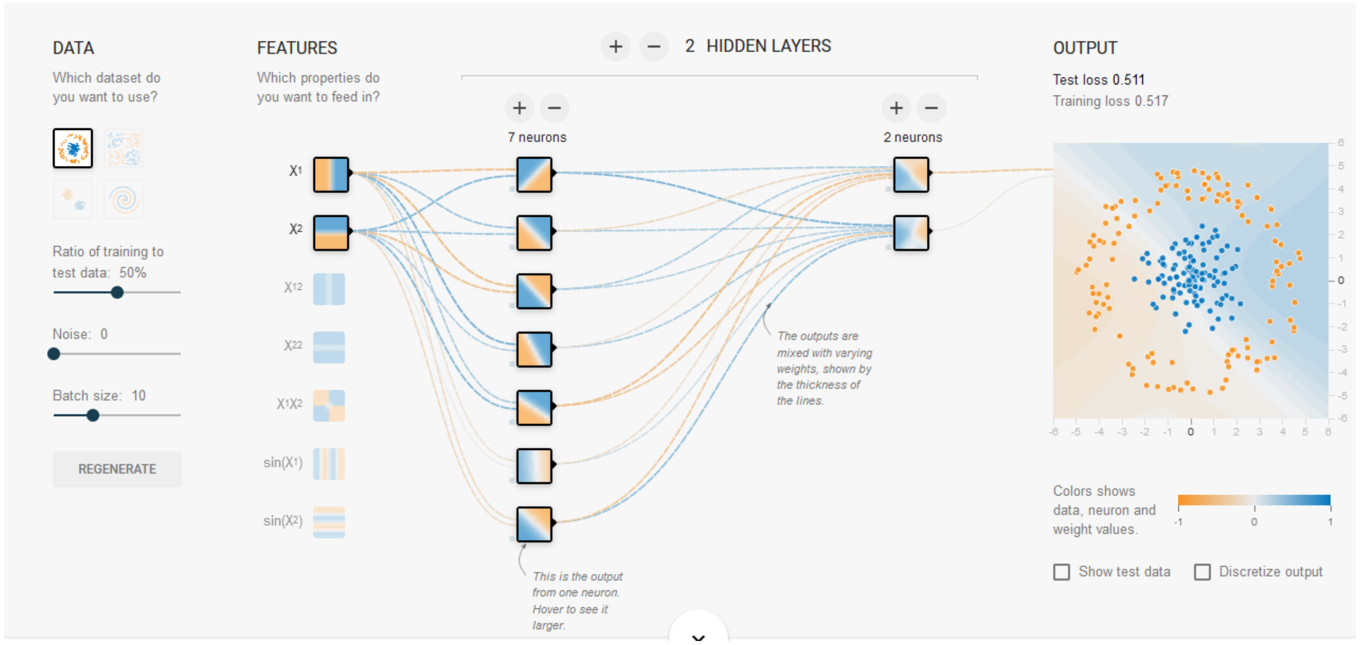
Selecting a Non-Linearity

Which **non-linearity** should you select?

- Unfortunately, **no one activation function is best** for all applications
- **ReLU** is most common starting point
  - Sometimes leaky ReLU can make a big difference
- **Sigmoid** is typically avoided unless clamping to values from [0,1] is needed

Demo

- http://playground.tensorflow.org


DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

7 neurons

2 neurons

The outputs are mixed with varying weights, shown by the thickness of the lines.

This is the output from one neuron. Hover to see it larger.

OUTPUT

Test loss 0.511

Training loss 0.517

Colors shows data, neuron and weight values.

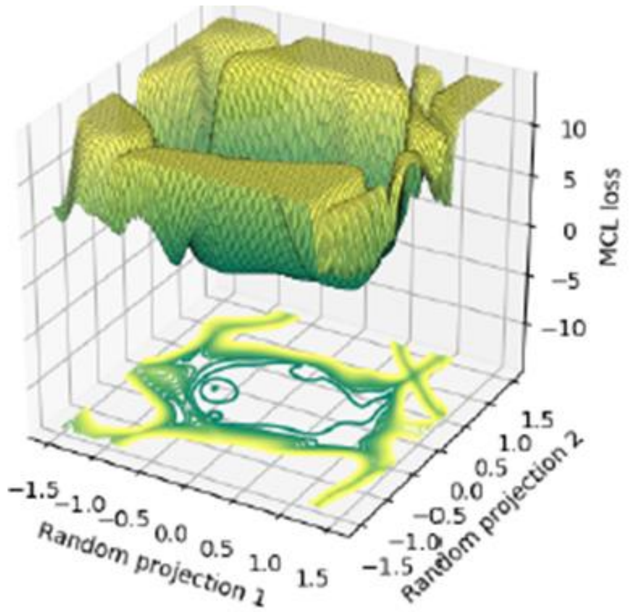
Optimizers

Deep learning involves **complex, compositional, non-linear functions**

The **loss landscape** is extremely **non-convex** as a result

There is **little direct theory** and a **lot of intuition/rules of thumbs** instead

- Some insight can be gained via theory for simpler cases (e.g. convex settings)

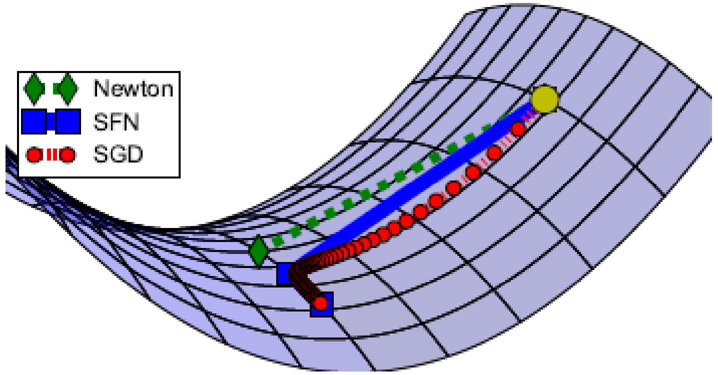


Loss Landscape

It used to be thought that **existence of local minima is the main issue** in optimization

There are other **more impactful issues**:

- Noisy gradient estimates
- Saddle points
- Ill-conditioned loss surface



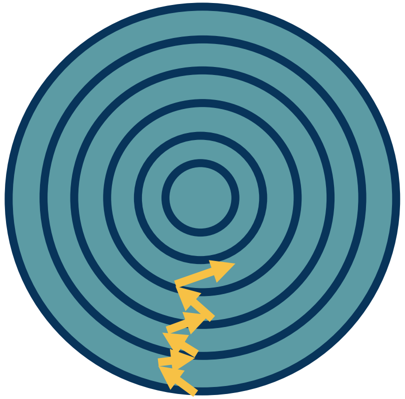
Saddle Point

From: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, Dauphi et al., 2014.

Loss Landscape

- We use a **subset of the data at each iteration** to calculate the loss (& gradients)
- This is an **unbiased** estimator but can have high variance
- This results in **noisy steps** in gradient descent

$$L = \frac{1}{M} \sum L\left(f(x_i, W), y_i\right)$$



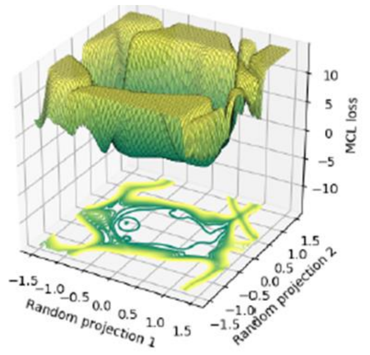
Noisy Gradients

Several **loss surface geometries** are difficult for optimization

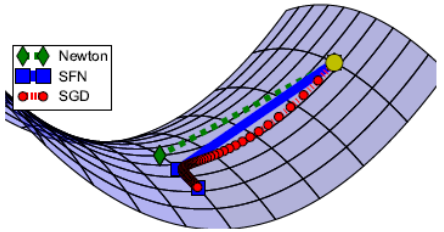
Several types of minima: Local minima, plateaus, saddle points

Saddle points are those where the gradient of orthogonal directions are zero

- But they **disagree** (it's min for one, max for another)



Plateau



Saddle Point

Loss Surface Geometry

Adding Momentum

- Gradient descent takes a step in the **steepest direction** (negative gradient)

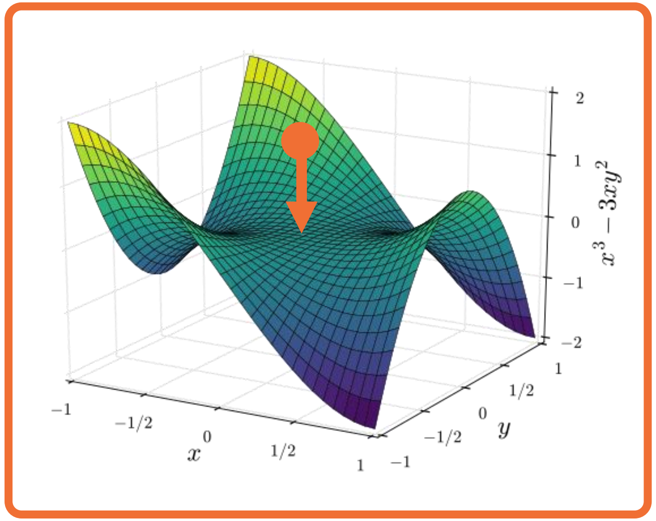
$$w_i = w_{i-1} - \alpha \frac{\partial L}{\partial w_i}$$

- **Intuitive idea:** Imagine a ball rolling down loss surface, and use **momentum** to pass flat surfaces

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$ **Update Velocity (starts as 0, $\beta = 0.99$)**

$$w_i = w_{i-1} - \alpha v_i$$ **Update Weights**

- Generalizes SGD ($\beta = 0$)



Georgia Tech

- Velocity term is an **exponential moving average** of the gradient

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$

$$v_i = \beta(\beta\, v_{i-2} + \frac{\partial L}{\partial w_{i-2}}) + \frac{\partial L}{\partial w_{i-1}}$$

$$= \beta^2 v_{i-2} + \beta \frac{\partial L}{\partial w_{i-2}} + \frac{\partial L}{\partial w_{i-1}}$$

- There is a **general class of accelerated gradient methods**, with some theoretical analysis (under assumptions)

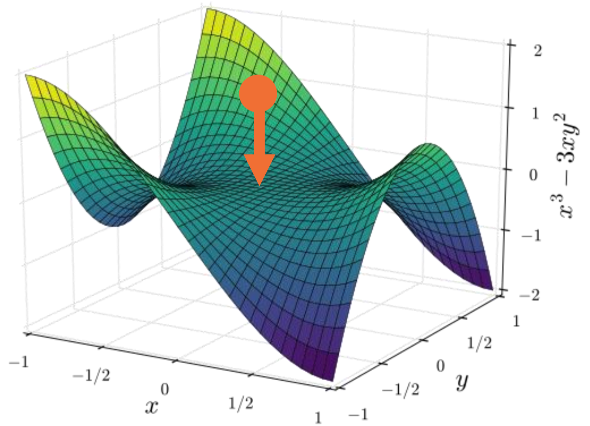
Accelerated Descent Methods

Georgia Tech

Equivalent formulation:

$$v_i = \beta v_{i-1} - \alpha \frac{\partial L}{\partial w_{i-1}}$$ **Update Velocity (starts as 0)**

$$w_i = w_{i-1} + v_i$$ **Update Weights**



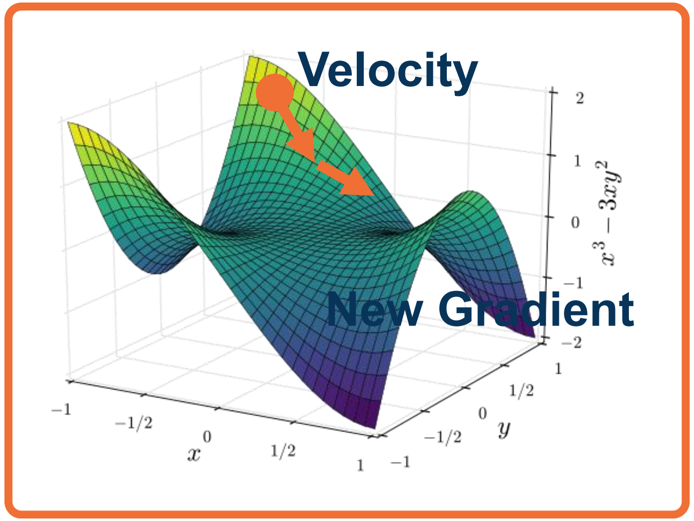
Key idea: Rather than combining velocity with current gradient, go along velocity **first** and then calculate gradient at new point

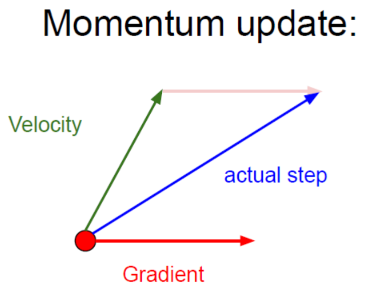
- We know velocity is probably a **reasonable direction**

$$\hat{w}_{i-1} = w_{i-1} + \beta v_{i-1}$$

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial \hat{w}_{i-1}}$$

$$w_i = w_{i-1} - \alpha\, v_i$$





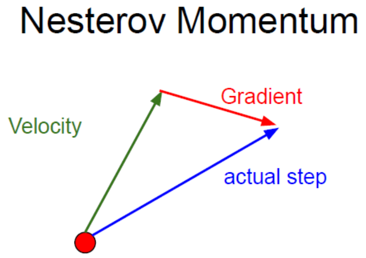


Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Nesterov Momentum

Momentum

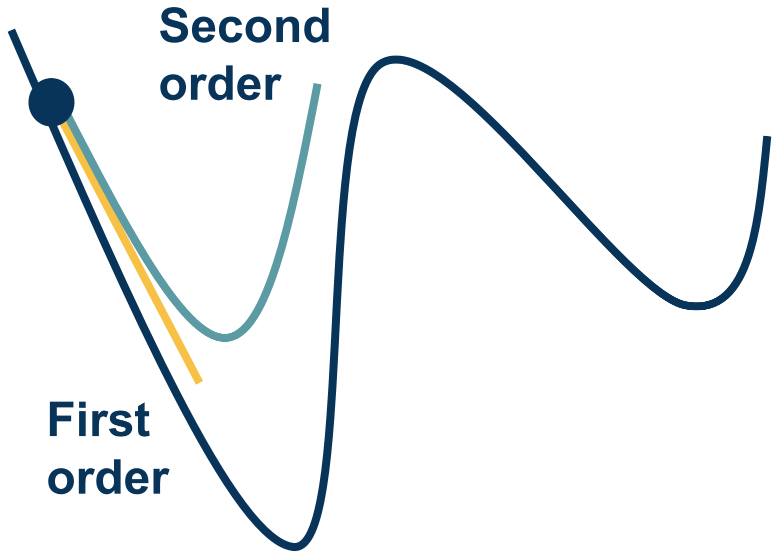
Note there are **several equivalent formulations** across deep learning frameworks!

Resource: https://medium.com/the-artificial-impostor/sgd-implementation-in-pytorch-4115bcb9f02c

- Various mathematical ways to **characterize the loss landscape**
- If you liked **Jacobians**… meet:

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\[2.2ex] \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\[2.2ex] \vdots & \vdots & \ddots & \vdots \\[2.2ex] \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Gives us information about the **curvature of the loss surface**



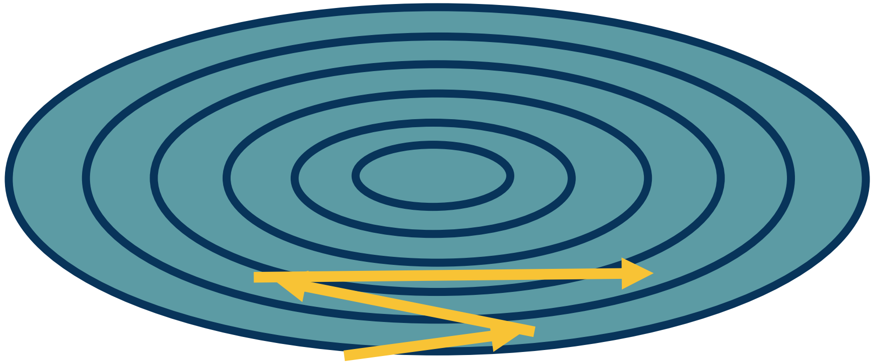
Hessian and Loss Curvature

Condition number is the ratio of the largest and smallest eigenvalue

- Tells us how different the curvature is along different dimensions

If this is high, SGD will make **big** steps in some dimensions and **small** steps in other dimension

Second-order optimization methods divide steps by curvature, but expensive to compute



Condition Number

Per-Parameter Learning Rate

Idea: Have a dynamic learning rate for each weight

Several flavors of **optimization algorithms**:

- RMSProp
- Adagrad
- Adam
- …

SGD can achieve similar results in many cases but with much more tuning

Idea: Use gradient statistics to reduce learning rate across iterations

Denominator: Sum up gradients over iterations

Directions with **high curvature will have higher gradients**, and learning rate will reduce

$$G_i = G_{i-1} + \left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{G_i} + \epsilon} \frac{\partial L}{\partial w_{i-1}}$$

As gradients are accumulated learning rate will go to zero

Duchi, et al., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"

Adagrad

Solution: Keep a moving average of squared gradients!

Does not saturate the learning rate

$$G_i = \beta G_{i-1} + (1-\beta)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{G_i + \epsilon}} \frac{\partial L}{\partial w_{i-1}}$$

RMSProp

Combines ideas from above algorithms

Maintains both first and second moment statistics for gradients

$$v_i = \beta_1 v_{i-1} + (1-\beta_1)\left(\frac{\partial L}{\partial w_{i-1}}\right)$$

$$G_i = \beta_2 G_{i-1} + (1-\beta_2)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$w_i = w_{i-1} - \frac{\alpha\, v_i}{\sqrt{G_i + \epsilon}}$$

But unstable in the beginning (one or both of moments will be tiny values)

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam

Solution: Time-varying bias correction

Typically $\beta_1 = 0.9,\ \beta_2 = 0.999$

So $\hat{v}_i$ will be small number divided by (1-0.9=0.1) resulting in more reasonable values (and $\widehat{G}_i$ larger)

$$v_i = \beta_1 v_{i-1} + (1-\beta_1)\left(\frac{\partial L}{\partial w_{i-1}}\right)$$

$$G_i = \beta_2 G_{i-1} + (1-\beta_2)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$\hat{v}_i = \frac{v_i}{1-\beta_1^t} \qquad \widehat{G}_i = \frac{G_i}{1-\beta_2^t}$$

$$w_i = w_{i-1} - \frac{\alpha\, \hat{v}_i}{\sqrt{\widehat{G}_i + \epsilon}}$$

Adam

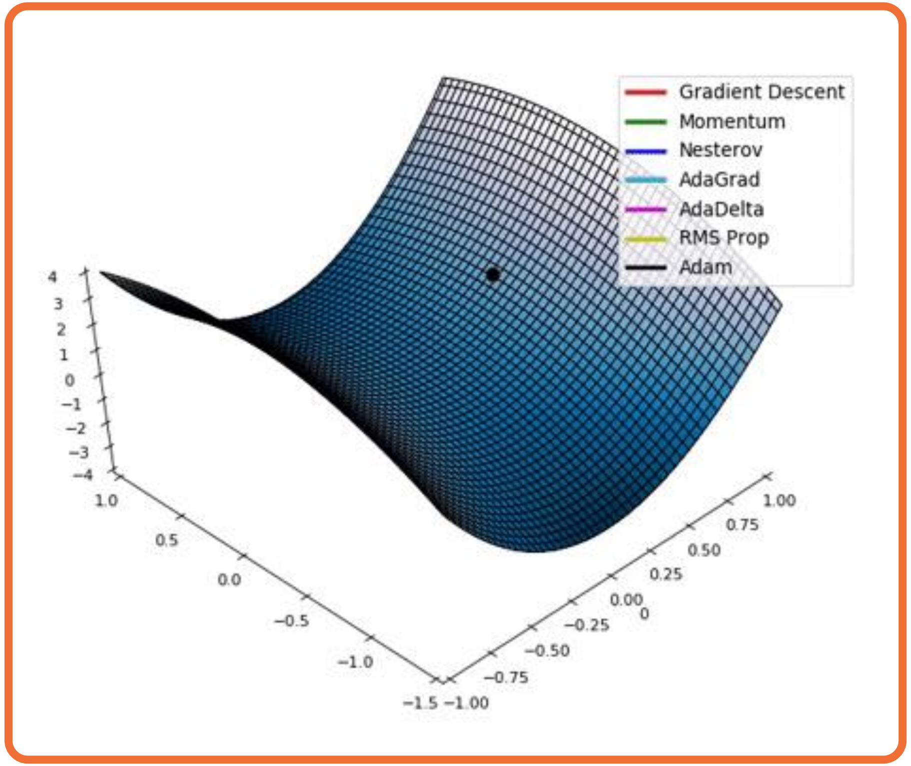
Georgia Tech

Optimizers behave differently **depending on landscape**

Different behaviors such as **overshooting, stagnating, etc.**

Plain SGD+Momentum can generalize better than adaptive methods, but requires more tuning

- **See:** *Luo et al., Adaptive Gradient Methods with Dynamic Bound of Learning Rate, ICLR 2019*



From: https://mlfromscratch.com/optimizers-explained/#/

Behavior of Optimizers

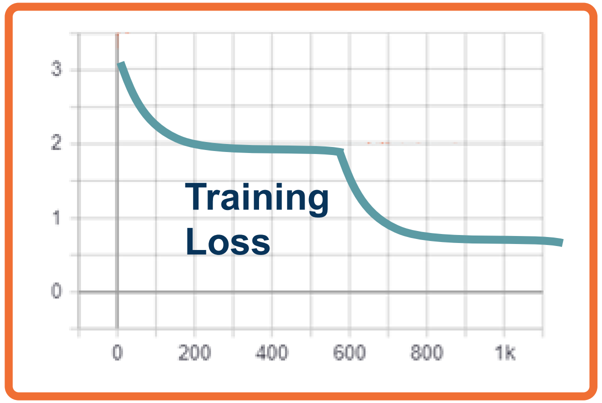
Georgia Tech

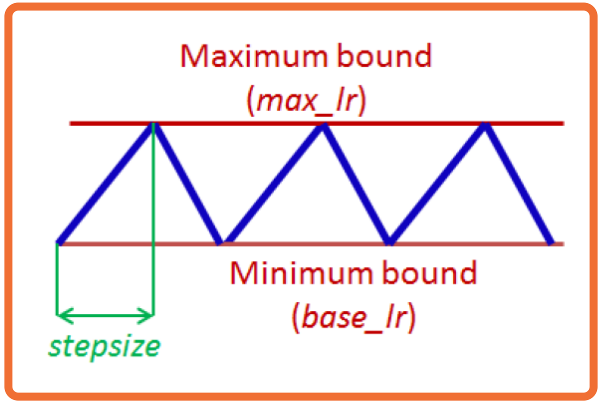
First order optimization methods have **learning rates**

Theoretical results rely on **annealed learning rate**

Several schedules that are typical:

- Graduate student!
- Step scheduler
- Exponential scheduler
- Cosine scheduler





From: Leslie Smith, "Cyclical Learning Rates for Training Neural Networks"

Learning Rate Schedules

Proper Methodology

Always start with **proper methodology**!

- **Not uncommon** even in published papers to get this wrong

Separate data into: **Training, validation, test set**

- **Do not look** at test set performance until you have decided on everything (including hyper-parameters)

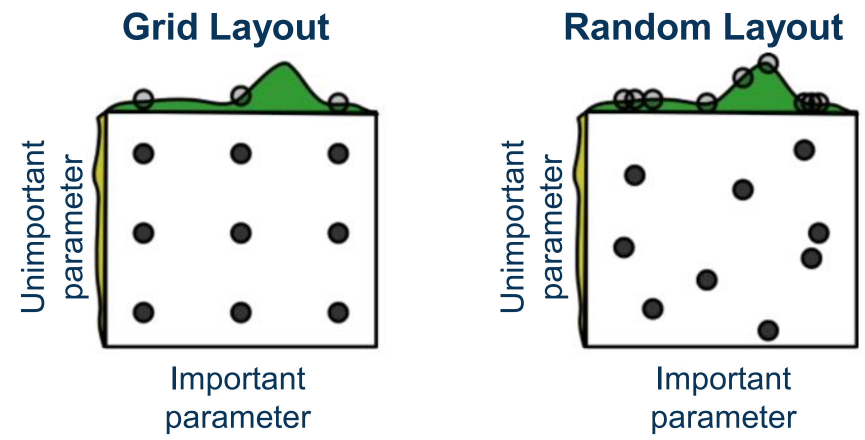
Use **cross-validation** to decide on hyper-parameters if amount of data is an issue

Many hyper-parameters to tune!

- Learning rate, weight decay crucial
- Momentum, others more stable
- **Always tune** hyper-parameters; even a good idea will fail un-tuned!

Start with coarser search:

- E.g. learning rate of {0.1, 0.05, 0.03, 0.01, 0.003, 0.001, 0.0005, 0.0001}
- Perform finer search around good values



From: Bergstra et al., "Random Search for Hyper-Parameter Optimization", JMLR, 2012

Automated methods are OK, but intuition (or random) can do well given enough of a tuning budget

Hyper-Parameter Tuning

Initialization

Initializing the Parameters

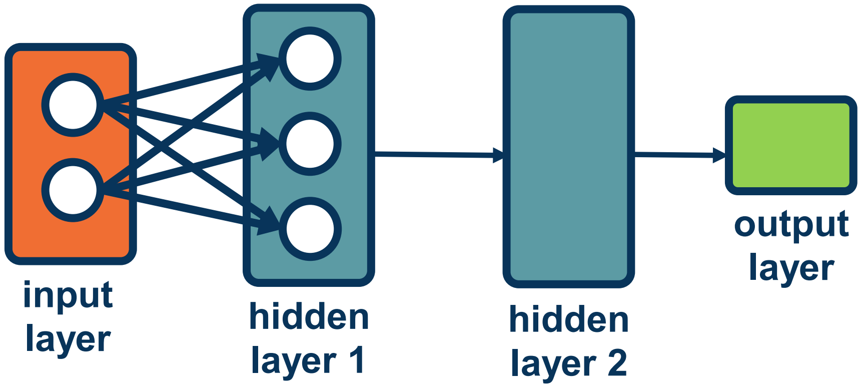
The parameters of our model must be **initialized to something**

- Initialization is **extremely important**!
  - Determines how **statistics of outputs** (given inputs) behave
  - Determines how well **gradients flow** in the beginning of training (important)
  - Could **limit use of full capacity** of the model if done improperly
- Initialization that is **close to a good (local) minima** will converge faster and to a better solution

Initializing values to a constant value leads to **a degenerate solution**!

- What happens to the **weight updates?**
- Each node has the same input from previous layers so gradients **will be the same**
- As a results, **all weights will be updated** to the same exact values

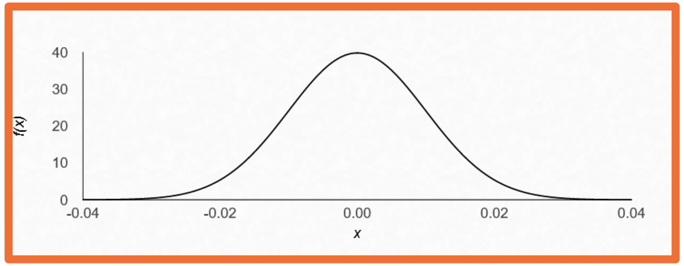
$$w_i = c \quad \forall i$$

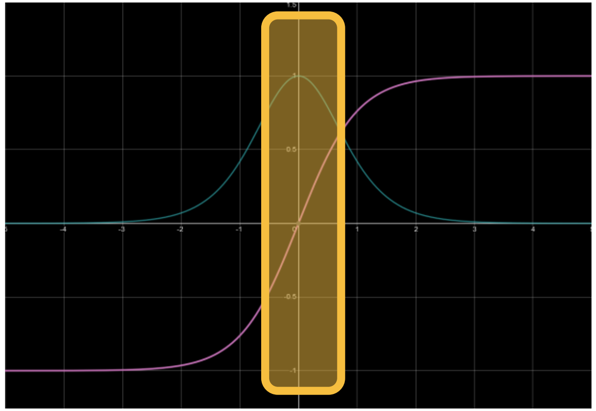


A Poor Initialization

Common approach is **small normally distributed random numbers**

- E.g. $\boldsymbol{N(\mu, \sigma)\ where\ \mu = 0, \sigma = 0.01}$
- **Small weights** are preferred since no feature/input has prior importance
- Keeps the model within the **linear region of most activation functions**

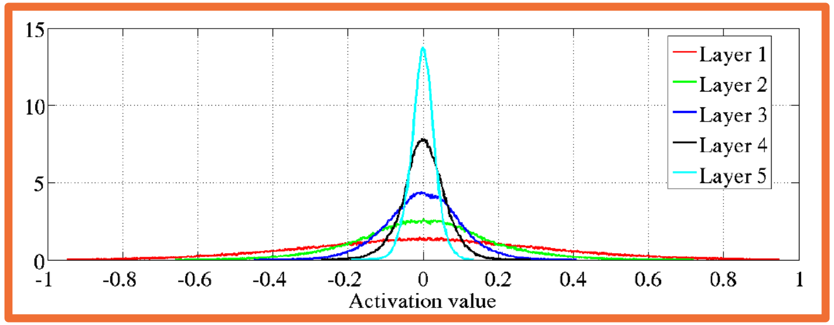




Gaussian/Normal Initialization

Deeper networks (with many layers) are more sensitive to initialization

- With a deep network, **activations (outputs of nodes) get smaller**
  - Standard deviation reduces significantly
- **Leads to small updates –** smaller values multiplied by upstream gradients



Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

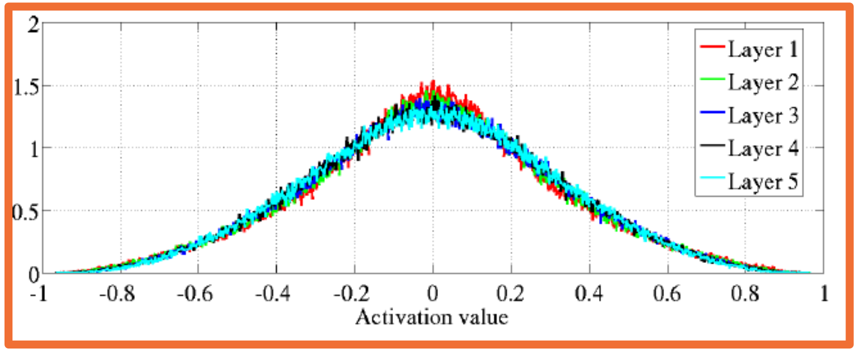
From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, 2010.

Ideally, we'd like to maintain the variance at the output to be similar to that of input!

- This condition leads to a **simple initialization rule**, sampling from uniform distribution:

$$\mathrm{Uniform}\left(-\frac{\sqrt{6}}{n_j+n_{j+1}}, +\frac{\sqrt{6}}{n_j+n_{j+1}}\right)$$

  - Where n_j is **fan-in** (number of input nodes) and n_{j+1} is **fan-out** (number of output nodes)



Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, **2010.**

In practice, **simpler versions** perform empirically well:

$$N(0,1) \; * \sqrt{\frac{1}{n_j}}$$

- This analysis holds for **tanh or similar activations**.
- Similar analysis for **ReLU activations** leads to:

$$N(0,1) \; * \sqrt{\frac{1}{n_j/2}}$$

"Delving Deep into Rectifiers:Surpassing Human-Level Performance on ImageNet Classification", ICCV, 2015.

(Simpler) Xavier and Xavier2 Initialization

Summary

Key takeaway: **Initialization matters!**

- Determines the **activation** (output) statistics, and therefore **gradient statistics**
- If gradients are **small**, no learning will occur and no improvement is possible!
- Important to reason about **output/gradient statistics** and analyze them for new layers and architectures

Summary

- **Activation Functions**: Use ReLU, GeLU, etc.
- **Initialization:** Important for initial activation and gradient statistics
- **Optimization:** Use momentum (helps w/ local minima, etc.)
  - **Next:** More sophisticated gradient history/statistics in update rule