Topics:

- Optimization

CS 4644-DL / 7643-A
ZSOLT KIRA

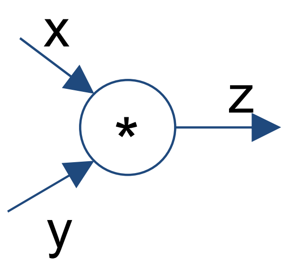
- **Assignment 1 – Due Friday!!!**

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

- **Piazza**: Start with public posts so that others can benefit!
  - Doesn't mean don't post!

- **Meta Lectures**: Data wrangling video available online, OH recordings available:
  - See dropbox link piazza @68 for lectures, @125 for first office hours **Thursday 4pm ET**
  - All OH are on the Canvas Zoom list!

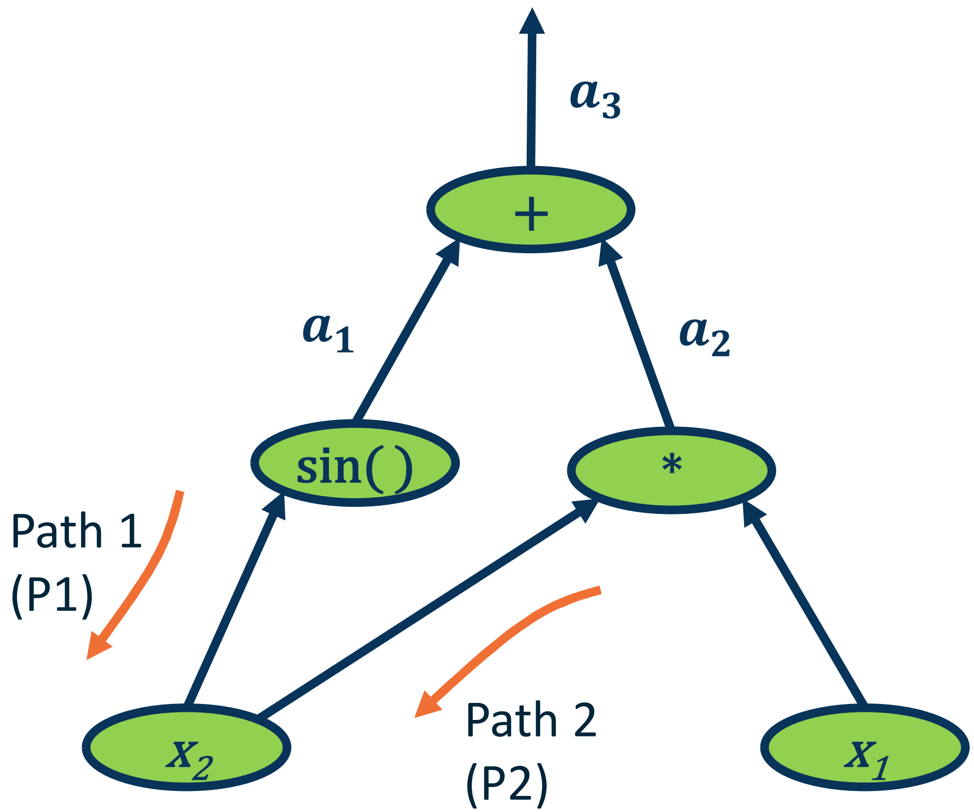
Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

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



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

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

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

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

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

Gradients from multiple paths **summed**

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

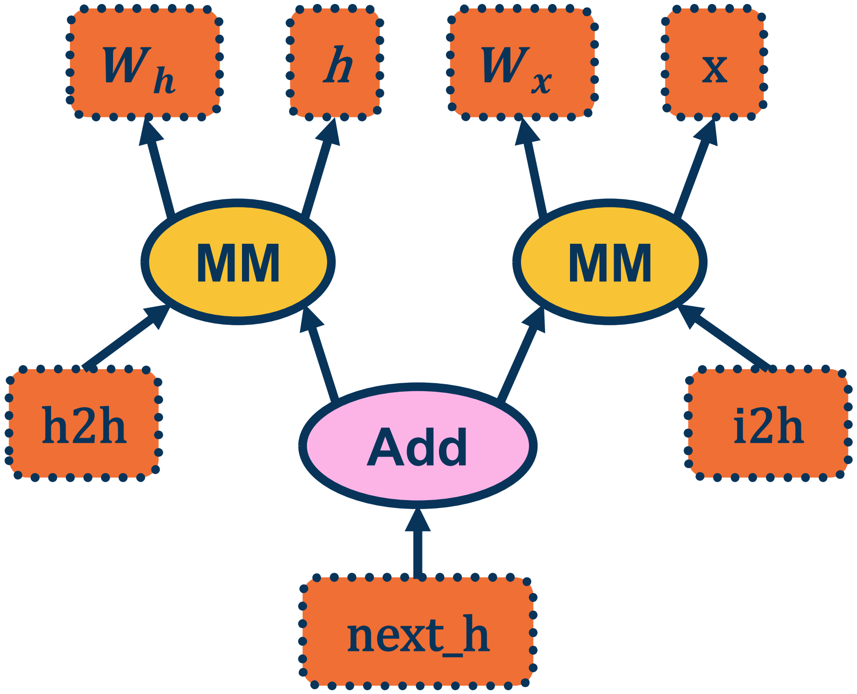
Example

A graph is created on the fly

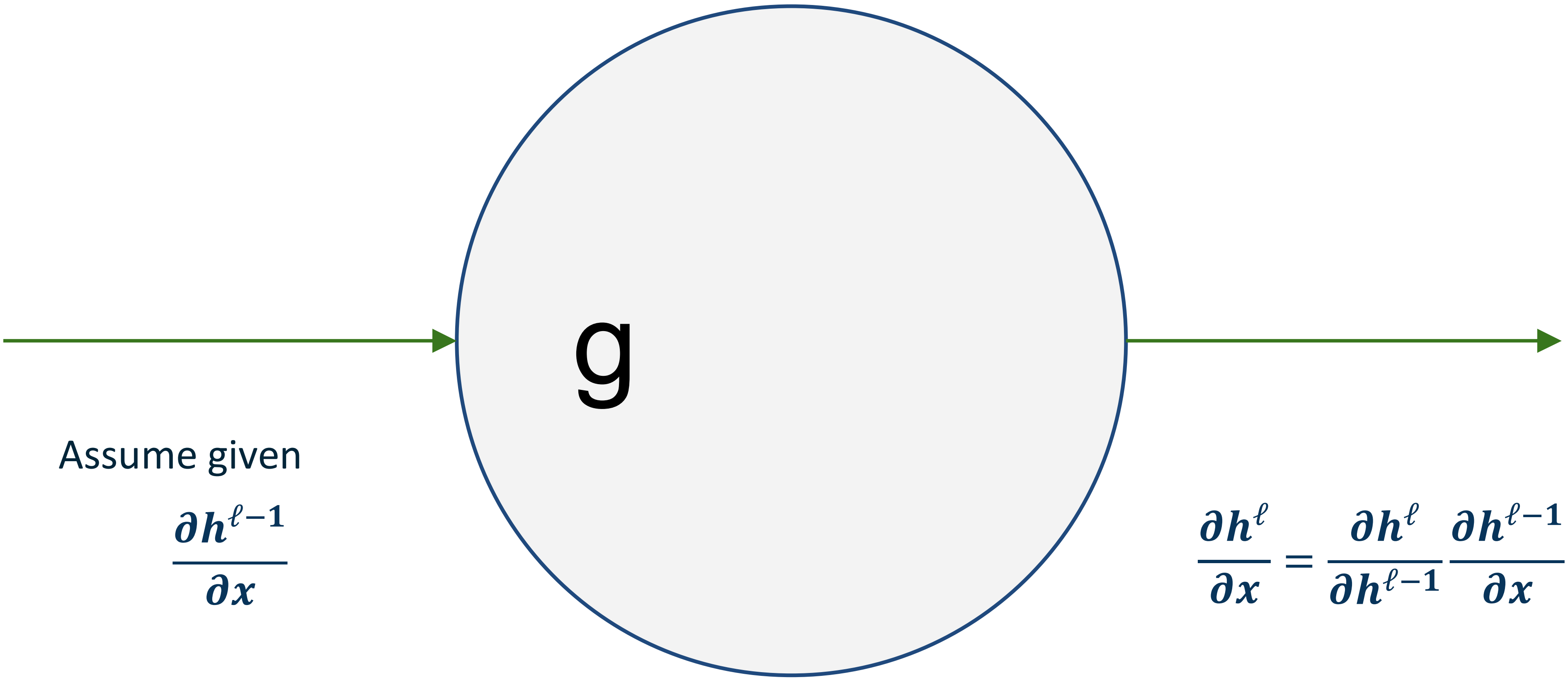
```
from torch.autograd import Variable

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)



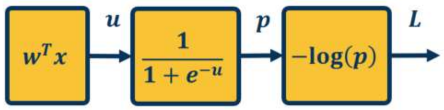
Assume given

$$\frac{\partial h^{\ell-1}}{\partial x}$$

$$\frac{\partial h^{\ell}}{\partial x} = \frac{\partial h^{\ell}}{\partial h^{\ell-1}} \frac{\partial h^{\ell-1}}{\partial x}$$

See https://www.cc.gatech.edu/classes/AY2020/cs7643_spring/slides/autodiff_forward_reverse.pdf

Forward Mode Autodifferentiation



$L = 1$

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

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

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

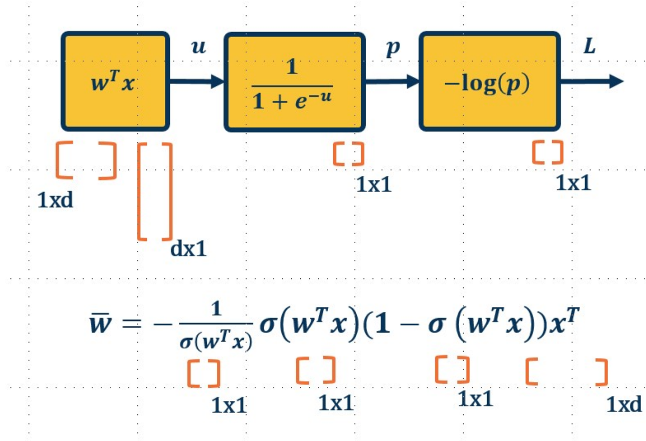
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

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

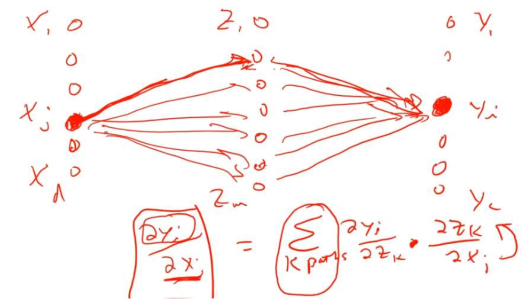
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

This effectively shows gradient flow along path from L to w

Computation Graph / Global View of Chain Rule

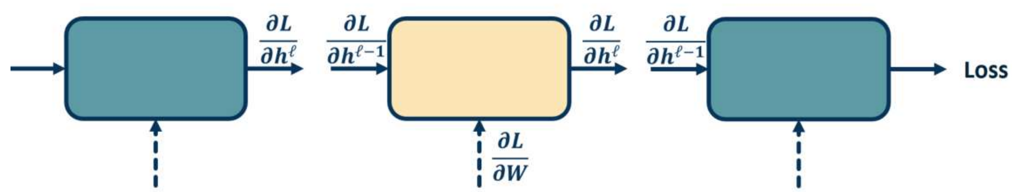


Computational / Tensor View



Graph View

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



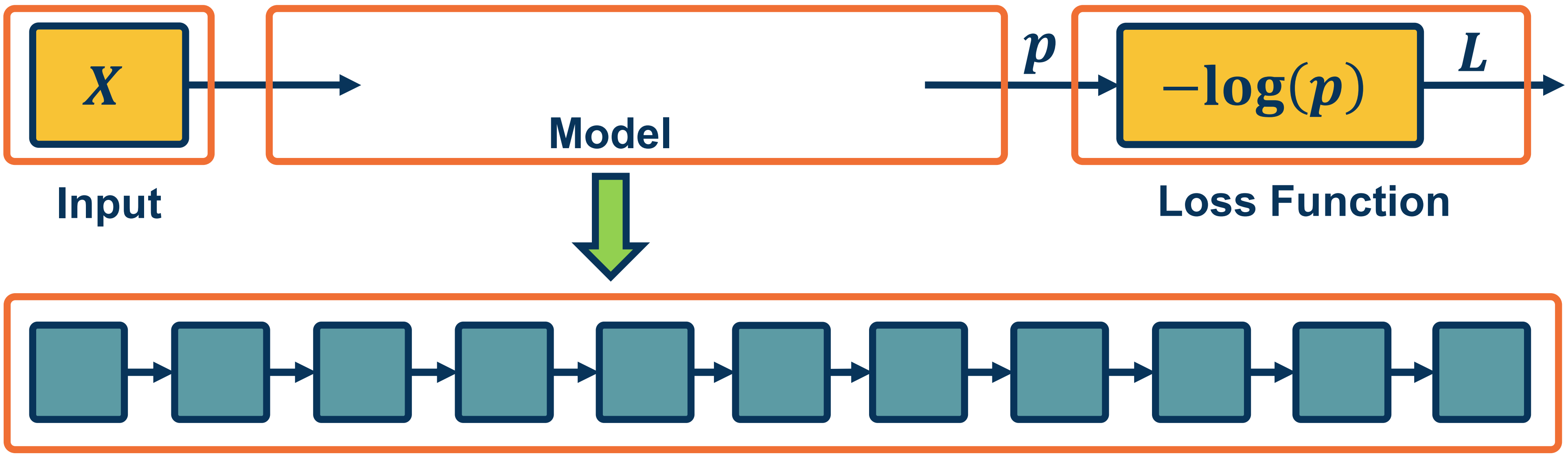
Backpropagation View (Recursive Algorithm)

Different Views of Equivalent Ideas

Georgia Tech

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

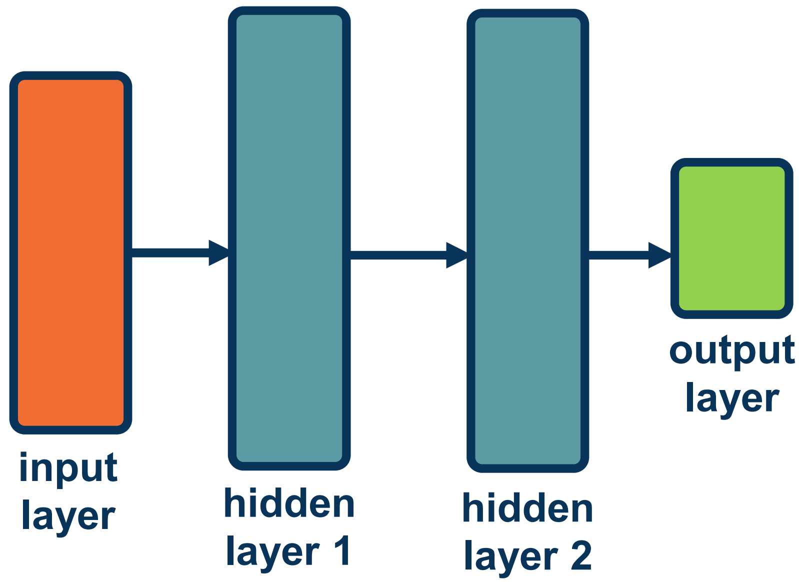
- **No need to modify** the learning algorithm!
- The complexity of the function is only limited by **computation and memory**



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

Convolutional network (AlexNet)

input image

weights

loss

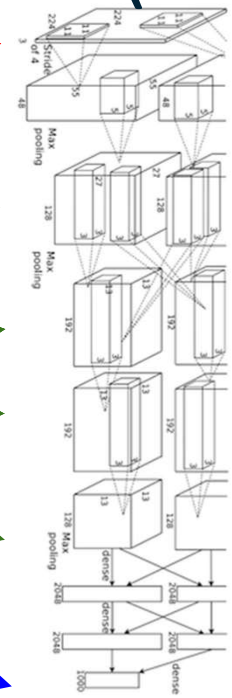


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Neural Turing Machine

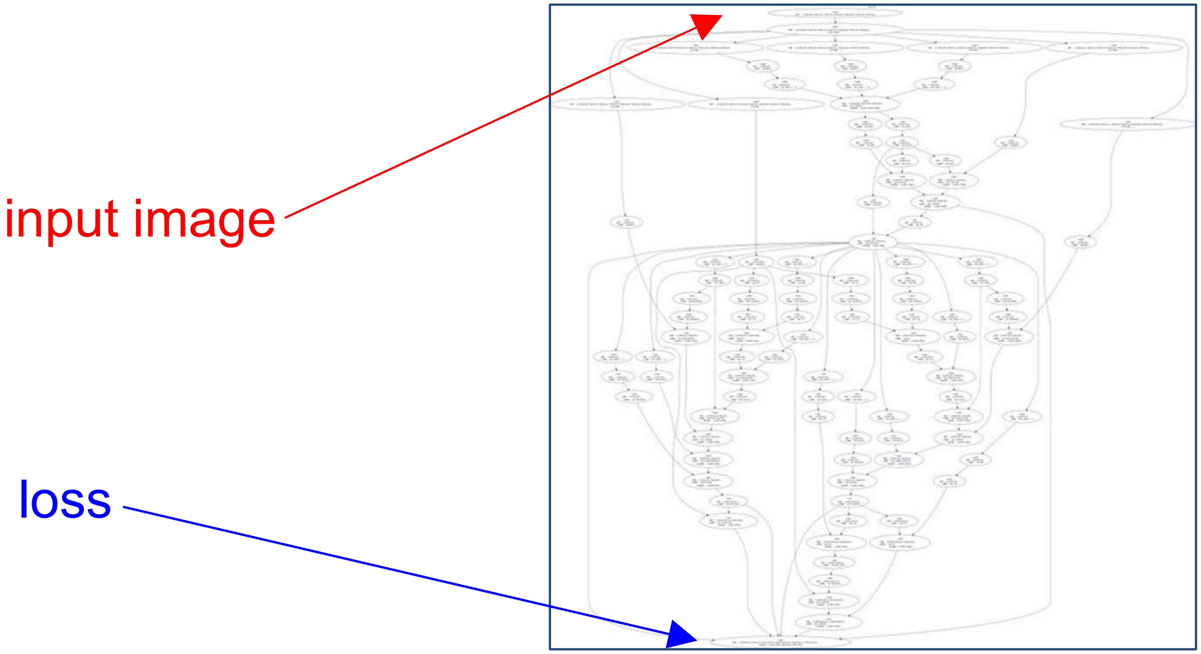
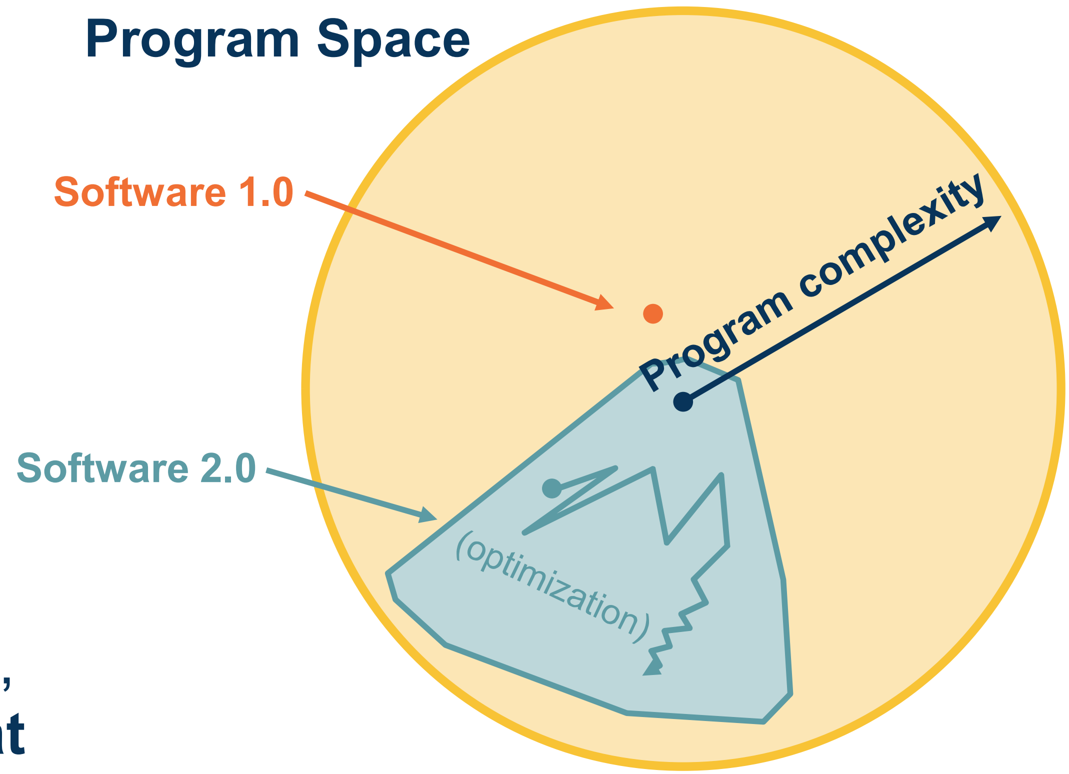


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

- Computation graphs are **not limited to mathematical functions!**
- Can have **control flows** (if statements, loops) and **backpropagate** through **algorithms**!
- Can be done **dynamically** so that **gradients are computed**, then **nodes are added, repeat**
- **Differentiable programming**

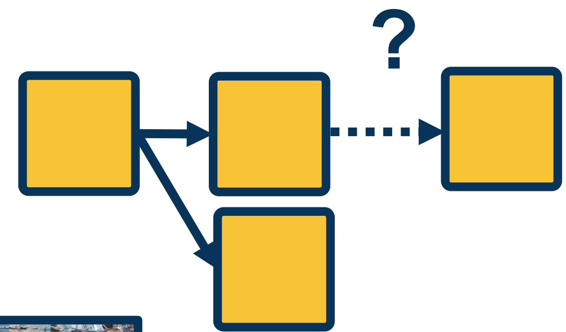


Adapted from figure by Andrej Karpathy

Power of Automatic Differentiation

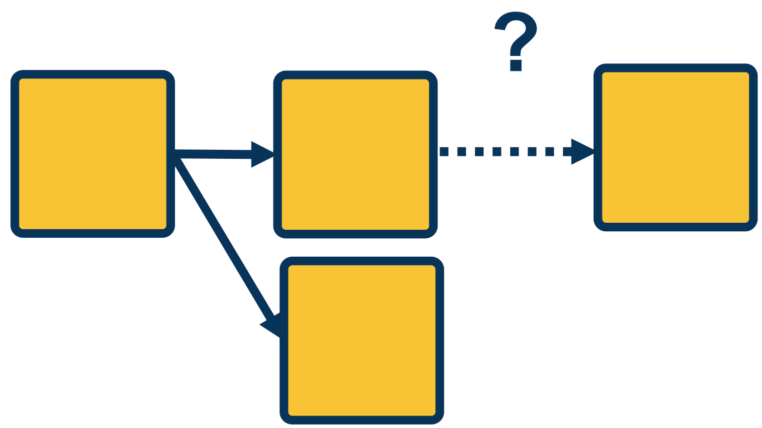
There are still many design decisions that must be made:

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

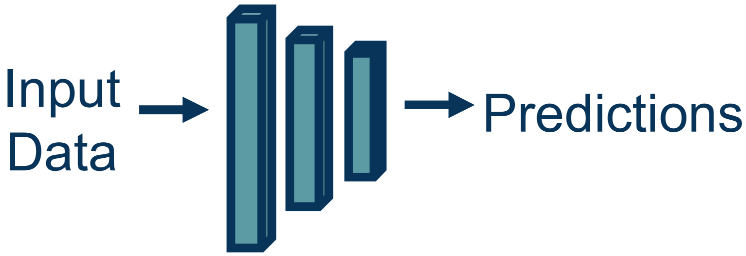


We must design the **neural network architecture:**

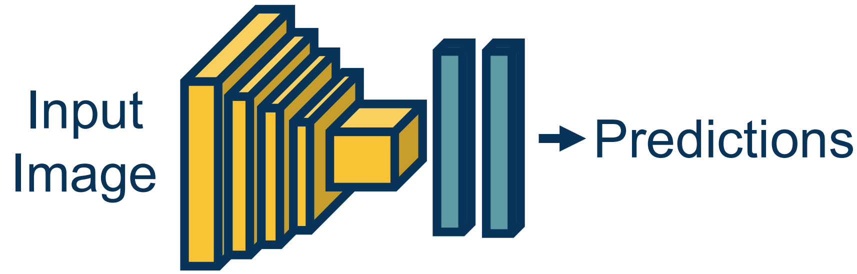
- What **modules (layers)** should we use?
- How should they **be connected together**?
- Can we use our **domain knowledge** to add architectural biases?



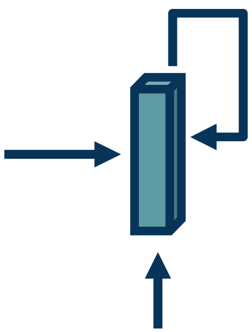
Architectural Considerations



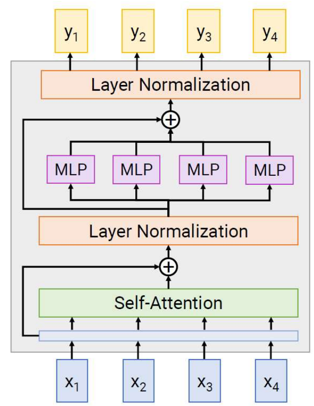
Fully Connected Neural Network



Convolutional Neural Networks



Recurrent Neural Network



Transformers

Different architectures are suitable for different applications or types of input

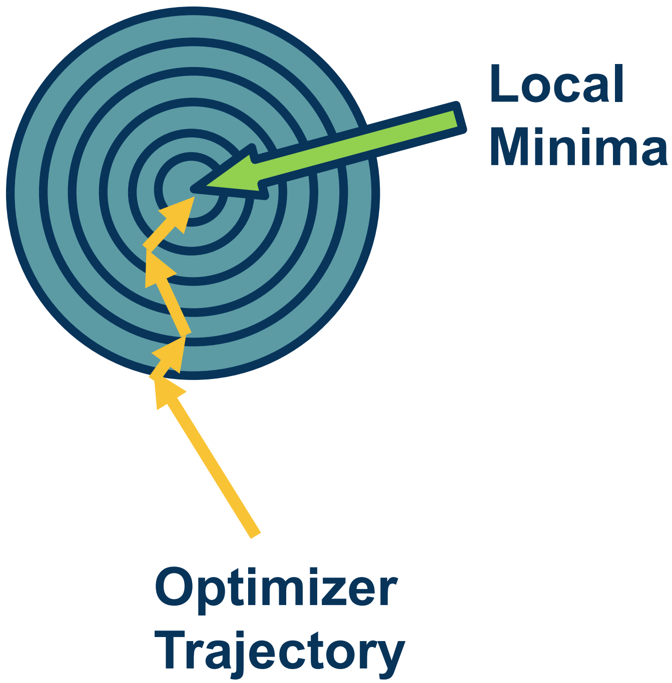
Example Architectures

As in traditional machine learning, **data** is key:

- Should we **pre-process** the data?
- Should we **normalize** it?
- Can we **augment** our data by adding noise or other perturbations?

Even given a good neural network architecture, we need a **good optimization algorithm to find good weights**

- What **optimizer** should we use?
  - Different optimizers make **different weight updates** depending on the gradients
- How should we **initialize** the weights?
- What **regularizers** should we use?
- What **loss function** is appropriate?



Optimization Considerations

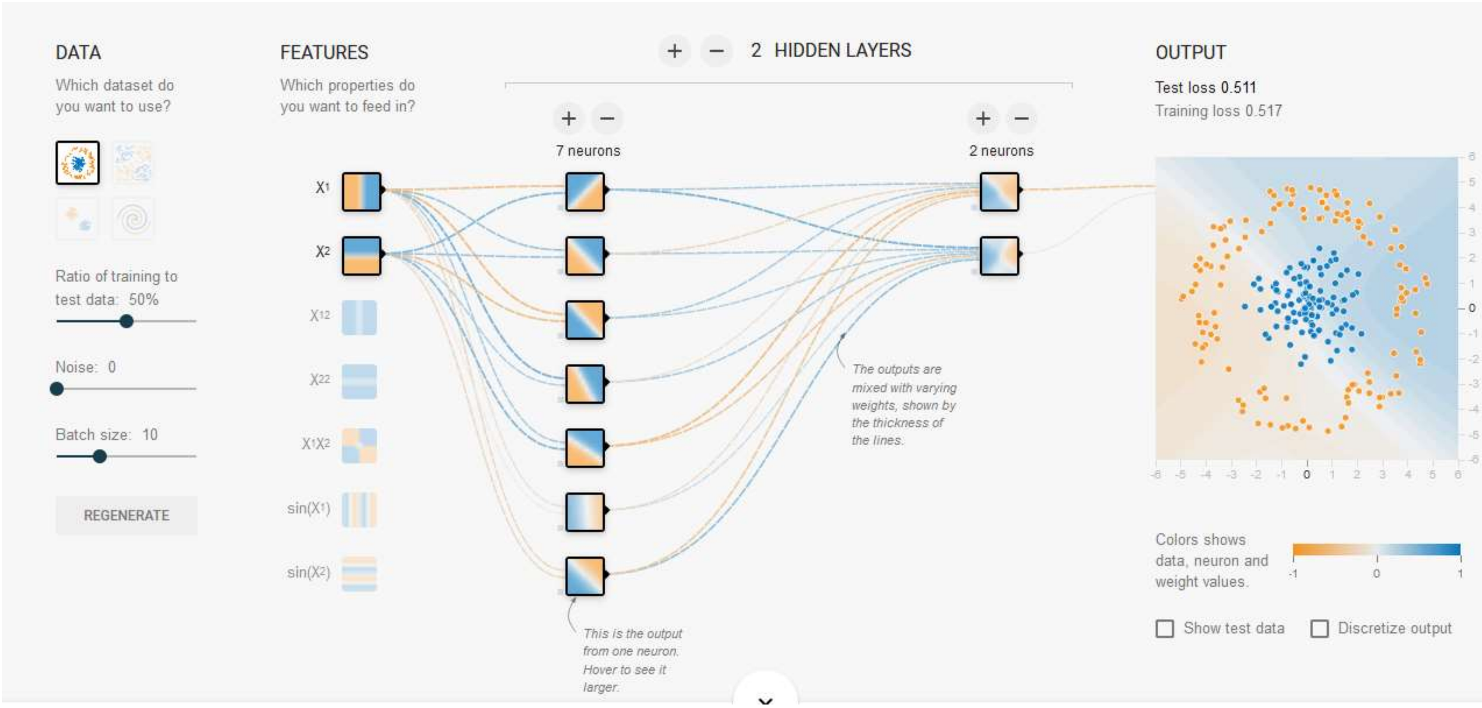
Machine Learning Considerations

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

Demo

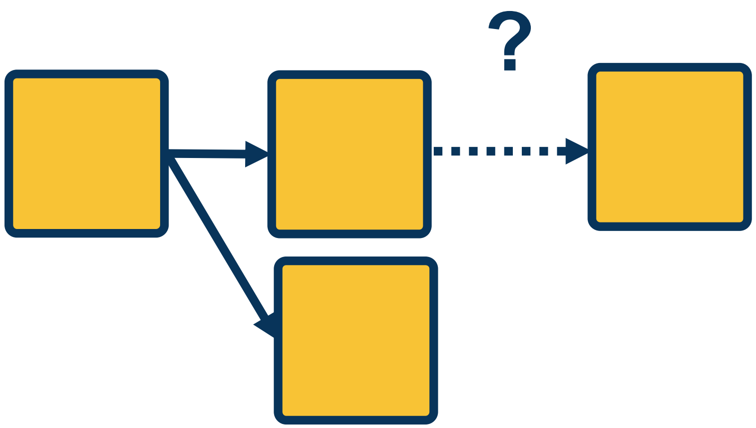
- http://playground.tensorflow.org



Architectural Considerations

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

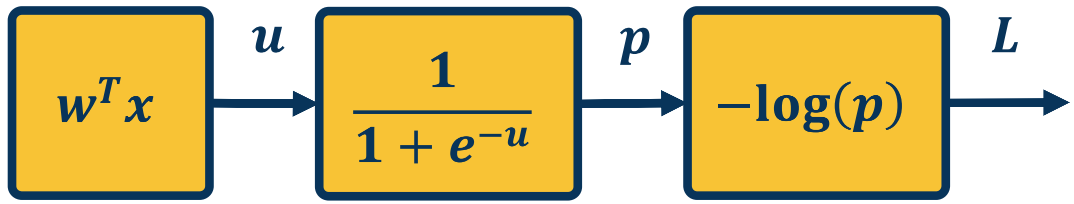
- Guided by the **type of data used** and its **characteristics**
  - Understanding your data is always the first step!
- **Lots of data types (modalities)** already have good architectures
  - Start with what others have discovered!
- **The flow of gradients** is one of the key principles to use when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

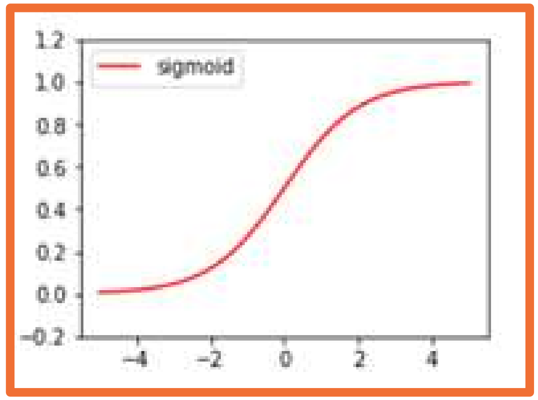


Designing the Architecture

- **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**

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

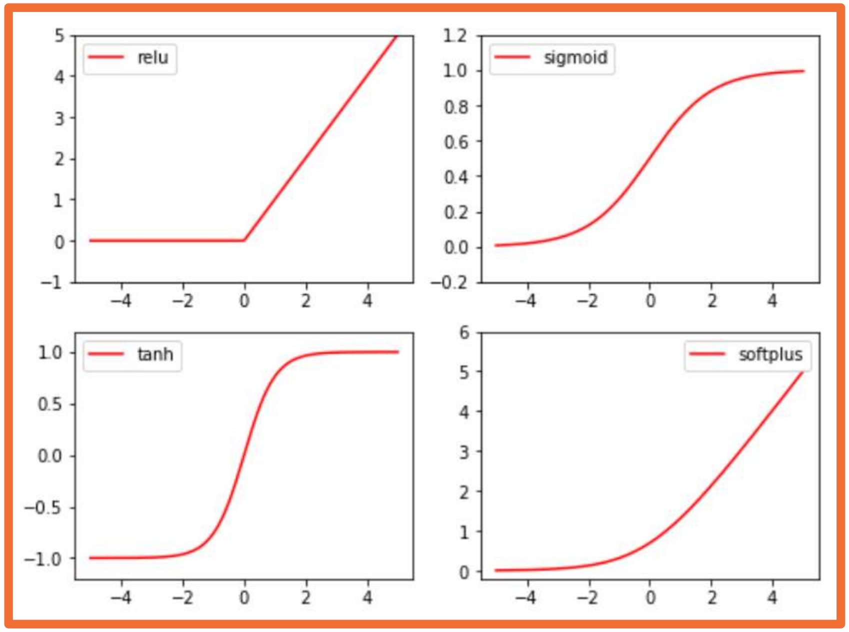




Linear and Non-Linear Modules

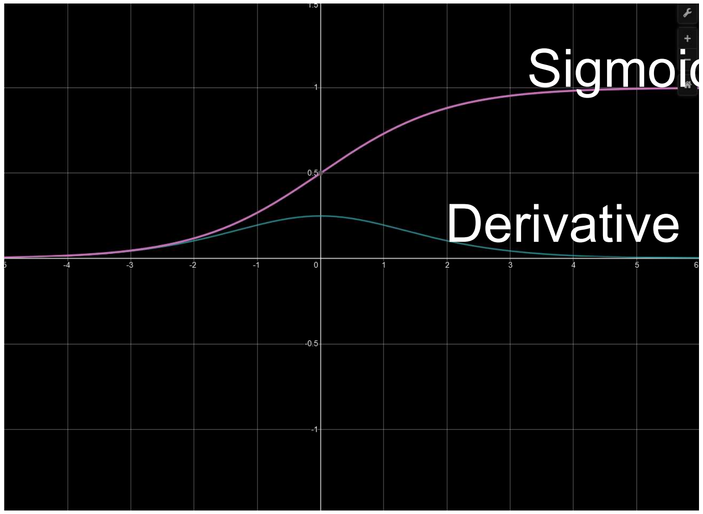
Several aspects that we can **analyze**:

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



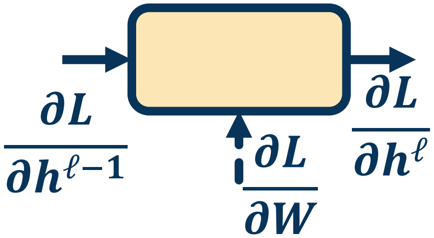
Analysis of Non-Linear Function

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



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

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

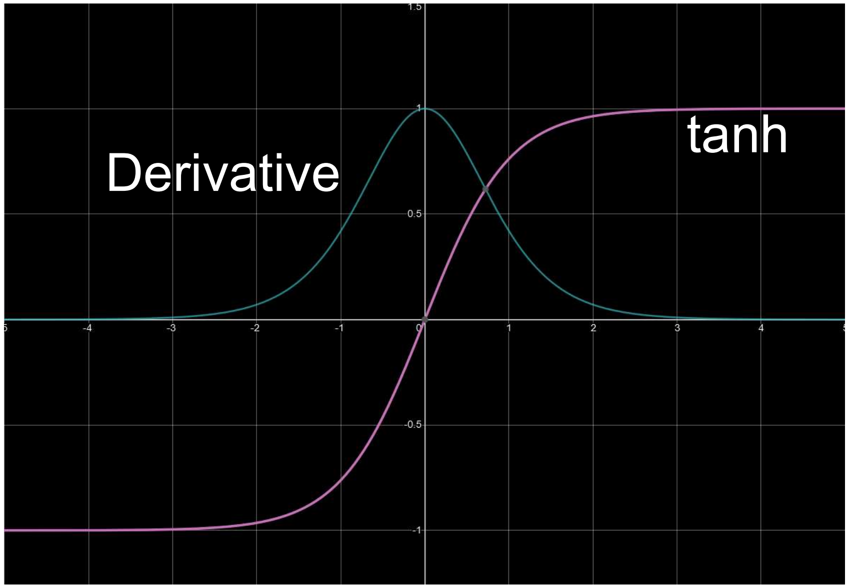


$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$$

Sigmoid Function

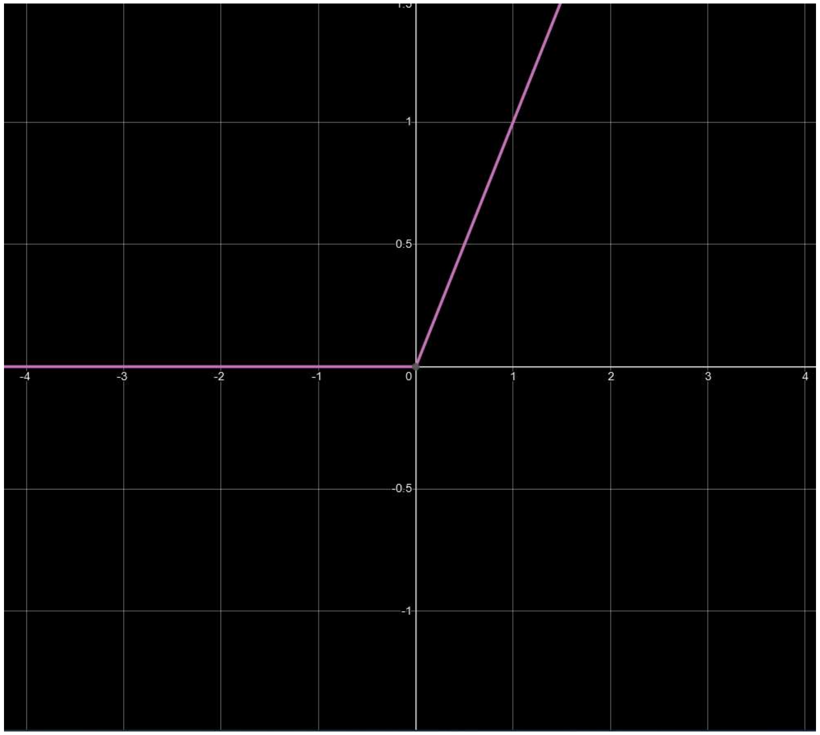
Georgia Tech

- **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**
  - $\mathbf{0}$ if $\mathbf{x} \leq \mathbf{0}$ (dead ReLU)
  - Constant otherwise (does not vanish)
- **Cheap to compute (max)**

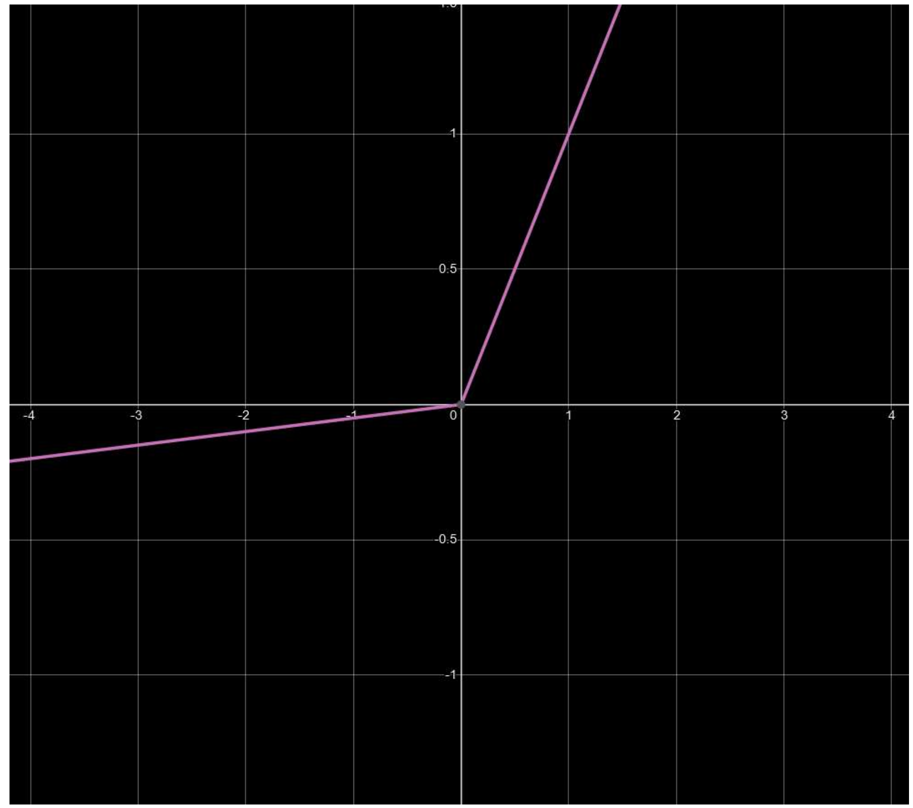
$$\boldsymbol{h^{\ell} = max(0, h^{\ell-1})}$$

Rectified Linear Unit

Georgia Tech

Leaky ReLU

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



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

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

Initialization

Initializing the Parameters

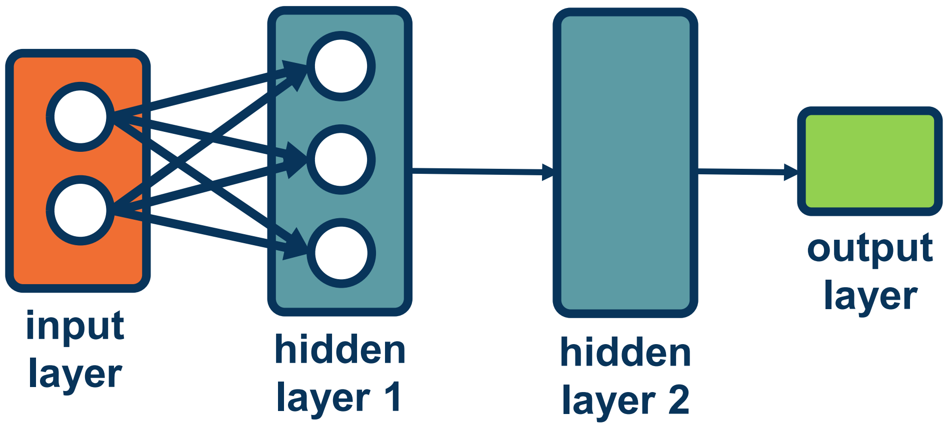
The parameters of our model must be **initialized to something**

- 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

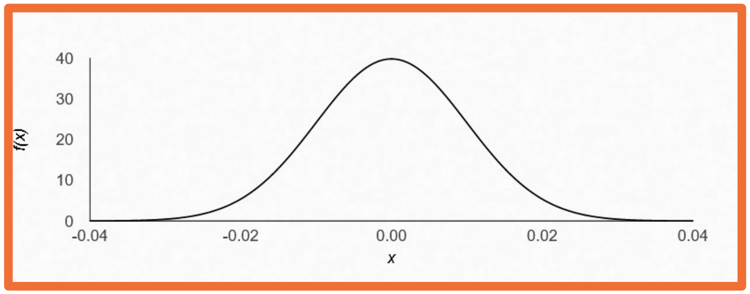
$$w_i = c \quad \forall i$$

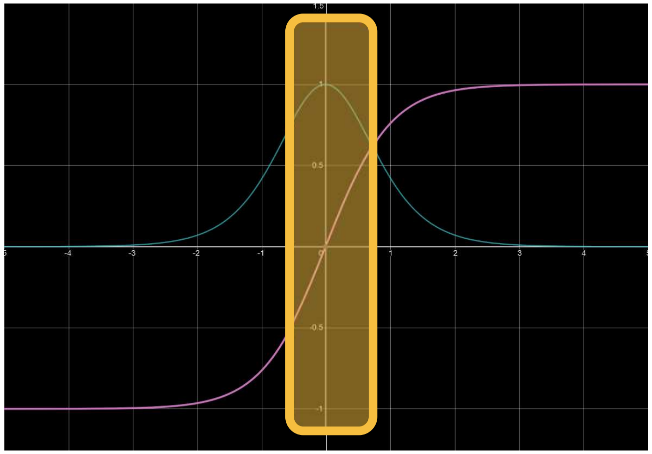


A Poor Initialization

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

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

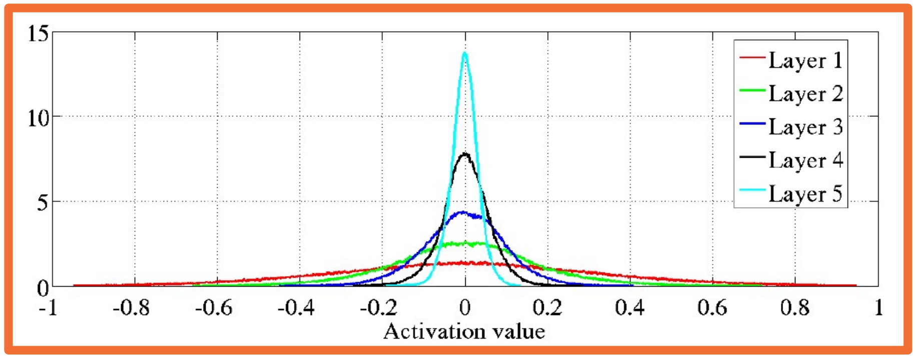




Gaussian/Normal Initialization

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.

Limitation of Small Weights

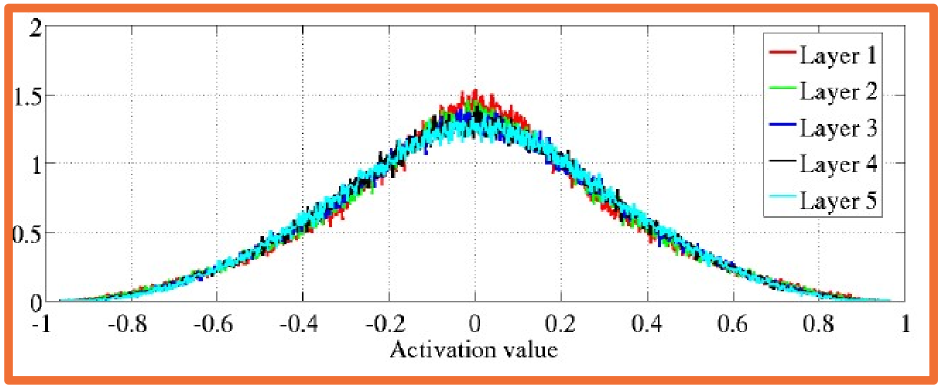
Georgia Tech

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:

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

- 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

Normalization, Preprocessing, and Augmentation

Importance of Data

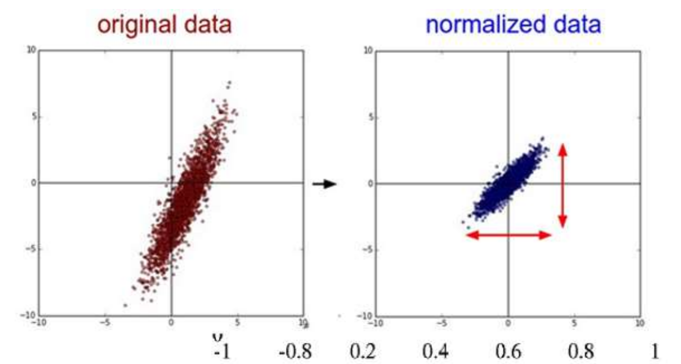
In deep learning, **data drives learning** of features and classifier

- Its **characteristics** are therefore extremely important
- Always **understand your data**!
- **Relationship** between output statistics, layers such as non-linearities, and gradients is important

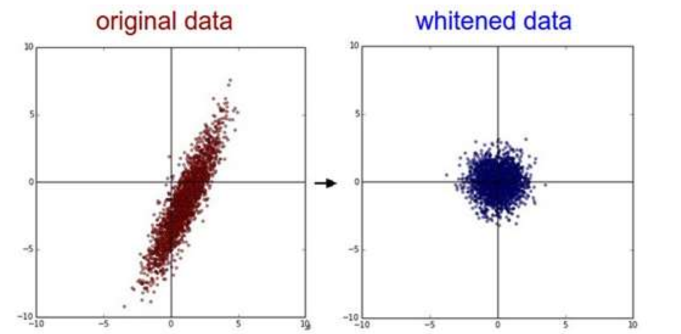
Just like initialization, **normalization** can **improve gradient flow and learning**

Typically **normalization methods** apply:

- Subtract mean, divide by standard deviation **(most common)**
  - This can be done **per** dimension
- Whitening, e.g. through Principle Component Analysis (PCA) **(not common)**



Data after subtracting mean, dividing by standard deviation



Data after whitening

Figure from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

- We can try to come up with a *layer* that can normalize the data across the neural network
- **Given:** A mini-batch of data $[\boldsymbol{B} \times \boldsymbol{D}]$ where $\boldsymbol{B}$ is batch size
- Compute mean and variance **for each dimension** $\boldsymbol{d}$

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

Making Normalization a Layer

Normalize data

$$\widehat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Note: This part does not involve new parameters

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

Normalizing the Data

- We can give the model flexibility through **learnable parameters γ (scale) and β (shift)**
- Network can learn to **not normalize** if necessary!
- This layer is called a **Batch Normalization (BN) layer**

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

Learnable Scaling and Offset

Some Complexities of BN

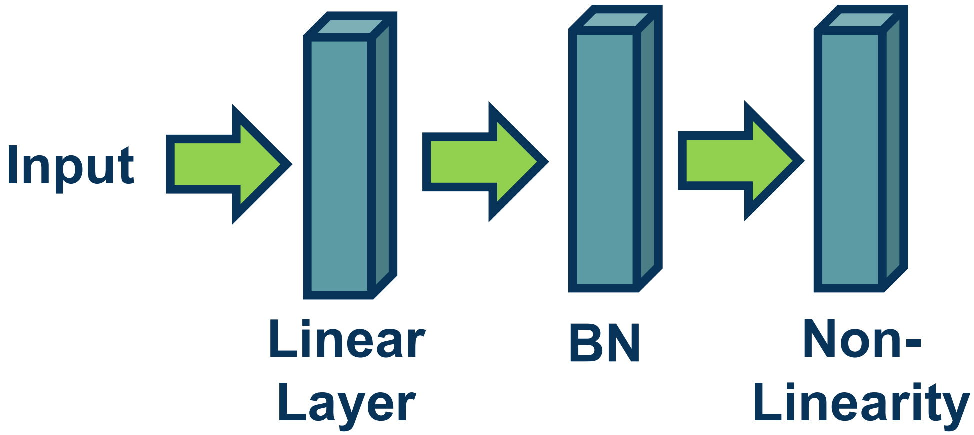
During inference, stored mean/variances calculated on training set are used

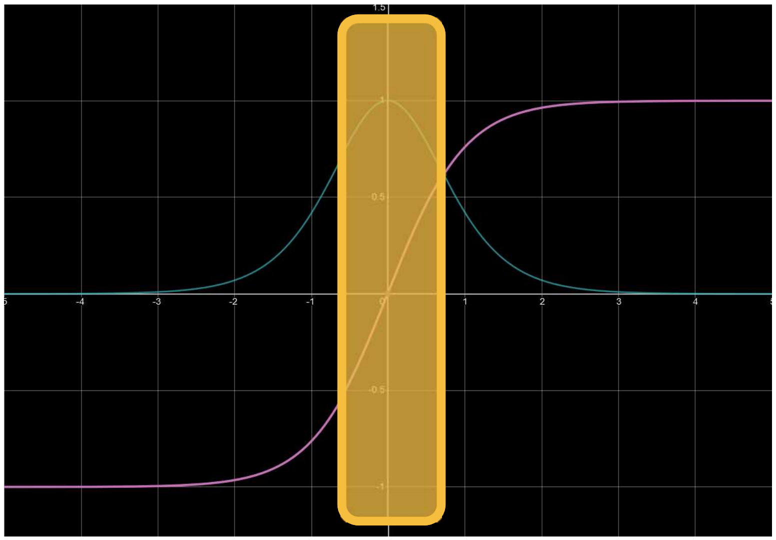
Sufficient batch sizes must be used to get stable per-batch estimates during training

- This is especially an issue when **using multi-GPU or multi-machine training**
- **Use torch.nn.SyncBatchNorm** to estimate batch statistics in these settings

Normalization especially important before **non-linearities**!

- Very low/high values (un-normalized/imbalanced data) cause **saturation**





Where to Apply BN

Generalization of BN

There are **many variations of batch normalization**

- See Convolutional Neural Network lectures for an example

Resource:

- **[ML Explained - Normalization]**

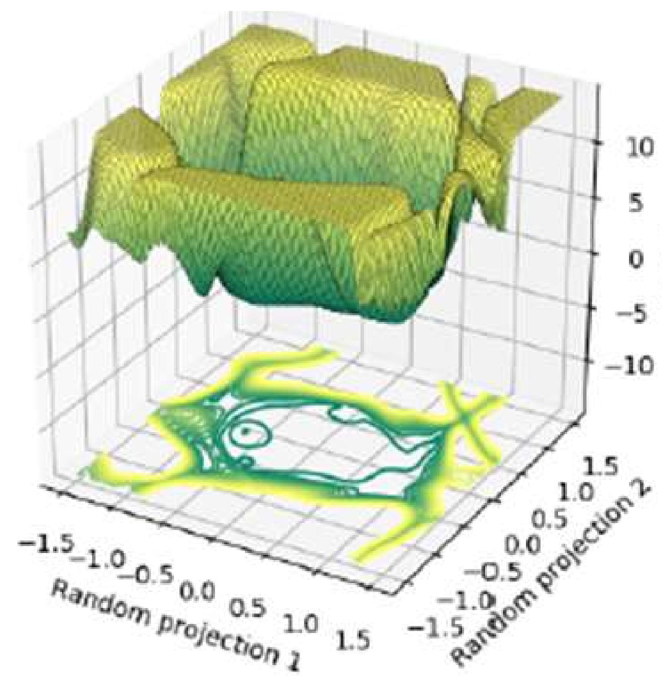
Optimizers

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)

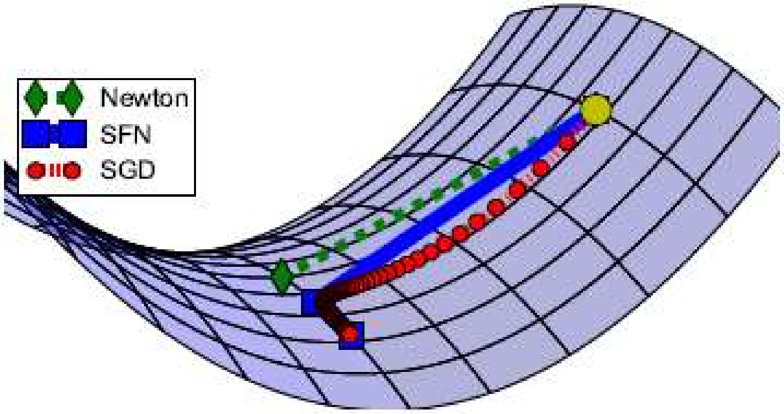


Loss Landscape

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

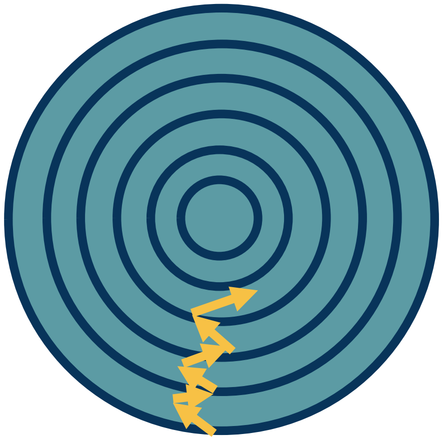


Saddle Point

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

- 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

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



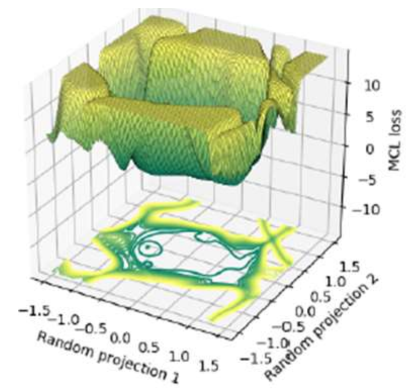
Noisy Gradients

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

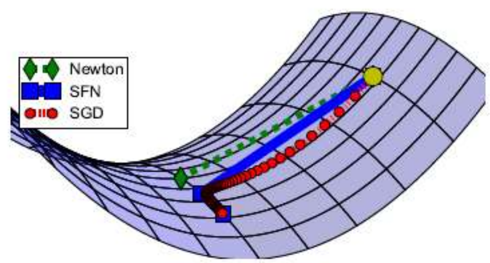
Several types of minima: Local minima, plateaus, saddle points

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

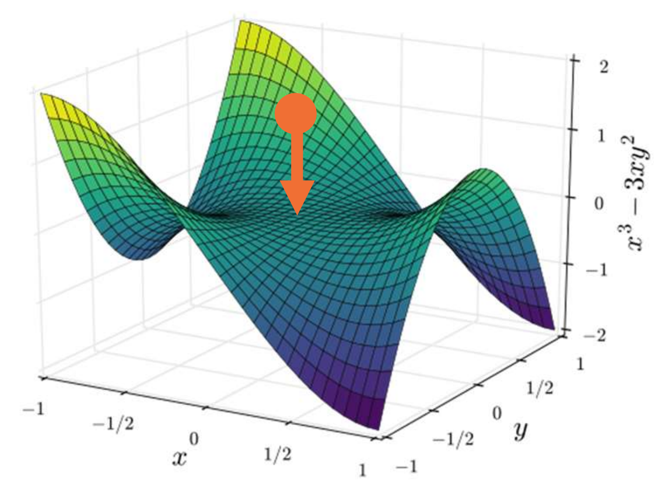
- Gradient descent takes a step in the **steepest direction** (negative gradient)

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

- **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 ($\beta = 0$)

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

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

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

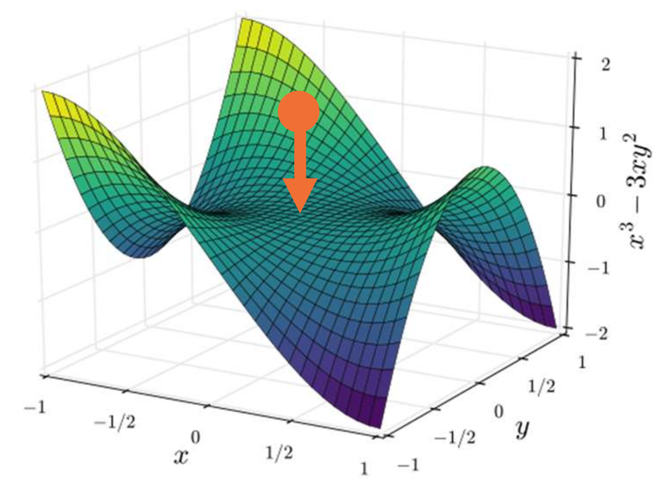
$$= \beta^2 v_{i-2} + \beta \frac{\partial L}{\partial w_{i-2}} + \frac{\partial L}{\partial w_{i-1}}$$

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

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**

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

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

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

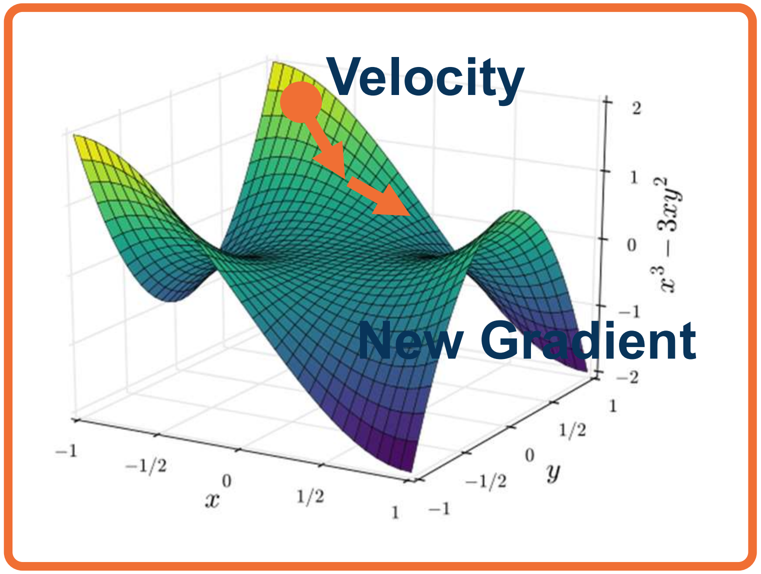


Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Nesterov Momentum

Momentum

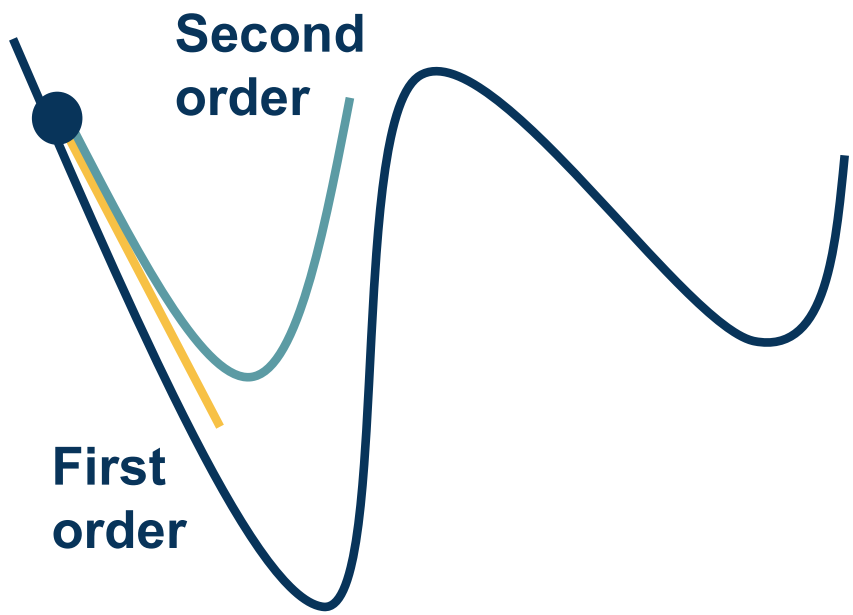
Note there are **several equivalent formulations** across deep learning frameworks!

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**



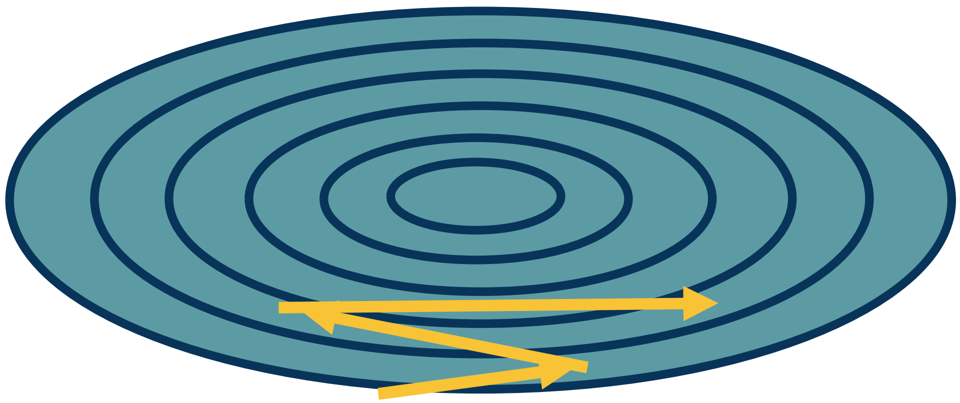
Hessian and Loss Curvature

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



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

Solution: Time-varying bias correction

Typically $\boldsymbol{\beta_1 = 0.9,\ \ \beta_2 = 0.999}$

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

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

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

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

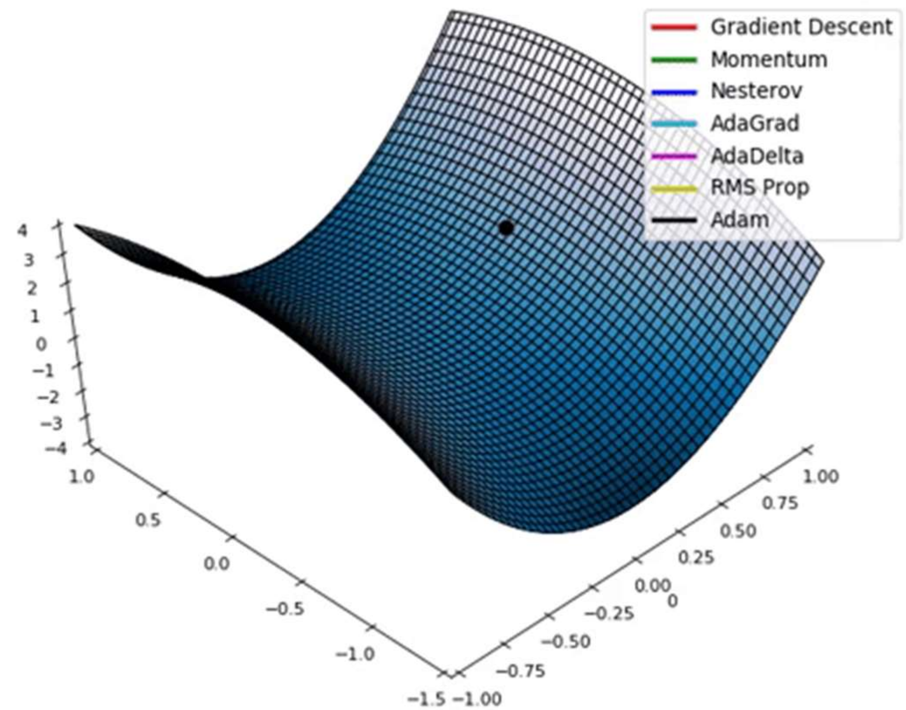
$$w_i = w_{i-1} - \frac{\alpha\, \hat{v}_i}{\sqrt{\hat{G}_i + \epsilon}}$$

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*



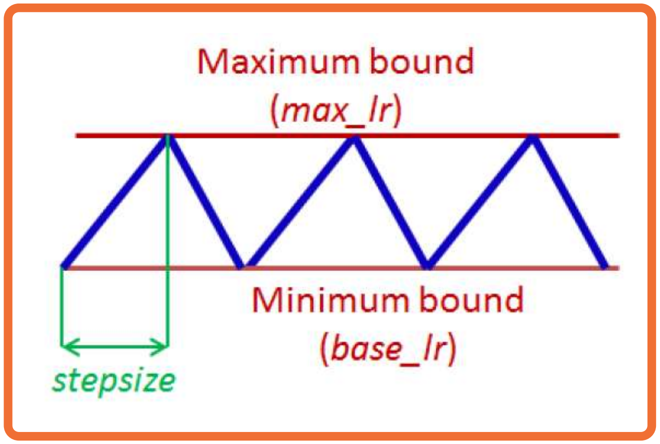
From: https://mlfromscratch.com/optimizers-explained/#/

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