Lecture Notes for

Machine Learning in Python

⟦👨‍🏫, 👨‍💻, 🐍, 👨‍🔬⟧

Professor Eric Larson

Neural Network Optimization and Activation

Class Logistics and Agenda

- Logistics
    - grading update (and AWS, quizzes, etc.)
    - flipped module next time! Cross validation.
- Agenda:
    - More optimization techniques and programming examples
        - ~~Momentum~~
        - Objective Functions
        - Initialization
        - More activations: Tanh, ReLU, SiLU
        - Adaptive learning: AdaGrad, AdaM, etc.

Class Overview, by topic

Table Data Visualization
Numpy, Pandas, Seaborn
Overviews with some in-depth discussion

Dimension Reduction and Image Processing
Scikit-learn, Scikit Image,
Intuition only, Some mathematics

Linear and Logistic Regression
Numpy, Recreate API for Scikit-learn
Detailed mathematics for simple optimization
intuition for advanced optimization

Neural Networks and Back Prop.
Numpy
Detailed mathematics for NN operations

Wide and Deep Networks | **Convolutional Networks** | **Recurrent Networks**
Keras, Tensorflow
Intuition, Detailed implement.

Ethics in Language Models
ConceptNet
Case studies

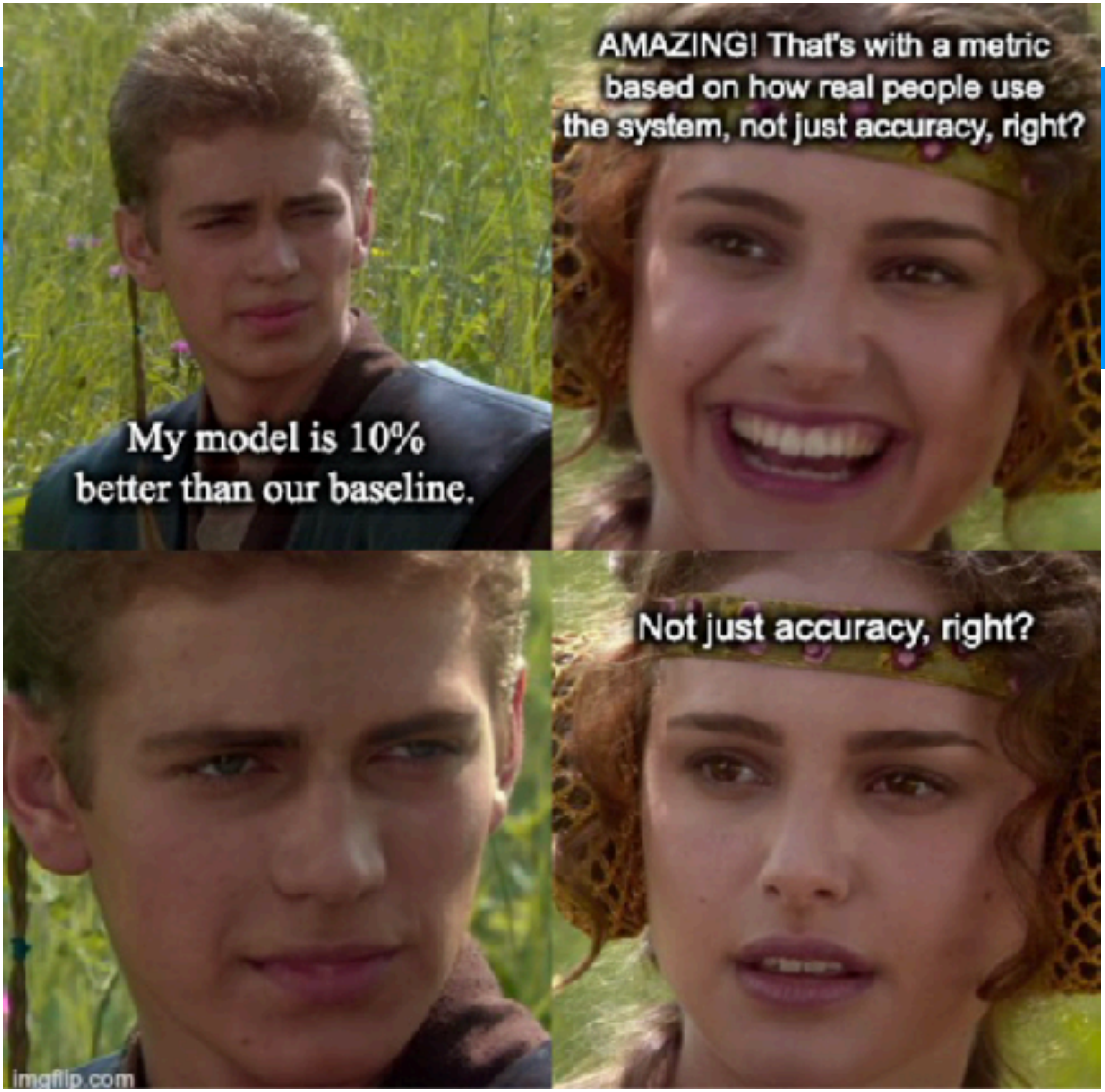
```
08a. Practical_NeuralNetsWithBias.ipynb
```

Quick Review:

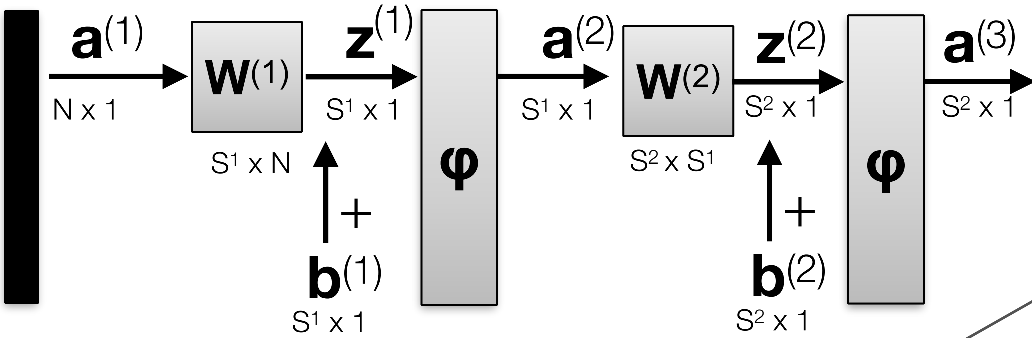
Momentum

Learning Rate Adaptation

Objective Function



Changing the Objective Function



$$\mathbf{V}^{(2)} = -2(\mathbf{Y} - \mathbf{A}^{(3)}) * \mathbf{A}^{(3)} * (1 - \mathbf{A}^{(3)})$$

$$\nabla^{(2)} = \mathbf{V}^{(2)} \cdot [\mathbf{A}^{(2)}]^T$$

$$\mathbf{V}^{(1)} = \mathbf{A}^{(2)} * (1 - \mathbf{A}^{(2)}) * [\mathbf{W}^{(2)}]^T \cdot \mathbf{V}^{(2)}$$

$$\nabla^{(1)} = \mathbf{V}^{(1)} \cdot [\mathbf{A}^{(1)}]^T$$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \nabla^{(l)}$$

1. Forward propagate to get **Z, A**
2. Get final layer gradient
3. Back propagate sensitivities
4. Update each $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$

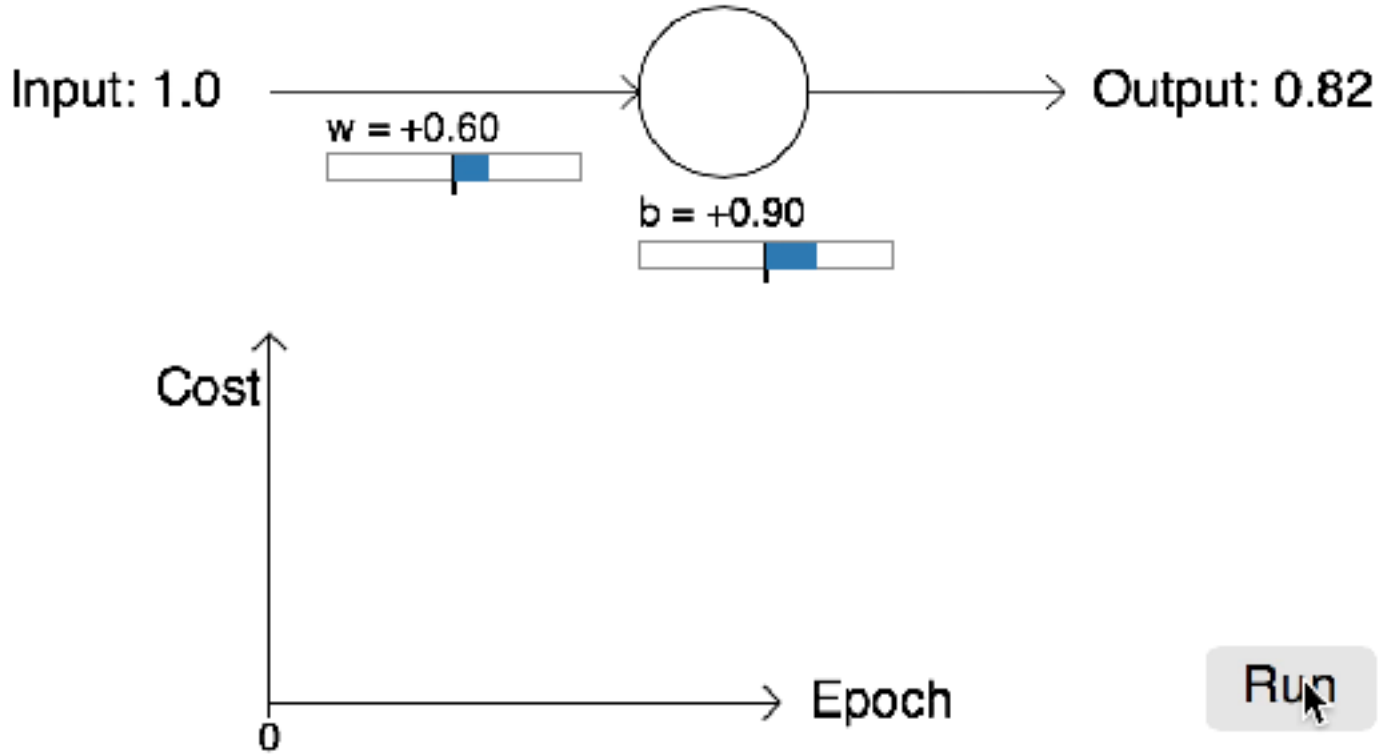
- **Self Test**:
  True or False: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer sensitivity calculation, $\mathbf{V}^{(2)}$, of the back propagation steps. The remainder of the algorithm is unchanged.
  - A. True
  - B. False

Practical Implementation of Architectures

- Mean squared error:

$$J(\mathbf{W}) = \sum_{k}^{M} (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

least squares objective,
tends to slow training initially



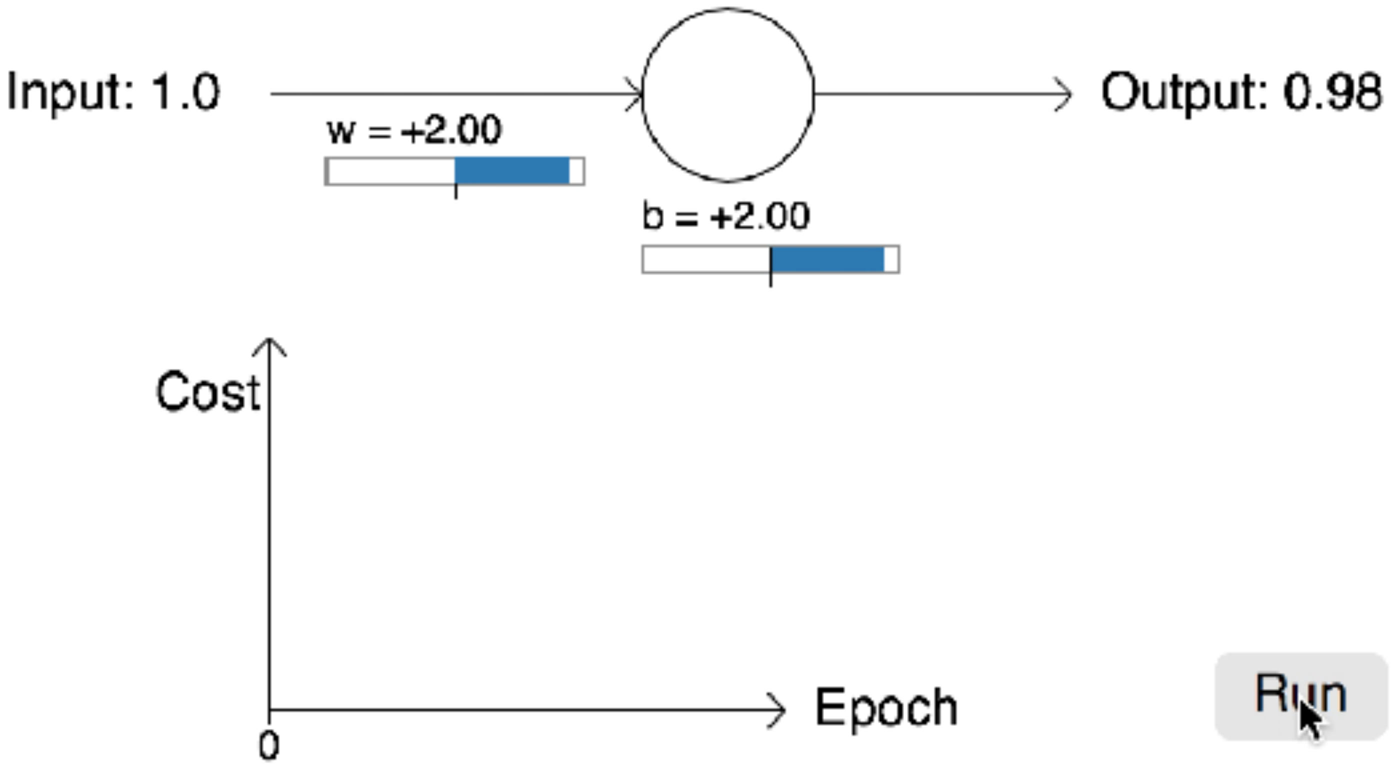
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- Mean squared error:

$$J(\mathbf{W}) = \sum_{k}^{M} (\mathbf{y}^{(k)} - \mathbf{a}^{(L)})^2$$

least squares objective, tends to slow training initially



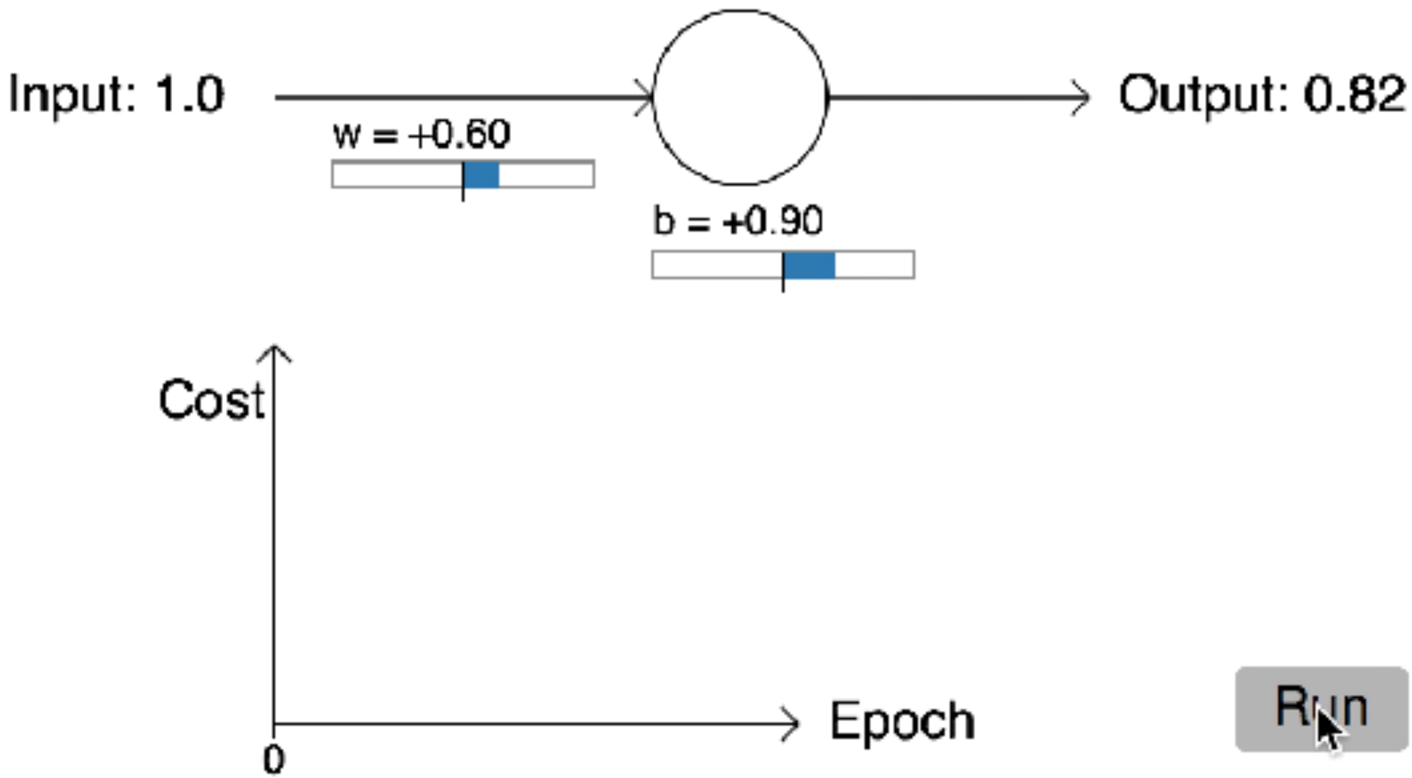
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- Our old friend, **Binary Cross entropy**

$$J(\mathbf{W}) = -\left[\mathbf{y}^{(i)} \ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1-\mathbf{y}^{(i)})\ln(1-[\mathbf{a}^{(L+1)}]^{(i)})\right]$$

speeds up initial training



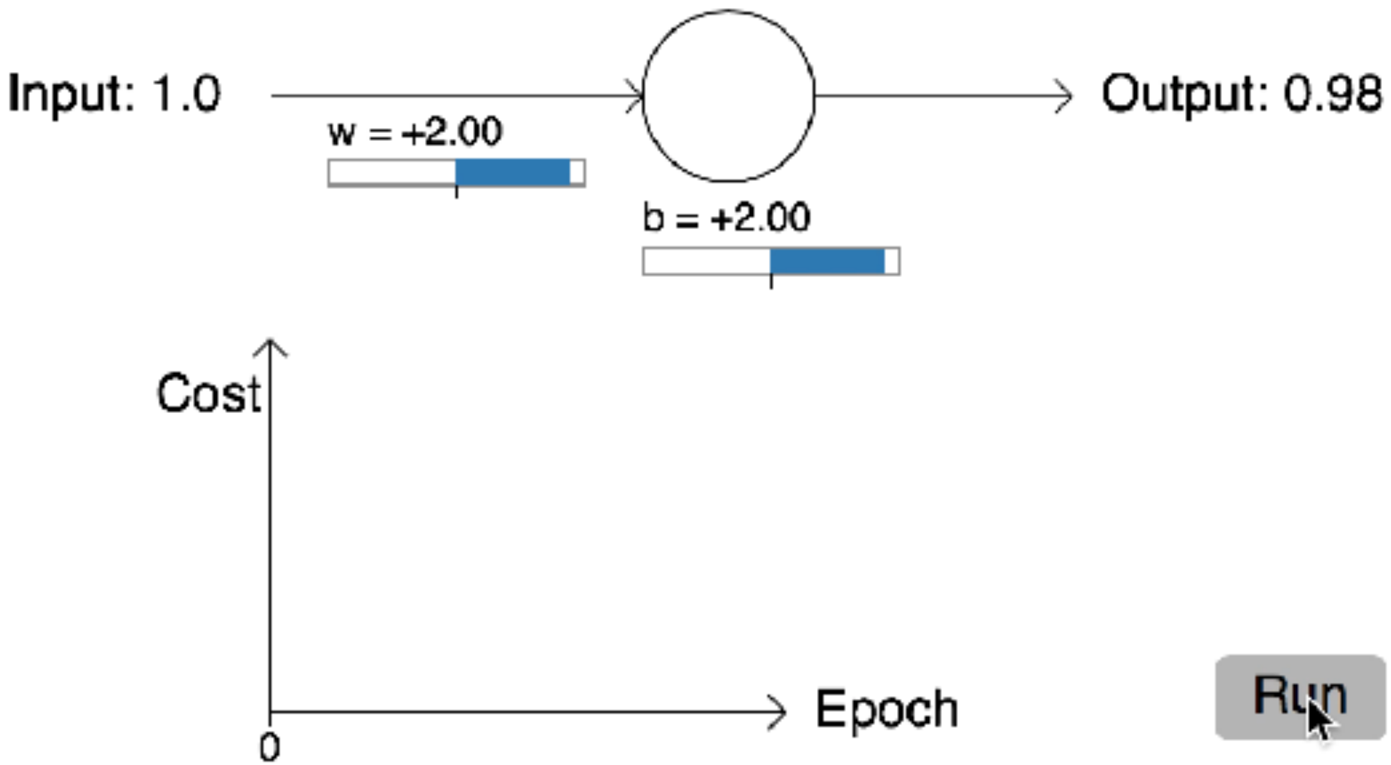
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- Negative of MLE: **Binary Cross entropy**

$$J(\mathbf{W}) = -\left[\mathbf{y}^{(i)} \ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1-\mathbf{y}^{(i)})\ln(1-[\mathbf{a}^{(L+1)}]^{(i)})\right]$$

speeds up initial training



Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

$$J(\mathbf{W}) = -\left[\mathbf{y}^{(i)}\ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1-\mathbf{y}^{(i)})\ln(1-[\mathbf{a}^{(L+1)}]^{(i)})\right]$$

likely to speed up initial training

$$\left[\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(L)}}\right]^{(i)} = -\frac{\partial}{\partial \mathbf{z}^{(L)}}\left[\mathbf{y}^{(i)}\ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1-\mathbf{y}^{(i)})\ln(1-[\mathbf{a}^{(L+1)}]^{(i)})\right]$$

only $\mathbf{a}$ has dependence on $\mathbf{z}$

$$= -\left[\mathbf{y}^{(i)}\frac{\partial}{\partial \mathbf{z}^{(L)}}\left(\ln([\mathbf{a}^{(L+1)}]^{(i)})\right) + (1-\mathbf{y}^{(i)})\frac{\partial}{\partial \mathbf{z}^{(L)}}\left(\ln(1-[\mathbf{a}^{(L+1)}]^{(i)})\right)\right]$$

$$= -\left[\mathbf{y}^{(i)}\frac{1}{[\mathbf{a}^{(L+1)}]^{(i)}}\left(\frac{\partial}{\partial \mathbf{z}^{(L)}}[\mathbf{a}^{(L+1)}]^{(i)}\right) + \frac{(1-\mathbf{y}^{(i)})}{1-[\mathbf{a}^{(L+1)}]^{(i)}}\left(-\frac{\partial}{\partial \mathbf{z}^{(L)}}[\mathbf{a}^{(L+1)}]^{(i)}\right)\right]$$

$$= -\left[\mathbf{y}^{(i)}\frac{1}{\cancel{[\mathbf{a}^{(L+1)}]^{(i)}}}\left(\cancel{[\mathbf{a}^{(L+1)}]^{(i)}}(1-[\mathbf{a}^{(L+1)}]^{(i)})\right) - \frac{(1-\mathbf{y}^{(i)})}{\cancel{1-[\mathbf{a}^{(L+1)}]^{(i)}}}\left([\mathbf{a}^{(L+1)}]^{(i)}\cancel{(1-[\mathbf{a}^{(L+1)}]^{(i)})}\right)\right]$$

$$= -\left[\mathbf{y}^{(i)}\left(1-[\mathbf{a}^{(L+1)}]^{(i)}\right) - (1-\mathbf{y}^{(i)})\left([\mathbf{a}^{(L+1)}]^{(i)}\right)\right]$$

$$= -\left[\mathbf{y}^{(i)} - \cancel{\mathbf{y}^{(i)}[\mathbf{a}^{(L+1)}]^{(i)}} - [\mathbf{a}^{(L+1)}]^{(i)} + \cancel{[\mathbf{a}^{(L+1)}]^{(i)}\mathbf{y}^{(i)}}\right] \qquad = [\mathbf{a}^{(L+1)}]^{(i)} - \mathbf{y}^{(i)}$$

$$\mathbf{V}^{(2)} = -2(\mathbf{Y}-\mathbf{A}^{(3)})\odot\mathbf{A}^{(3)}\odot(1-\mathbf{A}^{(3)})$$ old update

Practical Implementation of Architectures

- Back to our old friend: **Cross entropy**

$$J(\mathbf{W}) = -\left[\mathbf{y}^{(i)}\ln([\mathbf{a}^{(L+1)}]^{(i)}) + (1-\mathbf{y}^{(i)})\ln(1-[\mathbf{a}^{(L+1)}]^{(i)})\right]$$

likely to speed up initial training

$$\left[\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(L)}}\right]^{(i)} = [\mathbf{a}^{(L+1)}]^{(i)} - \mathbf{y}^{(i)}$$

$$\left[\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}}\right]^{(i)} = [\mathbf{a}^{(3)}]^{(i)} - \mathbf{y}^{(i)}$$

two layer network

```
# vectorized backpropagation
V2 = (A3-Y_enc) # <- this is only line t
V1 = A2*(1-A2)*(W2.T @ V2)
```

$$\mathbf{V}^{(2)} = \mathbf{A}^{(3)} - \mathbf{Y}$$

new update

bp-5

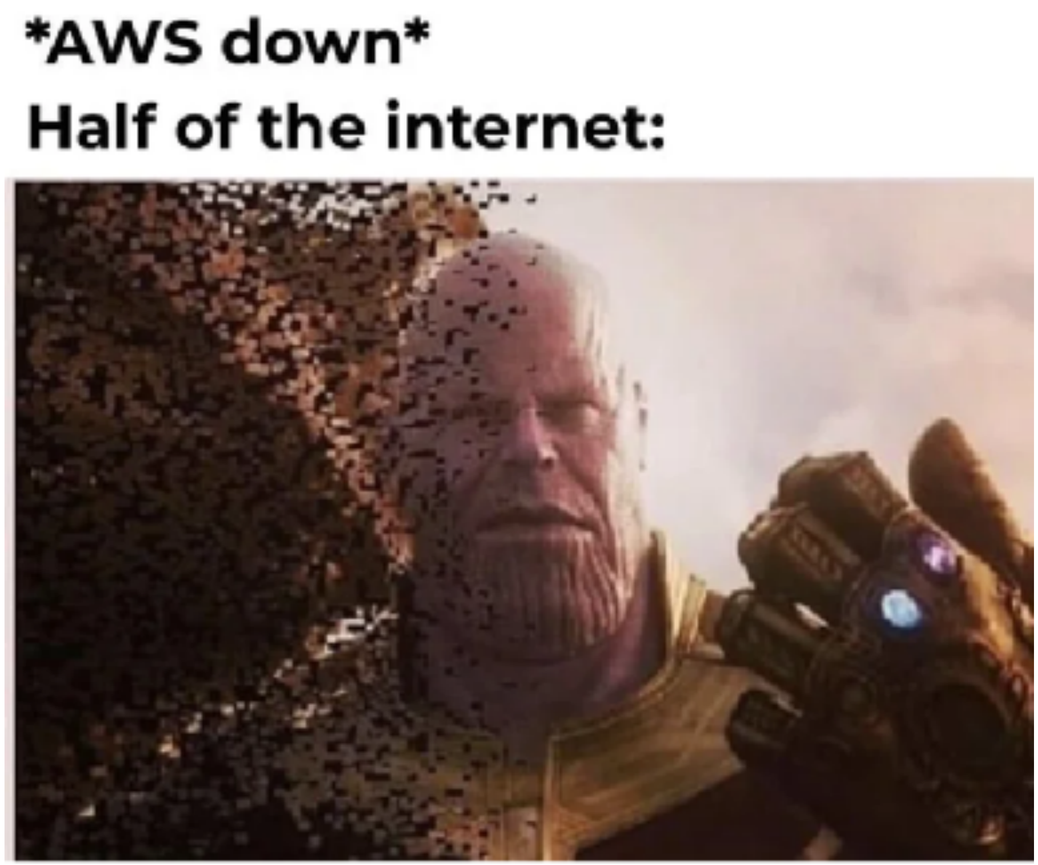
$$\mathbf{V}^{(2)} = -2(\mathbf{Y} - \mathbf{A}^{(3)}) \odot \mathbf{A}^{(3)} \odot (1 - \mathbf{A}^{(3)})$$ old update

Demo

08a. Practical_NeuralNetsWithBias.ipynb

~~Momentum~~
~~Learning Rate Adaptation~~
Cross Entropy

Practical Initialization of Architectures

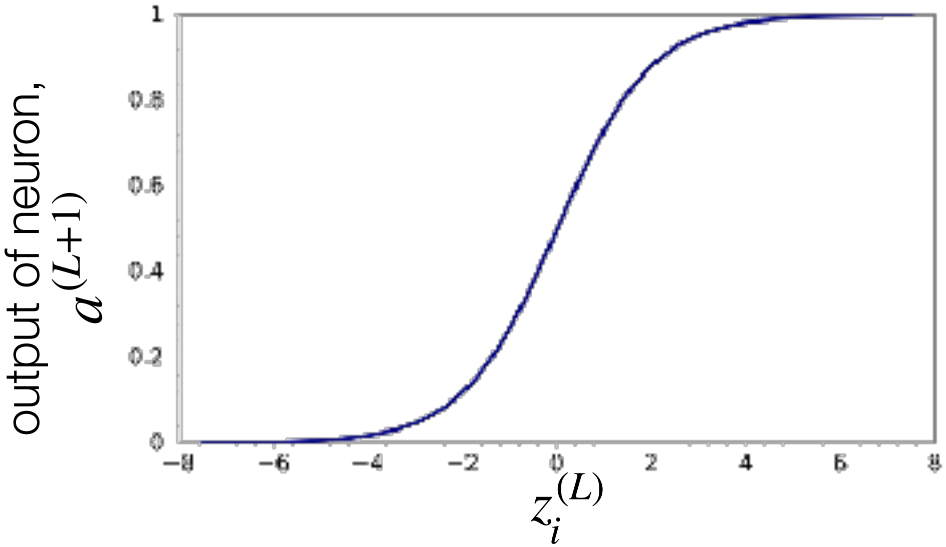


Formative Self Test

- for adding Gaussian random variables, variances add together

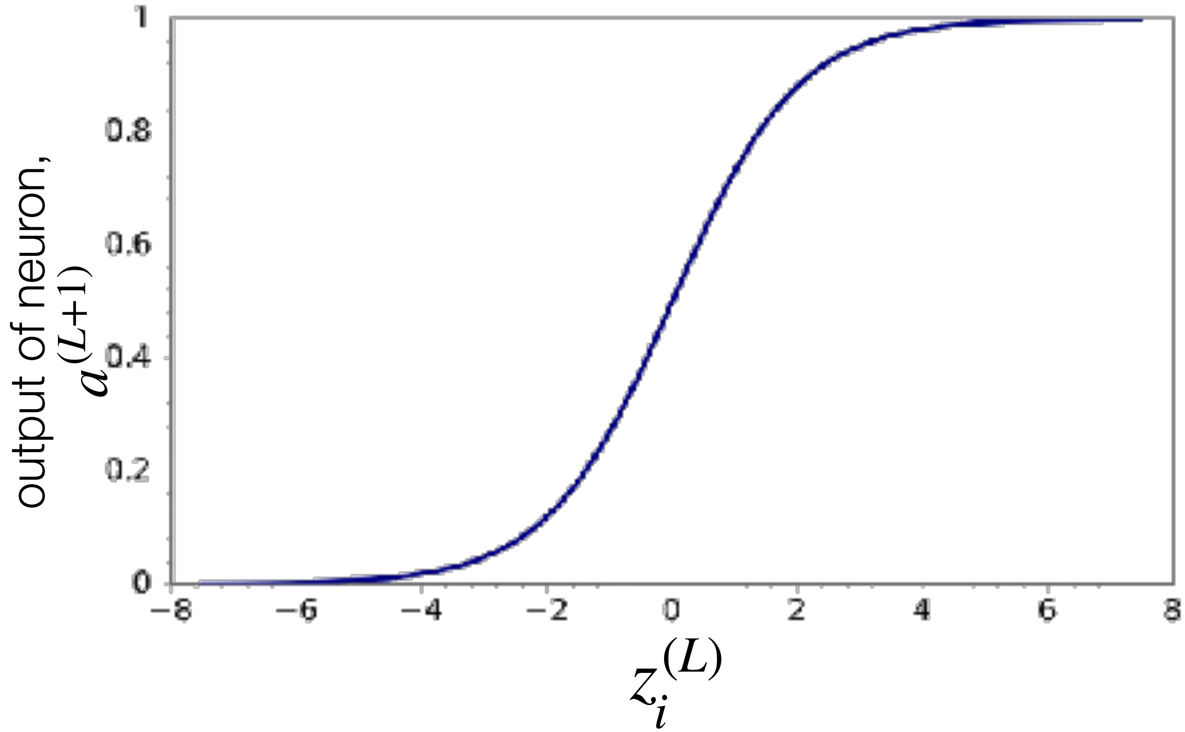
$$\mathbf{a}^{(L+1)} = \phi(\mathbf{W}^{(L)}\mathbf{a}^{(L)})$$ assume each element of $\mathbf{a}^{(L)}$ is Gaussian

- If you initialized the weights, $\mathbf{W}$, with too large variance, you would expect the output of the neuron, $\mathbf{a}^{(L+1)}$, to be:
  - A. saturated to “1”
  - B. saturated to “0”
  - C. could either be saturated to “0” or “1”
  - D. would not be saturated



Formative Self Test

- What is the derivative of a saturated sigmoid neuron?
  - A. zero
  - B. one
  - C. $a \times (1 - a)$
  - D. it depends

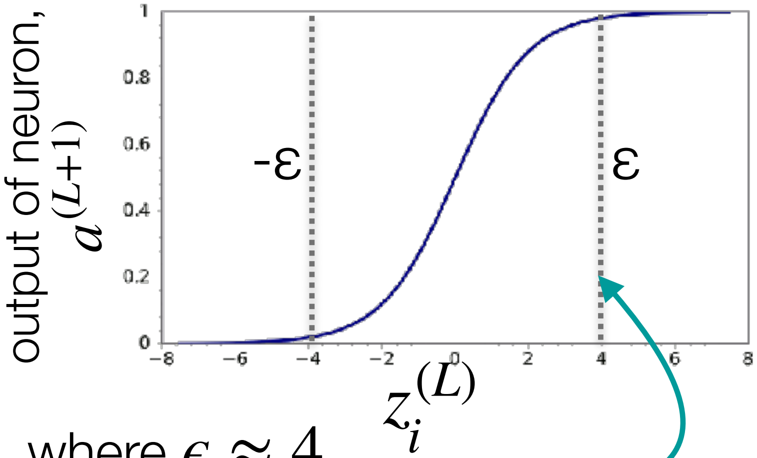


Practical Implementation of Architectures

Weight initialization: try not to saturate your neurons right away!

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)} \mathbf{a}^{(L)} \qquad z_i^{(L)} = \sum_j^{n^{(L)}} w_{ij} a_j^{(L)}$$

each row is summed before sigmoid



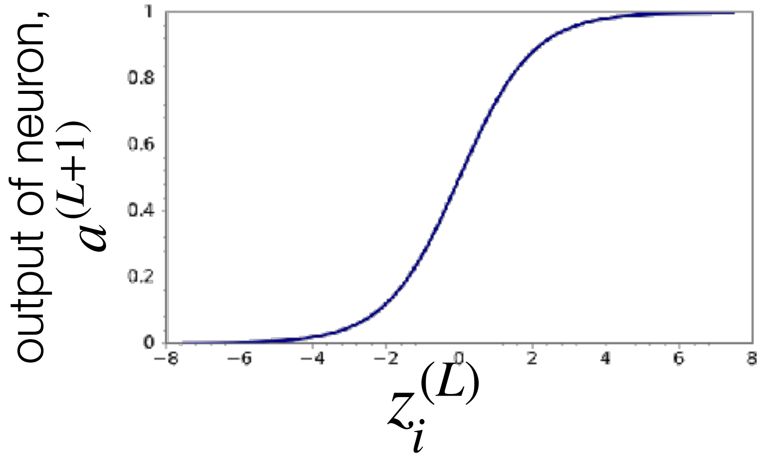
- want each $-\epsilon < z_i^{(L)} < \epsilon$ for no saturation, where $\epsilon \approx 4$
- if the Std$[z_i^{(L)}] \approx 4$ then the Std$[a_i^{(L+1)}] \approx 1$, which we want
  - because $\sigma(z_i^{(L)}) = a_i^{(L+1)}$, so it will be well distributed from [0,1]
- since $z_i^{(L)} = \sum w_{ij} a_j^{(L)}$, we want to "squash" initial weight magnitudes
- how do we squash w_{ij}, such that Std$[z_i^{(L)}] \approx 4$?
- **One solution**: draw w_{ij} from a Normal Distribution, $w_{ij} \sim \mathcal{N}(0,\alpha)$
  - solve for a good α such that **Goal**: Std$[z_i^{(L)}] \approx 4$

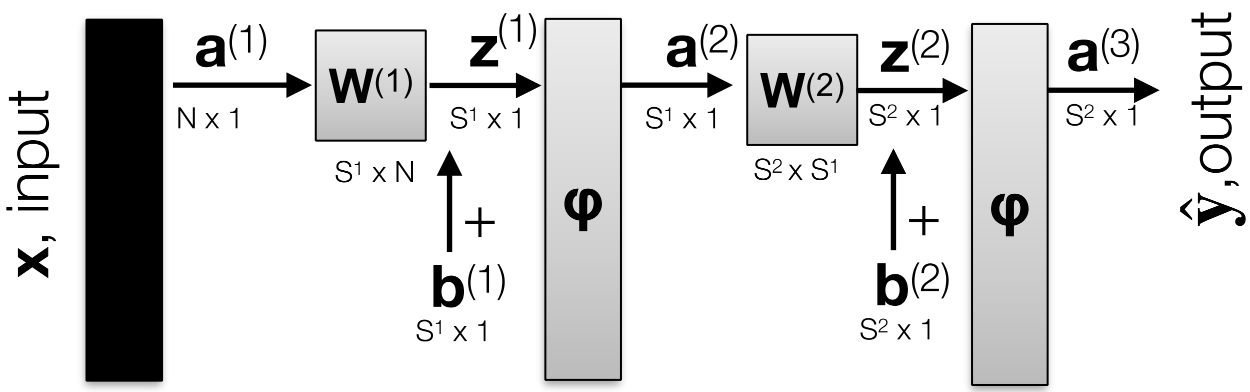
Initialization, check for understanding

Goal: Std[$z_i^{(L)}$] ≈ 4

Std[$a_i^{(L+1)}$] ≈ 1

$$w_{ij} \sim \mathcal{N}(0,\alpha)$$





$z_i^{(L)}$

$a_i^{(L+1)}$ are all zero mean to start with…

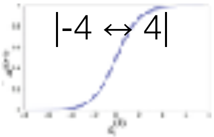
w_{ij}

$\mathbf{b}^{(L)}$

starts as zeros

Glorot Weight Initialization

Xavier Glorot JMLR 2010 Yoshua Bengio
DIRO, Université de Montréal, Montréal, Québec, Canada



try not to saturate

$$z_i^{(L)} = \sum_j^{n^{(L)}} w_{ij} a_j^{(L)} \qquad w_{ij} \sim \mathcal{N}(0,\alpha)$$

Goal: $\text{Std}[z_i^{(L)}] \approx 4$

Random variables add variances: $\text{Var}[z_i^{(L)}] = \sum_j^{n^{(L)}} \text{Var}[w_{ij} \cdot a_j^{(L)}]$

Variance of multiplication of random variables:

$$\text{Var}[z_i^{(L)}] = \sum_j^{n^{(L)}} \cancel{E[w_{ij}]^2 \text{Var}[a_j^{(L)}] + \text{Var}[w_{ij}] E[a_j^{(L)}]^2} + \text{Var}[w_{ij}]\text{Var}[a_j^{(L)}] \quad = \sum_j^{n^{(L)}} \text{Var}[w_{ij}]\text{Var}[a_j^{(L)}]$$

…both are zero mean…

same var, $\forall j$

$$\text{Var}[z_i^{(L)}] = n^{(L)}\text{Var}[w_{ij}]\text{Var}[a^{(L)}] = n^{(L)}\text{Var}[w_{ij}]$$

≈ 1 , from previous layer

$$\boxed{\text{Std}[z_i^{(L)}] = \sqrt{n^{(L)}} \cdot \text{Std}[w_{ij}]}$$

$$\text{Std}[z_i^{(L)}] = 4 = \sqrt{n^{(L)}} \cdot \text{Std}[w_{ij}]$$

$$\text{Std}[w_{ij}] = 4 \cdot \sqrt{\frac{1}{n^{(L)}}}$$

$$w_{ij}^{(L)} \sim \mathcal{N}\left(0, 4 \cdot \sqrt{\frac{1}{n^{(L)}}}\right)$$

forward
from sigmoid

Understanding the difficulty of training deep feedforward neural networks

Glorot Weight Initialization

$$\text{Std}[z_i^{(L)}] = 4 = \sqrt{n^{(L)}} \cdot \text{Std}[w_{ij}]$$

$$w_{ij}^{(L)} \sim \mathcal{N}\left(0,\, 4 \cdot \sqrt{\frac{1}{n^{(L)}}}\right)$$

forward
from sigmoid

$$\mathbf{v}^{(L)} = \mathbf{a}^{(L)}(1 - \mathbf{a}^{(L)})\mathbf{W}^{(L)} \cdot \mathbf{v}^{(L+1)}$$

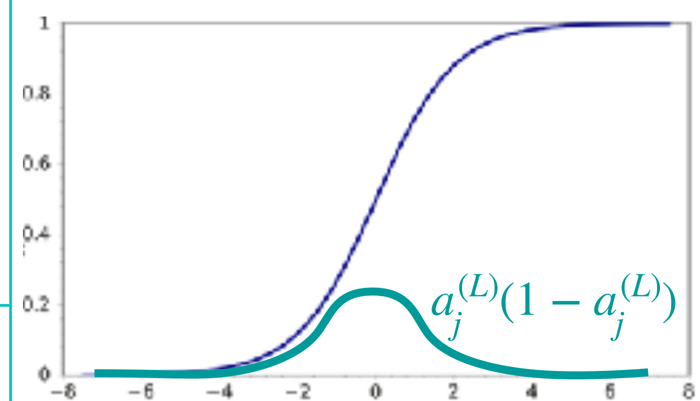
want to keep variance of $\mathbf{v}$ stable magnitude → stable gradient

Similar calculation for back prop.

$$\text{Var}[v_i^{(L)}] = n^{(L+1)}\text{Var}[w_{ij}]\text{Var}[v_j^{(L+1)} \cdot a_j^{(L)}(1 - a_j^{(L)})]$$

$$\text{Std}[v_i^{(L)}] = \sqrt{n^{(L+1)}} \cdot \text{Std}[w_{ij}] \cdot 0.25$$

want = 1



$$w_{ij}^{(L)} \sim \mathcal{N}\left(0,\, 4 \cdot \sqrt{\frac{1}{n^{(L+1)}}}\right)$$

backward
from sensitivity

$$w_{ij}^{(L)} \sim \mathcal{N}\left(0,\, 4 \cdot \sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}\right)$$

compromise

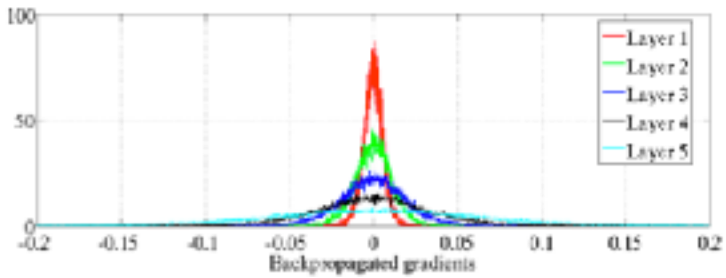
$$w_{ij}^{(L)} \sim U\left[\pm 4\sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}\right]$$

if drawn from uniform dist.

Glorot Weight Initialization

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada



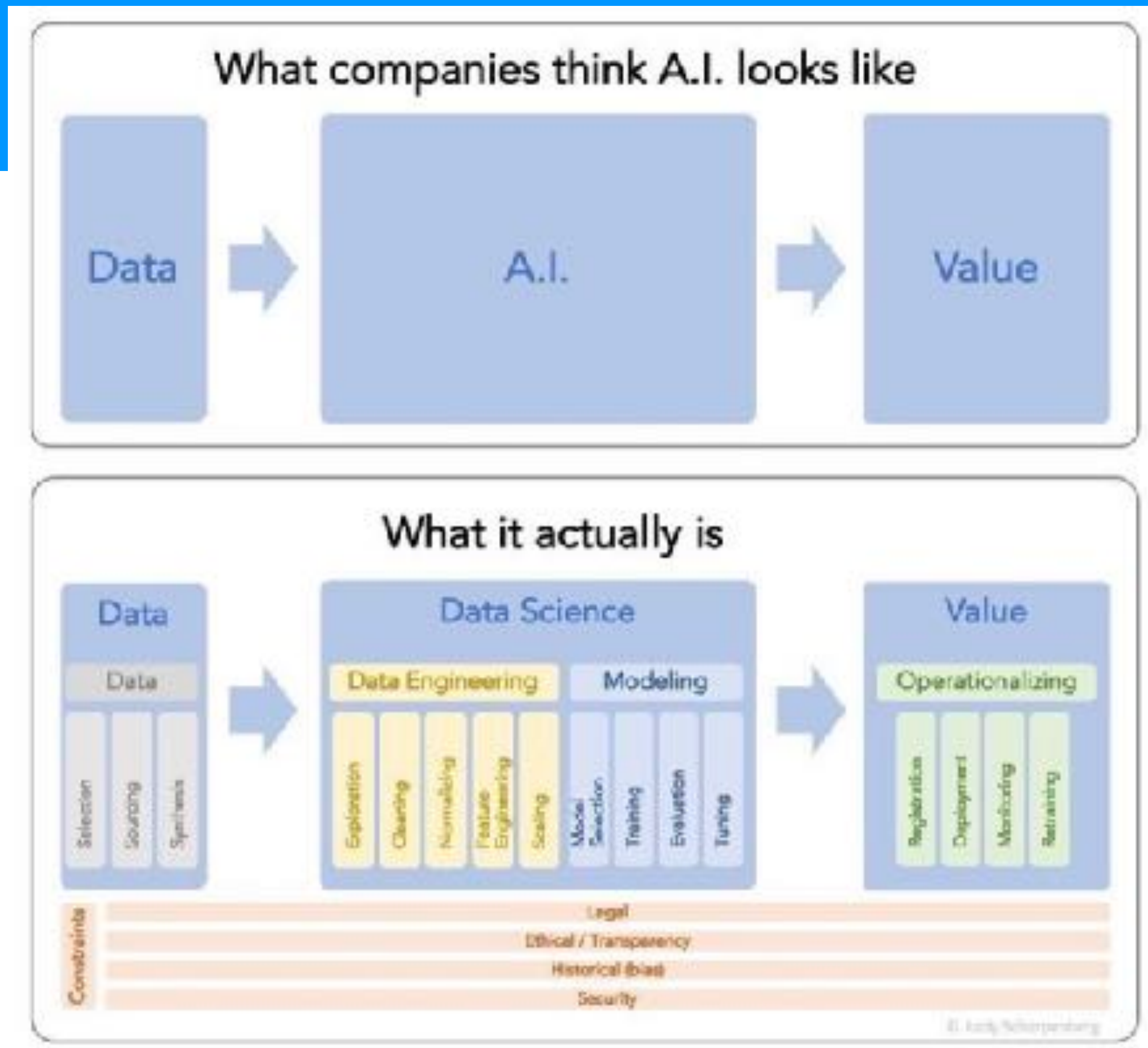
Starting gradient histograms per layer
standard initialization

Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

`08a. Practical_NeuralNetsWithBias.ipynb`

- ~~Momentum~~
- ~~Learning Rate Adaptation~~
- ~~Cross Entropy~~
- Smarter Weight Initialization

Beyond Sigmoid: Other Activations

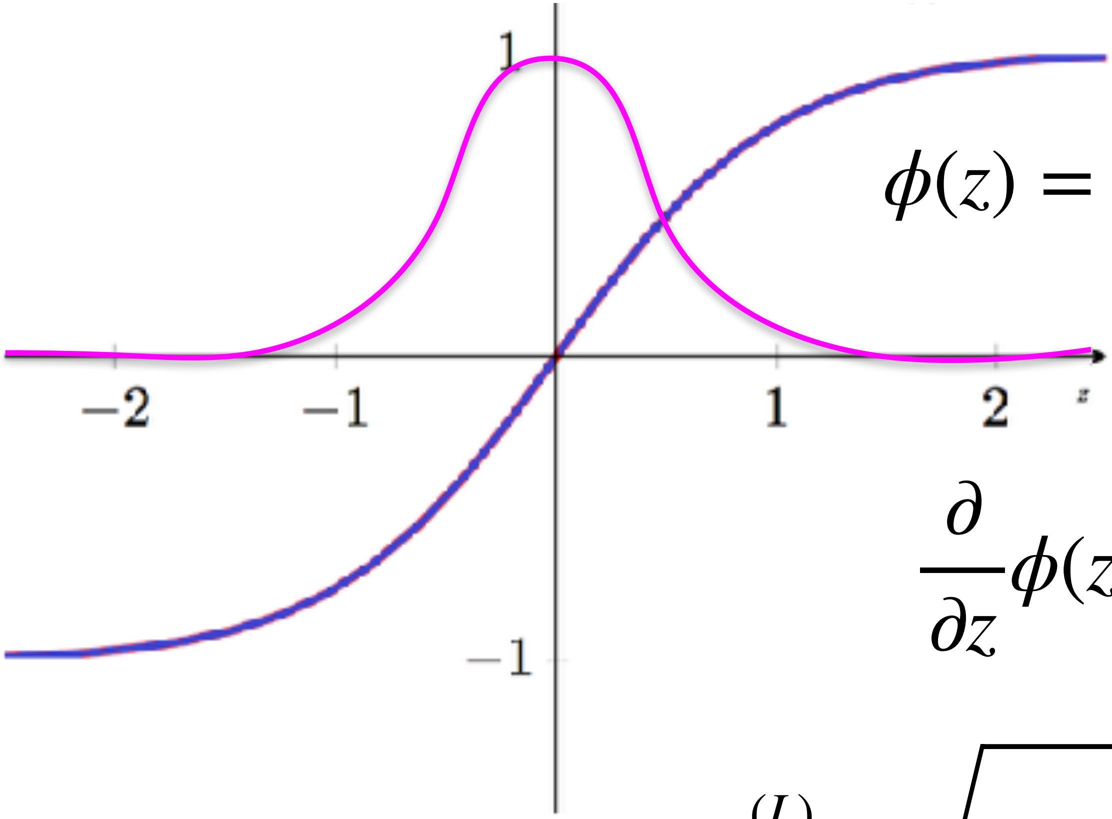


Can we do better than sigmoid?

- Perhaps! Sigmoid assumes [0, 1] firing, but that doesn't need to be true, right?
- If we change the activation, we need to:
  - derive the gradient for back prop
  - define what "saturation" means
  - update initialization of w_{ij} to not "saturate" in feedforward
  - update initialization so that sensitivity variance is about "1"
- Other researchers have done this for you! But you need to use it properly…

New Activation: Hyperbolic Tangent

- Basically a sigmoid from -1 to 1



$$\phi(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$\frac{\partial}{\partial z}\phi(z) = \mathrm{sech}^2(z)$$

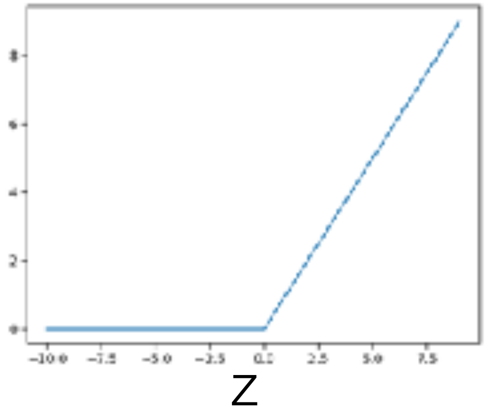
Back Prop

$$w_{ij}^{(L)} \sim \sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$$

Similar to sigmoid, ***no factor of 4***

New Activation: ReLU

- A new nonlinearity: **rectified linear units**



$$\phi(z) = \begin{cases} z, \text{ if } z > 0 \\ 0, \text{ else} \end{cases}$$

it has the advantage of **large gradients** and **extremely simple** derivative

$$\frac{\partial}{\partial z}\phi(z) = \begin{cases} 1, \text{ if } z > 0 \\ 0, \text{ else} \end{cases}$$

$$w_{ij}^{(L)} \sim \sqrt{2}\sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$$

New init

Other Activation Functions

- Sigmoid Weighted Linear Unit **SiLU** (also called Swish)
- Mixing of sigmoid, σ, and ReLU

$$\phi(z) = \sigma(z) \cdot z$$

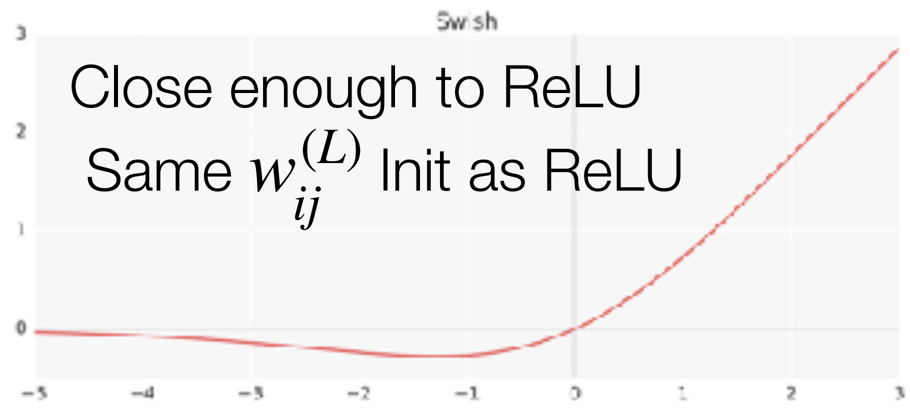


Figure 1: The Swish activation function.

$$\frac{\partial \phi(z)}{\partial z} = \frac{\partial}{\partial z}\sigma(z) \cdot z$$

$$= z \cdot \left[\frac{\partial}{\partial z}\sigma(z)\right] + \sigma(z) \cdot \left[\frac{\partial}{\partial z} z\right]$$

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

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

$$= \phi(z) + \sigma(z) \cdot (1 - \phi(z))$$

Elfwing, Stefan, Eiji Uchibe, and Kenji Doya. "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning." Neural Networks (2018).

Ramachandran P, Zoph B, Le QV. Swish: a Self-Gated Activation Function. arXiv preprint arXiv:1710.05941. 2017 Oct 16

Glorot and He Initialization

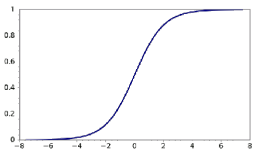
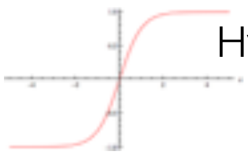
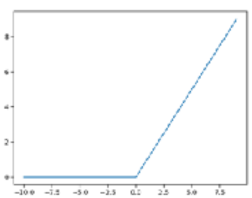
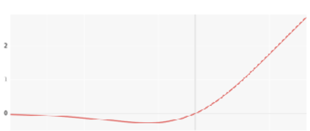
We have solved this assuming the activation output is in the range -4 to 4 (for a sigmoid) and assuming that we use Gaussian for sampling.

This range is different depending on the activation and assuming Gaussian or Uniform sampling.

	Uniform	Gaussian
Tanh	$w_{ij}^{(L)} \sim \sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}$	$w_{ij}^{(L)} \sim \sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$
Sigmoid	$w_{ij}^{(L)} \sim 4\sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}$	$w_{ij}^{(L)} \sim 4\sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$
ReLU SiLU	$w_{ij}^{(L)} \sim \sqrt{2}\sqrt{\frac{6}{n^{(L)} + n^{(L+1)}}}$	$w_{ij}^{(L)} \sim \sqrt{2}\sqrt{\frac{2}{n^{(L)} + n^{(L+1)}}}$

Summarized by Glorot and He

Activations Summary

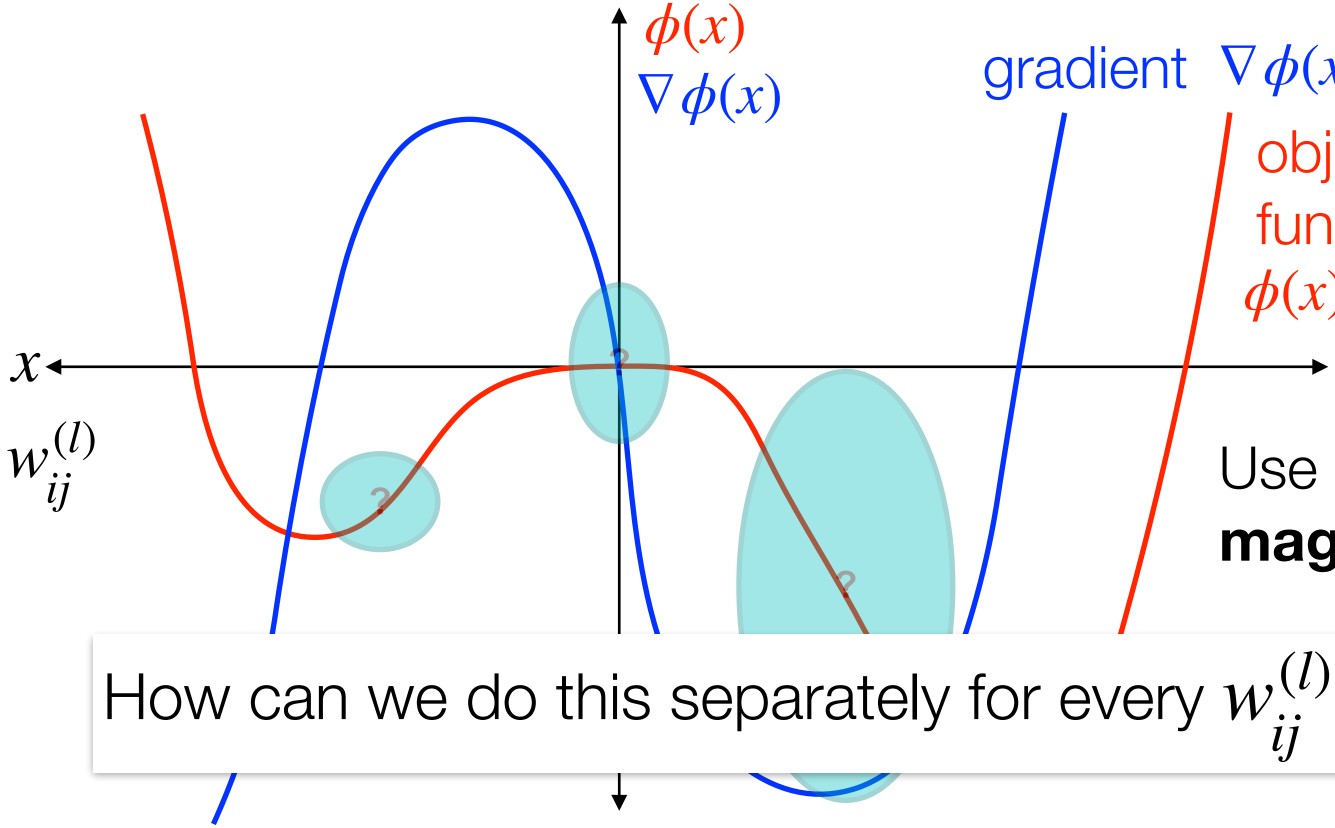
	Definition	Derivative	**Weight Init** *(Uniform Bounds)*
 Sigmoid	$\phi(z) = \dfrac{1}{1+e^{-z}}$	$\nabla\phi(z) = a(1-a)$	$w_{ij}^{(L)} \sim \pm 4\sqrt{\dfrac{6}{n^{(L)}+n^{(L+1)}}}$
 Hyperbolic Tangent	$\phi(z) = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$\nabla\phi(z) = \dfrac{4}{(e^{z}+e^{-z})^2}$	$w_{ij}^{(L)} \sim \pm \sqrt{\dfrac{6}{n^{(L)}+n^{(L+1)}}}$
 ReLU	$\phi(z) = \begin{cases} z, \text{ if } z>0 \\ 0, \text{ else} \end{cases}$	$\nabla\phi(z) = \begin{cases} 1, \text{ if } z>0 \\ 0, \text{ else} \end{cases}$	$w_{ij}^{(L)} \sim \pm \sqrt{2}\sqrt{\dfrac{6}{n^{(L)}+n^{(L+1)}}}$
 SiLU	$\phi(z) = \dfrac{z}{1+e^{-z}}$	$\nabla\phi(z) = \phi(z) + \sigma(z)\cdot(1-\phi(z))$	

Adaptive Optimization

Going beyond
changing the learning rate

Be adaptive based on Gradient Magnitude?

- Decelerate down regions that are steep
- Accelerate on plateaus



Use **inverse** of **magnitude** of **gradient**!

How can we do this separately for every $w_{ij}^{(l)}$ in every $\mathbf{W}^{(l)}$?

Momentum: be robust to **abrupt changes** in **steepness** (accumulate magnitudes)

http://www.technologyuk.net/mathematics/differential-calculus/higher-derivatives.shtml

Be adaptive based on Gradient Magnitude?

Inverse magnitude of gradient in multiple directions?

$$\mathbf{W}_{k+1} \leftarrow \mathbf{W}_k + \eta \frac{1}{\sqrt{\mathbf{G}_k + \epsilon}} \odot \nabla J(\mathbf{W}_k)$$

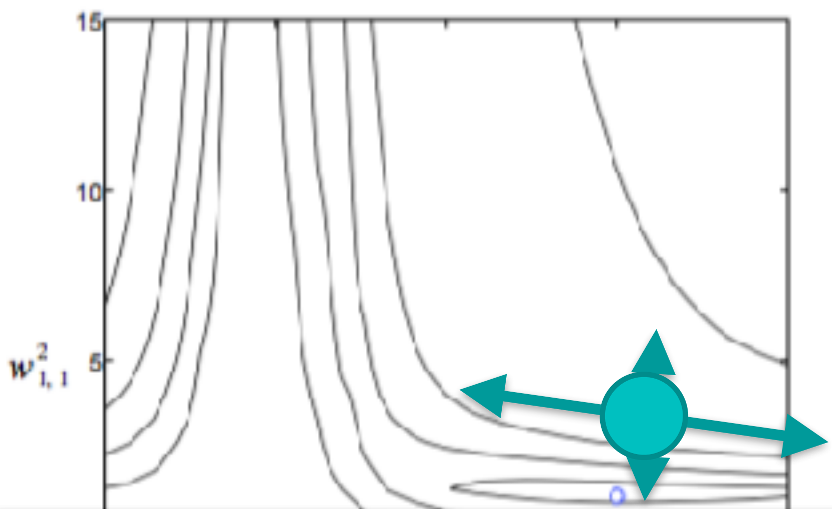
new matrix for normalizing

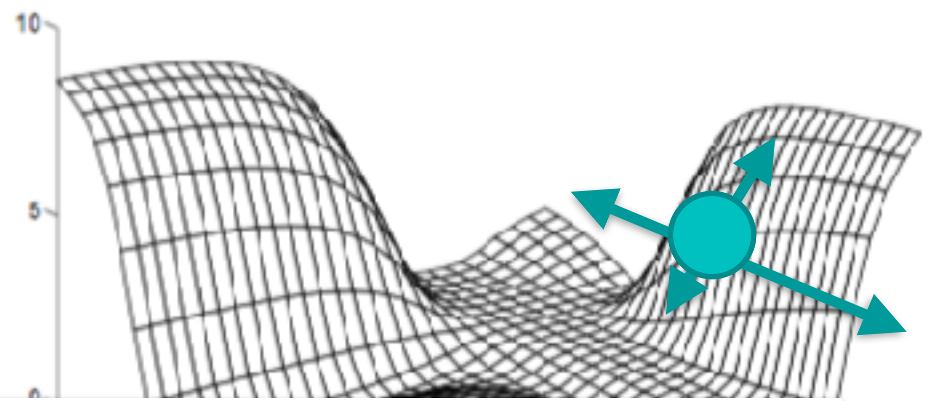
gradient

$$\mathbf{G}_k = \nabla J(\mathbf{W}_k) \odot \nabla J(\mathbf{W}_k)$$

same size as $\mathbf{W}$

```
G = gradW1 * gradW1
W1 += eta*gradW1 / sqrt(G+eps)
```





Now we just need to add momentum to $\mathbf{G}_k^{(l)}$

Note: $\mathbf{G}$ exists for every layer, but we will abuse layer notation

Common Adaptive Strategies $\mathbf{W}_{k+1} = \mathbf{W}_k - \eta \cdot \rho_k$

Adjust each element of gradient by the steepness

• AdaGrad all operations are per element	$\rho_k = \frac{1}{\sqrt{\mathbf{G}_k + \epsilon}} \odot \nabla J(\mathbf{W}_k)$	where $\mathbf{G}_k = \gamma \cdot \mathbf{G}_{k-1} + \nabla J(\mathbf{W}_k) \odot \nabla J(\mathbf{W}_k)$
• RMSProp all operations are per element	$\rho_k = \frac{1}{\sqrt{\mathbf{V}_k + \epsilon}} \odot \nabla J(\mathbf{W}_k)$	$\mathbf{G}_k = \nabla J(\mathbf{W}_k) \odot \nabla J(\mathbf{W}_k)$ $\mathbf{V}_k = \gamma \cdot \mathbf{V}_{k-1} + (1-\gamma) \cdot \mathbf{G}_k$
• AdaDelta all operations are per element	$\rho_k = \frac{\mathbf{M}_k}{\sqrt{\mathbf{V}_k + \epsilon}}$	$\mathbf{M}_{k+1} = \gamma \cdot \mathbf{M}_k + (1-\gamma) \cdot \nabla J(\mathbf{W}_k)$
• AdaM	$\mathbf{G}$ updates with decaying momentum of J and J^2	
• NAdaM	same as Adam, but with nesterov's acceleration	

None of these are **"one-size-fits-all"** because the space of neural network **optimization varies** by problem, AdaM is **popular** but **not a panacea**

Adaptive Momentum

All operations are element wise:

$$\beta_1 = 0.9,\ \beta_2 = 0.999,\ \eta = 0.001,\ \epsilon = 10^{-8}$$

$$k = 0,\ \mathbf{M}_0 = \mathbf{0},\ \mathbf{V}_0 = \mathbf{0}$$

Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma* University of Amsterdam, OpenAI

Jimmy Lei Ba* University of Toronto

For each epoch:

update iteration $k \leftarrow k + 1$

get gradient $\nabla J(\mathbf{W}_k)$

for large k, $\hat{\mathbf{M}} \approx \mathbf{M}$, $\hat{\mathbf{V}} \approx \mathbf{V}$

accumulated gradient $\mathbf{M}_k \leftarrow \beta_1 \cdot \mathbf{M}_{k-1} + (1 - \beta_1) \cdot \nabla J(\mathbf{W}_k)$

accumulated squared gradient $\mathbf{V}_k \leftarrow \beta_2 \cdot \mathbf{V}_{k-1} + (1 - \beta_2) \cdot \nabla J(\mathbf{W}_k) \odot \nabla J(\mathbf{W}_k)$

boost moments magnitudes (notice k in exponent)

$$\hat{\mathbf{M}}_k \leftarrow \frac{\mathbf{M}_k}{(1 - [\beta_1]^k)} \qquad \hat{\mathbf{V}}_k \leftarrow \frac{\mathbf{V}_k}{(1 - [\beta_2]^k)}$$

update gradient, normalized by second moment similar to AdaDelta

$$\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \eta \cdot \frac{\hat{\mathbf{M}}_k}{\sqrt{\hat{\mathbf{V}}_k + \epsilon}}$$

gradient with momentum

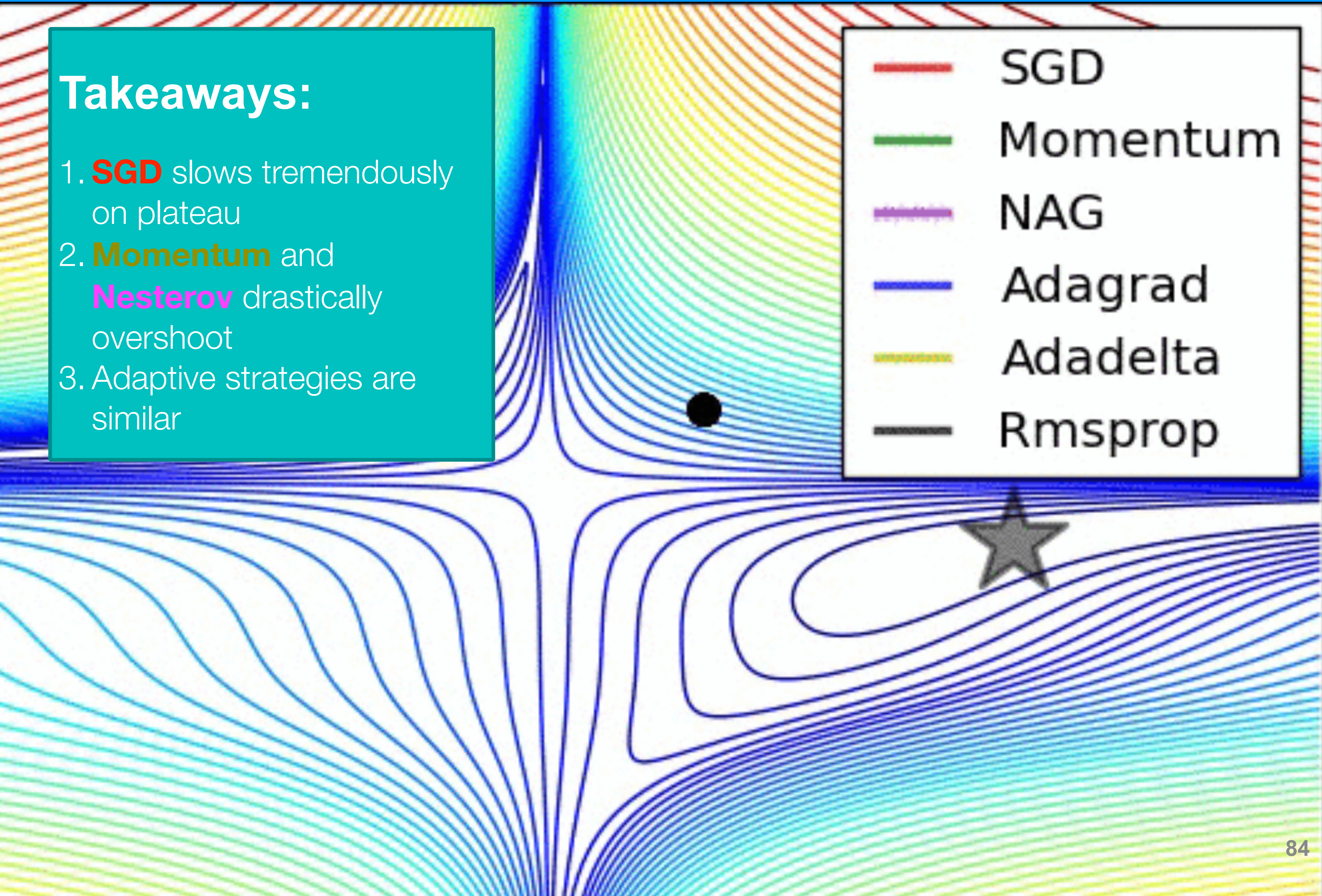
squared magnitude normalizer

Visualization of Optimization

https://ruder.io/optimizing-gradient-descent/

Takeaways:

1. **SGD** slows tremendously on plateau
2. **Momentum** and **Nesterov** drastically overshoot
3. Adaptive strategies are similar

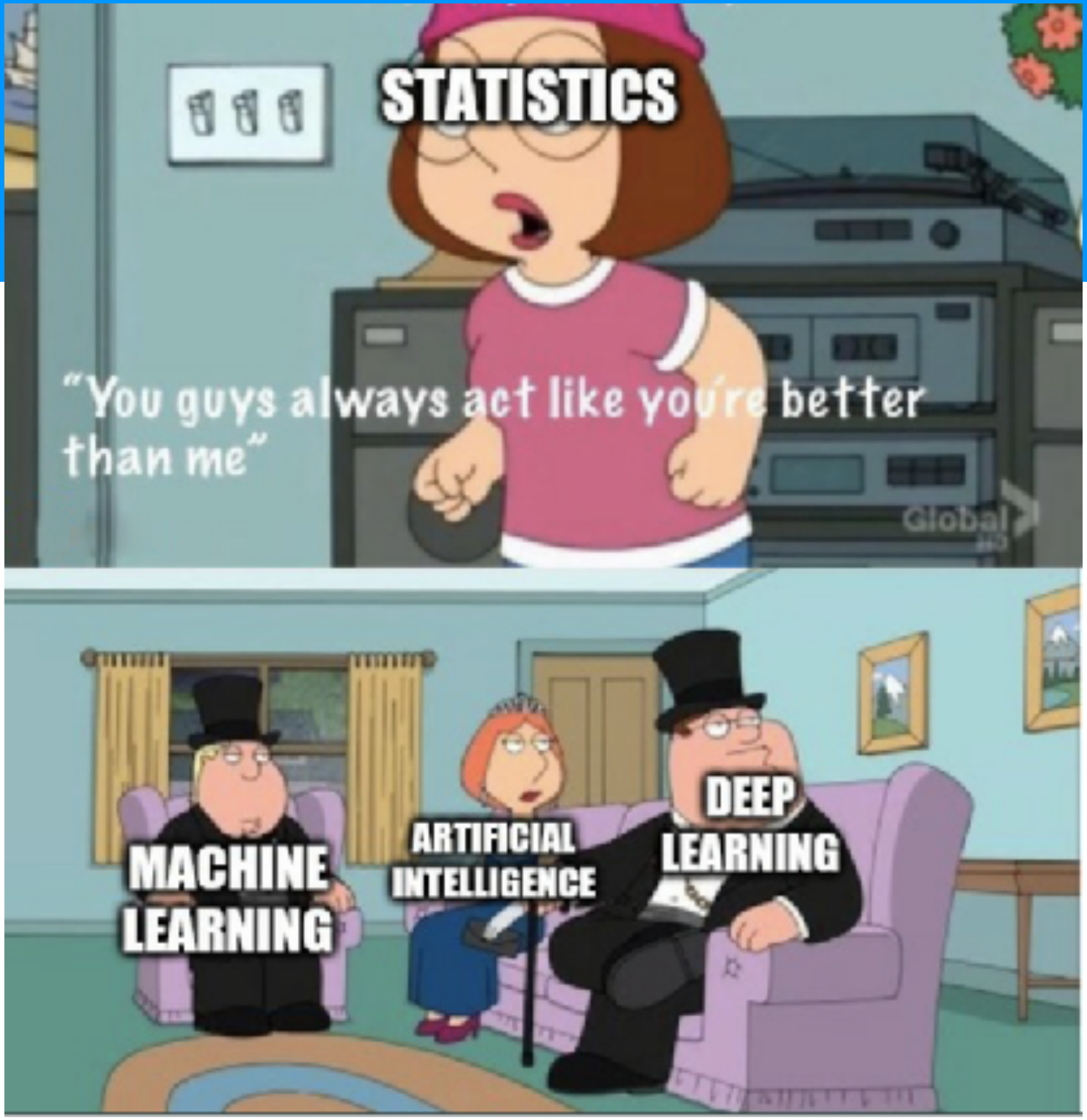


Demo

`08a. Practical_NeuralNetsWithBias.ipynb`

~~Momentum~~
~~Learning Rate Adaptation~~
~~Cross Entropy~~
~~Smarter Weight Initialization~~
ReLU Nonlinearities
Adaptive training with AdaGrad

Review


STATISTICS
"You guys always act like you're better than me"
Global
MACHINE LEARNING
ARTIFICIAL INTELLIGENCE
DEEP LEARNING

Review

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \rho_k$$

- Cross entropy

$$\mathbf{A}^{(3)} - \mathbf{Y}$$

new final layer update

- Momentum $\rho_k = \alpha \nabla J(\mathbf{W}_k) + \beta \nabla J(\mathbf{W}_{k-1})$

- Nesterov's Accelerated Gradient

$$\rho_k = \underbrace{\beta \nabla J\left(\mathbf{W}_k + \alpha \nabla J(\mathbf{W}_{k-1})\right)}_{\text{step twice}} + \alpha \nabla J(\mathbf{W}_{k-1})$$

- Mini-batching

←all data→

	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8	batch 9
Epoch 1									
Epoch 2									
Epoch 3									
Epoch 4									
...									

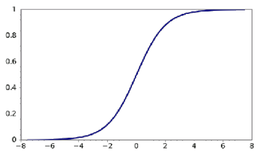
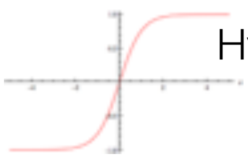
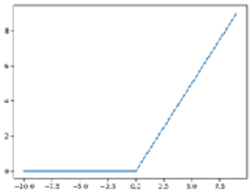
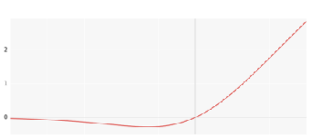
shuffle ordering each epoch and update W's after each batch

- Learning rate adaptation (eta)

$$\eta_e = \eta_0 \cdot d^{\lfloor \frac{e}{e_d} \rfloor}$$

$$\eta_e = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{e}{e_{max}}\pi\right)\right)$$

Review: Activations Summary

	Definition	Derivative	**Weight Init** *(Uniform Bounds)*
 Sigmoid	$\phi(z) = \dfrac{1}{1+e^{-z}}$	$\nabla\phi(z) = a(1-a)$	$w_{ij}^{(L)} \sim \pm 4\sqrt{\dfrac{6}{n^{(L)}+n^{(L+1)}}}$
 Hyperbolic Tangent	$\phi(z) = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$\nabla\phi(z) = \dfrac{4}{(e^{z}+e^{-z})^{2}}$	$w_{ij}^{(L)} \sim \pm\sqrt{\dfrac{6}{n^{(L)}+n^{(L+1)}}}$
 ReLU	$\phi(z) = \begin{cases} z, \text{ if } z>0 \\ 0, \text{ else} \end{cases}$	$\nabla\phi(z) = \begin{cases} 1, \text{ if } z>0 \\ 0, \text{ else} \end{cases}$	$w_{ij}^{(L)} \sim \pm\sqrt{2}\sqrt{\dfrac{6}{n^{(L)}+n^{(L+1)}}}$
 SiLU	$\phi(z) = \dfrac{z}{1+e^{-z}}$	$\nabla\phi(z) = \phi(z) + \sigma(z)\cdot(1-\phi(z))$	