Lecture Notes for

Machine Learning in Python

Professor Eric Larson

Neural Network Optimization and Activation

Class Logistics and Agenda

- Agenda:
  - More optimization and architectures
  - Programming Examples

Last Time

Problems with Advanced Architectures

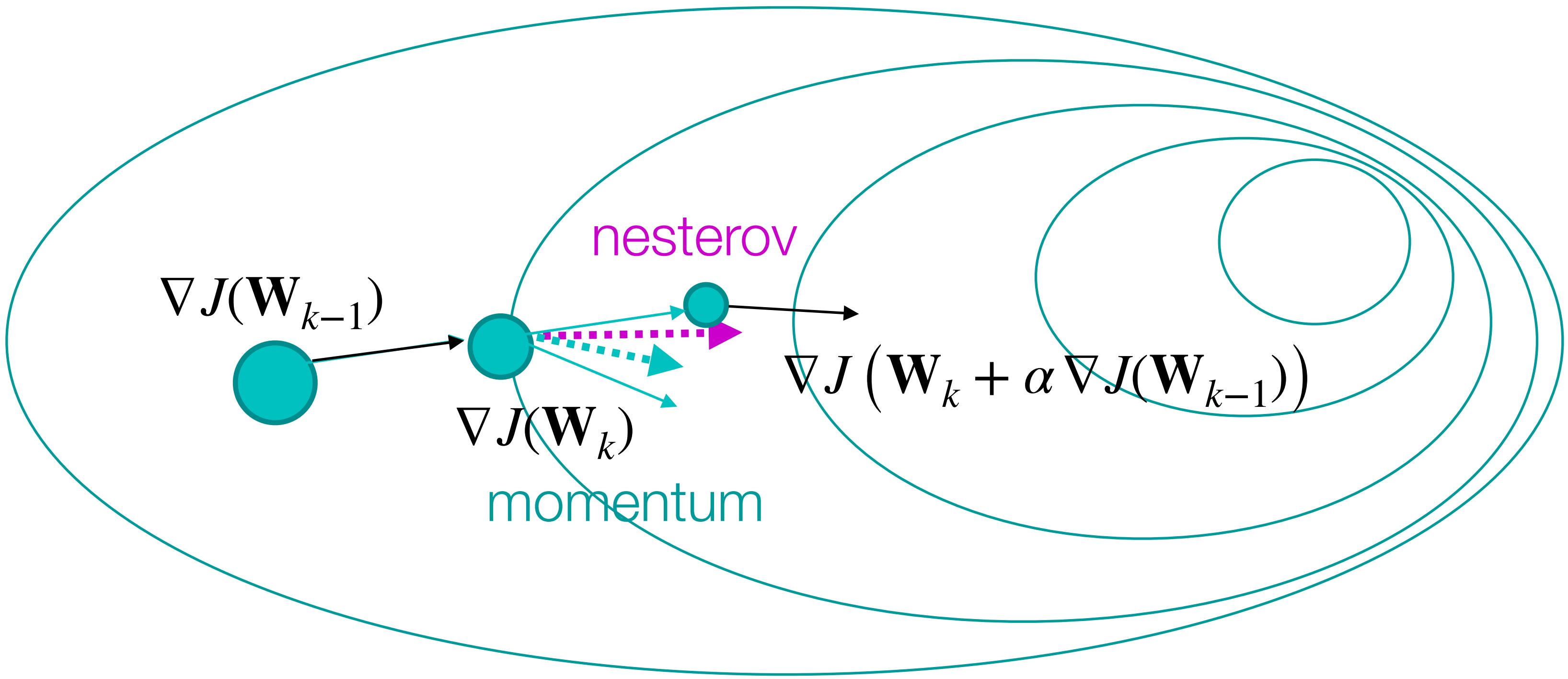
- Numerous weights to find gradient update
  - minimize number of instances
  - **solution**: mini-batch
- **new problem**: mini-batch gradient can be erratic
  - **solution**: momentum
    - use previous update in current update

Common Adaptive Strategies

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

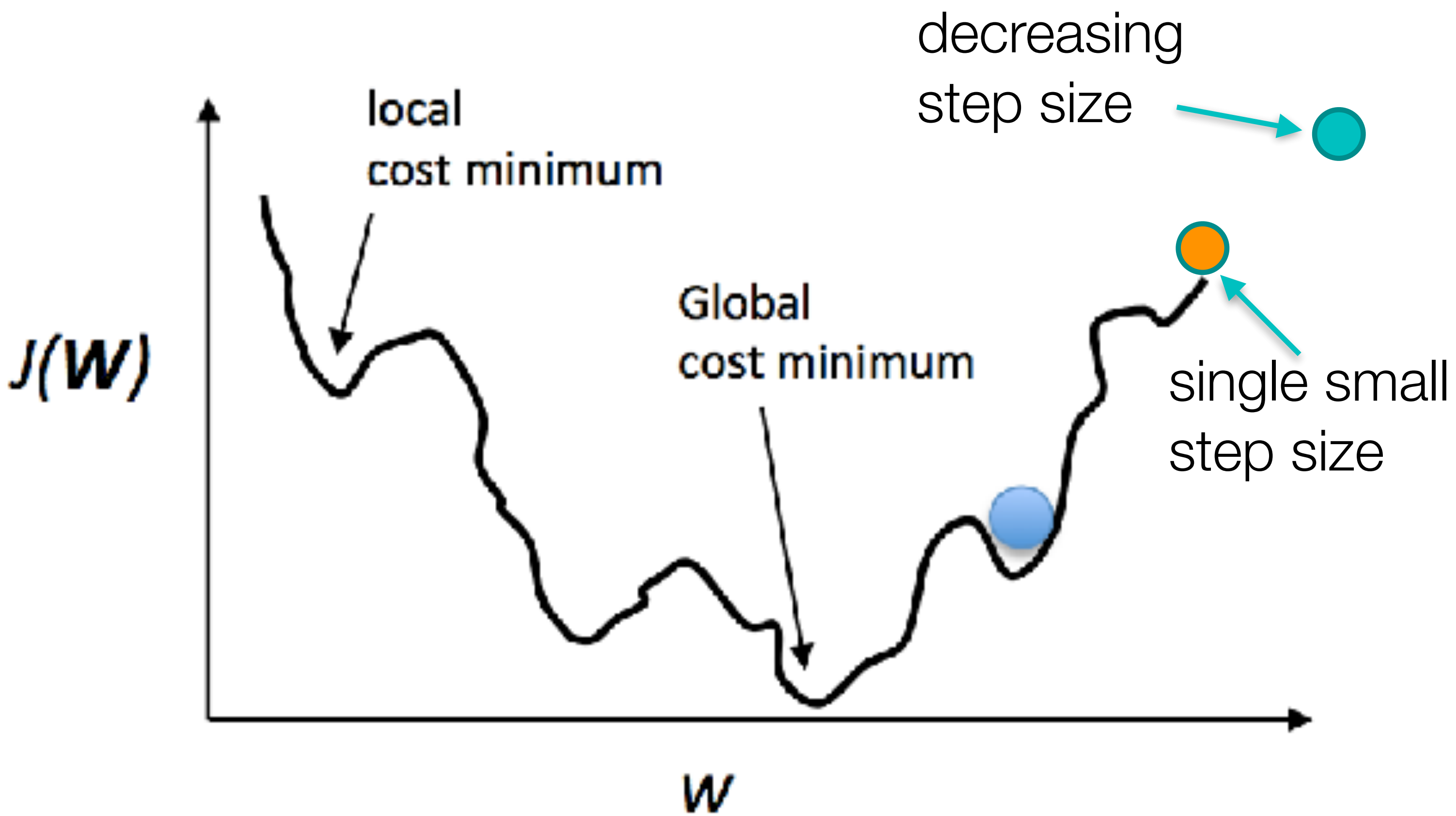
- Momentum $\rho_k = \alpha \nabla J(\mathbf{W}_k) + \beta \nabla J(\mathbf{W}_{k-1})$

- 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

- Space is no longer convex
  - **One solution**:
    - start with large step size
    - “cool down” by decreasing step size for higher iterations

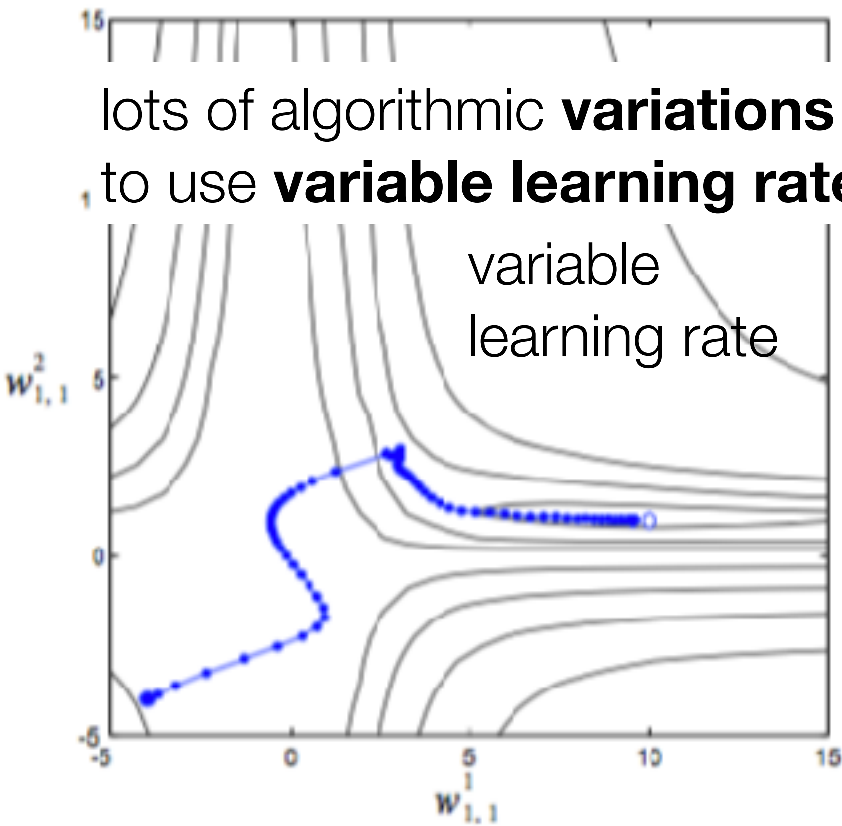


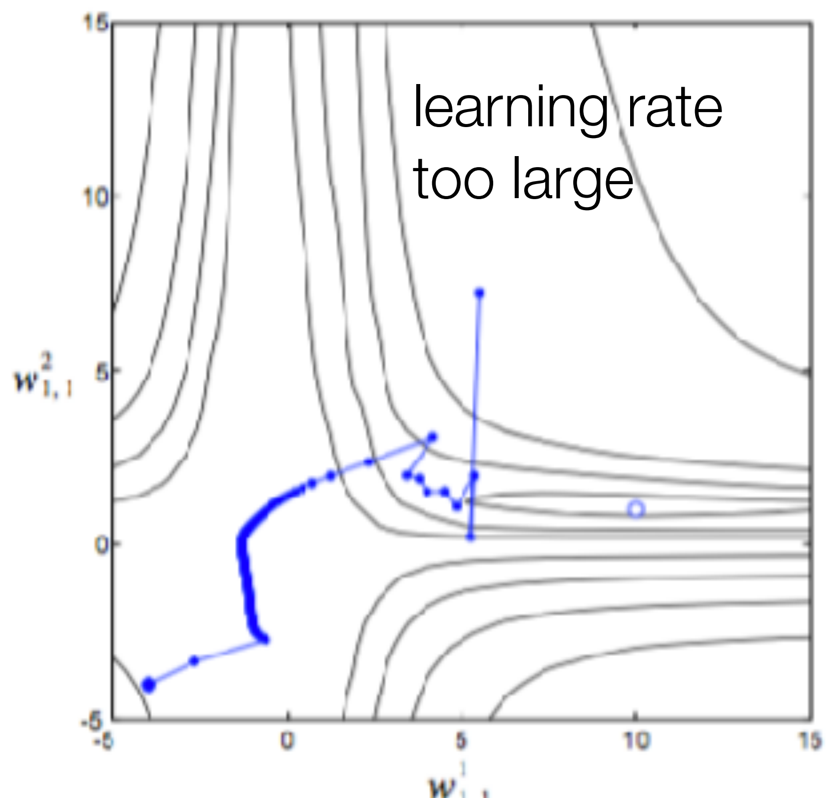
Adaptive Strategies

- Space is no longer convex
  - **another solution**:
    - start with arbitrary step size
    - only decrease when successive iterations do not decrease cost

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \frac{\eta}{1 + \epsilon \cdot k} \cdot \rho_k$$

lots of algorithmic **variations** to use **variable learning rate**





Neural Network Design, Hagan, Demuth, Beale, and De Jesus

Demo

`07. MLP Neural Networks.ipynb`

comparison:

mini-batch

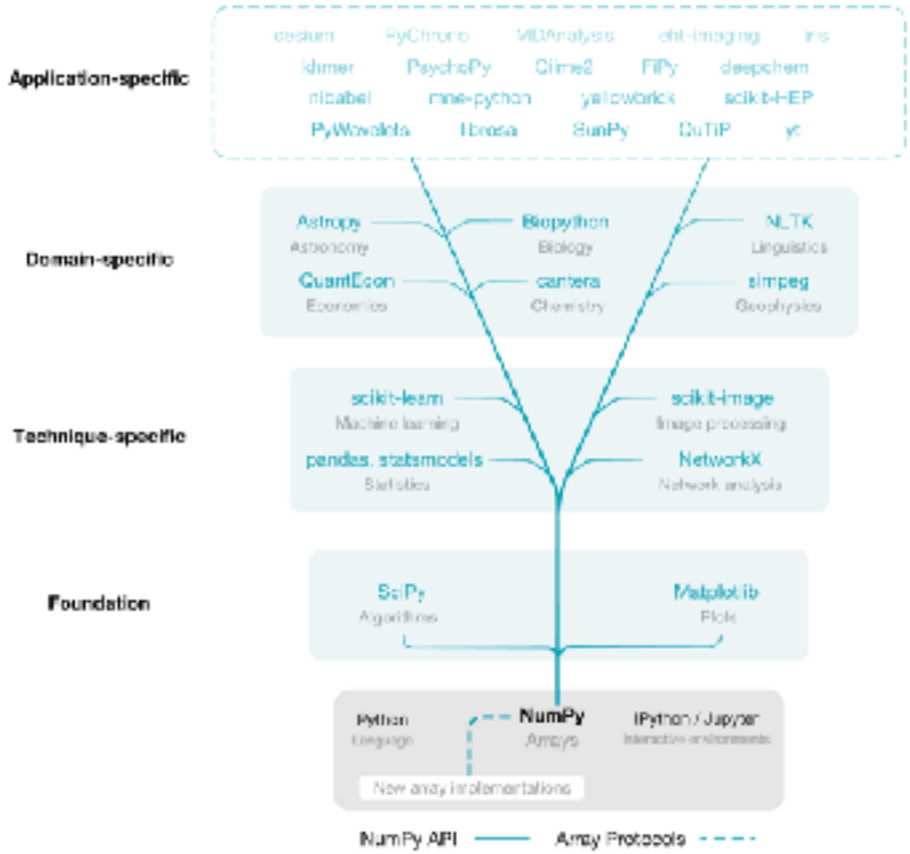
- momentum
- adaptive learning

L-BFGS

Adaptive Optimization

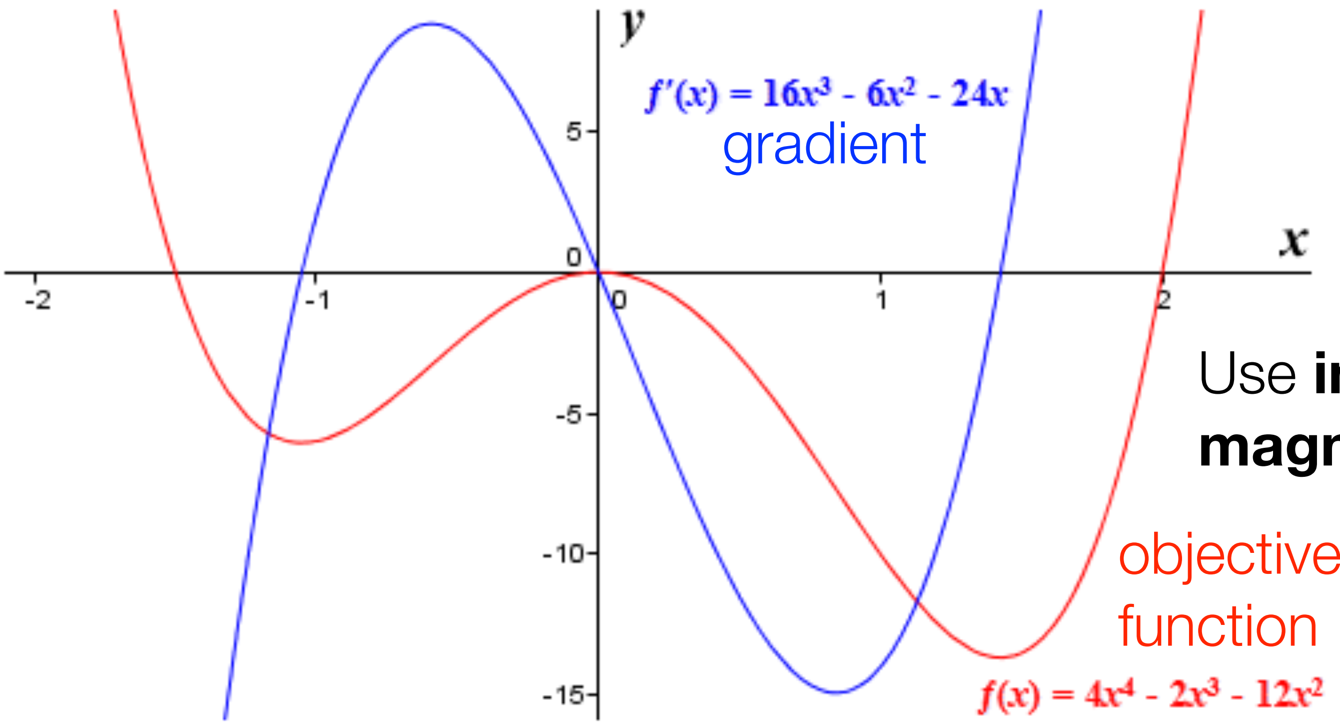
Fig. 2: NumPy is the base of the scientific Python ecosystem.

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**…

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

Common Adaptive Strategies

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

- Momentum

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

- Nesterov's Accelerated Gradient

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

- AdaGrad

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

where

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

all operations are per element

- RMSProp

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

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

$$V_k = \gamma \cdot V_{k-1} + (1-\gamma) \cdot G_k$$

all operations are per element

- AdaDelta

$$\rho_k = \eta \frac{M_k}{\sqrt{V_k + \epsilon}}$$

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

all operations are per element

- AdaM — 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 epoch $\quad k \leftarrow k + 1$

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

accumulated gradient $\quad \mathbf{M}_k \leftarrow \beta_1 \cdot \mathbf{M}_{k-1} + (1-\beta_1)\cdot \mathbf{G}_k$

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

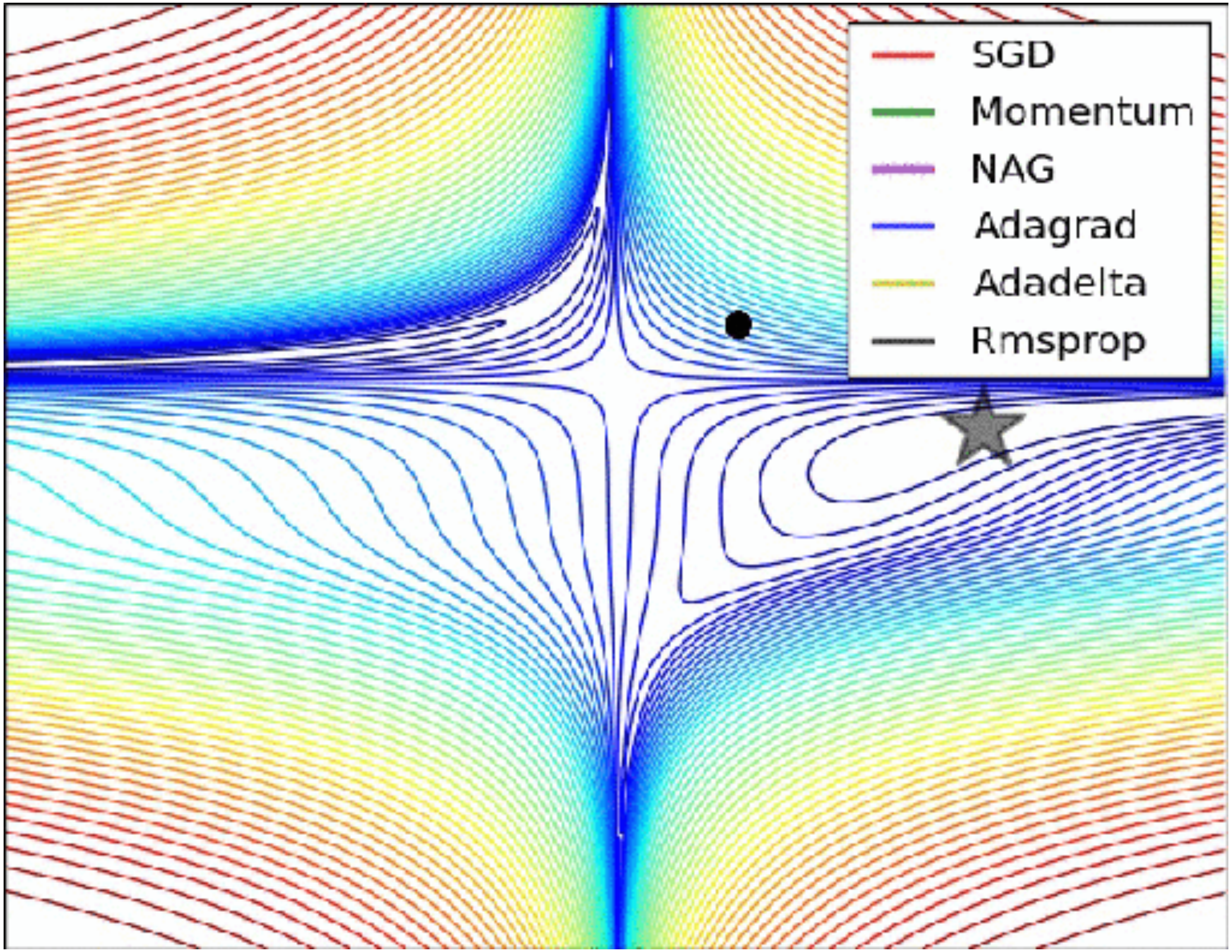
boost moments magnitudes (notice k in exponent)

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

update gradient, normalized by second moment similar to AdaDelta

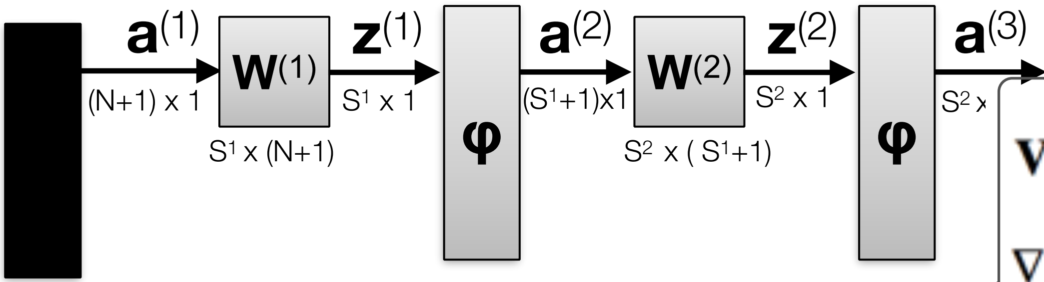
$$\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} - \eta \cdot \frac{\hat{\mathbf{M}}_k}{\sqrt{\hat{\mathbf{V}}_k + \epsilon}}$$

Visualization of Optimization



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

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

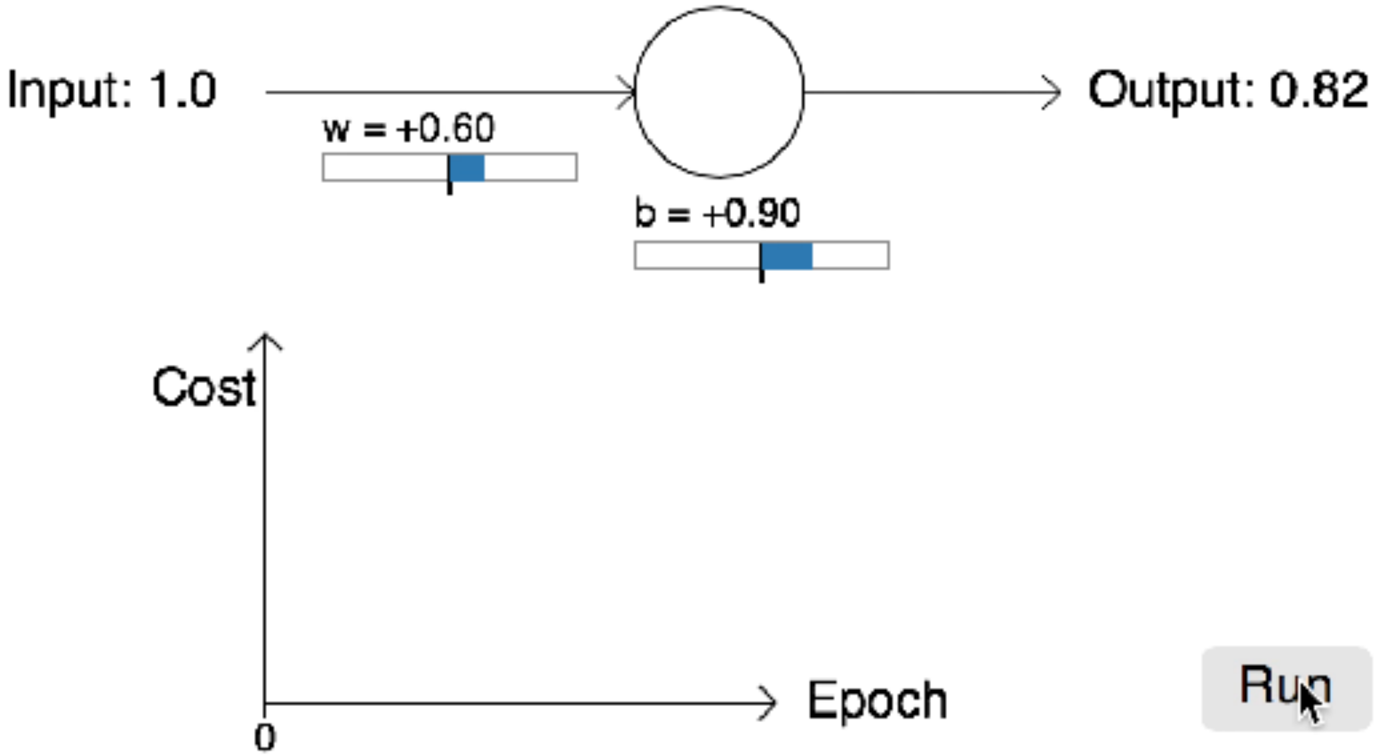
- **Self Test**:
  True or False: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer sensitivity calculation, $\mathbf{V}^{(2)}$, of the back propagation steps. The remainder of the algorithm is unchanged.
  - A. True
  - B. False

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



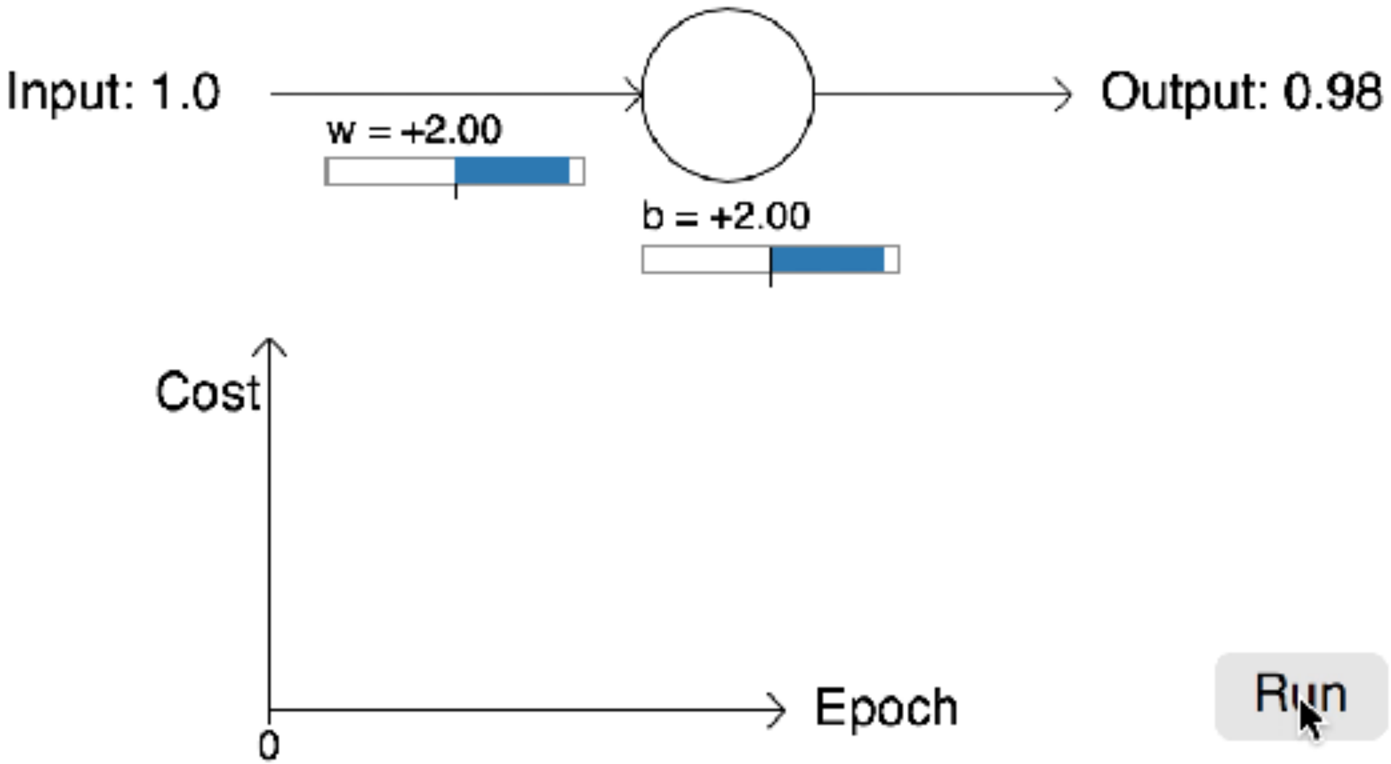
Neural Networks and Deep Learning, Michael Nielson, 2015

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

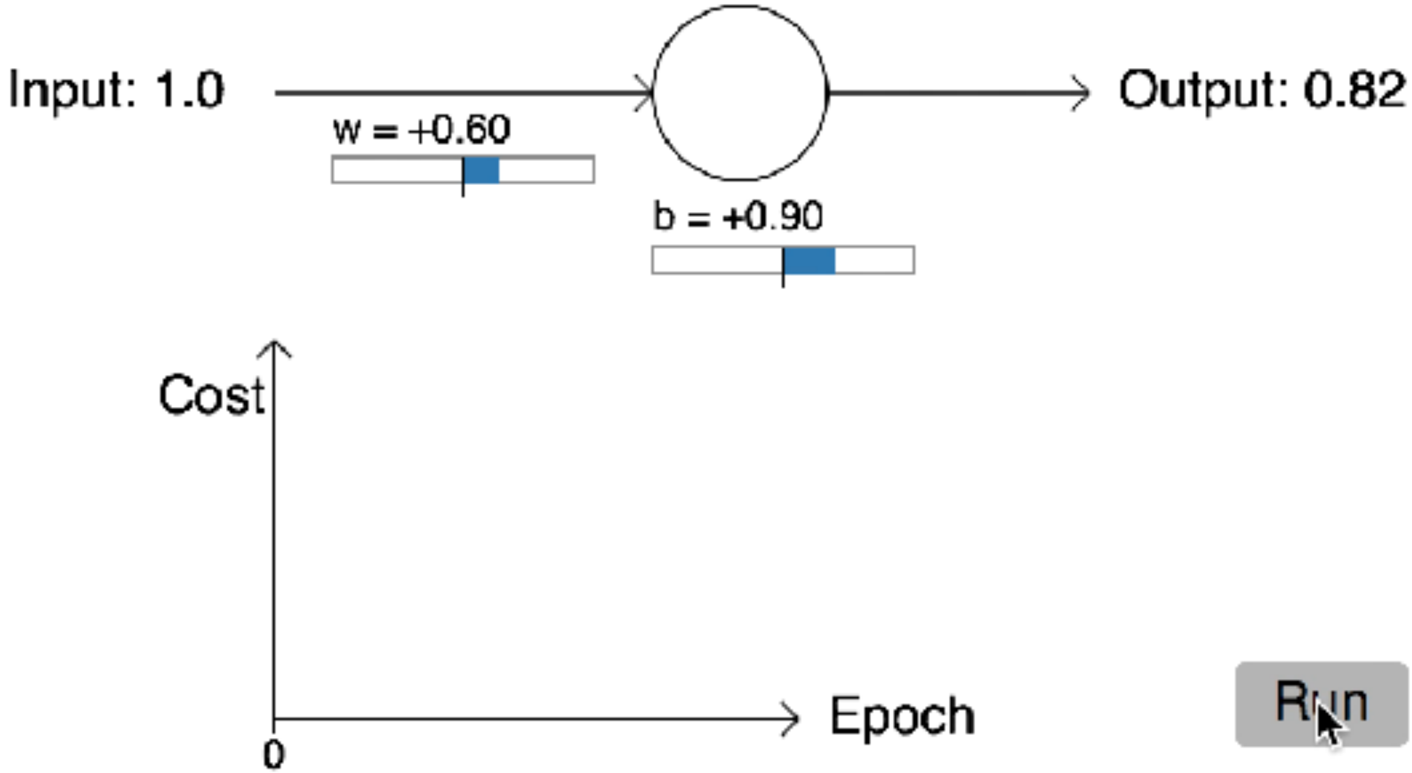


Neural Networks and Deep Learning, Michael Nielson, 2015

- 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



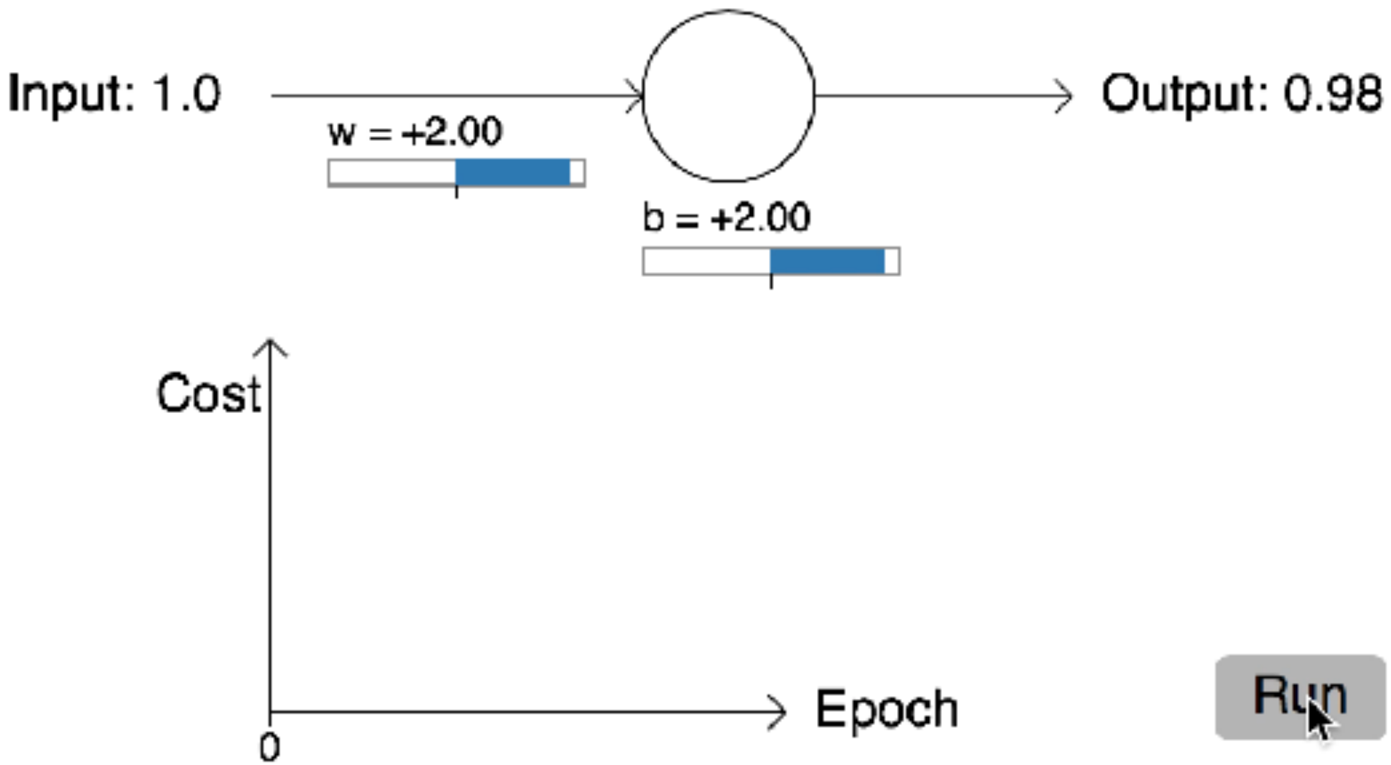
Neural Networks and Deep Learning, Michael Nielson, 2015

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

- 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

$$\left[\frac{\partial J(\mathbf{W})}{\mathbf{z}^{(L)}}\right]^{(i)}$$

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