Lecture Notes for

Machine Learning in Python

Professor Eric Larson

Neural Network Optimization and Activation

Class Logistics and Agenda

- Agenda:
  - More optimization techniques
    - Momentum
    - Adaptive learning rates
    - Initialization
    - More activations: Tanh, ReLU, SiLU
  - Programming Examples

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

Mini-batching

- Numerous instances to find one gradient update
  - **solution**: mini-batch

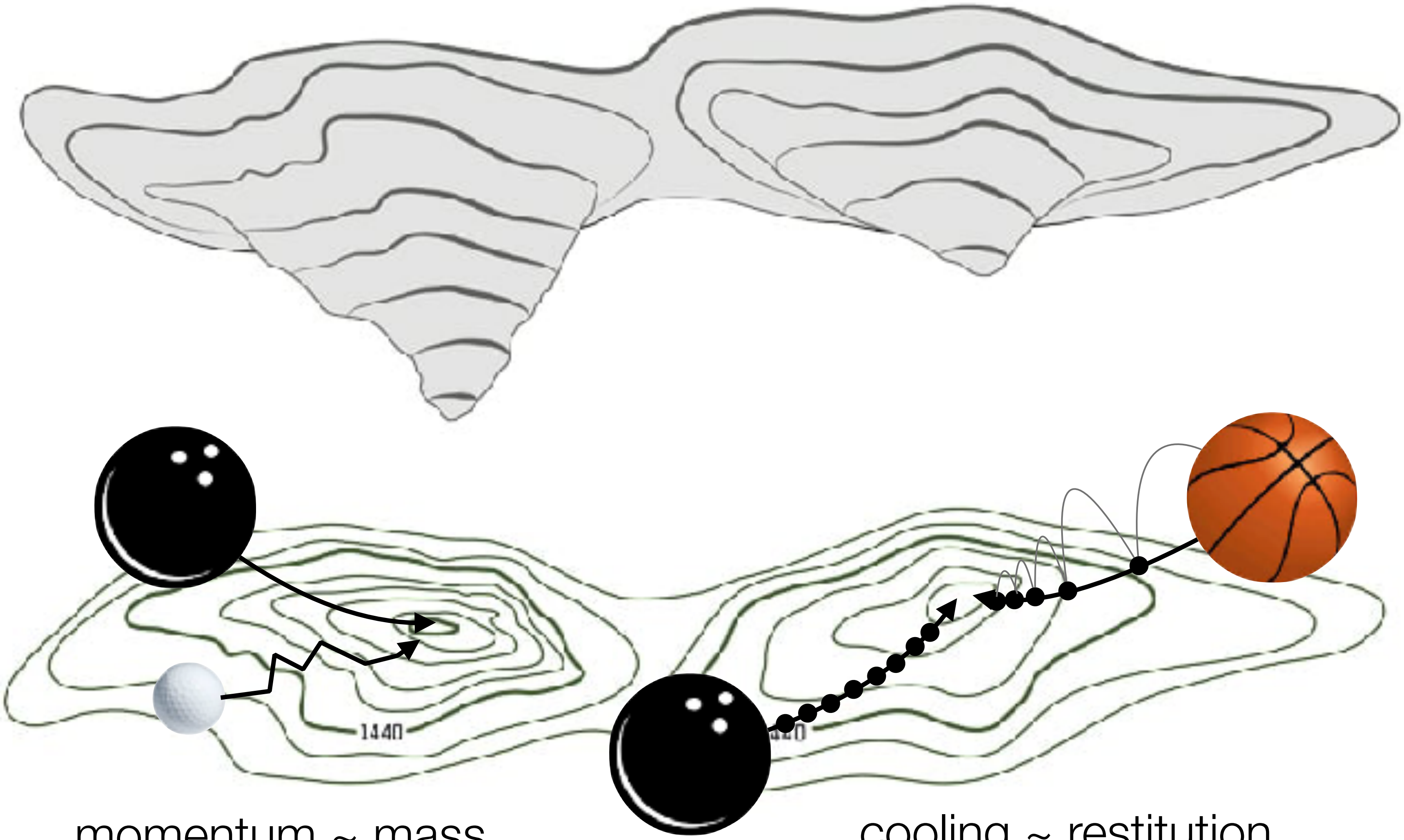
←all data→

	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

- **new problem**: mini-batch gradient updates erratic
  - **solutions**:
    - momentum
    - adaptive learning steps (cooling)

Momentum and Cooling Intuition



momentum ~ mass

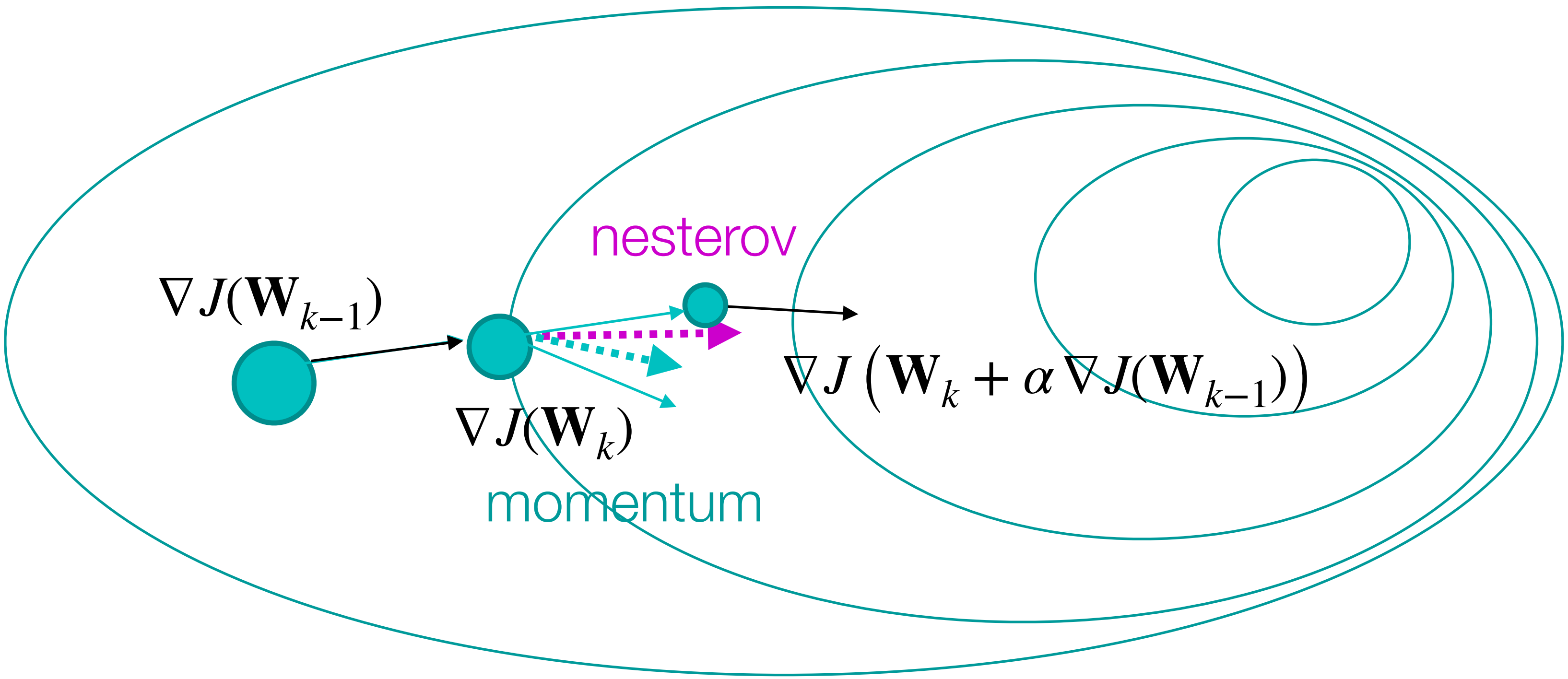
cooling ~ restitution

Momentum

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

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

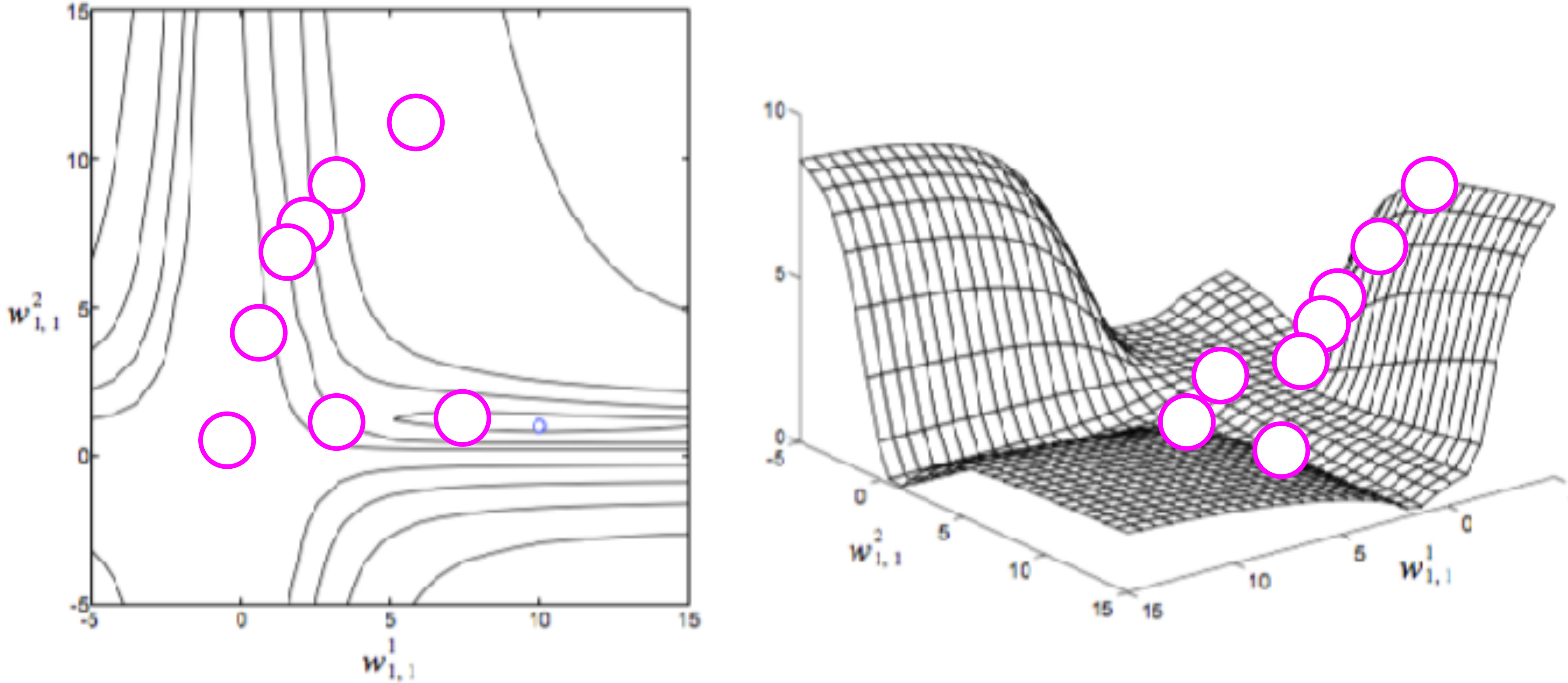
- Nesterov's Accelerated Gradient $\rho_k = \underbrace{\beta \nabla J \left(\mathbf{W}_k + \alpha \nabla J(\mathbf{W}_{k-1})\right)}_{\text{step twice}} + \alpha \nabla J(\mathbf{W}_{k-1})$



Adaptive Strategy: Cooling

- Fixed Reduction at Each Epoch
- Adjust on Plateau
  - make smaller if when J rapidly changes
  - make bigger when J not changing much

$$\eta_e = \eta_0^{(1+e\cdot\epsilon)}$$



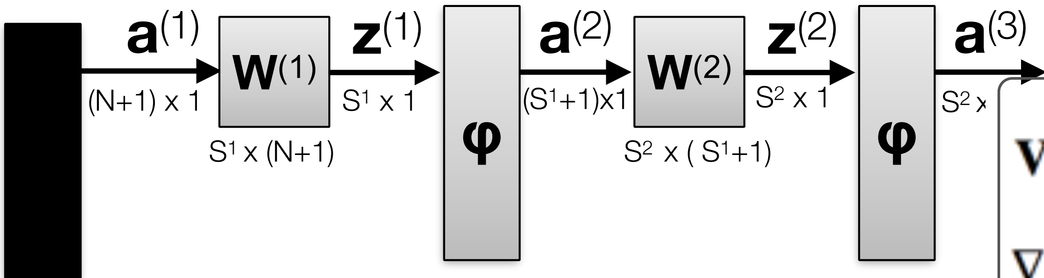
`07. MLP Neural Networks.ipynb`

comparison:

- mini-batch
    - momentum
    - adaptive learning
- L-BFGS

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 $\mathbf{W}^{(l)}$

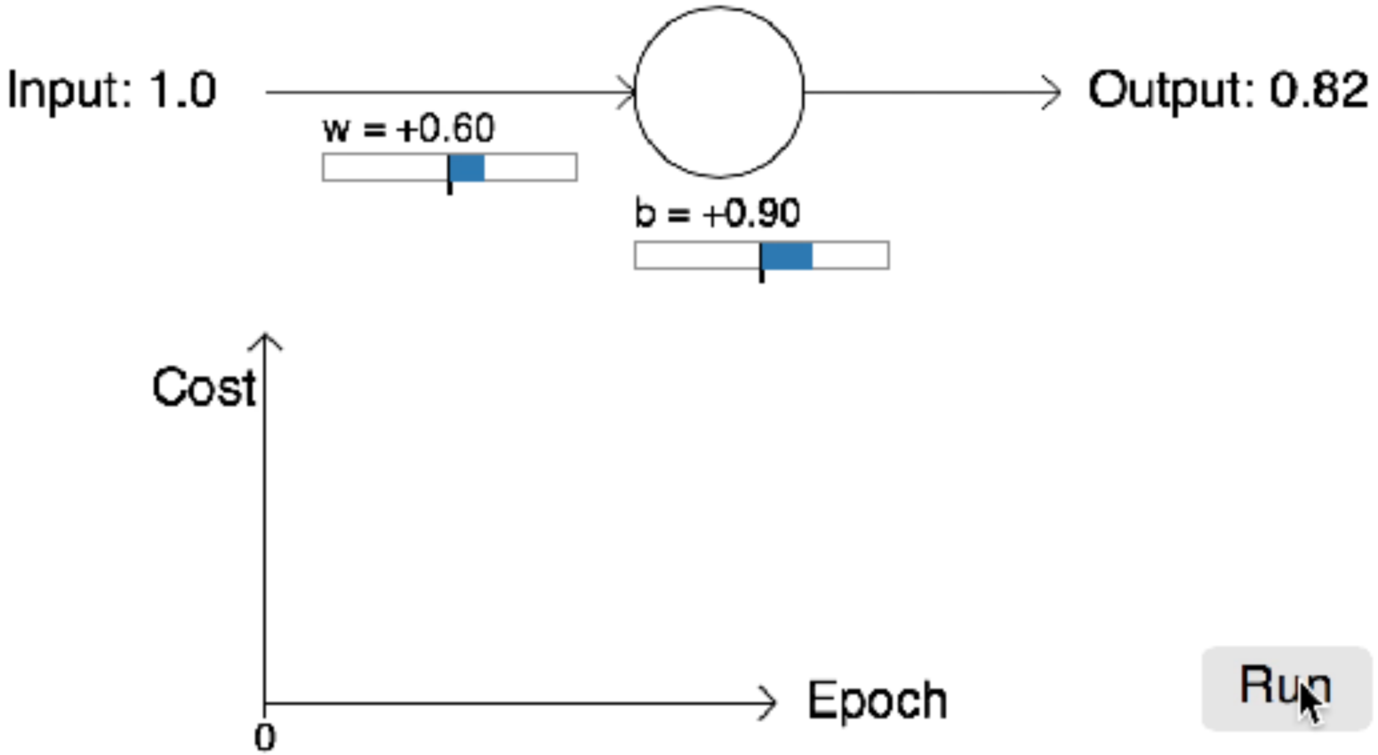
- **Self Test**:
  True or False: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer sensitivity calculation, $\mathbf{V}^{(2)}$, of the back propagation steps. The remainder of the algorithm is unchanged.
  - A. True
  - B. False

Practical Implementation of Architectures

- MSE

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

least squares objective,
tends to slow training initially



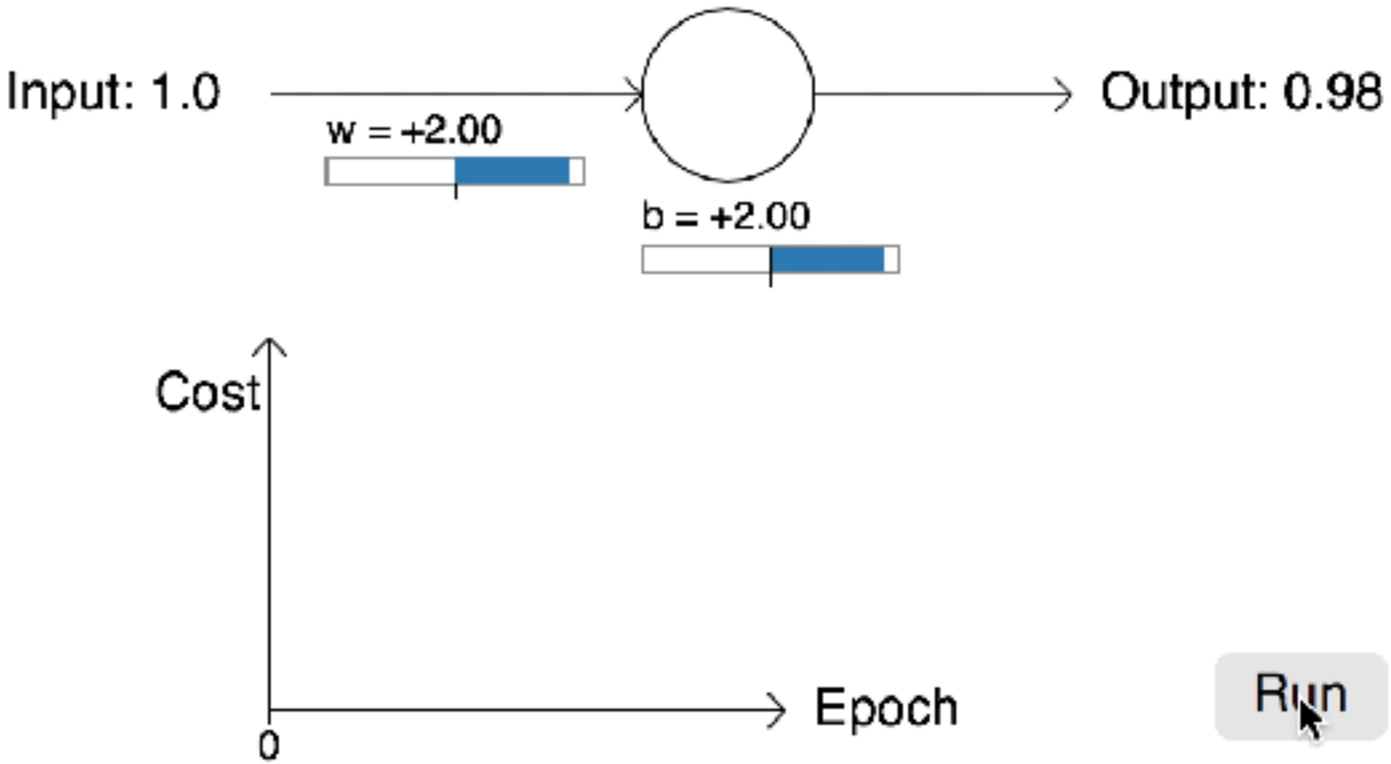
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- MSE

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

least squares objective,
tends to slow training initially



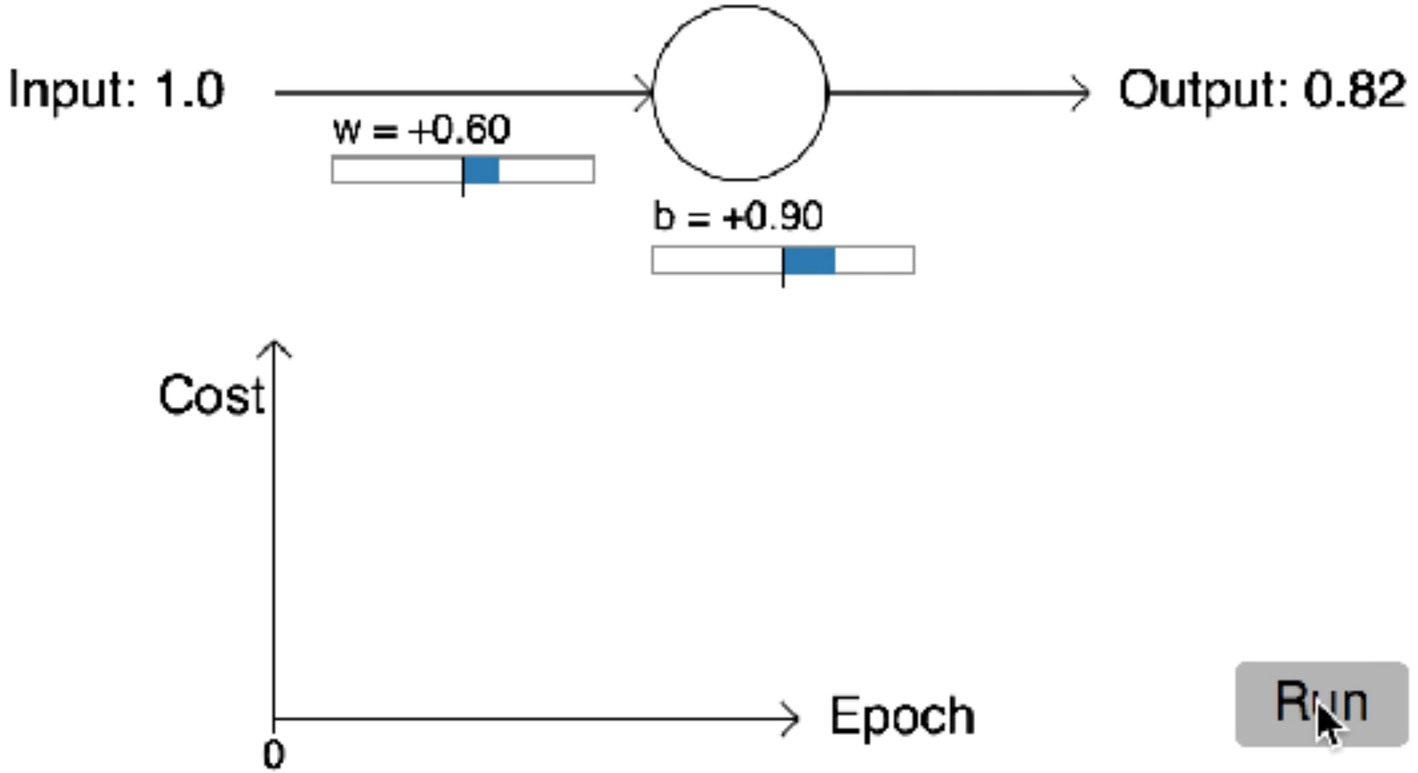
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- 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



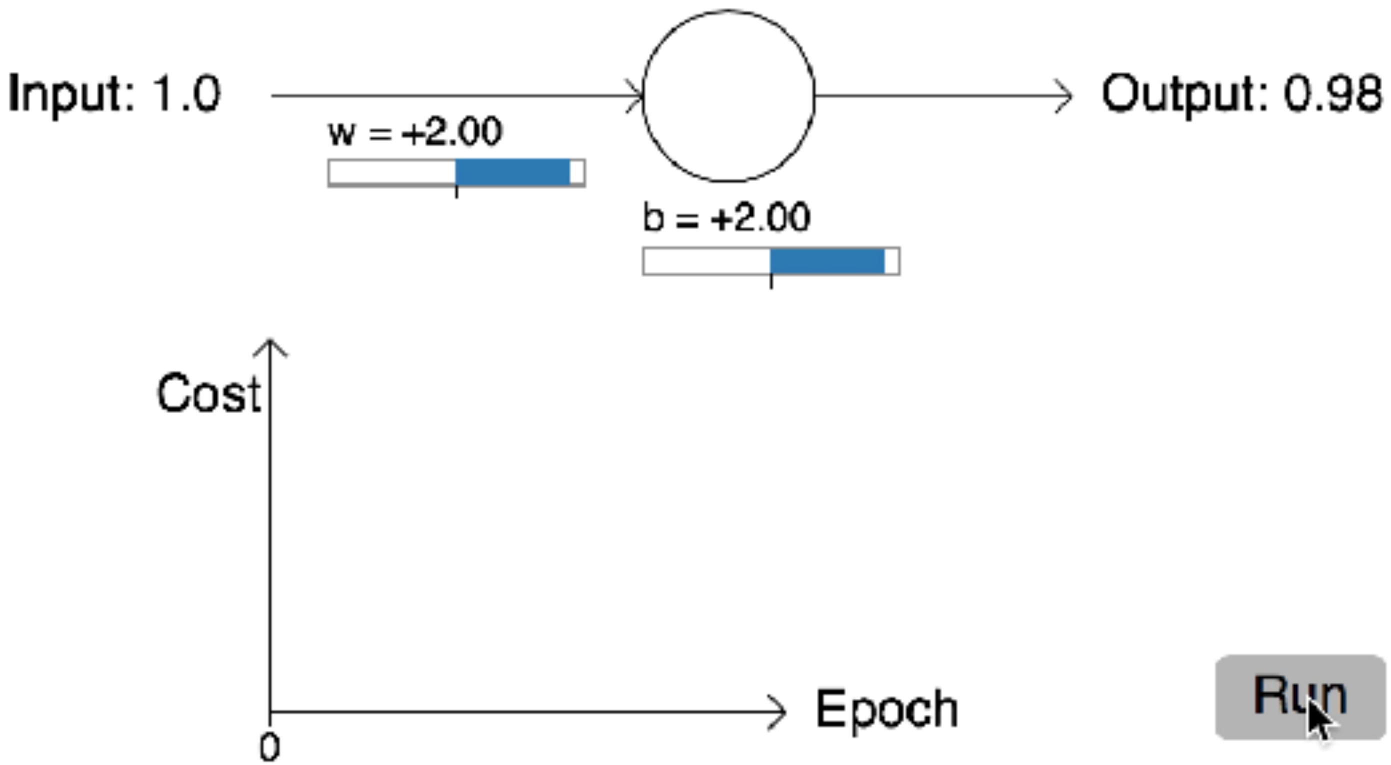
Neural Networks and Deep Learning, Michael Nielson, 2015

Practical Implementation of Architectures

- 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

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

speeds up initial training

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

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

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

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

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

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

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

Practical Implementation of Architectures

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

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

speeds up
initial training

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

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

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

grad2 = V2 @ A2.T
grad1 = V1[1:,:] @ A1.T
```

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

new update

bp-5

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

08. Practical_NeuralNets.ipynb

Demo

cross entropy

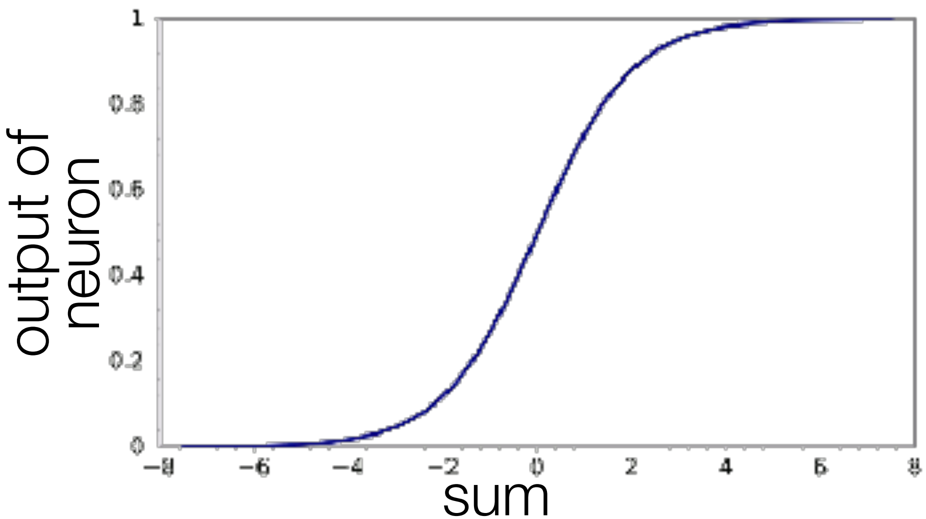
Practical Implementation of Architectures

Formative Self Test

- for adding Gaussian distributions, variances add together

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

- If you initialized the weights, **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

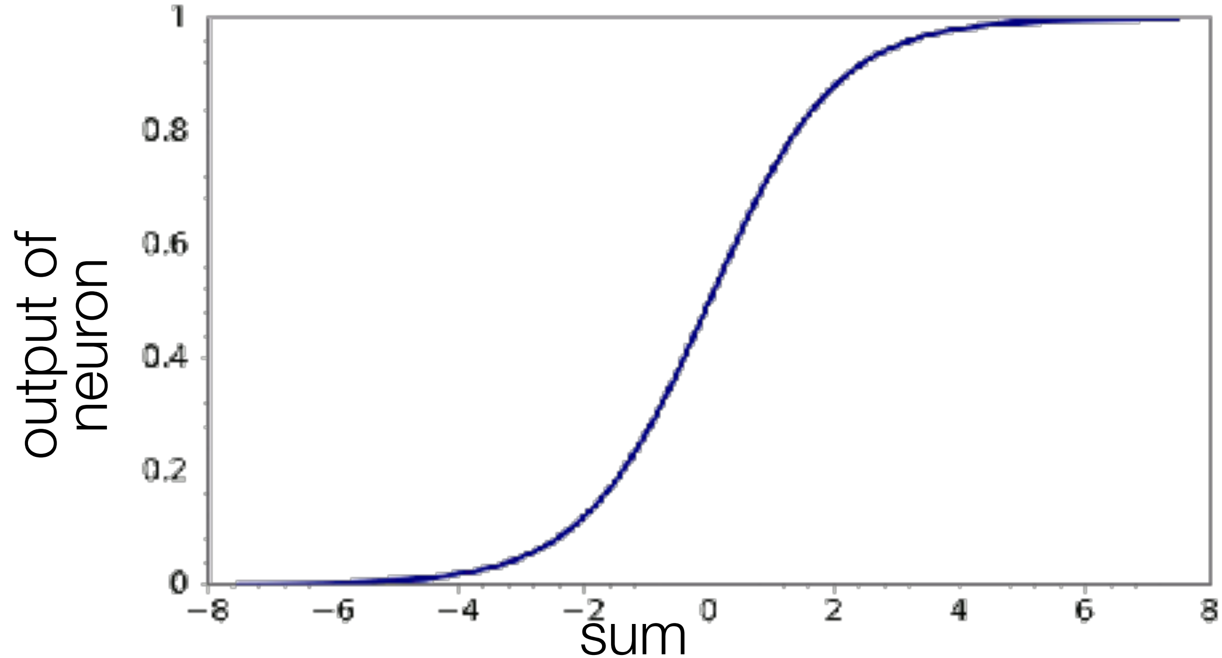


Formative Self Test

- for adding Gaussian distributions, variances add together

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

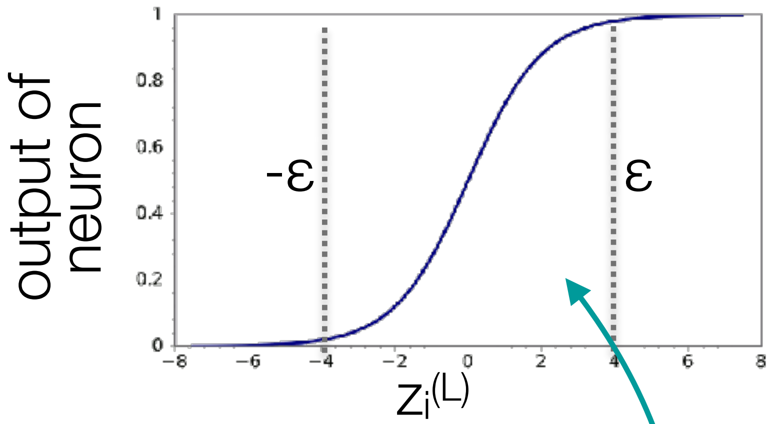
- What is the derivative of a saturated sigmoid neuron?
  - A. zero
  - B. one
  - C. a * (1-a)
  - D. it depends



- **Weight initialization**
  - try not to **saturate** your neurons right away!

$$\mathbf{a}^{(L+1)}=\phi(\mathbf{z}^{(L)})$$
$$\mathbf{z}^{(L)}=\mathbf{W}^{(L)}\mathbf{a}^{(L)}$$

each row is summed before sigmoid



want each $z^{(L)}$ to be between $-\varepsilon<\Sigma<\varepsilon$ for no saturation

solution: squash initial weights magnitude

- one choice: each element of **W** selected from a Gaussian with **zero mean** and **specific standard deviation**

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

For a sigmoid, want $-\varepsilon<z_i^{(L)}<\varepsilon$

$\varepsilon=4$

More Weight Initialization

Understanding the difficulty of training deep feedforward neural networks

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

Goal: We should not saturate **feedforward** or **back propagated** variance

Relate variance of current layer to variance in z, so $\sigma(z_i^{(L)})$ isn't saturated

try not to saturate z

$$z_i^{(L)} = \sum_j^{n^{(L)}} w_{ij} a_j^{(L)}$$

break down feed forward by each multiply

$$\text{Var}[z_i^{(L)}] = \sum_j^{n^{(L)}} \underbrace{E[w_{ij}]^2 \text{Var}[a_j^{(L)}] + \text{Var}[w_{ij}] E[a_j^{(L)}]^2}_{\text{0, if uncorrelated}} + \text{Var}[w_{ij}] \underbrace{\text{Var}[a_j^{(L)}]}_{\approx 1}$$

Want to keep Var[] ~1

assume i.i.d.
expand variance calc

$$\text{Var}[z_i^{(L)}] = 4 = n^{(L)} \text{Var}[w_{ij}] \text{Var}[a_j^{(L)}]$$

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

forward
from data

More Weight Initialization

$$\mathrm{Var}[z_i^{(L)}] = 4 = n^{(L)}\mathrm{Var}[w_{ij}]\mathrm{Var}[a_j^{(L)}]$$

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

forward

from data

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

Similar for back prop.

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

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

backward

from sensitivity

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

compromise

More Weight Initialization

Understanding the difficulty of training deep feedforward neural networks

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

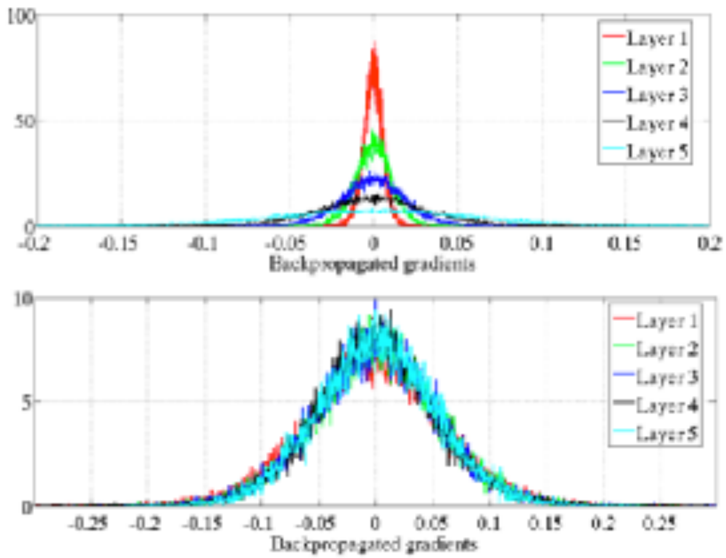


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Starting gradient histograms per layer
standard initialization

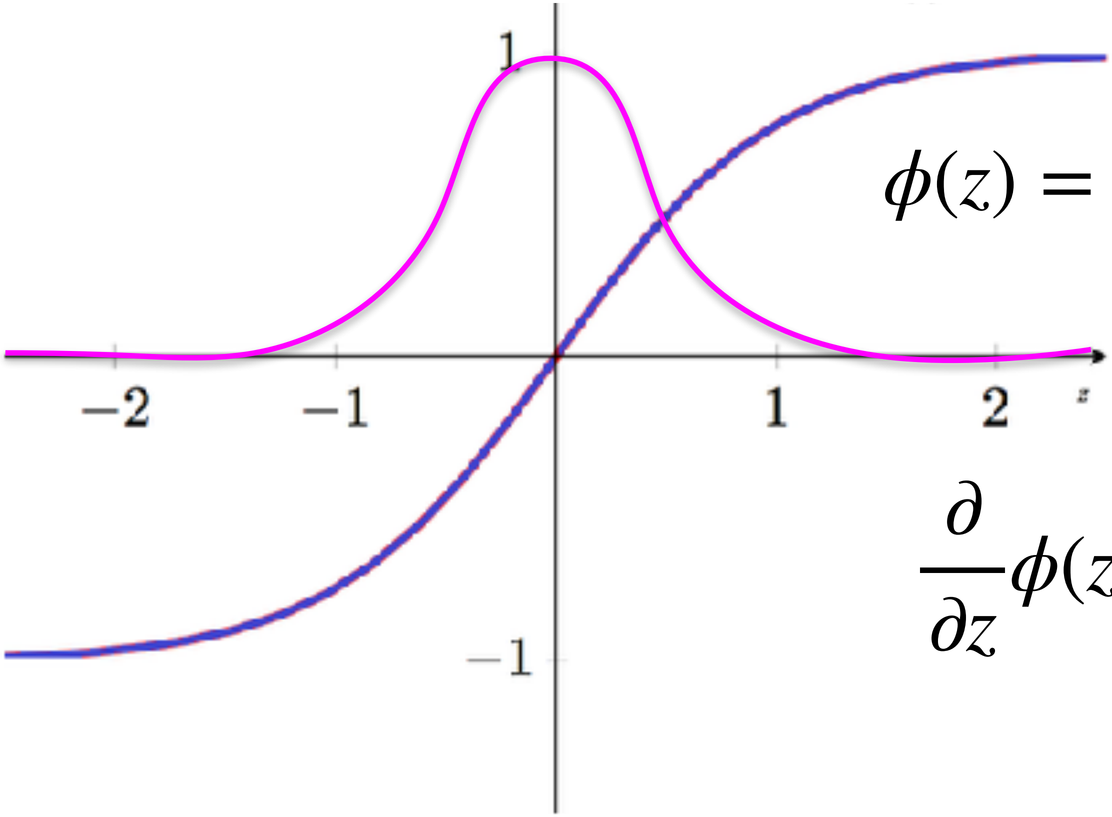
Starting gradient histograms per layer
Glorot initialization

08. Practical_NeuralNets.ipynb **Demo**

Smarter Weight Initialization

New Activation: Hyperbolic Tangent

- Basically a sigmoid from -1 to 1

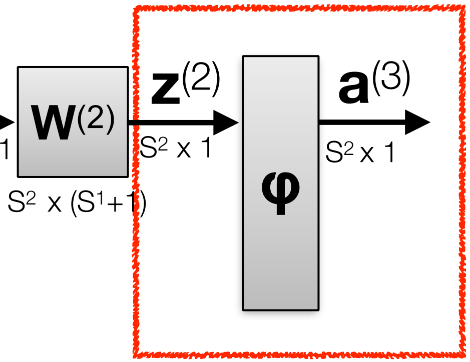


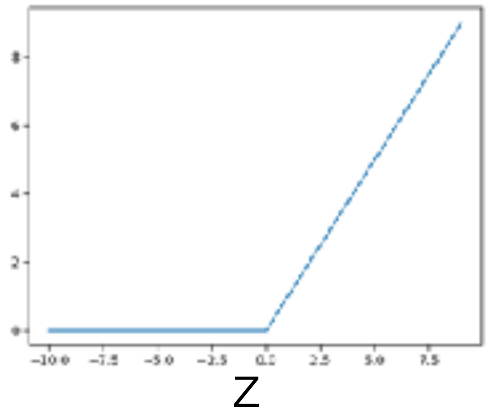
$$\phi(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

New Activation: ReLU

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



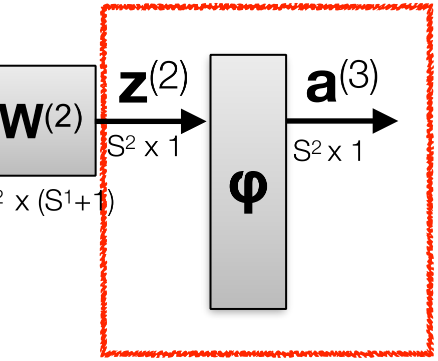


$$\phi(z) = \begin{cases} z, \text{ if } z > 0 \\ 0, \text{ else} \end{cases}$$

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

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

Other Activation Functions



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

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

$$\frac{\partial \varphi(z)}{\partial z} = \varphi(z) + \sigma(z)[1 - \varphi(z)]$$

$$= a^{(l+1)} + \sigma(z^{(l)}) \cdot [1 - a^{(l+1)}]$$

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

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

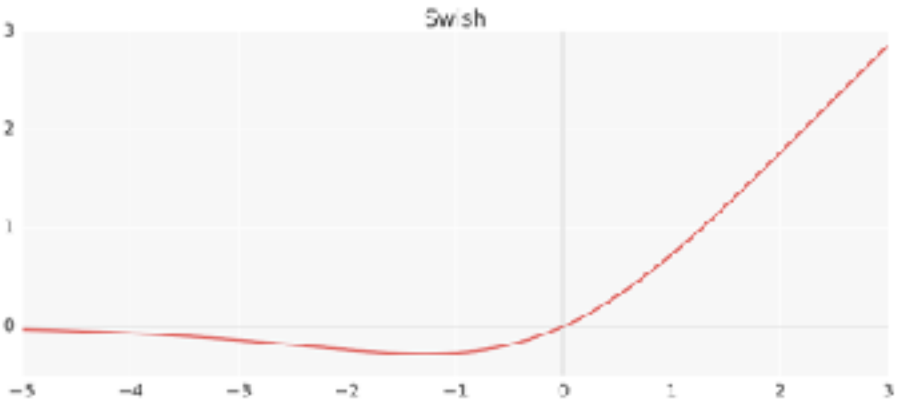


Figure 1: The Swish activation function.

Derivative Calculation:

$$= \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x))$$
$$= \sigma(x) + x \cdot \sigma(x) - x \cdot \sigma(x)^2$$
$$= x \cdot \sigma(x) + \sigma(x)(1 - x \cdot \sigma(x))$$

Glorot and He Initialization

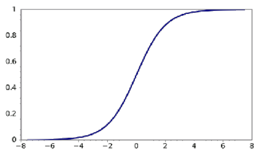
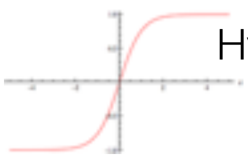
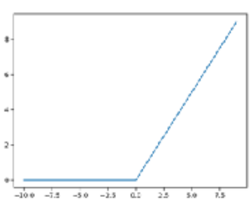
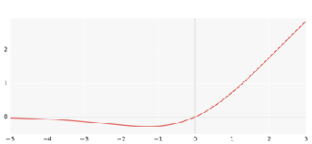
We have solved this assuming the activation output is in the range -4 to 4 (for a sigmoid) and assuming that x is distributed Gaussian

This range, epsilon, is different depending on the activation and assuming Gaussian or Uniform

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

Summarized by Glorot and He

Activations Summary

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

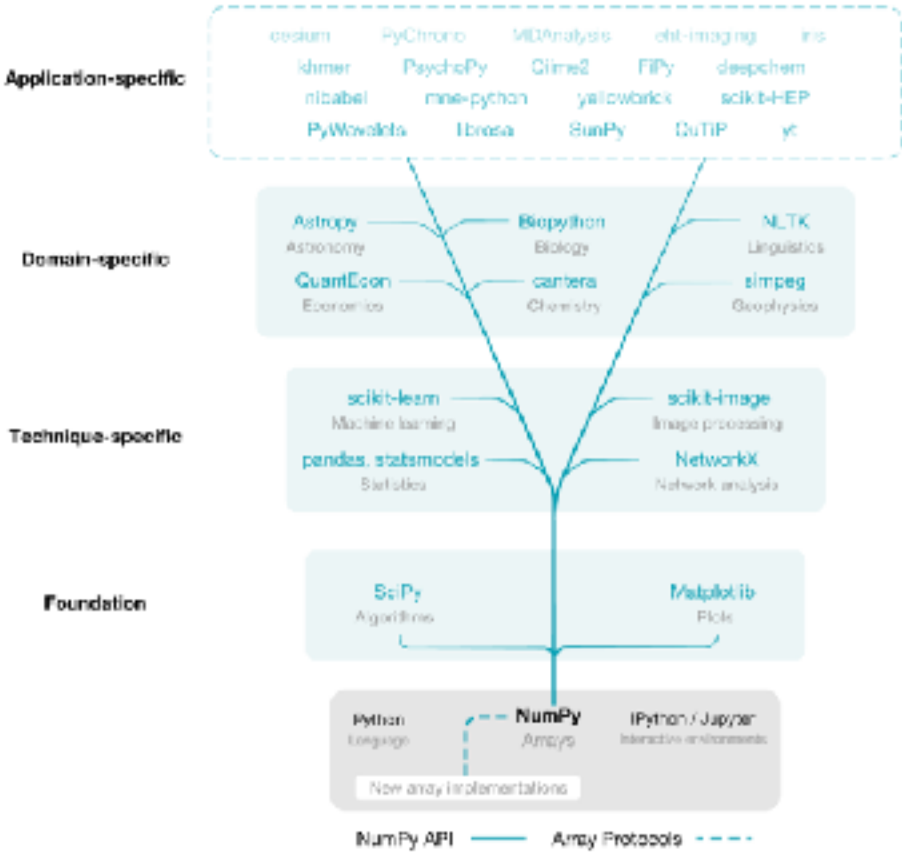
08. Practical_NeuralNets.ipynb **Demo**

ReLU Nonlinearities
Important for deep networks

More Adaptive Optimization

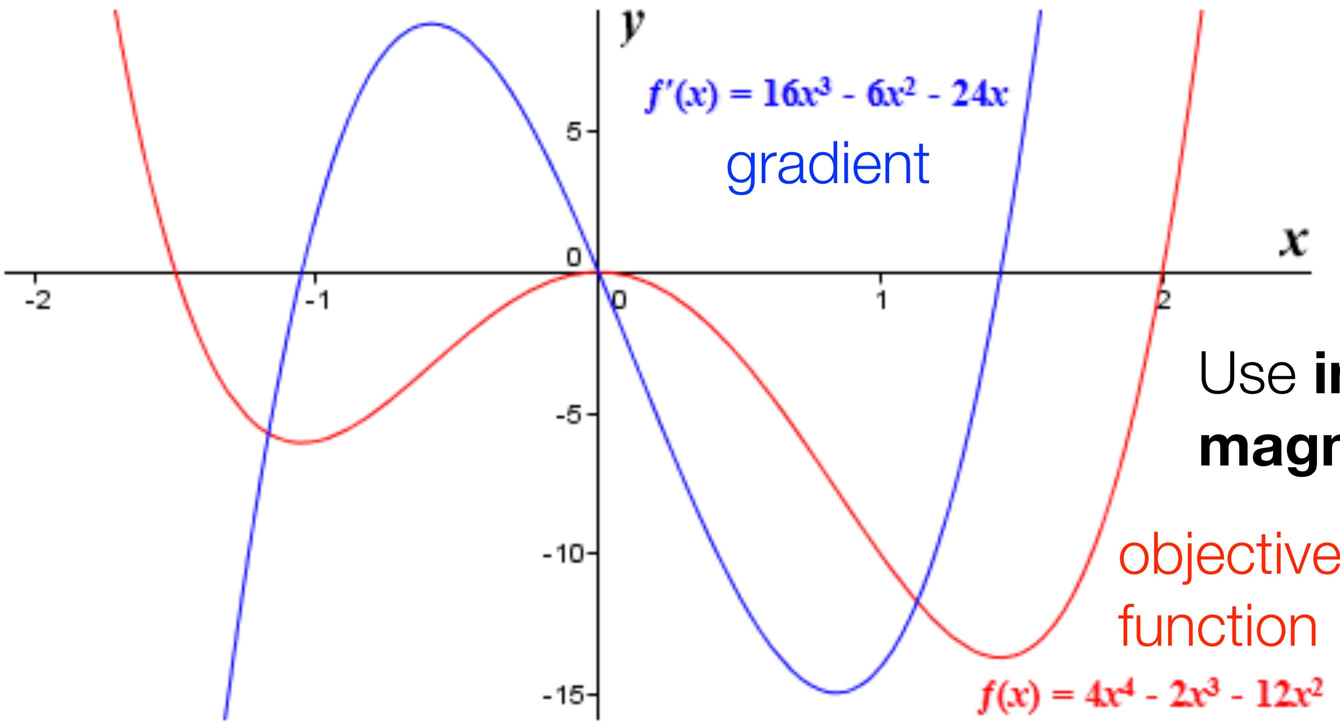
Fig. 2: NumPy is the base of the scientific Python ecosystem.

From: Array programming with NumPy



Be adaptive based on Gradient Magnitude?

- Decelerate down regions that are steep
- Accelerate on plateaus



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

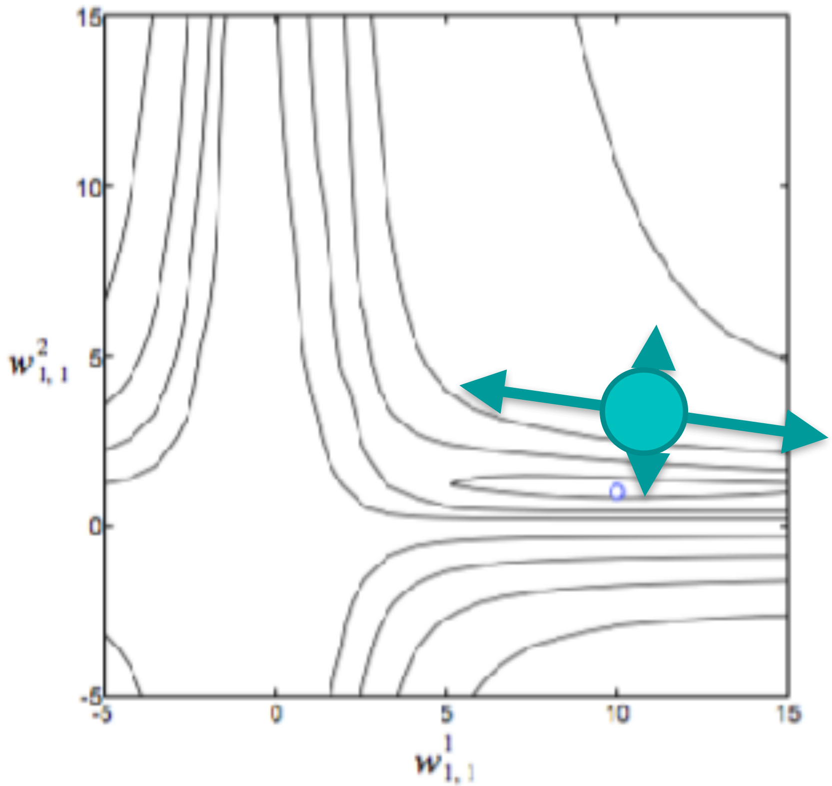
Also **accumulate inverse** to be robust to **abrupt changes** in **steepness**… momentum!!

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

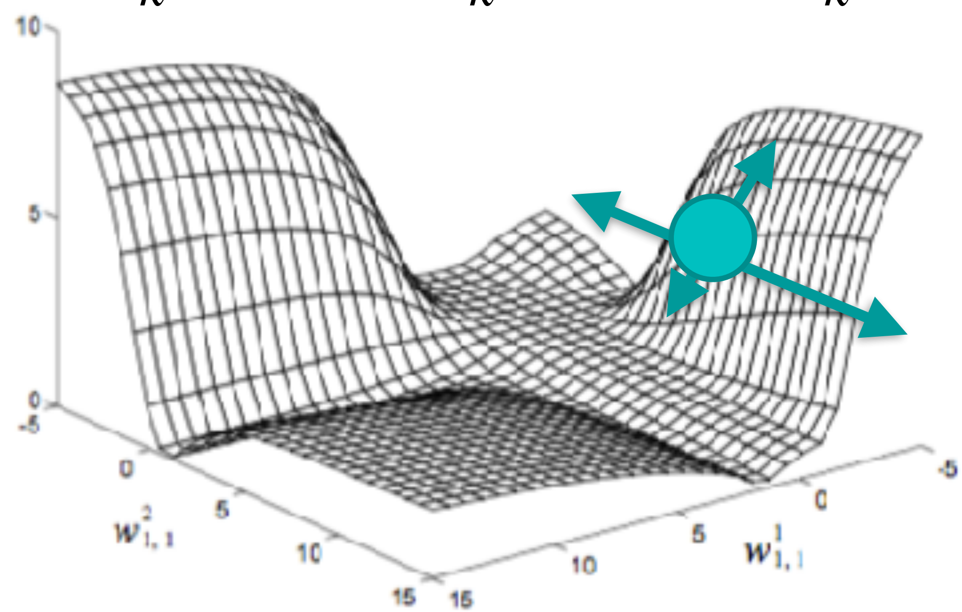
Be adaptive based on Gradient Magnitude?

Inverse magnitude of gradient in multiple directions?

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



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



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

$$\mathbf{G}_k = \mathbf{G}_{k-1} + \nabla J(\mathbf{W}_k) \odot \nabla J(\mathbf{W}_k)$$

- RMSProp

all operations are per element

$$\rho_k = \frac{1}{\sqrt{V_k + \epsilon}} \odot \nabla J(\mathbf{W}_k)$$

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

$$\mathbf{V}_k = \gamma \cdot \mathbf{V}_{k-1} + (1 - \gamma) \cdot \mathbf{G}_k$$

- AdaDelta

all operations are per element

$$\rho_k = \frac{\mathbf{M}_k}{\sqrt{\mathbf{V}_k + \epsilon}}$$

$$\mathbf{M}_k = \gamma \cdot \mathbf{M}_k + (1 - \gamma) \cdot \nabla J(\mathbf{W}_k)$$

- 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**