ANALYSING AND OPTIMIZING PERFORMANCE OF DEEP NEURAL NETWORKS

AARHUS
UNIVERSITY
DEPARTMENT OF BIOSCIENCE

MACHINE LEARNING FOR BIOLOGISTS
14 MAY 2019

JESPER ERENSKJOLD MOESLUND
RESEARCHER

SOME TYPICAL PARAMETERS FOR OPTIMIZING PERFORMANCE

In deep learning, there's no final recipe

Try, try, and try

You both want to enhance train speed and prediction!

Enable both progress speed and how well you solve problem

For example, one typically tweak a DNN by changing:

- Number of layers (i.e., the depth)
- Number og hidden units
- Activation functions
- Learning rate

TRAIN/DEV/TEST SETS

The base for progressing fast with DNNs is a good training, development and test dataset

With many training examples one typically use 90/5/5%, or maybe even 98/1/1%

In some cases you don't want a dev set, you only use train/dev

Distribution of data in each set?

What happens if the train data comes from a different distribution?

Find flowers on pics; then the images in the dev and test needs to come from same source, have same size etc. as train.

Can't go and find test-data afterwards form another distribution!

BIAS AND VARIANCE

BIAS AND VARIANCE

REDUCING VARIANCE - REGULARIZATION

Could get more training data...

Another method is to add regularization to your NN.

The regularization parameter

$$\mathcal{J}(W^{[1]}, b^{[1]},\ldots,W^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}||W^{[l]}||_F^2$$

$||W^{[l]}||_F^2 = \sum_i \sum_j (Wij^{[l]})^2$ = sum of squares of all elements

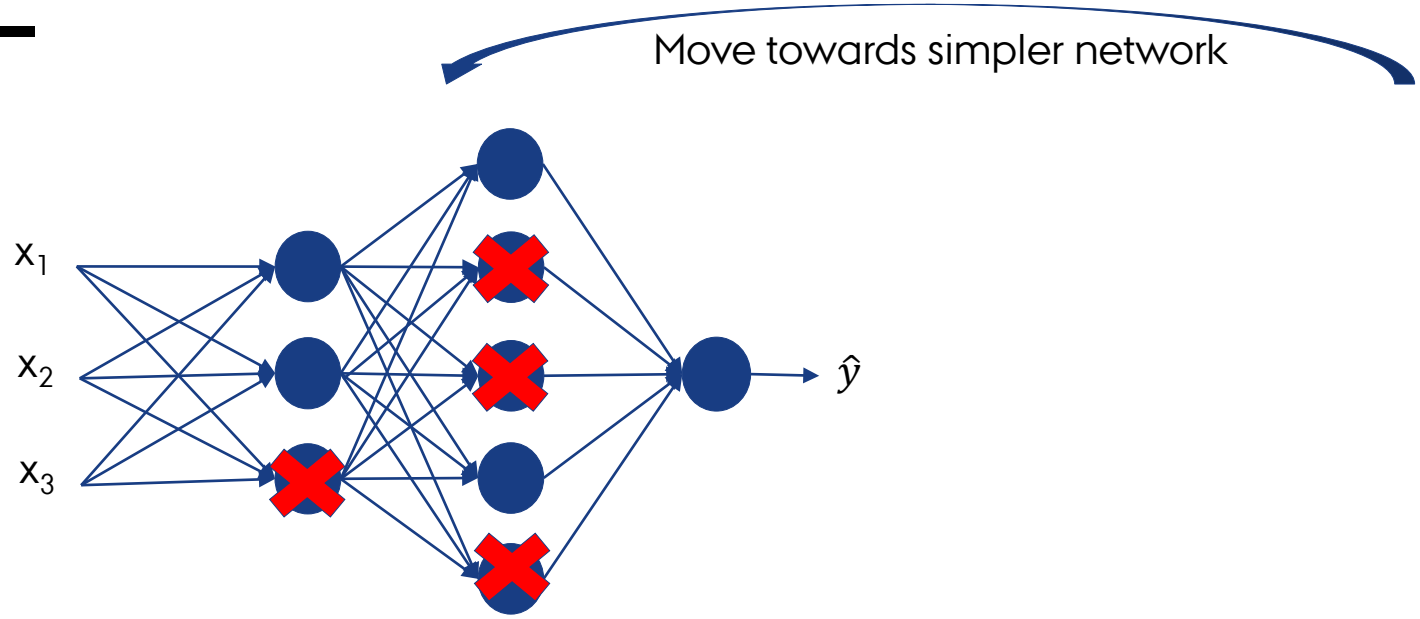
This is called **L2 regularization**

How to update Ws and bs then?

$dW^{[l]}$ = result from backprop + $\frac{\lambda}{m}W^{[l]}$

And then the usual updating...

WHY DOES L2 REGULARIZATION WORK?



Some $W^{[l]}$'s will more quickly approach zero -> not much influence -> simpler network

OTHER REGULARIZATION TECHNIQUES

Dropout – you randomly eliminate a given percentage of the nodes in each layer for each iteration

Works in more or less the same way as L2 – it creates simpler networks for each training iteration

Data augmentation – introduce random variations to training data: flipping, zooming distorting... -> you training data will grow

Early stopping – just stop learning at earlier stage when dev-error starts to grow

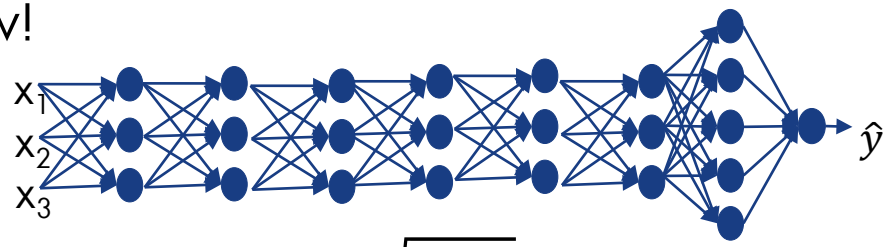
SPEEDING UP TRAINING – BASIC IDEAS

Normalize your training data!

- Makes the shape of the cost-function more optimal for gradient descent
- Saves a lot of oscillating back and forth before finally finding minimum

Initialize clever in very deep NNs:

- Gradients can explode or vanish making training very slow!
- Activations will grow or decrease exponentially with L
- Say your weights are just a tad above 1 and L = 150 !!!



- Partly solved by (for ReLU): Multiplying your randomly initiated weights by $\sqrt{\frac{2}{n^{[l-1]}}}$,
- n is the size of input to a given layer, i.e. 3 in most cases in the NN above
- Called **He initialization**. For other activation functions other methods exist (e.g. Xavier)

MINI-BATCH GRADIENT DESCENT

Having a big training dataset (and you often have in ML), gradient descent can be slow!

Actually you can split up your dataset in mini-batches:

$[(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), (\mathbf{x}^{(3)}, y^{(3)}), \ldots, (\mathbf{x}^{(1000)}, y^{(1000)}, \ldots, (\mathbf{x}^{(2000)}, y^{(2000)}), \ldots, (\mathbf{x}^{(7.000.000)}, y^{(7.000.000)})]$

Batch t: $\mathbf{X}^{\{t\}}$, $\mathbf{y}^{\{t\}}$

Now run training on individual mini-batches

Using $\mathcal{J}^{\{t\}}$ instead of $\mathcal{J}$ as basis for updating weights and biases

Much faster gradient descent: **Mini-batch gradient descent**

MINI-BATCH GRADIENT DESCENT

How does your cost function look while training?

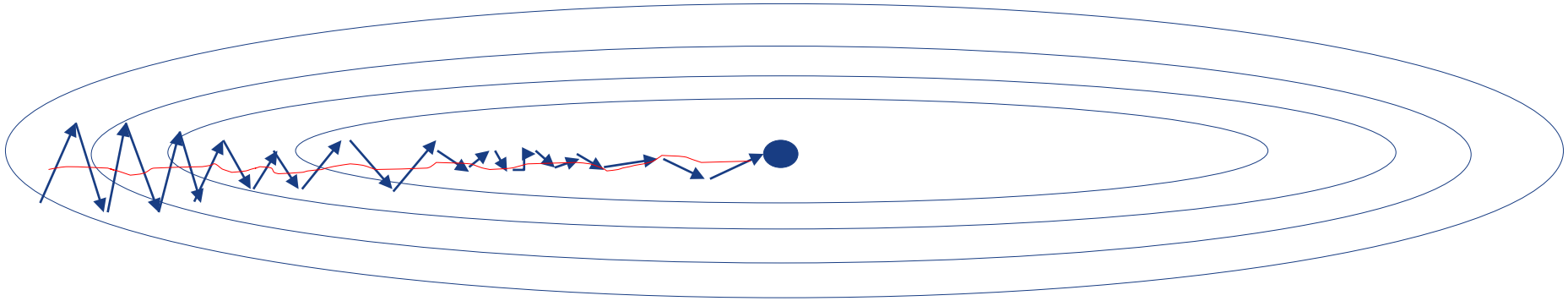
Batch size = m? Usual GD, optimal, but often too slow!

Batch size =1? Lose speed in vectorization (for loop), never converges!

Somewhere in-between ☺

Typical mini-batch size: 64, 128, 256, 512

GRADIENT DESCENT WITH MOMENTUM



On iteration t:

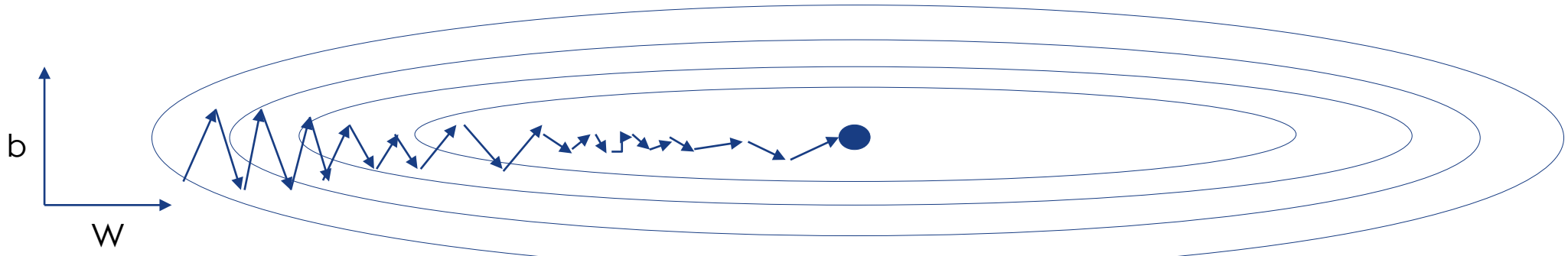
- Compute dW and db on current mini-batch
- $v_{dW} = \beta v_{dW} + (1 - \beta)dW$
- $v_{db} = \beta v_{db} + (1 - \beta)db$
- Update W and B:
  - $W = W - \alpha v_{dW}$
  - $b = b - \alpha v_{db}$

Initiated to 0

Exponentially weighted mean

Typical $\beta = 0.9$

RMSPROP



On iteration t:

- Compute dW and db on current mini-batch
- $s_{dW} = \beta_2 s_{dW} + (1 - \beta_2) dW^2$
- $s_{db} = \beta_2 s_{db} + (1 - \beta_2) db^2$ ← element-wise
- Update W and b:
  - $W = W - \alpha \frac{dW}{\sqrt{s_{dW}}}$
  - $b = b - \alpha \frac{db}{\sqrt{s_{db}}}$

Note that s_{dW} and s_{db} now holds the exponentially weighted means of the squared derivatives

ADAM OPTIMIZATION

On iteration t:

- Compute dW and db on current mini-batch
- $v_{dW} = \beta_1 v_{dW} + (1 - \beta_1)dW$
- $s_{dW} = \beta_2 s_{dW} + (1 - \beta_2)dW^2$
- $v_{db} = \beta_1 v_{db} + (1 - \beta_1)db$
- $s_{db} = \beta_2 s_{db} + (1 - \beta_2)db^2$
- $v_{dW}^{corrected} = \frac{v_{dW}}{(1 - \beta_1^t)}$, $v_{db}^{corrected} = \frac{v_{db}}{(1 - \beta_1^t)}$
- $s_{dW}^{corrected} = \frac{s_{dW}}{(1 - \beta_2^t)}$, $s_{db}^{corrected} = \frac{s_{db}}{(1 - \beta_2^t)}$

Hyperparameters

α: Typically the most important to tune

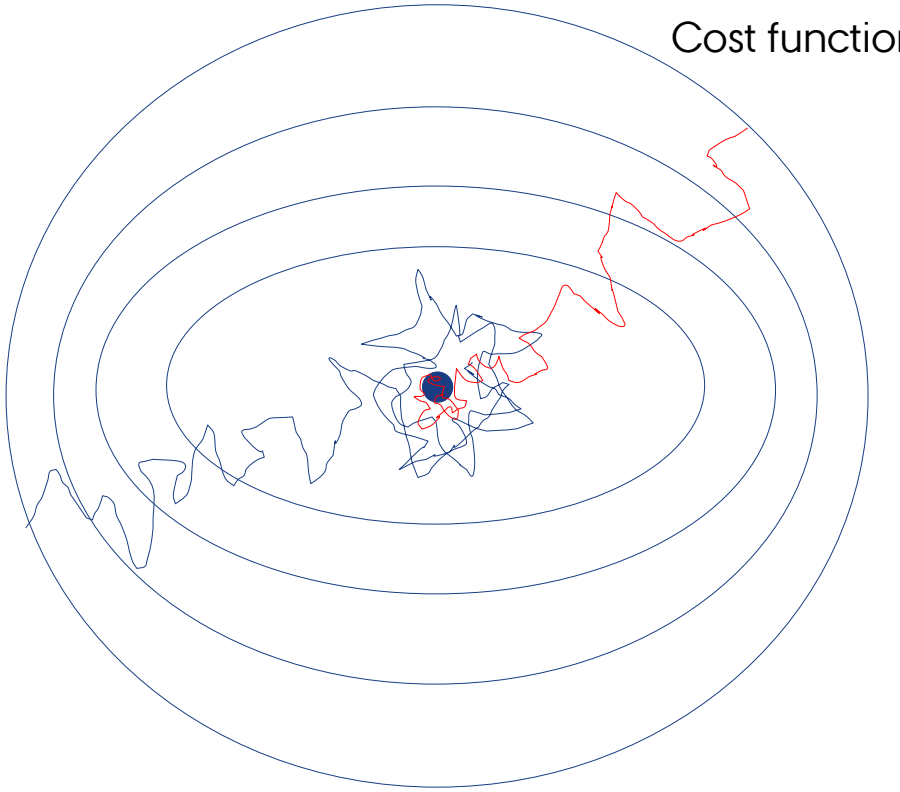
β_1: 0.9 (could tune)

β_2: 0.999 (could tune)

Update W and b:

- $W = W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}}}$
- $b = b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected}}}$

LEARNING RATE DECAY



$$\alpha = \frac{1}{1+decay\ rate+epoch\ number}\alpha_0$$

RECIPE FOR TUNING HYPERPARAMETERS

- α
- β
- β_1, β_2
- Number of layers
- Number of hidden units
- Decay rate (learning rate)
- Mini-batch size

Don't search in a grid fashion!
Try random values!
Zoom in
Think about the scale

AARHUS
UNIVERSITY