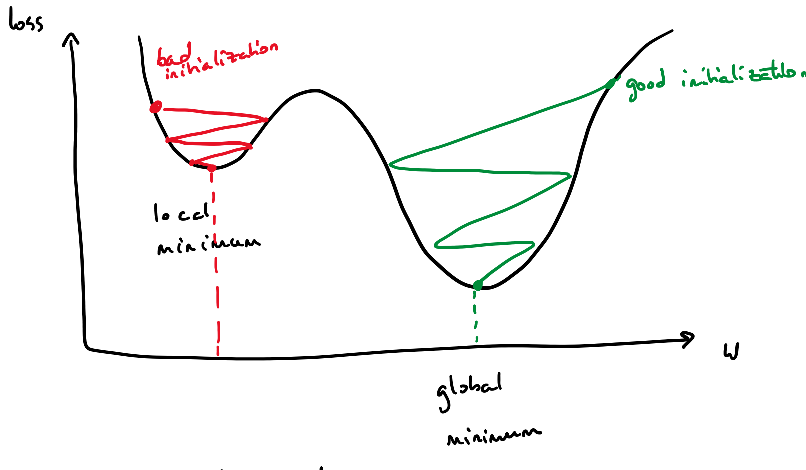
Video 23

Monday, November 2, 2020 5:13 PM



Avoid local minima : <u>basins of attraction.</u>

<u>Why do these local minima exist in deep neural network training?</u>

If training is convex, there is no <u>local</u> minima
(includes linear regression, logistic regression)

Neural network training is not convex.

- observe that DNNs have weight space symmetry
  - ↳ permute all hidden neurons on a layer
  - ↳ we obtain the same model
- if we average all permutations of the parameters, we would get a 'degenerate' neural network where all hidden neurons are identical
- if the cost function for training the neural network was convex, the average network (defined above) should perform <u>better</u> than the original network

because when f is convex

$$f(\lambda_1 x_1 + \dots + \lambda_N x_N) \leq \lambda_1 f(x_1) + \dots + \lambda_N f(x_N) \quad \text{for } \lambda_i \geq 0, \ \sum_i \lambda_i = 1$$

f: loss function; $f(\lambda_1 x_1 + \dots + \lambda_N x_N)$: degenerate neural network

$\lambda_i = \frac{1}{N}$

$\vec{x}_i$ = model parameters for <u>one</u> of the permutated architectures

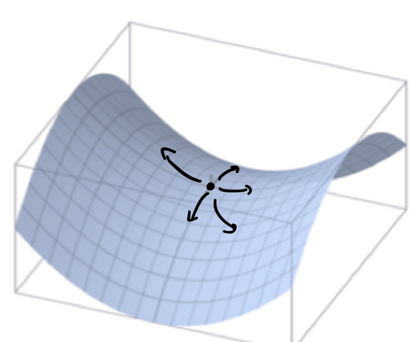
$f(\vec{x}_i)$ is the loss of one of the permutated architectures

$$f(\vec{x}_1) = f(\vec{x}_2) \dots = f(\vec{x}_N)$$

$$\sum_i \lambda_i f(\vec{x}_i) = f(\vec{x}_i)$$

$f(\vec{x}_i)$: loss of our original model

<u>Saddle points</u>



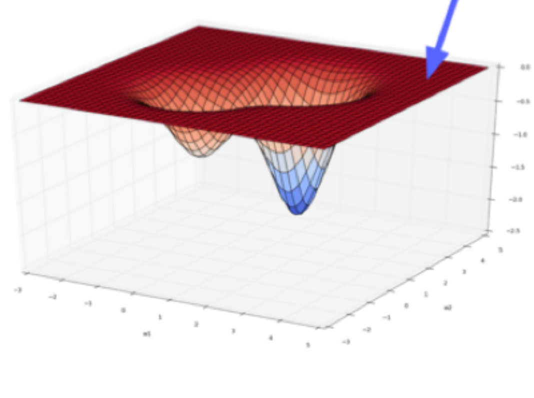
At a saddle point, we have $\frac{\partial \mathcal{L}}{\partial \vec{w}} = 0$ even though we are <u>NOT</u> at a minimum.

Exactly at the saddle point : gradient descent is stuck

For a slight <u>perturbation</u> of the saddle point : gradient descent gets unstuck because we no longer have $\frac{\partial \mathcal{L}}{\partial w} = 0$

When initializing the random weights of our model, we should avoid using weights that are initially set to 0 and instead we should prefer <u>random small values</u>.

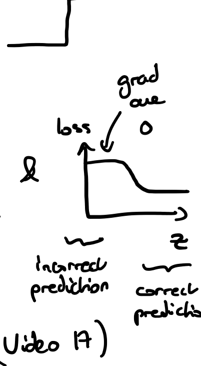
<u>Plateaux</u>

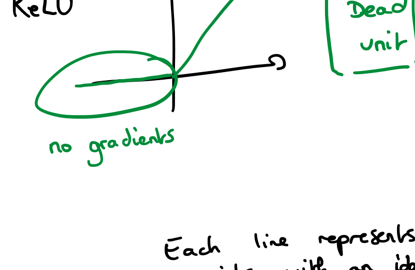


Flat region in the loss surface

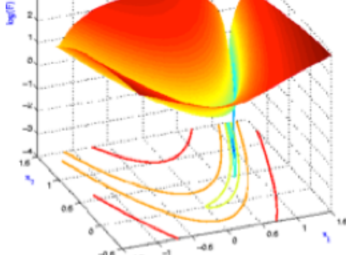
Examples:

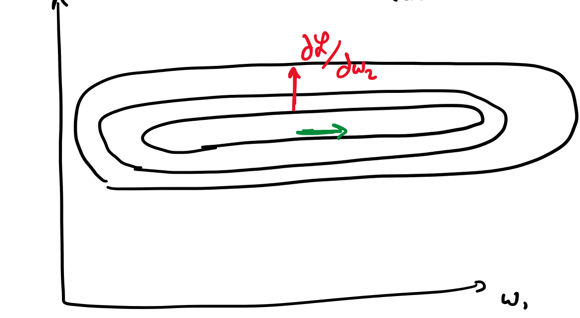
- → 0-1 loss
- → hard threshold activation
- → logistic activations & least squares.
- → saturated activation function (Video 17)





<u>Ravines</u>





To avoid ravines : Ⓐ make sure inputs are centered & have unit variance

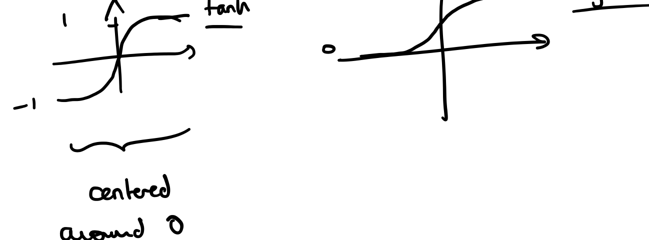
$$x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}$$

Especially important when features have ≠ arbitrary units

Ⓑ Hidden units / neurons should also have centered activations.

Harder to achieve :

→ use tanh activation instead of logistic activation



→ batch normalization : centers each hidden activation and can speed up training

Ⓒ Momentum

without momentum : $\vec{w} \leftarrow \vec{w} - \alpha \frac{\partial \mathcal{L}}{\partial \vec{w}}$

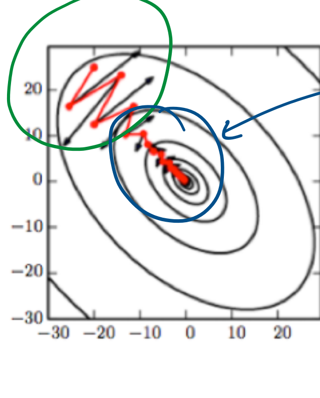
with momentum : $\vec{p} \leftarrow \mu \vec{p} - \alpha \frac{\partial \mathcal{L}}{\partial \vec{w}}$

$\vec{w} \leftarrow \vec{w} + \vec{p}$

μ: damping parameter (it should be slightly less than 1)

α: learning rate

high curvature region
gradients will cancel each other out
⇒ momentum will dampen the oscillations



low curvature region
gradients all point in the same direction
⇒ momentum will accelerate the speed of learning

<u>Second-order optimization</u>

We need more information about the <u>curvature</u> of the loss function

We need to compute 2nd order derivatives

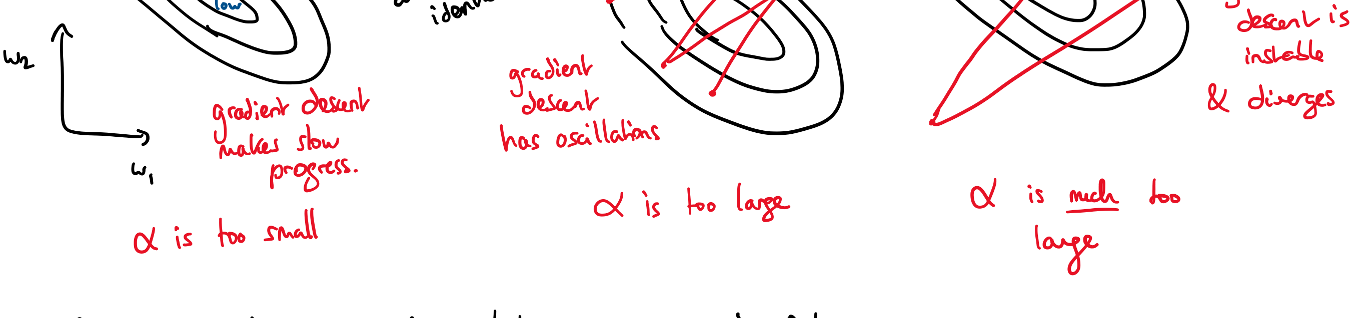
- ↳ complicated to compute
- ↳ difficult to scale them to DNNs (especially when training on large datasets)

One optimizer that uses a little bit of curvature information, yet is still practical for DNNs

Adam typically works better than gradient descent

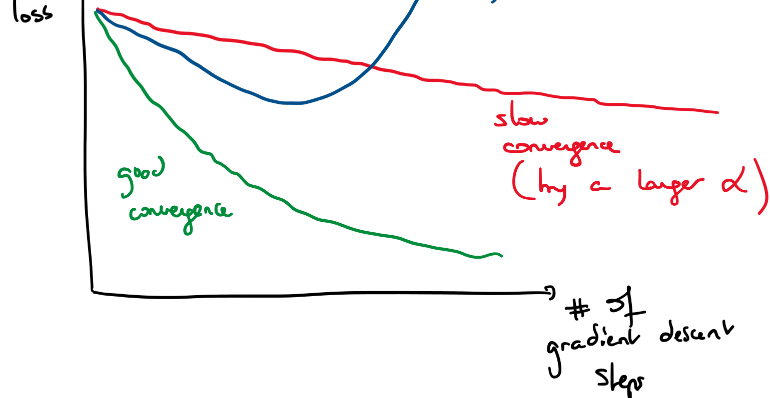
<u>Learning rate</u>

α is a hyperparameter that we need to tune using train / validation / test sets



Good values of α are often between 0.001 and 0.1

<u>Training curves</u>



⚠ training curves are useful to spot major problems with convergence, they do <u>not</u> guarantee convergence.