Lecture 25

Neural Networks:

$$h(\vec{x}) = \phi(x)^T w + b$$

*** Here, we learn the parameters $\vec{w}, b$ and $\phi(x)$

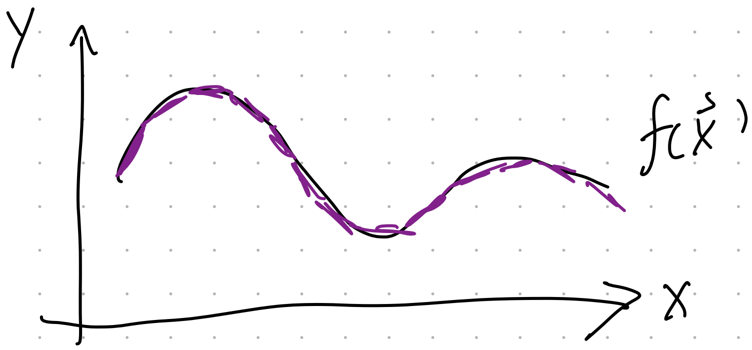
representation of $\vec{x}$ in higher space.

$$x \to \phi(x)$$

$$\phi(x) = \underbrace{\sigma}_{\text{nonlinear transition function}}(\overbrace{U\vec{x}}^{\text{linear transformation}})$$

ReLU: $\sigma(z) = \max(z, 0)$

Universal approximation theorem: Any function $f(\vec{x})$ that maps $\vec{x}$ to output $\vec{y}$ can be learned by Neural Networks.



Layers in Neural Networks

Single layer: $\phi(\vec{x}) = \sigma(U\vec{x})$

Multiple layers: ("Deep" layers)

matrix multiplication

$$\phi(\vec{x}) = \sigma(\underbrace{U\,\phi'(\vec{x})})$$

$$\phi'(\vec{x}) = \sigma(\underbrace{U'\,\phi''(\vec{x})})$$

$$\phi(\vec{x}) = \sigma(\underbrace{U''\,\phi'''(\vec{x})})$$

$$\vdots$$

Equivalence between single-layer neural networks and multiple-layers neural networks: Any function can be learns by both.

$$U = \underbrace{\begin{bmatrix} & & \\ & & \end{bmatrix}}_{d} \Big\} m \Rightarrow \phi(x) = \begin{bmatrix} \\ \\ \end{bmatrix} \Big\} m$$

- But it turns out that, for single layer, the matrix U has to be very large.
- The multiple layers benefits from exponential effects from multiplying the number of possibities resulted by previous layers.

How Nearal Networks learn?

- Again, let ℓ be any well-defined loss function

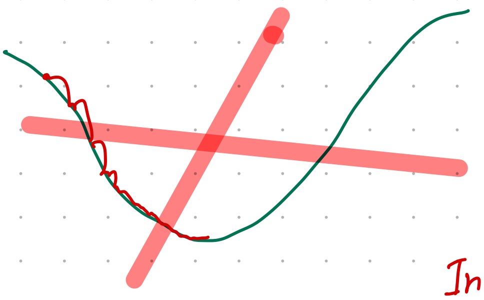
e.g $\ell(h) = \frac{1}{2} \sum_{i=1}^{n} (h(\vec{x}_i) - y_i)^2$

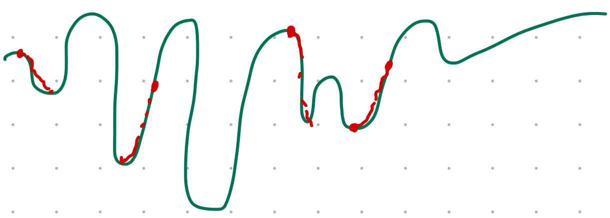
The classifier h has parameters $\vec{w}, U$ ($\vec{w}, U, c, b$ for full)

- Let's use simple Gradient Descent (single layer neural network)

Repeat
$$\left(\begin{aligned} \vec{w}_{t+1} &= \vec{w}_t - \alpha \frac{\partial \ell}{\partial w} \\ U_{t+1} &= U_t - \alpha \frac{\partial \ell}{\partial U} \end{aligned} \right.$$

Bad news: Our optimization is no longer convex like before because non-linear transition function





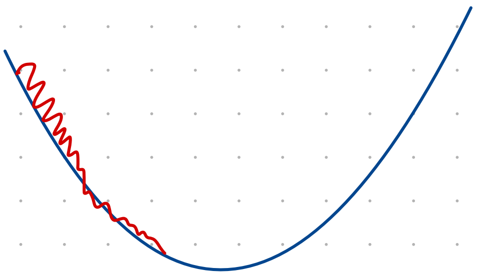
In fact, the optimation function is highly non-convex, meaning there are many local minima.

Leaning with multiple (three) layers

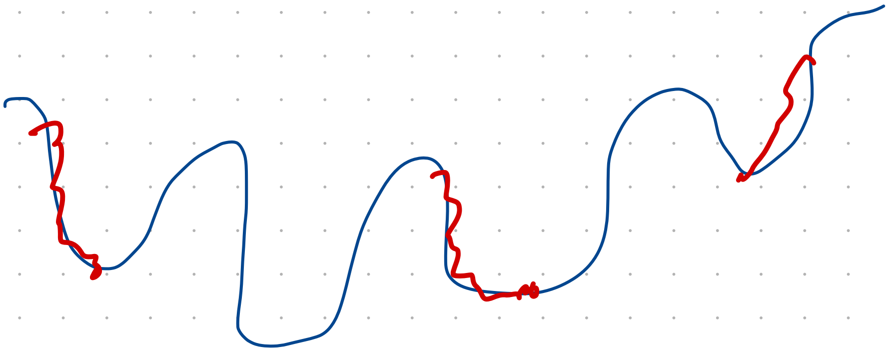
- $\ell(h) = \sum_{i=1}^{n} \ell(h(\vec{x}_i), y_i) = \frac{1}{2}(h(\vec{x}_i) - y_i)^2$
- $h(\vec{x}) = \vec{w}^T \phi(\vec{x}) \longrightarrow \frac{\partial \ell}{\partial w} = \sum_{i=1}^{n} (\vec{w}^T \phi(x) - y_i)\, \phi(x_i)$
- $\phi(x) = \sigma(\underbrace{U \phi'(x)}_{a(x)}) \longrightarrow \frac{\partial \ell}{\partial U} = \frac{\partial \ell}{\partial a} \frac{\partial a}{\partial u}$

  $a(x) = U \phi(x)$

  $\frac{\partial a}{\partial u} = \phi(x)$
- $\phi'(x) = \sigma(\underbrace{U' \phi''(x)}_{a'(x)}) \longrightarrow \frac{\partial \ell}{\partial U'} = \frac{\partial \ell}{\partial a} \frac{\partial a}{\partial a'} \frac{\partial a'}{\partial u'}$
- $\phi''(x) = \sigma(\underbrace{U'' x}_{a''(x)}) \longrightarrow \frac{\partial \ell}{\partial U''} = \frac{\partial \ell}{\partial a} \frac{\partial a}{\partial a'} \frac{\partial a''}{\partial u''}$

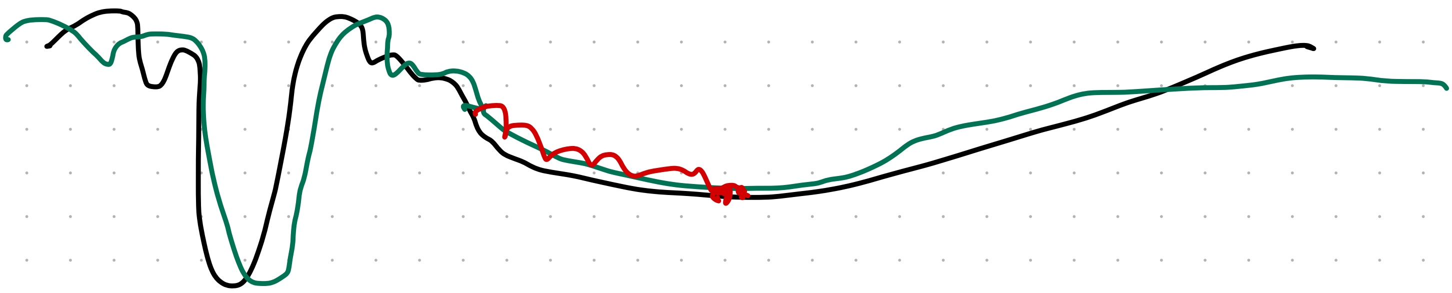


convex



nonconvex

- ℓ is convex with repect to U, U', U'' because of σ



Stochastic Gradient Descent (SGD):

- A variation Gradient Descent algorithm for optimization.

- In the original Gradient Descent, we take

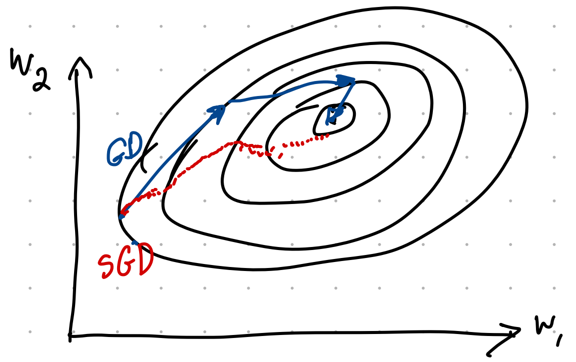
<u>Gradient</u>: $\nabla \ell = \sum_{i=1}^{n} \frac{\partial \ell(h(x_i), y_i)}{\partial w_i}$

e.g. $\ell(\vec{w}) = \frac{1}{2} \sum_{i=1}^{n} (h(x_i) - y_i)^2 \Rightarrow \nabla \ell = \sum_{i=1}^{n} (w^T \phi(x) - y_i) \phi'(x)$

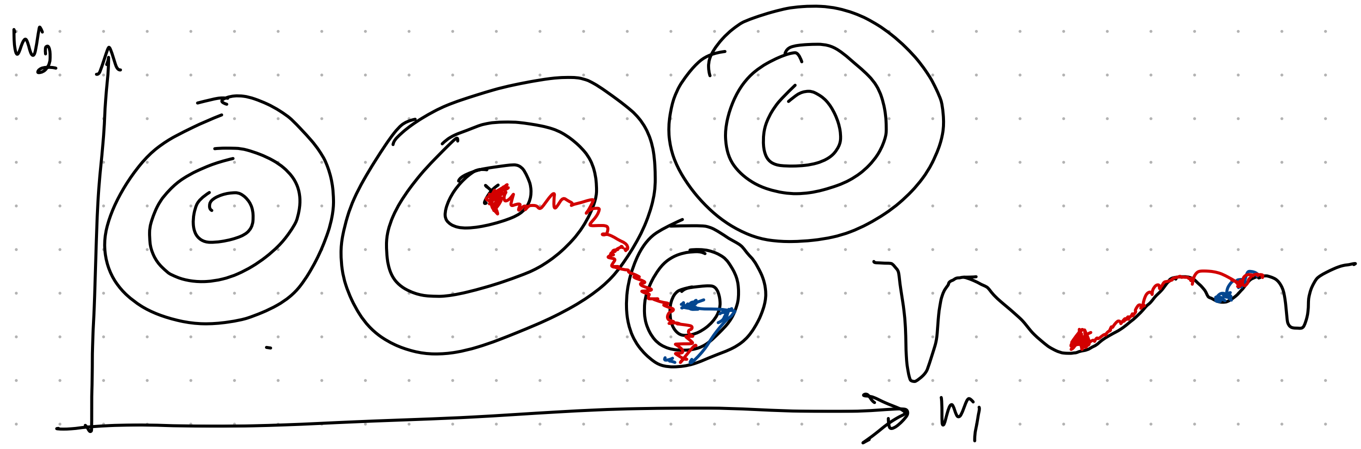
- SGD approximate the gradient with only 1 (or $m \leq n$) sample

$\nabla \ell \approx \frac{\partial \ell(h(x_i), y_i)}{\partial w}$ ← only single x_i

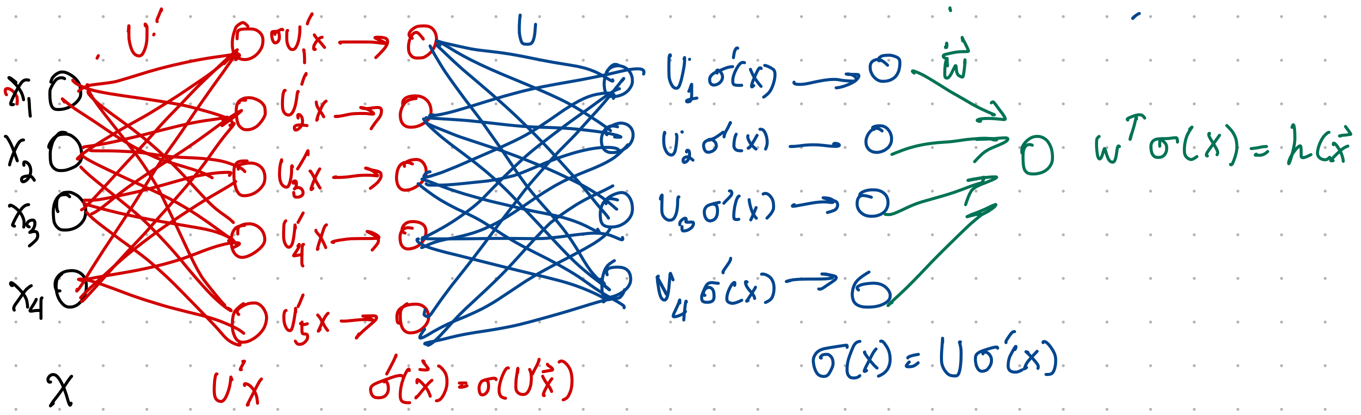
This means we have one tiny update for each sample



- SGD behaves very noisy. It will mostly never land at the local minima and saddle points where precise optimization methods would land to.



Normal Picture of Deep Learning (Graph Representation)



- The picture makes you feel like the network is the brain learning something, but it is definitely not!!!

Compute the Prediction (Forward Propogation):

$$z_0 = \vec{x}$$

For $d = 1 : \ell$

$$a_d = U^d z_{d-1}$$

$$z_d = \sigma_d(a_d)$$

End

Return z_d

Gradient update (Backword Propogation)

$$\vec{\delta}_d = \frac{\delta \ell}{\delta z_d} \odot \sigma_d'(a_d)$$

for $d = \ell : -1 : 1$

$$U_d = U_d - \alpha \vec{\delta}_d z_{\ell-1}^T$$

$$\vec{\delta}_{d-1} = \sigma_{d-1}'(a_{d-1}) \odot (W_d^T \delta_d)$$

End.

Hadamard product: compute elementwise product

$$\begin{pmatrix} a \\ b \end{pmatrix} \odot \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \cdot c \\ b \cdot d \end{pmatrix}$$

σ_d' is the gradient of σ_d