Lecture Notes for

Machine Learning in Python

Professor Eric Larson

Adaptive Neural Network Optimization

Class Logistics and Agenda

- Agenda:
  - More optimization techniques
    - Review
    - Adaptive Learning Strategies
  - Town Hall for MLP

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

Review

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

- Cross entropy

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

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→

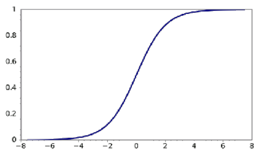
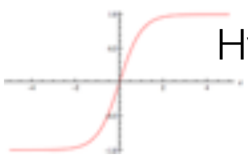
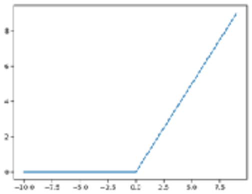
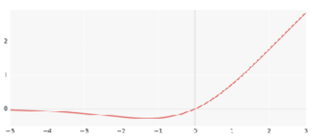
	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8	batch 9
Epoch 1									
Epoch 2									
Epoch 3									
Epoch 4									
...									

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

- Learning rate adaptation (eta)

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

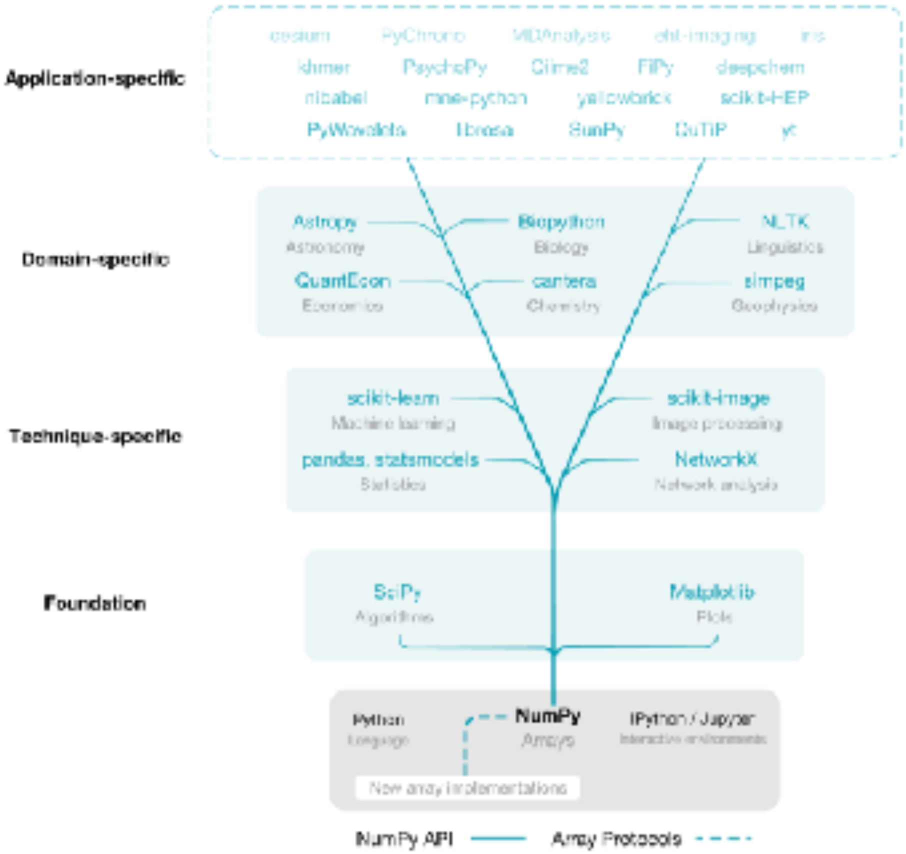
Review: Activations Summary

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

More Adaptive Optimization

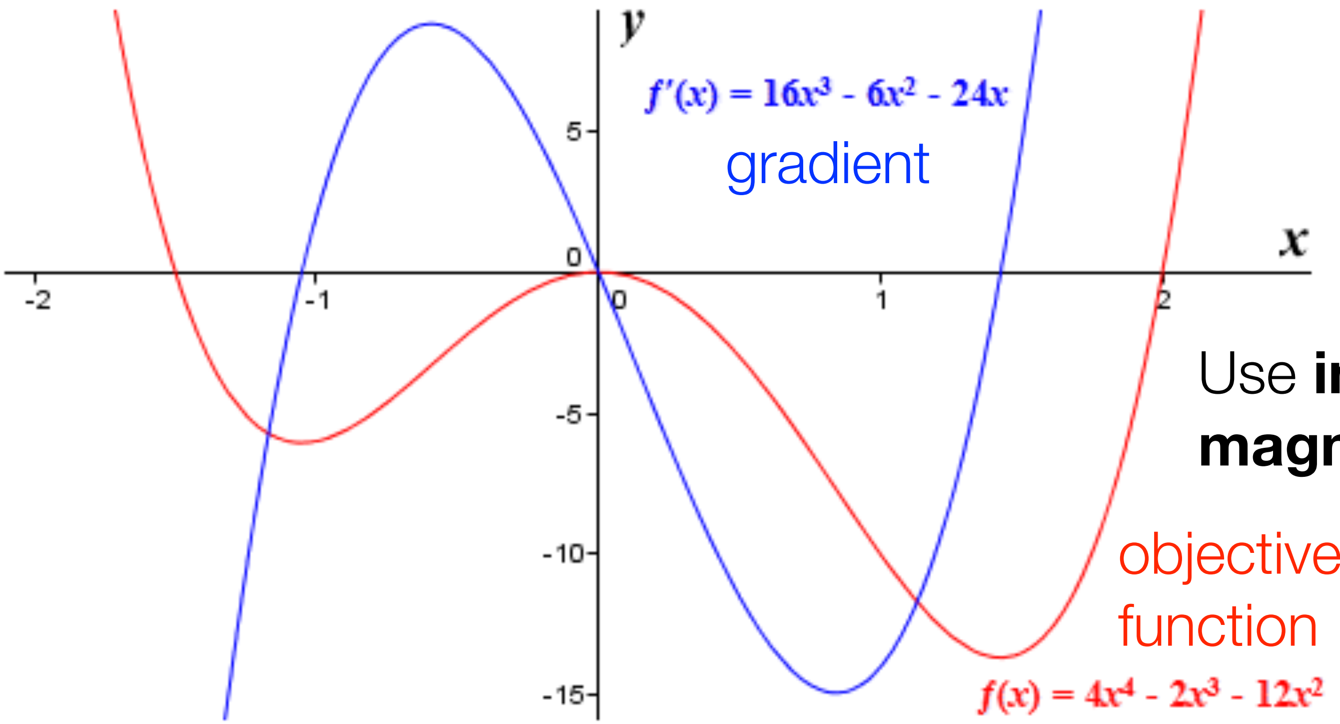
Fig. 2: NumPy is the base of the scientific Python ecosystem.

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**!

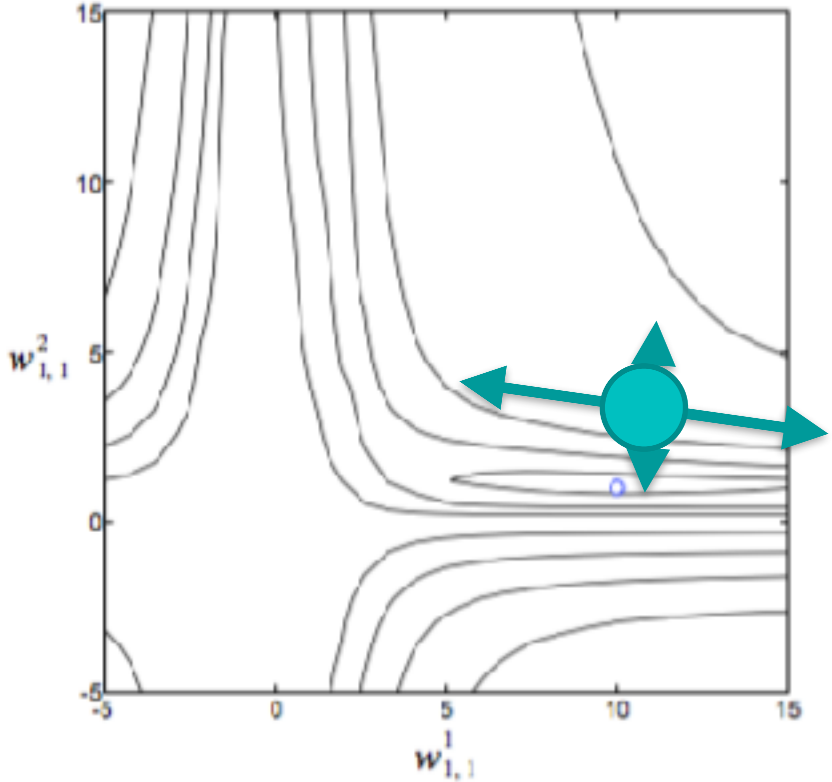
Also **accumulate inverse** to be robust to **abrupt changes** in **steepness**… momentum!!

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

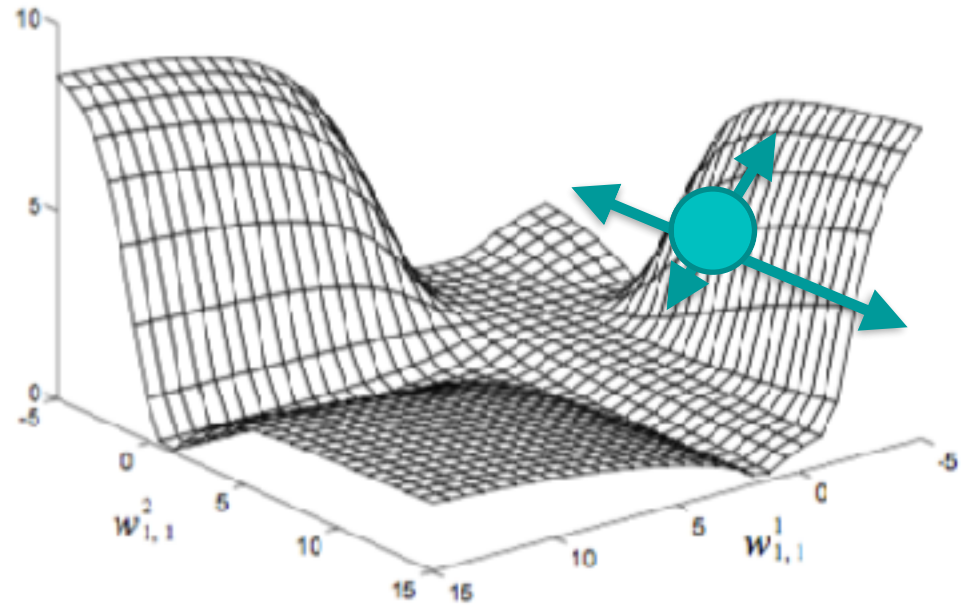
Be adaptive based on Gradient Magnitude?

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

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

where

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

all operations are per element

- RMSProp

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

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

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

all operations are per element

- AdaDelta

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

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

all operations are per element

- 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 epoch $k \leftarrow k + 1$

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

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)

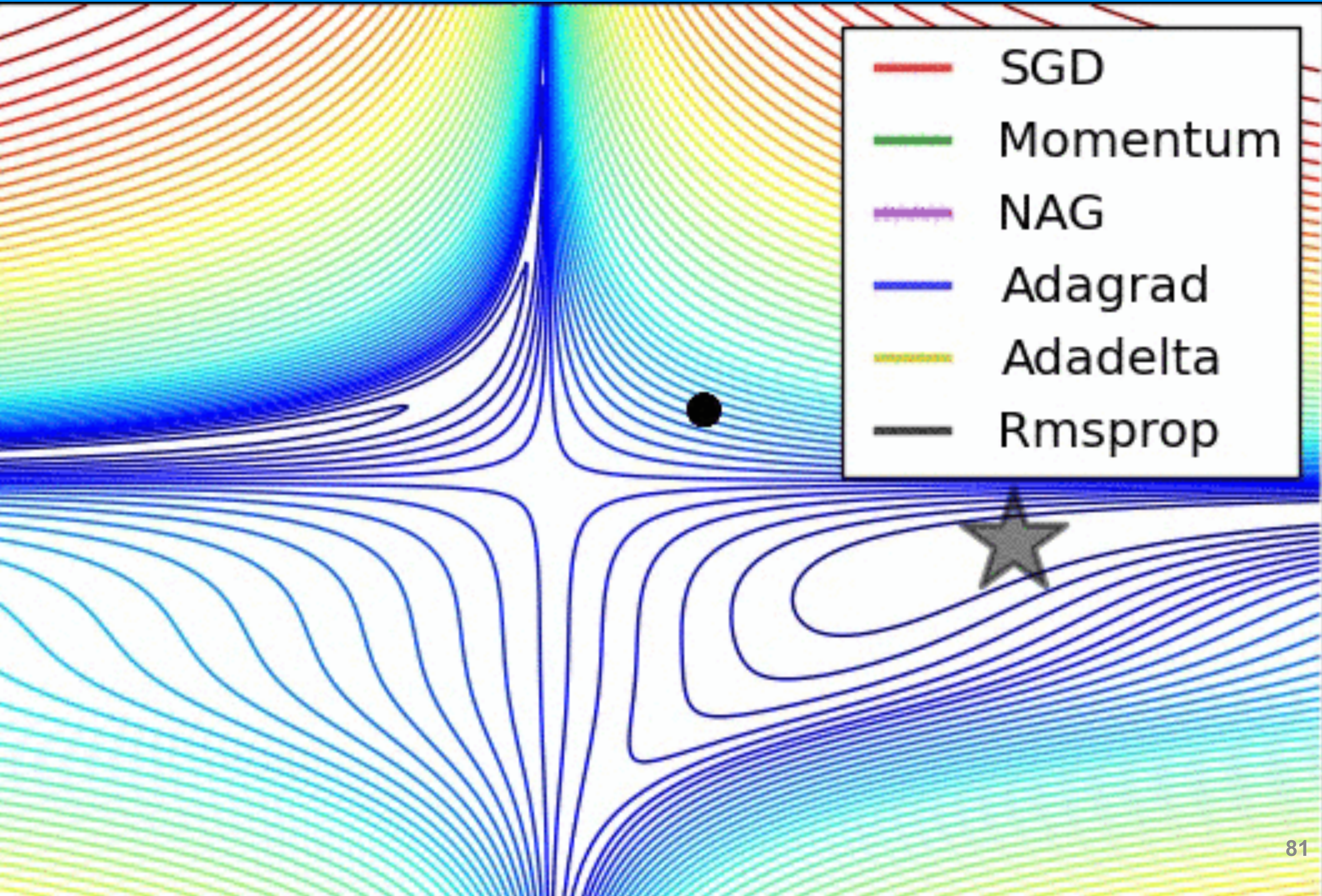
$$\hat{\mathbf{M}}_k \leftarrow \frac{\mathbf{M}_k}{(1 - [\beta_1]^k)} \qquad \hat{\mathbf{V}}_k \leftarrow \frac{\mathbf{V}_k}{(1 - [\beta_2]^k)}$$

update gradient, normalized by second moment similar to AdaDelta

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

Visualization of Optimization

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

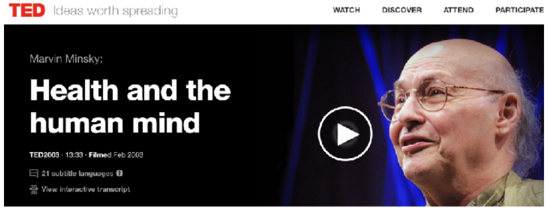


Practical Details

- Neural networks can separate any data through multiple layers. The true realization of Rosenblatt:

"Given an elementary α-perceptron, a stimulus world W, and any classification C(W) for which a solution exists; let all stimuli in W occur in any sequence, provided that each stimulus must reoccur in finite time; then beginning from an arbitrary initial state, an error correction procedure will always yield a solution to C(W) in finite time…”

- **Universality**: No matter what function we want to compute, we know that there is a neural network which can do the job.



- One nonlinear hidden layer with an output layer can perfectly train any problem with enough data, but might just be memorizing…
  - … it might be better to have even more layers for decreased computation and generalizability

End of Session

- Next Time: Final Flipped Module!
- Then: Deep Learning in Keras

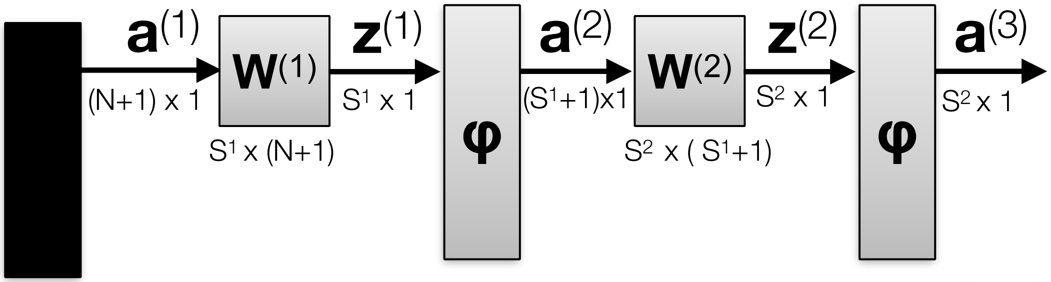
Town Hall

Back Up Slides

Back propagation

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

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \boxed{\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}}}$$



use chain rule: $$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} \boxed{\frac{\partial \mathbf{z}^{(l)}}{\partial w_{i,j}^{(l)}}}$$

BP-1

$$\frac{\partial z^{(l)}}{\partial w^{(l)}_{i,j}} = \begin{bmatrix} \frac{\partial z_1^{(l)}}{\partial w_{i,j}^{(l)}} \\ \vdots \\ \frac{\partial z_{s^l}^{(l)}}{\partial w_{i,j}^{(l)}} \end{bmatrix}$$

$$z^{(l)} = W^{(l)} a^{(l)}$$

MATRIX

$$z_k^{(l)} = \underbrace{w_k^{(l)}}_{k^{TH}\ \text{ROW}} a^{(l)}$$

$$= \sum_{j=1}^{s^l} w_{k,j}^{(l)} a_j^{(l)}$$

$$\frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \sum_{j=1}^{s^l} \frac{\partial}{\partial w_{i,j}^{(l)}} a_j^{(l)} w_{k,j}^{(l)}$$

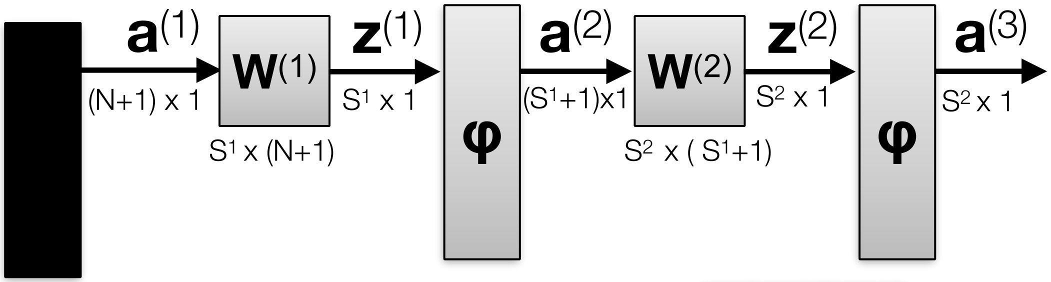
$$= 0 \text{ if } k \neq i$$

$$= a_j^{(l)} \text{ if } k = i$$

$$\boxed{\frac{\partial z^{(l)}}{\partial w_{i,j}^{(l)}} = a_j^{(l)}} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ a_j^{(l)} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow i^{TH} \text{ POSITION}$$

Back propagation

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



$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} a_j^{(l)}$$

can we use chain rule again?

$$\frac{\partial J(\mathbf{W})}{\partial w_{i,j}^{(l)}} = \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} a_j^{(l)}$$

- Steps:
  - propagate weights forward
  - calculate gradient at final layer
  - back propagate gradient for each layer
    - via recurrence relation

BP-2, setup $J(W)/\partial z^{(l+1)}$

CHAIN RULE AGAIN

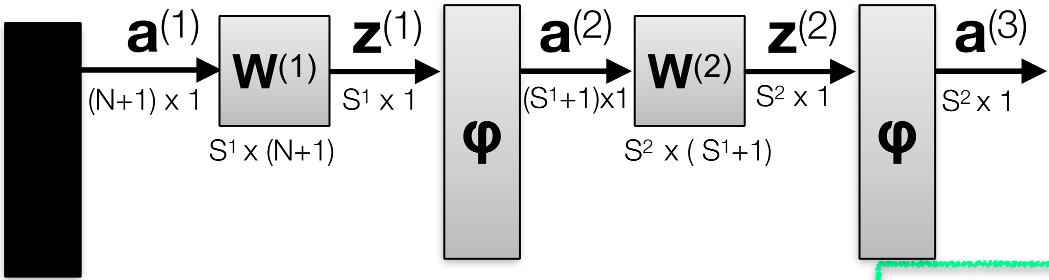
$$\frac{\partial J(W)}{\partial z^{(l)}} = \underbrace{\frac{\partial J(W)}{\partial z^{(l+1)}}}_{\substack{\text{VECTOR} \\ 1\times s^{(l+1)}}} \underbrace{\frac{\partial z^{(l+1)}}{\partial z^{(l)}}}_{\substack{\text{MATRIX} \\ s^{(l+1)}\times s^{(l)}}}$$

$$\begin{bmatrix} \frac{\partial J(W)}{\partial z_1^{(l+1)}} \\ \vdots \\ \frac{\partial J(W)}{\partial z_{s^{(l+1)}}^{(l+1)}} \end{bmatrix} \quad \begin{bmatrix} \frac{\partial z_1^{(l+1)}}{\partial z_1^{(l)}} & \frac{\partial z_2^{(l+1)}}{\partial z_1^{(l)}} & \cdots & \frac{\partial z_{s^{(l+1)}}^{(l+1)}}{\partial z_1^{(l)}} \\ & \ddots & & \\ \frac{\partial z_1^{(l+1)}}{\partial z_{s^{(l)}}^{(l)}} & \cdots & & \frac{\partial z_{s^{(l+1)}}^{(l+1)}}{\partial z_{s^{(l)}}^{(l)}} \end{bmatrix}$$

$$\frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} = \frac{\partial}{\partial z_j^{(l)}} \sum_{k=1}^{s^{(l+1)}} w_{i,k}^{(l+1)} a_k^{(l+1)}$$

Back propagation

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



$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} a_j^{(l)}$$

can we get this gradient?

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}} \boxed{\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}}}$$

BP-3

$$\frac{\partial z_i^{(\ell+1)}}{\partial z_j^{(\ell)}} = \frac{\partial}{\partial z_j^{(\ell)}} \left(\sum_{k=1}^{s^{(\ell+1)}} w_{i,k}^{(\ell+1)} a_k^{(\ell+1)} \right)$$

$$= w_{i,j}^{(\ell+1)} \frac{\partial}{\partial z_j^{(\ell)}} \phi(z_j^{(\ell)})$$

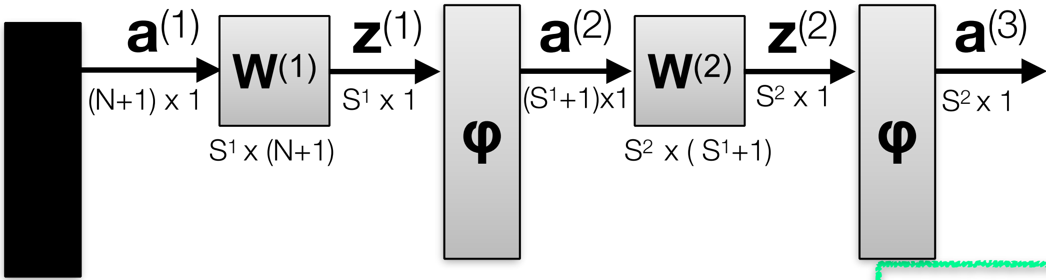
$$= w_{i,j}^{(\ell+1)} a_j^{(\ell+1)} (1 - a_j^{(\ell+1)})$$

- - - - - - - - - - - - - - - - - - - -

$$\frac{\partial z^{(\ell+1)}}{\partial z^{(\ell)}} = \underbrace{\cancel{\underbrace{W^{(\ell+1)}}_{s^{\ell+1} \times s^{\ell}}} \; \underbrace{\underset{j}{\operatorname{diag}}^{s^{\ell+1}} \left(a_j^{(\ell+1)} (1 - a_j^{(\ell+1)}) \right)}_{s^{(\ell+1)} \times s^{(\ell+1)}}}_{\text{ENTIRE MATRIX } s^{\ell+1} \times s^{\ell}} \; \underbrace{W^{(\ell+1)}}_{s^{\ell+1} \times s^{\ell}}$$

Back propagation

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



$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} a_j^{(l)}$$

use chain rule: $\dfrac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} = \dfrac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}} \boxed{\dfrac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}}}$

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}} = \mathrm{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)}$$

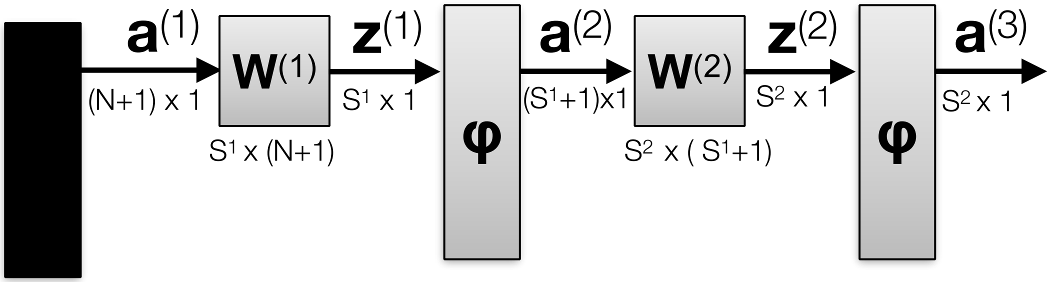
$$\boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}}} = \mathrm{diag}[\mathbf{a}^{(l+1)} * (1 - \mathbf{a}^{(l+1)})] \cdot \mathbf{W}^{(l+1)} \boxed{\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}}$$

recurrence relation

If we know last layer, we can **back propagate** towards previous layers!

Back propagation

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



$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l)}} a_j^{(l)}$$

one more step
need last layer gradient:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = \frac{\partial}{\partial \mathbf{z}^{(2)}} (\mathbf{y}^{(k)} - \phi(\mathbf{z}^{(2)}))^2$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(l+1)}}$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{z}^{(2)}} = -2(\mathbf{y}^{(k)} - \mathbf{a}^{(3)}) * \mathbf{a}^{(3)} * (1 - \mathbf{a}^{(3)})$$

Final Layer BP-4

$$\frac{\partial J(W)}{\partial z_i^{(L)}} = \frac{\partial}{\partial z_i^{(L)}} \sum_{k=1}^{M} \left(y^{(k)} - \phi(z^{(L)})\right)^2$$

$$= -\sum_k 2\left(y^{(k)} - a^{(L+1)}\right) \frac{\partial}{\partial z_i^{(L)}} \phi(z^{(L)})$$

$$= -2\sum_k \left(y^{(k)} - a^{(L+1)}\right) a_i^{(L+1)}\left(1-a_i^{(L+1)}\right) \cancelto{1}{\frac{\partial}{\partial z_i^{(L)}} z^{(L)}}$$

$$= -2\sum_k \left(y^{(k)} - a^{(L+1)}\right) a_i^{(L+1)}\left(1-a_i^{(L+1)}\right)$$

$$\frac{\partial J}{\partial z^{(L)}} = -2\sum_k \left(y^{(k)} - a^{(L+1)}\right) * a^{(L+1)} * \left(1-a^{(L+1)}\right)$$

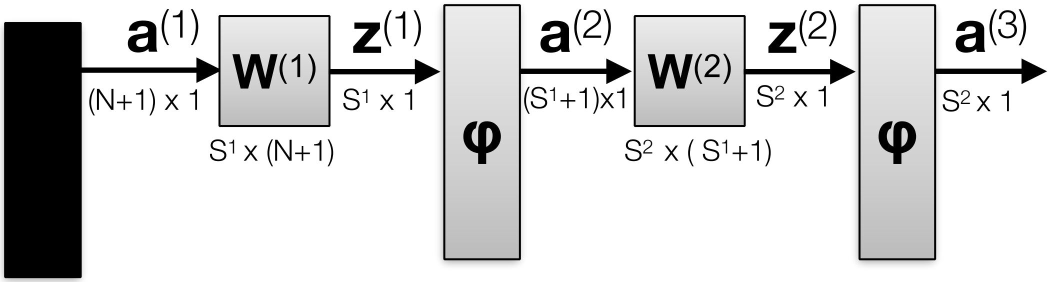
↑ONE DIM, TAKE OUT K

OR w/ layers

$$\frac{\partial J(W)}{\partial z^{(2)}} = \frac{\partial}{\partial z^{(2)}} \sum_k \left(y^{(k)} - \phi(z^{(2)})\right)^2$$

$$= -2\sum_k \left(y^{(k)} - a^{(3)}\right) * a^{(3)} * \left(1-a^{(3)}\right)$$

Back propagation summary



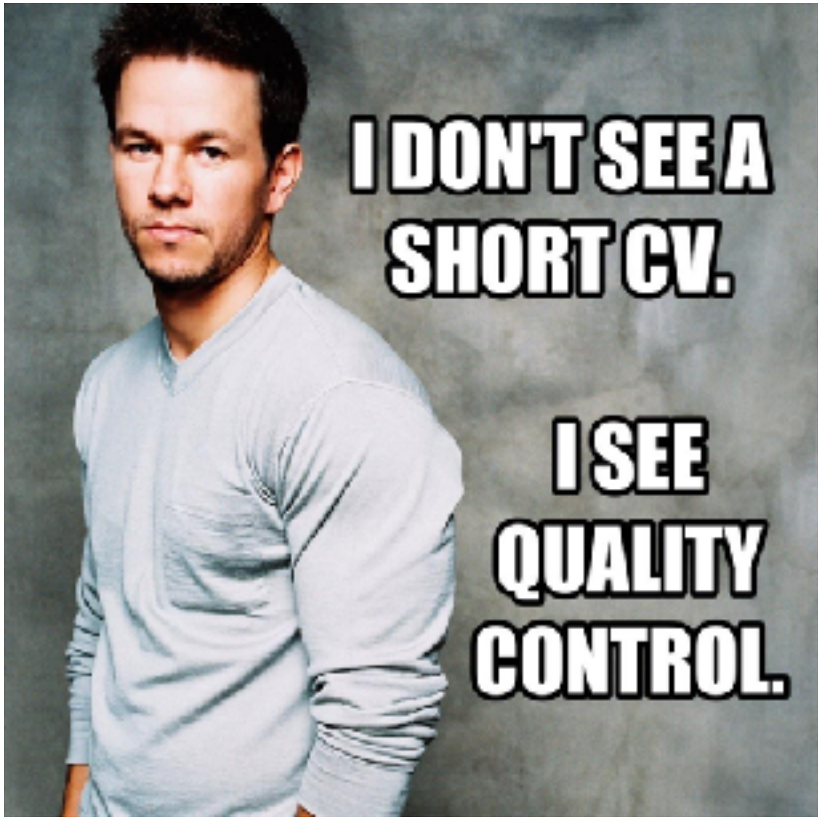
- **Self Test**:
  True or False: If we change the cost function, $J(\mathbf{W})$, we only need to update the final layer calculation of the back propagation steps. The remainder of the algorithm is unchanged.
  - A. True
  - B. False

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

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

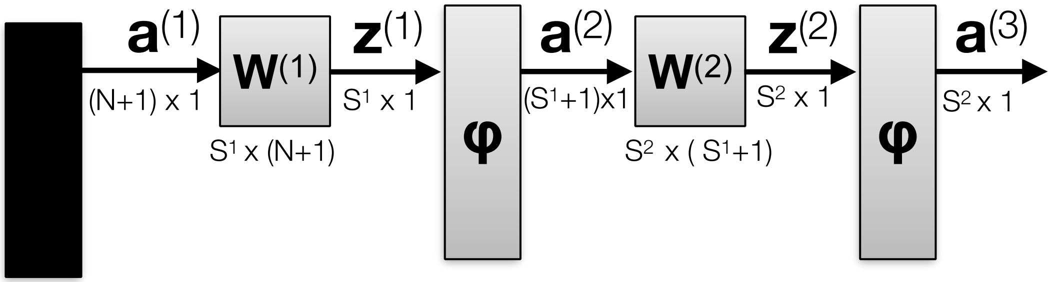
Programming Multi-layer Neural Networks

`07. MLP Neural Networks.ipynb`



Guided Example

Recall from the in-class asisgnment



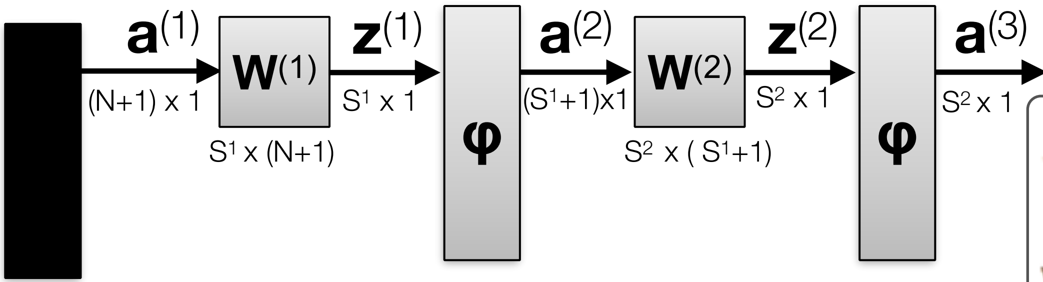
1. Forward propagate to get **Z**, **A**

```python
# feedforward all instances
A1, Z1, A2, Z2, A3 = self._feedforward(X_data,self.W1,self.W2)
```

```python
def _feedforward(self, X, W1, W2):
    A1 = self._add_bias_unit(X.T, how='row')
    Z1 = W1 @ A1
    A2 = self._sigmoid(Z1)
    A2 = self._add_bias_unit(A2, how='row')
    Z2 = W2 @ A2
    A3 = self._sigmoid(Z2)
    return A1, Z1, A2, Z2, A3
```

these are more than just vectors for **one instance**!
these are for **all** instances

Back propagation implementation



1. Forward propagate to get **Z**, **A**
2. Get final layer gradient
3. Back propagate sensitivities

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

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

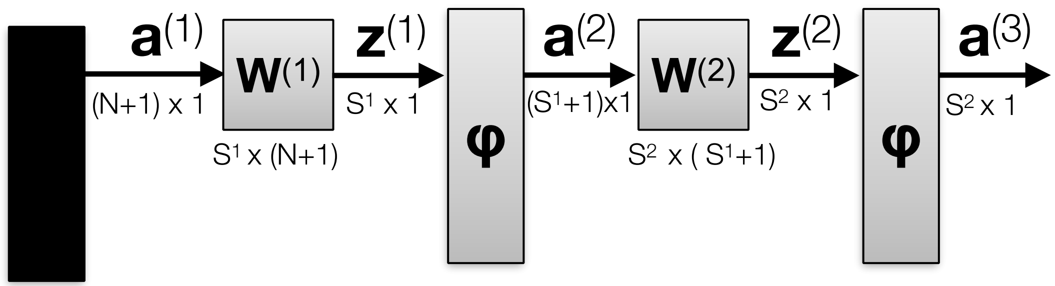
$$\mathbf{V}^{(1)} = \mathbf{A}^{(2)} * (1 - \mathbf{A}^{(2)}) * [\mathbf{W}^{(2)}]^T \cdot \mathbf{V}^{(2)}$$

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

```python
def _get_gradient(self, A1, A2, A3, Z1, Z2, Y_enc, W1, W2):
    """ Compute gradient step using backpropagation.
    """
    # vectorized backpropagation
    V2 = -2*(Y_enc-A3)*A3*(1-A3)
    V1 = A2*(1-A2)*(W2.T @ V2)

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

Back propagation implementation



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

for each layer:

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

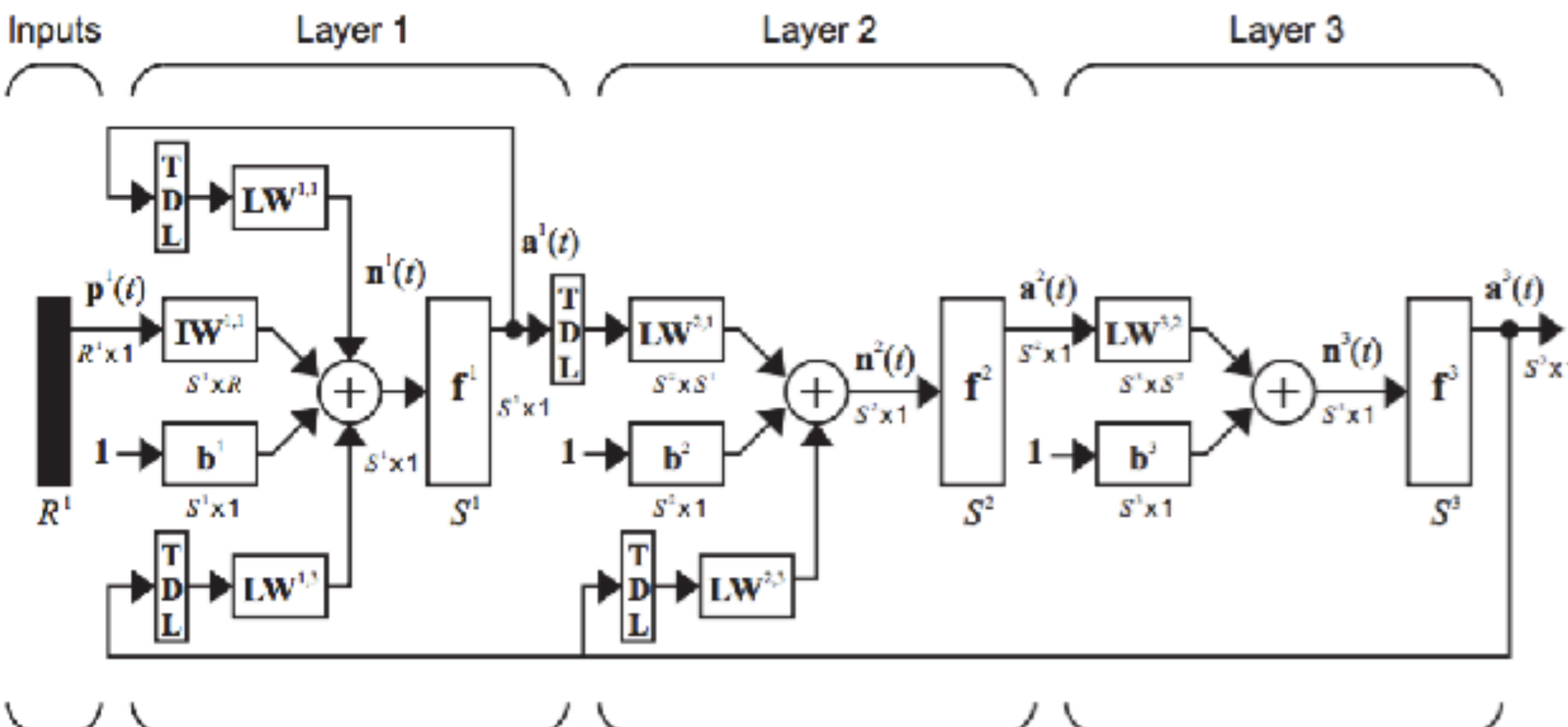
```python
# feedforward all instances
A1, Z1, A2, Z2, A3 = self._feedforward(X_data,self.W1,self.W2)

# compute gradient via backpropagation
grad1, grad2 = self._get_gradient(A1=A1, A2=A2,
                                  A3=A3, Z1=Z1,
                                  Y_enc=Y_enc,
                                  W1=self.W1, W2=self.W2)


self.W1 -= self.eta * grad1
self.W2 -= self.eta * grad2
```

More Advanced Architectures

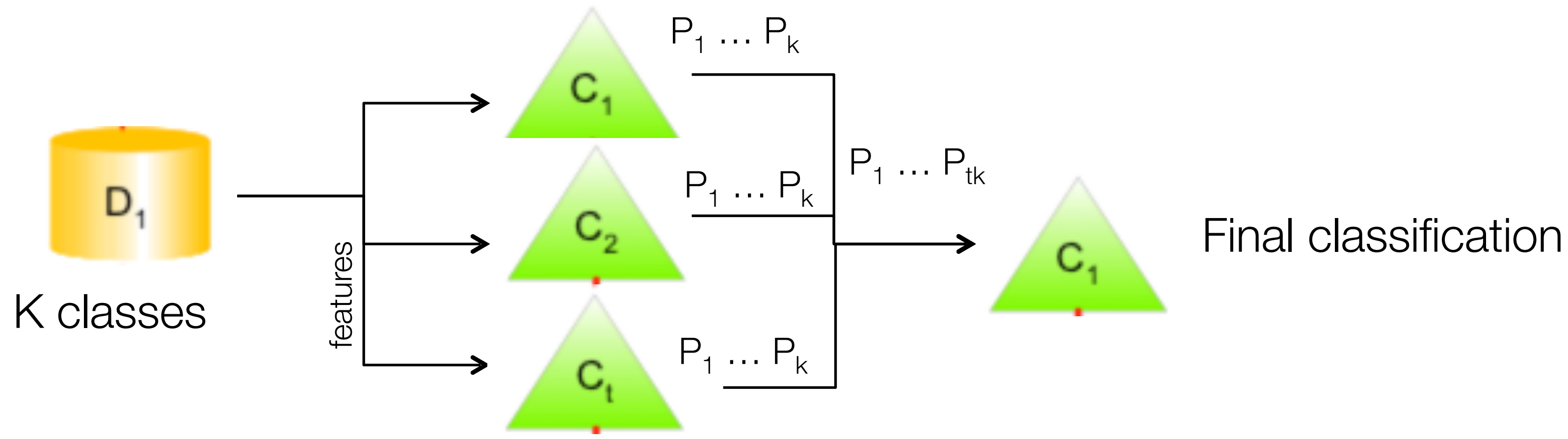
- Dynamic Networks (recurrent networks)
  - can use current and previous inputs, in time
  - still popular, but ultimately extremely hard to train
  - **highly successful variant**: long short term



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

Stacking and Cascading ensembles

- Train an initial classifier (or ensemble)
- Use the output probabilities of each class (from each classifier, if ensemble), as features for another classifier
- Train final classifier on inferences from previous classes



Self test

- How many features (attributes) does scikit-learn use in each iteration of bagging for a random forest?

- Which best completes this statement: In scikit, the number of features considered changes
  - a: once for each tree in the forest
  - b: once for each split in each tree

each iteration!!!

max_features=sqrt(n_features)

SVMs Summary: Linear

- Linear SVMs
  - Architecture identical to logistic regression, except:
    - maximize margin
    - constrained optimization
  - **Self Test**: What are the trained parameters for the linear SVM?
    - A: Coefficients of w
    - B: Intercept
    - C: Slack Variables
    - D: Support Vector locations

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

$$\text{to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, i = 1, \ldots, n$$

SVMs Summary: Linear

- Linear SVMs
  - Architecture near identical to logistic regression, except:
    - maximize margin
    - constrained optimization

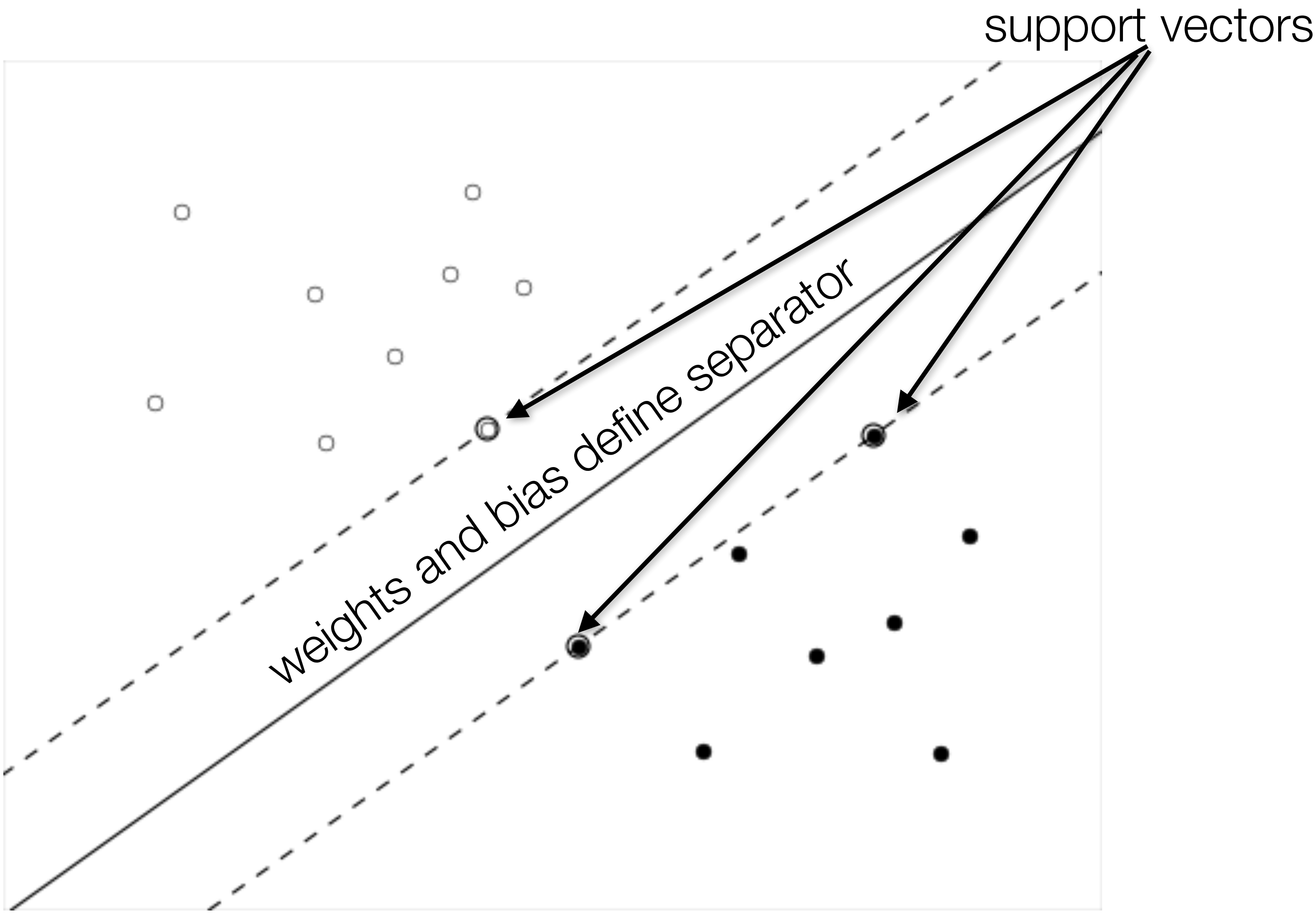
$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

$$\text{to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

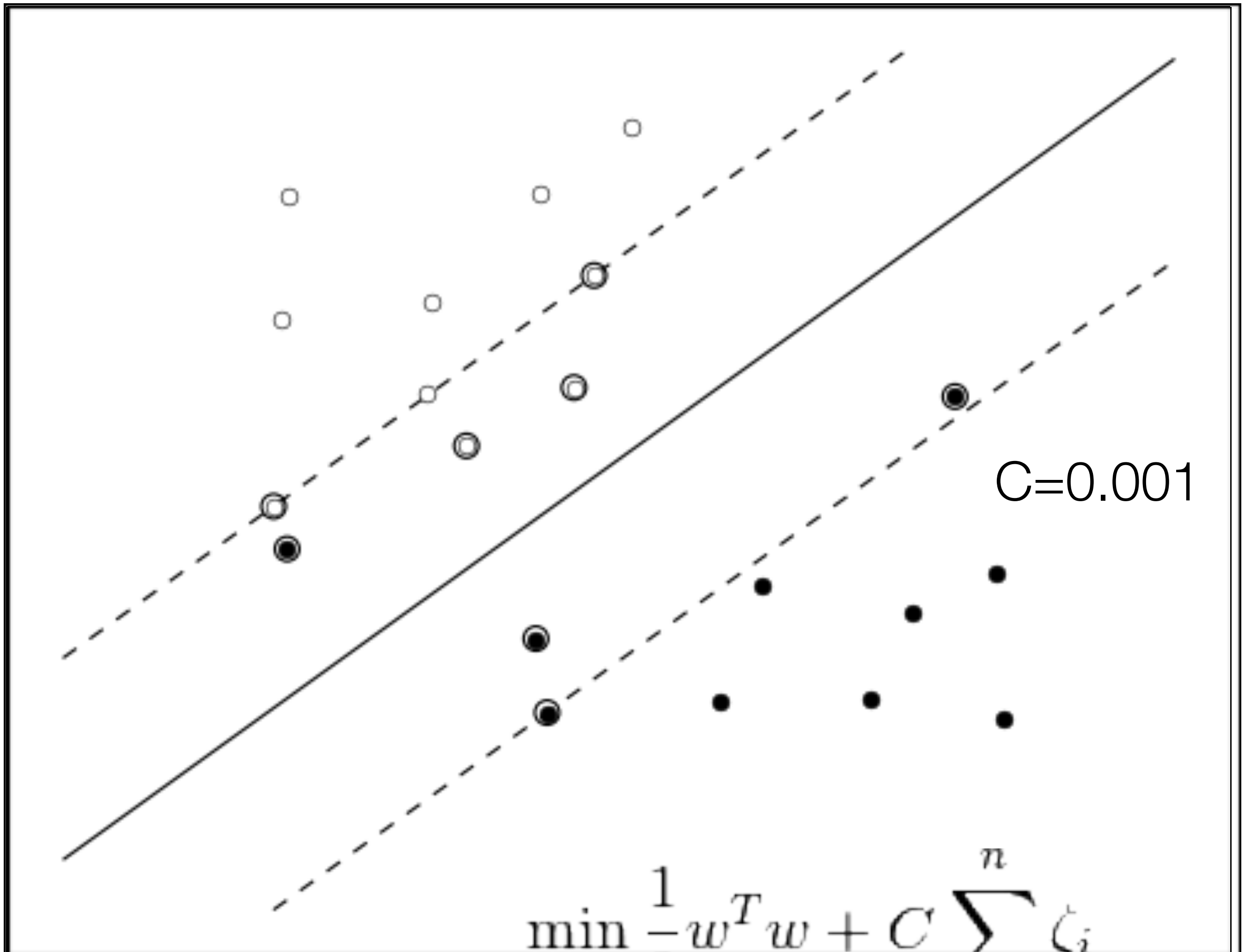
$$\zeta_i \geq 0, i = 1, ..., n$$

  - Trained Parameters:
    - intercept and weights for each class
    - support vectors chosen for margin calculation
    - slack variables only needed during training

SVMs Summary: Linear



SVMs Summary: Linear

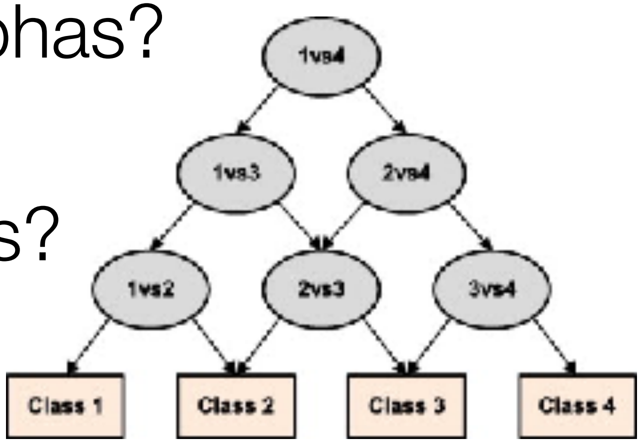


$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

tradeoff margin and accuracy

SVMs Summary: non-linear

- Non-linear SVMs
  - Architecture not like logistic regression
    - kernels == high dimensional dot-product
    - impossible to store weights
      - use kernel trick so no need to store them!
  - Trained Parameters
    - biases? how many?
    - selected support vectors? alphas?
    - slack variables?
    - parameters specific to kernels?



SVMs Summary: non-linear

- Popular Kernels
  - polynomial

gamma coef0 degree

$$(\gamma\langle x, x'\rangle + r)^d$$

  - radial basis function

gamma

$$\exp(-\gamma|x - x'|^2)$$

  - sigmoid

gamma coef0

$$\tanh(\gamma\langle x, x'\rangle + r)$$

SVM parameterization

Demo

svm_gui.py

The End of SVMs