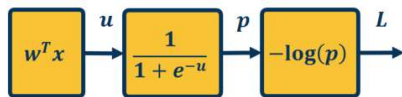


Topics:

- Optimization

CS 4644-DL / 7643-A
ZSOLT KIRA

- **Assignment 1 – grace period ends today!!!**
- **Assignment 2**
 - Implement convolutional neural networks
- **Piazza:** Start with public posts so that others can benefit!
 - Doesn't mean don't post!
- **Facebook Lectures:** Data wrangling video available online
 - See dropbox link piazza @8 and M1L4 folder
 - Opportunity to talk to them Wed. 02/17 4-5pm
- **Project details coming soon**



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

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

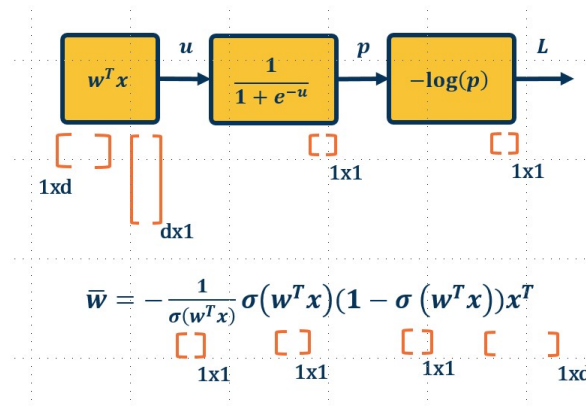
$$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \bar{p} \sigma(1 - \sigma)$$

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

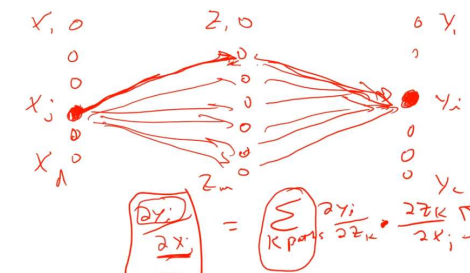
We can do this in a combined way to see all terms together:

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

This effectively shows gradient flow along path from L to w



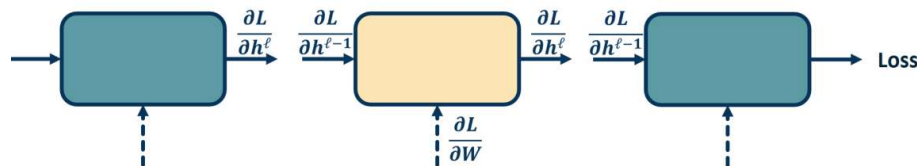
Computational / Tensor View



Graph View

Computation Graph / Global View of Chain Rule

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



Backpropagation View
(Recursive Algorithm)

Different Views of Equivalent Ideas

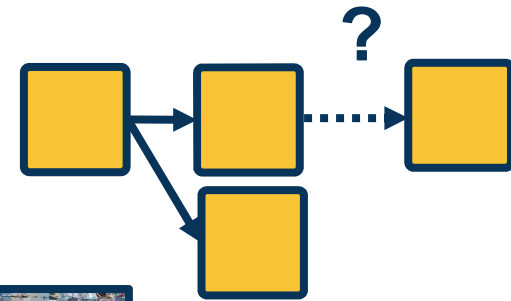
- **Backpropagation:** Recursive, modular algorithm for chain rule + gradient descent
- **When we move to vectors and matrices:**
 - Computation graph (composition of functions) => Multiplication of Jacobians along path
- **Automatic differentiation:**
 - Reduction of modules to simple operations we know (simple multiplication, etc.)
 - Automatically build computation graph in background as write code
 - Automatically compute gradients via backward pass
- **We now have a generic algorithm! Considerations:**
 - Architecture
 - Data Considerations
 - Training and Optimization
 - Machine Learning Considerations



Where We Are

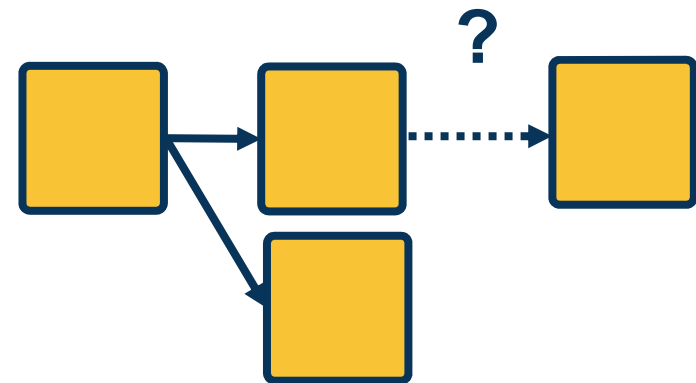
There are still many design decisions that must be made:

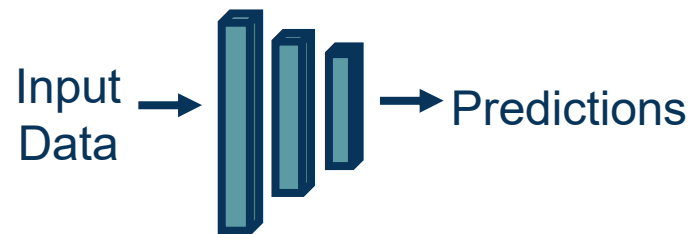
- ✦ **Architecture**
- ✦ **Data Considerations**
- ✦ **Training and Optimization**
- ✦ **Machine Learning Considerations**



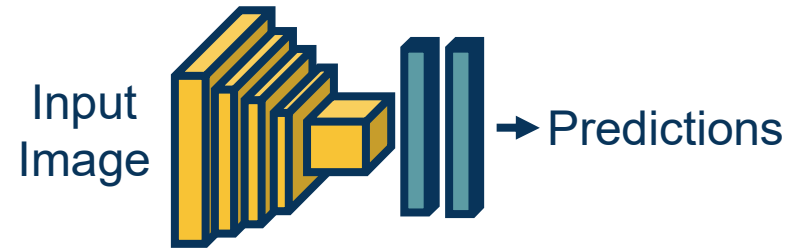
We must design the **neural network architecture**:

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

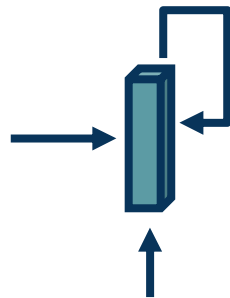




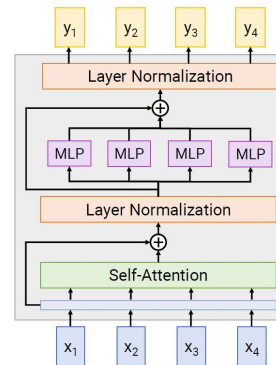
**Fully Connected
Neural Network**



**Convolutional Neural
Networks**



Recurrent Neural Network



Transformers

**Different architectures are
suitable for different
applications or types of input**

Example Architectures

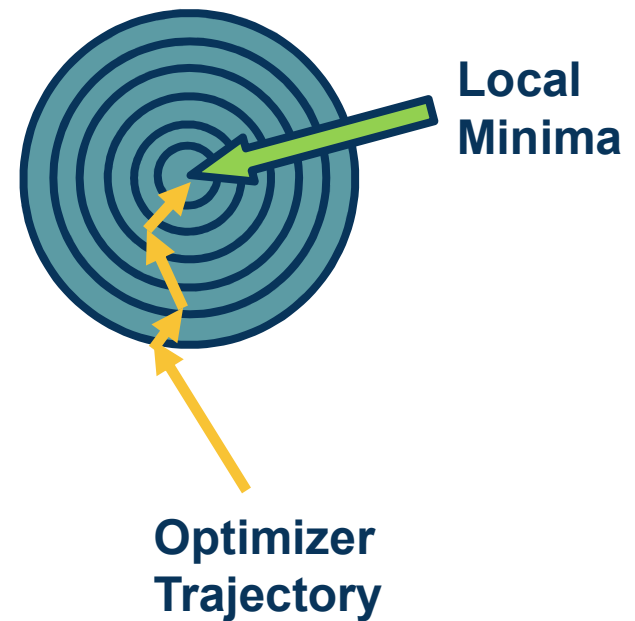
As in traditional machine learning, **data** is key:

- ◆ 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?



Machine Learning Considerations

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

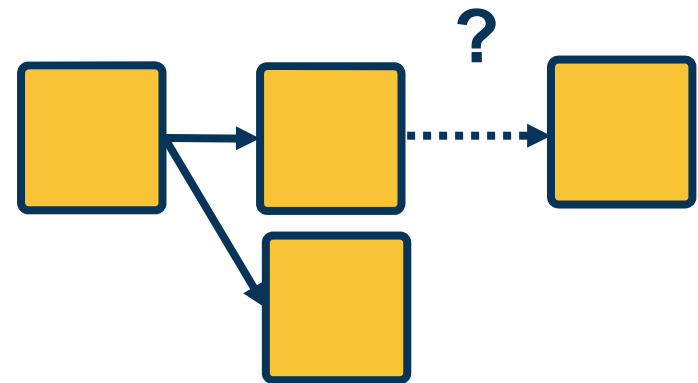
- Trade-off between **model capacity** (e.g. measured by # of parameters) and **amount of data**
- Adding **appropriate biases** based on knowledge of the domain



Architectural Considerations

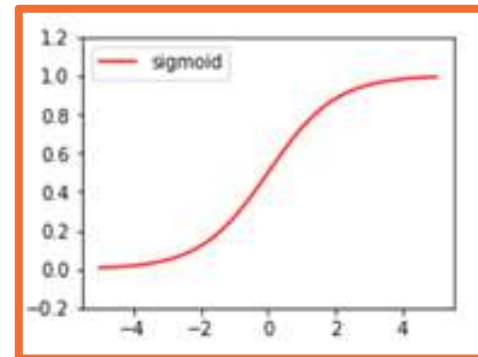
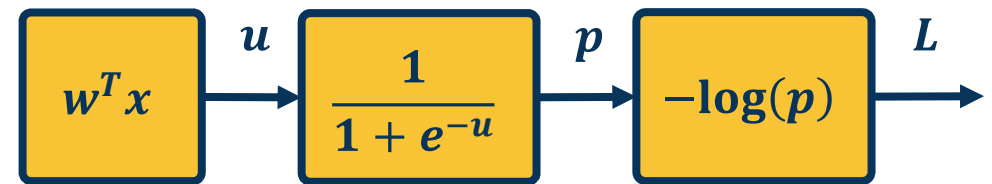
Determining what modules to use, and how to connect them is part of the **architectural design**

- ✦ 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



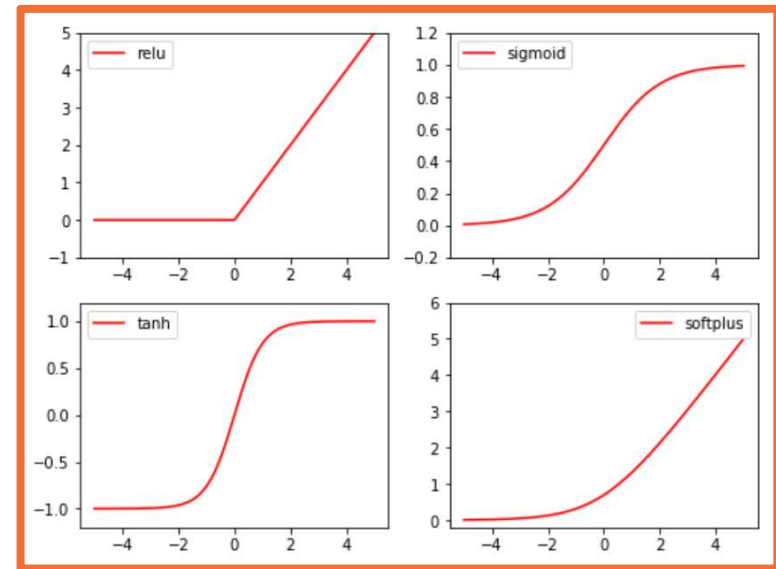
- Combination of linear and non-linear layers
- Combination of **only** linear layers has same representational power as one linear layer
- Non-linear layers** are crucial
 - Composition of non-linear layers **enables complex transformations of the data**

$$w_1^T(w_2^T(w_3^T x)) = w_4^T x$$



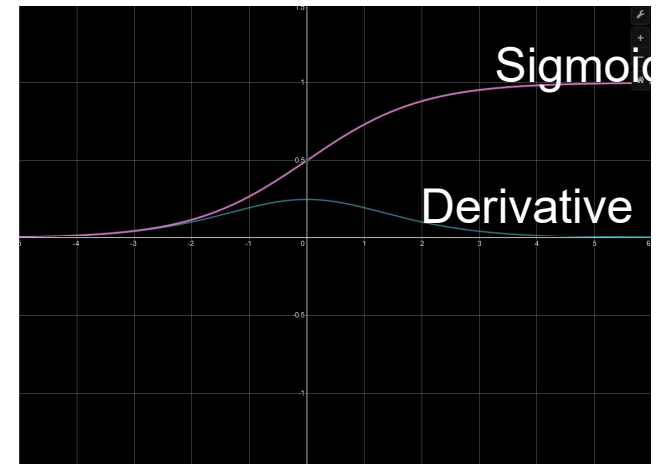
Several aspects that we can **analyze**:

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



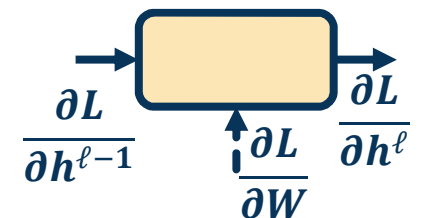
Analysis of Non-Linear Function

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



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

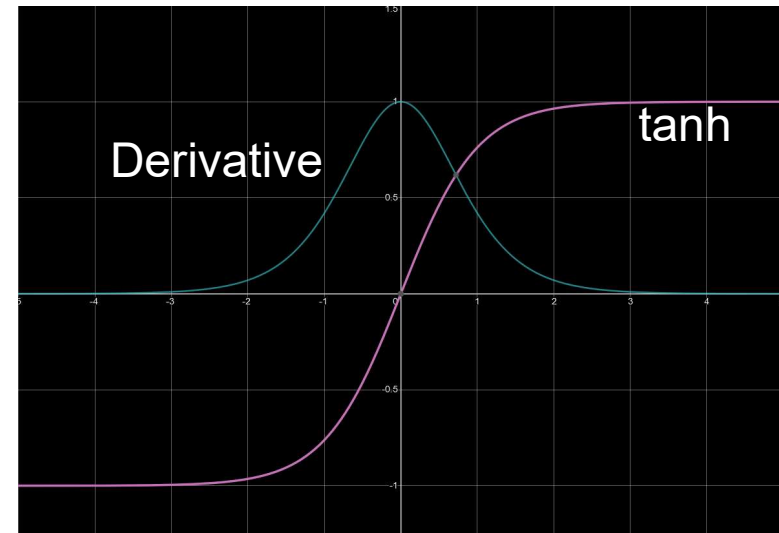
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



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

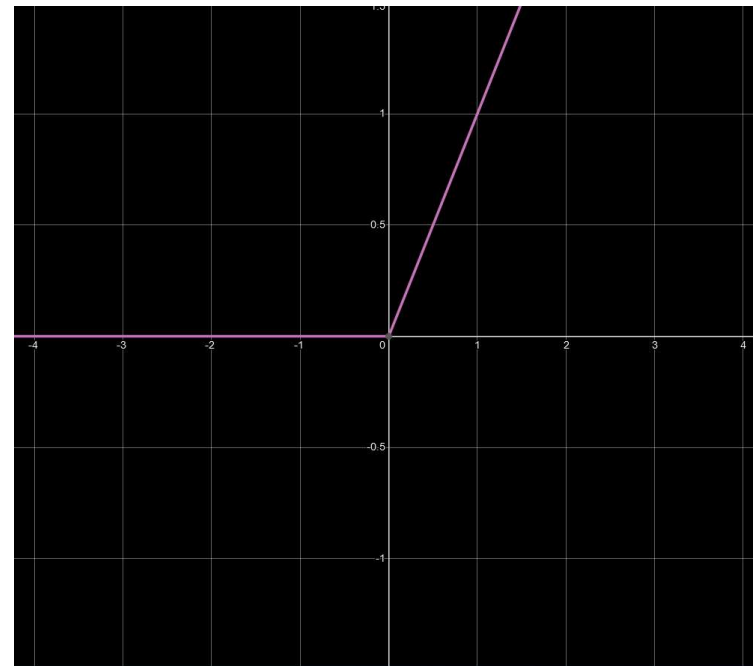
Sigmoid Function

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



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

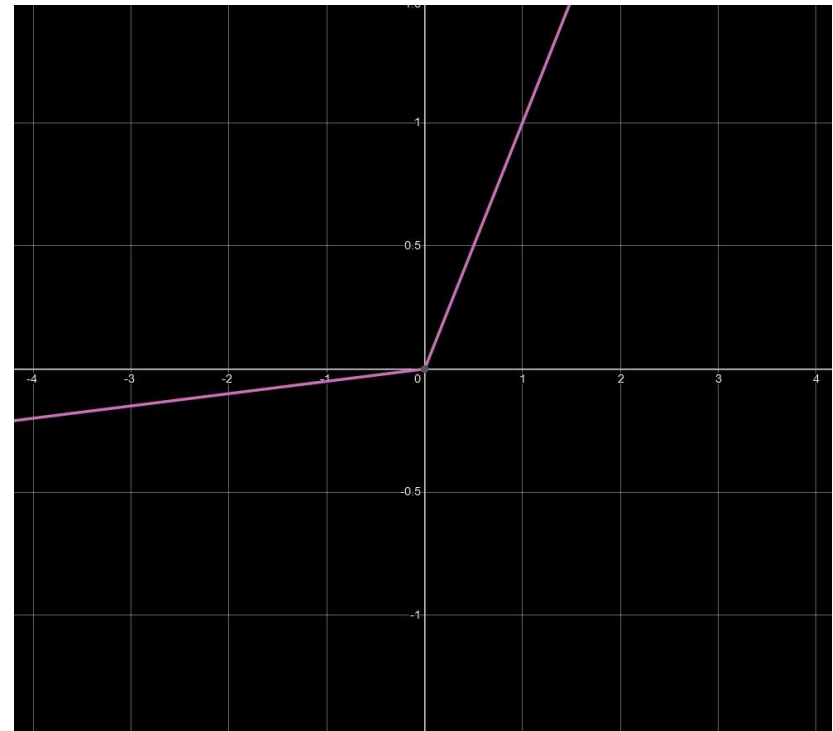
- ⬡ **Min: 0, Max: Infinity**
- ⬡ Output always **positive**
- ⬡ **No saturation** on positive end!
- ⬡ **Gradients**
 - ⬡ 0 if $x \leq 0$ (dead ReLU)
 - ⬡ Constant otherwise (does not vanish)
- ⬡ **Cheap** to compute (max)



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

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

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

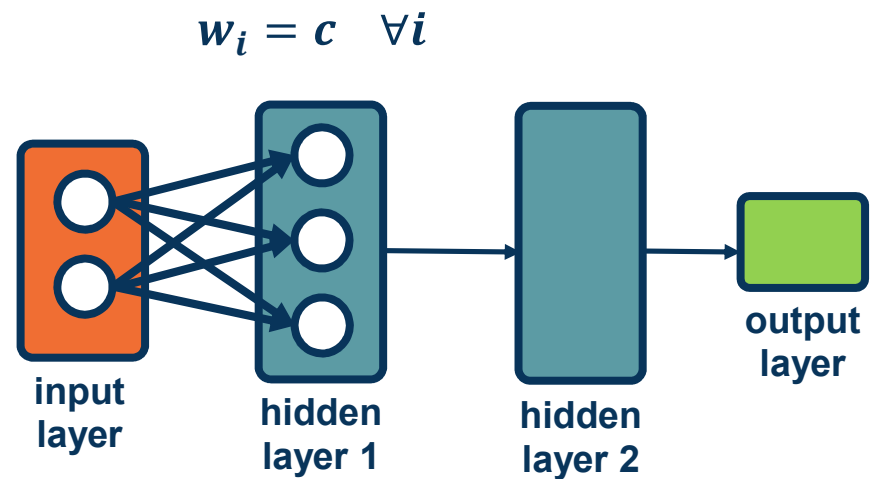
The parameters of our model must be **initialized to something**

- 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!**

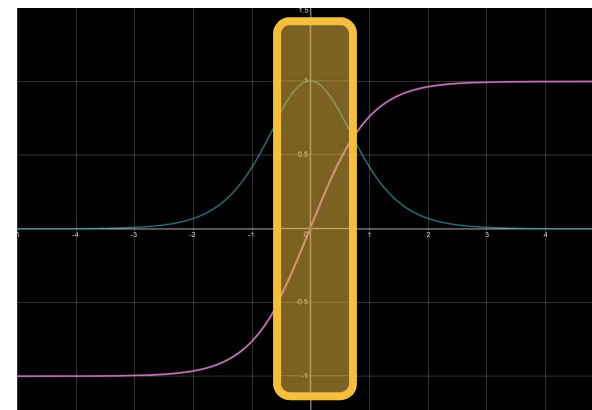
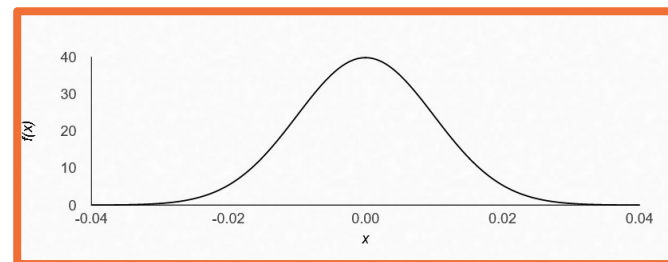
- What happens to the **weight updates**?
- Each node has the same input from previous layers so gradients **will be the same**
- As a results, **all weights will be updated** to the same exact values



A Poor Initialization

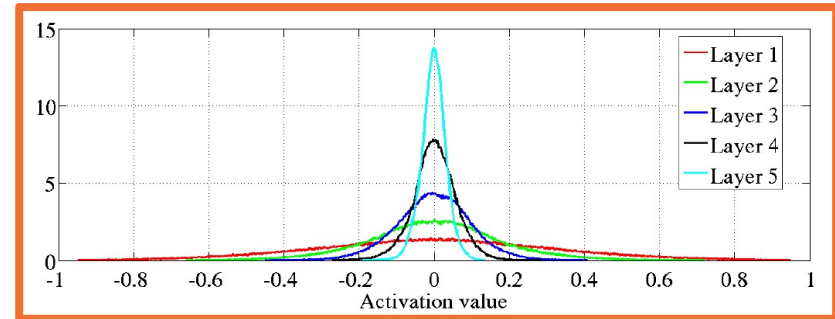
Common approach is small normally distributed random numbers

- E.g. $N(\mu, \sigma)$ 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



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 nonlinearities, for increasingly deep layers

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

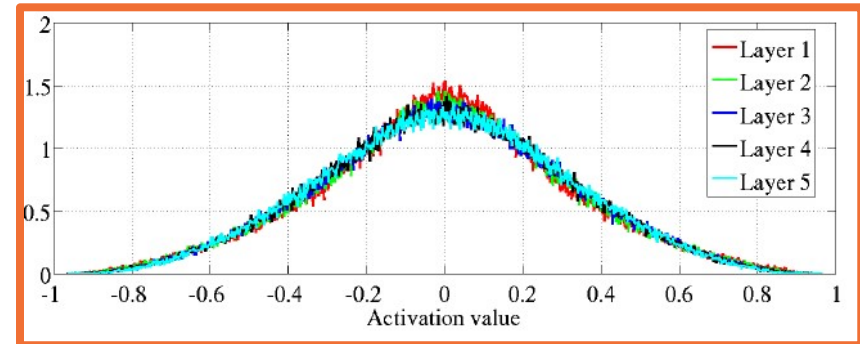
Limitation of Small Weights

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.

Xavier Initialization

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

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

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

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

"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**)

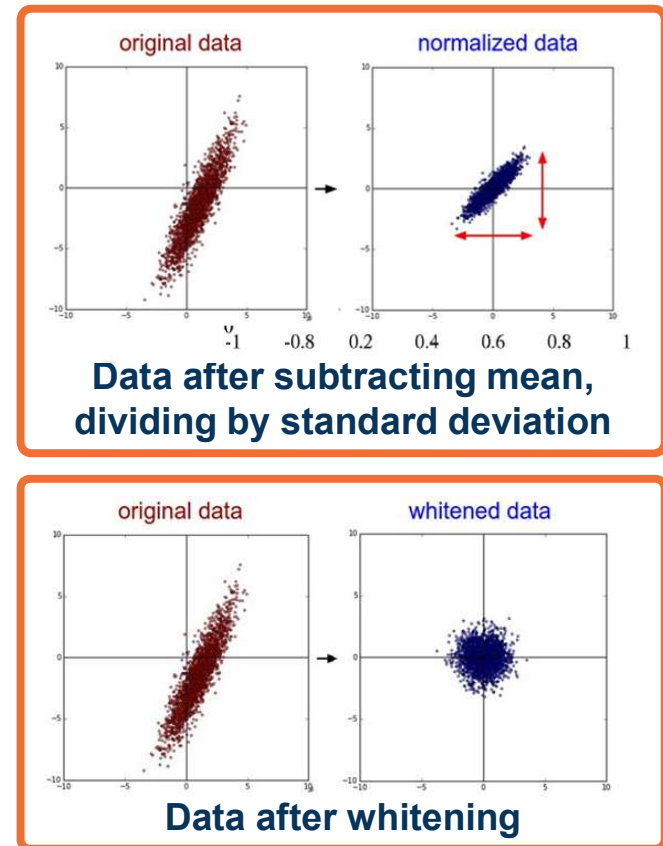


Figure from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

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

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\};$

Parameters to be learned: γ, β

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

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

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

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

Making Normalization a Layer



Normalize data

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

Note: This part does not involve new parameters

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

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

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

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

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

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: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

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

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

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

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

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

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

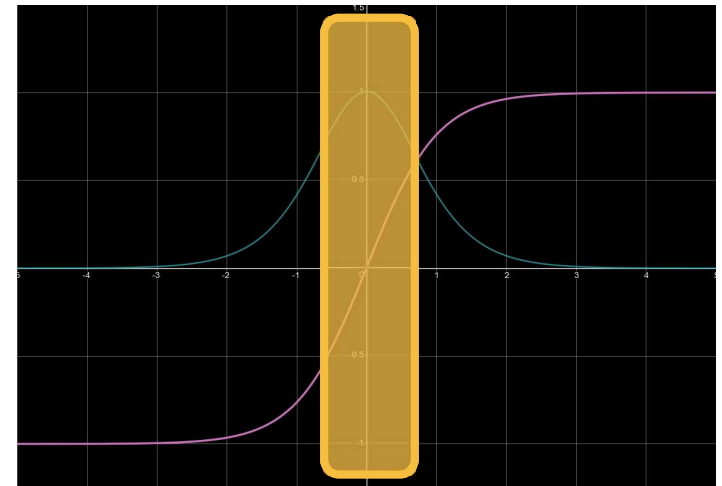
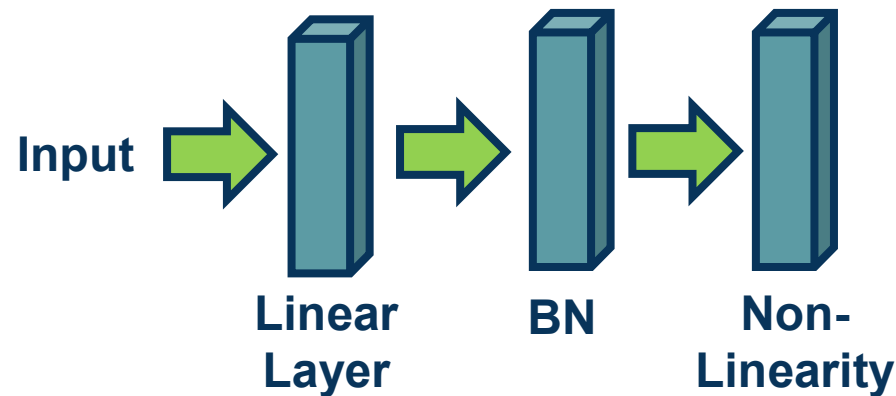
Sufficient batch sizes must be used to get stable per-batch estimates during training

- ✦ 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](#)



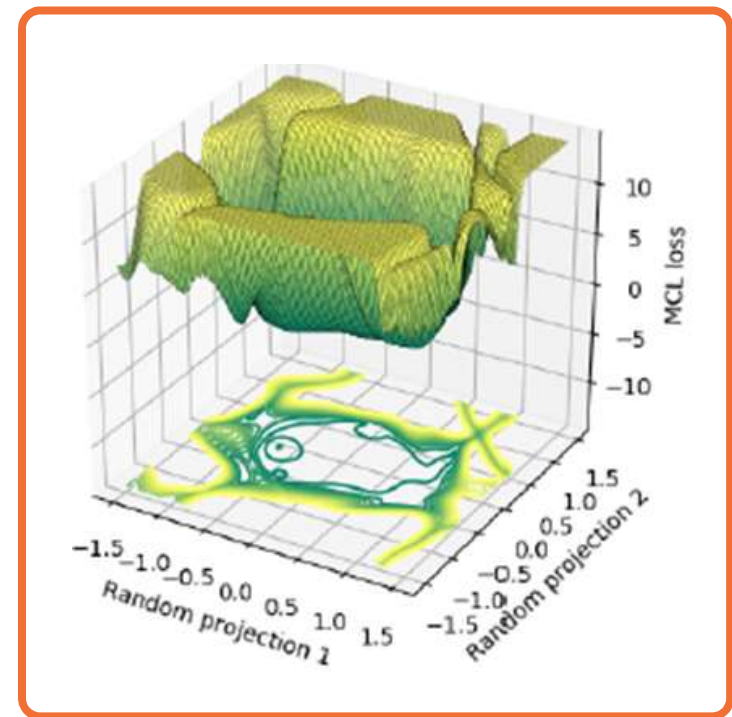
Optimizers

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

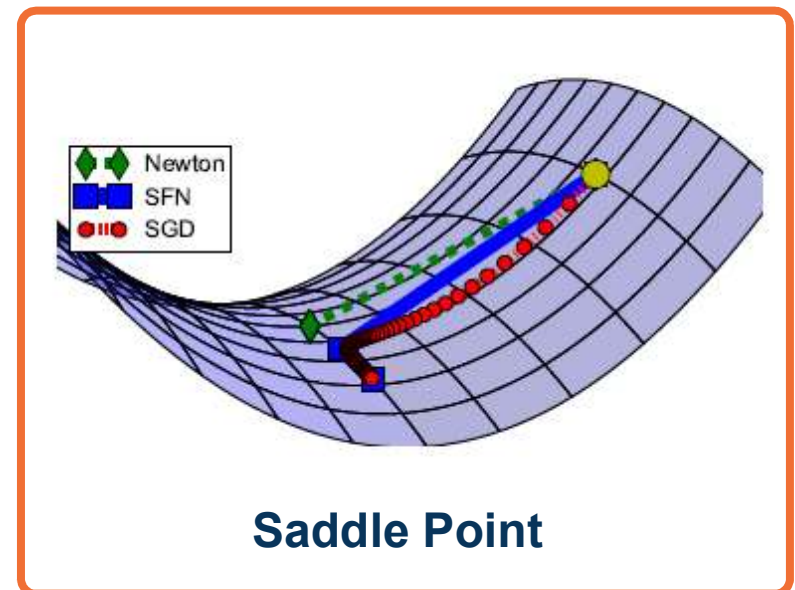
- Some insight can be gained via theory for simpler cases (e.g. convex settings)



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

There are other **more impactful issues**:

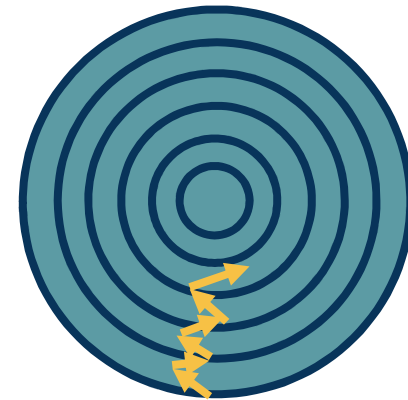
- ✦ Noisy gradient estimates
- ✦ Saddle points
- ✦ Ill-conditioned loss surface



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

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

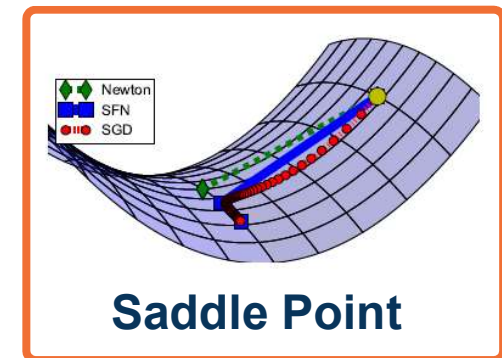
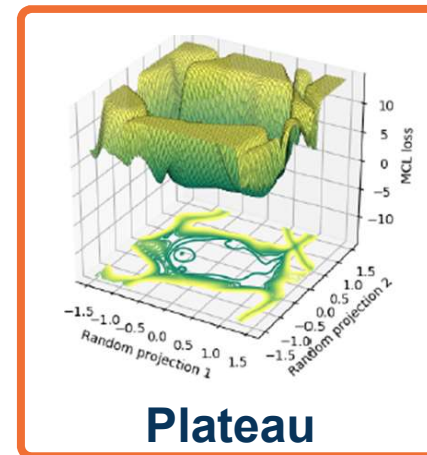


Several **loss surface geometries** are difficult for optimization

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)



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

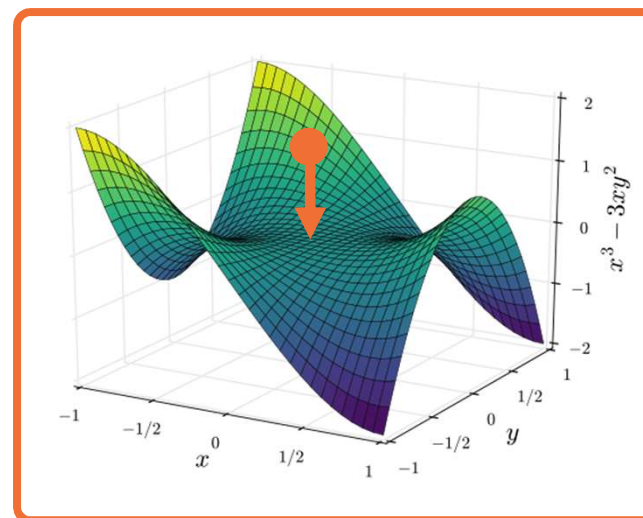
- Intuitive idea:** Imagine a ball rolling down loss surface, and use **momentum** to pass flat surfaces

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

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

- Generalizes SGD ($\beta = 0$)

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



- 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)

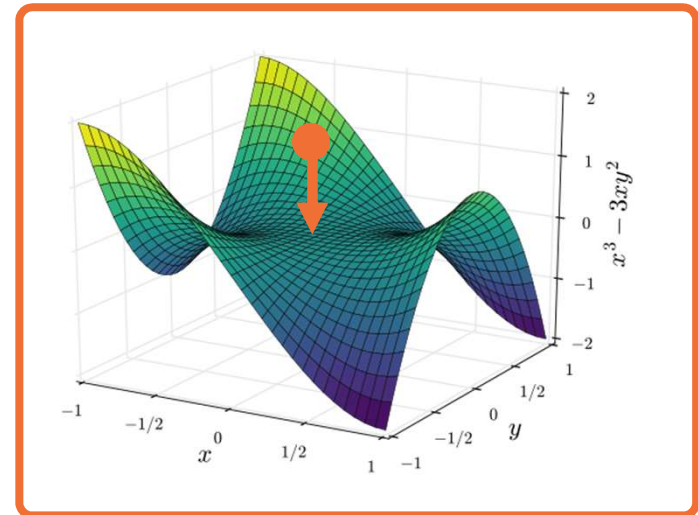
Equivalent formulation:

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

Update Velocity
(starts as 0)

$$w_i = w_{i-1} + v_i$$

Update Weights



Equivalent Momentum Update

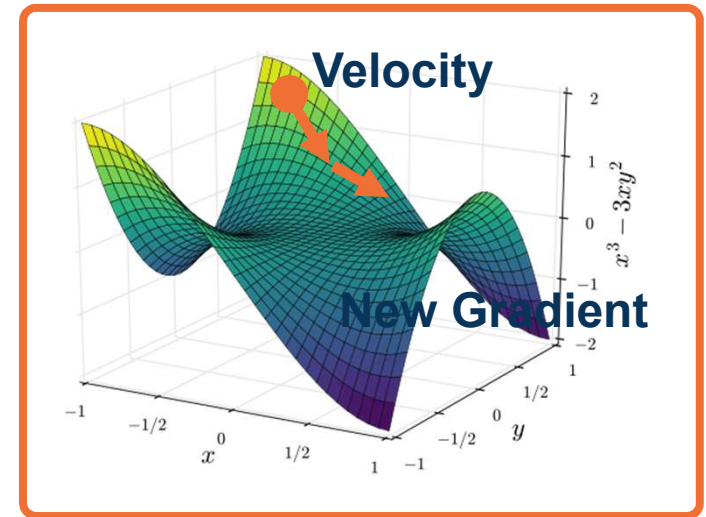
Key idea: Rather than combining velocity with current gradient, go along velocity **first** and then calculate gradient at new point

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

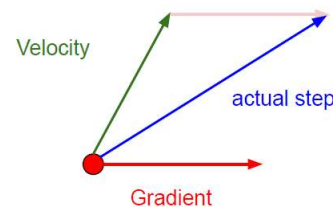
$$\hat{w}_{i-1} = w_{i-1} + \beta v_{i-1}$$

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

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



Momentum update:



Nesterov Momentum

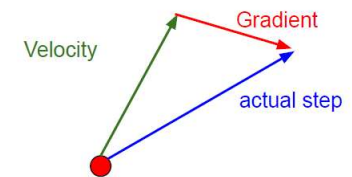


Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Nesterov Momentum

Momentum

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

Resource:

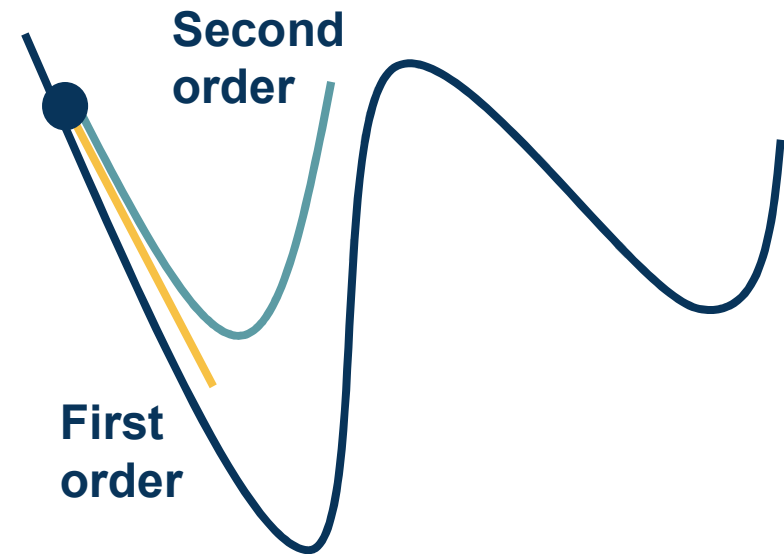
<https://medium.com/the-artificial-impostor/sgd-implementation-in-pytorch-4115bcb9f02c>



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

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

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

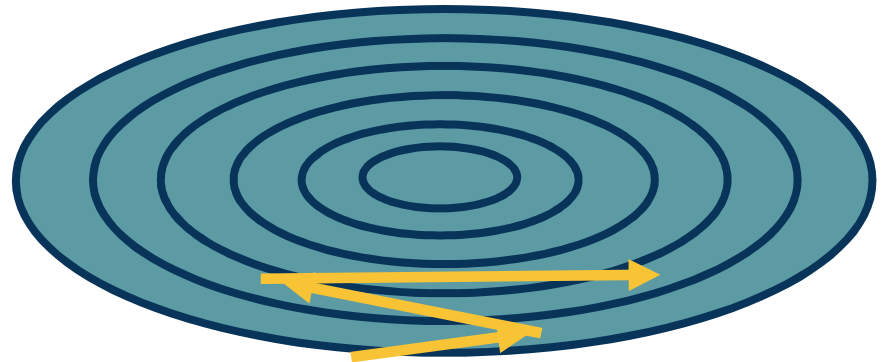


Condition number is the ratio of the largest and smallest eigenvalue

- 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"

Solution: Keep a moving average of squared gradients!

Does not saturate the learning rate

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

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

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

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)

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

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

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

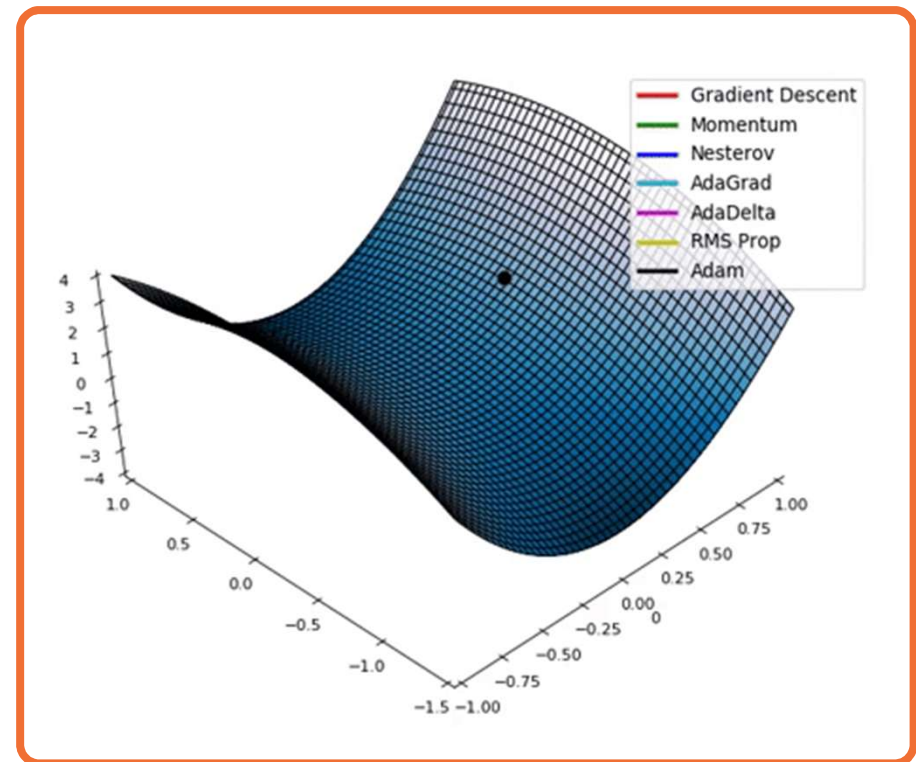
$$w_i = w_{i-1} - \frac{\alpha \hat{v}_i}{\sqrt{\hat{G}_i + \epsilon}}$$

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*



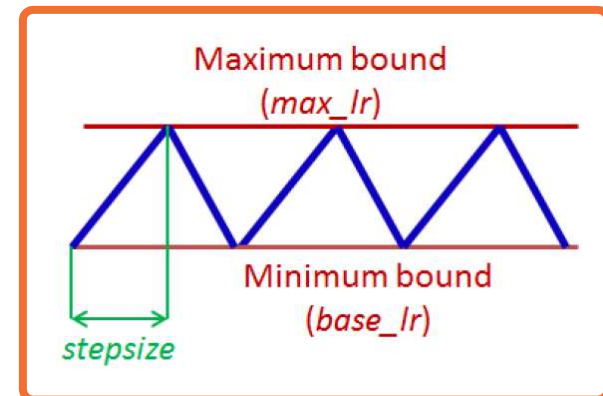
From: <https://mlfromscratch.com/optimizers-explained/#/>

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