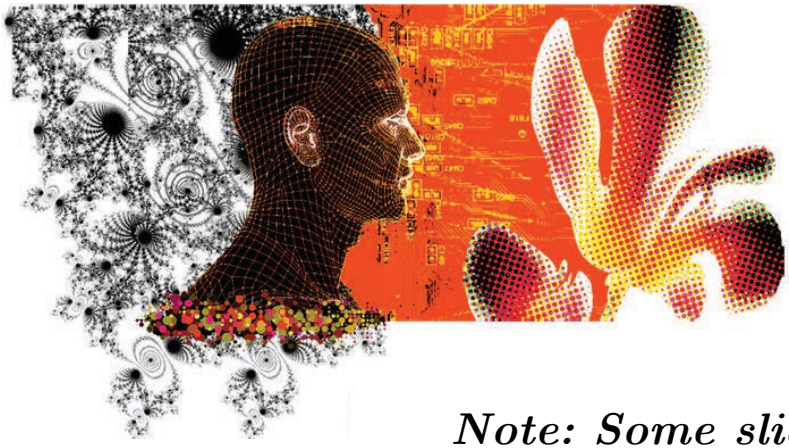
Data Science Training

November 2017

Training Neural Networks

Xavier Bresson

Data Science and AI Research Centre
NTU, Singapore



http://data-science-optum17.tk

Note: Some slides are from Li, Karpathy, Johnson's course on Deep Learning.

Outline

- **Generic Gradient Descent Techniques**
- Backpropagation
- Activation
- Weight Initialization
- Neural Network Optimization
- Dropout
- Conclusion

Loss Function for Classification

- **Classification:** Use **training data** (x_i, y_i) to design a **score function** s for classification:

$$s = f(W, x) = Wx$$

- **Weight W:** They are found by **minimizing a loss function** which quantifies how well the training data have been classified:

(1) SVM loss:
$$L_i(W) = \sum_{j \neq i} \max(0, s_j - s_i + 1)$$

(2) Softmax loss:
$$L_i(W) = -\log \frac{e^{s_i}}{\sum_j e^{s_j}}$$

(3) Regularization:
$$E(W) = \sum_i L_i(W) + \lambda R(W)$$

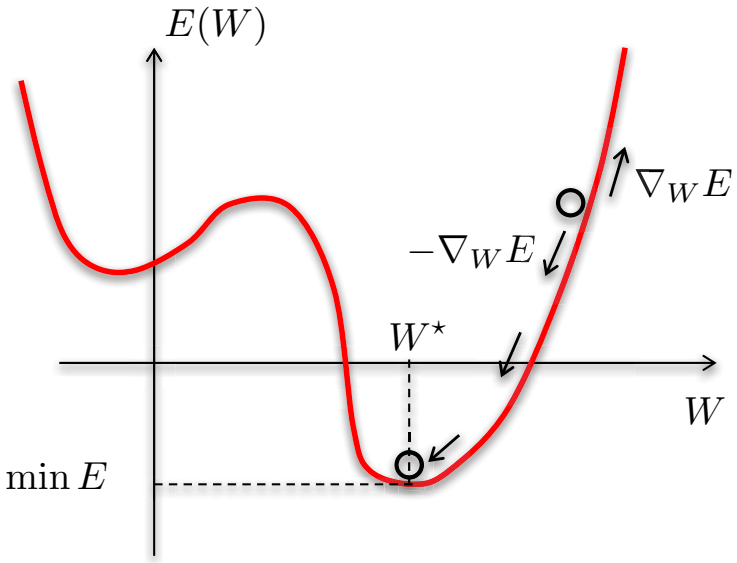
- **Q: How to minimize loss functions?**

  A: *Steepest gradient descent*

Gradient Descent Techniques

- Gradient descent: **Most standard optimization technique!**

  Note: This class of techniques are **weak** in optimization, but they are the **most generic** when the energy landscape is difficult, that is non-convex. Training neural networks is (very) **slow** because of the gradient descent **bottleneck** $\Rightarrow$ new research is on-going to speed up NN optimization.



Gradient Operator

- **Two types:**

  (1) Analytic gradient: $\nabla_W E = \frac{\partial E}{\partial W} =$ explicit formula

  (2) Numerical gradient: $\frac{\delta E}{\delta W} = \frac{E(W + \Delta W) - E(W)}{\Delta W}$

- **Properties of numerical gradient:**

  (i) Approximation of analytic gradient.

  (ii) Slow to evaluate (compared to analytic gradient).

Evaluation the gradient numerically:

```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

current W:	**W + h** (first dim):	**gradient dW:**
[0.34,	[0.34 + **0.0001**,	[**-2.5**,
-1.11,	-1.11,	?,
0.78,	0.78,	?,
0.12,	0.12,	
0.55,	0.55,	
2.81,	2.81,	
-3.1,	-3.1,	
-1.5,	-1.5,	
0.33,…]	0.33,…]	?,
loss 1.25347	**loss 1.25322**	?,…]

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h}$$

Analytic Gradient

- **Properties:**

  (1) Exact value (use Calculus) $\quad E(W) = \|W\|_F^2 \quad \rightarrow \quad \nabla_W E = \dfrac{\partial E}{\partial W} = 2W$

  (2) Fast to evaluate.

- Common practice: **Gradient Check**

  Always use analytical gradient but check its implementation with numerical gradient.

Update Rule

- **Update:**

$$\frac{\delta W}{\delta t} = \frac{W^{m+1} - W^m}{\tau} = -\nabla_W E(W^m)$$

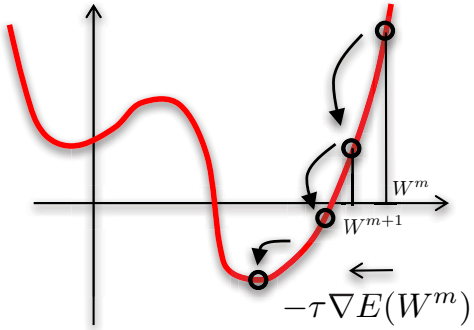
$$\Downarrow$$

$$W^{m+1} = W^m - \tau \nabla_W E(W^m)$$

Time step/
Learning rate/
Step size

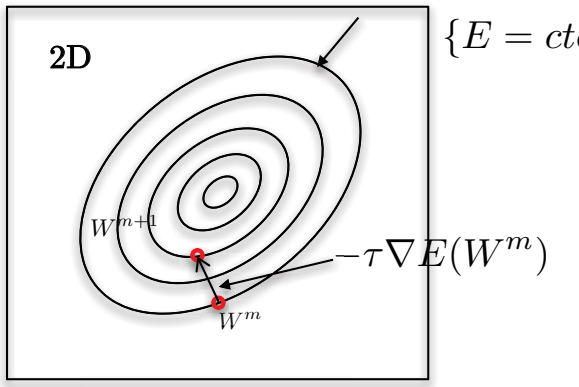
Speed of
Gradient descent
techniques

negative



Level sets of E:

$\{E = cte\}$

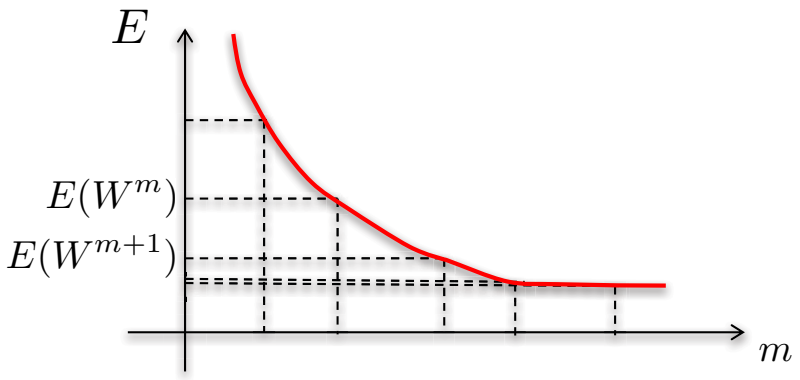


- **Code:**

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Monotonicity

- **Loss/energy value decreases monotonically at each iteration m:**



Q: What happens with big data?

Large value, e.g. n=billions

$$E(W) = \sum_{i=1}^{n} L_i(W) + \lambda R(W)$$

Analytic gradient uses all data at the same time ⇒ *it is* ***not*** *possible to load all data in memory.*

Mini-batch/Stochastic Gradient Descent

- *Special property of loss functions:* **Additively separable functions**, i.e. functions that are the **sum of a single data function L_i** (independent of all other data):

$$\min_W \; L(W) = \sum_{i=1}^{n} L_i(W)$$

$$= \sum_{i \in n_1} L_i(W) + \sum_{i \in n_2} L_i(W) + ... + \sum_{i \in n_q} L_i(W)$$

$$n = n_1 + n_2 + ... + n_q$$

$$\Downarrow$$

$$\nabla L(W) = \sum_{i \in n_1} \nabla L_i(W) + \sum_{i \in n_2} \nabla L_i(W) + ... + \sum_{i \in n_q} \nabla L_i(W)$$

$$\Downarrow$$

*only use a **small portion** of the training set to compute the gradient*

- **Stochastic gradient descent:**

$$W^{m+1} = W^m - \tau \sum_{i \in n_j} \nabla L_i(W^m)$$

- **Deterministic gradient descent:**

$$W^{m+1} = W^m - \tau \nabla L(W^m)$$

All data

Mini-batch/Stochastic Gradient Descent

- **More details:**

Iterate n_e epochs:

For each epoch, iterate over all mini-batches $j=1,\ldots,n_q$:

$$W^{m+1} = W^m - \tau \sum_{i \in n_j} \nabla L_i(W^m)$$

Note1: An **epoch** is a complete pass of all training data.

Note2: At each new epoch, **randomly shuffle** training data (improve significantly results).

Note3: **Stochastic consistency:**

$$\mathbb{E}\Big(\sum_{i \in n_j} \nabla L_i(W^m) \Big) \overset{m \to \infty}{\to} \frac{1}{n} \sum_{i=1}^{n} \nabla L_i(W^m)$$

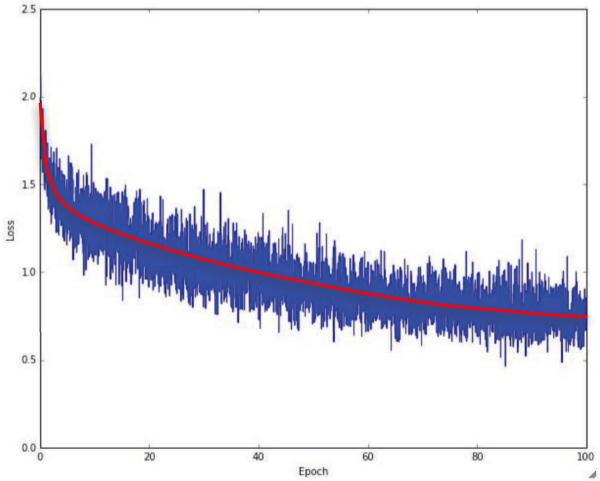
$$\mathbb{E}(W^{m+1}) \overset{m \to \infty}{\to} W^{m+1}$$

Stochastic Monotonicity

- **Code:**

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- **Loss vs epochs** (iteration m):



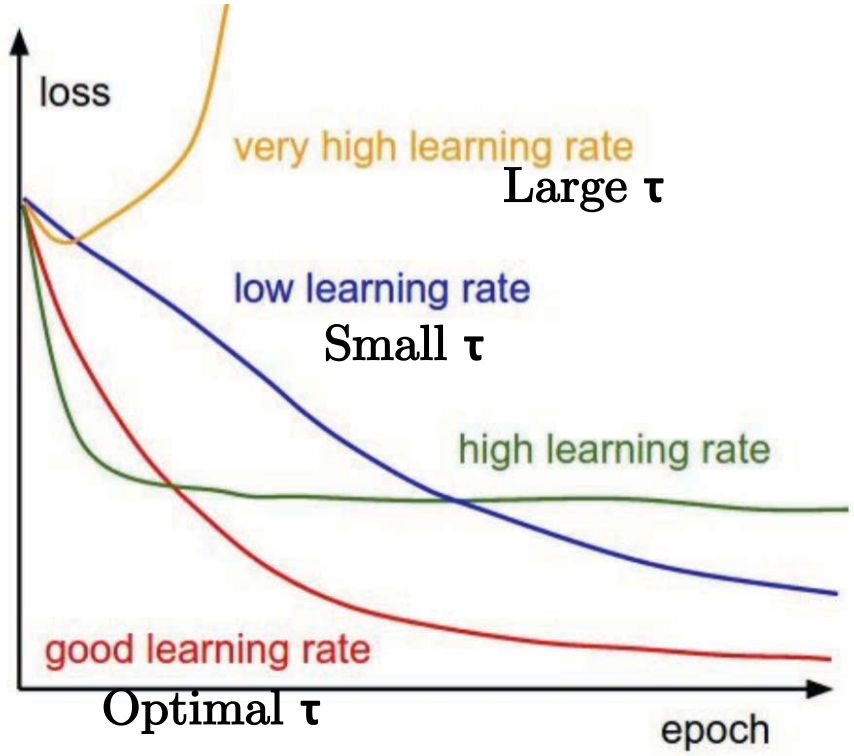
Note1: Mini-batch size: 32, 64, 128, limited by GPU memory.

Note2: Several works to speed up optimization: Momentum, Adagrad, Adam, etc (later discussed), but still major bottleneck of NN training.

Note3: **Stochastic gradient descent technique** is not only used for large-scale neural networks, but also **for most big data problems**: k-means clustering, SVM classification, Lasso regression, recommendation, etc.

Influence of Learning Rate τ

- **Challenging** problem to find the **optimal** step size τ:



Outline

- Generic Gradient Descent Techniques
- **Backpropagation**
- Activation
- Weight Initialization
- Neural Network Optimization
- Dropout
- Conclusion

Computational Graph

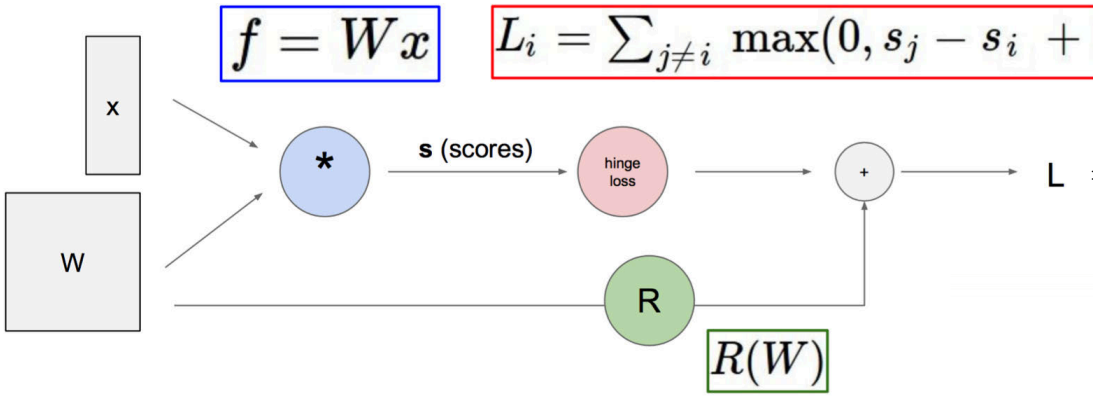
- **Neural networks** (NNs) are represented by **computational graphs** (CGs).

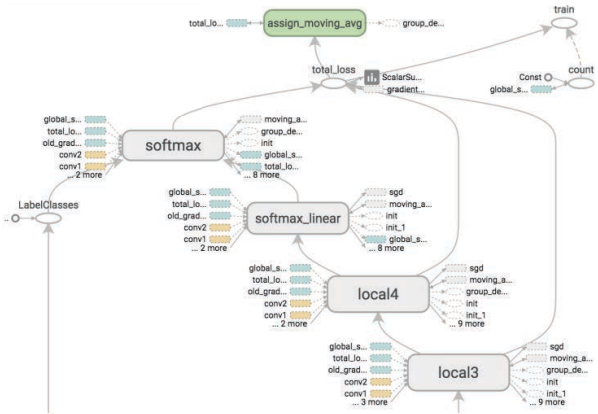
  Definition: A series of operators applied to inputs. Easy to combine (lego strategy), can be huge.

  Usefulness: Clear **visualization** of NN operations (great for debugging).

  CG are essential to derive gradients by **backpropagation**.

Computational Graph:





Google Tensorflow

Backpropagation

- **Definition: A recursive application of chain rule along a computational graph (CG)** $\Rightarrow$ provides the gradients of all weights, intermediate variables.

$$W^{m+1} = W^m - \tau \nabla_W E(W^m)$$

- **Chain rule:** Calculus:

$$\frac{\partial L}{\partial x} = \frac{\partial L(F(x))}{\partial x} = \frac{\partial L}{\partial F} \cdot \frac{\partial F(x)}{\partial x}$$

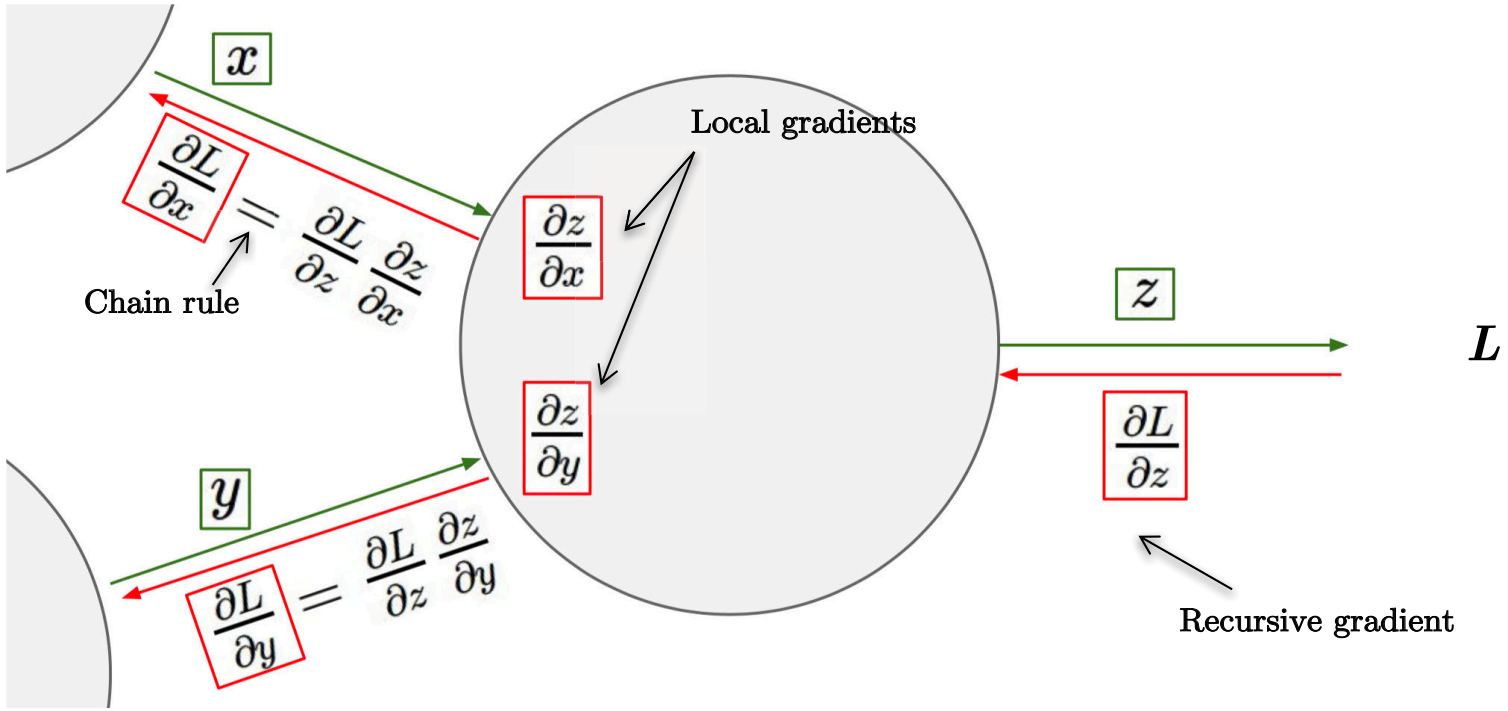
- **Essential property of backpropagation:** It can compute the gradient of any variable in the CG by a **simple local rule**, independently of the size of the CG (very deep NNs).

Local Rule

- Any computational graph is a **series of elementary neurons** (also called nodes, gates) ⇒ The gradient of the loss w.r.t. the inputs x, y of the local neurons can be computed with the **local rule**:

Gradient L w.r.t. x, y =

Recursive gradient * Local gradient w.r.t. x, y

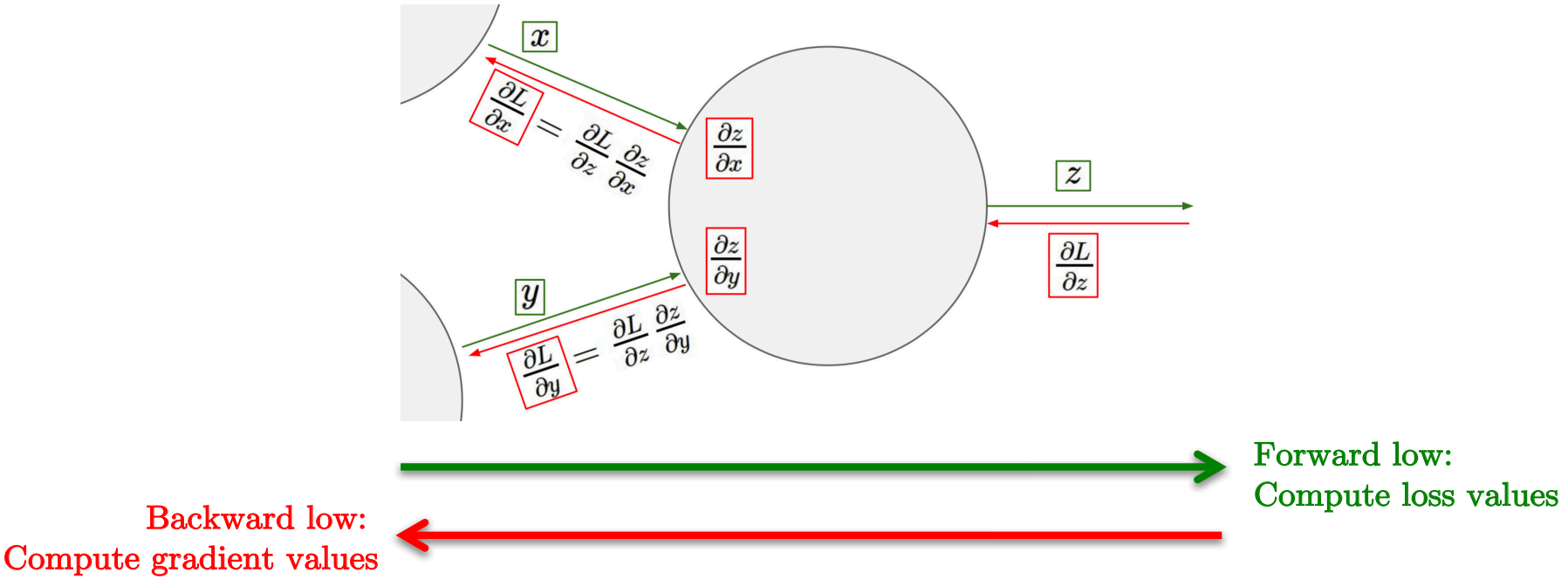


Backpropagation Techniques

- Backpropagation consists of two steps:

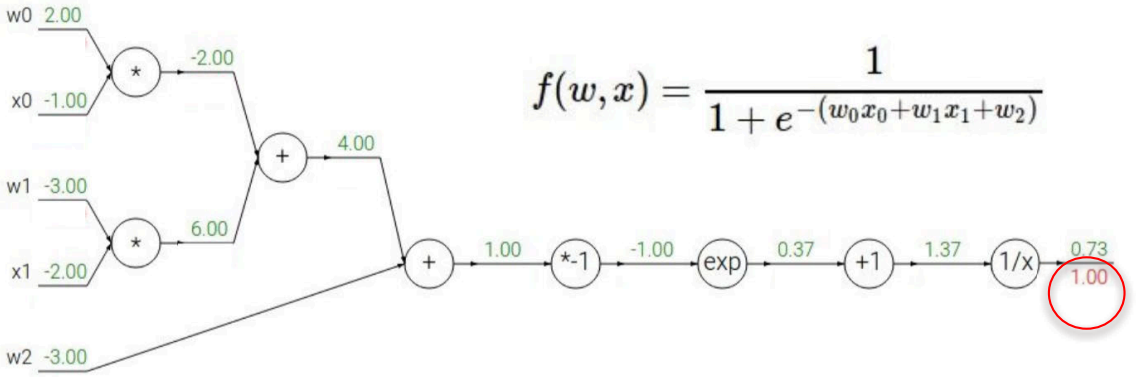
  (1) Forward pass/flow: *Compute final loss value and all intermediate output values of neurons/nodes. Save them in memory for gradient computations (in backward step).*

  (2) Backward pass/flow: *Compute the gradient of the loss functions w.r.t. all variables on the network using the local gradient rule.*



An Example

- Step 1:

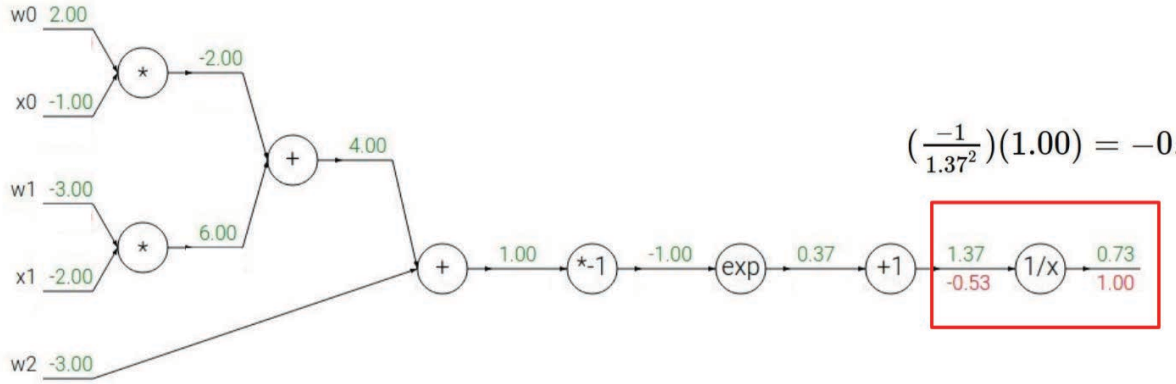


$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

$$\frac{\partial f}{\partial f} = 1$$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

- Step 2:

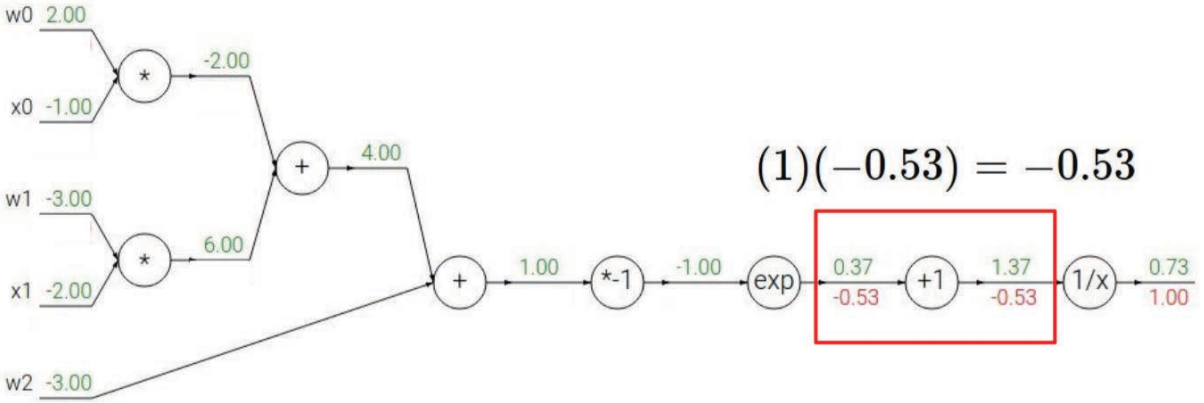


$$(\tfrac{-1}{1.37^2})(1.00) = -0.53$$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

An Example

- Step 3:



$$(1)(-0.53) = -0.53$$

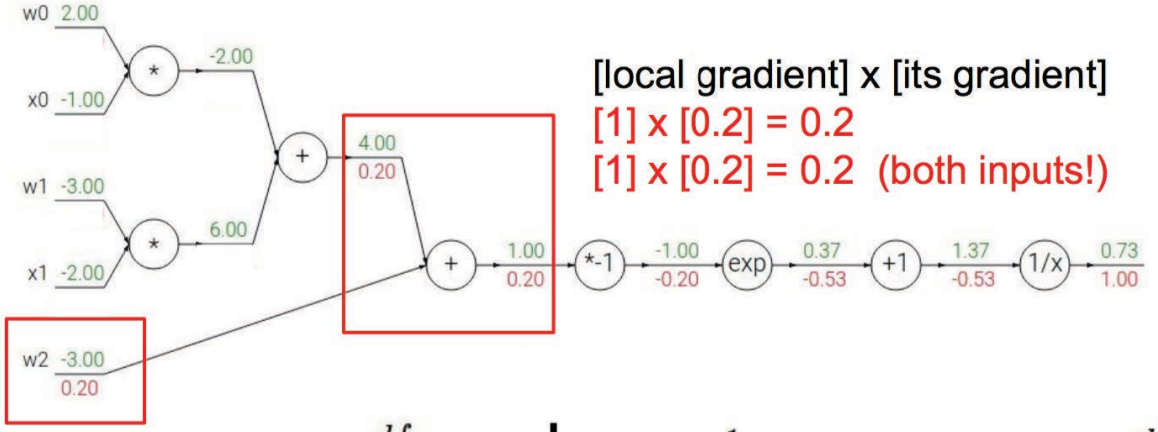
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

- Step ...:



[local gradient] x [its gradient]
[1] x [0.2] = 0.2
[1] x [0.2] = 0.2 (both inputs!)

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

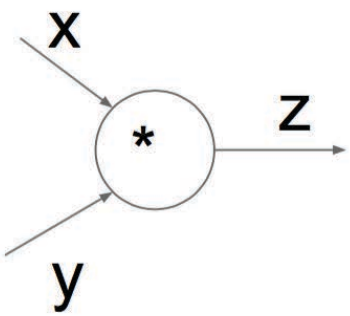
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Backpropagation Implementation
Forward and Backward Functions

- Code:

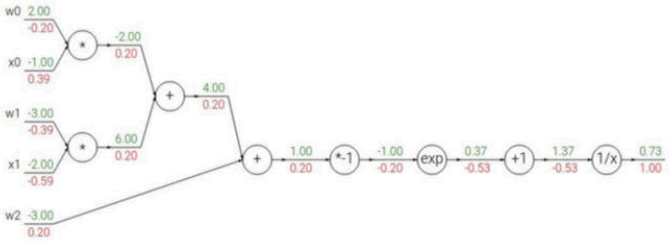



```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

(x,y,z are scalars)

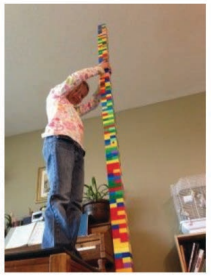
Backpropagation Implementation
Forward and Backward Functions

- Pseudo-code:



Graph (or Net) object.

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```



Gradient with Vectorized Code

- When variables x,y,z are row vectors:

$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial L}{\partial f}$$

Jacobian Matrix

$$\frac{\partial f}{\partial x} = \left[\frac{\partial f_i}{\partial x_j} \right]$$

Q: **What is the size of a Jacobian matrix if input size is 4096?**

A: J size is **4096 x 4096 = 16M variables** !

$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial L}{\partial f}$$

4096 x 1 | 4096 x 4096 | 4096 x 1

- **Good news:** Most computations are **element-wise** in computational graphs, i.e. apply to each element x independently of the other elements. In this case, **Jacobian = diagonal matrix** ⇒ Very fast computations in parallel with *vectorized operations on CPUs.*

Backpropagation Cost

- **Cost forward ≈ cost backward** (slightly higher)

- **Backward requires to store forward values!**

- **Mini-batch optimization with backpropagation:**
  (1) Sample randomly a batch of data
  (2) Forward propagate to get loss values
  (3) Backward propagate to get gradients
  (4) Update neural network weights and other intermediate variables

  $\Rightarrow$ *Works on huge computational graphs.*

Outline

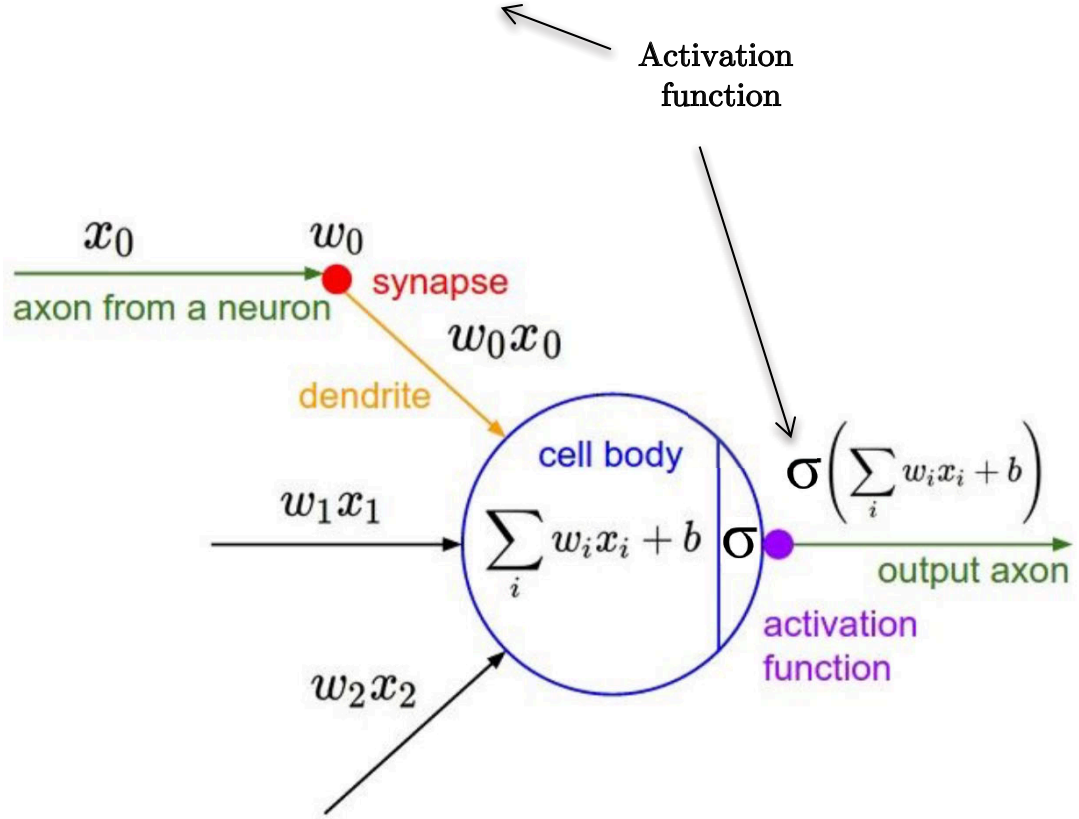
- Generic Gradient Descent Techniques
- Backpropagation
- **Activation**
- Weight Initialization
- Neural Network Optimization
- Dropout
- Conclusion

Activation Functions

- Reminder: *Neural network classifiers are succession of linear classifications and non-linear activations.*

  Exs: 2-layer classifier: $f = W_2 \max(W_1 x, 0)$

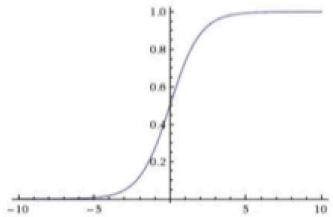
  3-layer classifier: $f = W_3 \max(W_2 \max(W_1 x, 0), 0)$



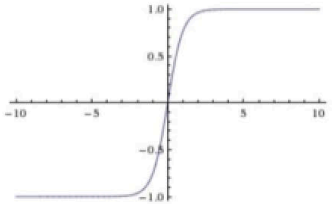
Class of Activation Functions

Sigmoid

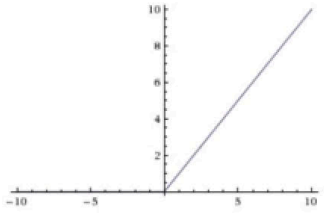
$$\sigma(x) = 1/(1+e^{-x})$$



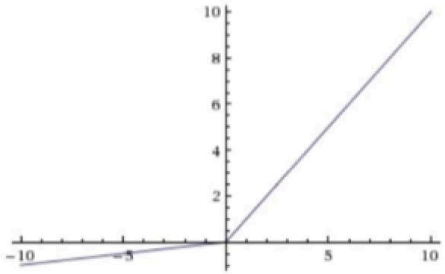
tanh tanh(x)



ReLU max(0,x)

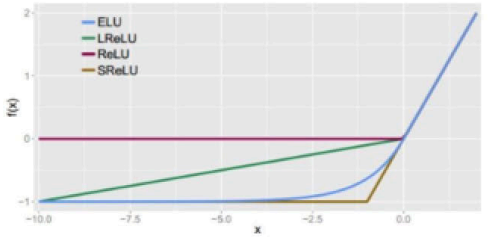


Leaky ReLU
max(0.1x, x)



ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

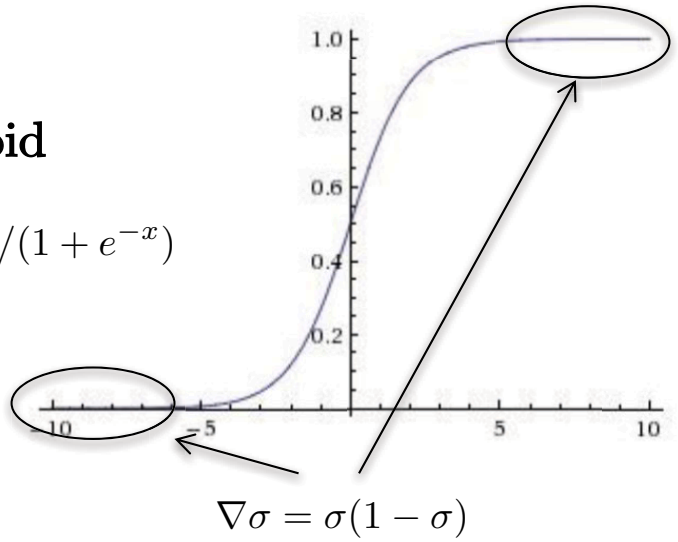


Sigmoid Activation

- **Historically popular** by analogy with neurobiology.

Sigmoid

$$\sigma(x) = 1/(1+e^{-x})$$



$$\nabla\sigma = \sigma(1-\sigma)$$

- **Three issues:**

(1) Saturated neurons kill gradients

$\Rightarrow$ **Vanishing gradient problem**

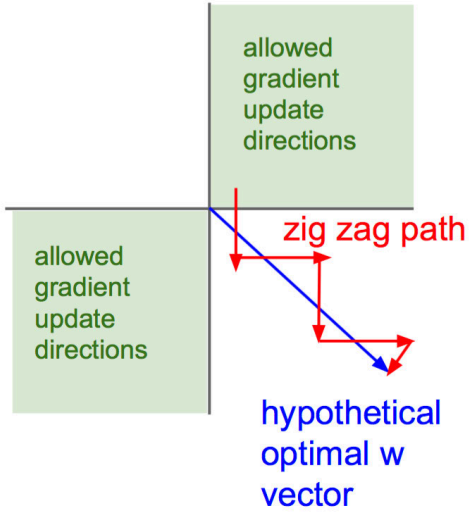
(2) Exp is a bit computationally expensive.

(3) Sigmoid are not zero-centered:

Suppose input neurons are always positive, then gradients are either all positive, or all negative:

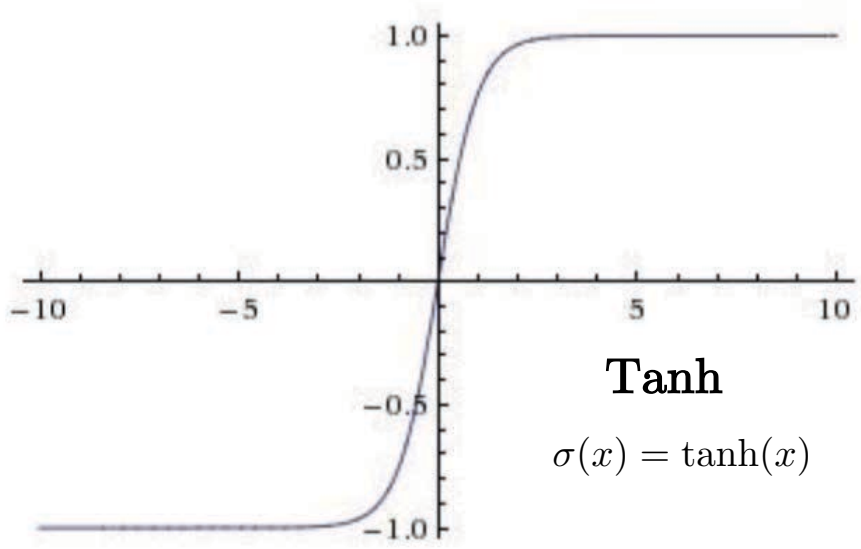
$$f(z = w_1x_1 + w_2x_2) \rightarrow \frac{\partial f}{\partial w_i} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial f}{\partial z}x_i, \quad x_i \geq 0$$

$\Rightarrow$ **Always zero-center your data!**



Tanh Activation [LeCun-et.al'91]

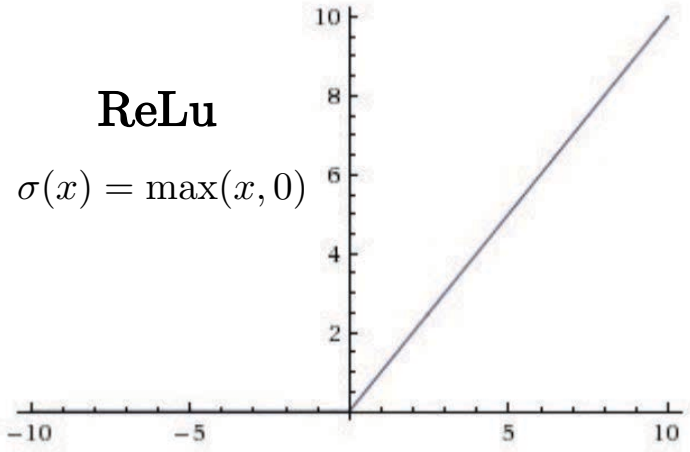
- **Advantage:**
  Zero-centered function ☺



- **Issue:**
  Kill gradients $\Rightarrow$ *Vanishing gradient problem*

ReLu Activation [Hinton-et-al'12]

- **ReLU (Rectified Linear Unit):**



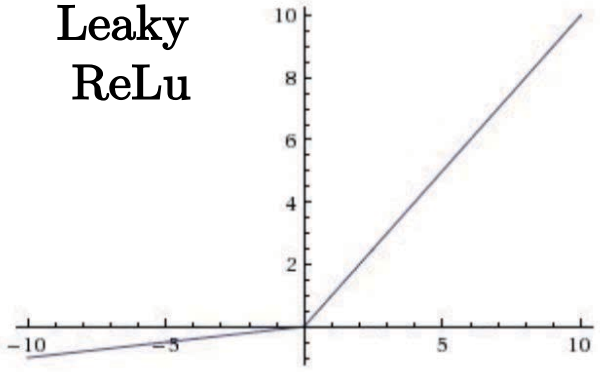
- **Advantages:**
  - *(1) Converges 6x faster than sigmoid/tanh*
  - *(2) Does not saturate in positive region*
  - *(3) Max is computationally efficient*

- **Limitations:**
  - *(1) Not zero-centered function*
  - *(2) It kills gradient when input is negative*

  $\Rightarrow$ Standard trick: Initialize neurons with a small positive biases like 0.01.

Leaky ReLu [Mass-et.al'13]

- **Advantages: ReLU (Rectified Linear Unit)**
  - *(1) Converges 6x faster than sigmoid/tanh*
  - *(2) Does not saturate in positive region*
  - *(3) Max is computationally efficient*
  - *(4) It does not kill the gradient*
  - *(5) Parameter α can be learned by backpropagation.*



$$\sigma(x) = \max(x, \alpha x), \ \alpha = 0.01$$

- **In practice: Use ReLU**, try out Leaky ReLu, expect less from tanh and sigmoid.

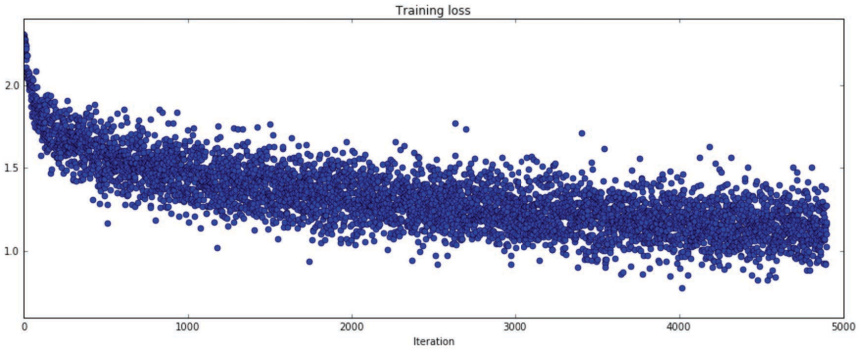
Demo: Train Fully Connected Neural Networks with Backpropagation

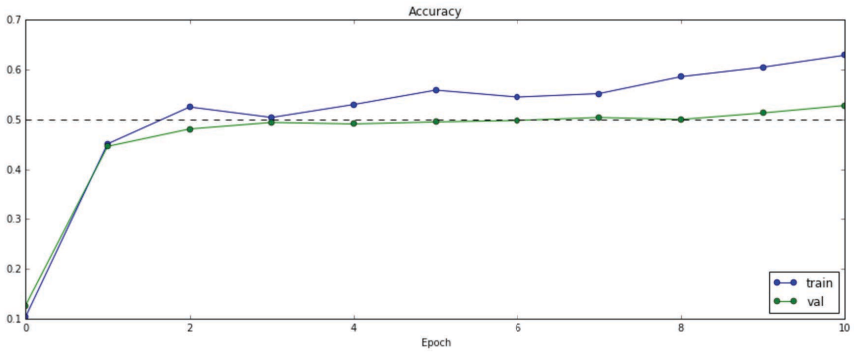
- Run **code01.ipynb**

```python
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return

```python
def layer_backward(dout, cache):
  """
  Receive derivative of loss with respect to outputs and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```





Outline

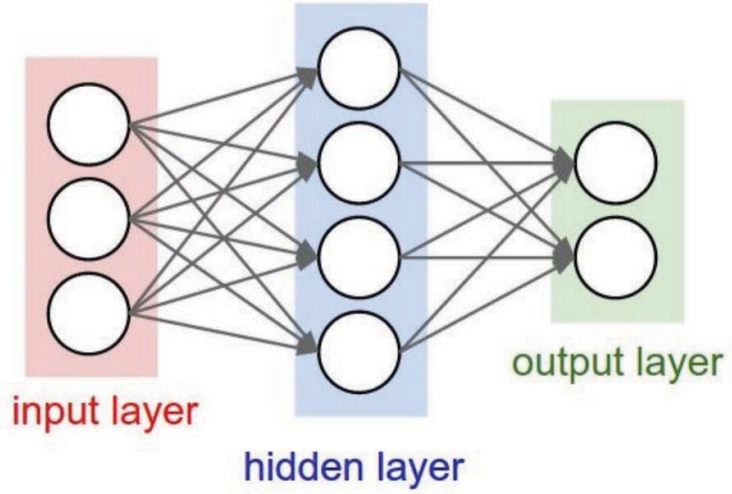
- Generic Gradient Descent Techniques
- Backpropagation
- Activation
- **Weight Initialization**
- Neural Network Optimization
- Dropout
- Conclusion

Weight Initialization

Q: What happens when initial W=0 is used?

A: All neurons compute the **same** outputs and the weights are the same.

⇒ Need to break symmetry!



➢ *Natural idea:* Use **small random numbers** for initialization.

W = Gaussian/normal distribution with zero mean and 1e-2 standard deviation

```
W = 0.01* np.random.randn(D,H)
```

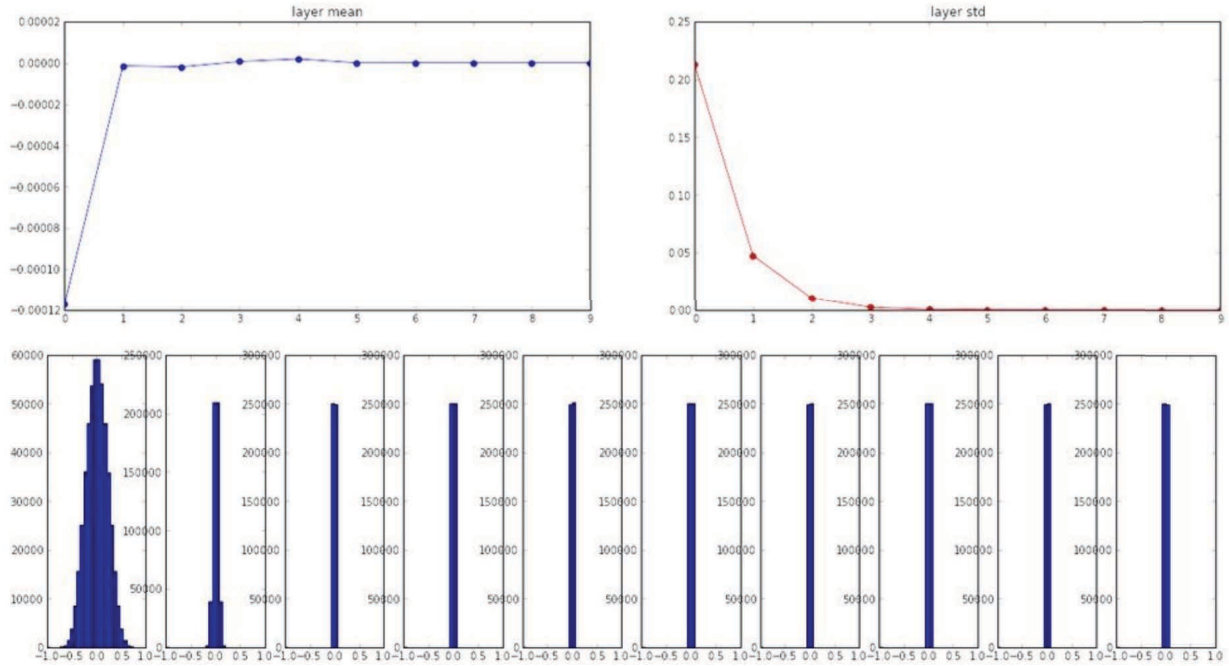
⇒ *Works well for **small** networks, but **not** for deep networks.*

Vanishing Gradient Problem

- **Example:** 10-layer net, 500 neurons on each layer, tanh activation, and initialization: `W = 0.01* np.random.randn(D,H)`

- **Collect weight statistics at each layer:** *means, variances, histograms*

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!
Q: Why?

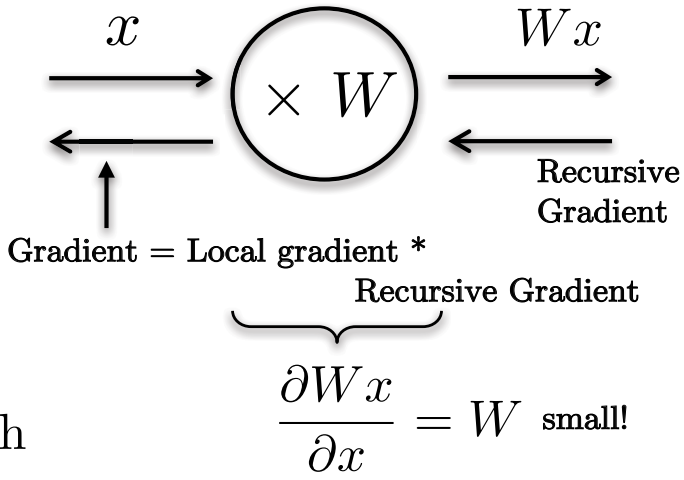
Vanishing Gradient Problem

- *Vanishing gradient:* **At initialization, all weights W are small**. This has two consequences:

  (1) At each layer, output backpropagated gradient is small

  (2) We chain all recursive gradients by backpropagation:

  W x W x W x ... x W $\Rightarrow$ exponential decay at each output (e.g. W=0.1^{10})

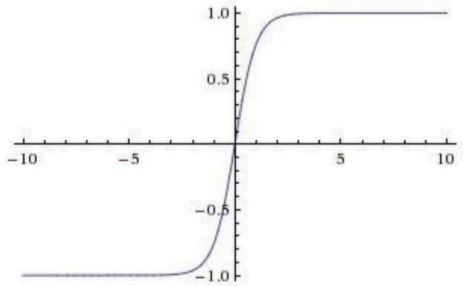


$\Rightarrow$ *The deeper the network the lower the gradient!*
This is the **vanishing gradient problem** (was a big issue for long time).

Exploding Gradient Problem

- Let us try the opposite: `W = 1 * np.random.randn(D,H)`

$\Rightarrow$ *All neurons get saturated to -1 and +1 (tanh activation), and then the gradients = 0.*



$\Downarrow$

$\Rightarrow$ *Tricky to set a good value for the mean of the normal distribution:*

$\Downarrow$

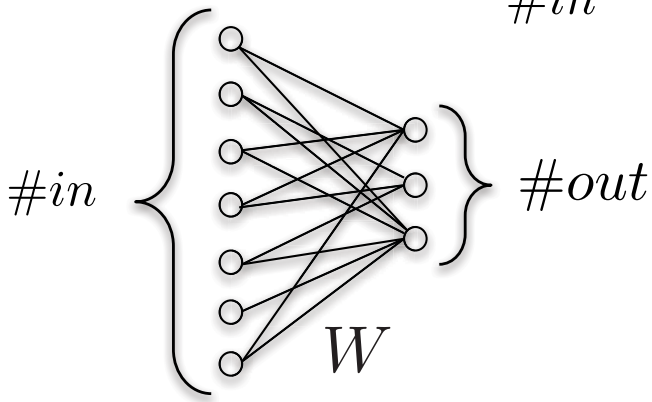
`W = η * np.random.randn(D,H)`

$\eta \in [0.01, 1]$

Xavier's Initialization [Glorot-et-al'10]

- **Idea:** *Pick an **initial** W that keep all layers with variance 1:*

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```



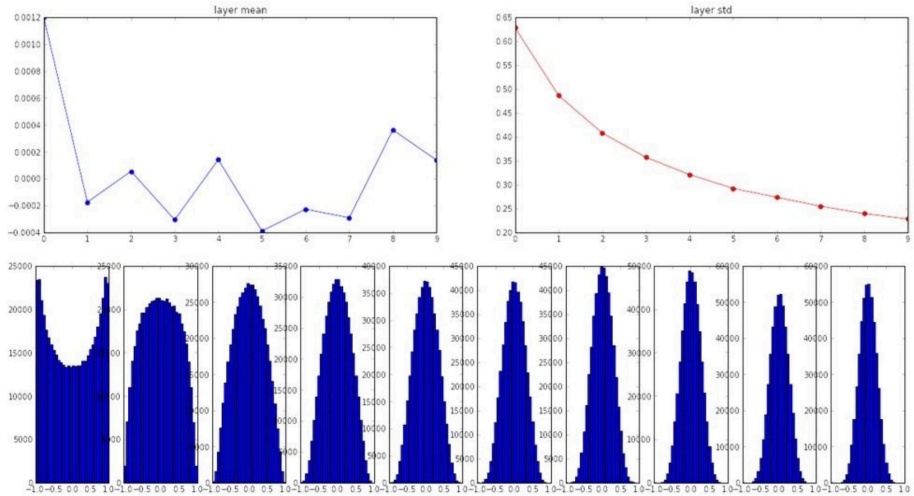
$$Var = \sum_{\#in} W^2 = \#in\left(\frac{1}{\sqrt{\#in}}\right)^2 = 1$$

$\Rightarrow$ *It scales the gradient:*

(1) Large #inputs $\Rightarrow$ small weights

(2) Small #inputs $\Rightarrow$ large weights

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```



- **Theory:** Reasonable initialization but mathematical derivation assumes **linear** activations $\Rightarrow$ It works well for tanh, but not for ReLU. It needs a small change:

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

Batch Normalization [Ioffe-Szegedy'15]

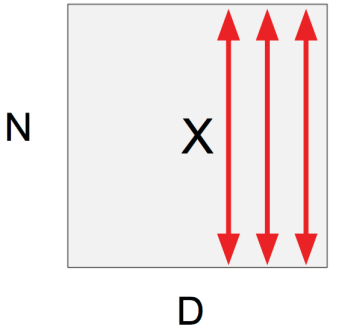
- **Motivation:** Unit Gaussian activations are desirable all over the network
  $\Rightarrow$ let us **enforce** this property!

- **Formula:**

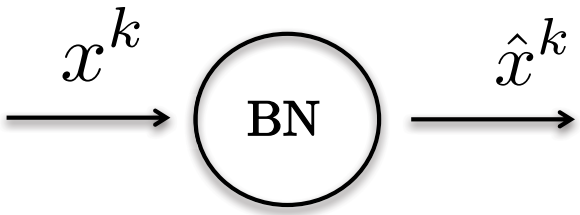
layer

$$\hat{x}^k = \frac{x^k - \mathbb{E}(x^k)}{\sqrt{\mathrm{Var}(x^k)}}$$

BN along each dimension:



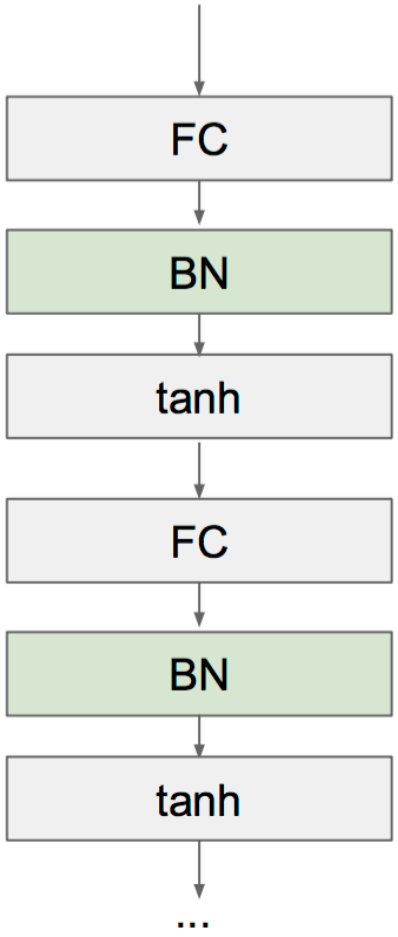
- **Node/gate in NN:**



Smooth and differentiable function
$\Rightarrow$ Backpropagation can be carried out ☺

Where to Insert BN in NN?

- Usually inserted **after Fully Connected layers** (or convolutional layers, next lecture) and **before nonlinearity:**



- *Q: But do we necessarily want a unit Gaussian input to a tanh layer?*

  A: **Not necessarily** $\Rightarrow$ Let the network decide by backpropagation.

$$\Downarrow$$

Normalize: $$\hat{x}^k = \frac{x^k - \mathbb{E}(x^k)}{\sqrt{\mathrm{Var}(x^k)}}$$

Then allow the network to change the range if it wants to:

$$y^k = \gamma^k \hat{x}^k + \beta^k$$

Note the network can learn the identity mapping if it wants to: $$\begin{cases} \gamma^k = \sqrt{\mathrm{Var}(x^k)} \\ \beta^k = \mathbb{E}(x^k) \end{cases}$$

Properties

- **Properties:**
  - *(1) Reduces strong dependence on initialization*
  - *(2) Improves the gradient flow through the network*
  - *(3) Allows higher learning rates*
    ⇒ Learn faster the network
  - *(4) Acts as regularization*

Pseudo-code:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- **Price:** 30% more computational time.

- **At test time:** Mean and variance are estimated during training and average values are selected.

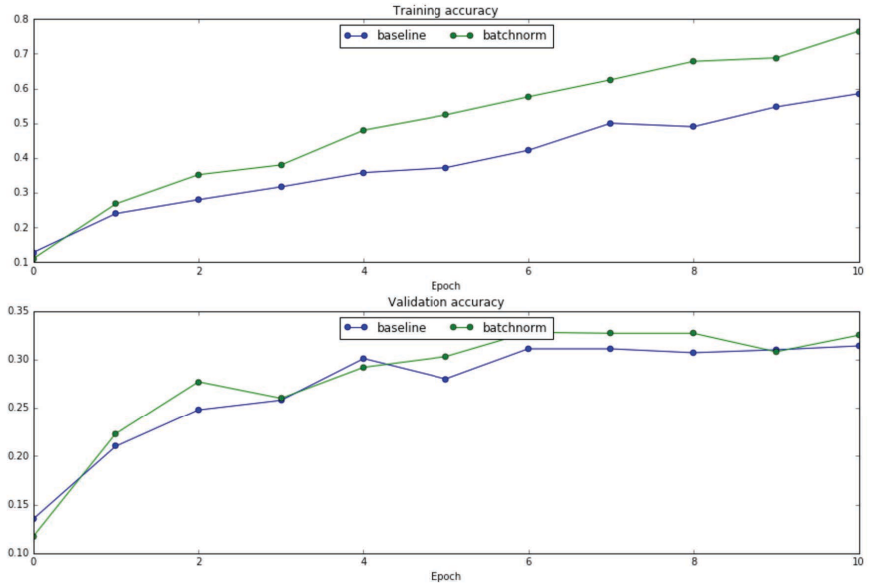
Demo: Batch Normalization

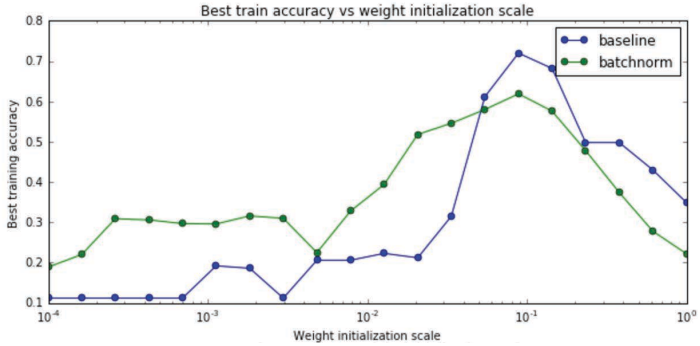
- Run **code02.ipynb**

Batch normalization: Forward

In the file `lab/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. to test your implementation.

```
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization
```



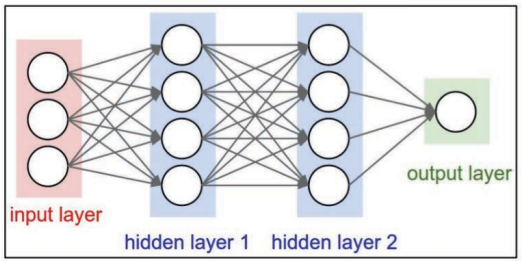


Outline

- Generic Gradient Descent Techniques
- Backpropagation
- Activation
- Weight Initialization
- **Neural Network Optimization**
- Dropout
- Conclusion

Optimization for NNs

- **Training Neural Networks:**
  - *(1) Sample a batch of data.*
  - *(2) Forward prop it through the graph, get loss values*
  - *(3) Backprop to calculate the gradients*
  - *(4) Update the parameters using gradients*

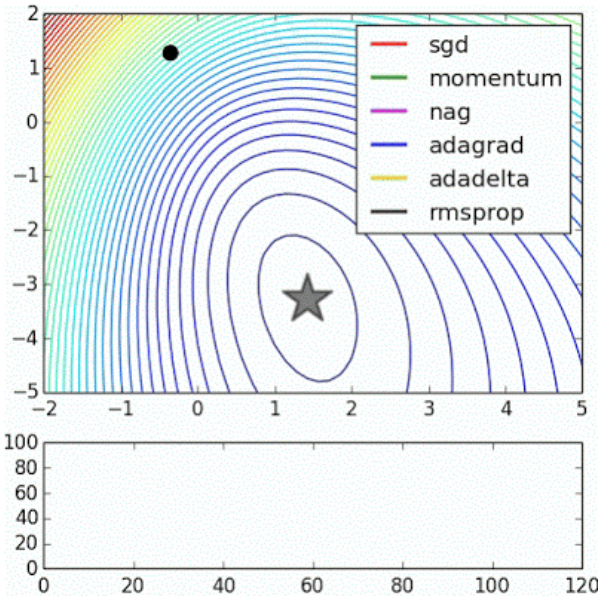


- **Code (main loop):**

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

Q: Can we do better than simple stochastic gradient descent (SGD) update?

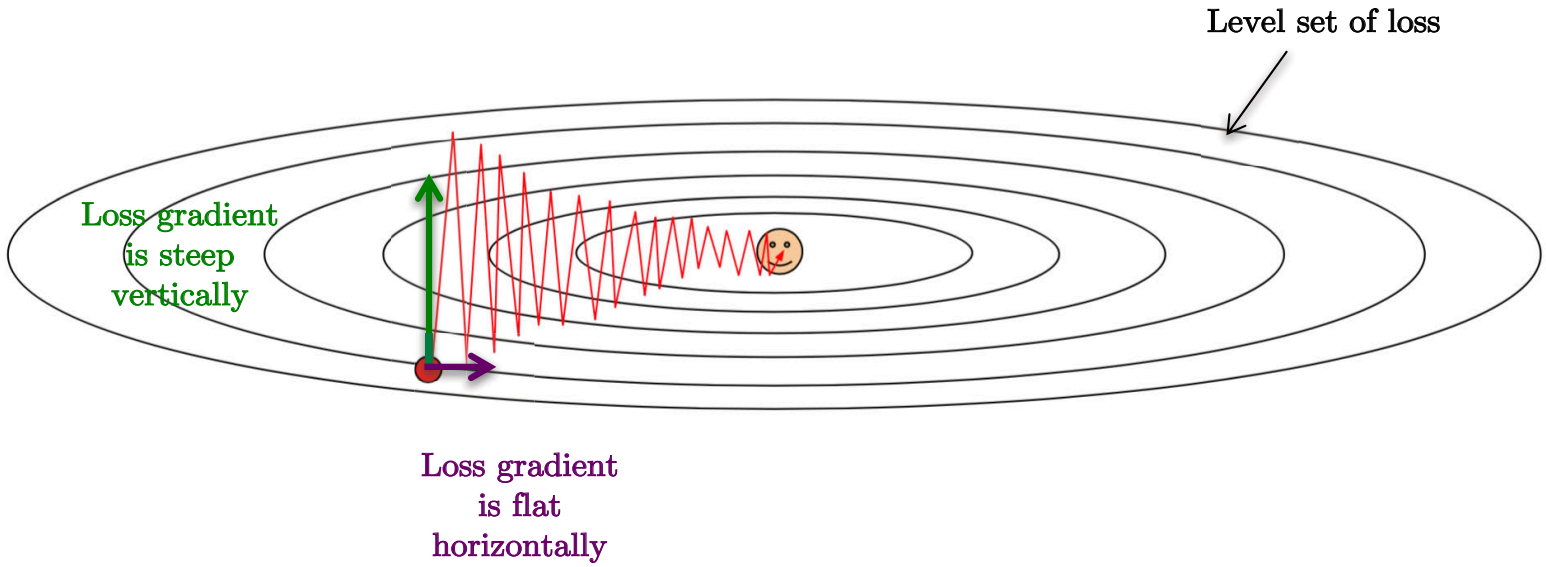
A: **Yes.** Extensive works but again, major bottleneck.



sgd is the slowest!

Why SGD is slow?

- **Illustration:**



Q: What is the trajectory along which we converge to the minimum with SGD?

A: Trajectory is **jittering** (up and down) because **very slow progress along flat directions**.

$\Rightarrow$ Solution: *Momentum update.*

Momentum [Hinton-et.al'86]

- **New update rule:**

```
# Gradient descent update
x += - learning_rate * dx
```

Friction

Acceleration

Velocity

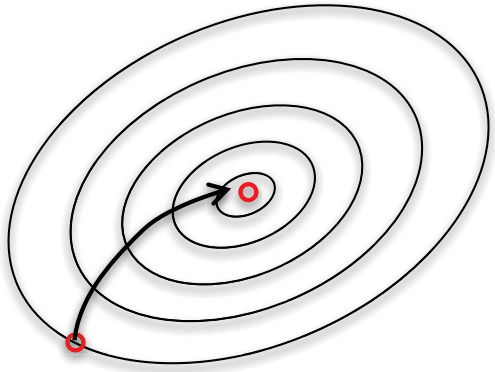
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Physical interpretation: *Momentum update is like rolling a ball along the landscape of the loss function.*

- **Advantages:**

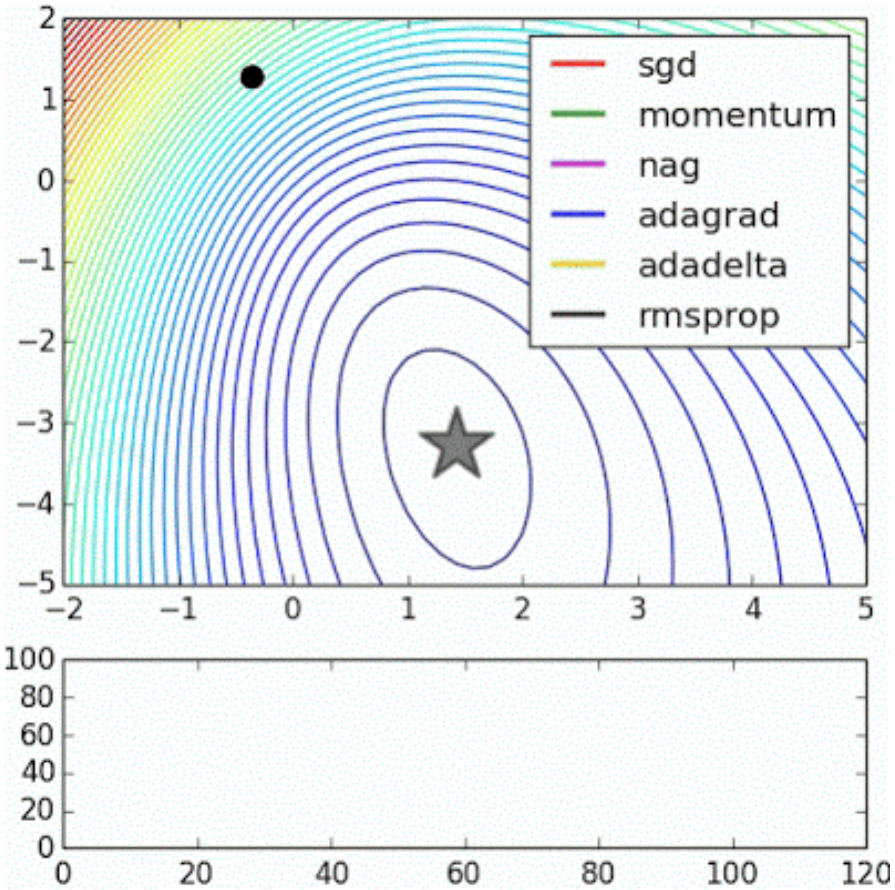
  (1) Velocity builds up along flat directions.

  (2) Decrease velocity in steep directions.



Limitation of Momentum

- Momentum **overshoots** the minimum (too much velocity) but overall gets faster than SGD.



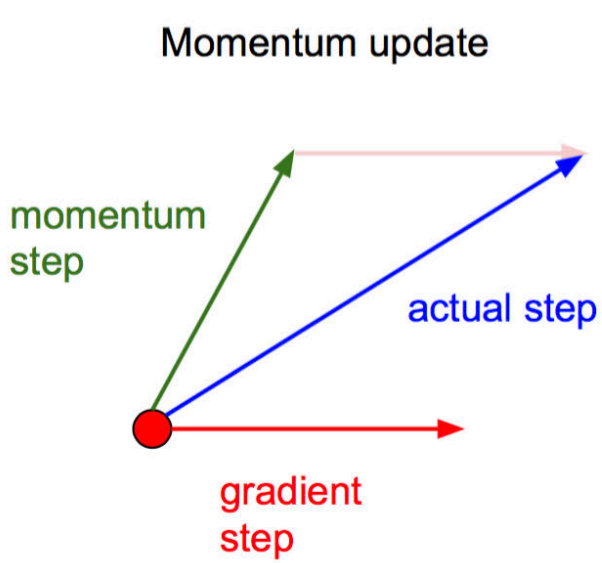
- **In practice:**

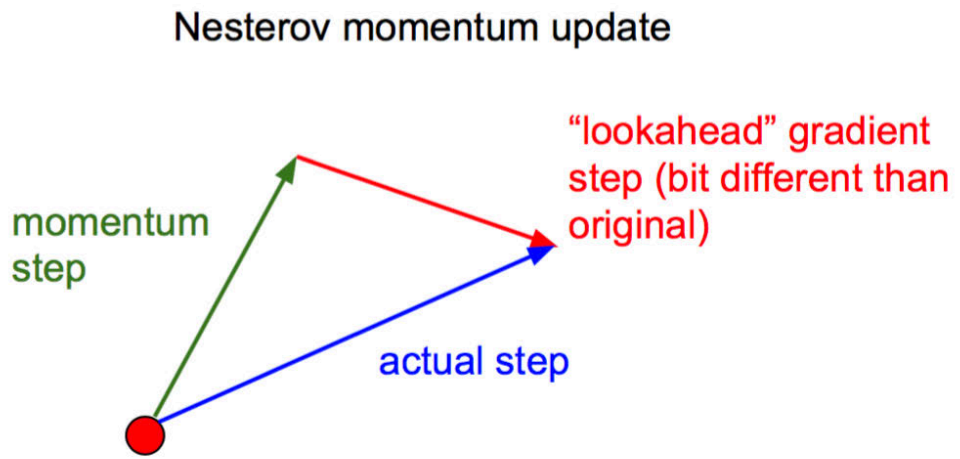
  (1) $\mu = 0.5, 0.9$

  (2) Initialization: v=0

Nesterov Momentum

- Nesterov accelerated gradient (NAG) technique used for momentum update:





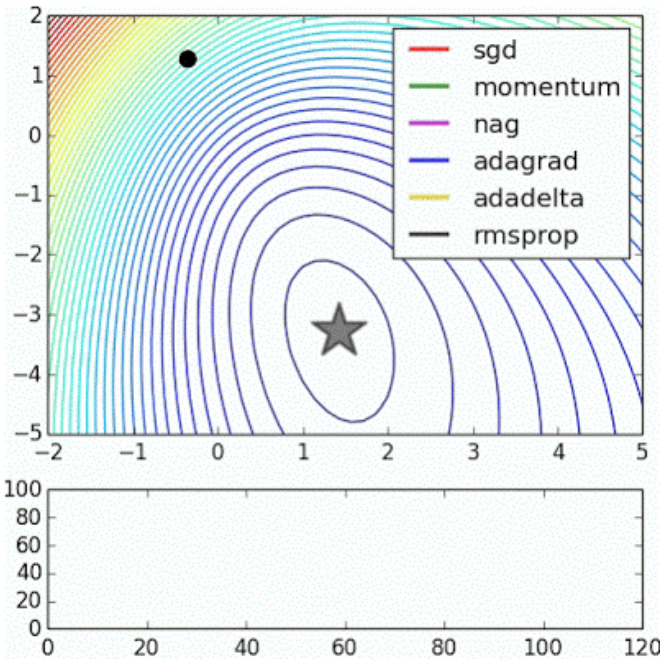
Nesterov update:

only change

$$v^t = \mu v^t - \tau \nabla f(x^t + \mu v^t)$$

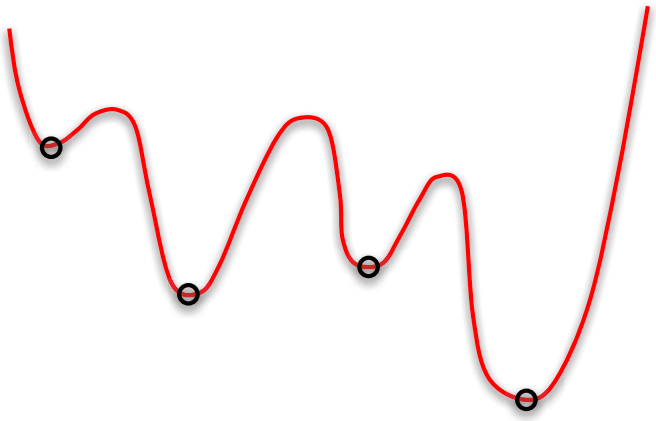
$$x^{t+1} = x^t + v^t$$

$\Rightarrow$ It can correct its trajectory faster than momentum



Energy Landscape of NNs

- Energy landscapes of NNs are non-convex
  $\Rightarrow$ Existence of local minimizers, which usually is a big issue as most local solutions are bad.



- Surprisingly, **most local minimizers in large-scale networks are satisfactory**! Why? Nobody knows.

  So, initialize with random function, then it will always end up to a good solution, **no** bad local minimizers!

- The problem of local minimizers is for small-scale networks.

AdaGrad [Duchi-et.al'11]

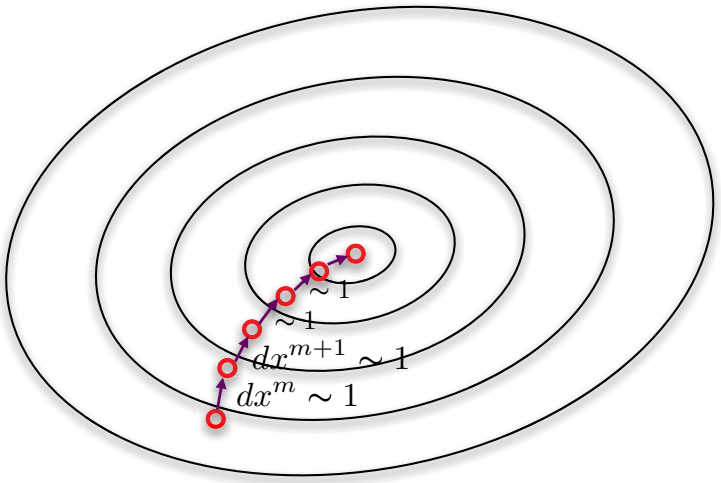
- **Origin:** Convex optimization.

- **Update rule:**

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Prevents division by 0.

$$\frac{dx^m}{\sqrt{|v^{n+1}|^2}} \approx \text{sign}(dx) = \pm 1$$



The dynamics is controlled because the **speed is ±1** ⇒ It equalizes the step size in steep and flat directions.

- **Issue:** When v^m gets large then x^m will **stop** moving!

  ⇒ Solution: RMSProp

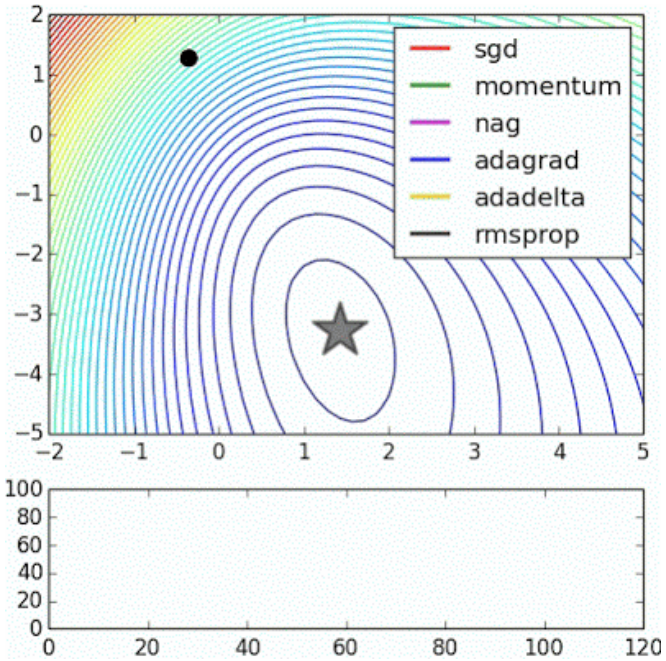
RMSProp [Hinton'12]

- **RMSProp update rule:** It does not stop the learning process.

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Adam [Kingma-Ba'14]

- **Adam = Momentum + Adagrad/RMSProp**

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

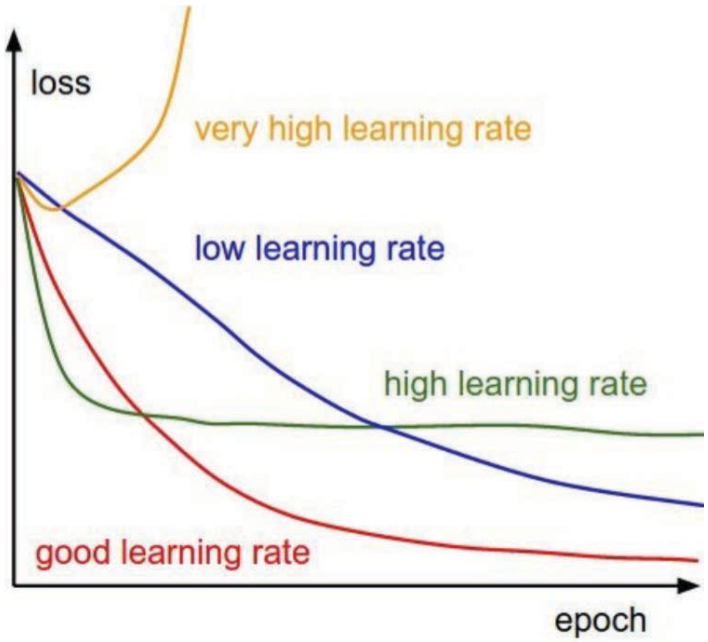
RMSProp-like

- Adam = Default current optimization technique for NNs.
- Hyperparamaters: $\beta_1 = 0.9$, $\beta_2 = 0.9$.

Global Learning Rate τ

- **All optimization algorithms have a learning rate τ as hyperparameter:**

$$W^{m+1} = W^m - \tau \nabla_W E(W^m)$$



- **Q: Is a constant learning rate is good?**

  A: No. A learning rate should decay over time.

  Decay learning rate at each epoch:

  (1) Exponential decay $\tau = \tau_0 e^{-\lambda_0 m}$

  (2) Polynomial decay $\tau = \tau_0 / (1 + \lambda_0 m)$

- **Common good practice:** Babysit the loss value and the learning rate
  ⇒ Later

Demo: Update Rules/ Neural Network Optimization

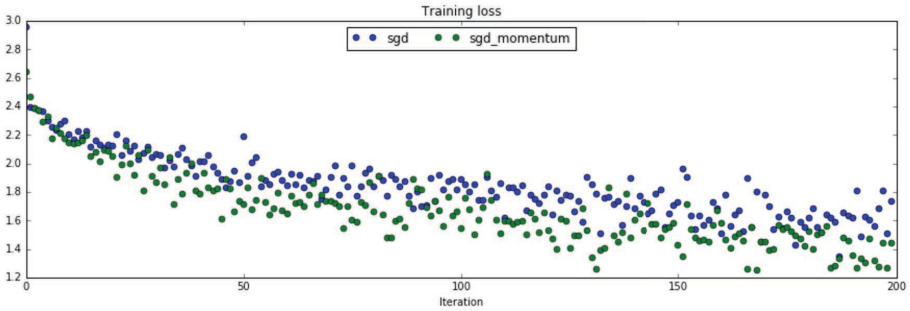
- Run **code03.ipynb**

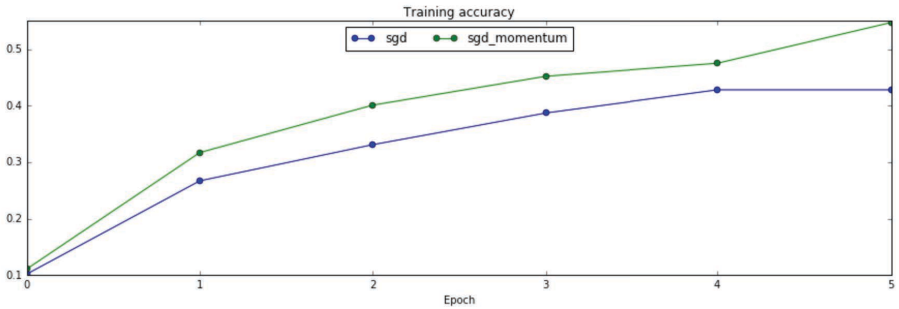
SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule
descent.

Open the file `lib/optim.py` and read the documentation at the top of th
rule in the function `sgd_momentum` and run the following to check your im

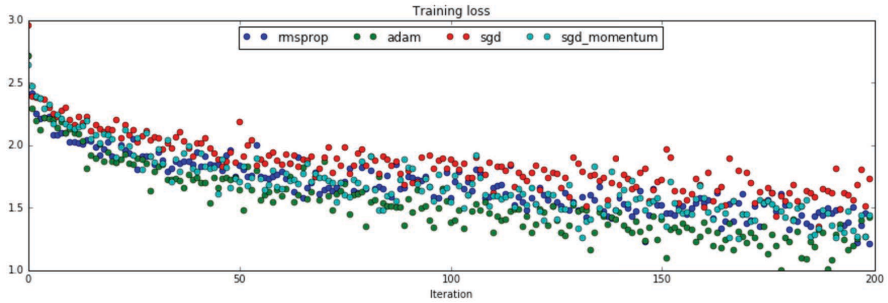
```
from optim import sgd_momentum
```

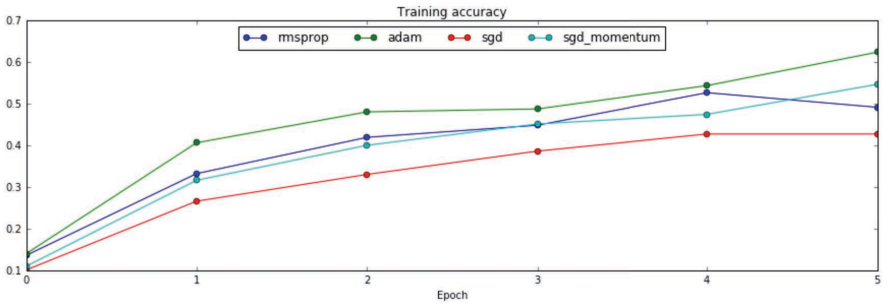




RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates





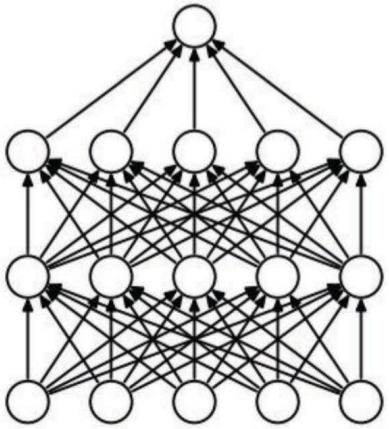
Outline

- Generic Gradient Descent Techniques
- Backpropagation
- Activation
- Weight Initialization
- Neural Network Optimization
- **Dropout**
- Conclusion

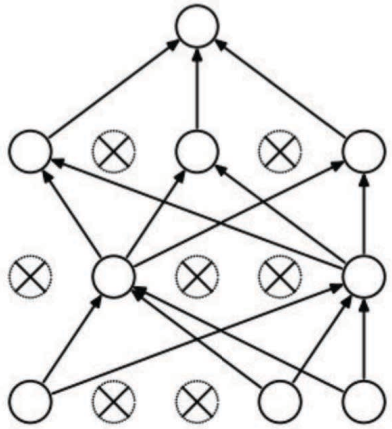
Dropout Regularization [Hinton'14]

- **Dropout:** A regularization process tailored to NNs

  Idea: *Randomly set some neurons to zero in the forward pass.*



(a) Standard Neural Net



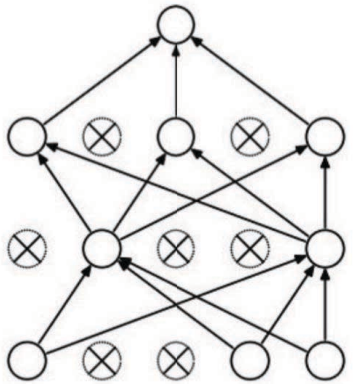
(b) After applying dropout.

- **How to deactivate neurons?**

  Use simple binary masks U.

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Why It Works?

- **It prevents overfitting:** It reduces the number of parameters to learn for the NN.

- **It increases robustness of learning process:** For each dropout unit, dropout **sub-samples** the NN and we learn the best weights for this sub-network. And we do it for **many different sub-networks**. Besides all these sub-networks **share weights** ⇒ **Global consistency** of weight values and better robustness because **we learn on smaller networks**.

Code

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

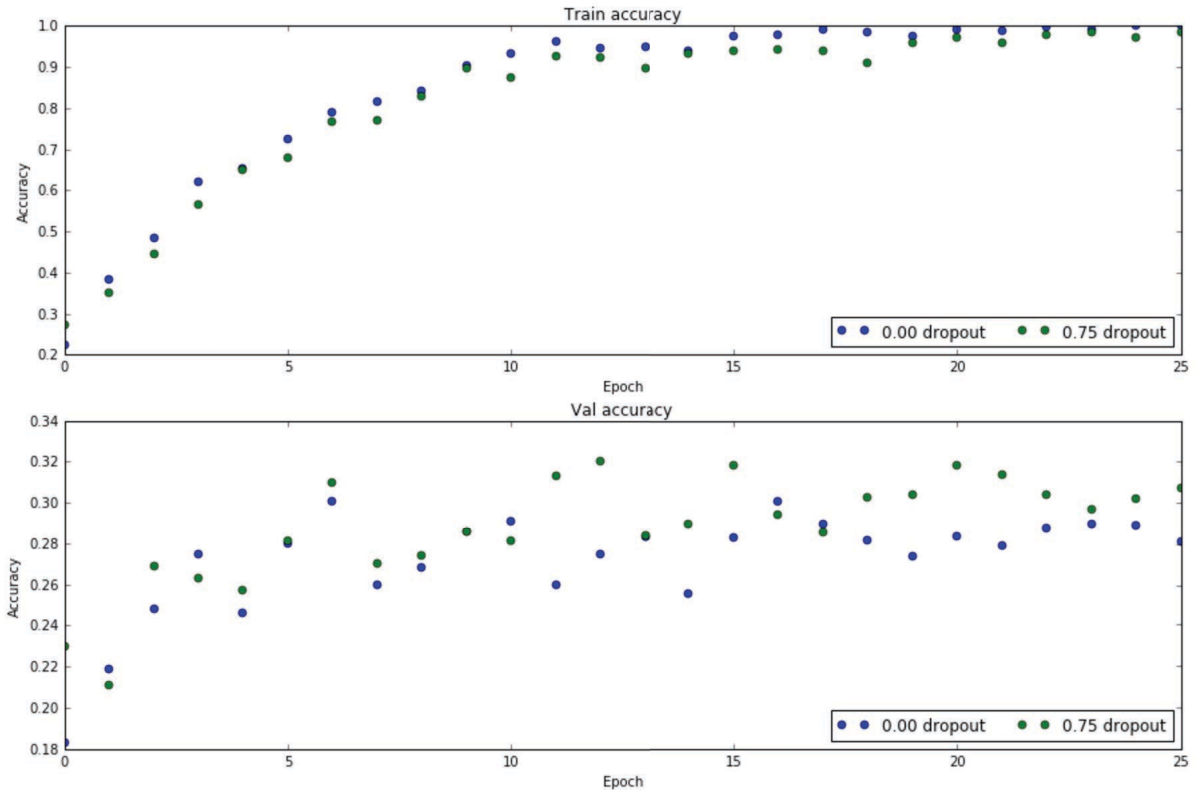
Demo: Dropout

- Run **code04.ipynb**

Dropout forward pass

In the file `lib/layers.py`, implement the forward pass for dropout. Since dropout behaves differently operation for both modes.

Once you have done so, run the cell below to test your implementation.

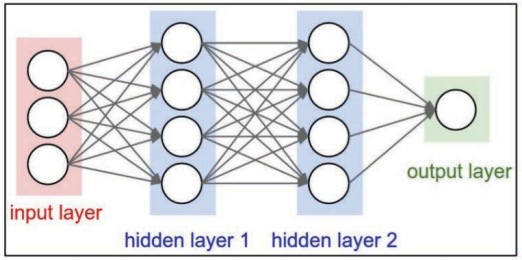


Outline

- Generic Gradient Descent Techniques
- Backpropagation
- Activation
- Weight Initialization
- Neural Network Optimization
- Dropout
- **Conclusion**

Summary

- **Training Neural Networks:**
  - *(1) Sample a batch of data.*
  - *(2) Forward prop it through the graph, get loss values*
  - *(3) Backprop to calculate the gradients*
  - *(4) Update the parameters using gradients*



- **Neural Networks = Computational Graphs**

  Lego approach of building large-scale NNs

- **Activation functions:**
  - *(1) Sigmoid* (try)
  - *(2) Tanh* (try)
  - *(3) ReLU* (default)
  - *(4) Leaky ReLU* (try)

Summary

- **Weight initializations:**
  - *(1) Xavier's initialization (default)*
  - *(2) Batch Normalization (30% additional cost)*

- **Parameter updates/optimization:**
  - *(1) SGD (default)*
  - *(2) Momentum*
  - *(3) Nesterov momentum*
  - *(4) Adagrad/RMSProp*
  - *(5) Adam (default)*

- **Dropout regularization**

Data Science Training

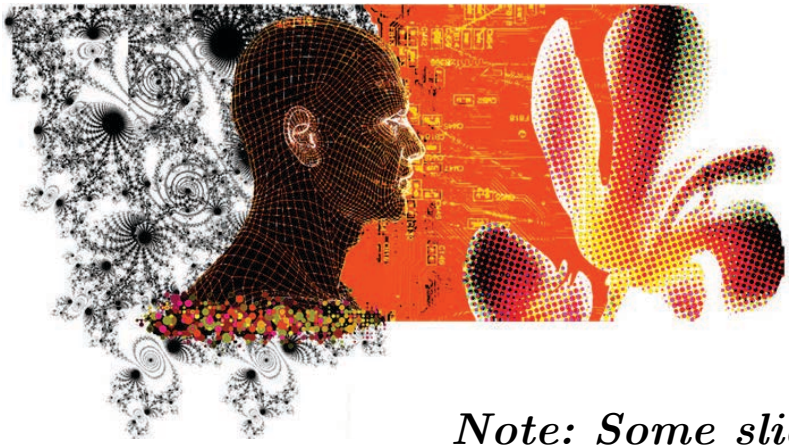
November 2017

Good Practices for NNs Training

Xavier Bresson

Data Science and AI Research Centre

NTU, Singapore



http://data-science-optum17.tk

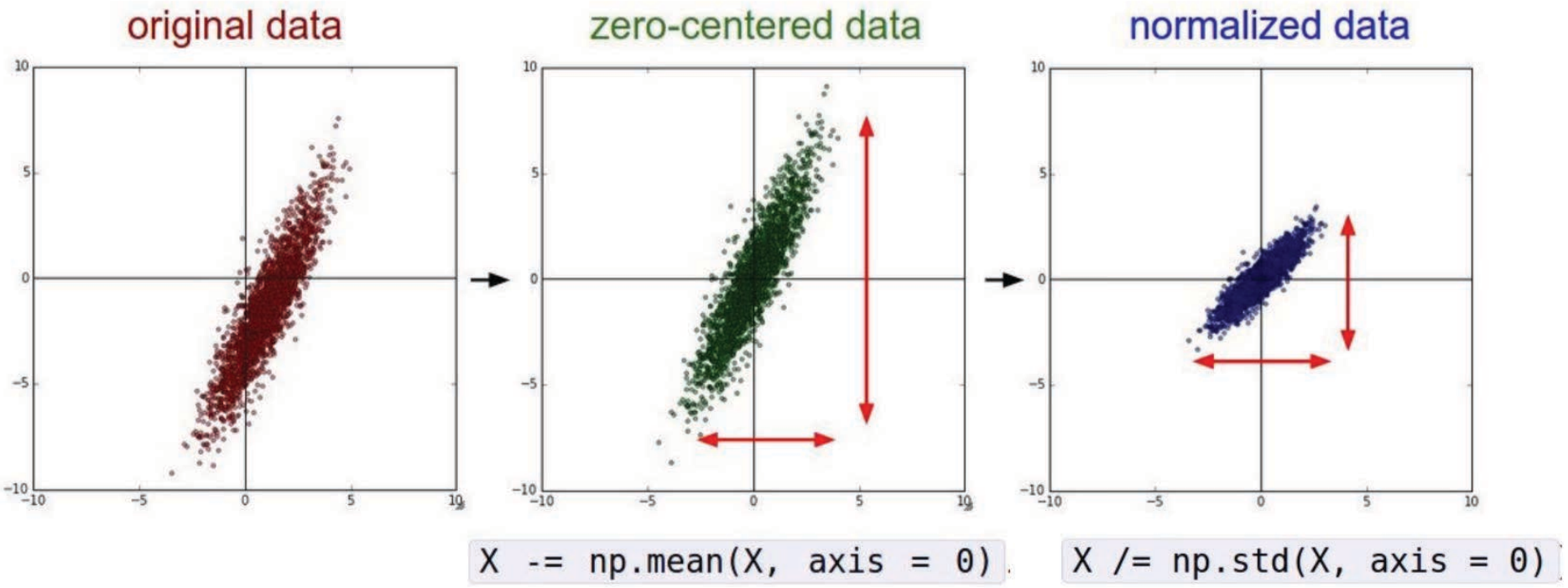
Note: Some slides are from Li, Karpathy, Johnson's course on Deep Learning.

Outline

- **Step 1: Pre-process data**
- Step 2: Choose NN Architecture
- Step 3: Monitor Loss Decrease
- Step 4: Hyperparameter Optimization
- Step 5: Monitor Test Accuracy

Step 1: Pre-Process Data

- *Assume a n x d data matrix X:*



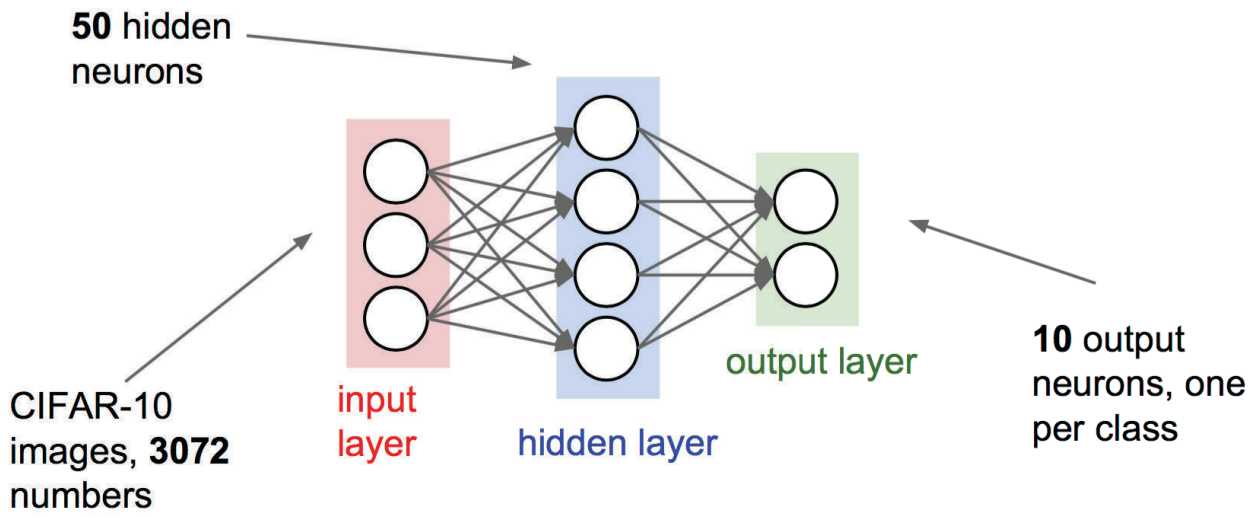
Outline

- Step 1: Pre-process data
- **Step 2: Choose NN Architecture**
- Step 3: Monitor Loss Decrease
- Step 4: Hyperparameter Optimization
- Step 5: Monitor Test Accuracy

Step 2: Choose NN Architecture

- **Start small:** 1 hidden layer then increase number of layers.

Example CIFAR:



- **Initialization:**

  Xavier's initialization

Outline

- Step 1: Pre-process data
- Step 2: Choose NN Architecture
- **Step 3: Monitor Loss Decrease**
- Step 4: Hyperparameter Optimization
- Step 5: Monitor Test Accuracy

Step 3: Monitor Loss Decrease

- **Initialization:** Loss value = - log(1/#classes)

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

- **Add regularization:** Loss value increases ⇒ Good sanity check

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

Step 3: Monitor Loss Decrease

- **Let us train:**

(1) First, overfit small portion of data with SGD, reg=0

⇒ Easy way to find a good value for the global learning rate τ.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

Very small loss, train accuracy 1.00, nice!

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Step 3: Monitor Loss Decrease

- **Let us train:**

(2) Second, reg=small then find τ

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

(3) Last, increase reg value

Outline

- Step 1: Pre-process data
- Step 2: Choose NN Architecture
- Step 3: Monitor Loss Decrease
- **Step 4: Hyperparameter Optimization**
- Step 5: Monitor Test Accuracy

Step 4: Hyperparameter Optimization

- **Cross-validation strategy:**

  (1) First step: Only use a few epochs to get an idea of what values work

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning rate=lr, verbose=False)
```

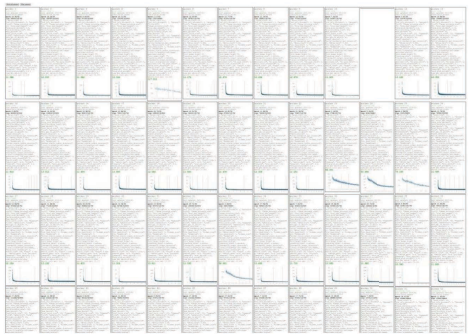
note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

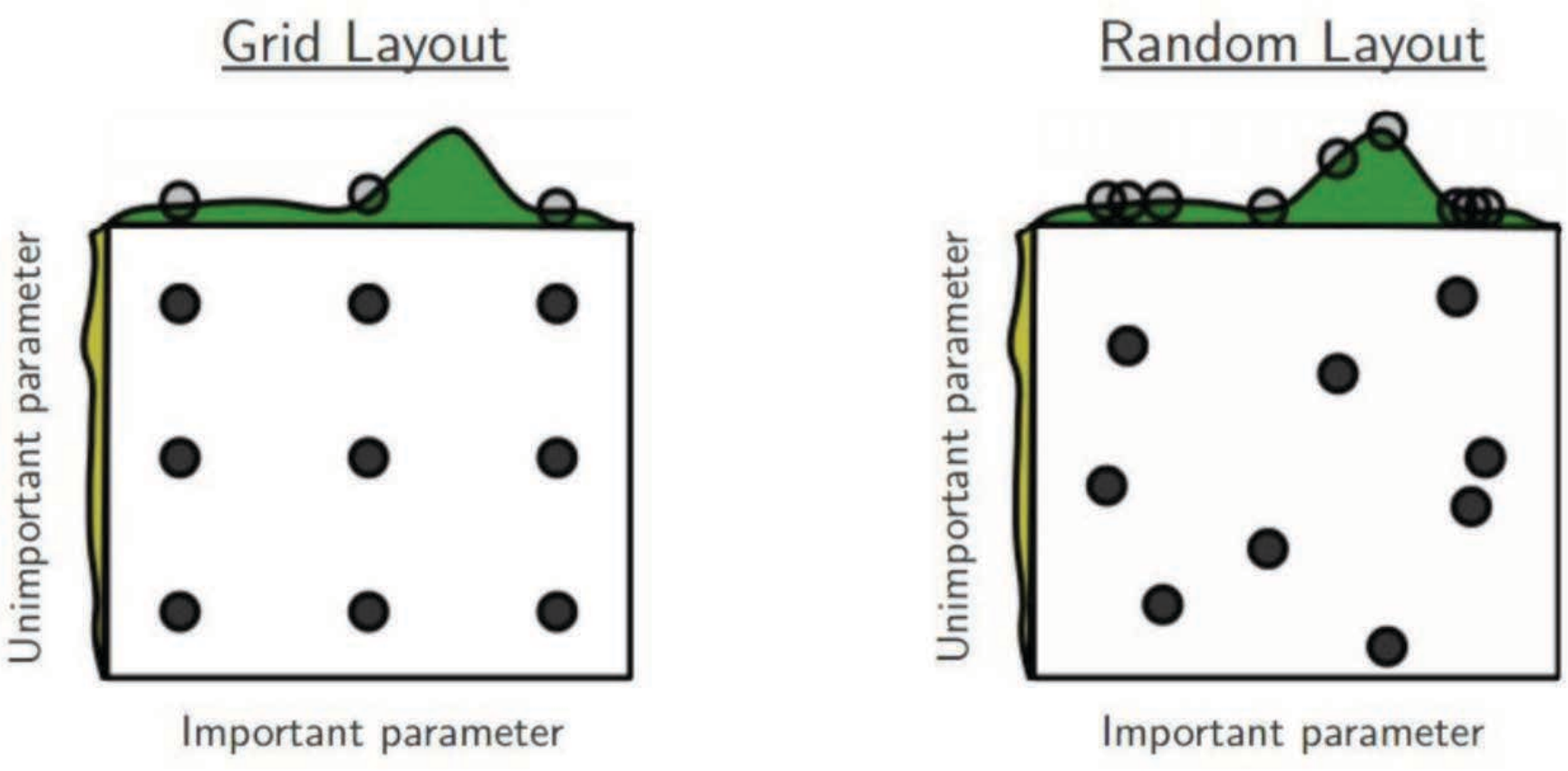
nice

(2) Second step: Try out many values

$\Rightarrow$ Very long computational times

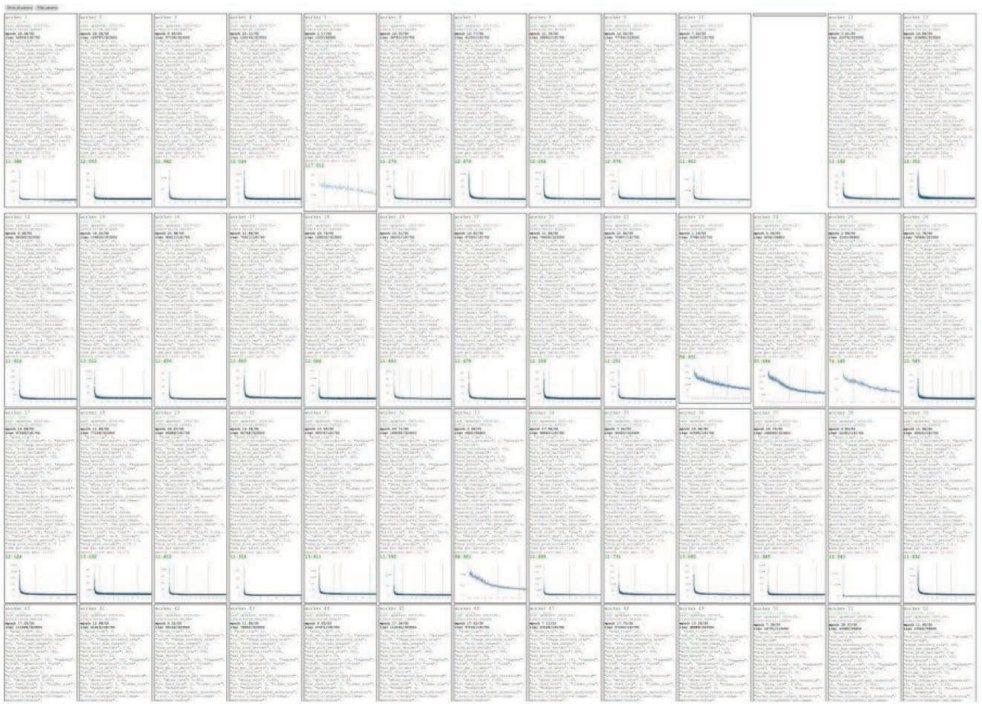


Grid vs. Random Search



Hyperparameters

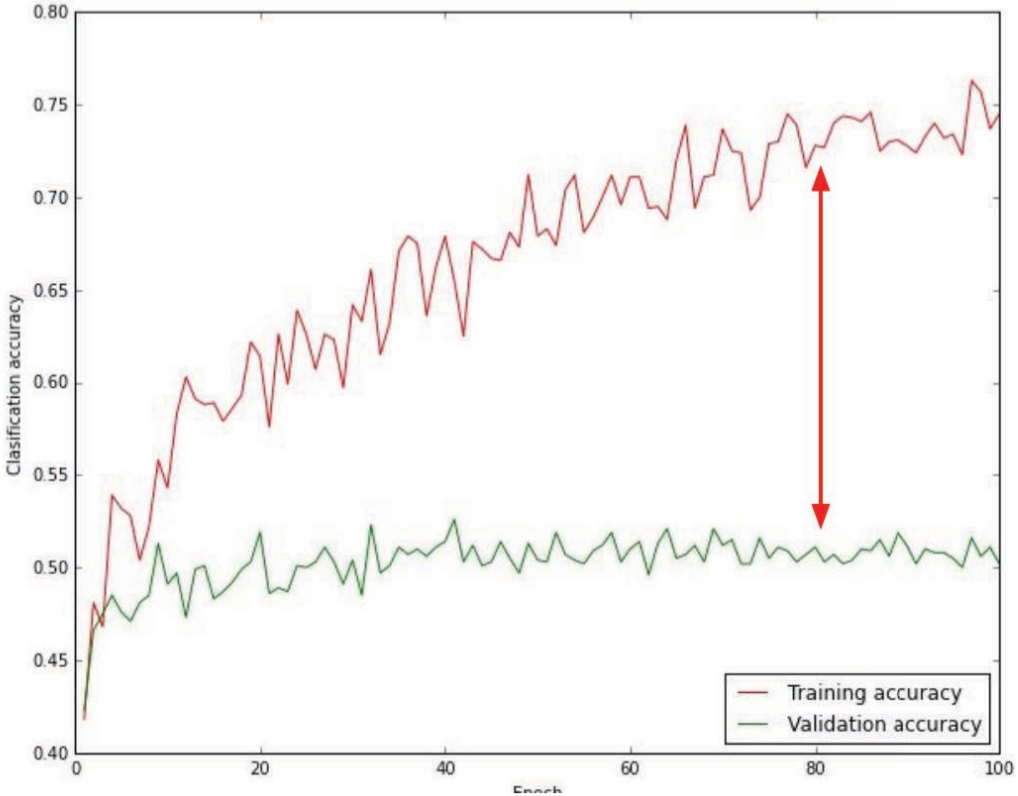
- **List:**
  (1) Network architecture
  (2) Learning rate, decay schedule
  (3) Regularization: L2 and dropout



Outline

- Step 1: Pre-process data
- Step 2: Choose NN Architecture
- Step 3: Monitor Loss Decrease
- Step 4: Hyperparameter Optimization
- **Step 5: Monitor Test Accuracy**

Step 5: Monitor Test Accuracy



big gap = overfitting

=> increase regularization strength

no gap

=> increase model capacity

Data Science Training

November 2017

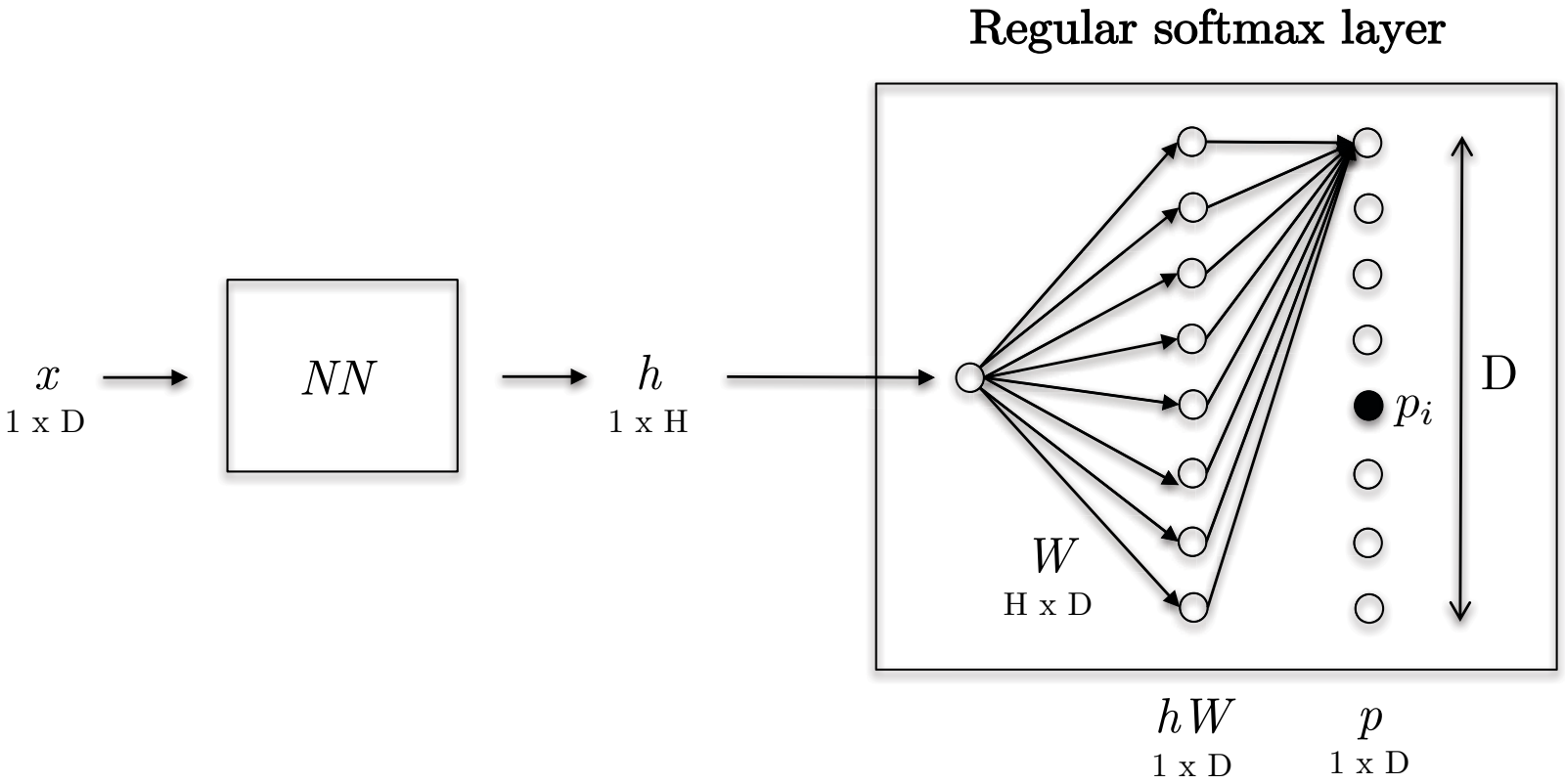
Learning Large Number of Classes

Xavier Bresson

Data Science and AI Research Centre
NTU, Singapore

http://data-science-optum17.tk

Standard Approach

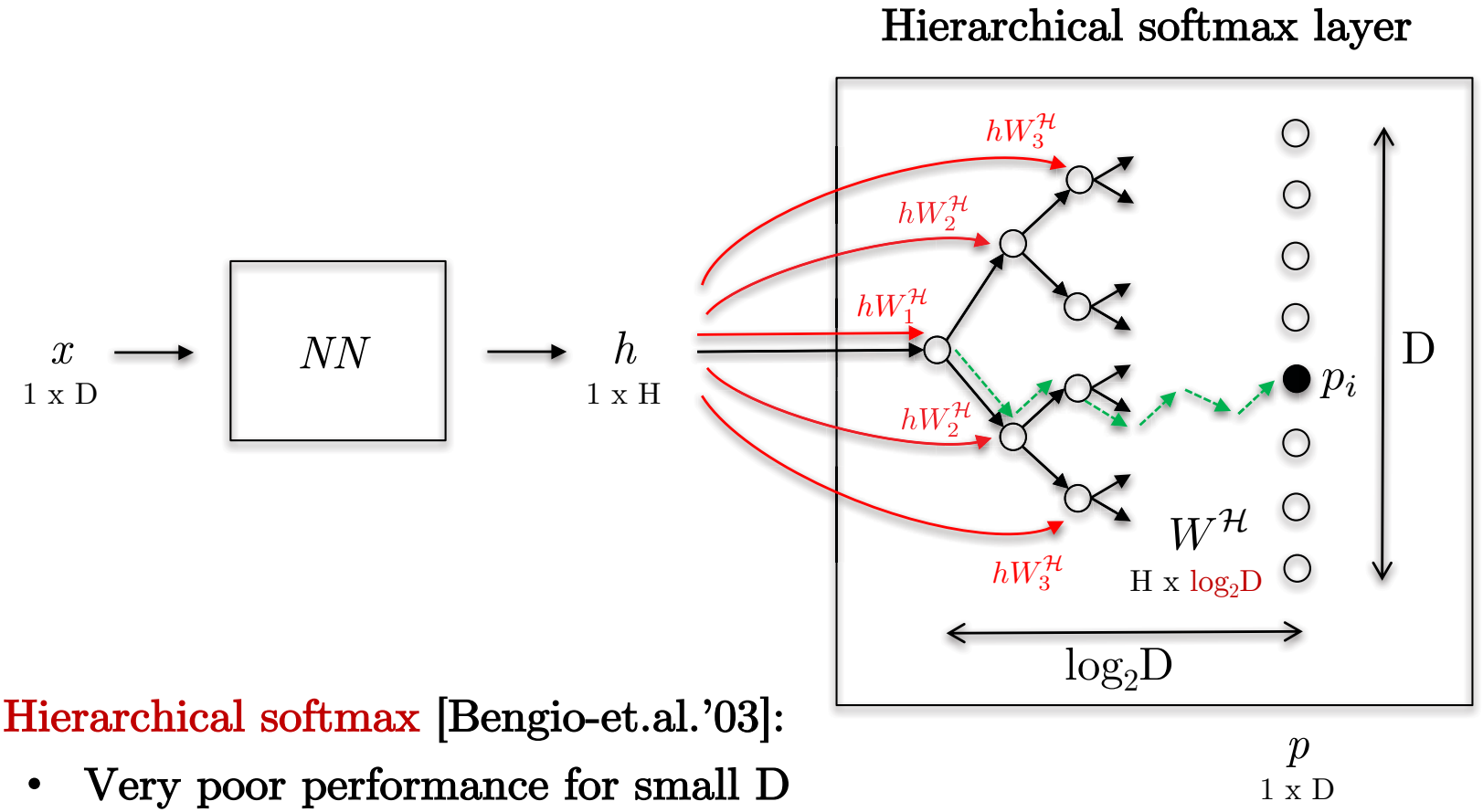


- Regular softmax:
  - Very good performance for small D
  - Very poor performance for large D

    Large #param (H.D) and large sum (D)

$$\text{with } p_i = \frac{e^{hW_i}}{\sum_{j=1}^{D} e^{hW_j}}$$

Hierarchical Softmax



- **Hierarchical softmax** [Bengio-et.al.'03]:
  - Very poor performance for small D
  - Very good performance for large D

    Small #param (H.$\log_2$D) and

    small product ($\log_2$D)

- Speed up: 258x

with $p_i = \Pi_{\ell=1}^{\log_2 D} \; p_{i,\ell}^{\mathcal{H}}$

and $p_{i,\ell}^{\mathcal{H}} = \sigma(\pm_{i,\ell} \; hW_\ell^{\mathcal{H}})$

$\sigma(-z) = 1 - \sigma(z)$

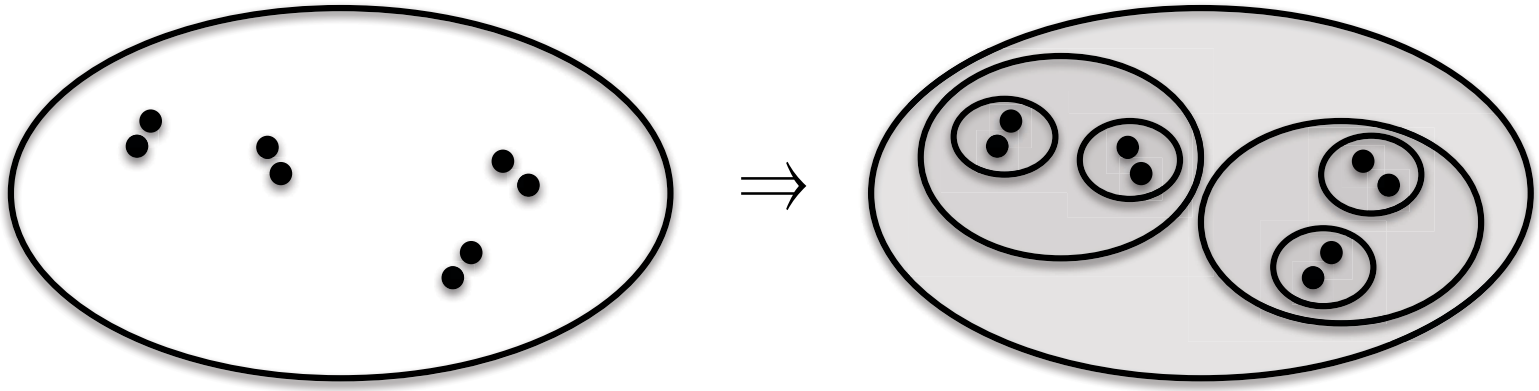
Word Hierarchy

- Define word hierarchy:
  1. Use random generated tree (average performance)
  2. Use domain expertise (average+ performance)
  3. Learn the hierarchy of words [Mnih-Hinton'08]:

     Step 1: Use random tree and learn the hierarchical model.

     Step 2: Represent each word with features $hW^{\mathcal{H}}$

     Step 3: Apply recursive bi-partitioning algorithm to derive the tree.



Questions?