Introduction to Deep Learning

Armand Joulin

What is deep learning?


“person”

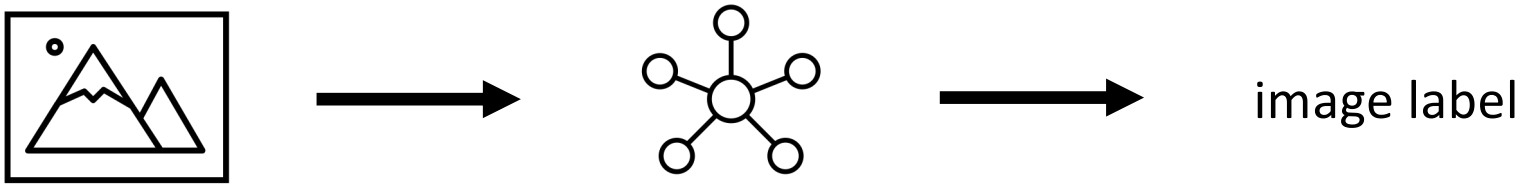
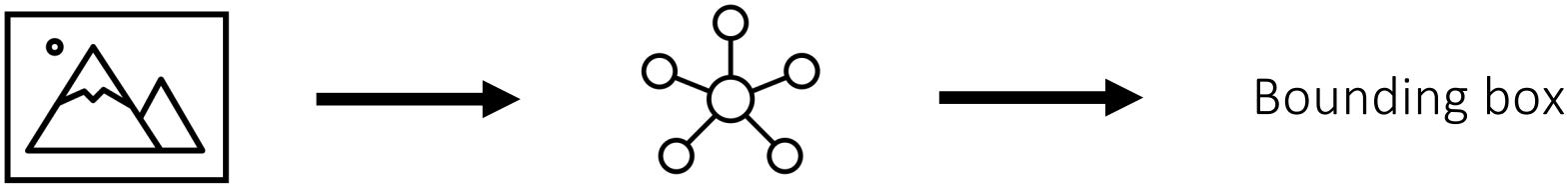
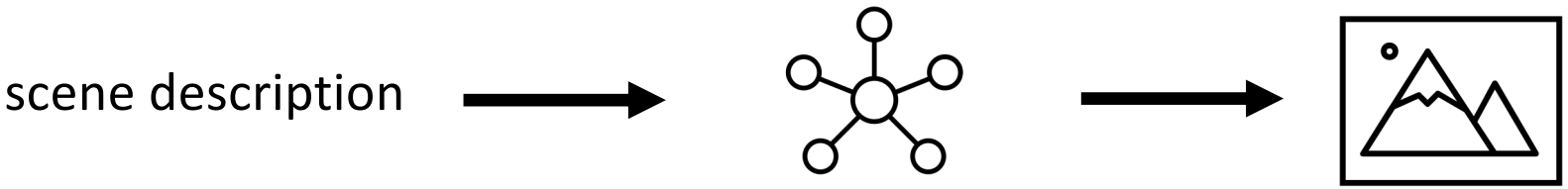
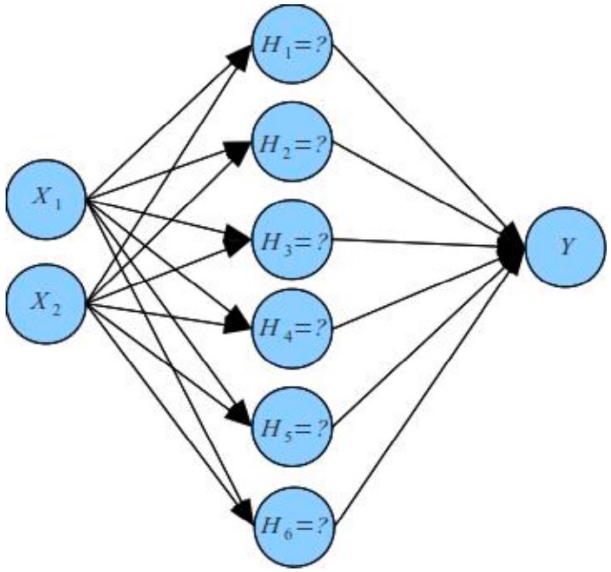


image label


Bounding box

“person running in the street”


scene description



$$f(X) = \sum_j w_j H_j(X) + b = \sum_j w_j g(v_j^\top X) + b$$

Dataset: (X_i, Y_i) pairs, $i = 1, \ldots, N$.

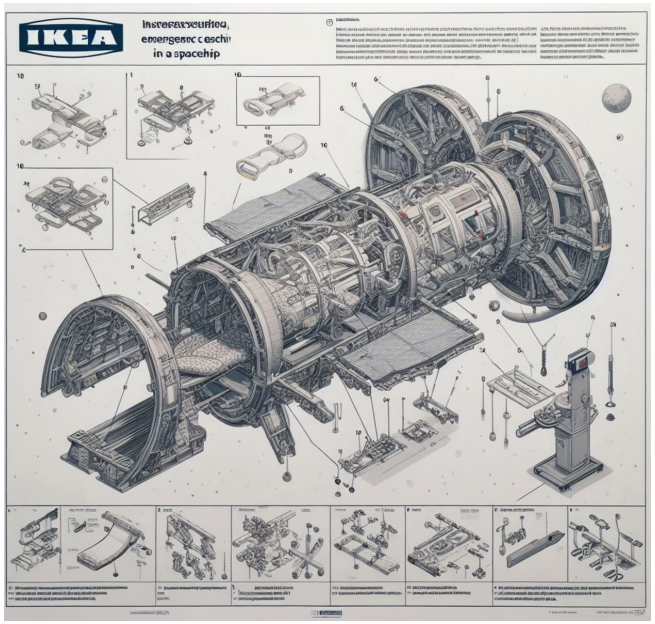
Goal: Find V and W to minimize

$$\sum_i \ell(f(X_i), Y_i) = \sum_i \ell(Wg(VX_i), Y_i)$$

Why do we need Deep Learning?

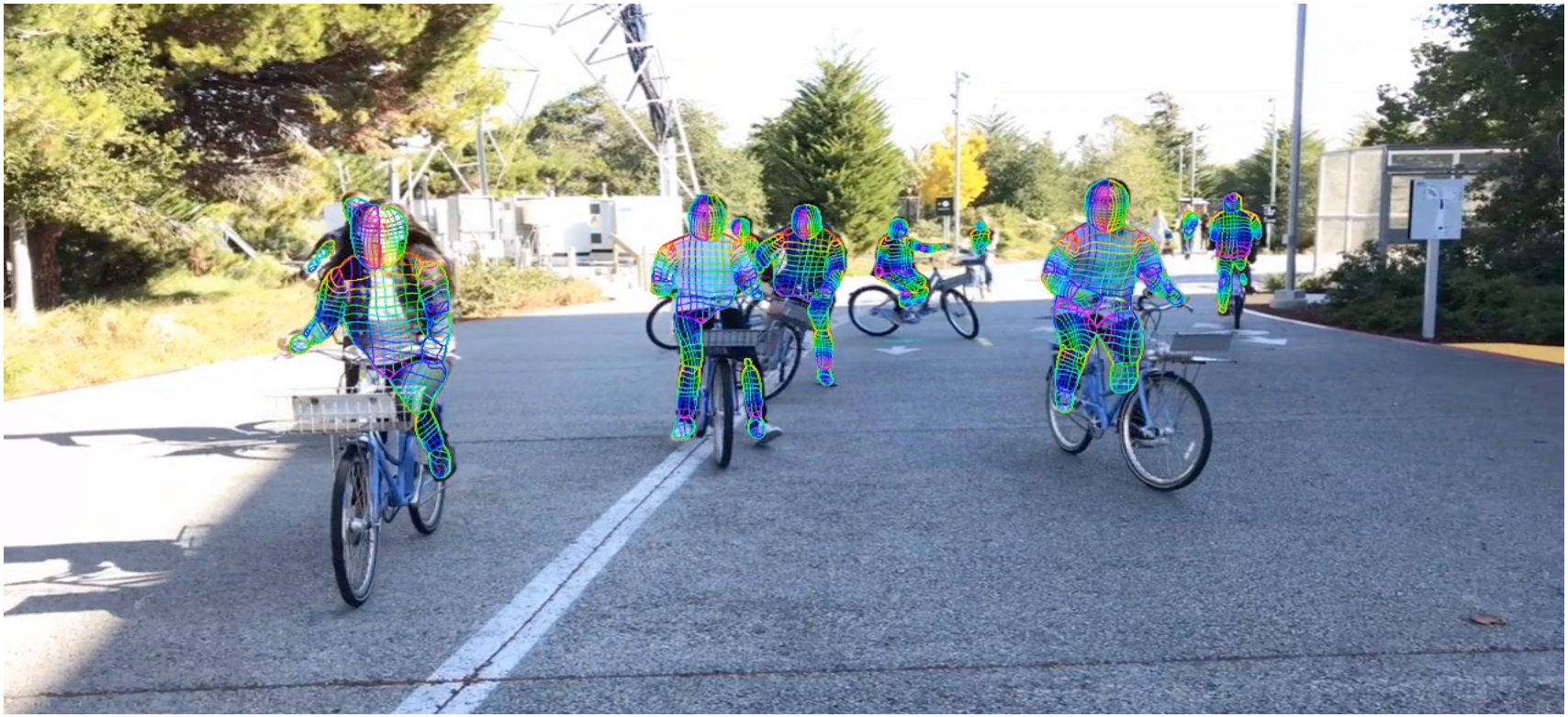
Powering the revolution in knowledge accessibility

“Grandma knitted a portal to hell”



“Ikea instructions for fMRI machine”

…



How does Deep learning work?

Cover in this lecture

- Basic supervised deep learning
- Modeling and training
- Introduction to sequence modeling

Supervised deep learning

Supervised classification

- **Supervision**: Each input X has a fixed given output Y
- **Classification**: Y represents a class label
- **Linear classifier**: learn linear mapping between X and Y

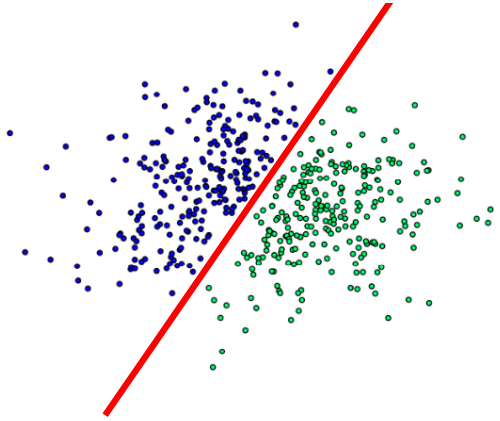
Linear classifier

Dataset:

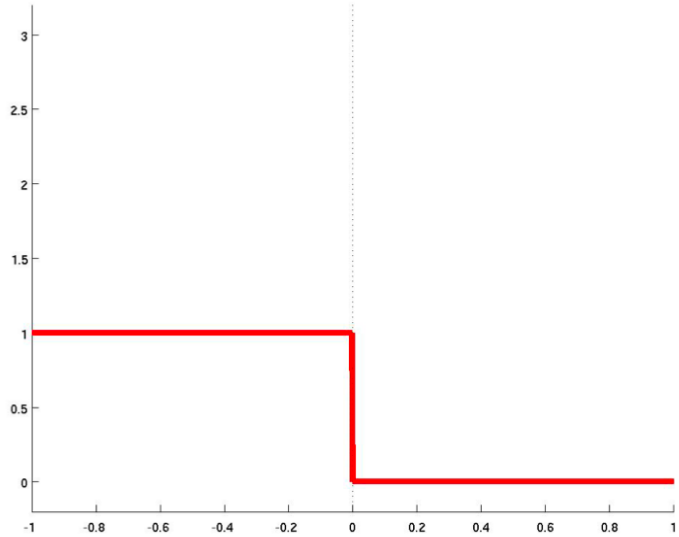
- (X_i, Y_i) pairs, $i = 1, \ldots, N$.
- $X_i \in \mathbb{R}^n$, $Y_i \in \{-1, 1\}$.

Goal:

- Find w and b such that
  $\forall i \in \{1, \ldots, N\}, \ \mathrm{sign}(w^\top X_i + b) = Y_i$.

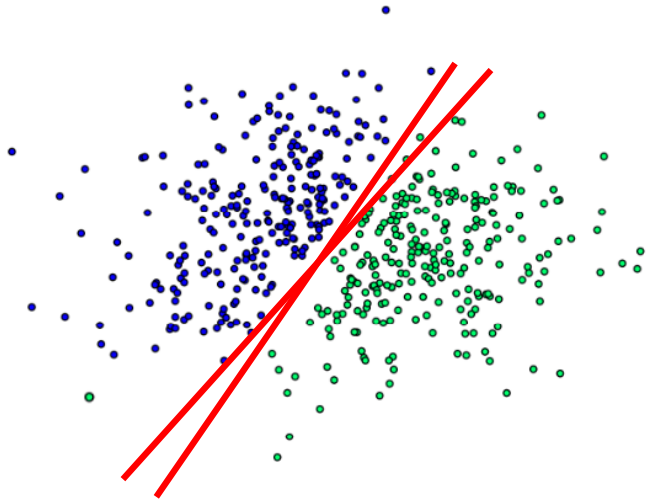


Classification loss

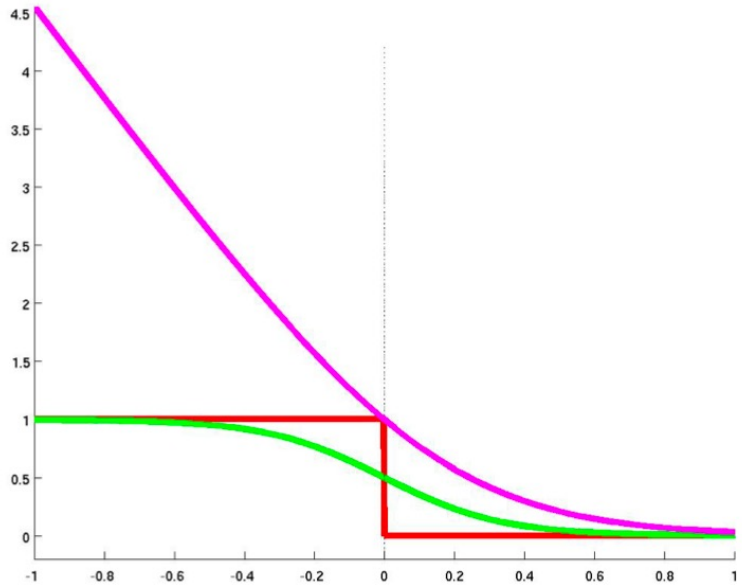


Perceptron algorithm

- $w_0 = 0,\ b_0 = 0$
- $\widehat{Y}_i = \text{sign}(w^\top X_i + b)$
- $w_{t+1} \leftarrow w_t + \sum_i \left(Y_i - \widehat{Y}_i\right) X_i$
- $b_{t+1} \leftarrow b_t + \sum_i \left(Y_i - \widehat{Y}_i\right)$



Doesn’t converge!



P**urple**: logistic regression

Properties: smooth and convex

$$\text{(Perceptron)} \quad \ell(w) = \sum_i 1_{\text{sign}(w^\top X_i + b) \neq Y_i}$$

$$\text{(Logistic regression)} \quad \ell(w) = \sum_i \log \sigma((w^\top X_i + b) Y_i)$$

with $\sigma(x) = \frac{1}{1+\exp -x}$.

Gradient descent

$$L(\theta) = \frac{1}{N} \sum_i \ell(\theta, X_i, Y_i)$$

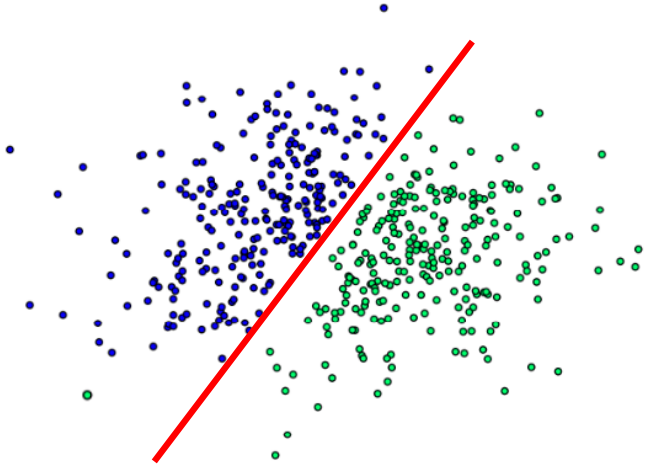
$$\theta_{t+1} \rightarrow \theta_t - \frac{\alpha_t}{N} \sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

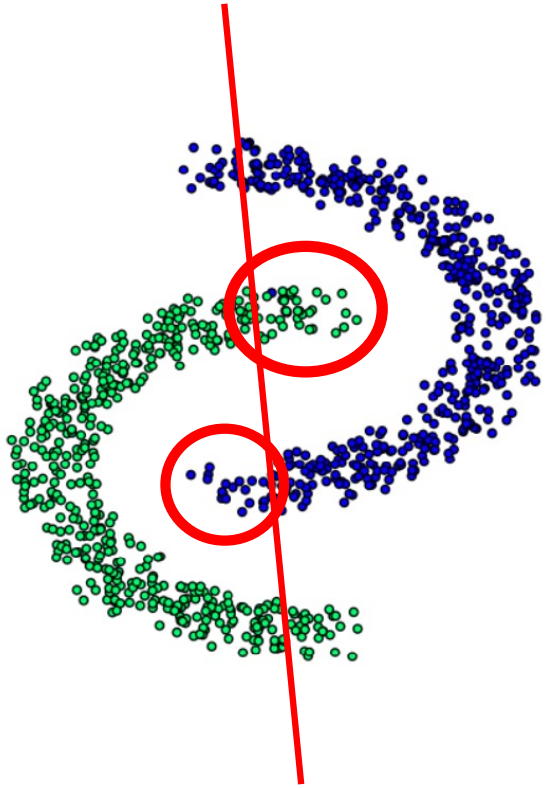
- Take step in direction of gradient to minimize loss
- Guarantee to converge to global minimum if **loss is convex**

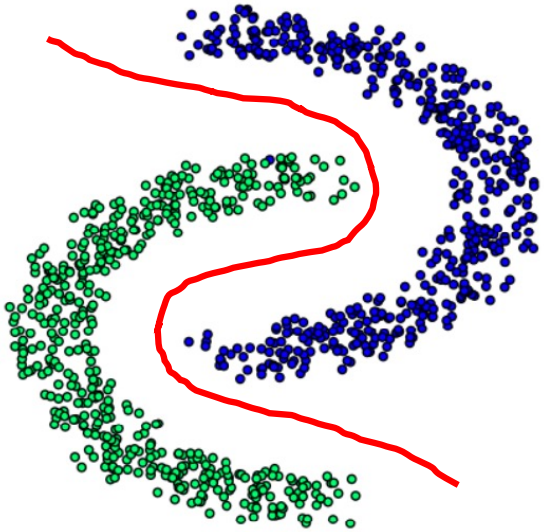
Logistic regression with gradient descent

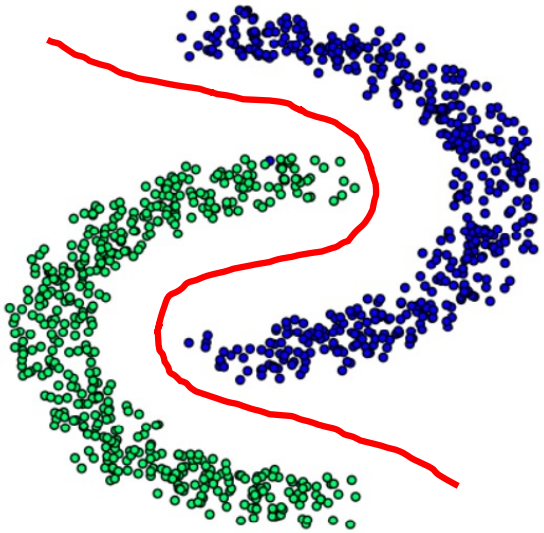
- $w_0 = 0,\ b_0 = 0$
- $\widehat{Y}_n = \sigma(w^\top X_n + b)$
- $w_{t+1} \leftarrow w_t + \frac{\alpha_t}{N} \sum_n \left(Y_n - \widehat{Y}_n\right) X_n$
- $b_{t+1} \leftarrow b_t + \frac{\alpha_t}{N} \sum_n \left(Y_n - \widehat{Y}_n\right)$

where $\alpha_t > 0$ are "step sizes"

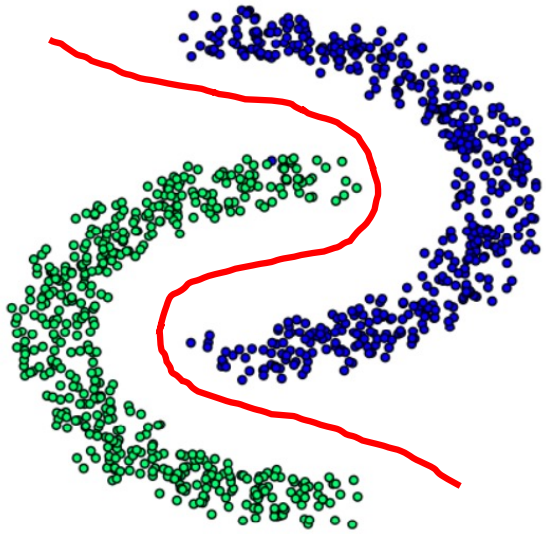




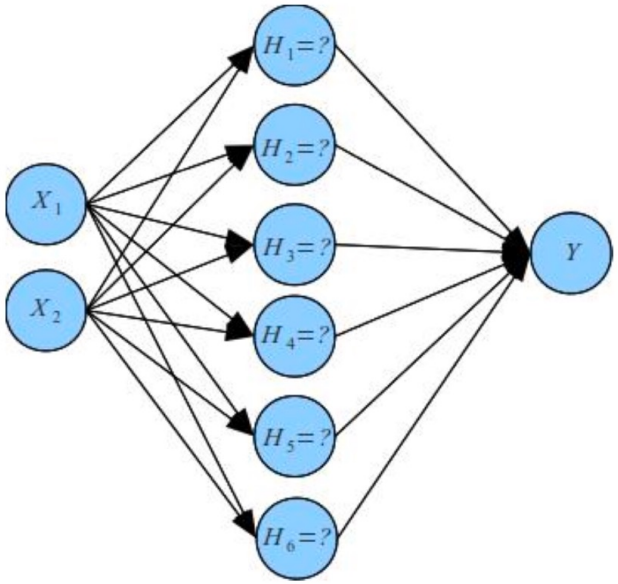




- 2-D example: $X_i = (x_{i1}, x_{i2})$
- Features: $x_{i1}, x_{i2} \rightarrow$ linear classifier
- Features: $x_{i1}, x_{i2}, x_{i1}x_{i2}, x_{i1}^2, \ldots \rightarrow$ non-linear classifier



non-linear classifier = linear classifier of non-linear features.


X_1
X_2
$H_1=?$
$H_2=?$
$H_3=?$
$H_4=?$
$H_5=?$
$H_6=?$
Y

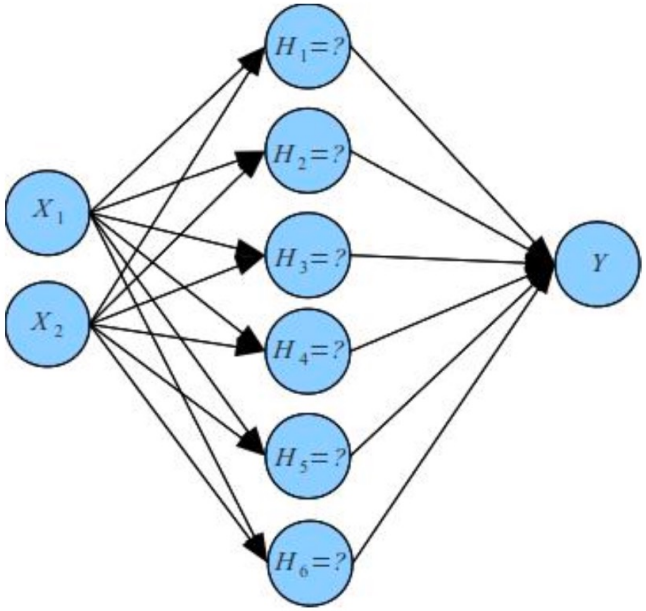
Which non-linear features?

$$H(X) = x_1^p x_2^q$$

$$H_j(X) = \exp -\lambda \|X - X_j\|_2^2 \text{ (rbf kernel)}$$

Why not learn the non linear features too?

That is the goal of **deep learning**

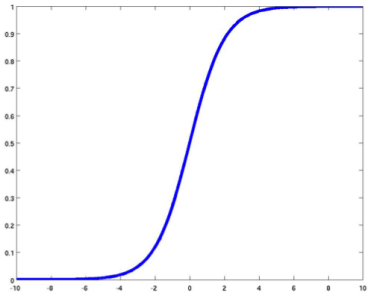


- Usually $H_j = g(v_j^\top X)$

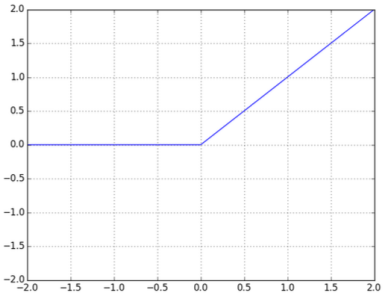
- H_j: Hidden unit
- v_j: Input weight
- g: Transfer function

$$f(X) = \sum_j w_j H_j(X) + b = \sum_j w_j g(v_j^\top X) + b$$

- g is the *transfer function*.

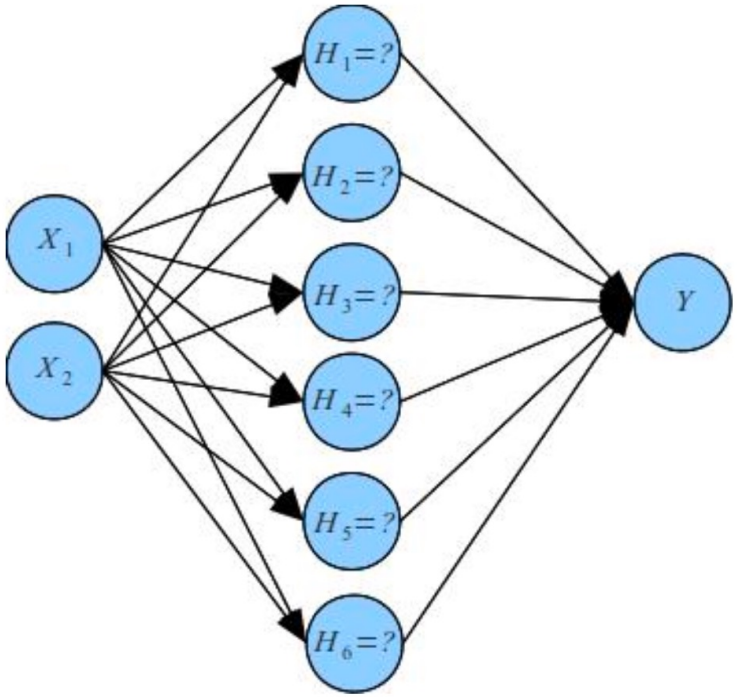


Sigmoid



Rectified linear Unit

- Transformations tend to be non-decreasing and differentiable



$$f(X) = \sum_j w_j H_j(X) + b = \sum_j w_j g(v_j^\top X) + b$$

- Dataset: (X_i, Y_i) pairs, $i = 1, \ldots, N$.
- Goal: Find V and W to minimize

$$\sum_i \ell(f(X_i), Y_i) = \sum_i \ell(Wg(VX_i), Y_i)$$

Training a neural network

Given a (X, Y) pair:

- **Forward pass**: apply network to X to produce an output $\hat{Y}$
- **Evaluation**: Compute loss function, i.e., $\ell(\hat{Y}, Y)$
- **Backward pass**: compute the gradient with backprogation
- **Update:** Take a step in the direction of the gradient

Backpropagation

Backpropagation – chain rule

- Goal: Find W to minimize:

$$\sum_i \ell(Wg(VX_i), Y_i)$$

- We need to compute the gradient of:

$$\ell_i(W, V) = \ell(Wg(VX_i), Y_i)$$

- Chain rule:

$$\begin{aligned}\frac{\partial \ell_i(W,V)}{\partial W} &= \frac{\partial \ell_i(W,V)}{\partial f(X)} \frac{\partial f(X)}{\partial W} \\ &= \frac{\partial \ell_i(W,V)}{\partial f(X)} g(VX)^\top\end{aligned}$$

Backpropagation – chain rule

- Goal: Find V to minimize:

$$\sum_i \ell(Wg(VX_i), Y_i)$$

- We will rewrite $Wg(VX) = WH$ with $H = g(VX)$.
- Chain rule:

$$\begin{aligned}\frac{\partial \ell_i(W,V)}{\partial V} &= \frac{\partial \ell_i(W,V)}{\partial f(X)}\frac{\partial f(X)}{\partial V} \\ &= \frac{\partial \ell_i(W,V)}{\partial f(X)}\frac{\partial f(X)}{\partial H}\frac{\partial H}{\partial V} \\ &= W\frac{\partial \ell_i(W,V)}{\partial f(X)}g'(VX)X^\top\end{aligned}$$

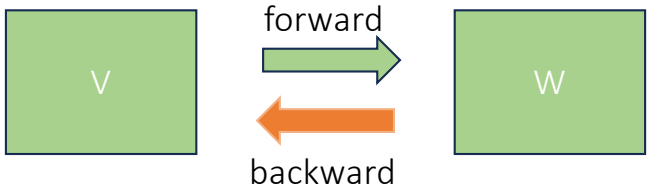
Backpropagation - memoization

gradient in V shares some elements with gradient in W:

$$\frac{\partial \ell_i(W,V)}{\partial W} = \frac{\partial \ell_i(W,V)}{\partial f(X)} g(VX)^\top,$$
$$\frac{\partial \ell_i(W,V)}{\partial V} = W \frac{\partial \ell_i(W,V)}{\partial f(X)} g'(VX) X^\top$$

Same computation can be re-used

Backpropagation - memoization



Computing gradient from end to beginning → re-use partial computation

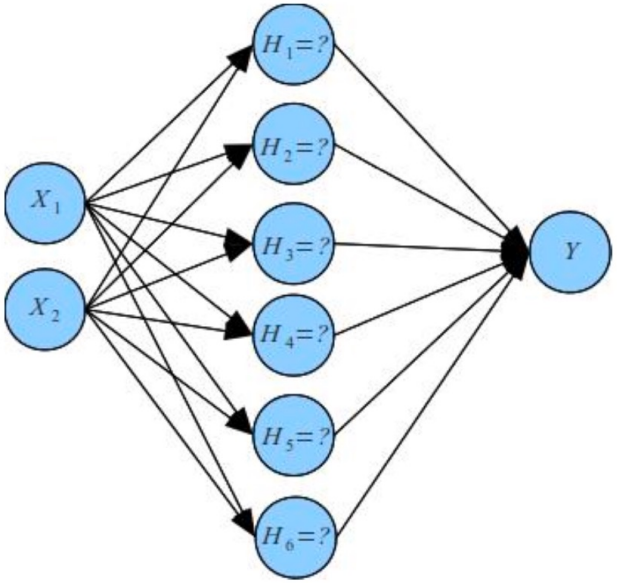
Optimal computation of gradient at the cost of memory

Can be generalized and automated along a DAG (autograd)

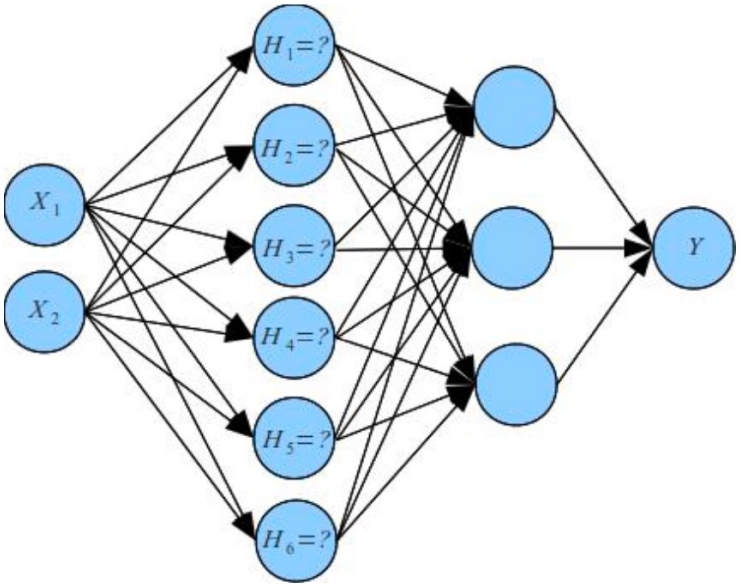
Backpropagation = chain rule + memoization

Impact of deeper network on gradients

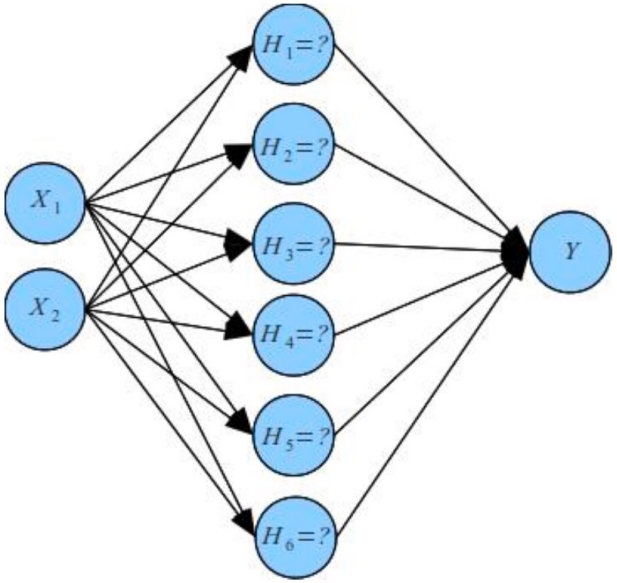
Going deeper



Going deeper



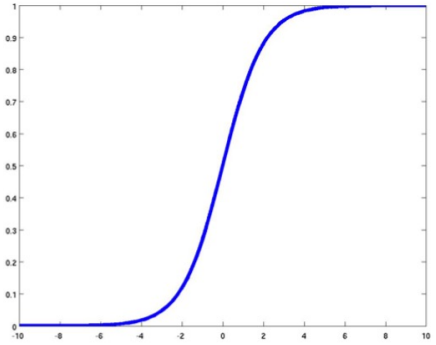
Going deeper – vanishing gradient problem



Norm of gradient flowing from output to input

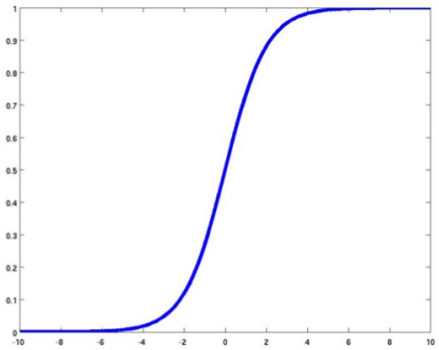
Why?

- Non linearity puts gradient to 0
- Matrix multipliciation with eigenvalues < 1

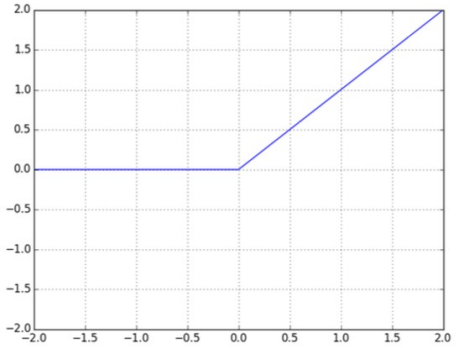


Sigmoid

Going deeper – vanishing gradient problem



Sigmoid



Rectified linear Unit

Use non-linearity with less zero-region

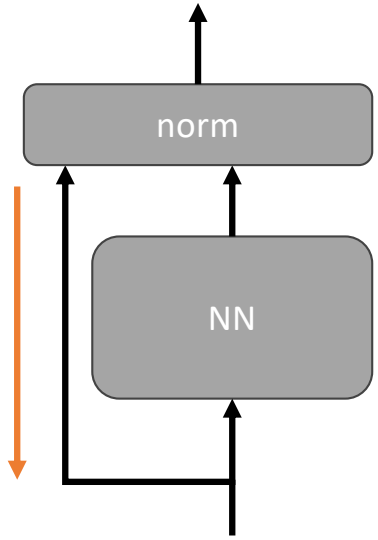
Going deeper – vanishing gradient problem

Skip connection+normalization:

- Given a network block **nn** and input **x**
- The output **y** is computed as

$$\mathbf{y} = \mathbf{norm}(\mathbf{x} + \mathbf{nn}(\mathbf{x}))$$

where **norm** normalize the input



Optimization

Gradient descent

$$L(\theta) = \frac{1}{N} \sum_i \ell(\theta, X_i, Y_i)$$

$$\theta_{t+1} \rightarrow \theta_t - \frac{\alpha_t}{N} \sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

- Require pass over entire dataset
- Dataset contains millions/billions examples

Stochastic gradient descent (SGD)

$$\text{(Gradient descent)} \qquad \theta_{t+1} \to \theta_t - \frac{\alpha_t}{N} \sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

$$\text{(Stochastic gradient descent)} \quad \theta_{t+1} \to \theta_t - \alpha_t \frac{\partial \ell(\theta, X^{(i_t)}, Y^{(i_t)})}{\partial \theta}$$

Stochastic gradient descent (SGD)

$$\text{(Gradient descent)} \qquad \theta_{t+1} \to \theta_t - \frac{\alpha_t}{N} \sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

$$\text{(Stochastic gradient descent)} \quad \theta_{t+1} \to \theta_t - \alpha_t \frac{\partial \ell(\theta, X^{(i_t)}, Y^{(i_t)})}{\partial \theta}$$

What are the pros and cons?

Stochastic gradient descent (SGD)

i.i.d. sampling + **stochastic gradients = full gradient in expectancy** (no bias)

- pros:
  - For one full gradient, time to do N updates with SGD -> **N times faster**
  - works on **infinite data** or online
- cons:
  - introduces variance in gradients

Batch SGD

- Pick K random points instead of picking 1 (with $K << N$):

$$\theta_{t+1} \rightarrow \theta_t - \frac{\alpha_t}{K} \sum_{i=1}^{K} \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

- K offers trade-off between variance but speed

$$L(\theta) = \frac{1}{N} \sum_i \ell(\theta, X^{(i)}, Y^{(i)})$$

$$\theta_{t+1} \to \theta_t - \alpha_t \frac{\partial \ell(\theta, X^{(i_t)}, Y^{(i_t)})}{\partial \theta}$$

- How to initialize the parameters?
- How to set learning rates α_t?
- Can we do better than plain gradient descent?

Weight regularization

Why is it important?

- Many networks produce same results
- Examples: permutations of weights, invariant to multiplication…
- We can reduce space of exploration to smaller set of networks

→ Faster convergence

Weight regularization

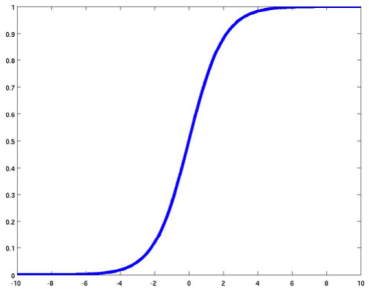
3 complementary approaches:

- Initialziation
- Normalization of activations
- Regularizing the weights

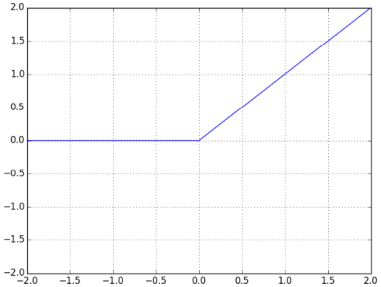
Initialization

- If two units are equal, they stay equal
- Waste of capacity
- Random initialization breaks symmetry

Initialization



Sigmoid



Rectified linear Unit

- Many nonlinearities have regions with 0 norm gradients
- Initialization must avoid saturated areas
- Alernatively use nonlinearities with no saturation:

$$\text{Leaky ReLU} = \text{ReLU}(x) + \alpha x, \text{ with } \alpha > 0.$$

Fan-in initialization

- **Fan-in**: number of inputs used to compute a hidden units
- Large fan-in implies larger changes in hidden variables
- Need smaller initialization
- Typically, weights $\approx 1/\sqrt{\text{fan-in}}$

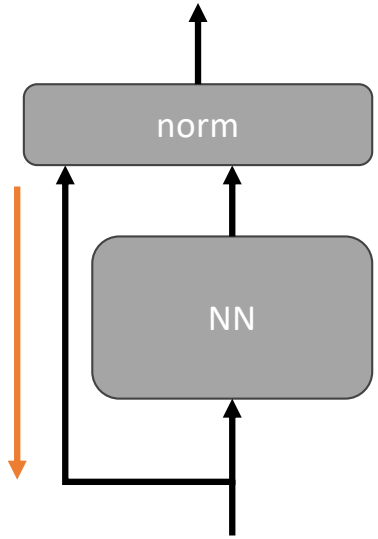
Data normalization (whitening)

- Update of a layer is proportional to its input
- Example:
  - Assume $X_1 = 100$ and $X_2 = 101$
  - $\nabla \ell_1 = +1$ and $\nabla \ell_2 = -1$
  - Mean of updates is small ($\propto -0.5$) but each update is huge ($\propto 100$)
- Center data is important!
- Centering is transforming x_i into $\frac{x_i - \mu_i}{\sigma_i}$

Intermediate normalization

Normalize intermediate features to keep values in range of non-linearities

- Different solutions:
  - Batch normalization
  - Layer normalization
  - RSMnorm
  - …



Example: batch normalization

- Centering is transforming x_i into $\frac{x_i - \mu_i}{\sigma_i}$
- For the upper layers, μ_i and σ_i change over time
- We shall learn them and update the parameters accordingly

Example: batch normalization

$$o_i = BN_{\alpha,\beta}(h_i)$$

$$\mu_B \leftarrow \frac{1}{b}\sum_{i=1}^{b} h_i$$

$$\sigma_B^2 \leftarrow \frac{1}{b}\sum_{i=1}^{b}(h_i - \mu_B)^2$$

Compute batch statistics

$$\hat{h}_i \leftarrow \frac{h_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

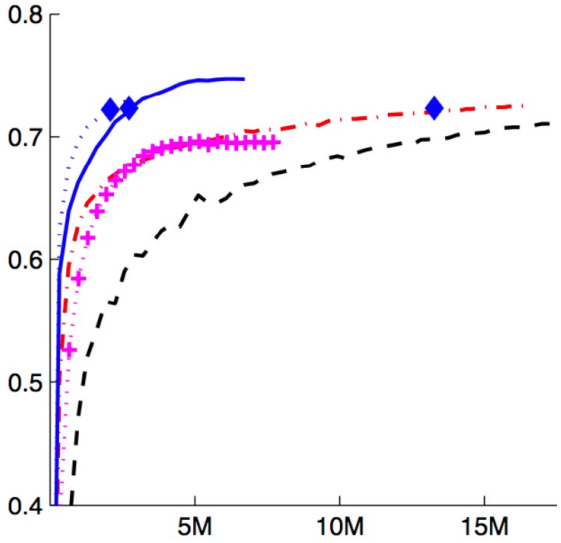
Normalize hidden state h_i

$$o_i \leftarrow \alpha \hat{h}_i + \beta$$

Shift the normalized hidden

α and β are learned over time.

- Normalization reduces space of parameters to explore
- faster convergence
- Layer norm is prefered over batch norm



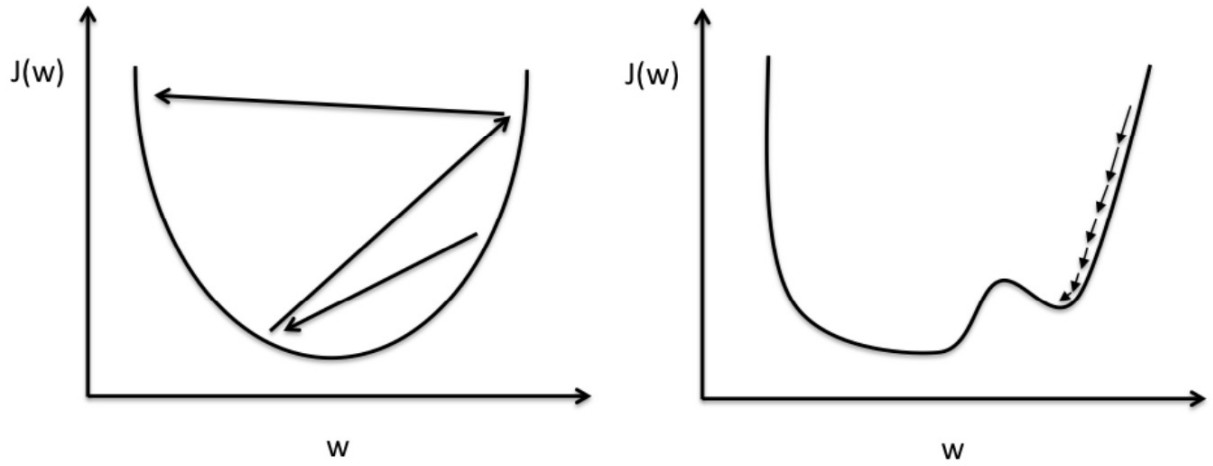
Weight decay

- Apply a L_2 regularization on the parameters
- In our simple neural network, this is equivalent to:

$$\sum_i \ell(Wg(VX_i), Y_i) + \mu_t \|V\|_2^2 + \mu_t \|W\|_2^2$$

- $\mu_t > 0$ decreases during training with the learning rate
- Different from standard regularizartion where $\mu > 0$ is fixed.

Setting learning rate


J(w)
w
J(w)
w

Solution 0: fixed learning rate

- Start with a large stepsize
- If you diverge or oscillate, reduce it
- If progress is slow but consistent, increase it
- Then keep it constant

Solution 1: linear decay

- Linear decay: $\alpha_t = a/(b+t)$
- Divide learning rate by a factor when loss on validation set does not decrease
- Fix number of iterations T and set learning rate accordingly: $\alpha_t = \alpha_0(T-t)/T$

Solution 2: cosine scheduler

- Same as linear linear decay with a cosine function: $\alpha_t = \alpha_0 \cos(t/Tpi/2)$
- Last learning rate often equals to 0.1 of initial value

Beyond vanilla SGD

Not all direction are equal

Not all direction are equal

We want to go fast in some directions, slow in others

Vectorized SGD: one step size per dimension

Scalar stepsize:

$$\theta_{t+1,i} = \theta_{t,i} - \alpha_t \boldsymbol{g}_{t,i}$$

Vector stepsize:

$$\theta_{t+1,i} = \theta_{t,i} - \alpha_{t,i} \boldsymbol{g}_{t,i}$$

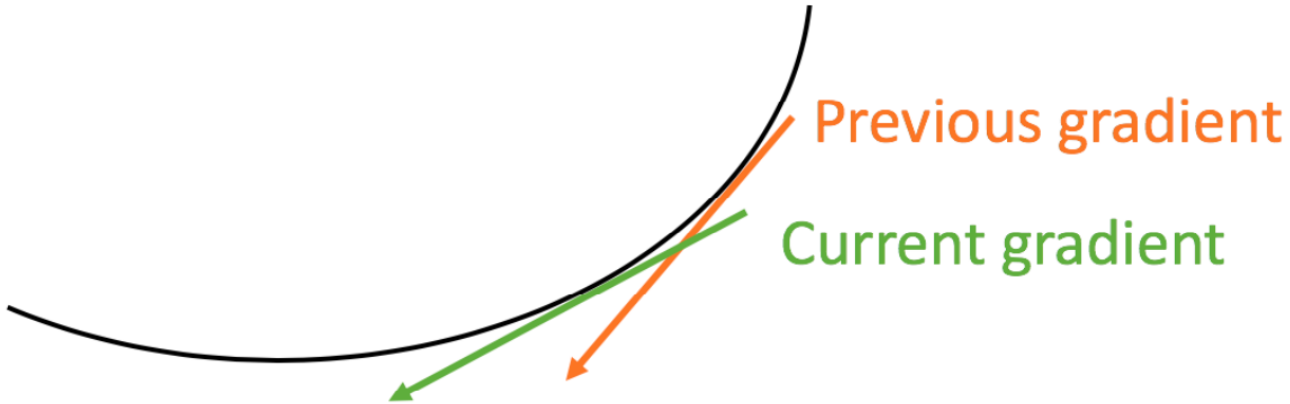
Example: Adagrad

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

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \varepsilon}} g_{t,i}$$

- No need to set a learning rate schedule
- $G_{t,i}$ is the accumulation of the squared gradients
- Squared norm avoids exploding or vanishing gradient
- ε avoids numerical issues.

Use previous gradients



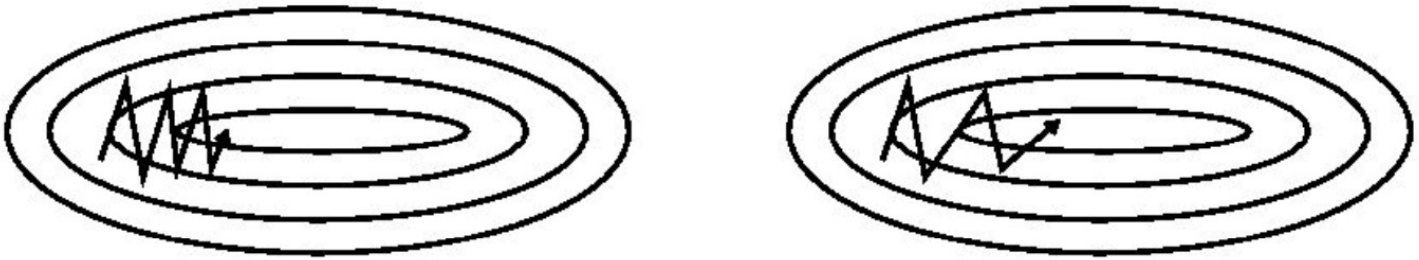
Previous gradients are not bad estimates of current curvature

Example: momentum (or heavy ball)

$$M_t = \gamma M_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - M_t$$

- γ controls the inertia
- M_t is almost a moving average
- Typically, $\gamma = 0.9$ or 0.99

Example: momentum (or heavy ball)



Momentum + vectorized stepsize = ADAM

$$M_{t,i} = \frac{1}{1-\beta^t}(\beta M_{t-1,i} + (1-\beta) g_{t,i})$$

$$G_{t,i} = \frac{1}{1-\gamma^t}(\gamma G_{t-1,i} + (1-\gamma) g_{t,i}^2)$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \varepsilon}} M_{t,i}$$

- $M_{t,i}$ = moving average of gradients, as in momentum.
- $G_{t,i}$ = moving average of squared gradients, as in Adagrad.
- ε avoids numerical issues

Avoiding gradient explosion

Why it exists?

- Multiplying matrices with eigenvalues > 1
- Numerical unstability when dealing with large number of params

Gradient clipping

Solution is to clip the value of gradient below some norm:

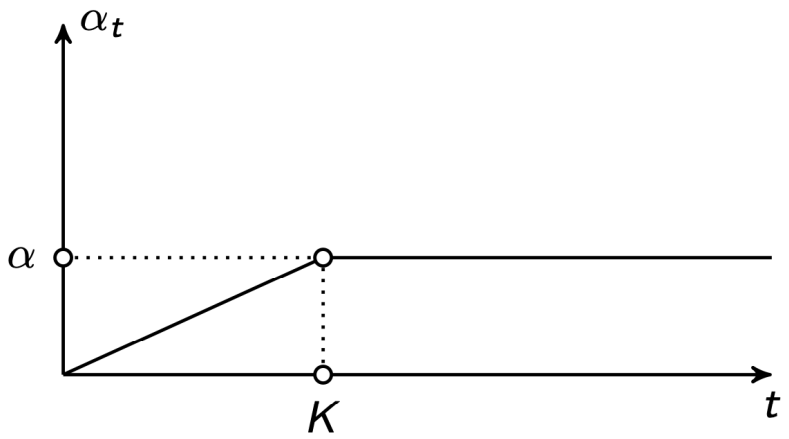
$$G = \min(\mu, \|G\|)\frac{G}{\|G\|}$$

with $\mu > 0$

Warm-up

- Most gradient explosition happens at the beginning of training
- Because matrices are poorly set and learning rates are large
- Solution: start with small learning and increases it

Warm-up



Learning rate scheduler $(\alpha_t)_t$

- Set a target learning rate α

$$\alpha_t = \min(1, \frac{t}{K})\alpha$$

where K is the "warm-up" parameter

Summary of optimization

Standard optimization:

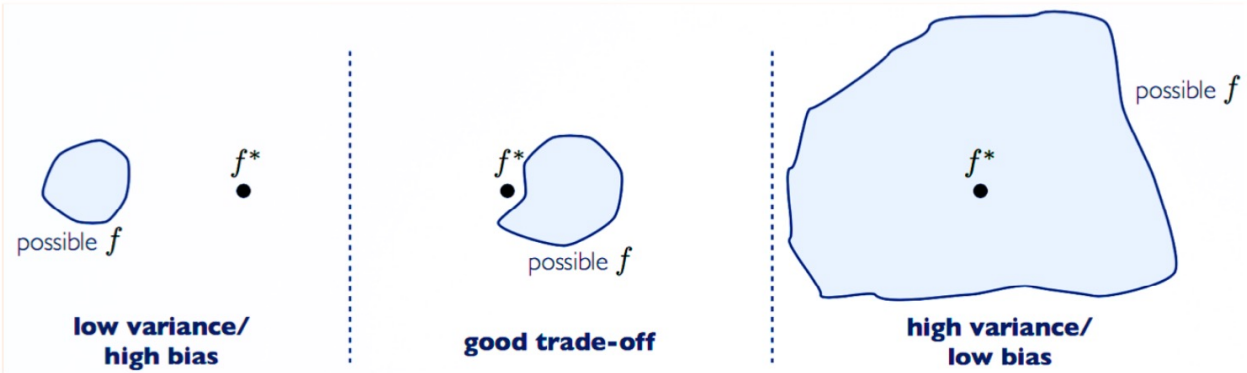
- ADAM (or AdamW)
- clipping
- cosine scheduler
- Warm-up
- Init based on fan-in
- greed search over initial learning rate and weight decay

Underfitting and overfitting

What is it?

- Underfitting:
  - not enough parameters to express complexity in data
  - low performance on training and test set

- Overfitting:
  - too many parameters match too well complexity in training data
  - high performance on training set, low on test set



- Complexity of model increases with number of layers
- Easier to overfit on data

True in the ``low'' data regime

- Problem is that training set is small (few millions data)
- Easy to memorize training set
- No generalization

Not true in the ``infinite'' data regime

- Overfitting on infinite data is **good**, most models underfit
- In this setting, there is no more ``test'' set
- Example: large language models are in the ``infinite'' data regime

What to do in this regime?

- Find **scaling rules** of parameters versus data
- Estimate numbers of parameters when scaling in data

Switching to sequence modeling…