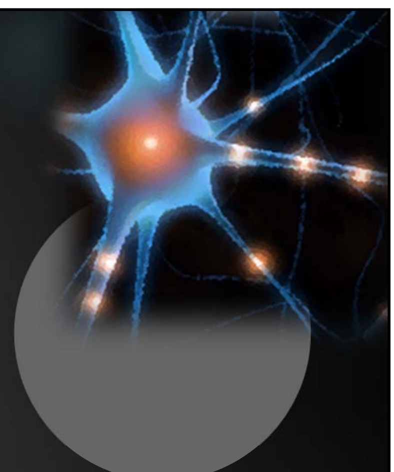
Deep Learning Applied

NEURAL ARCHITECTURES FOR THE SOCIAL SCIENCES

ALEXANDER G. ORORBIA II

THE PENNSYLVANIA STATE UNIVERSITY

SBP-BRIMS TUTORIAL 2016



Objectives

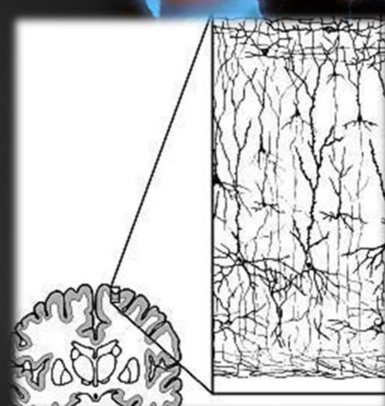
- Motivations
  - Why do we want to use neural architectures?
- Some preliminaries
  - A crash course in necessary mathematical basics
- Neural architectures
  - Basic relevant topologies
- Automatic differentiation: calculating parameter gradients
- Parameter optimization
- Hyper-parameter optimization
- Data pre-processing (text)
- An application: Automatic content coding
- Resources & references

Motivations

WHY DO WE WANT TO USE NEURAL ARCHITECTURES?

Why? Previous Results

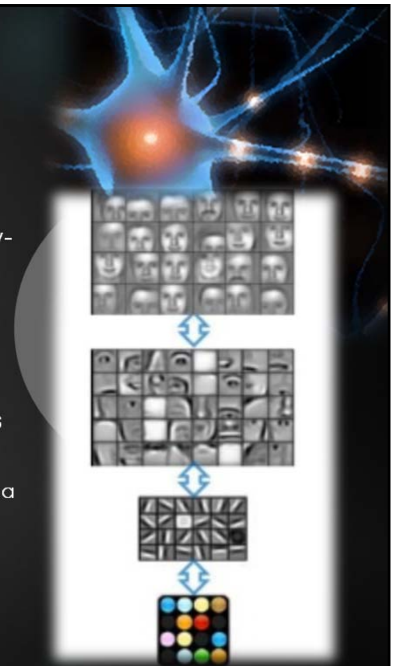
- Make for a good candidate learning algorithm
  - Evidence of layered architectures in neuro-scientific research (i.e., cortical structures)
- Applied circuit theory & efficient representations of complex functions (Hastad, 1987)
  - Can capture may factors of variation in data
- Early success of specialized yet deep architectures (i.e., Convolutional Networks, NeoCognitron)
- Local, unsupervised pre-training puts SGD-based models near good basins of attraction
  - Often escape poor local minima that plague bad random initializations
  - Works well in supervised & semi-supervised contexts



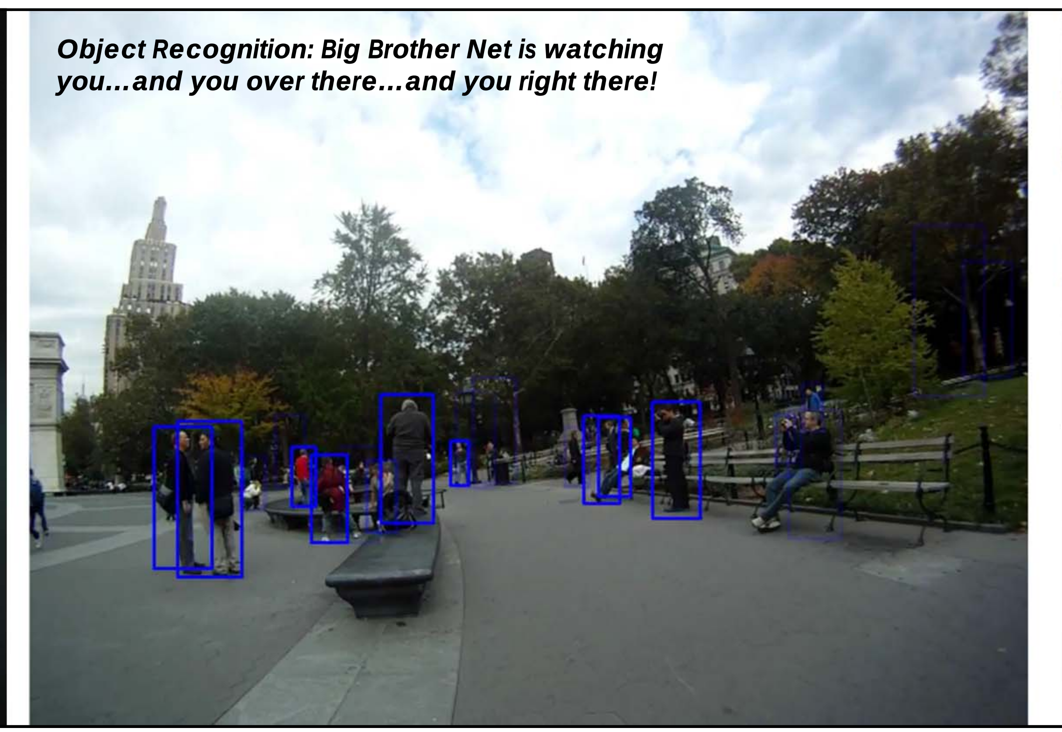
http://cs.brown.edu/~tld/projects/cortex/

Why? Feature Abstraction

- Raw features, such as pixel values of image, viewed as "low-level" representation of data
  - Can be complex & high-dimensional
  - Observed variables ("nature", observed/recorded data)
- **Abstract representations** = layers of feature detectors
  - Latent /unobserved variables that describe observed variables
    - Capture key aspects of data's underlying stochastic process
    - Many concepts can be represented as (strict) hierarchies (such as a taxonomy of species) → goal of model is to "learn" a plausible, structured unknown hierarchy
  - Idea: extracting "structure" from "unstructured"/messy data
- **Automatic feature engineering/crafting**



http://www.slideshare.net/roelofp/2014-1021-sicsdlnlpg



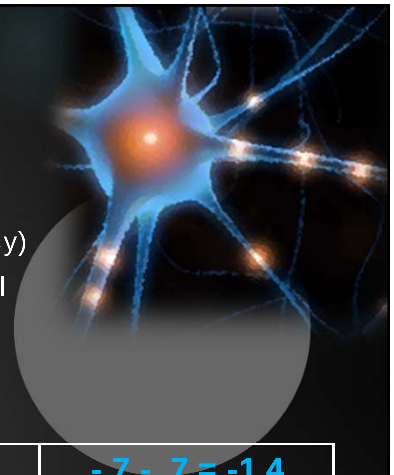
What this tutorial should give you?

- Levels of understanding:
  - *The Driver*: An informed user capable of effectively applying neural architectures to real-world data-driven problems
    - Goal: Solve real-world problems using neural architectures
  - *The Engineer*: Works at the level of implementation, develops new algorithms and architectures
    - Goal: Design new models & learning algorithms
  - *The Theorist*: Works at most abstract level, understanding performance in the limit, proving convergence, developing theoretical results
    - Goal: Develop theory to explain strengths & weaknesses of learning algorithms
- This tutorial aims to make you *The Driver*
  - Plenty of resources/references in these slides to go down "deeper" if you like (i.e., to become an *The Engineer* or *The Theorist*)

Preliminaries

THE BASICS OF WHAT YOU NEED TO "DRIVE" NEURAL ARCHITECTURES

Matrix Addition/Subtraction



- This tutorial assumes column-major matrices (for efficiency)
- Add/subtract operators follow basic properties of normal add/subtract
  - Matrix A + Matrix B is computed element-wise

0.5	-0.7
-0.69	1.8

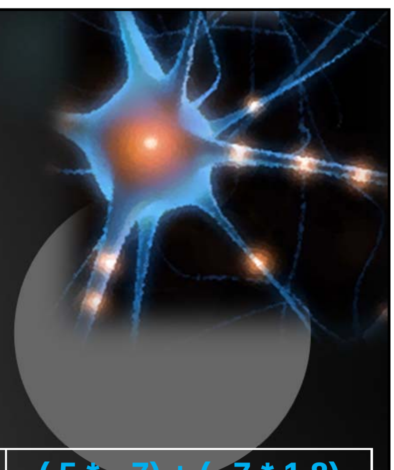
+

0.5	-0.7
-0.69	1.8

=

.5 + .5 = 1.0	-.7 - .7 = -1.4
-.69 - .69 = -1.38	1.8 + 1.8 = 3.6

Matrix-Matrix Multiply (Outer Product)



- Matrix-Matrix multiply (outer product)
- A usual workhorse of learning algorithms
- Vectorizes sums of products

0.5	-0.7
-0.69	1.8

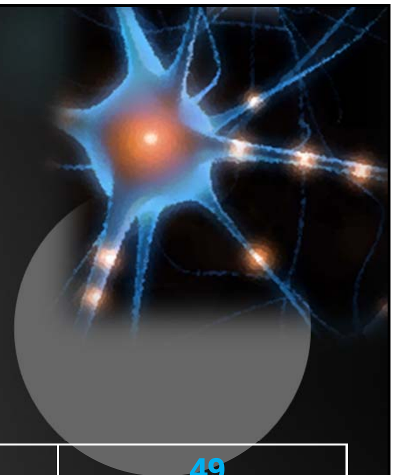
*

0.5	-0.7
-0.69	1.8

=

(.5 * .5) + (-.7 * -.69)	(.5 * -.7) + (-.7 * 1.8)
(-.69 * .5) + (1.8 * -.69)	(-.69 * -.7) + (1.8 * 1.8)

Hadamard Product



- Multiply each A(I, j) to each corresponding B(I, j)
  - Element-wise multiplication

0.5	-0.7
-0.69	1.8

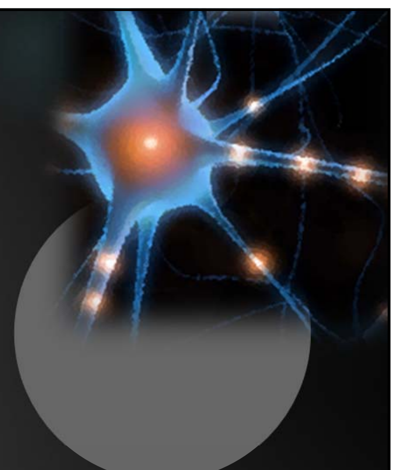
*@

0.5	-0.7
-0.69	1.8

=

.5 * .5 = .25	.49
-.69 * -.69 = .4761	1.8 * 1.8 = 3.24

Elementwise Functions



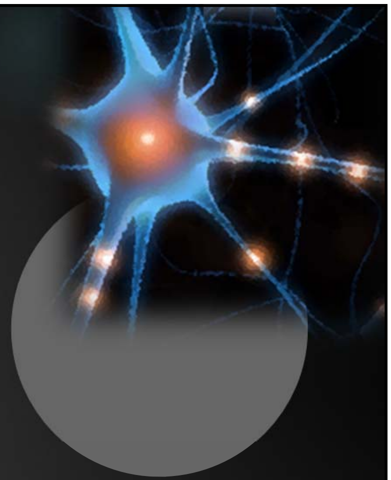
- Applied to each element (i, j) of matrix argument
- Identity: $\varphi(v) = v$
- Logistic Sigmoid: $\varphi(v) = \sigma(v) = \frac{1}{1+e^{-v}}$
- Linear Rectifier: $\varphi(v) = \max(0, v)$
- Softmax: $\varphi(v_c) = \frac{e^{-v_c}}{\sum_{c=1}^{c=C} e^{-v_c}}$

φ(

0.5	-0.7
-0.69	1.8

) =

(1.0) = 1	(-1.4) = 0
(-1.38) = 0	(3.6) = 1.8



Neural Architectures

DESIGN CHOICES & WHAT THESE GET YOU

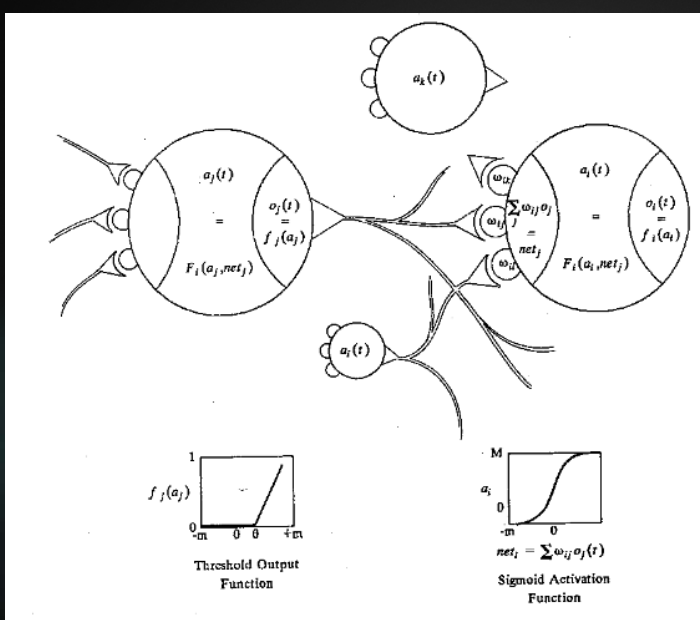
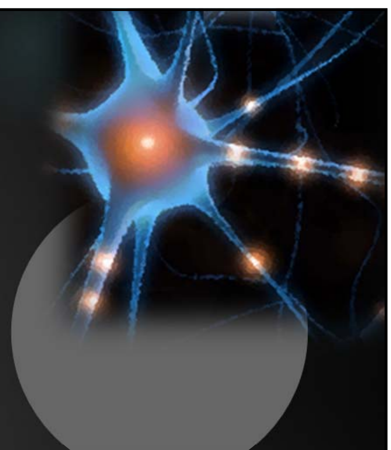
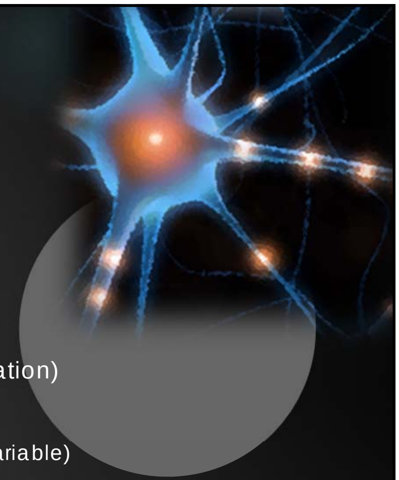


FIGURE 1. The basic components of a parallel distributed processing system.

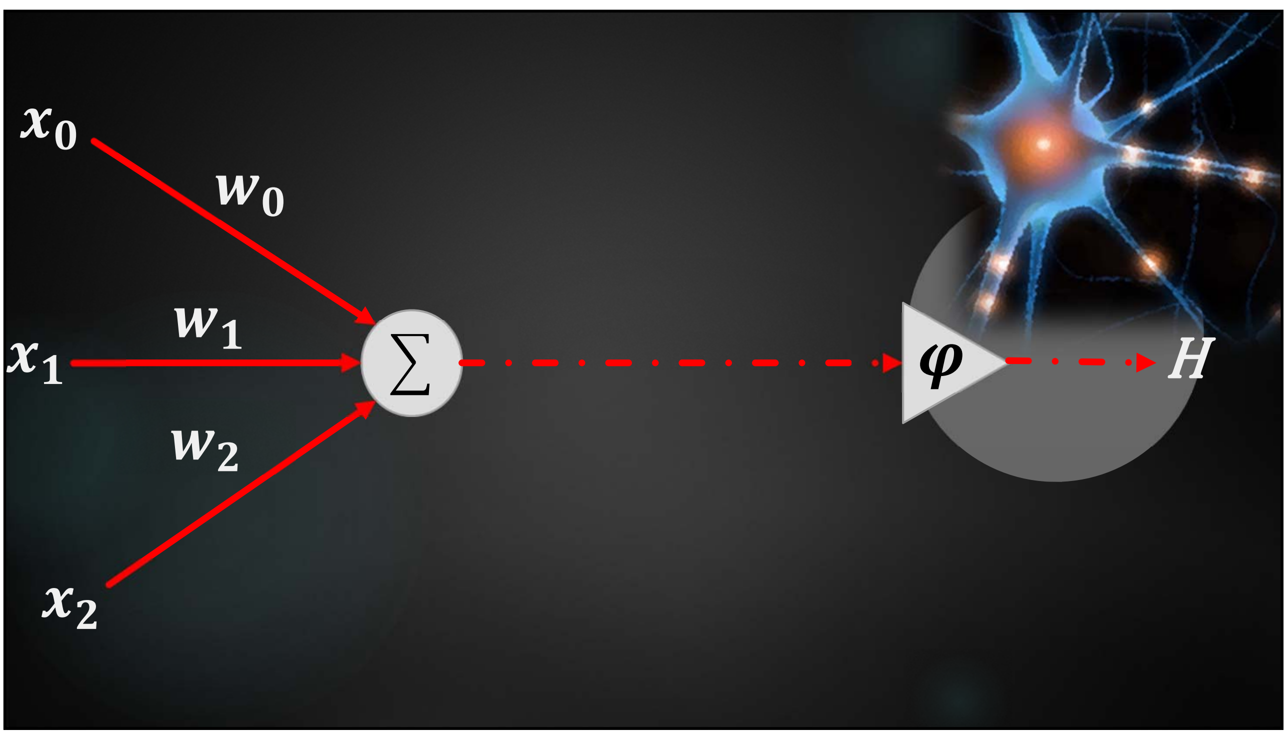


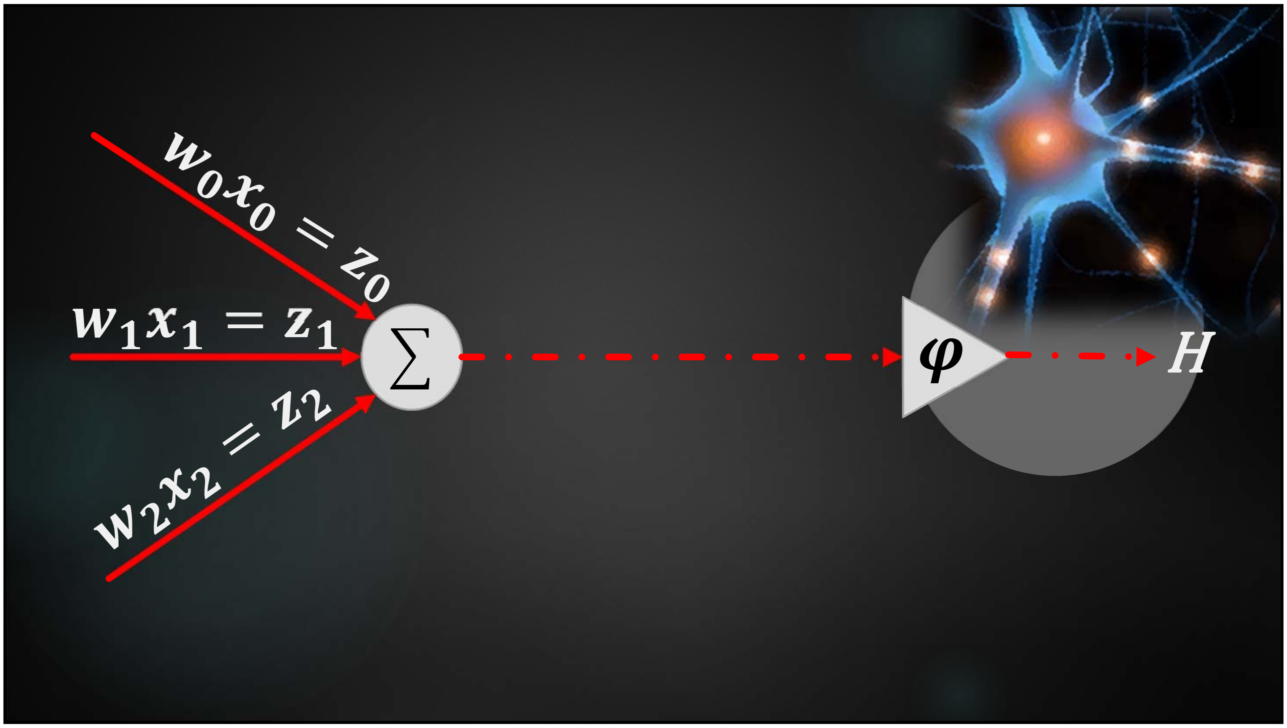
An organic take on a neural system (Rummelhart, Hinton, & McClelland, 1986).

The Processing Element (PE)

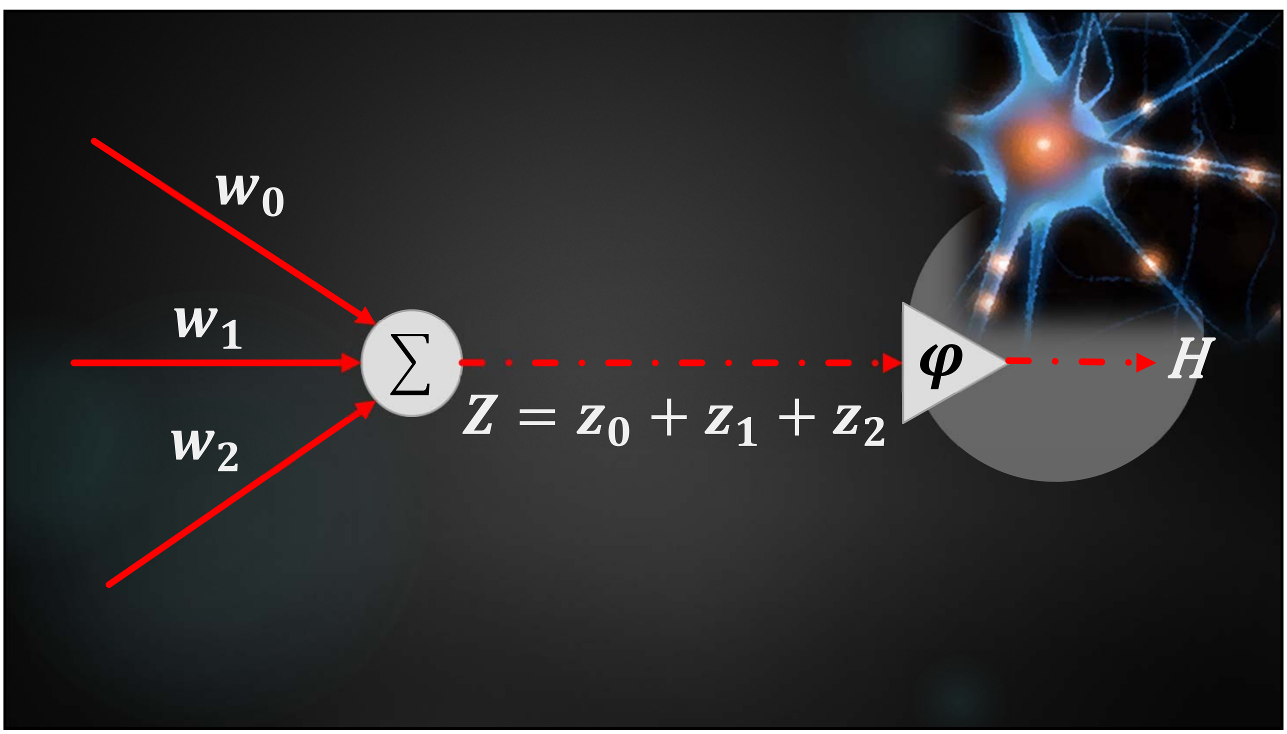


- Basic unit is integrate-then-fire "neuron"
  - Input: Takes in outputs of all its parents in a directed graph
  - Integrates all inputs via summation (pre-activation)
  - Output: Non-linearity $\varphi(v)$ applied to pre-activation (activation)
  - PE Types
    - Sensor: merely takes in input and passes it along (observed variable)
    - Processor: transforms inputs to an output signal (latent variables)
    - Actuator: merely displays "action" or decision (output variables), but could be an action such as move a robot arm left 10 degrees...

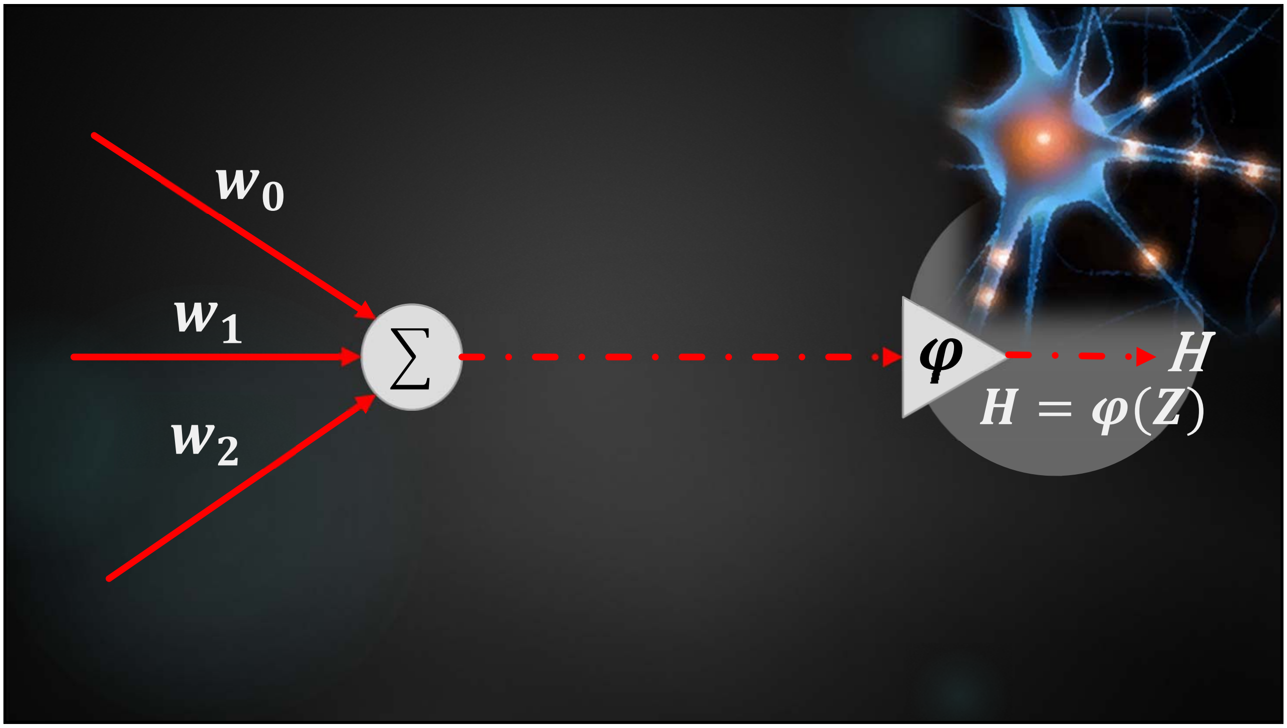




$w_0x_0 = z_0$
$w_1x_1 = z_1$
$w_2x_2 = z_2$
$\sum$
φ
H

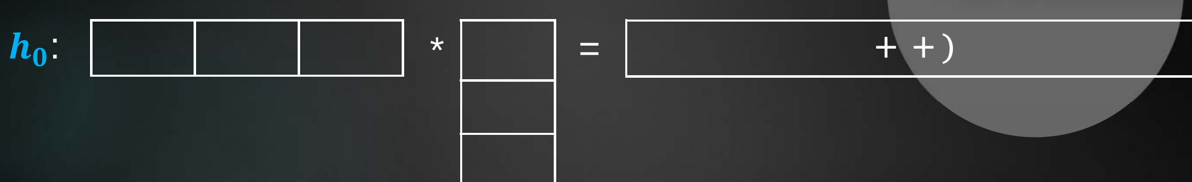


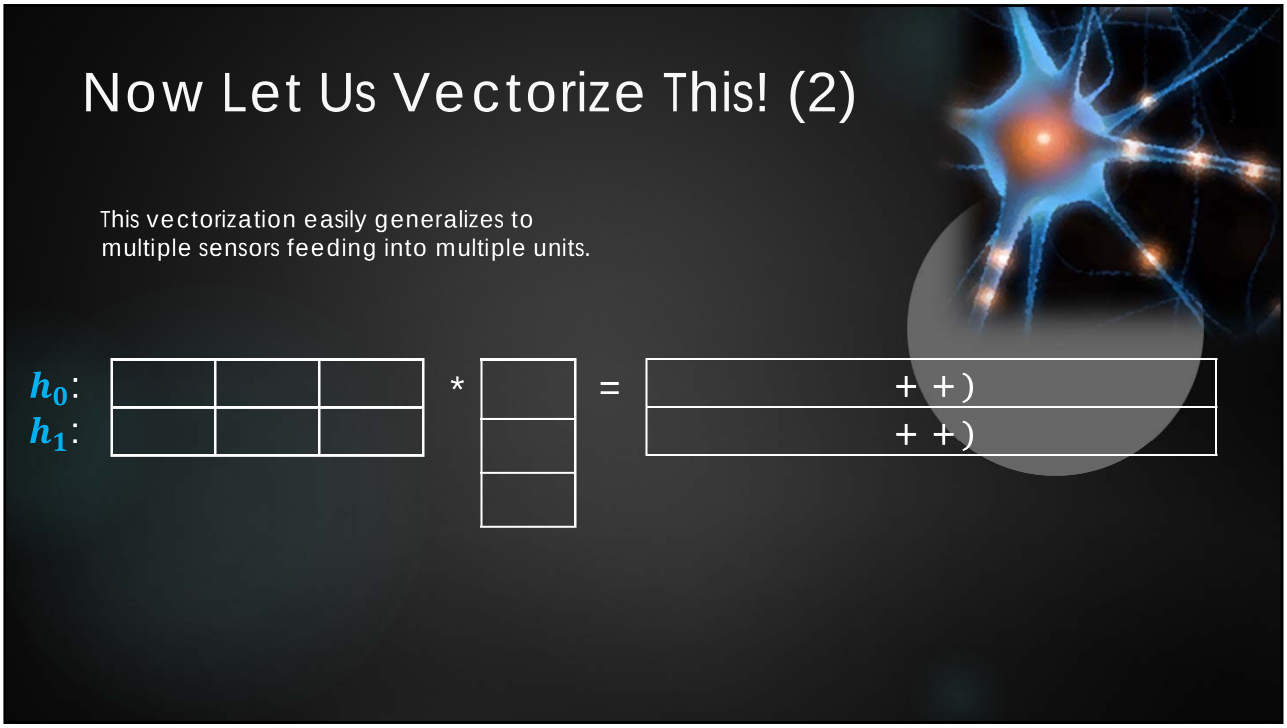
w_0
w_1
w_2
$\sum$
$Z = z_0 + z_1 + z_2$
φ
H

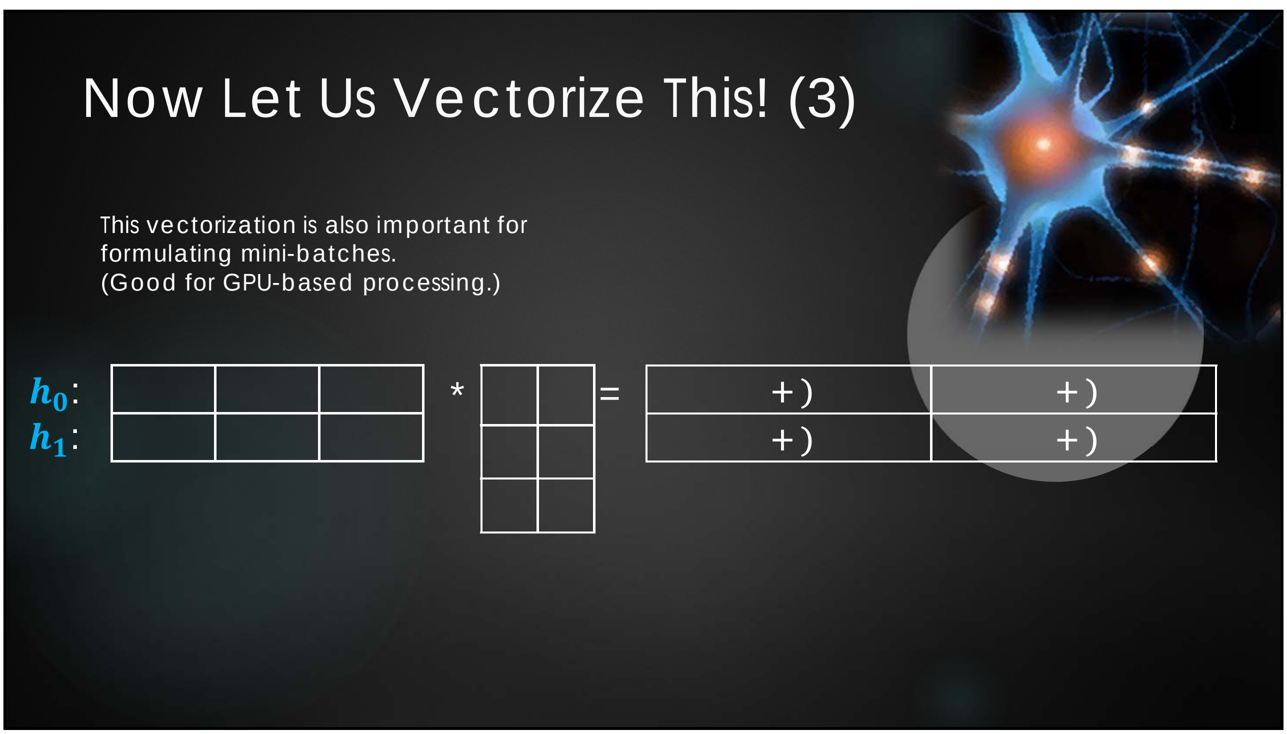


Now Let Us Vectorize This! (1)

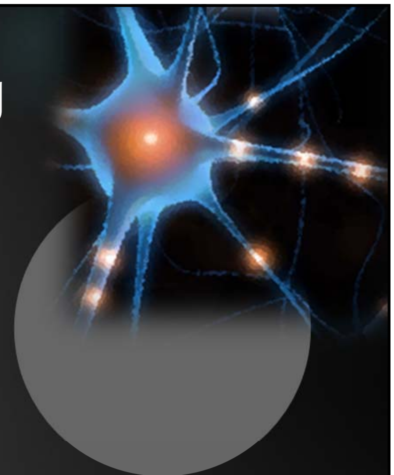
This calculates activation value of single hidden unit that is connected to 3 sensors.




Now Let Us Vectorize This! (2)
This vectorization easily generalizes to
multiple sensors feeding into multiple units.
h_0:
h_1:
*
=
+ +)
+ +)


Now Let Us Vectorize This! (3)
This vectorization is also important for
formulating mini-batches.
(Good for GPU-based processing.)
h_0:
h_1:
*
=
+)
+)
+)
+)

Combining PEs Into Processing Layers



- A complex, self-organizing system is built by combining multiple PEs
  - For simplicity, organized in blocks or layers
  - No intra-layer connections, i.e., we do not model pairwise correlations
- Each layer *i* of PEs processes activations of layer *i-1*
- Repeat process of last few slides, but each *h* becomes "data" input to layer(s) above
  - Repeat until output layer (actuators) is reached

Composing Layers: Feedforward Architectures

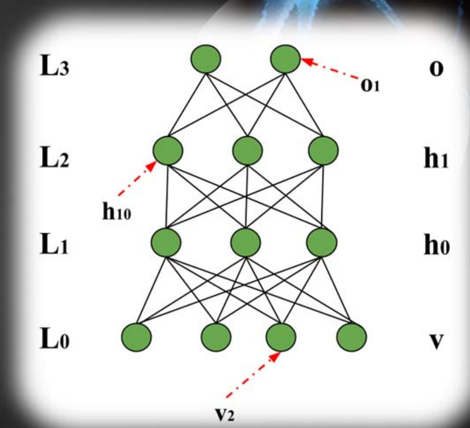
- Stack several layers to craft a simple, chain-like architecture
  - (At least) one input layer
  - (At least) one output layer
  - 0 or more processing (hidden) layers
- Feedforward refers directed nature of graph
  - No self-loops, edges not bi-directional
  - Inference is simply a sequence of matrix multiplies (& application of non-linear operators)
- Information propagated forward (bottom-up)

Architecture: Linear Chain (1)

- Simple chain composed of 1 input layer, 0 or more processing layers, & 1 output layer
  - Multi-layer perceptron
  - Maps input 1 input-vector space to output target vector-space (i.e., labels)
- Very common
  - Linear/logistic regression (0 hidden layers)
    - 1 output unit (identity activation or sigmoidal activation)
  - Support Vector Machine (0 hidden layers)
    - Linear kernel when using multi-class hinge loss (and L2 penalty)
  - Multi-layer perceptron (1 or more hidden processing layers)

Architecture: Linear Chain (2)

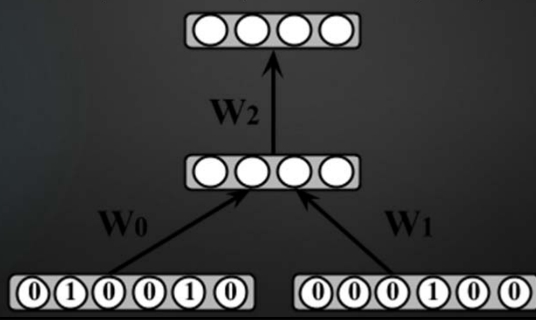
- For non-linearly separable data
  - Add non-linearity to activations
    - Linear threshold
    - Ex: Logistic Sigmoid: $\mathrm{h} = \sigma(v) = \frac{1}{1+e^{-Wv+b}}$, b *is the bias.*
- Multiple layers could lead to more "expressive" architectures
- Universal Approximators (Hornik, Stinchcombe, & White, 1989)



Multilayer ANN Architecture.

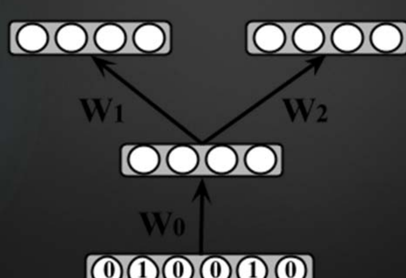
Architecture: Multi-Input

- Maps n-tuples of feature vectors to single target vectors
- If softmax output units are used, approximately learns a conditional model:
  - $p(input \mid input_0, \ldots, input_{N-1})$, where $input_n$ is a vector of features of N total input vectors (n indexes this space)



Architecture: Multi-Output

- Also known as multi-task learning
  - At some point, branches such that there are multiple, disjoint output layers → multiple tasks share same representation
  - Multi-objective loss function
  - ***Bonus***: Can use auxiliary tasks to regularize task you care about!



Complex Branching Architectures

- Architectures can be very complex
  - Combine multi-input, multi-output, and linear chains to create very deep models
  - Can allow for learning signals to be injected @ various levels
  - Examples:
    - GoogLeNet
    - Deep Residual Networks (skip every 2 layers)
    - Deep Highway Networks (skip variable layers through gating)

Automatic Differentiation

HOW PARAMETER GRADIENTS ARE CALCULATED

Objective Functions

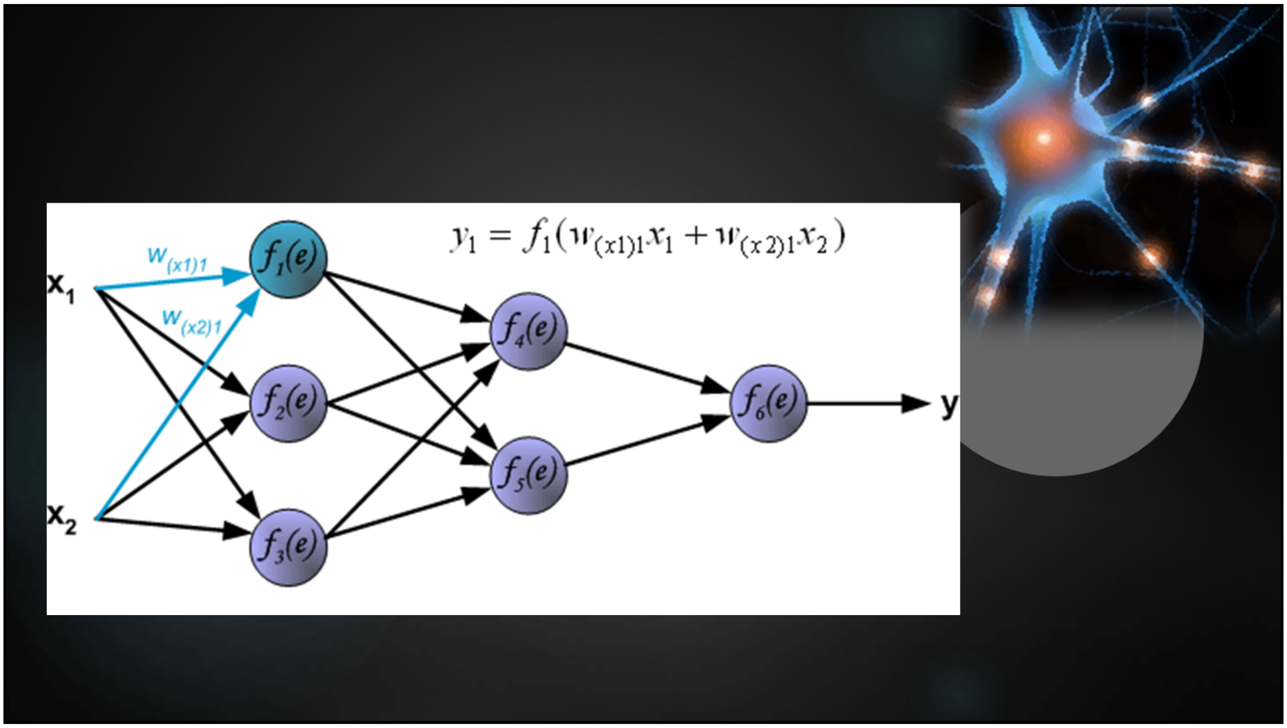
- Mean-squared error
  - is one-hot encoding of y, is output of neural architecture
- Cross-entropy
  - Often much better than mean-squared error in practice
- Categorical cross-entropy
    - Can be derived from standard cross-entropy in case of one-hot vectors
  - Equivalent to minimizing negative log likelihood
- And many others (hinge loss, etc.)

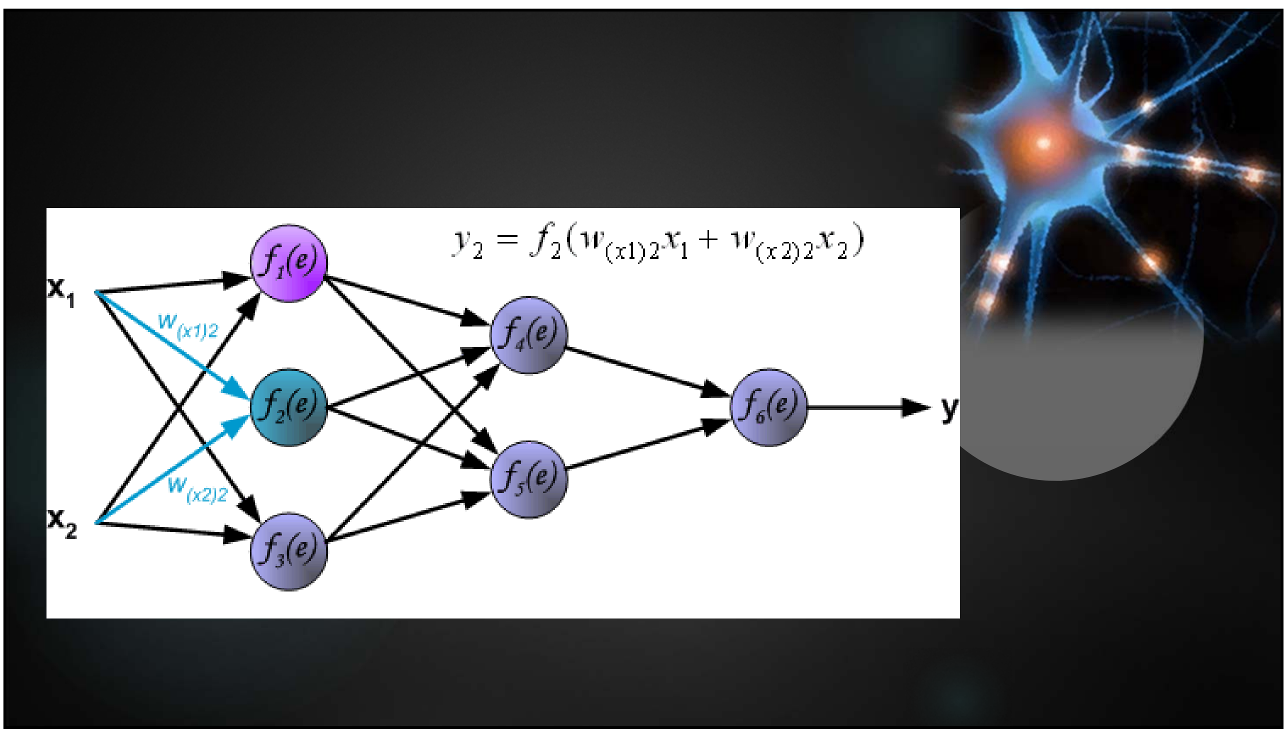
Run It in Reverse: Back-Propagation of Errors

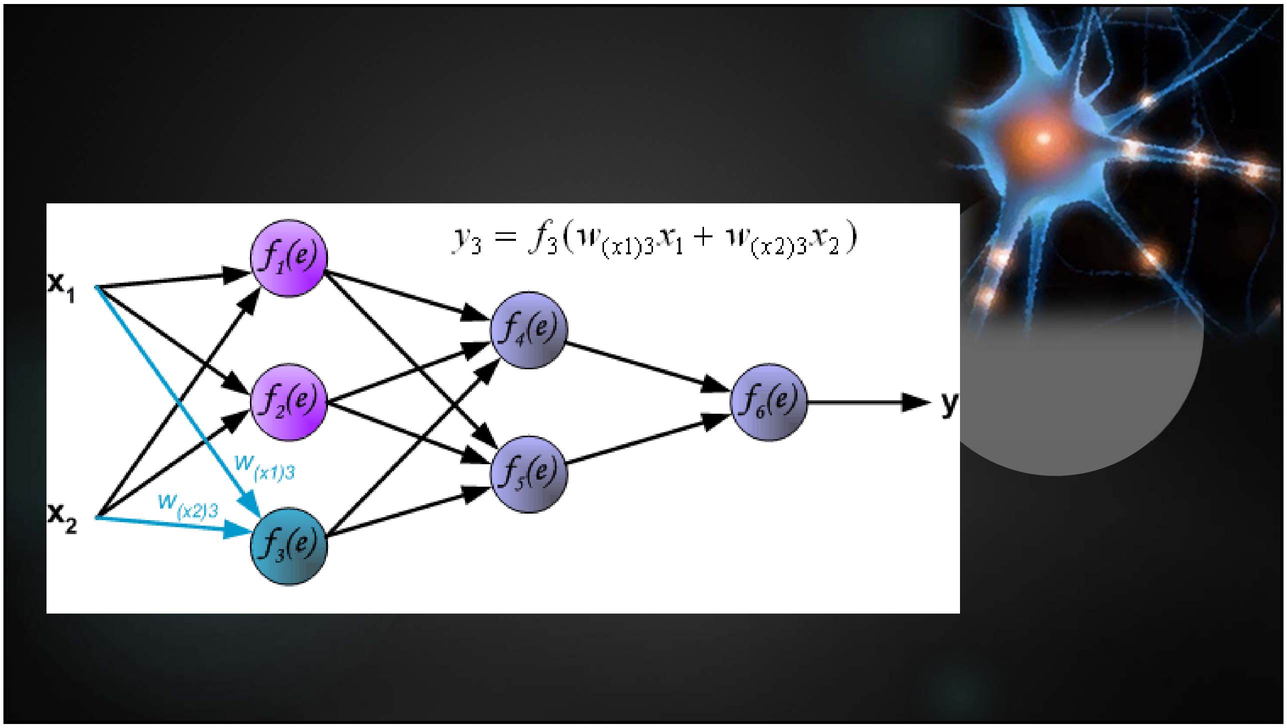
- Take error at output & prop backwards through network
  - Derivatives of objective w/ respect to variables
  - Similar process for temporal models (i.e., recurrent neural nets)
  - Good for discriminative training (layers of representation) (Rumelhart, Hinton, & Williams, 1986)
- **Problem**: the gradients, they vanish?! (Hochreiter, 1998)
  - In practice: 1-2 hidden layers was good enough!
  - **Solution**: Use better activations (i.e., linear rectifier)

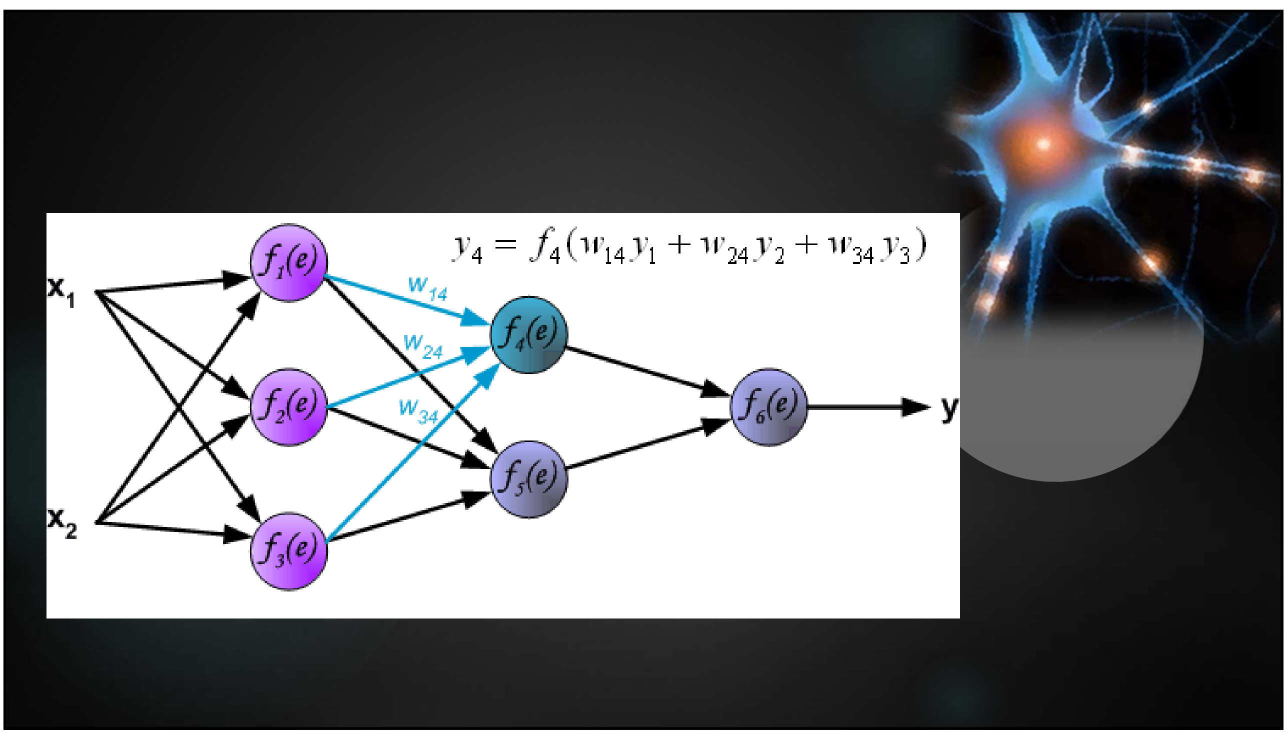
Reverse Mode Differentiation

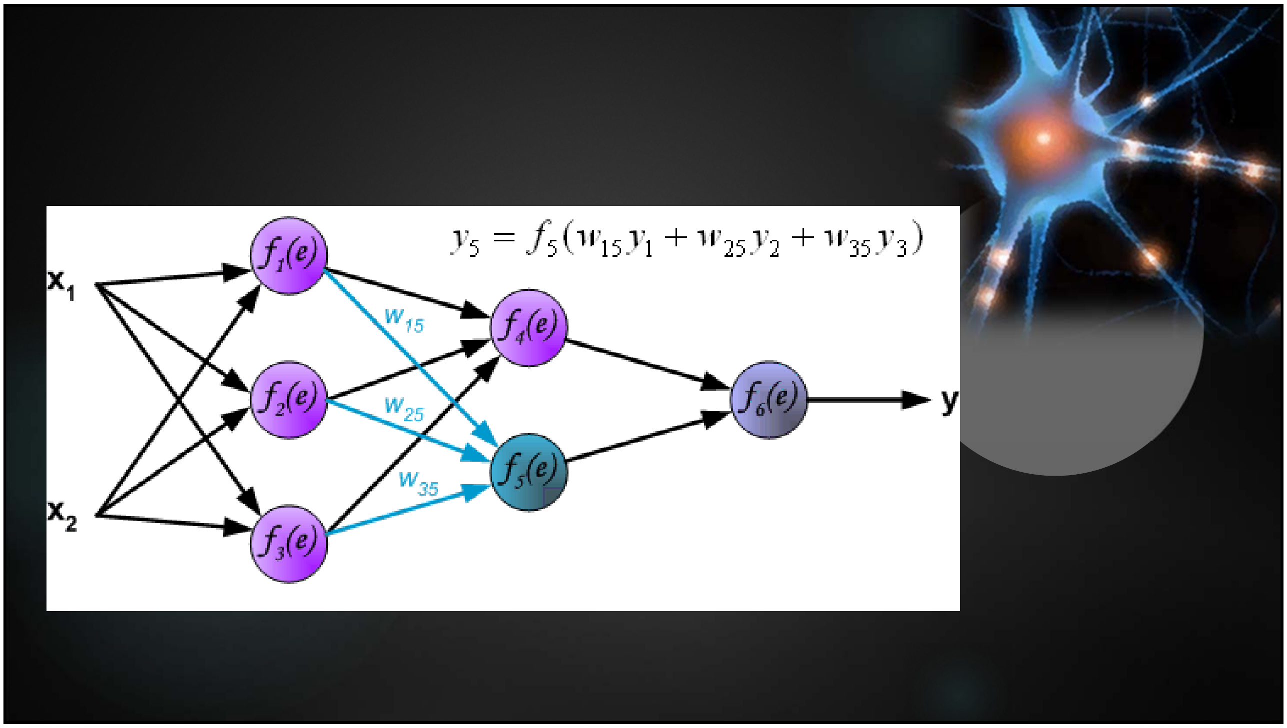
- Application of the chain-rule from calculus
- Can view this at lowest level—computation graph
  - Follow graph of operators (plus, multiply, parameter, variable, etc.) & get partial derivatives using sub-rules (sum rule, product rule, etc.)
  - Complex but highly flexible
- Can view this at level of PEs—neuronal graph
  - Follow graph of PEs
  - Limited flexibility, but simple to understand when starting off
- This tutorial shall follow latter approach for pedagogical purposes
  - Need another tutorial to fully develop arbitrary computation graphs

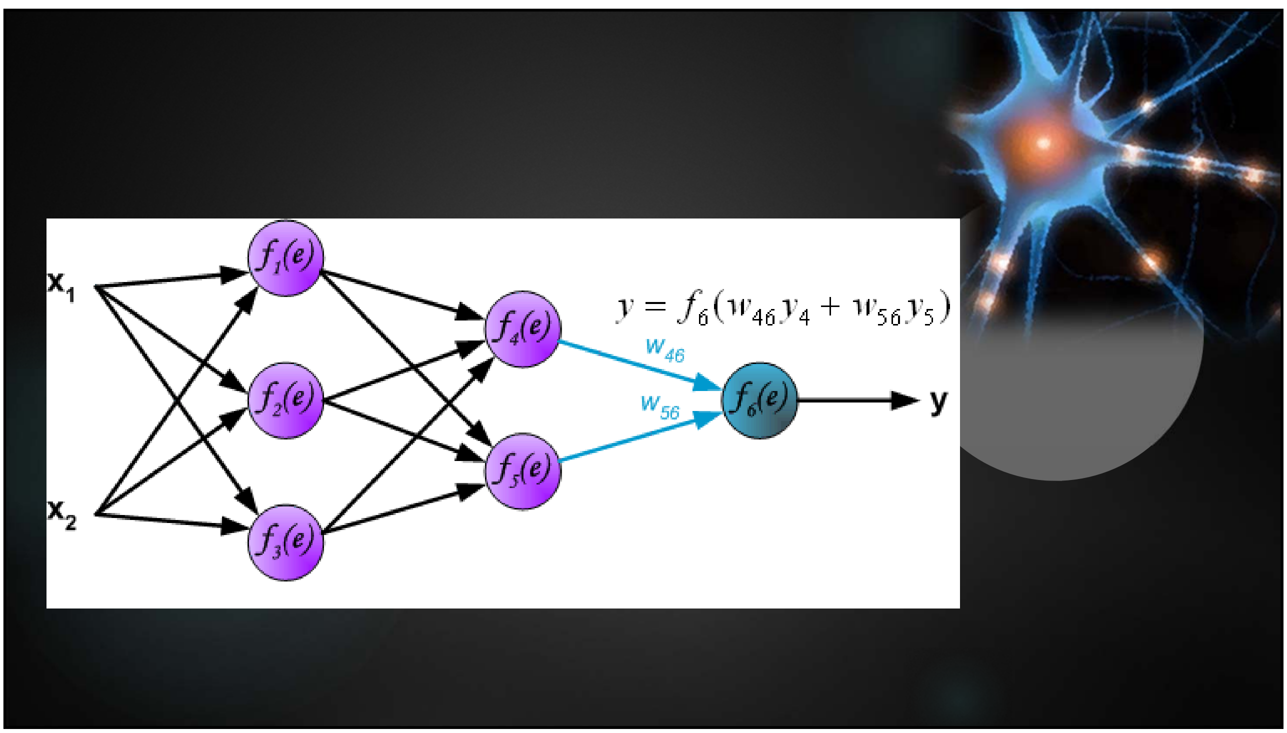

$y_1 = f_1(w_{(x1)1}x_1 + w_{(x2)1}x_2)$
x_1
x_2
$w_{(x1)1}$
$w_{(x2)1}$
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
y

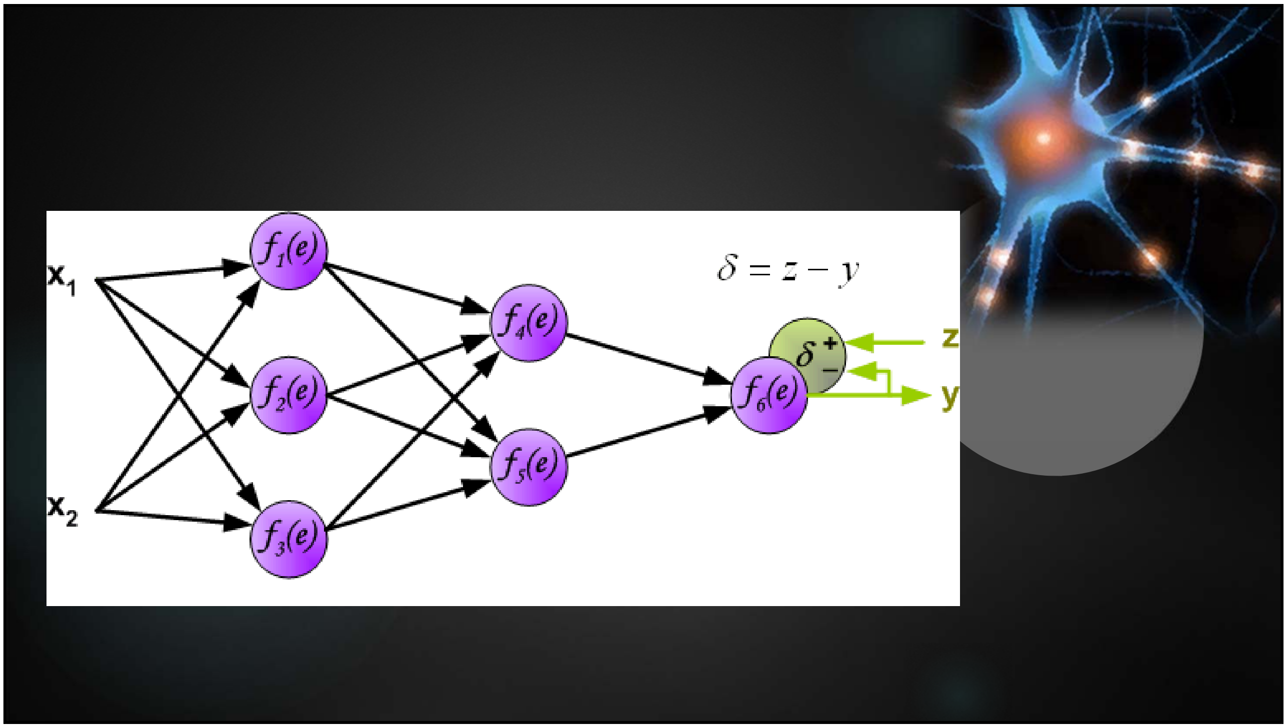

$y_2 = f_2(w_{(x1)2}x_1 + w_{(x2)2}x_2)$
x_1
x_2
$w_{(x1)2}$
$w_{(x2)2}$
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
y

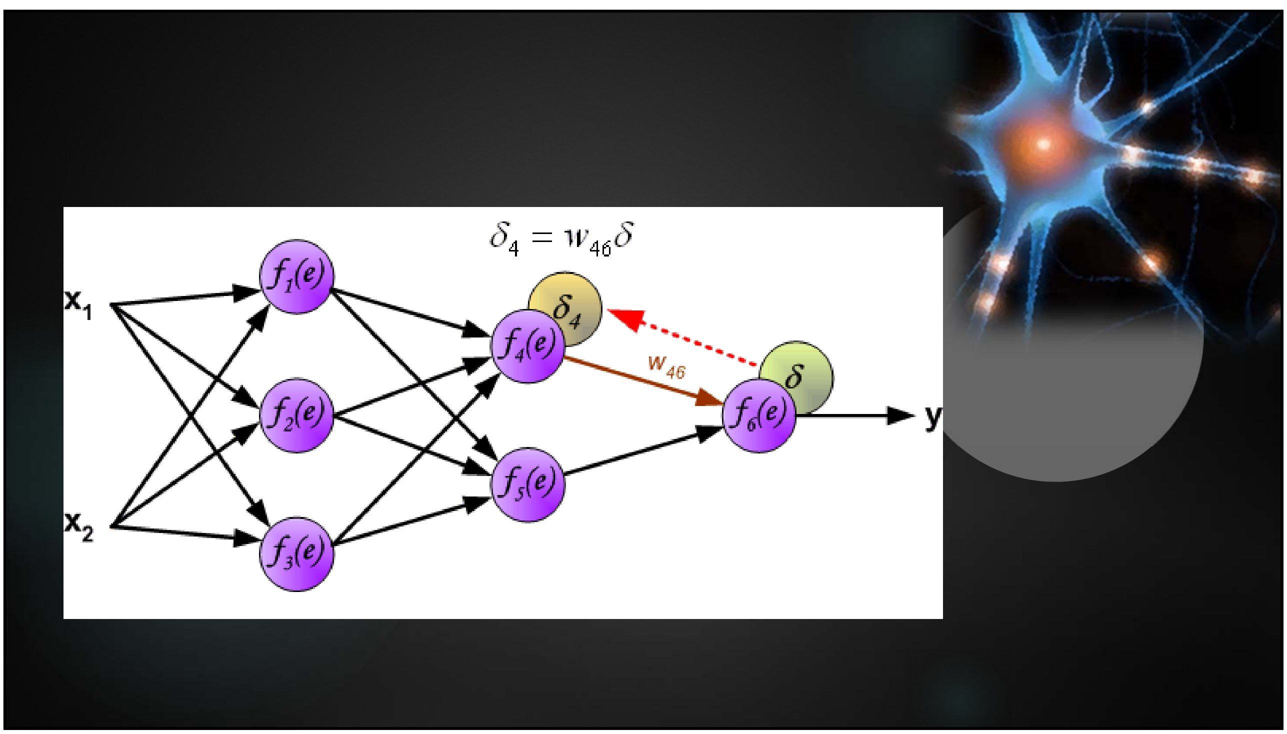

$y_3 = f_3(w_{(x1)3}x_1 + w_{(x2)3}x_2)$
x1
x2
w(x1)3
w(x2)3
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
y

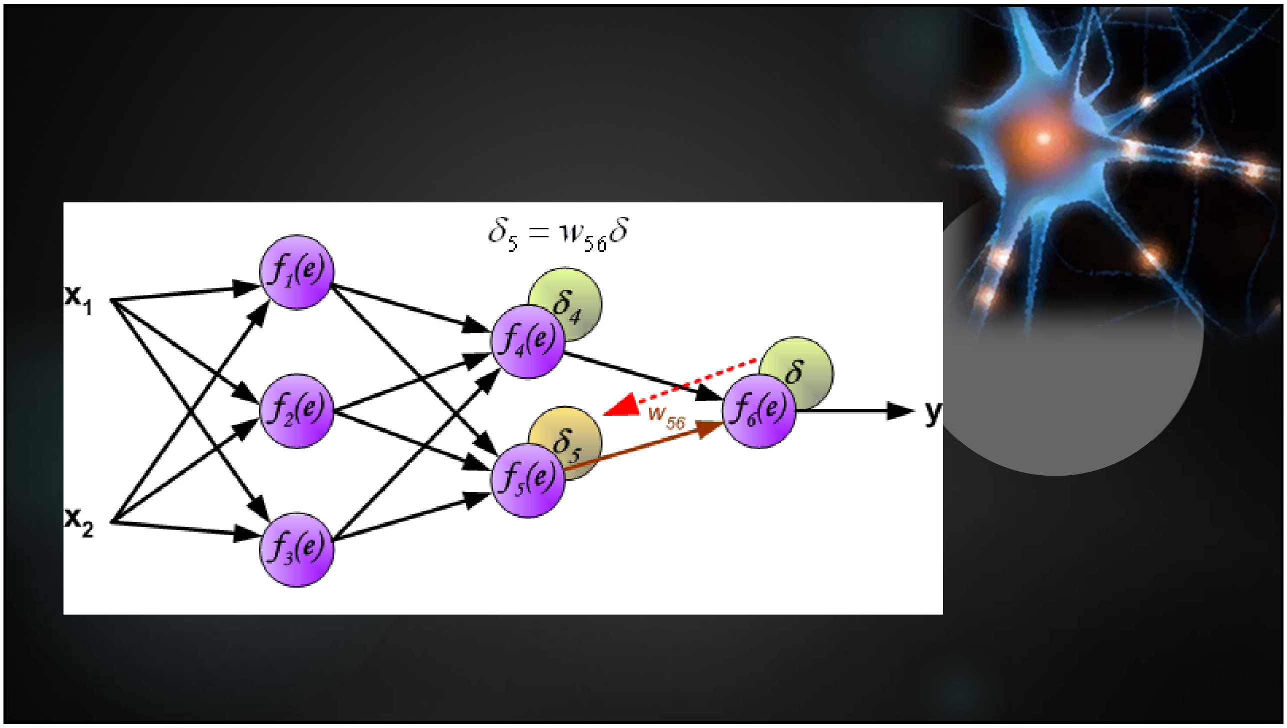

$y_4 = f_4(w_{14}y_1 + w_{24}y_2 + w_{34}y_3)$
x1
x2
w14
w24
w34
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
y

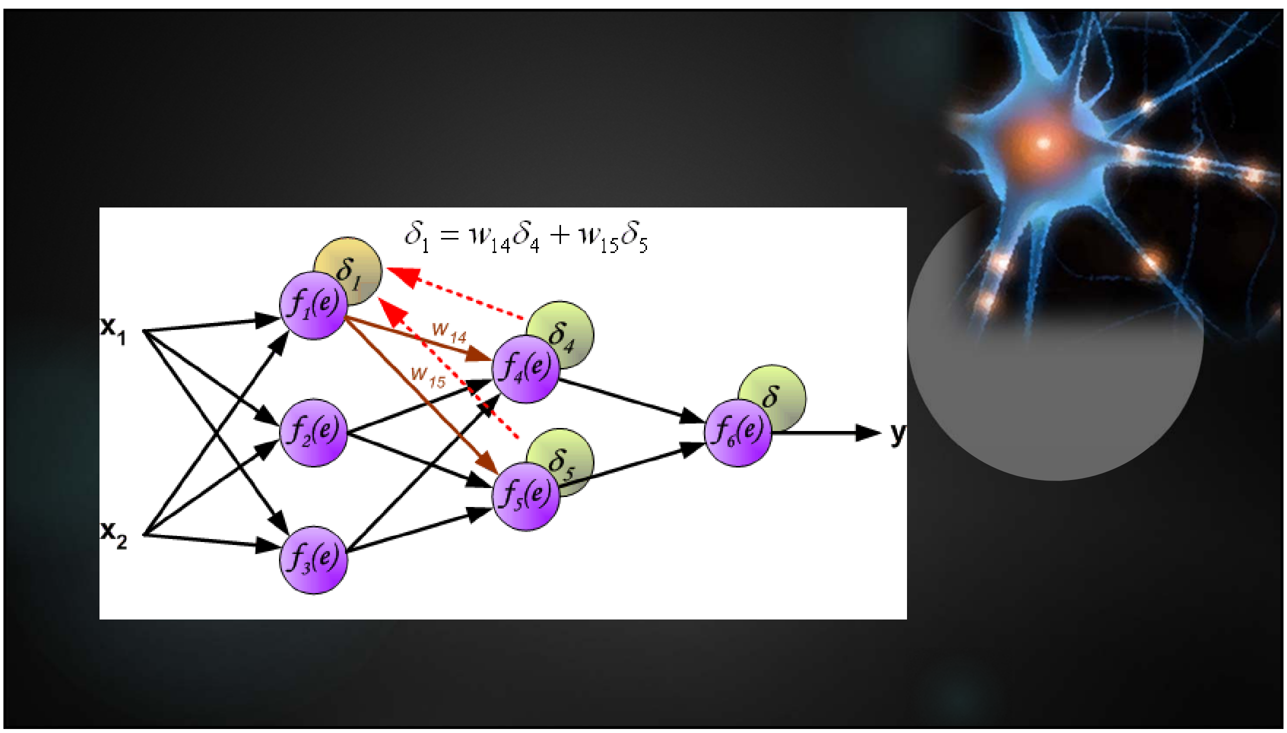

$y_5 = f_5(w_{15}y_1 + w_{25}y_2 + w_{35}y_3)$
x_1
x_2
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
w_{15}
w_{25}
w_{35}
y

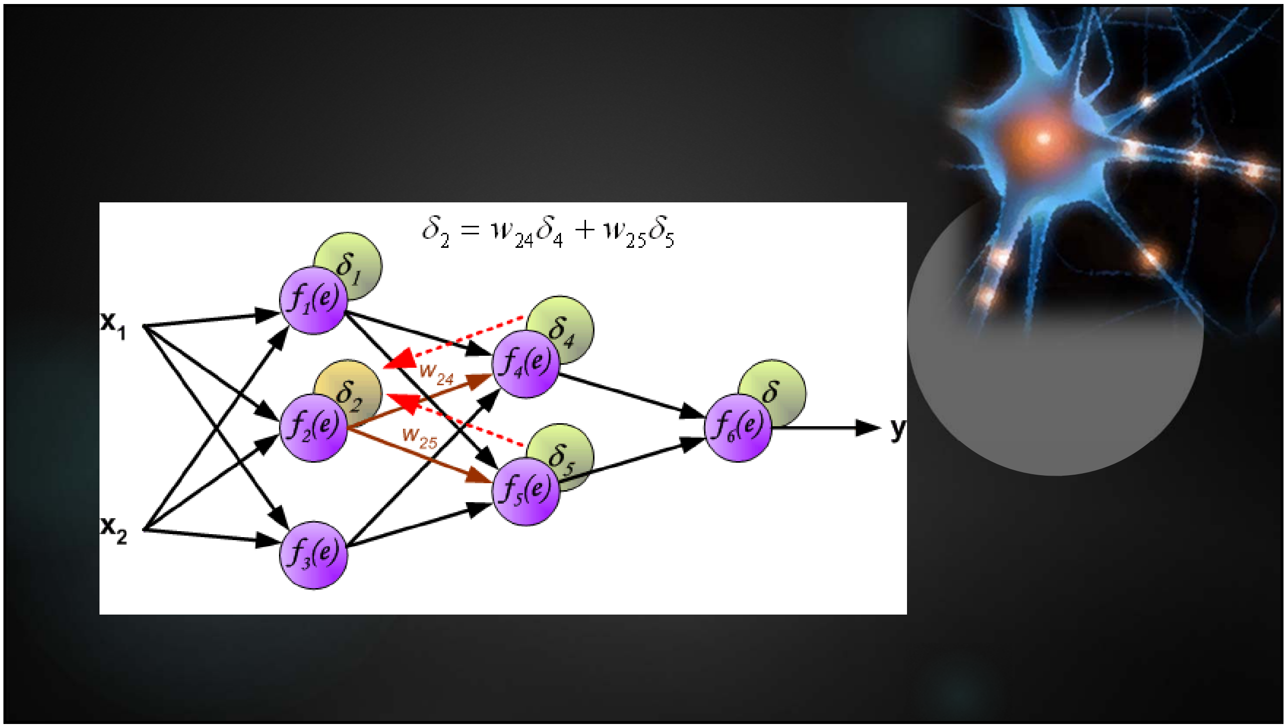

$y = f_6(w_{46}y_4 + w_{56}y_5)$
x_1
x_2
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
w_{46}
w_{56}
y

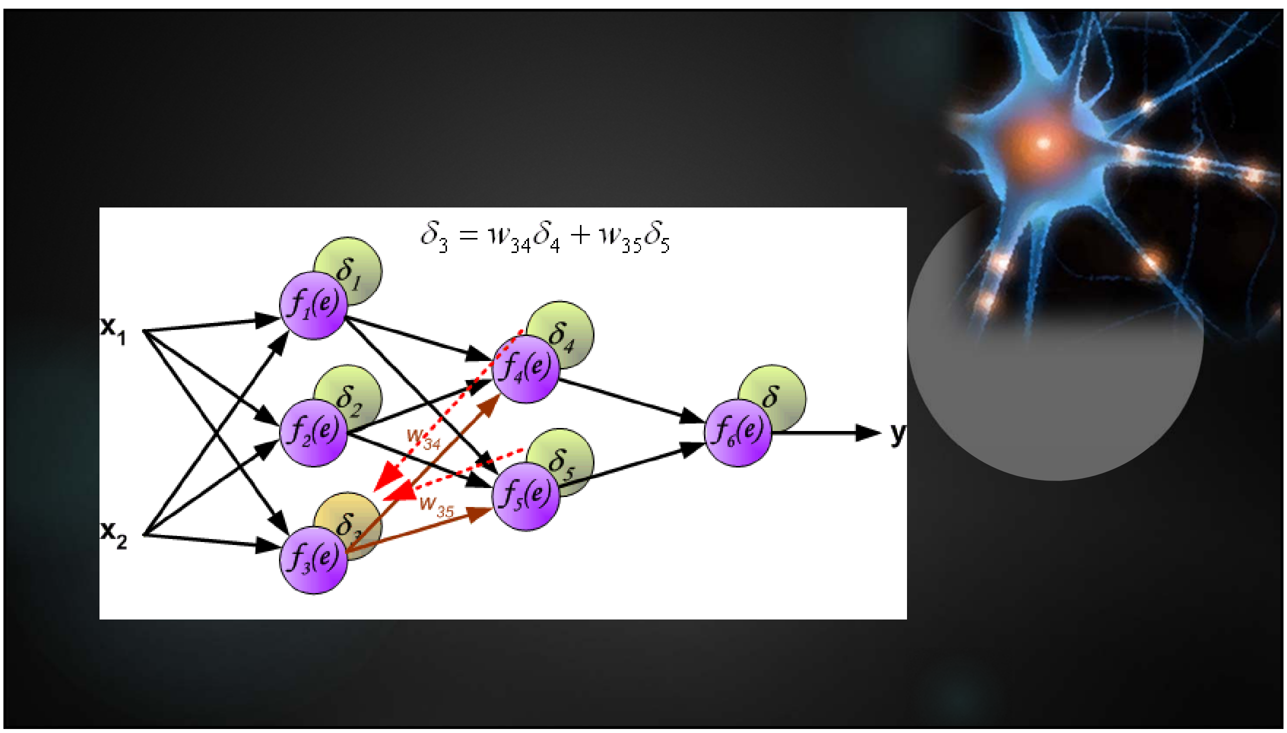

$\delta = z - y$
x_1
x_2
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
δ
z
y

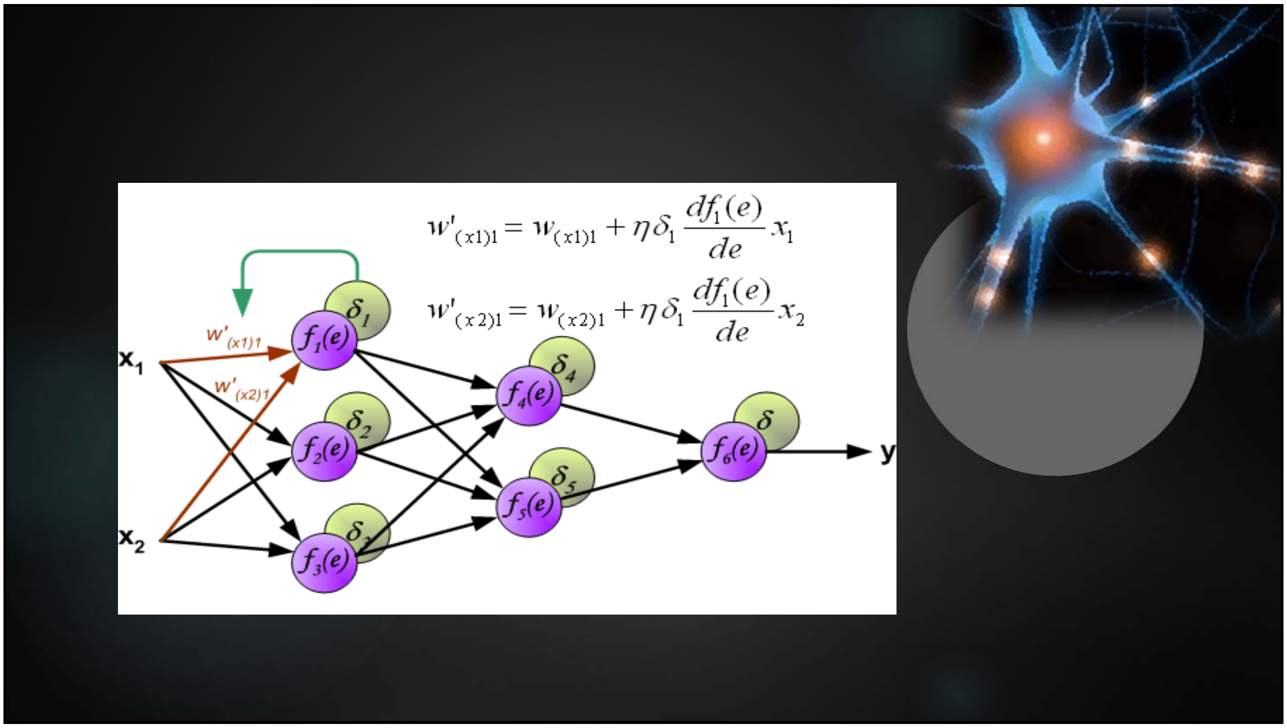

$\delta_4 = w_{46}\delta$
x_1
x_2
$f_1(e)$
$f_2(e)$
$f_3(e)$
δ_4
$f_4(e)$
$f_5(e)$
w_{46}
δ
$f_6(e)$
y

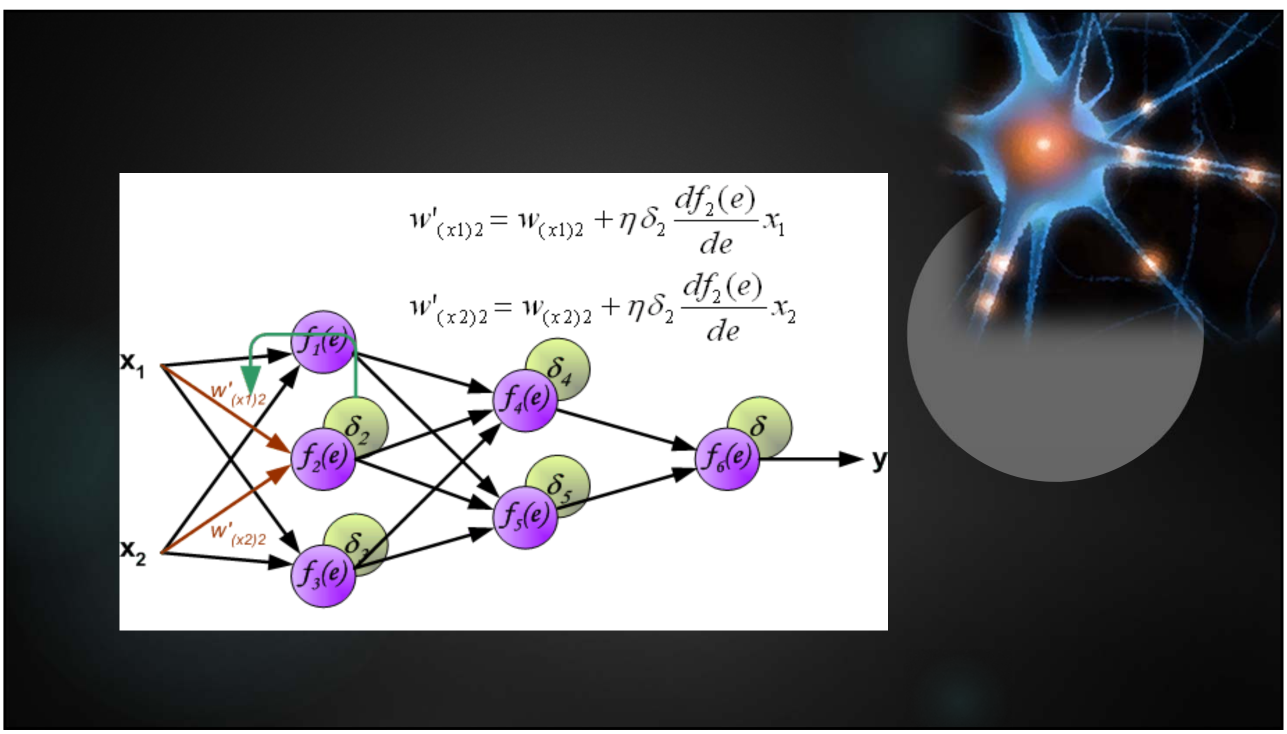

$\delta_5 = w_{56}\delta$
x1
x2
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
δ4
δ5
δ
w56
y


$\delta_1 = w_{14}\delta_4 + w_{15}\delta_5$
x1
x2
f1(e)
f2(e)
f3(e)
f4(e)
f5(e)
f6(e)
δ1
δ4
δ5
δ
w14
w15
y


$\delta_2 = w_{24}\delta_4 + w_{25}\delta_5$
x_1
x_2
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
δ_1
δ_2
δ_4
δ_5
δ
w_{24}
w_{25}
y

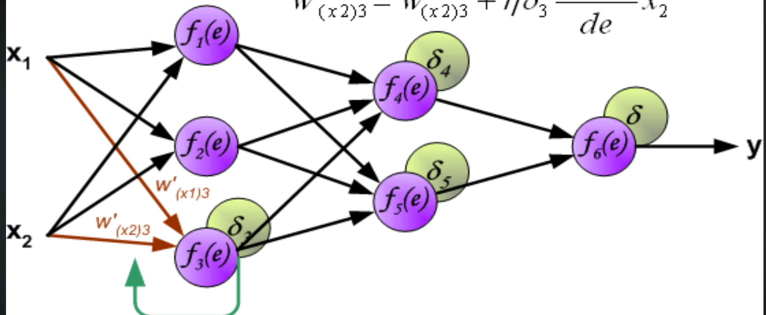

$\delta_3 = w_{34}\delta_4 + w_{35}\delta_5$
x_1
x_2
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
δ_1
δ_2
δ_3
δ_4
δ_5
δ
w_{34}
w_{35}
y


$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{de} x_1$
$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{de} x_2$
x_1
x_2
$w'_{(x1)1}$
$w'_{(x2)1}$
δ_1
$f_1(e)$
δ_2
$f_2(e)$
$f_3(e)$
δ_4
$f_4(e)$
δ_5
$f_5(e)$
δ
$f_6(e)$
y


$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 \frac{df_2(e)}{de} x_1$
$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 \frac{df_2(e)}{de} x_2$
x_1
x_2
$w'_{(x1)2}$
$w'_{(x2)2}$
$f_1(e)$
δ_2
$f_2(e)$
$f_3(e)$
δ_4
$f_4(e)$
δ_5
$f_5(e)$
δ
$f_6(e)$
y

$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \frac{df_3(e)}{de} x_1$$

$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \frac{df_3(e)}{de} x_2$$



The Vanishing Gradient Problem

- Solving credit assignment problem with back-propagation too difficult
  - Difficult to know how much importance to accord to remote inputs (Bengio et al., 1994)
  - Information passed through a chain of multiplications back through network
    - Any value slightly less than 1 in hadamard product, and derivative signal quickly shrinks to useless values
  - Learning long-term dependencies in temporal sequences becomes near impossible
- Complementary problem: Exploding gradients
  - Any value greater than 1 in hadamard, derivative signal increases dramatically (numerical overflow)

Parameter Optimization

HOW TO USE GRADIENTS TO UPDATE THE ARCHITECTURE

Optimization Schemes

- Steepest (mini-batch) gradient descent
  - Use an estimator (i.e., back-prop) to get gradient, update parameters
  - Also referred to as stochastic gradient descent (SGD)
- Parameter initialization
- Modifications:
  - Momentum
  - Regularization terms (L1, L2, DataGrad)
  - Gradient clipping & parameter renormalization
  - Drop-out
  - Alternative optimizers

Parameter Initialization

- Simple distributional schemes
  - Fan-in Uniform
    - Uniform distribution scaled by square root of (2 / # inputs to layer)
  - Gaussian
    - Gaussian distribution centered @ 0 (usually w/ variance <= 1)
  - Fan-in Gaussian
    - Gaussian distribution scaled by square root of (2 / # inputs to layer)

Steepest Gradient Descent

- Simplest update rule
- Combine with early stopping (tracking loss/error on validation set)
  - A simple form of regularization (as weights will be smaller)

```
# Vanilla update
x += - learning_rate * dx
```

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

Simple Momentum

- Maintains a rolling average of previous gradients
  - Smooths descent of loss minimization algorithm
- Many variants: Nesterov's Accelerated Gradient, etc.

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

Regularization: L2 Penalty

$$C = -\frac{1}{n} \sum_{xj} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] + \frac{\lambda}{2n} \sum_w w^2. \quad (85)$$

The first term is just the usual expression for the cross-entropy. But we've added a second term, namely the sum of the squares of all the weights in the network. This is scaled by a factor $\lambda/2n$, where $\lambda > 0$ is known as the *regularization parameter*, and n is, as usual, the size of our training set. I'll discuss later how λ is chosen. It's also worth noting that the regularization term *doesn't* include the biases.

$$\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \end{aligned}$$

http://neuralnetworksanddeeplearning.com/chap3.html

Regularization: L1 Penalty

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|. \qquad (95)$$

Intuitively, this is similar to L2 regularization, penalizing large weights, and tending to make the network prefer small weights. Of

regularization. The resulting update rule for an L1 regularized network is

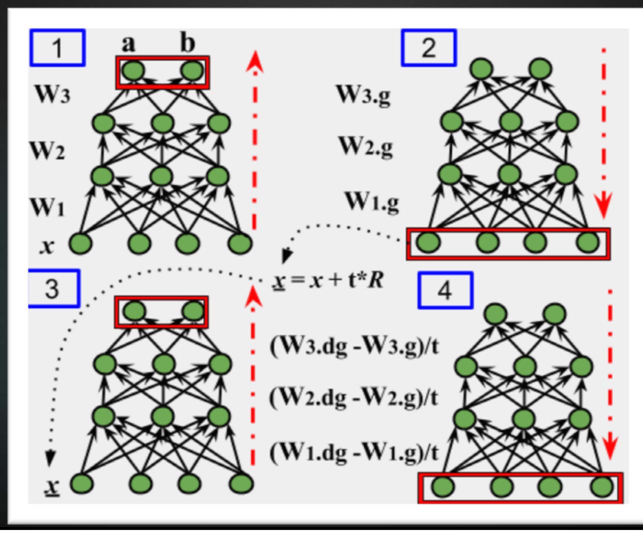
$$w \rightarrow w' = w - \frac{\eta \lambda}{n} \operatorname{sgn}(w) - \eta \frac{\partial C_0}{\partial w}, \qquad (97)$$

http://neuralnetworksanddeeplearning.com/chap3.html

Regularization: DataGrad Penalty (1)

- Blind-spot problem—can trick neural nets into making incorrect prediction by perturbing input data
  - Prominent in images
  - Coined “adversarial” samples
- Can employ methods for building robustness against adversarial samples into any data problem
  - Can improve generalization
  - Similar to data augmentation (creating artificial additional images to increase data size)
  - Potentially “sees” more of the underlying data manifold
- DataGrad: an “adversarial” prior (Ororbia et al., 2016)

Regularization: DataGrad Penalty (2)

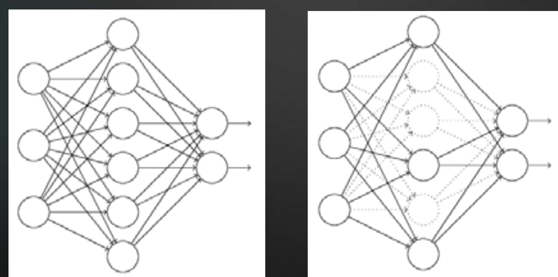


Gradient Clipping & Parameter Renormalization

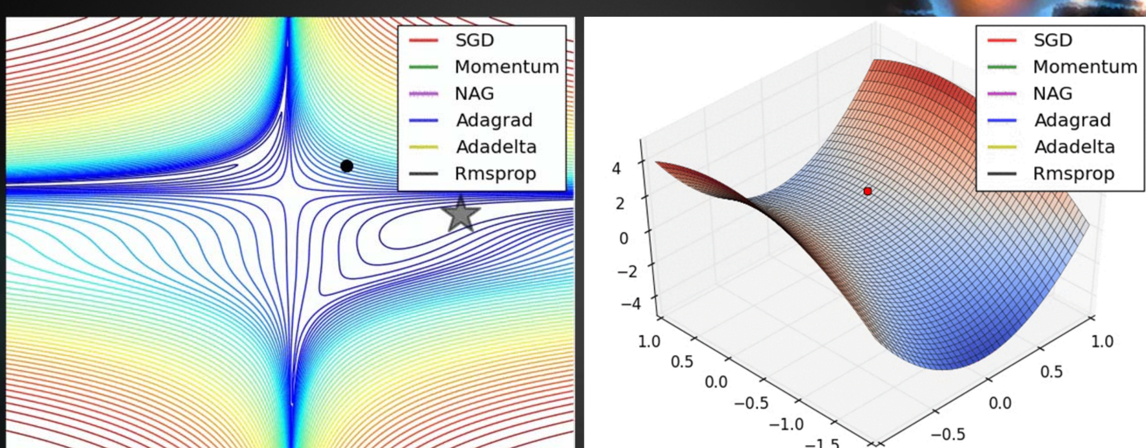
- Largely resolves exploding gradients problem
  - Simply threshold magnitudes of each dimension of gradient to some reasonable value (i.e., 1 or 5)
  - Can combine this with other tricks (such as “skipping over” bad gradients)
- Track norm of parameters and rescale by normalizing values of gradient by norm
  - Must cross-validate norm threshold

Drop-out

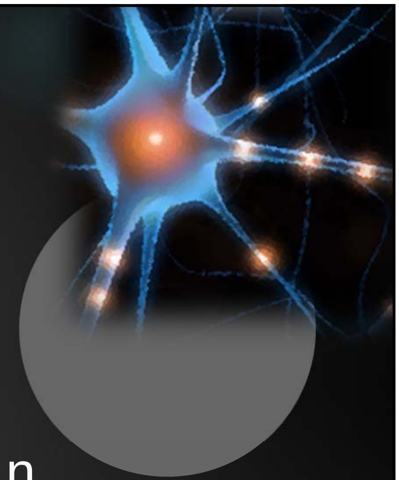
- Each iteration (within an epoch), simply omit some units with a given probability (binary masks)
  - At inference time, simply multiply activations by probability
- In single hidden layer model, equivalent to Bayesian model averaging
- A form of architectural regularization
  - Controls for overfitting (for models with many parameters)



Other Optimizers

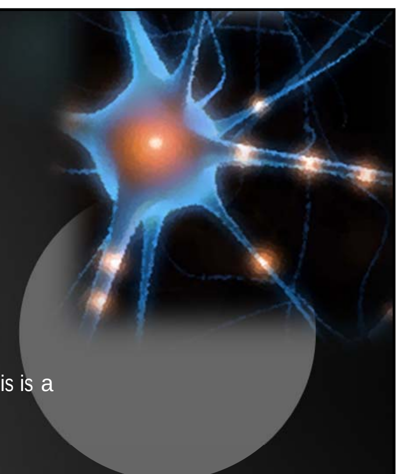


http://cs231n.github.io/neural-networks-3/#hyper



Hyper-parameter Optimization

HOW TO TUNE A LEARNING ARCHITECTURE



Manual & Grid-Search

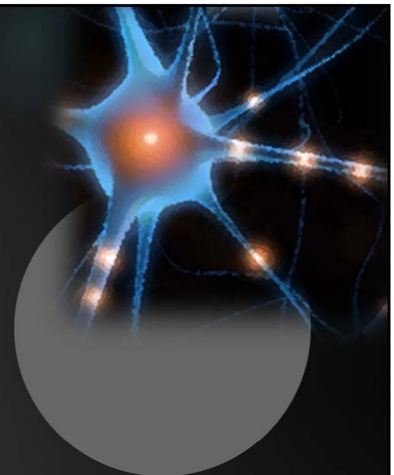
- Manual Search
  - Explore a few configurations, based on literature/heuristics
  - Select lowest validation loss configuration
- Grid Search
  - Compose an n-dimensional hypercube, where along each axis is a hyper-parameter (length determined by max & min values to explore)
  - Exhaustively calculate loss/error for each configuration (or combination of meta-parameter values) in hypercube
    - Choose lowest error/minimal loss configuration as optimal model
    - Loss/error is calculated on a held-out validation/development set (or in held-out set in cross-fold validation schemes)
  - Will ultimately find optimal model (given coarseness of grid-search), but will take a really long time

Random Search

- Draw *k* sample configurations from hypercube & calculate validation loss for each (w/o replacement)
  - Repeat *T* trials, can use optimal of each trial to inform subsequent trials
  - Can "guide" or "target" next set of random samples based on best last found point
  - A more stochastic search
- Surprisingly effective, moreso than manual search & faster than grid search

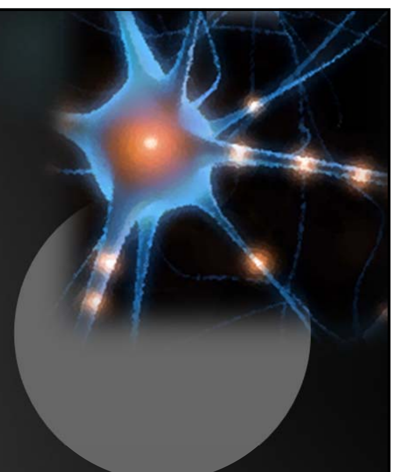
Bayesian Optimization: Meta Machine Learning

- Use machine learning to do your research for you…
  - Sequential Model Optimization (SMO)
  - Gaussian Processes for surface-response modeling
  - Gradient-based: Use another neural network
    - How do we tune this higher-level parametric model?
    - Meta-meta-meta-….-machine learning??
- High-level idea:
  - Build a meta-model (with some prior that encodes intuition about hyper-parameter space)
  - Draw samples from space (i.e., run a few configurations of your model)
  - Update your meta-model using these samples
  - Your meta-model selects next best point to evaluate
    - Balancing criterion such as minimal error and minimal compute time



Data Pre-processing

HOW TO PREPARE DATA FOR TRAINING A NEURAL ARCHITECTURE



Process of Vectorization

- Feature transformations
  - Standardization (0 mean, unit variance)
  - Re-scaling (to range of [0,1])
- Surface statistics representation, or Bag of Words (BOW)
  - Binary occurrence (multi-hot vector)
  - Term frequency
  - Term Frequency – Inverse Document Frequency (TF-IDF)
- Context window modeling (beyond scope of this tutorial)
  - Encode a target word and its surrounding context as a multi-hot vector
  - Word2Vec: Skip-Gram, Continuous Bag of Words (CBOW)
- Sequence window modeling (beyond scope of this tutorial)
  - Encode a sequence or ordered inputs as a 3D tensor, or a vector of matrices, where each matrix is a vector of one-hot encodings
  - Good for temporal models like recurrent neural architectures

BOW Modeling (Text)

- Process:
  - Apply any string transforms (lower-casing, stemming, stop-word removal)
  - Construct a dictionary V of unique symbols in corpus mapped to a unique integer [0, |V|-1]
  - For each document, construct a vector, filling in each index i with a number if symbol at i occurs in dictionary
    - Fill in slot with 1 → binary presence, frequency in document → term counts TF
    - Convert to TF-IDF or log(1 + TF) if real-valued representation desired

Context Window Modeling (Text)

- As opposed to BOW modeling, slide a window of fixed or variable size across each document, encoding each word & its surrounding context into multi-hot binary vectors
  - E.g. “The cat sat on the mat.” – target word = “sat”, left context (2) = “The cat”, right context (3) = “on the”
  - Train model to predict target word given context or vice versa
  - Can combine this with word-embeddings (from word2vec or GloVE, for example)

An Application: Automatic Content Coding

PUTTING IT TOGETHER IN AN APPLICATION

Content Coding Setup

- From a machine learning perspective, may be posed as a classification problem
  - Becomes semi-supervised at scale (i.e., when you have lots and lots of documents/texts)
- General approach:
  - 1) Come up with themes/categories, starting off as usual
  - 2) Take a representative sample & code it manually
  - 3) Fit a model to both annotated & non-annotated documents
- For labeled dataset D_{train}
  - D-dimensional pattern vectors : $\hat{v} \in (v_0, v_1, \ldots, v_D)$ with C-dimensional label vectors $\hat{y} \in (y_0, y_1, \ldots, y_C)$

The (Neural) Modeling Paradigm

- It's a matter of posing the problem
  - What is the low-level representation of your sample? (i.e., low-level feature vectors)
  - Is there an output we are interested in?
    - Regression: a real-valued target
    - Categorization: a multi-class/decision target
- How much data do you have?
  - More data is better! (MNIST is 60K)
  - Only a small sample? Go with Bayesian Neural Networks!
- What kind of hardware do you have?
  - Multi-CPU settings
  - GPUs

Semi-supervised Neural Architecture

- We design a multi-objective optimization problem:
  - $L(x, y, u) = \gamma * L(x, y) + \beta * L(u)$, setting $\gamma = 1$ & $\beta = (0,1]$
- We will demo a simple approach: Entropy-Regularization (i.e., self-training)
  - Use a deep rectifier network, with drop-out, trained using its own predictions for unlabeled samples
  - Anneal weight β applied to unlabeled loss function
- Experiment for this tutorial: performance as function of proportion of labeled samples

Some Demo Results

CNAE-9	Supervised	Semi-Supervised (= 0.15)
25 % Labeled	0.1712963	**0.13425928**
75 % Labeled	0.05092591	**0.0462963**
Fully Labeled	0.037037015	N/A

LETTERS	Supervised	Semi-Supervised (= 0.15)
25 % Labeled	0.13046736	**0.11822045**
75 % Labeled	**0.07398152**	0.091477156
Fully Labeled	0.075231194	N/A

https://github.com/ago109/SBP-BRIMS-2016-Tutorial.git

(Ororbia et al., 2015, ECML)

76

CAPTCHA character categorization performance.

	Error	Precision	Recall	F1-Score
MaxEnt	0.475 ± 0.010	0.535 ± 0.011	0.524 ± 0.010	0.522 ± 0.010
SVM	0.461 ± 0.011	0.564 ± 0.010	0.537 ± 0.011	0.526 ± 0.011
2-Rect [13, 23]	0.365 ± 0.011	0.651 ± 0.011	0.634 ± 0.011	0.627 ± 0.013
HRBM [20]	0.368 ± 0.009	0.643 ± 0.010	0.631 ± 0.009	0.629 ± 0.009
5-SBEN	$\mathbf{0.324 \pm 0.008}$	$\mathbf{0.681 \pm 0.009}$	$\mathbf{0.675 \pm 0.008}$	$\mathbf{0.671 \pm 0.009}$
5-HSDA	0.359 ± 0.011	0.650 ± 0.011	0.640 ± 0.011	0.633 ± 0.011

Stanford OCR performance.

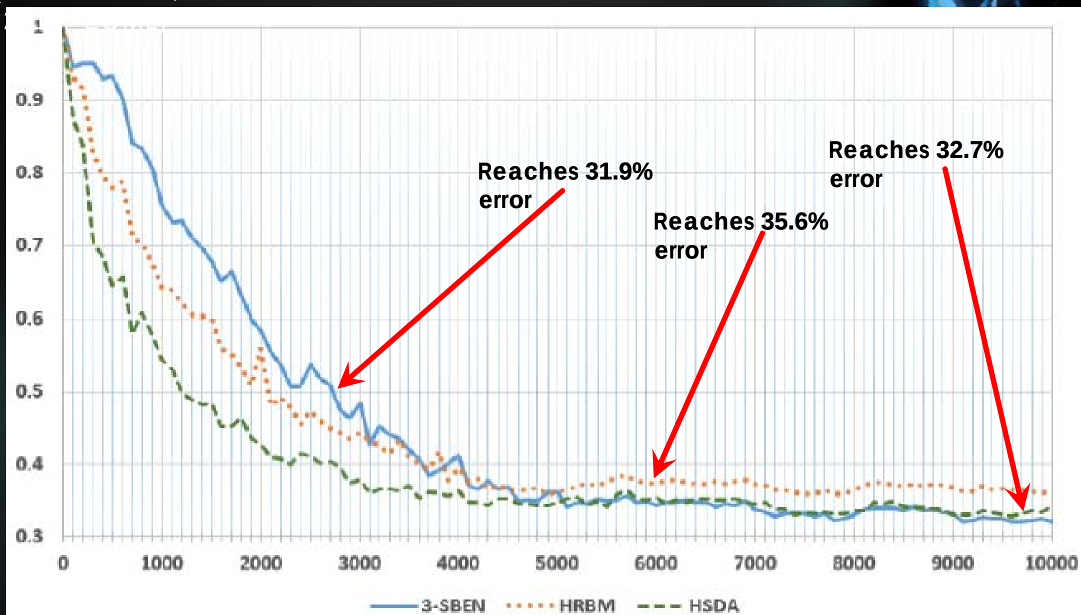
	Error	Precision	Recall	F1-Score
MaxEnt	0.425 ± 0.009	0.508 ± 0.006	0.563 ± 0.005	0.512 ± 0.006
SVM	0.428 ± 0.008	0.504 ± 0.004	0.582 ± 0.011	0.510 ± 0.007
3-Rect [13, 23]	0.387 ± 0.009	0.549 ± 0.009	0.592 ± 0.014	0.548 ± 0.011
HRBM [20]	0.399 ± 0.019	0.565 ± 0.009	0.606 ± 0.016	0.552 ± 0.014
3-SBEN	$\mathbf{0.333 \pm 0.009}$	$\mathbf{0.602 \pm 0.009}$	$\mathbf{0.668 \pm 0.009}$	$\mathbf{0.610 \pm 0.012}$
3-HSDA	0.399 ± 0.012	0.546 ± 0.007	0.601 ± 0.012	0.537 ± 0.009

WEBKB text classification performance.

	Error	Precision	Recall	F1-Score		Error	Precision	Recall	F1-Score
MaxEnt	0.510	0.386	0.387	0.384	*3-SBEN*	0.210	0.788	0.770	0.769
SVM	0.524	0.404	0.378	0.387	*3-HSDA*	0.219	0.757	0.780	0.765

Note: SBEN & HSDA were trained greedily.

(Ororbia et al.,

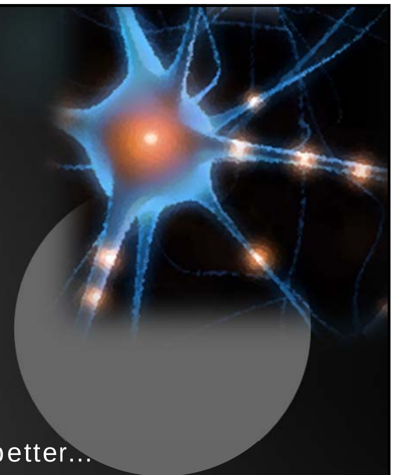


CAPTCHA: Online error (next 1000 samples) vs. iteration. SBEN & HSDA trained with BU algorithm.

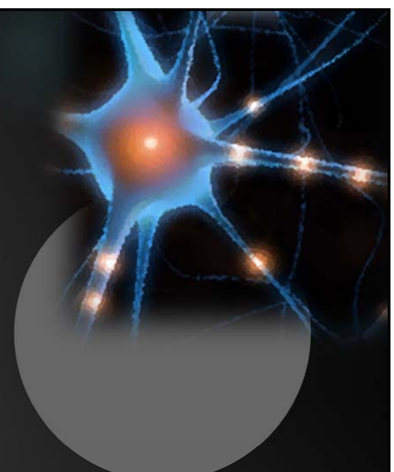
Ways to Improve Model

- More data (especially more labeled samples)
- Use drop-out
- Try different number of hidden layers & sizes
- Grid-search learning rate and β
  - Anneal β (low at start, high towards end)
  - Anneal learning rate (low at towards end)
- Use a different optimizer (i.e., AdaDelta, AdaGrad, RMSProp, etc.)
  - Adapt learning rate automatically

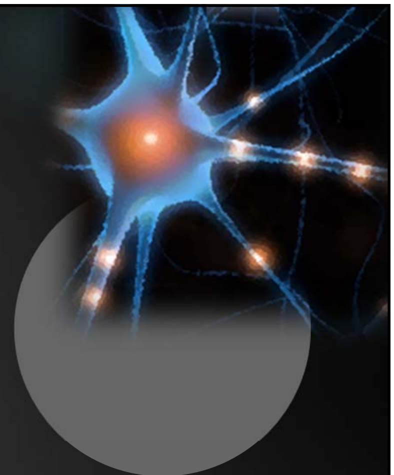
Other Approaches



- Entropy-Regularization is only a simple neural approach
  - More principled, joint modeling frameworks
    - Deep hybrid models (Ororbia et al., 2015….)
  - Semi-supervised Ladder Networks
  - Manifold Tangent Classifier
- Do not have to use neural models, sometimes simpler is better…
  - Transductive SVMs (use test set in training)
  - Self-training SVMs ((via entropy regularization)
  - Naïve Bayes via Expectation Maximization (McCallum …)

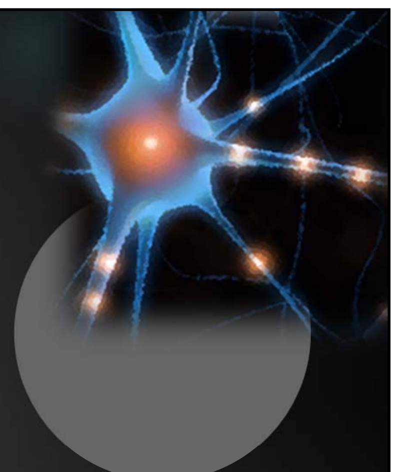


Questions?



Resource & References

FOR FURTHER, PERSONAL EXPLORATION
(CURRENTLY UPDATING…)



Resources

- Deep Learning Hub:
  - http://deeplearning.net
- Deep Learning Book (MIT Press):
  - http://www.deeplearningbook.org/
- Deep Learning frameworks:
  - Theano (has automatic differentiation built-in naturally)
    - http://deeplearning.net/software/theano/
  - TensorFlow
    - https://www.tensorflow.org
  - Keras (good for starting out)
    - http://keras.io

Bibliography

- Anderson, J. A., Silverstein, J. W., Ritz, S. A., & Jones, R. S. (1977). Distinctive features, categorical perception, and probability learning: some applications of a neural model. *Psychological Review*, *84*(5), 413.
- Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems*, *19*, 153.
- Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." *Neural Networks, IEEE Transactions on* 5.2 (1994): 157-166.
- Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273–297.
- Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*(2), 179–211.
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*(4), 193–202.
- Hastad, J. (1987). *Computational limitations of small-depth circuits*. MIT press. Retrieved from http://dl.acm.org/citation.cfm?id=SERIES9056.27031
- Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

Bibliography

- Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), 107–116.
- Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366.
- Le, Quoc V., Navdeep Jaitly, and Geoffrey E. Hinton. "A simple way to initialize recurrent networks of rectified linear units." *arXiv preprint arXiv:1504.00941* (2015).
- Minsky, M., & Papert, S. (1969). *Perceptions*. MIT Press.
- Ororbia II, Alexander G., Giles, C. Lee, and Kifer, Daniel. (2016) Unifying Adversarial Training Algorithms with Flexible Deep Data Gradient Regularization. *arXiv preprint arXiv: 1601.07213 [cs.LG].*
- Rosenblatt, F. (1957). *The perceptron -- a perceiving and recognizing automaton* (No. 85-460-1). Buffalo, NY: Cornell Aeronautical Laboratory.
- Rummelhart, Hinton, & McClelland. (1986). A General Framework for Parallel Distributed Processing. *Parallel Distributed Processing*, *1*(2).
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. doi:10.1038/323533a0
- Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, *11*, 3371–3408.