Introduction to Neural Networks

November 17th 2020
KMI-2020
Nagoya University

Kazuhiro Terao
SLAC National Lab

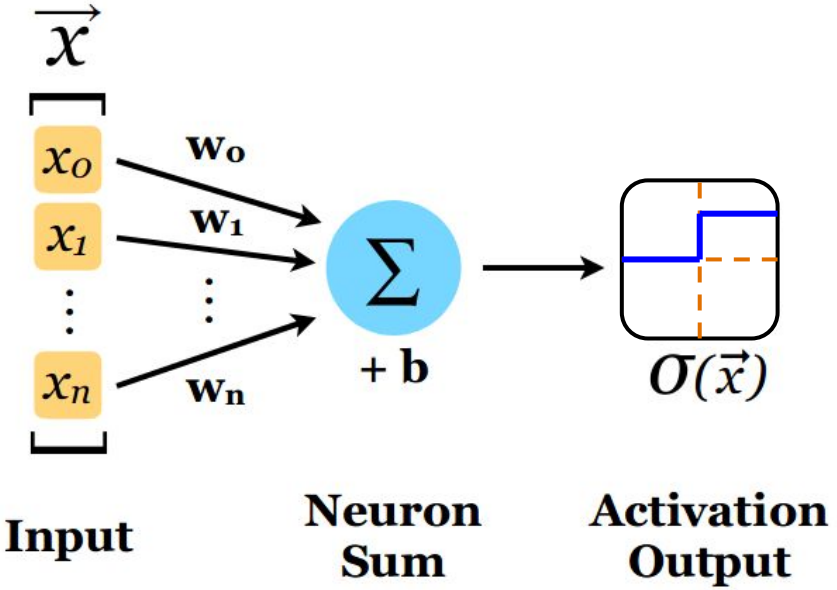
U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

Introduction to Neural Networks
Perceptron

The basic unit of a neural net is the ***perceptron*** (loosely based on a real neuron)

Takes in a vector of inputs (x). Commonly inputs are summed with weights (w) and offset (b) then run through activation.

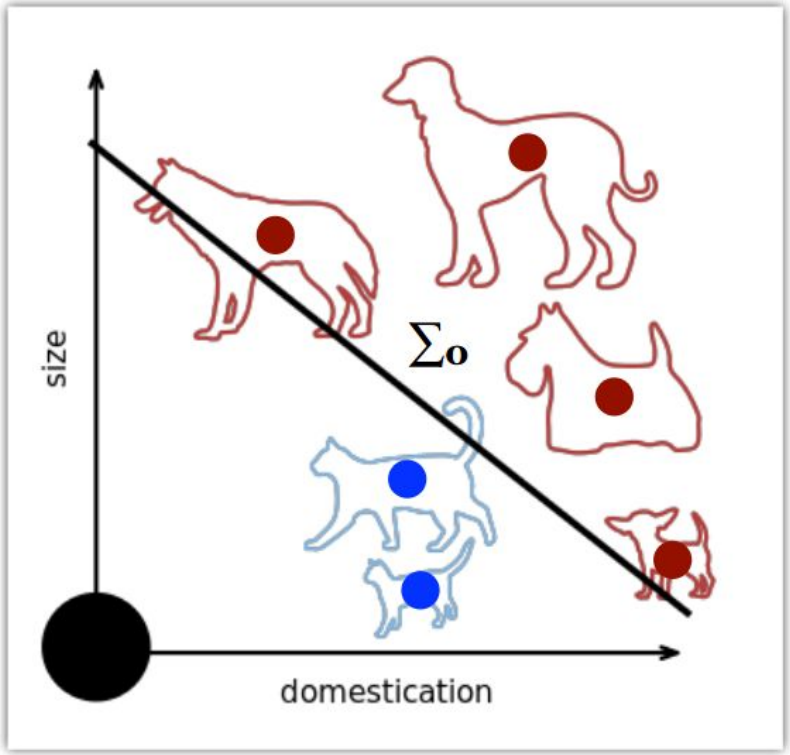


$$\sigma(\vec{x}) = \begin{cases} 1 & \vec{w}_i \cdot \vec{x} + b_i \geq 0 \\ 0 & \vec{w}_i \cdot \vec{x} + b_i < 0. \end{cases}$$

Introduction to Neural Networks
Perceptron

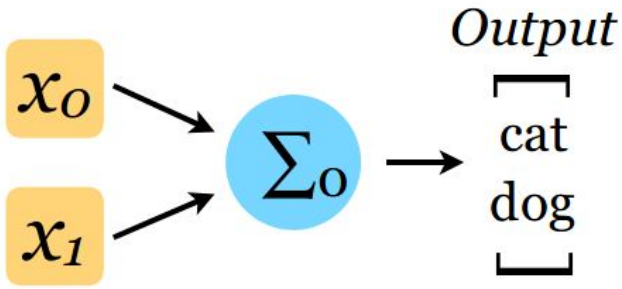
SLAC

Imagine using two features to separate cats and dogs



from wikipedia

$$\sigma(\vec{x}) = \begin{cases} 1 & \vec{w}_i \cdot \vec{x} + b_i \geq 0 \\ 0 & \vec{w}_i \cdot \vec{x} + b_i < 0. \end{cases}$$

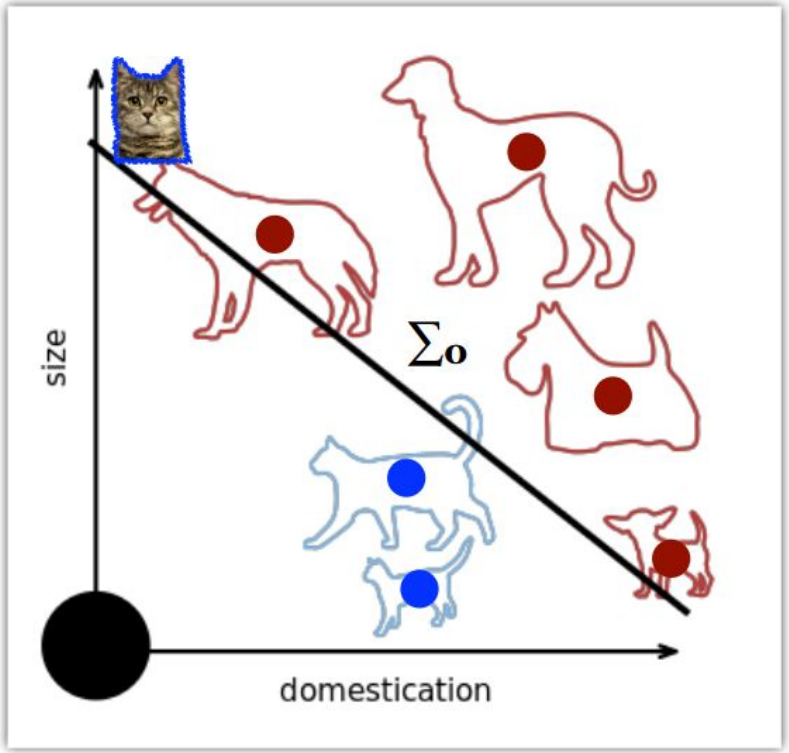


By picking a value for **w** and **b**, we define a boundary between the two sets of data

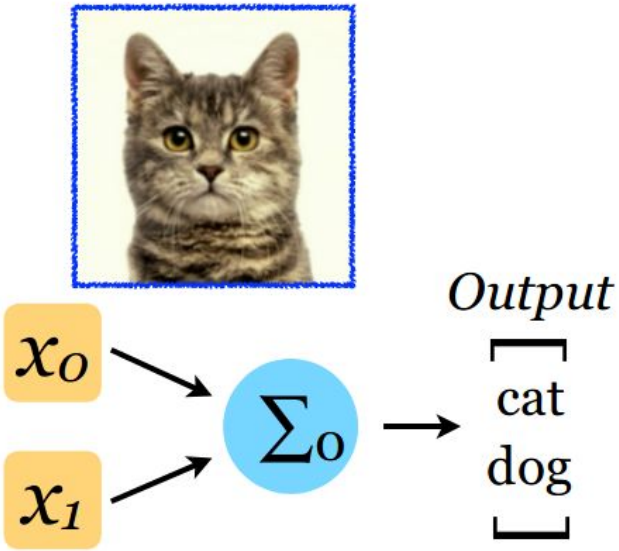
Introduction to Neural Networks
Perceptron

SLAC

What if we have a new data point?



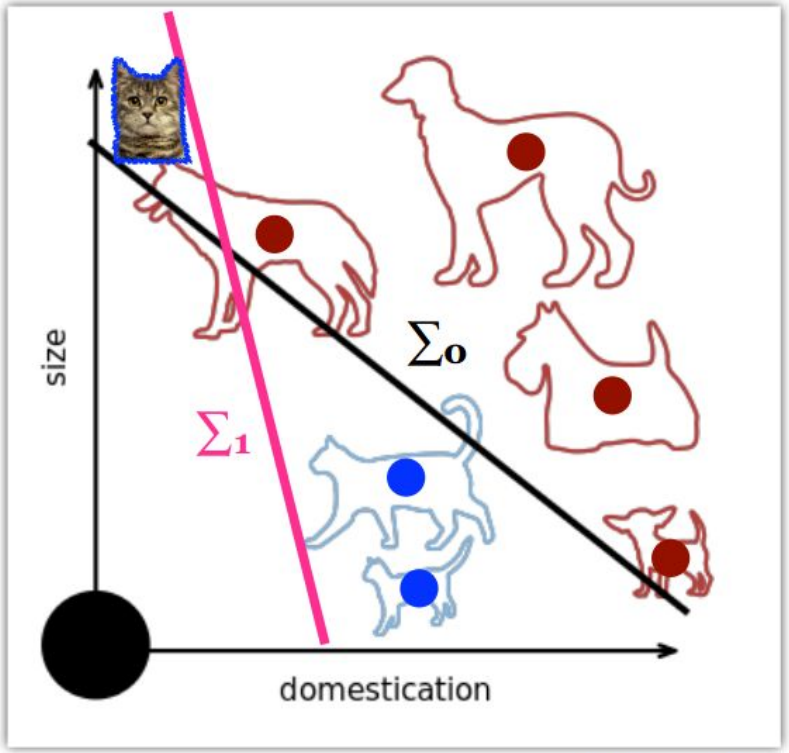
from wikipedia



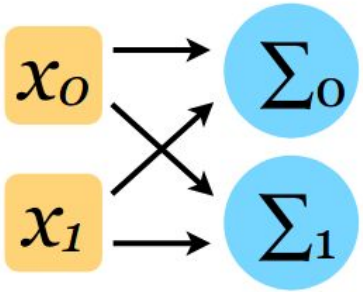
Introduction to Neural Networks
Perceptron

SLAC

What if we have a new data point?

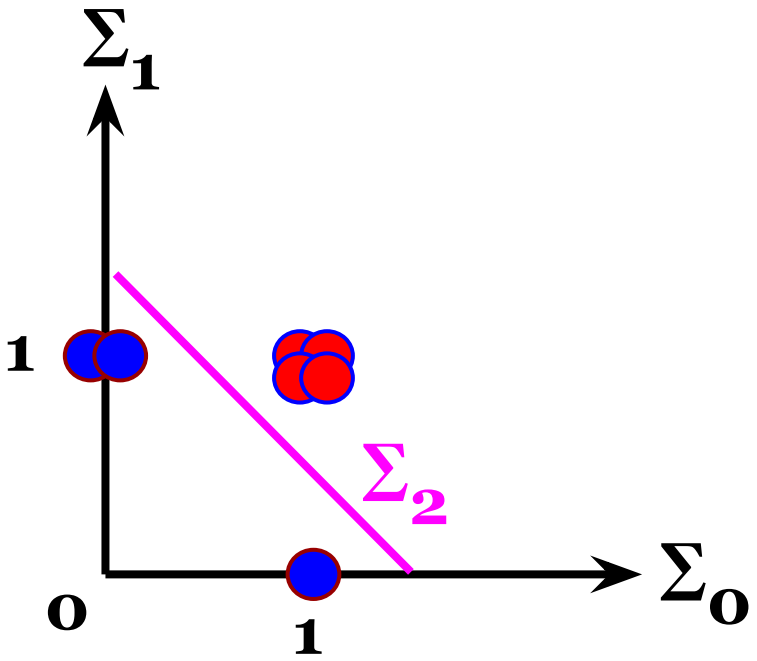


from wikipedia

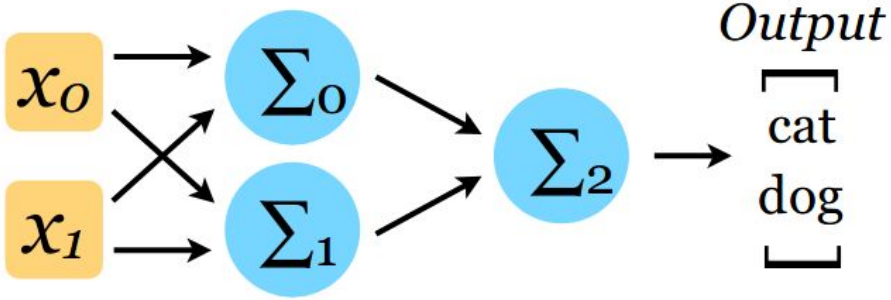


We can add **another neuron** to help (but does not yet solve the problem)

What if we have a new data point?



from wikipedia

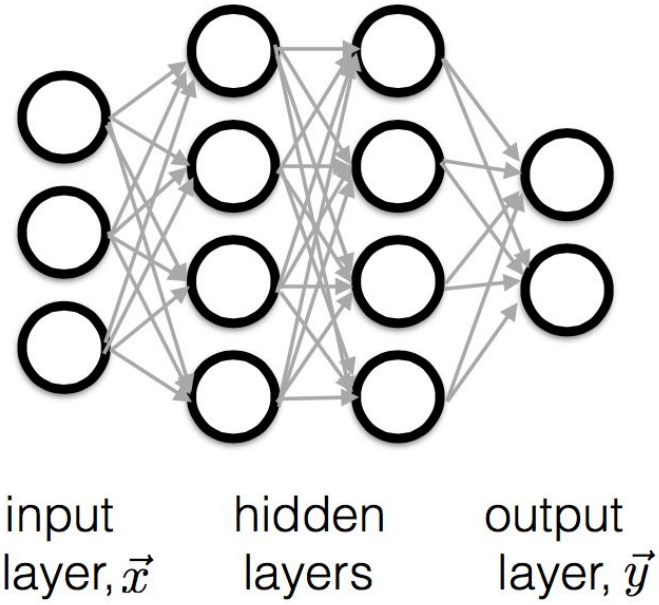


Another layer can classify based on preceding layer's output (of **non-linear activation**)

SLAC

"Traditional neural net"

Fully-Connected Multi-Layer Perceptrons



A traditional neural network consists of a stack of layers of such neurons where each neuron is ***fully connected*** to other neurons of the neighbor layers

Training Feed-forward Neural Networks Gradient-based Method

U.S. DEPARTMENT OF ENERGY
Office of Science

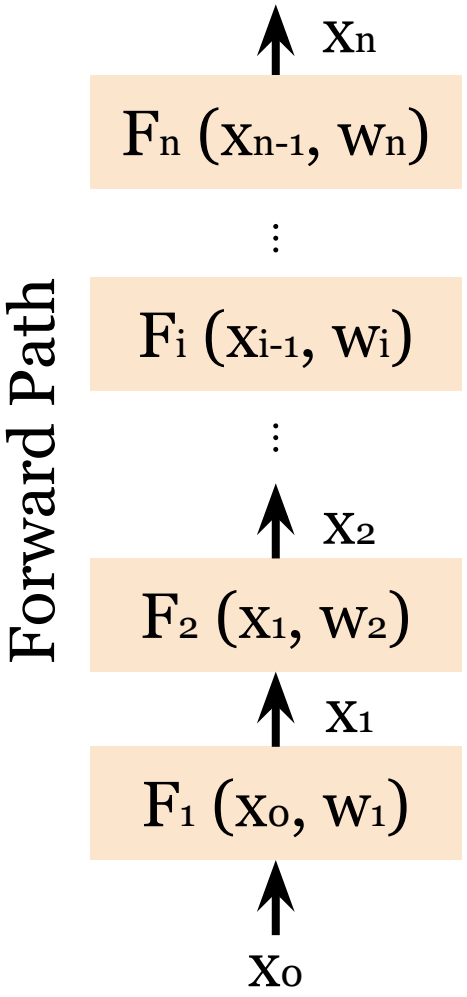
SLAC NATIONAL ACCELERATOR LABORATORY

Introduction to Neural Networks
Training (Optimization)

SLAC

N-layer MLP

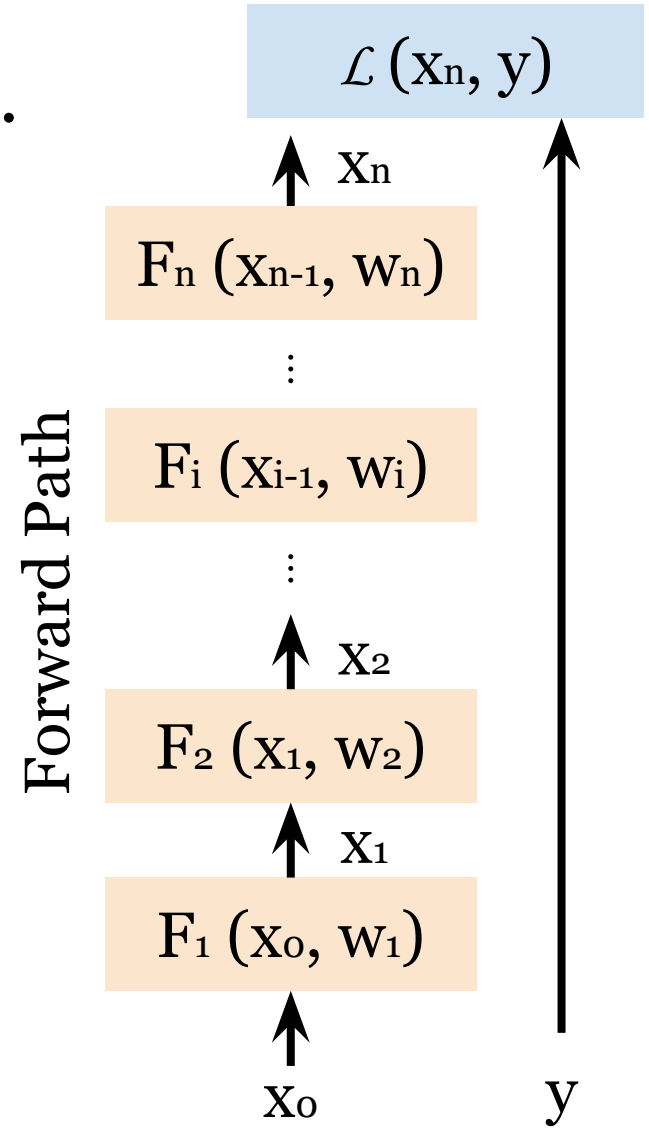
x_0 ... input data
F_i ... i-th hidden layer
x_i ... i-th layer output
w_i ... i-th layer weights
y ... correct answer
$\mathcal{L}$ ... loss (cost) function



Loss: a cost to minimize (e.g. error).

N-layer MLP

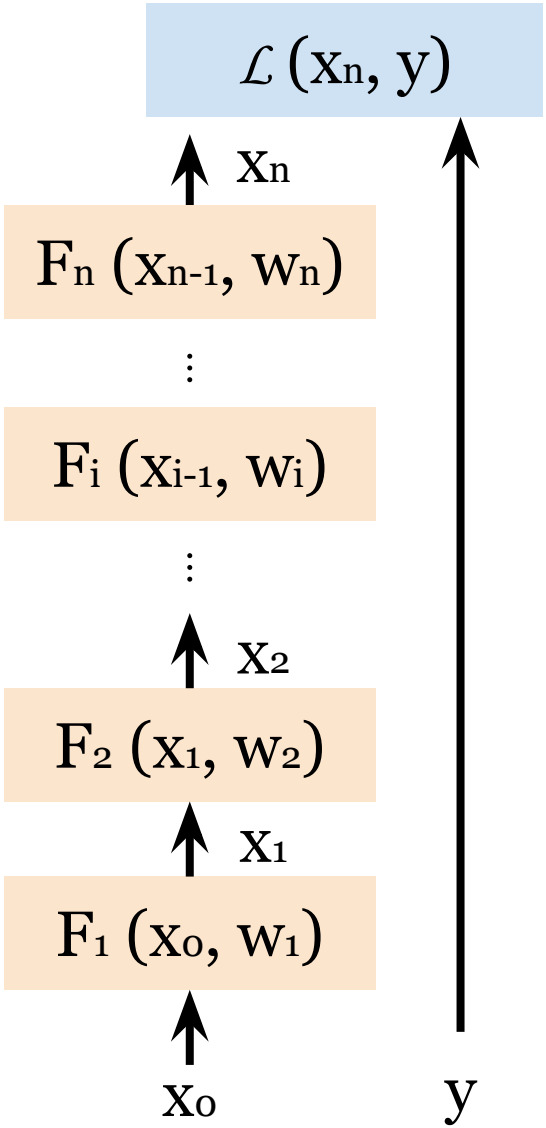
x_0 … input data
F_i … i-th hidden layer
x_i … i-th layer output
w_i … i-th layer weights
y … correct answer
$\mathcal{L}$ … loss (cost) function



Training (Optimization)

Loss: a cost to minimize (e.g. error).

Training: an optimization of parameters by minimizing loss.



Training (Optimization)

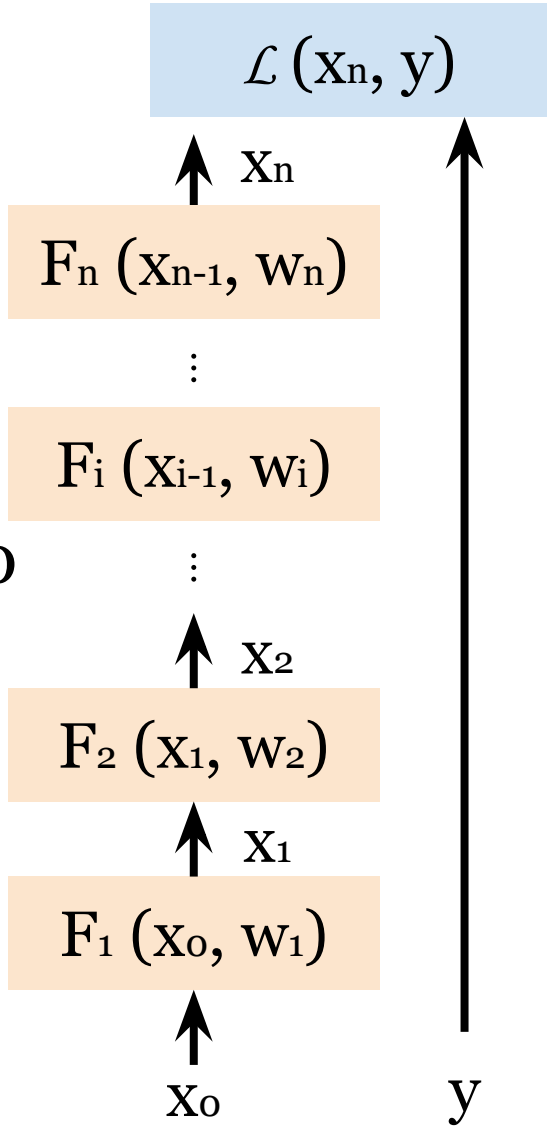
Loss: a cost to minimize (e.g. error).

Training: an optimization of parameters by minimizing loss.

Gradient Descend: an iterative approach to optimize weights in order to minimize the loss.

- Define & measure the error
- Compute $\partial \mathcal{L}/\partial w_n = \nabla W$
- Update weights: $W_{new} = W - \lambda \nabla W$

... where λ is called a "learning rate"



Backpropagation: how to compute gradients

How can we compute the gradient for the *i*-th layer?

$$\mathcal{L}(x_n, y) = \mathcal{L}(F_n(x_{n-1}, w_n), y) = \ldots = \mathcal{L}(F_n(\ldots F_i(x_{i-1}, w_i)\ldots), y)$$

Backpropagation: how to compute gradients

How can we compute the gradient for the *i*-th layer?

$$\mathcal{L}(x_n, y) = \mathcal{L}(F_n(x_{n-1}, w_n), y) = \ldots = \mathcal{L}(F_n(\ldots F_i(x_{i-1}, w_i)\ldots), y)$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_{i+1}}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_i} \quad \text{(chain rule)}$$

Backpropagation: how to compute gradients

How can we compute the gradient for the *i*-th layer?

$$\mathcal{L}(x_n, y) = \mathcal{L}(F_n(x_{n-1}, w_n), y) = \ldots = \mathcal{L}(F_n(\ldots F_i(x_{i-1}, w_i)\ldots), y)$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \boxed{\frac{\partial \mathcal{L}}{\partial x_n}} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_{i+1}}{\partial x_i} \cdot \frac{\partial x_i}{\partial w_i} \quad \text{(chain rule)}$$

Example: quadratic loss $= \sum_{m=1}^{m} (x_n - y)^2$

$$\frac{\partial \mathcal{L}}{\partial x_n} = \sum_{m=1}^{m} (2x_n - 2y)$$

... or $$\frac{\partial \mathcal{L}}{\partial x_n} = \sum_{m=1}^{m} (2x_n - 2y)^T$$ if x_n and y are column vector

Backpropagation: how to compute gradients

How can we compute the gradient for the *i*-th layer?

$$\mathcal{L}(x_n, y) = \mathcal{L}(F_n(x_{n-1}, w_n), y) = \ldots = \mathcal{L}(F_n(\ldots F_i(x_{i-1}, w_i)\ldots), y)$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \boxed{\frac{\partial \mathcal{L}}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_{i+1}}{\partial x_i}} \cdot \frac{\partial x_i}{\partial w_i} \quad \text{(chain rule)}$$

$$= \frac{\partial \mathcal{L}}{\partial x_i} \quad \ldots \textbf{ can compute iteratively}$$

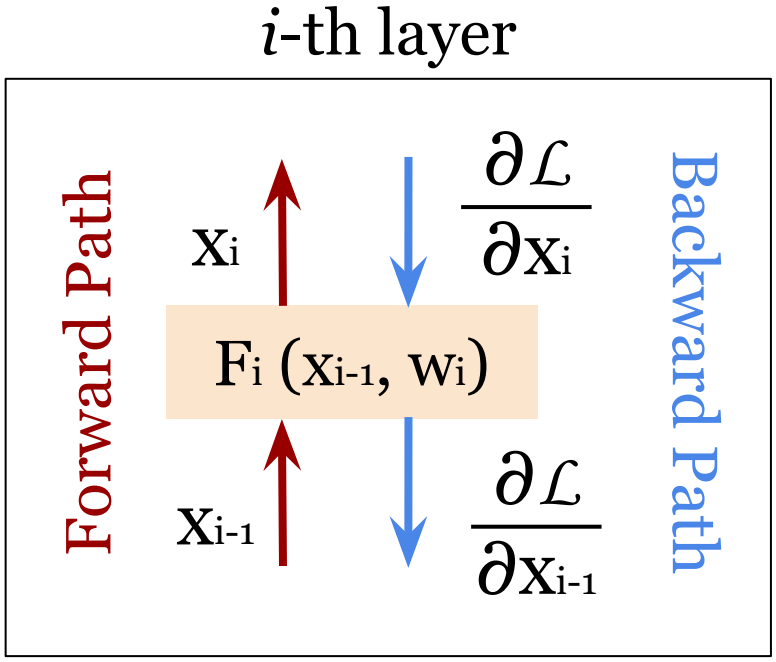
("Back" propagation)

If we know $\frac{\partial \mathcal{L}}{\partial x_i}$ (gradient at i-th layer), then we can calculate:

$$\frac{\partial \mathcal{L}}{\partial x_{i-1}} = \frac{\partial \mathcal{L}}{\partial x_i} \cdot \frac{\partial x_i}{\partial x_{i-1}} = \frac{\partial \mathcal{L}}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

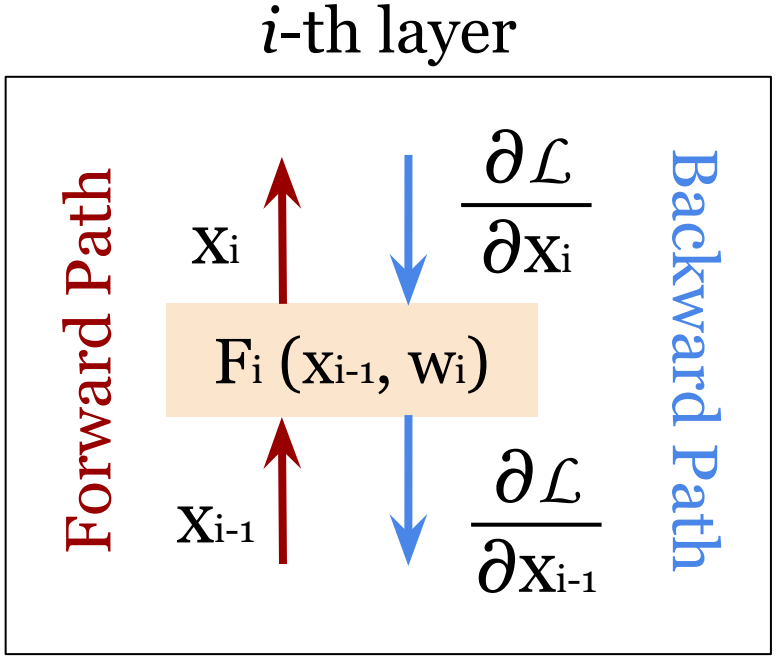
Backpropagation: how to compute gradients

How can we compute the gradient for the i-th layer?



Backpropagation: how to compute gradients

How can we compute the gradient for the *i*-th layer?



Input: x_{i-1} $\frac{\partial \mathcal{L}}{\partial x_i}$ (given)

Output (we compute):

$$x_i = F_i(x_{i-1}, w_i)$$

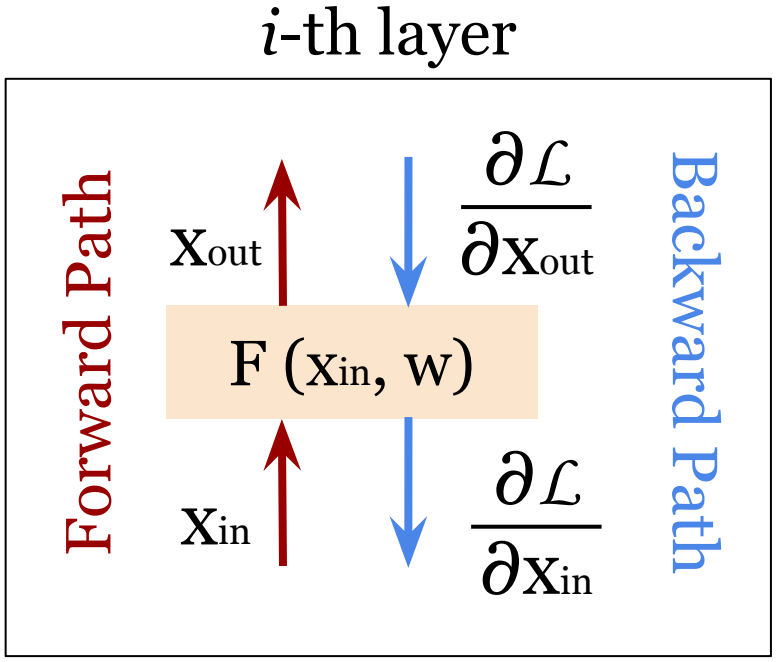
$$\frac{\partial \mathcal{L}}{\partial x_{i-1}} = \frac{\partial \mathcal{L}}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

Weight Update: $w_i - \lambda \nabla w_i$

where: $\nabla w_i = \frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$

Backpropagation: how to compute gradients

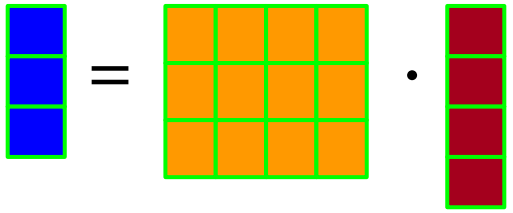
How can we compute the gradient for the *i*-th layer?



Example: linear transformation

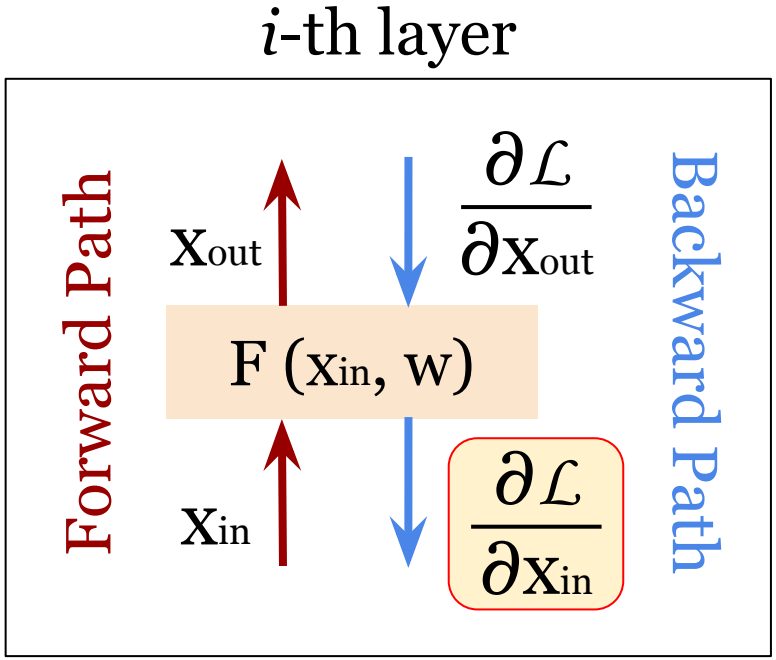
$$F(x_{in}, w) = w \cdot x_{in}$$

Let's generalize for a tensor input

e.g.) 

Backpropagation: how to compute gradients

How can we compute the gradient for the i-th layer?

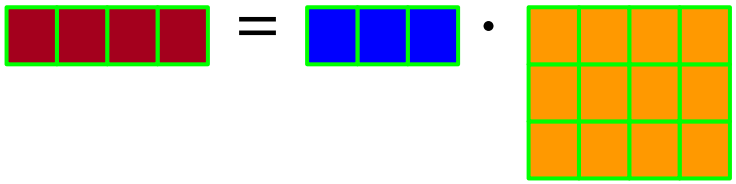


Example: linear transformation

$$F(x_{in}, w) = w \cdot x_{in}$$

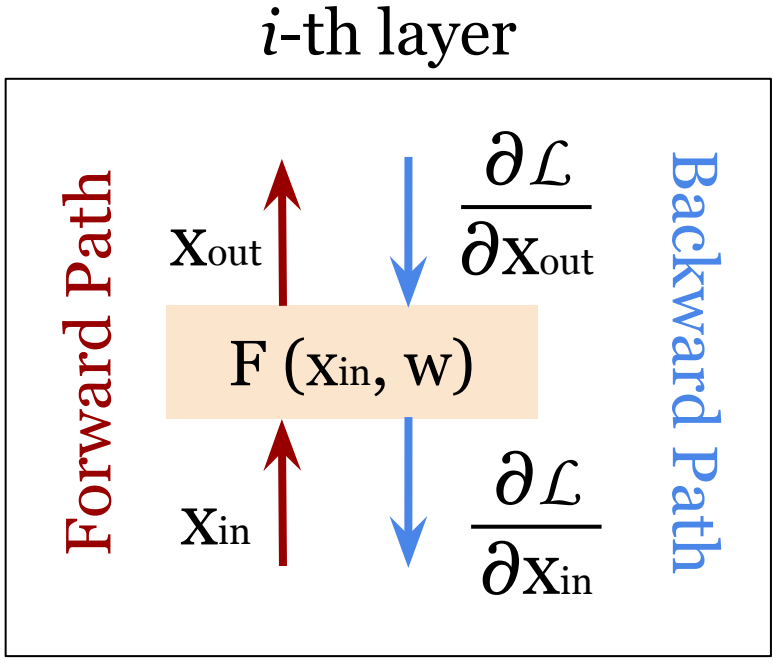
Let's generalize for a tensor input

$$\frac{\partial \mathcal{L}}{\partial x_{in}} = \frac{\partial \mathcal{L}}{\partial x_{out}} \cdot w$$

e.g.) 

Backpropagation: how to compute gradients

How can we compute the gradient for the i-th layer?



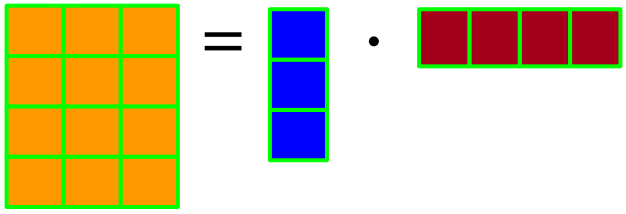
Example: linear transformation

$$F(x_{in}, w) = w \cdot x_{in}$$

Let's generalize for a tensor input

$$\frac{\partial \mathcal{L}}{\partial x_{in}} = \frac{\partial \mathcal{L}}{\partial x_{out}} \cdot w$$

$$\frac{\partial \mathcal{L}}{\partial w} = x_{in} \cdot \frac{\partial \mathcal{L}}{\partial x_{out}}$$

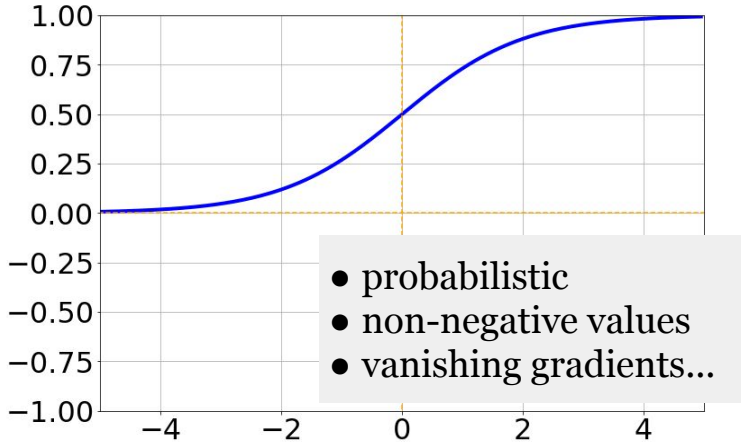


$$w_{new} = w - \lambda \left(\frac{\partial \mathcal{L}}{\partial w}\right)^T$$

Introduction to Neural Networks
Activation Functions

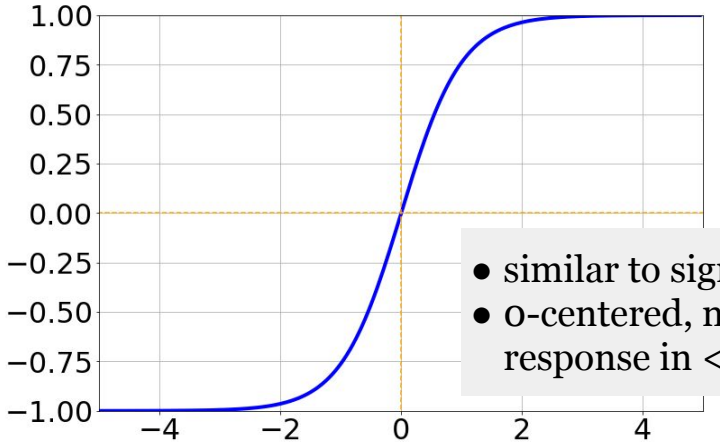
SLAC

Sigmoid



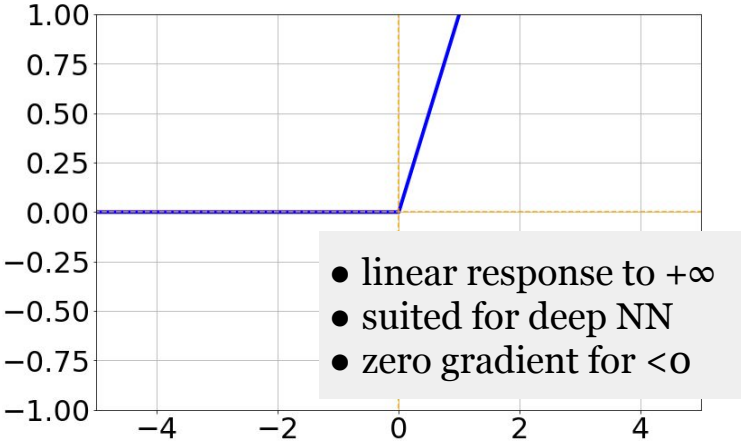
- probabilistic
- non-negative values
- vanishing gradients...

Tanh



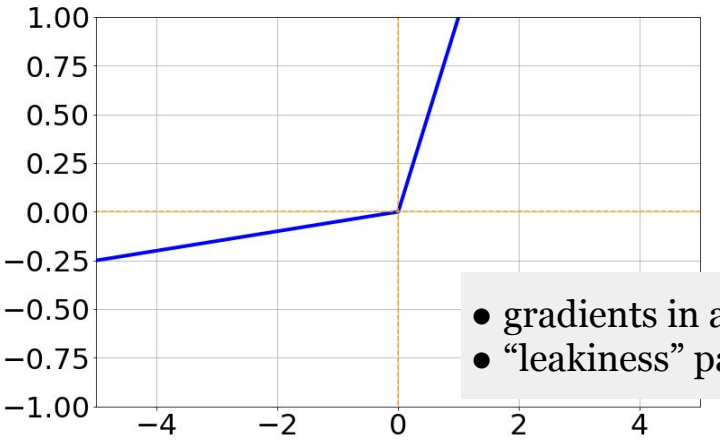
- similar to sigmoid
- 0-centered, non-zero response in <0

ReLU



- linear response to +∞
- suited for deep NN
- zero gradient for <0

Leaky ReLU



- gradients in all regions
- “leakiness” parameter

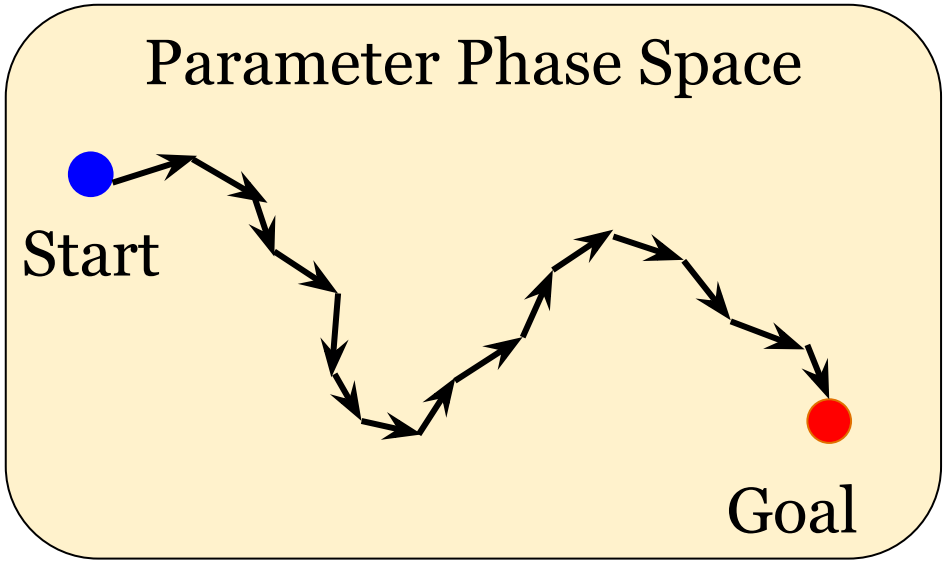
Weights Initialization

- **Random values** to set different neuron states
- **Not too large or too small**: gradients become negligible for *tanh* and *sigmoid* activation layers! (positive large value not an issue for ReLU)
- **Gaussian** @ $(\mu,\sigma) = (0,1)$?
  - Slight modification: normalize such that the variance of a signal strength stays similar (0~1) across layers (depends on # of filters)
  - See more details here (CS231)
  - Xavier initialization, He (MSRA) initialization

Start: some random initial set of parameters

Goal: minimize the overall loss = sum over all samples

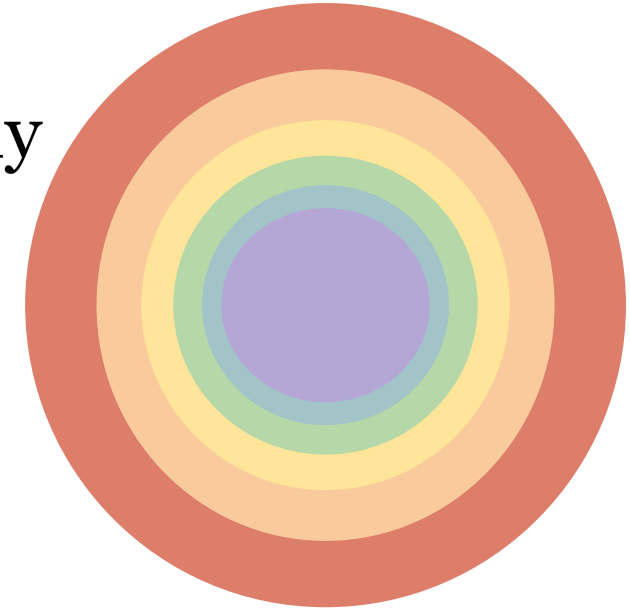
How: SGD from yesterday :)



Adaptive learning rate (LR)

Ideally start with a larger LR to quickly converge, then decrease the LR to fine-tune near the minimum.

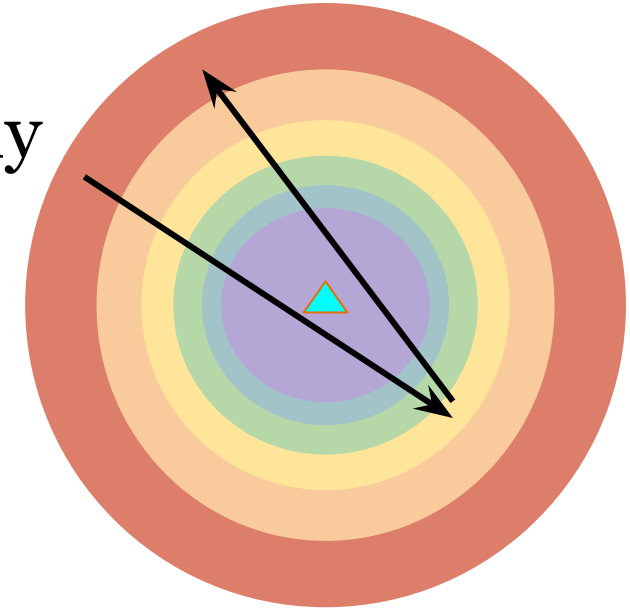
Warning! too small LR rate is not only slow but can trap in false minimum.



Adaptive learning rate (LR)

Ideally start with a larger LR to quickly converge, then decrease the LR to fine-tune near the minimum.

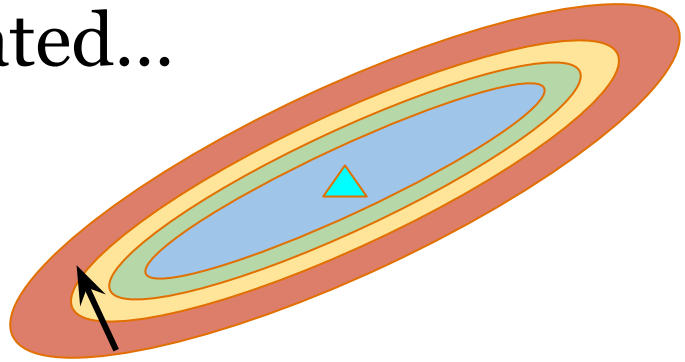
Warning! too small LR rate is not only slow but can trap in false minimum.



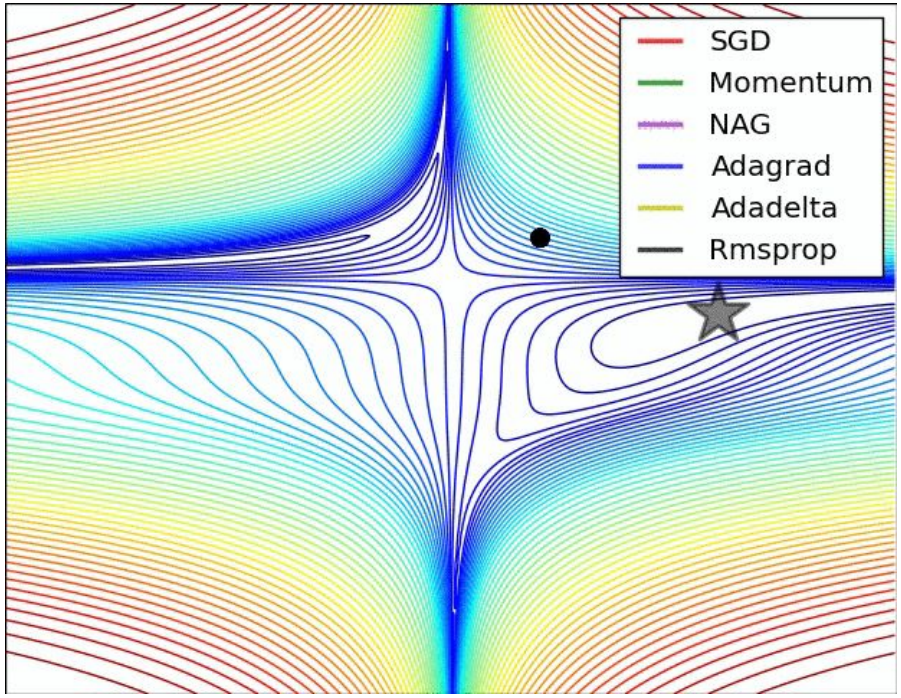
Momentum

The direction of the steepest descent can be almost perpendicular if the ellipse is elongated…

A solution is to average the history of past updates. This accumulates small updates in the right direction



Popular Visualization of Optimizers (not mine)



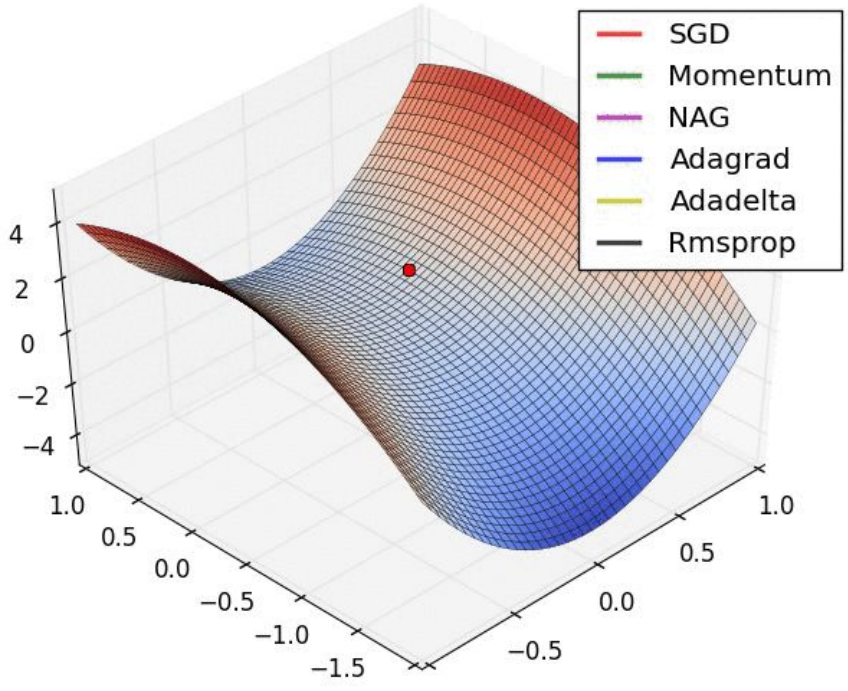


Image Credit: Alec Radford

- Excellent lecture here (CS232)
- Want to try by yourself ? You can get started here.
- Popular choices: vanila SGD and Adam

Neural Network

- Each neuron produces a “feature”, activation function can add non-linearity, and additional layer can represent non-linear functional approximation.

Gradient-based Optimization

- Essentially a composite function = chain rule!
- Initialization should be non-zero random values
- Flavors of gradient-based optimizers to take into account for the error surface

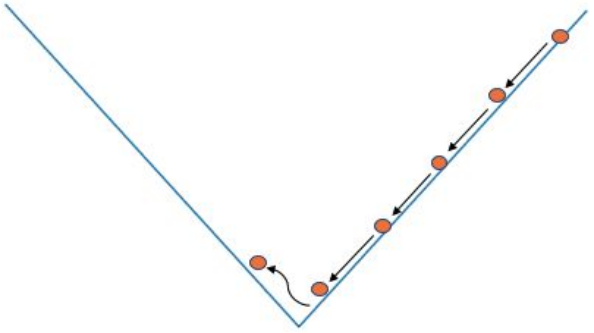
Backup Slides

Cost Functions & Activation Functions

1. **Mean Absolute Error** (MAE, L1 loss)

$$\mathrm{MAE} = \frac{\sum^{n} \left| x_{\mathrm{pred}} - y_{\mathrm{true}} \right|}{n}$$

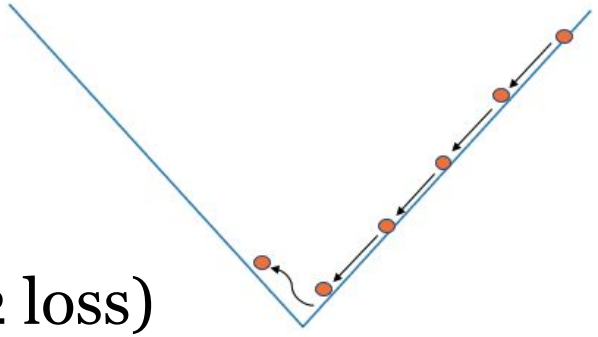
- Constant magnitude gradient everywhere
- Adaptive LR critical near the minimum



1. **Mean Absolute Error** (MAE, L1 loss)

$$MAE = \frac{\sum^{n} |x_{pred} - y_{true}|}{n}$$

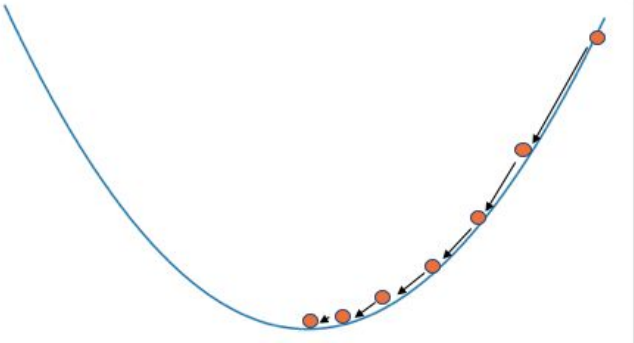
- Constant magnitude gradient everywhere
- Adaptive LR critical near the minimum



2. **Mean Square Error** (MSE, L2 loss)

$$MSE = \frac{\sum^{n} (x_{pred} - y_{true})^2}{n}$$

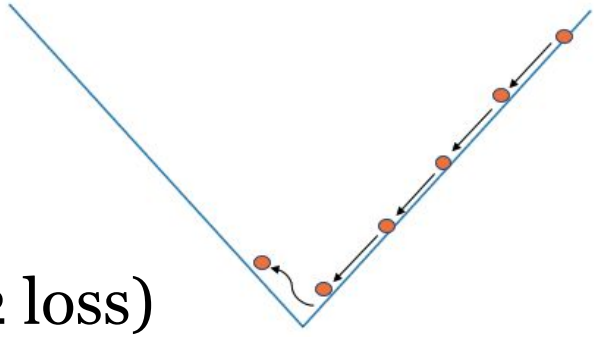
- Adaptive LR not needed
- Huge gradient far from the minimum



1. **Mean Absolute Error** (MAE, L1 loss)

$$MAE = \frac{\sum^{n} | x_{pred} - y_{true} |}{n}$$

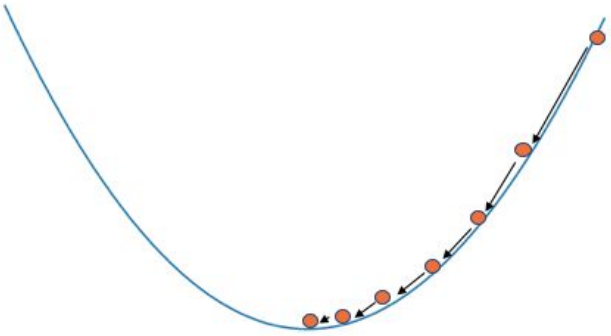
- Constant magnitude gradient everywhere
- Adaptive LR critical near the minimum



2. **Mean Square Error** (MSE, L2 loss)

$$MSE = \frac{\sum^{n} (x_{pred} - y_{true})^2}{n}$$

- Adaptive LR not needed
- Huge gradient far from the minimum



3. **Hubar Loss**

$$Hubar = \begin{cases} 0.5\,(x_{pred} - y_{true})^2 \cdots \text{if } | x_{pred} - y_{true} | < \delta \\ \delta\,| x_{pred} - y_{true} | - 0.5\,\delta^2 \cdots \text{otherwise} \end{cases}$$

Cross-Entropy Loss (entropy, softmax)

- Also called multinomial logistic loss
- Probability of being a class *i* in N categories

$$p_i(y_i) = \frac{\exp(x_i)}{\sum_{j=1}^{N}\exp(x_j)}$$

- Binomial is a special case where probability becomes **sigmoid**
- Entropy loss minimization is **maximum likelihood** method.
- Others: hinge (SVM) loss

- **Cross-entropy loss**

$$\mathcal{L}(x_i, y_i) = -\log\left[\frac{\exp(x_i)}{\sum_{j=1}^{N}\exp(x_j)}\right]$$