Deep Learning for Natural Language Processing

Base neural architectures

Plan of the lecture

- **Part 1**: Logistic regression
- **Part 2**: Feedforward neural network
- **Part 3**: Training of neural networks

Inspired by the brain: neuron

- **In nature**:
  - A **neuron** is an electrically excitable cell that processes and transmits information by **electrochemical signalling**.
  - The average human brain has about **86 billion neurons**.
  - Each neuron may be **connected to up to 10,000** other neurons.
  - Each neuron is passing signals to each other via as many as **1,000 trillion synaptic connections**.

- **In computers**:
  - T5-Large: **0.7 billion parameters**
  - T5-XXL: **11 billion parameters**
  - GPT3: **175 billion parameters**
  - BLOOM**: 176 billion parameters**
  - PaLM**: 540 billion parameters**
  - Switch Transformer: **1.6 trillion parameters**

Source: https://human-memory.net/brain-neurons-synapses/

Artificial neurons

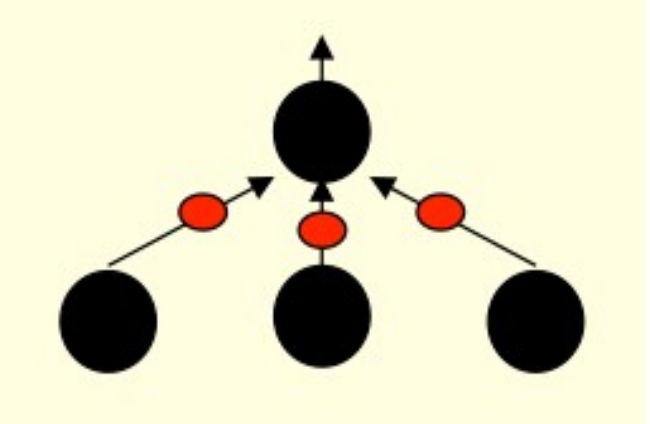
Most of them are wrong models of real neurons ...

- For instance, use **real values** to communicate

… but allow to apply effective algorithms to do useful things.

Artificial neurons first compute weighted sum of inputs (preactivation):

$$z = b + \sum_i x_i * w_i$$

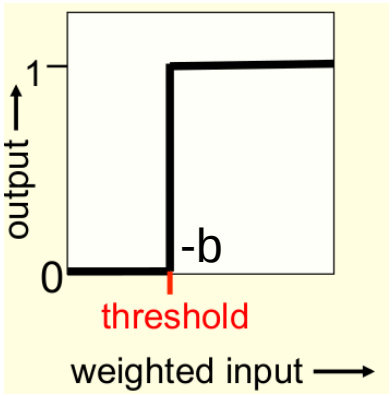


Figures from Hinton. Neural Networks for Machine Learning course @ coursera.org

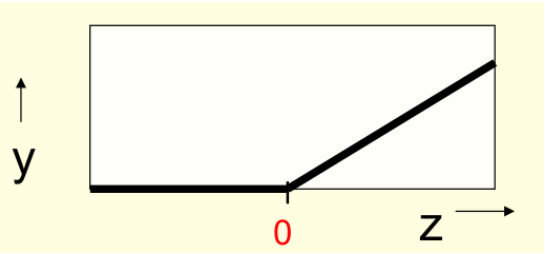
Artificial neuron activations

Then apply activation function:

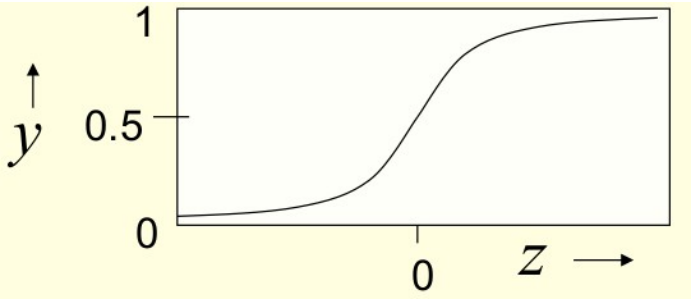
- Linear neuron: $y = z$
- Binary threshold neuron: $y = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$



- Rectified Linear unit: $y = max(0, z)$



- Sigmoid neuron: $y = \frac{1}{1+e^{-z}}$



- Tanh neuron: $y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

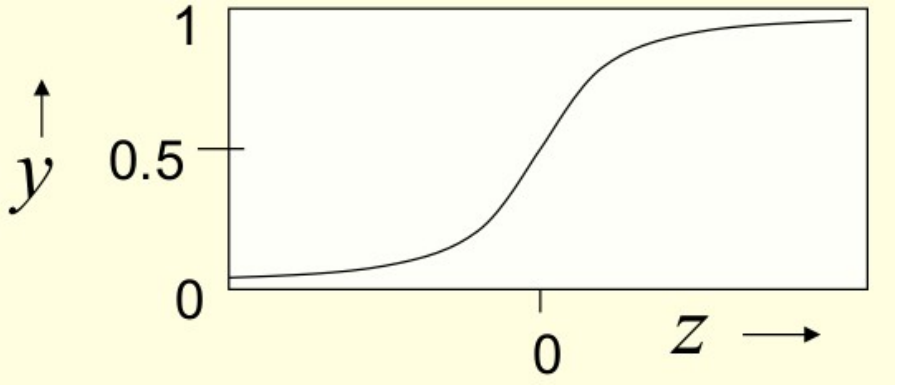
Figures from Hinton. Neural Networks for Machine Learning course, 2012 @ coursera.org

Logistic Regression (LR)

- Logistic regression hypothesis:

$$z = b + w^T x = b + \sum_{j=1}^{m} w_i * x_i$$

$$\hat{y} = h_w(x) = \sigma(z) = \frac{1}{1+e^{-z}}$$



- Implementation tricks:
  - Treat bias as w_0:

$$z = [1; x]^T w \left(concatenate\, x_0 = 1, then\, w_0 = b\right)$$

  - Vectorize: compute for all examples in parallel:

$$Z = Xw$$
$$\hat{Y} = \sigma(Z)$$

Figure from Hinton. Neural Networks for Machine Learning course, 2012 @ coursera.org

Cross-entropy loss

- Measures divergence of two probability distributions
- Binary cross-entropy loss for classification

$$E(w) = -\frac{1}{N}\sum_{i=1}^{N} y_{\{i\}} \log\left(h_w\left(x_{\{i\}}\right)\right) + \left(1 - y_{\{i\}}\right) \log\left(\left(1 - h_w\left(x_{\{i\}}\right)\right)\right)$$

  - for LR – convex w.r.t. weights
  - Justified by Maximum Likelihood

- Cross-entropy loss for multiclass

$$E(w) = -\frac{1}{N}\sum_{i=1}^{N} \log\left(\left[h_w\left(x_{\{i\}}\right)\right]_t\right), t = argmax\, y_{\{i\}}\left(true\ class\right)$$

Binary Logistic Regression

- Since there are two discrete outcomes (1 or 0) this is a Bernoulli distribution:

$$p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$$

- Maximize the log of the probability:

$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^y (1-\hat{y})^{1-y}\right] \\ &= y \log \hat{y} + (1-y) \log(1-\hat{y}) \end{aligned}$$

- Flip the sign:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$$

- Plug in the definition of $\hat{y} = \sigma(w \cdot x + b)$:

$$L_{CE}(w, b) = -[y \log \sigma(w \cdot x + b) + (1-y) \log (1 - \sigma(w \cdot x + b))]$$

- Exam

$$\begin{aligned} L_{CE}(w, b) &= -[y \log \sigma(w \cdot x + b) + (1-y) \log (1 - \sigma(w \cdot x + b))] \\ &= -[\log \sigma(w \cdot x + b)] \\ &= -\log(.69) \\ &= .37 \end{aligned}$$

Softmax

- The softmax function takes a vector $z = [z_1,z_2,\ldots,z_k]$ of k arbitrary values and maps them to a probability distribution:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le k$$

$$\text{softmax}(z) = \left[\frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}, \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}, \ldots, \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}} \right]$$

Multinomial Logistic Regression

- Dot product between a weight vector *w* and an input vector *x* (plus a bias).
- But now we'll need separate weight vectors (and bias) for each of the *K* classes:

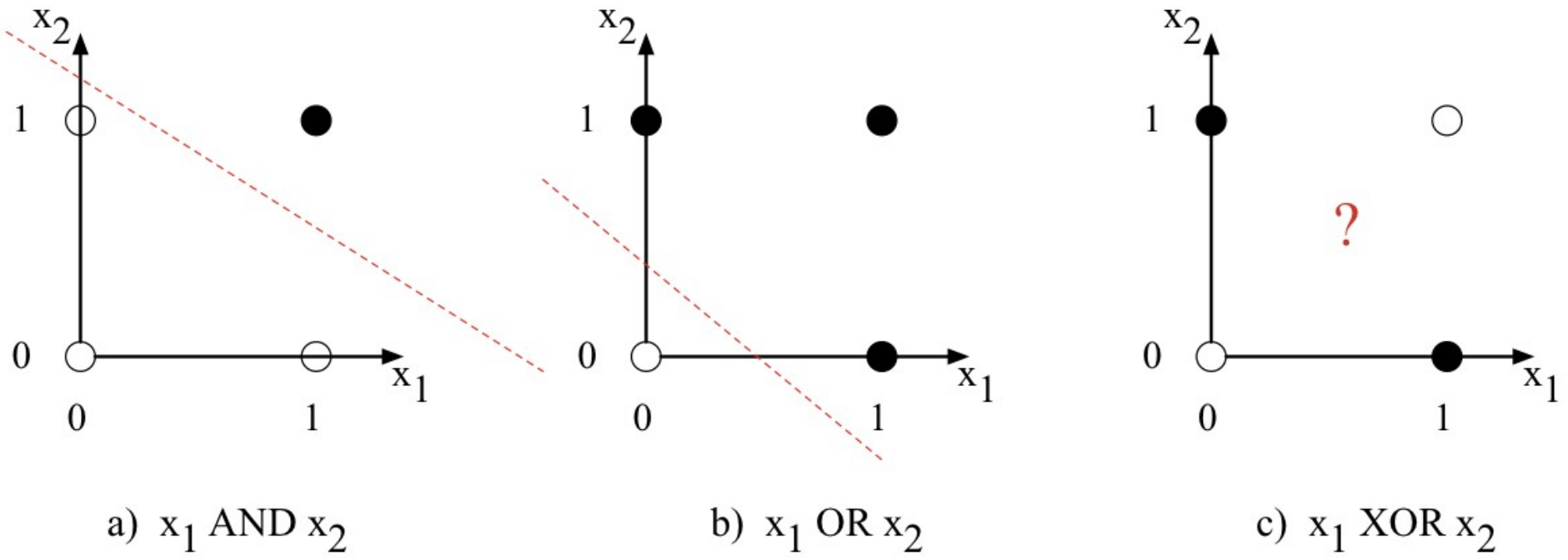
$$p(y=c|x) = \frac{e^{w_c \cdot x + b_c}}{\sum_{j=1}^{k} e^{w_j \cdot x + b_j}}$$

Plan of the lecture

- **Part 1**: Logistic regression
- **Part 2**: Feedforward neural network
- **Part 3**: Training of neural networks

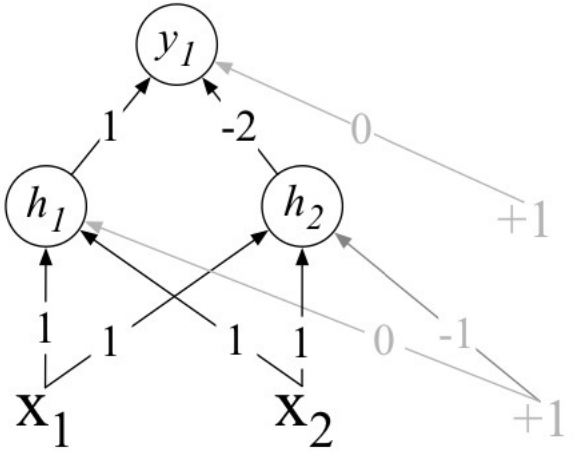
XOR problem

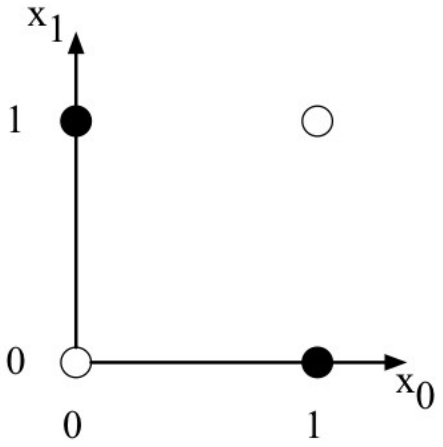
- There is no way to draw a line that correctly separates the two categories for XOR:



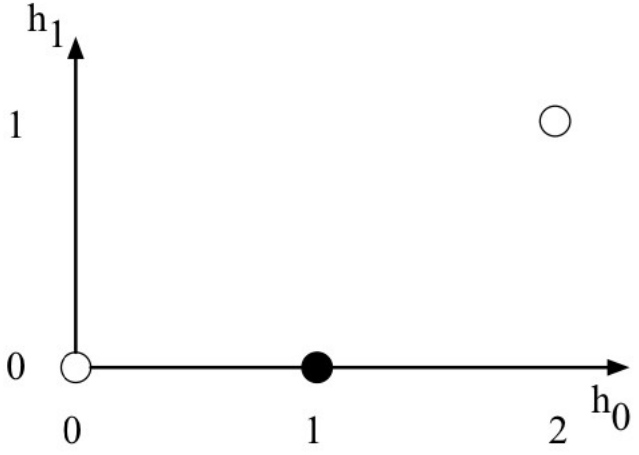
XOR problem solution

The hidden layer forming a new representation of the input. The input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR.





a) The original *x* space



b) The new *h* space

Representation power of FFNNs

- From Goldberg (2017), Chapter 4.3:

In terms of representation power, it was shown by Hornik et al. [1989] and Cybenko [1989] that MLP1 is a universal approximator—it can approximate with any desired non-zero amount of error a family of functions that includes all continuous functions on a closed and bounded subset of $\mathbb{R}^n$, and any function mapping from any finite dimensional discrete space to another.[6] This

[5]Strictly convex functions have a single optimal solution, making them easy to optimize using gradient-based methods.

[6]Specifically, a feed-forward network with linear output layer and at least one hidden layer with a "squashing" activation function can approximate any Borel measurable function from one finite dimensional space to another. The proof was later extended by Leshno et al. [1993] to a wider range of activation functions, including the ReLU function $g(x) = \max(0, x)$.

- In theory, FFNN is all you need.
- In practice, there are caveats…

Feed-Forward Neural Network (FFNN)

Output units

- Maps a network's output back to original problem
- Linear units
- Sigmoid units
- Softmax units

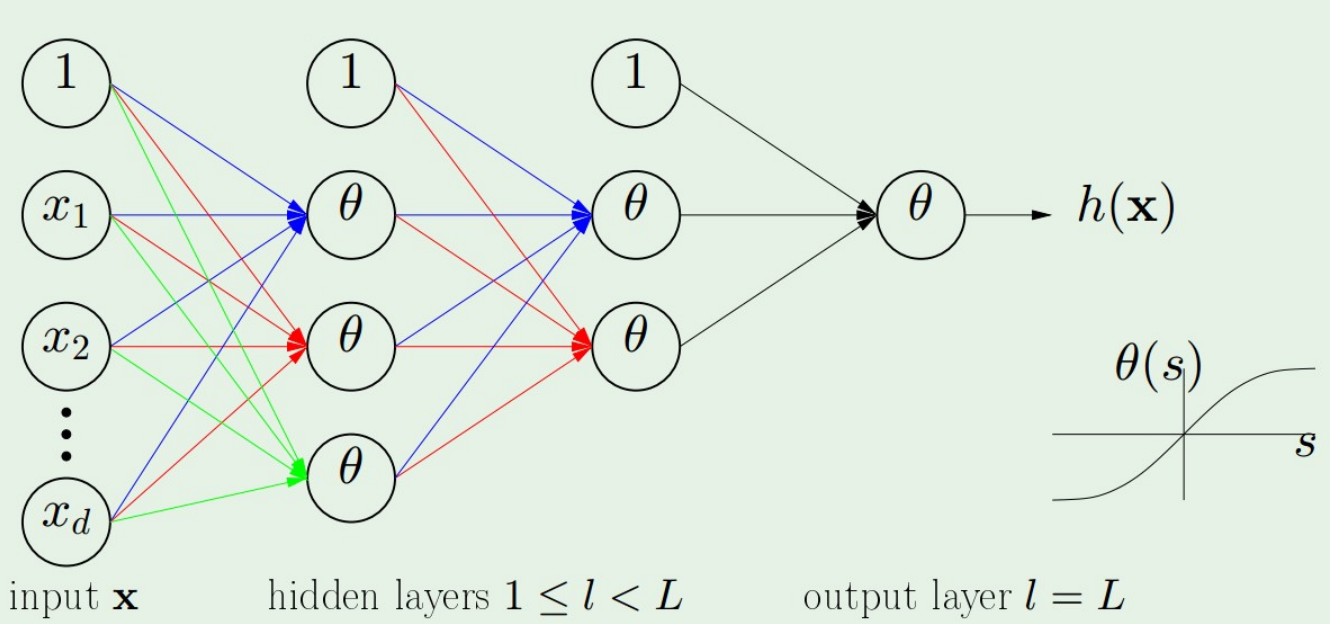


Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

Feed-Forward Neural Network (FFNN)

Hidden units

- Intermediate values of a network
- Rectified Linear units
- Logistic Sigmoid (σ) and hyperbolic tangent (tanh)

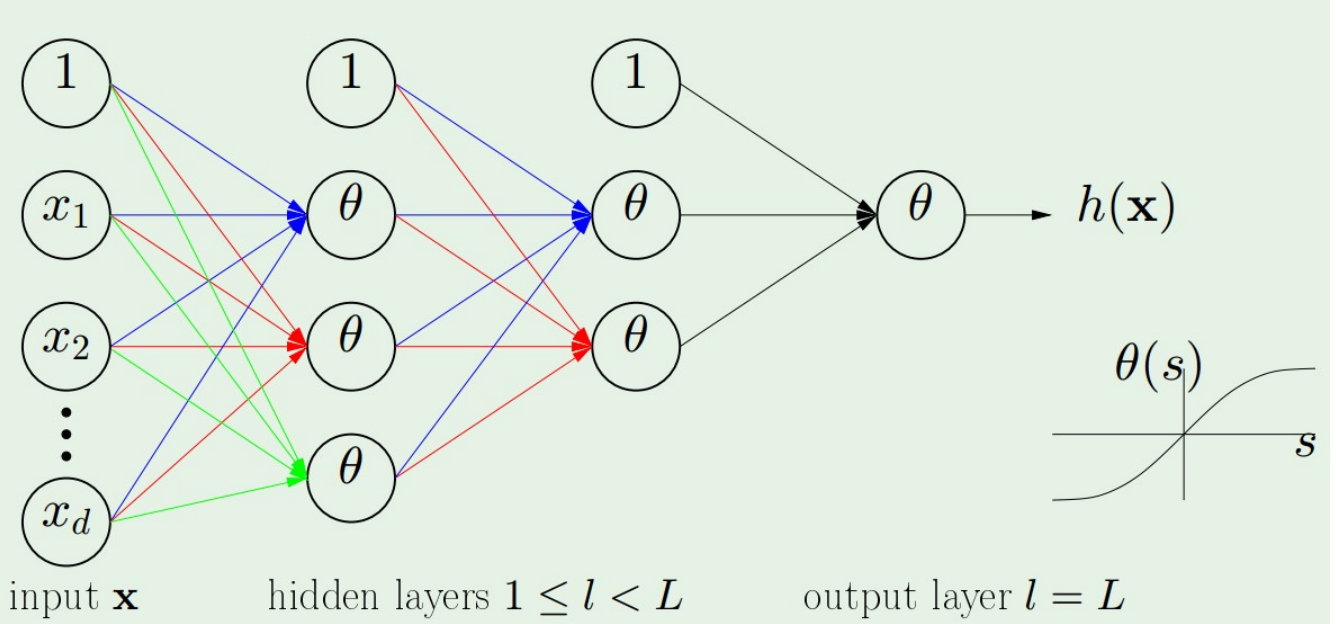
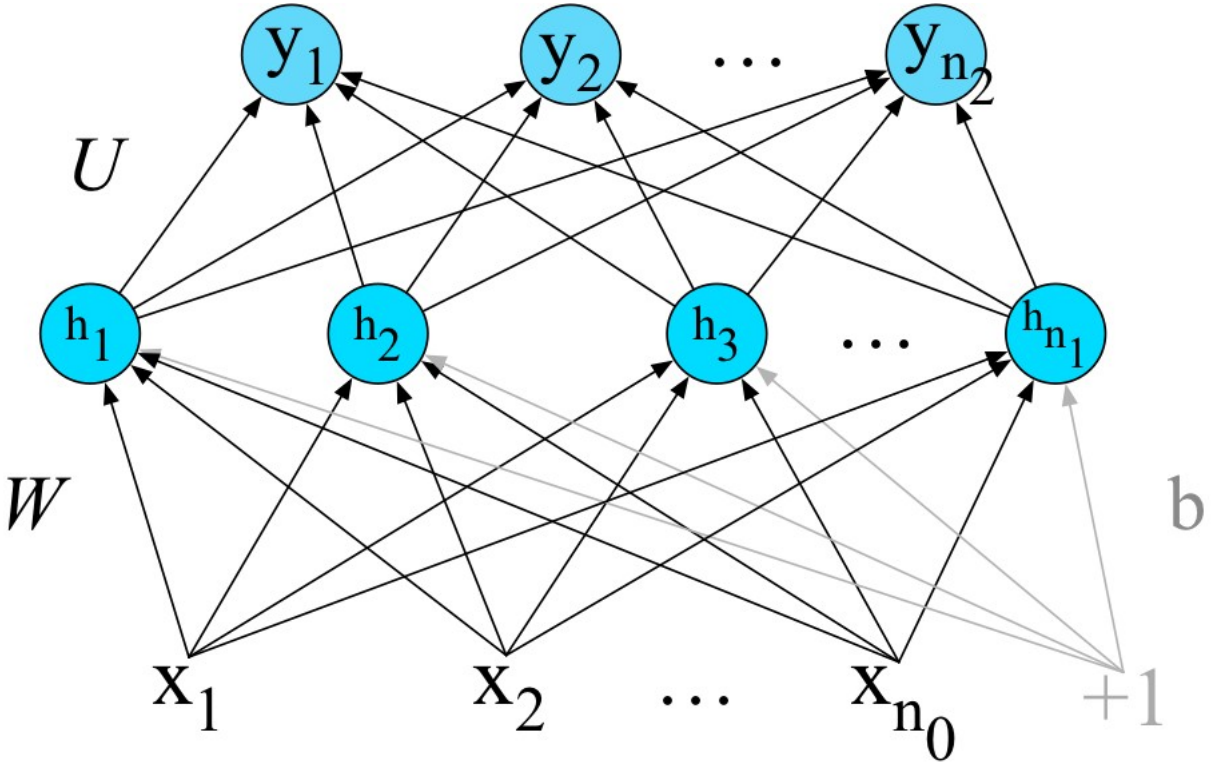


Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

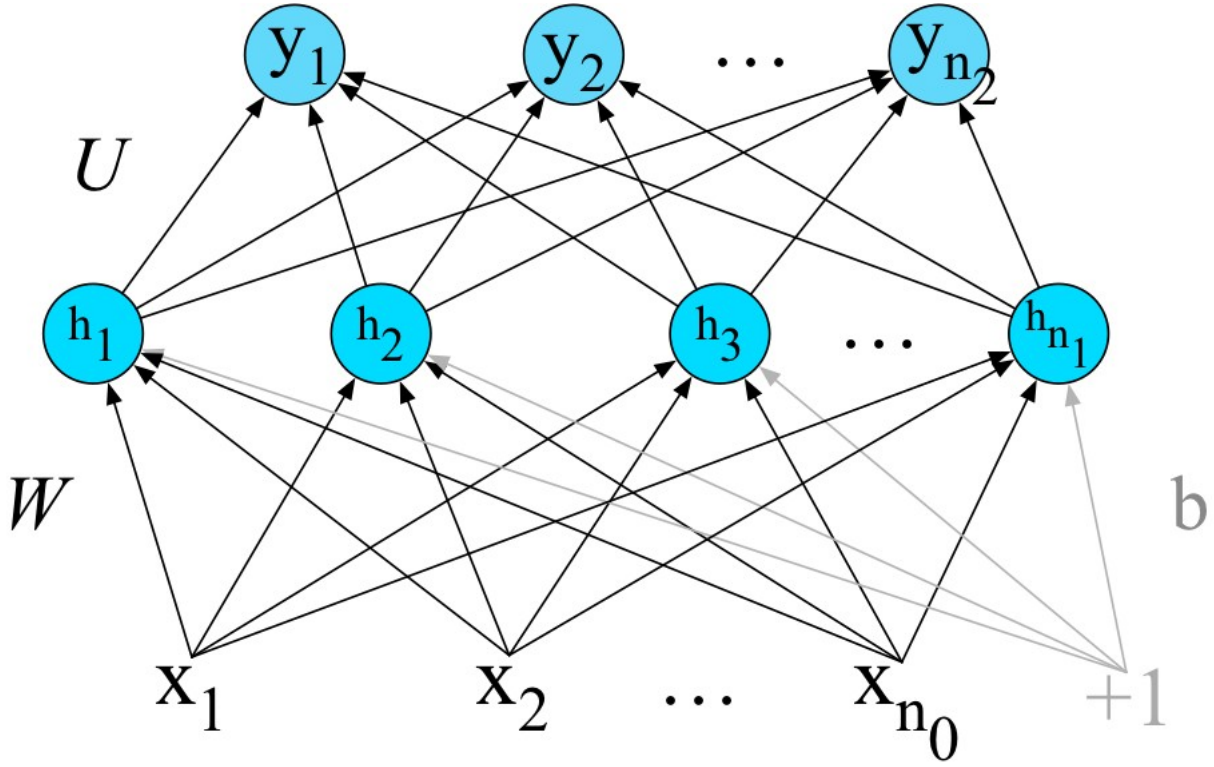
Matrix notation for FFNN

- A single hidden unit has parameters w (the weight vector) and b (the bias scalar)
- The parameters for the entire hidden layer by combining the weight vector $\mathbf{w}_i$ and bias b_i for each unit i into a single weight matrix W and a single bias vector **b** for the whole layer.

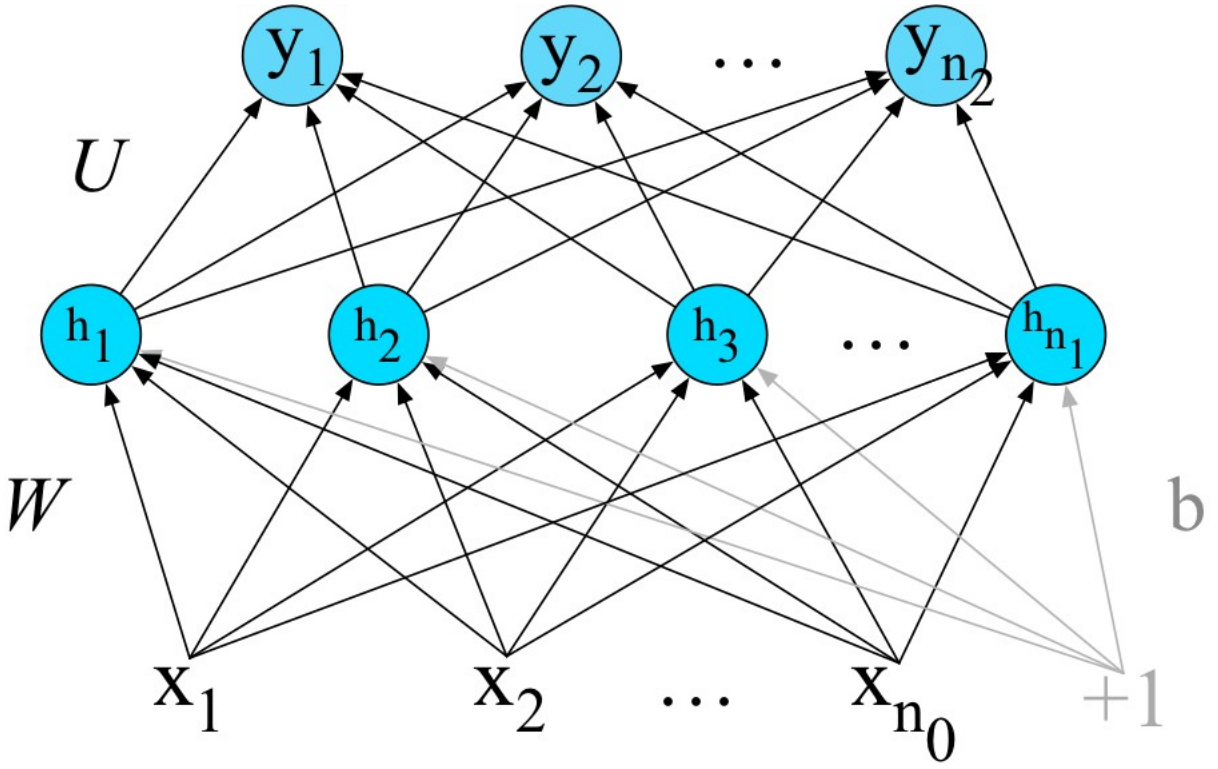


Matrix notation for FFNN

- Each element W_{ij} of the weight matrix W represents the weight of the connection from the i-th input unit x_i to the j-th hidden unit h_j.
- The output of the hidden layer, the vector h: $h = \sigma(Wx + b)$



Matrix notation for FFNN



$W \in \mathbb{R}^{n_1 \times n_0}$

$h \in \mathbb{R}^{n_1}$

$x \in \mathbb{R}^{n_0}$

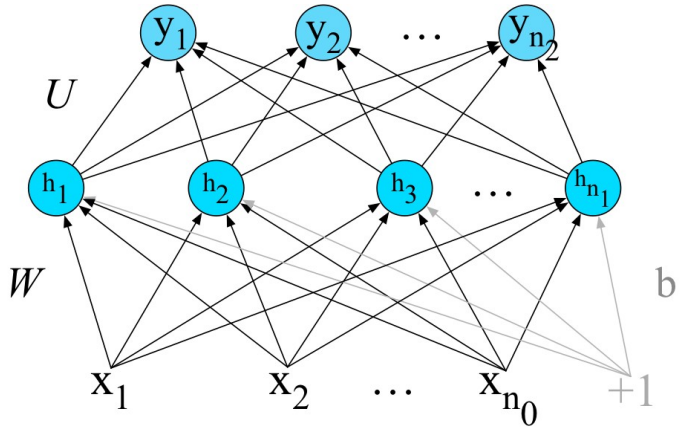
$b \in \mathbb{R}^{n_1}$

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \le i \le d$$

$$\begin{aligned} h &= \sigma(Wx+b) \\ z &= Uh \\ y &= \text{softmax}(z) \end{aligned}$$

Matrix notation for FFNN: more than two layers

- $W^{[i]}$ - weight matrix for the *i*-th hidden layer;
- $b^{[i]}$ - bias vector for the *i*-th hidden layer;
- $a^{[i]}$ - the output from the *i*-th layer;
- $z^{[i]}$ – combination of weights and biases:
  - $z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}$



$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

FFNN: Forward step in matrix notation

for i **in** 1..n

$$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}$$

$$a^{[i]} = g^{[i]}(z^{[i]})$$

$$\hat{y} = a^{[n]}$$

Plan of the lecture

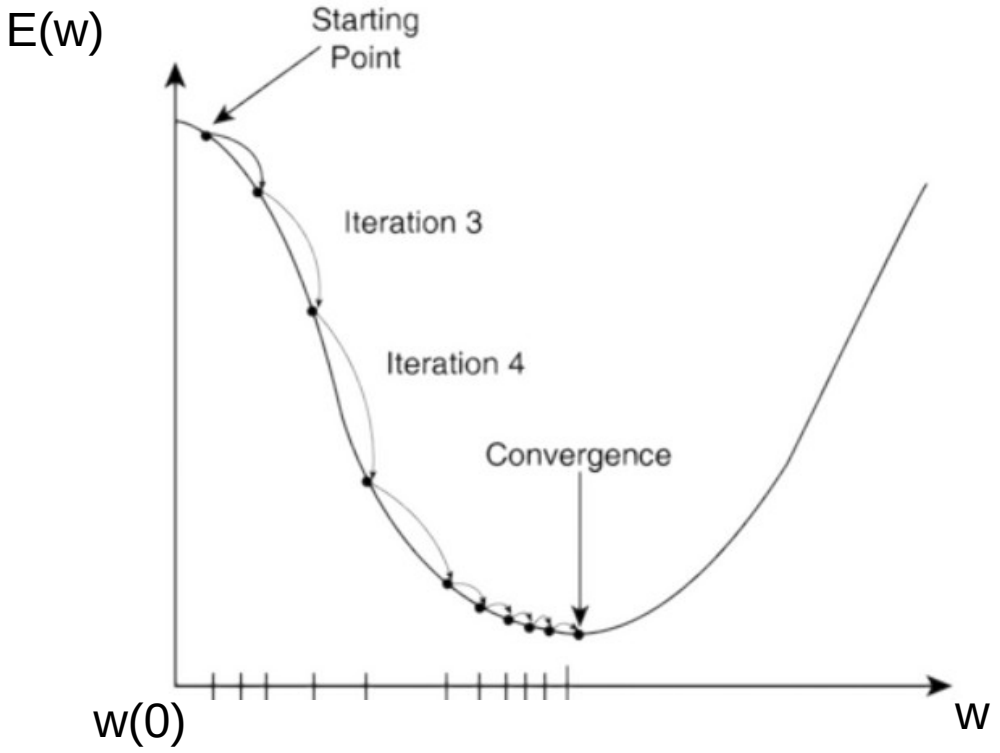
- **Part 1**: Logistic regression
- **Part 2**: Feedforward neural network
- **Part 3**: Training of neural networks

1D gradient descent: idea

- Start with random weights w(0)
- Until convergence:

  take a (small) step in the direction of descent

  - Left if E’(w)>0
  - Right E’(w)<0

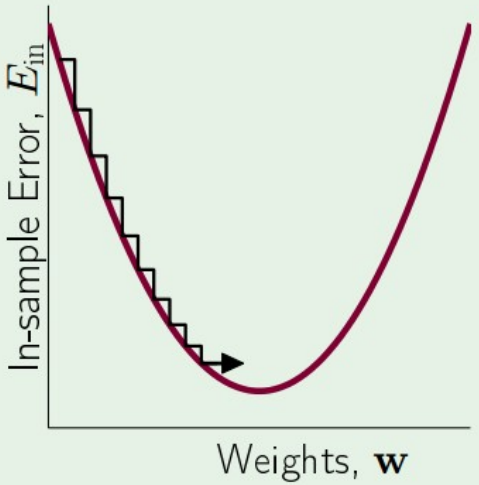


Gradient descent: fixed step size

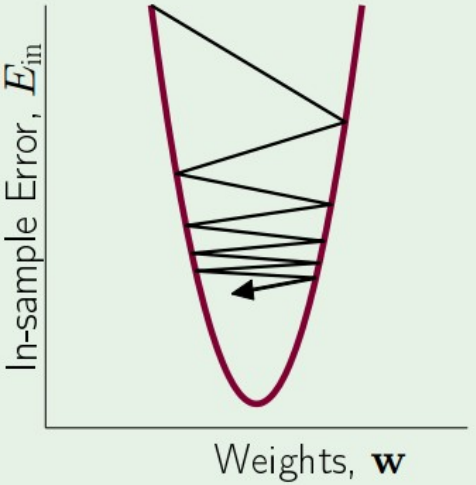
- Start with random weights w(0)
- Until convergence:

$$w(t+1) := w(t) - \eta \, sign(E'(w(t)))$$

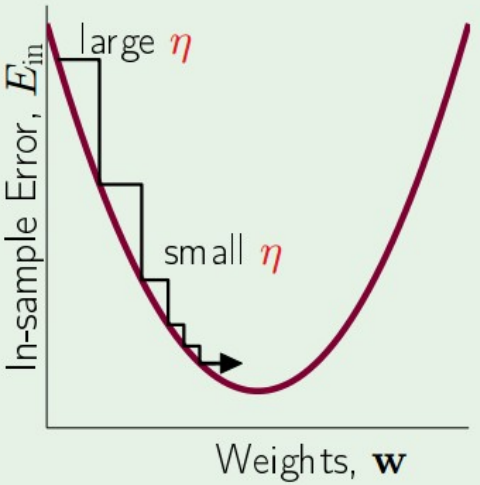
How η affects the algorithm:



η too small



η too large



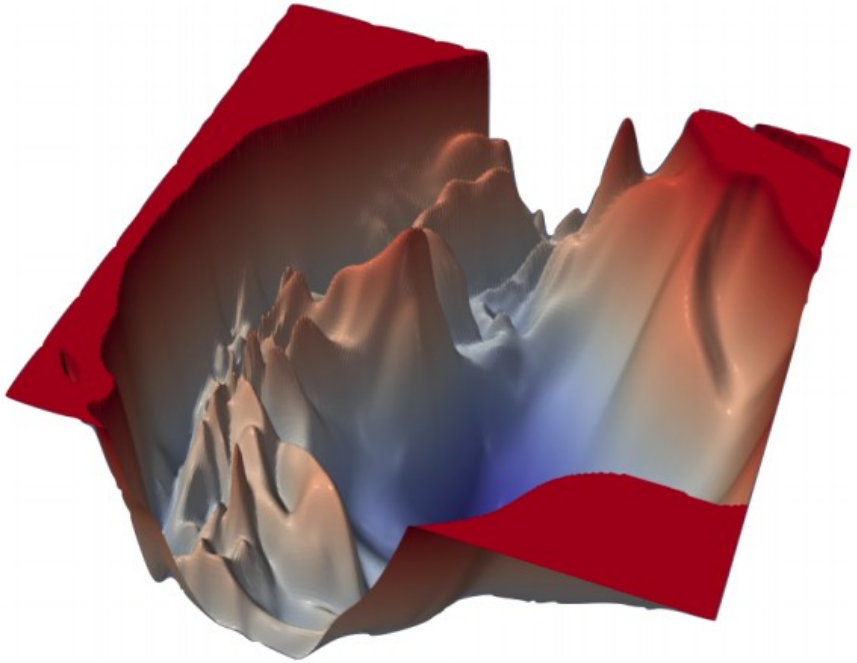
variable η – just right

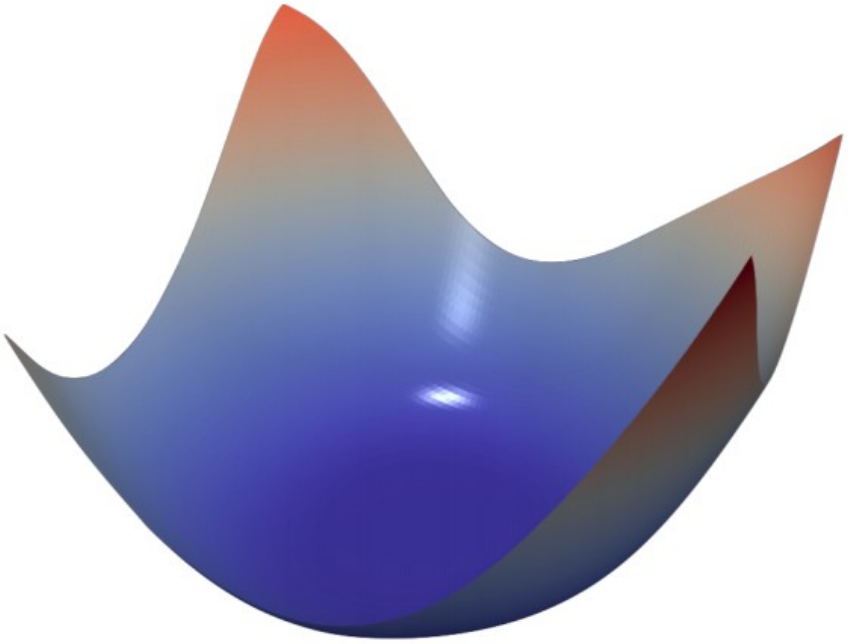
η should increase with the slope

Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

Loss surface visualization

- Because of their non-linearity, cost functions are non-convex
- No global guarantee of convergence
- Still, methods based on gradient-descent are used
- Computing gradient is more complicated
- More care to how weights are initialized





Images from Li et al. VISUALIZING THE LOSS LANDSCAPE OF NEURAL NETS, 2018

Gradient descent in practice

Gradient descent needs many (thousands, millions) steps to converge, but when 1 step can be slow (sum over whole dataset).

- **Mini-batch training**:
  - In between full batch training and stochastic gradient descent
  - At each iteration, samples a batch of m examples from training set and computes an update
  - Estimate loss & its gradient on 1-100 examples: known as SGD (stochastic/mini-batch gradient descent)

Gradient descent in practice

- Batch gradient descent: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

- Stochastic gradient descent: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

- Mini-batch gradient descent: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Source: https://arxiv.org/pdf/1609.04747.pdf

Gradient descent in practice

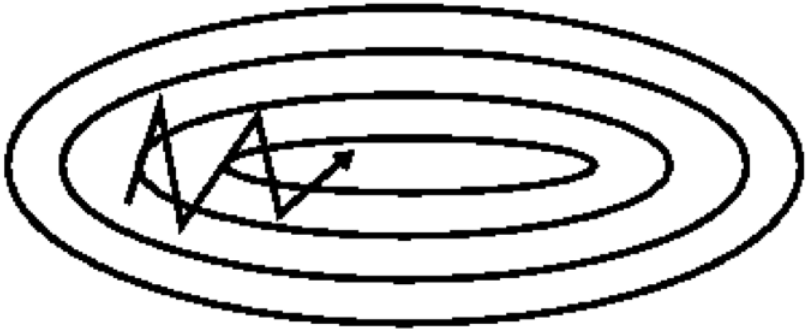
Gradient descent needs many (thousands, millions) steps to converge, but when 1 step can be slow (sum over whole dataset).

- **Shuffle dataset**
  - What happens if we showed first all cats, then all dogs?
- **Lower learning rate while training**
  - known as learning rate decay / annealing

Momentum

- SGD has trouble navigating ravines:
  - areas where the surface curves much more steeply in one dimension than in another;
  - which are common around local optima.
- Momentum helps accelerate SGD in the relevant direction and dampens oscillations

(a) SGD without momentum



(b) SGD with momentum

Source: https://arxiv.org/pdf/1609.04747.pdf

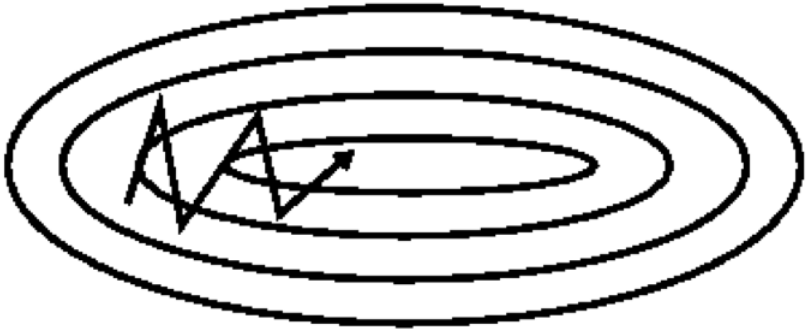
Momentum

- Momentum adds a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

- Term γ is usually set to 0.9 or a similar value

(a) SGD without momentum



(b) SGD with momentum

Source: https://arxiv.org/pdf/1609.04747.pdf

Momentum

- When using momentum, we push a ball down a hill.
  - The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity, if there is air resistance, i.e. $\gamma < 1$).
- The same thing happens to our parameter updates:
  - The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.
  - As a result, we gain faster convergence and reduced oscillation.
  - Used in multiple variants of SGD:
    - Adam, RMSProp

Source: https://arxiv.org/pdf/1609.04747.pdf

Adaptive learning rate

- Adaptive learning rates for each parameter: compute gradient for a given parameter:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

- Perform update depending on the gradient value
- Compare SGD to Adagrad:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

- Here G_t is a diagonal matrix where each diagonal element ii is the sum of the squares of the gradients.

Source: 2nd CCIS summer school, Umass Lowel, https://ruder.io/optimizing-gradient-descent/index.htm

Various other adaptive learning rate optimization algorithms

- RMSProp

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- Adamax

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$$

- Adam

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

Adaptive learning rate (Adam)

- An **adaptive learning rate** optimization algorithm
  - adaptive learning rates for each parameter
- Incorporates concept of **momentum**
  - Decaying averages of past and past squared gradients m_t and v_t
- Momentum is realized by an (exponentially decaying) estimate of the first-order moment of the gradient v_t
  - Corrects bias in estimation of first and second-order moments of gradients

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

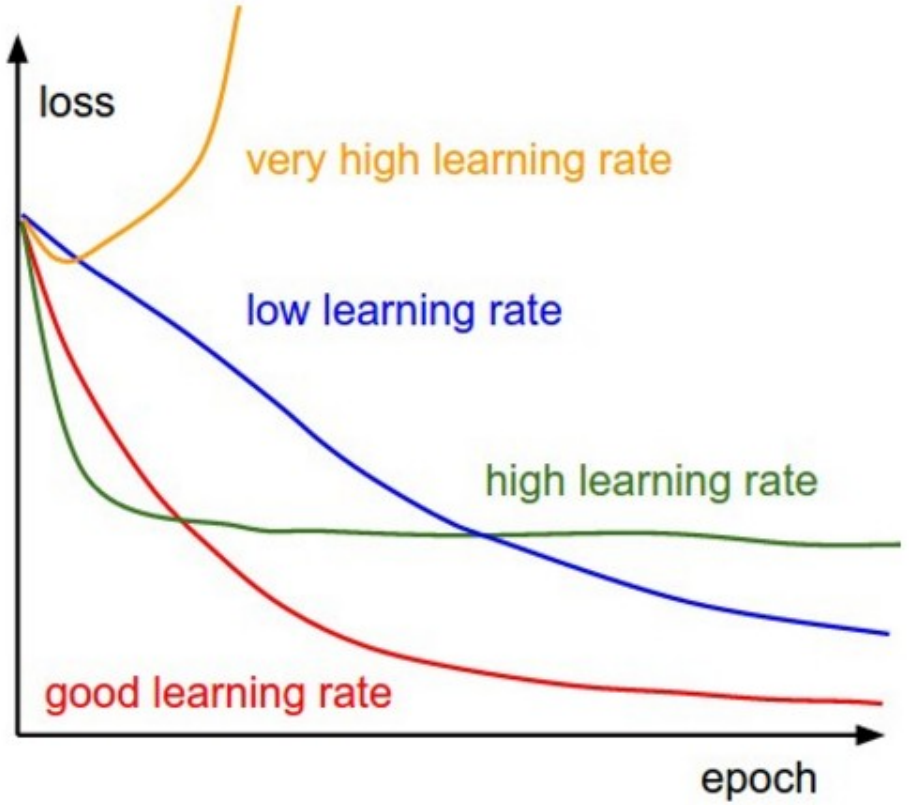
default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ.

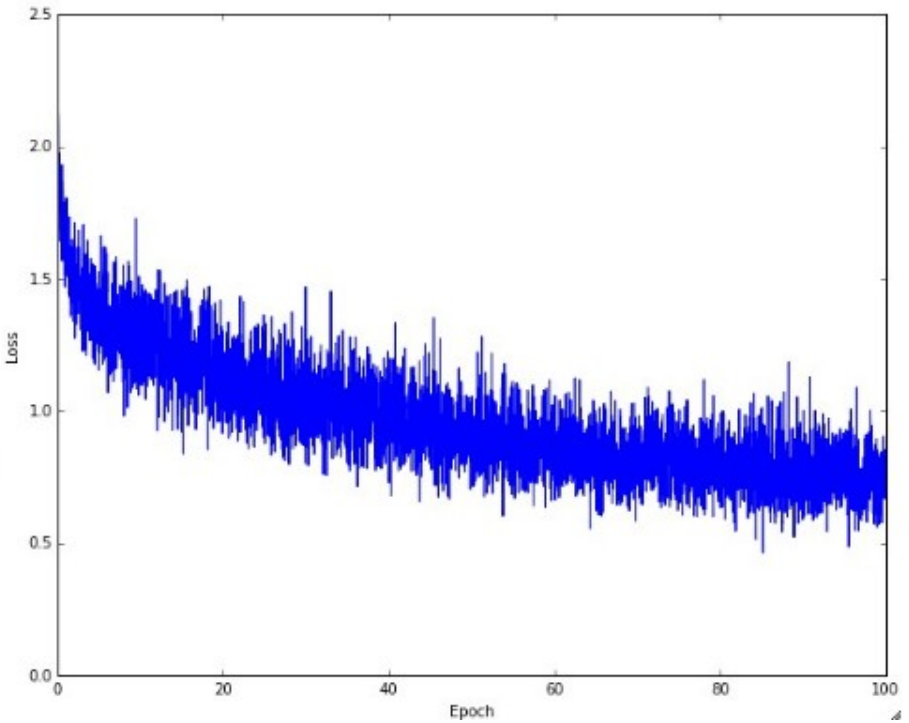
https://ruder.io/optimizing-gradient-descent/index.html#adam https://arxiv.org/pdf/1412.6980.pdf

Learning curves

Plot learning curves to see how loss/accuracy changes while training!

- Select learning rate resulting in fastest loss decrease
- Lower learning rate when the loss plateaus



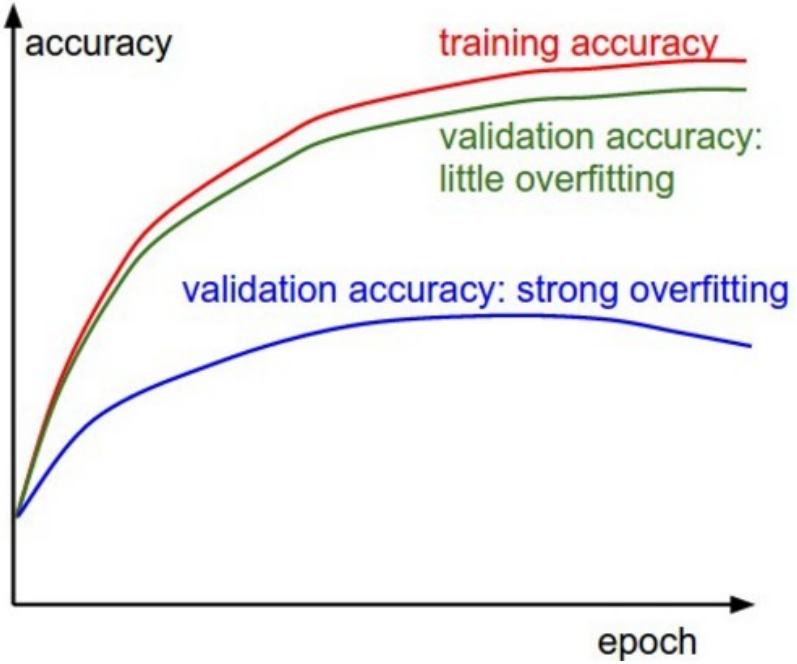


Images from CS231n Convolutional neural Networks for Visual Recognition course http://cs231n.github.io/neural-networks-3/

Learning curves

Plot learning curves for both train and validation sets!

- Remember our final goal: work on new examples!
- Distinguish bad model vs. bad optimization
  - Loss/accuracy is bad on validation set, but good on train set
    => bad model (overfitting)
  - Loss/accuracy on train set is bad
    => maybe optimization problems?



Images from CS231n Convolutional neural Networks for Visual Recognition course http://cs231n.github.io/neural-networks-3/

Gradient of Logistic Regression

- Loss function (cross-entropy):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{CE}(y^{(i)}, x^{(i)}; \theta)$$

$$L_{CE}(w, b) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log (1 - \sigma(w \cdot x + b))]$$

- We need to find the gradient of this function:

$$\nabla_\theta L(f(x;\theta), y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta), y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta), y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta), y) \end{bmatrix}$$

- For one component:

$$\frac{\partial L_{CE}(w, b)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$

Derivation of the gradient can be found at: https://web.stanford.edu/~jurafsky/slp3/5.pdf

Multinomial Logistic Regression

- Derivative is the difference between the value for the true class *k* (which is 1) and the probability the classifier outputs for class *k*, weighted by the value of the input x_k.
- Function 1{} which evaluates to 1 if the condition in the brackets is true and to 0 otherwise:

$$\begin{aligned}\frac{\partial L_{CE}}{\partial w_k} &= -(1\{y=k\} - p(y=k|x))x_k \\ &= -\left(1\{y=k\} - \frac{e^{w_k \cdot x + b_k}}{\sum_{j=1}^{K} e^{w_j \cdot x + b_j}}\right) x_k\end{aligned}$$

Training LR with SGD

- Initialize w randomly (0s are o.k. for LR, but not for NN)
- Until convergence
  - From train set sample mini-batch of 1-100 examples
  - On mini-batch Compute loss E(w) and it's gradient
  - Update weights $$w(t+1) := w(t) - \lambda \nabla_w E(w(t))$$
  - ?Change learning rate if necessary?

Training FFNN with SGD

- Initialize w randomly
  - 0s don't work: need symmetry breaking
  - Use small random weights, better from specific distributions (Xavier, He, Glorot initialization)
- Until convergence
  - From train set sample mini-batch of 1-100 examples
  - On mini-batch Compute loss E(w) and it's **gradient**
  - Update weights

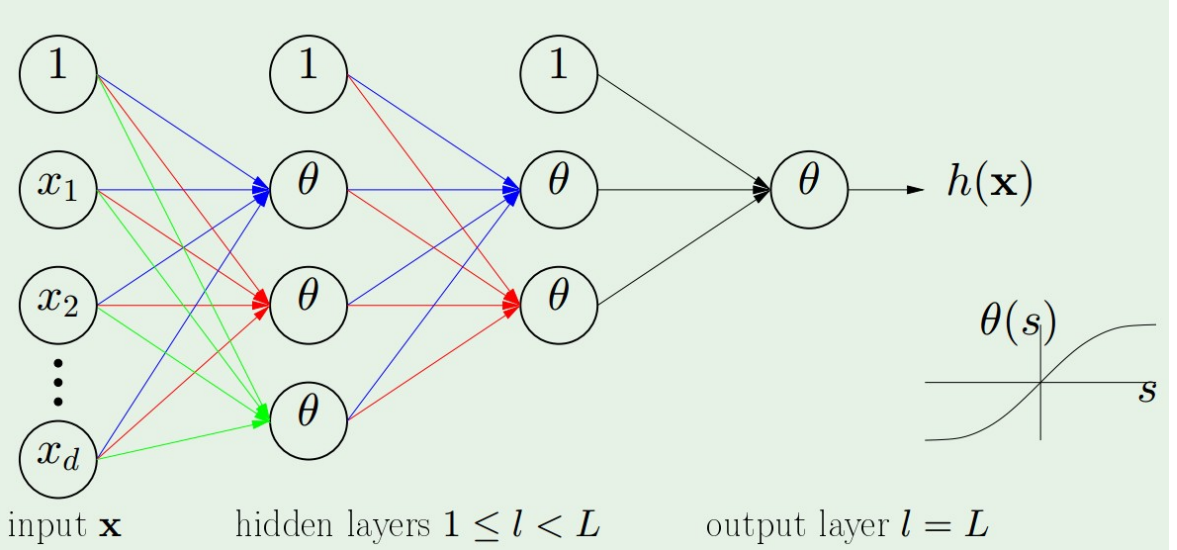
$$w(t+1) := w(t) - \lambda \nabla_w E(w(t))$$

  - Change learning rate if necessary

Gradient for FFNN

- Hypothesis:

$$a^{(0)} = x$$
$$z^{(l)} = W^{(l)}[1; a^{(l-1)}]$$
$$a^{(l)} = \sigma(z)$$
$$\hat{y} = a^{(L)}$$



- Loss:

$$E(w) = -\frac{1}{N}\sum_{i=1}^{N} y_{\{i\}} \log\left(h_w\left(x_{\{i\}}\right)\right) + \left(1 - y_{\{i\}}\right) \log\left(\left(1 - h_w\left(x_{\{i\}}\right)\right)\right)$$

- Derivatives:
  - Can use symmetric difference approximation ← very slow, use for gradient checks!
  - Can write down formula for E(w) containing all w explicitly and differentiate ← too complicated and errorprone
  - Backprop – an efficient way to calculate derivatives recurrently, based on chain rule

Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

Backpropagation

- (Rumelhart et al., 1986): error backprop-agation or backprop
- Uses the chain rule of calculus
- Determines the gradient of a function that is the composition of functions whose gradients are known

Computing the gradient for FFNN

- Negative log likelihood loss for a hard classification task:

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i = -\log \hat{y}_i = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- For the output layer we can directly compute the gradient:

$$\begin{aligned} \frac{\partial L_{CE}(w,b)}{\partial w_j} &= (\hat{y} - y)\, x_j \\ &= (\sigma(w \cdot x + b) - y)\, x_j \end{aligned}$$

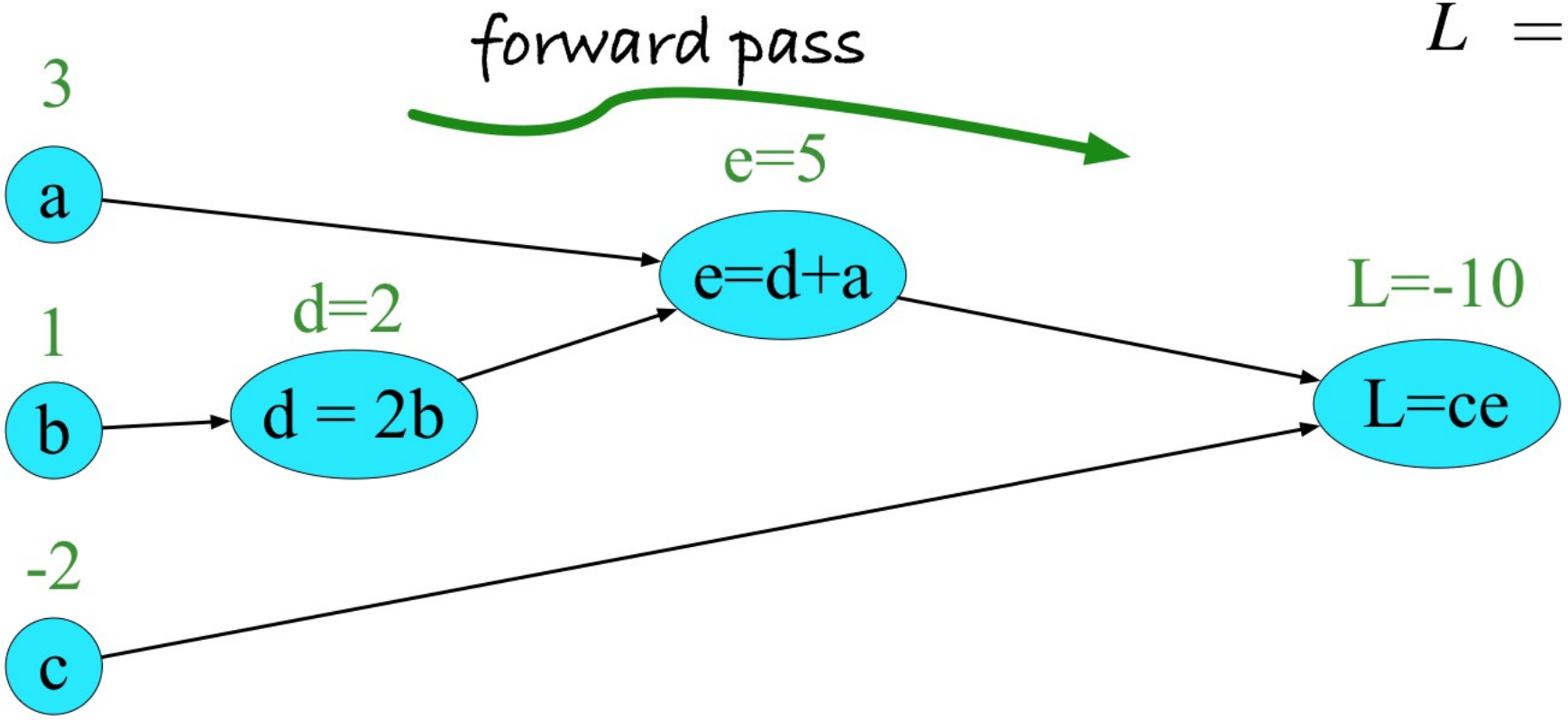
$$\begin{aligned} \frac{\partial L_{CE}}{\partial w_k} &= (1\{y = k\} - p(y = k|x))x_k \\ &= \left(1\{y = k\} - \frac{e^{w_k \cdot x + b_k}}{\sum_{j=1}^{K} e^{w_j \cdot x + b_j}}\right) x_k \end{aligned}$$

Source: https://web.stanford.edu/~jurafsky/slp3/7.pdf

Computational graphs

- Consider function: $L(a,b,c) = c(a+2b)$
- Its computational graph is:

$$d = 2 * b$$
$$e = a + d$$
$$L = c * e$$



Source: https://web.stanford.edu/~jurafsky/slp3/7.pdf

Chain rule

- For 1 argument:

$$E(w)=f(g(w))$$
$$E'(w)=f'(g(w))g'(w)$$

$$f(x) = u(v(x)) \qquad \frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

$$f(x) = u(v(w(x))), \quad \frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

- For 2 arguments: ← sum over internal functions

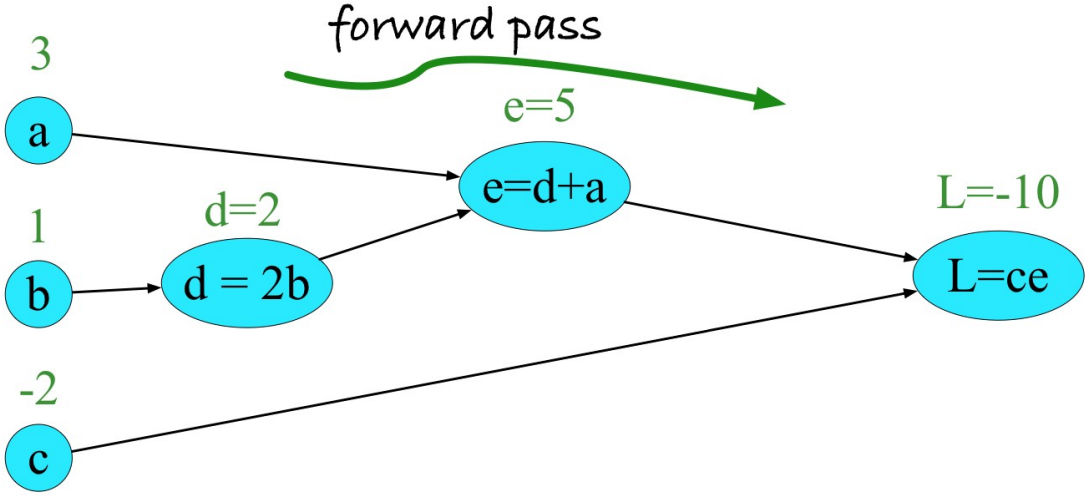
$$E(w)=f(g(w),h(w))$$
$$\frac{\partial E(w)}{\partial w}=\frac{\partial f(g(w))}{\partial g(w)}\frac{\partial g(w)}{\partial w}+\frac{\partial f(g(w))}{\partial h(w)}\frac{\partial h(w)}{\partial w}$$

Computational graphs

- In the backward pass, we compute each of these partials along each edge of the graph from right to left, multiplying the necessary partials to result in the final derivative we need.
- We begin by annotating the final node with ∂ L /∂ L= 1.
- Moving to the left, we then compute ∂ L and ∂ L , and so on, until we have annotated the graph all the way to the input variables.

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$



$$L = ce \;:\quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d \;:\quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \;:\quad \frac{\partial d}{\partial b} = 2$$

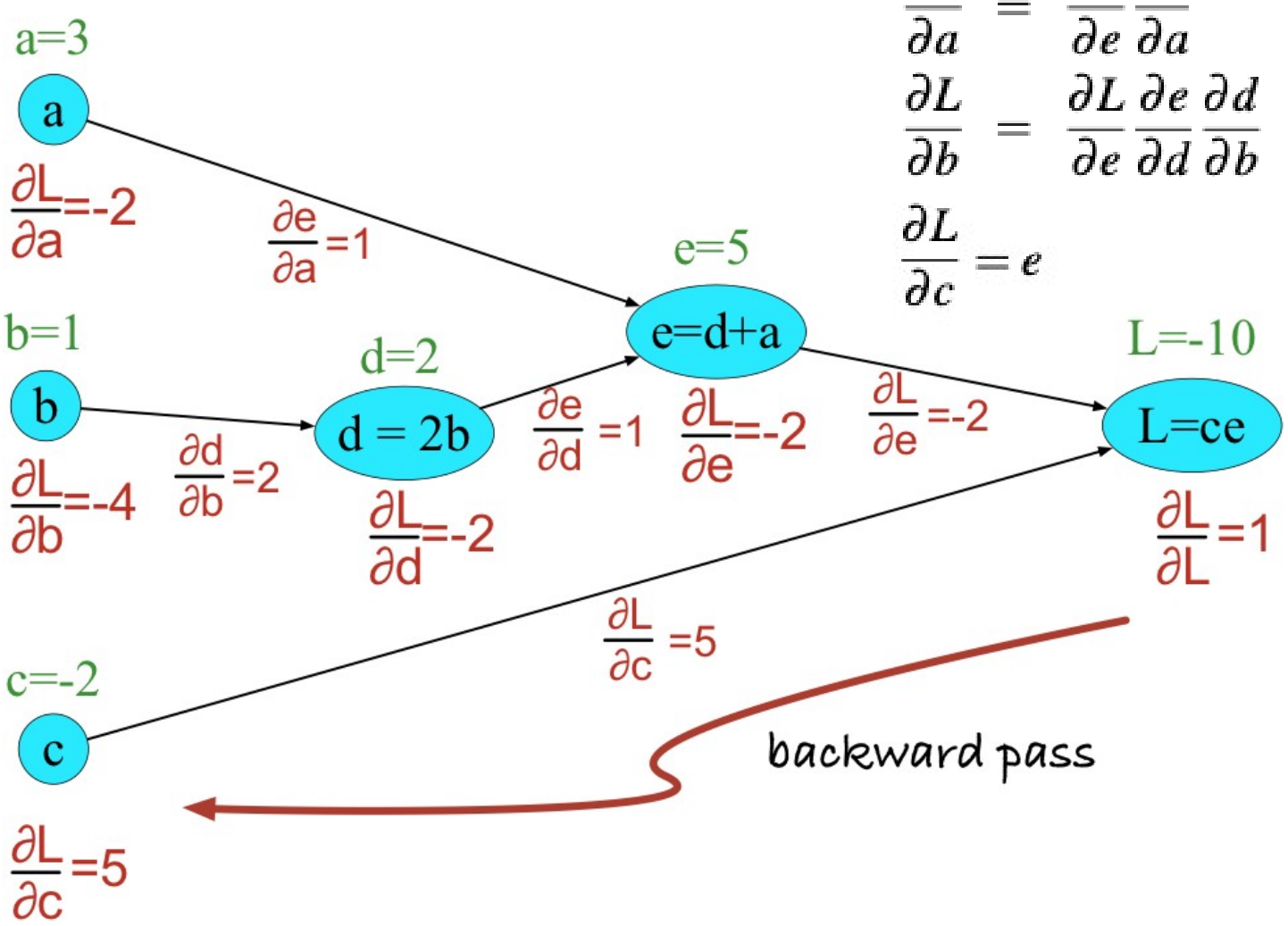
Computational graphs

- At each node we need to compute the local partial derivative with respect to the parent, multiply it by the partial derivative that is being passed down from the parent, and then pass it to the child.

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$\frac{\partial L}{\partial c} = e$$



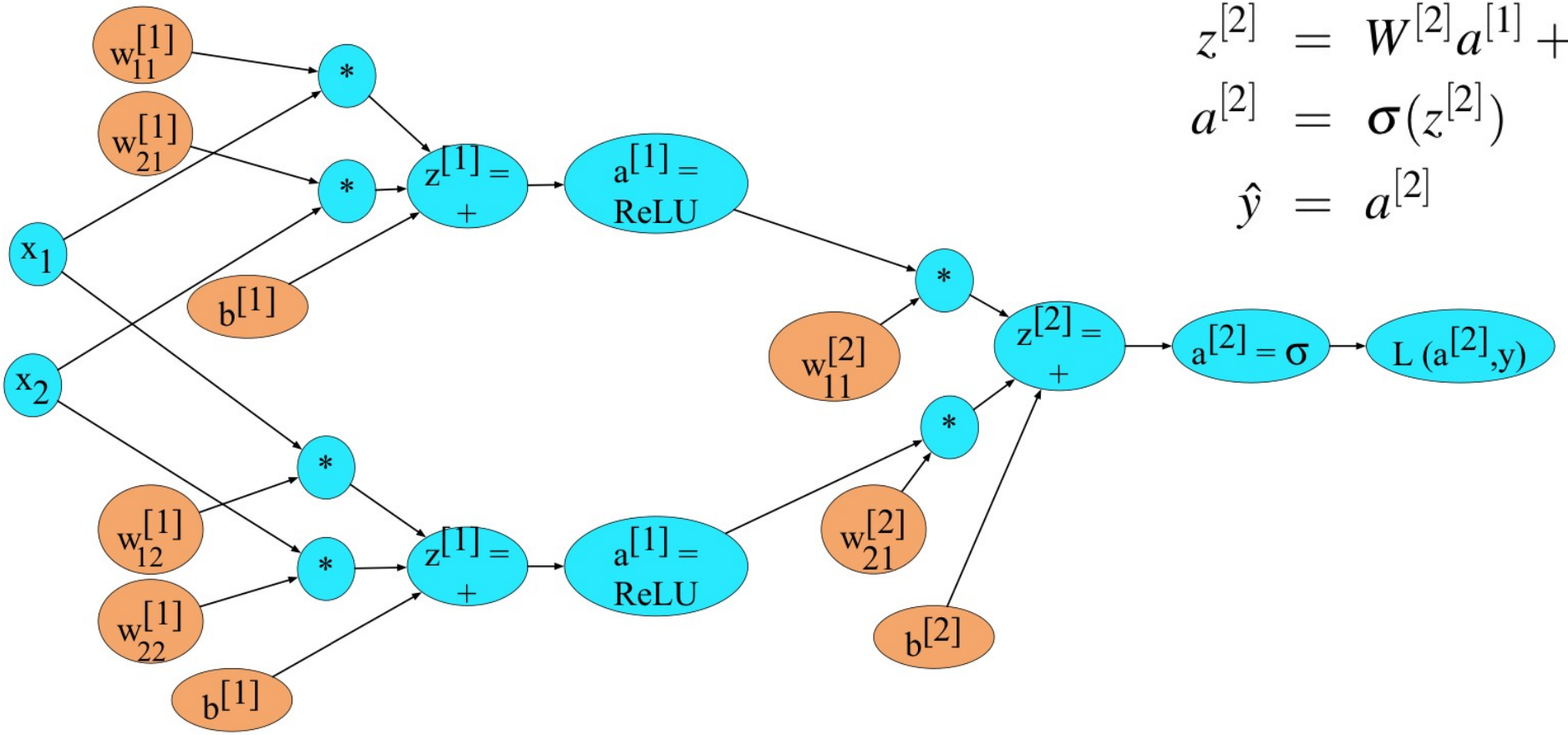
Source: https://web.stanford.edu/~jurafsky/slp3/7.pdf

Computation graphs for real neural networks are much more complex

- Computation graph for a 2-layer neural network:

$n_0 = 2,\ n_1 = 2,\ \text{and}\ n_2 = 1$

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



Source: https://web.stanford.edu/~jurafsky/slp3/7.pdf

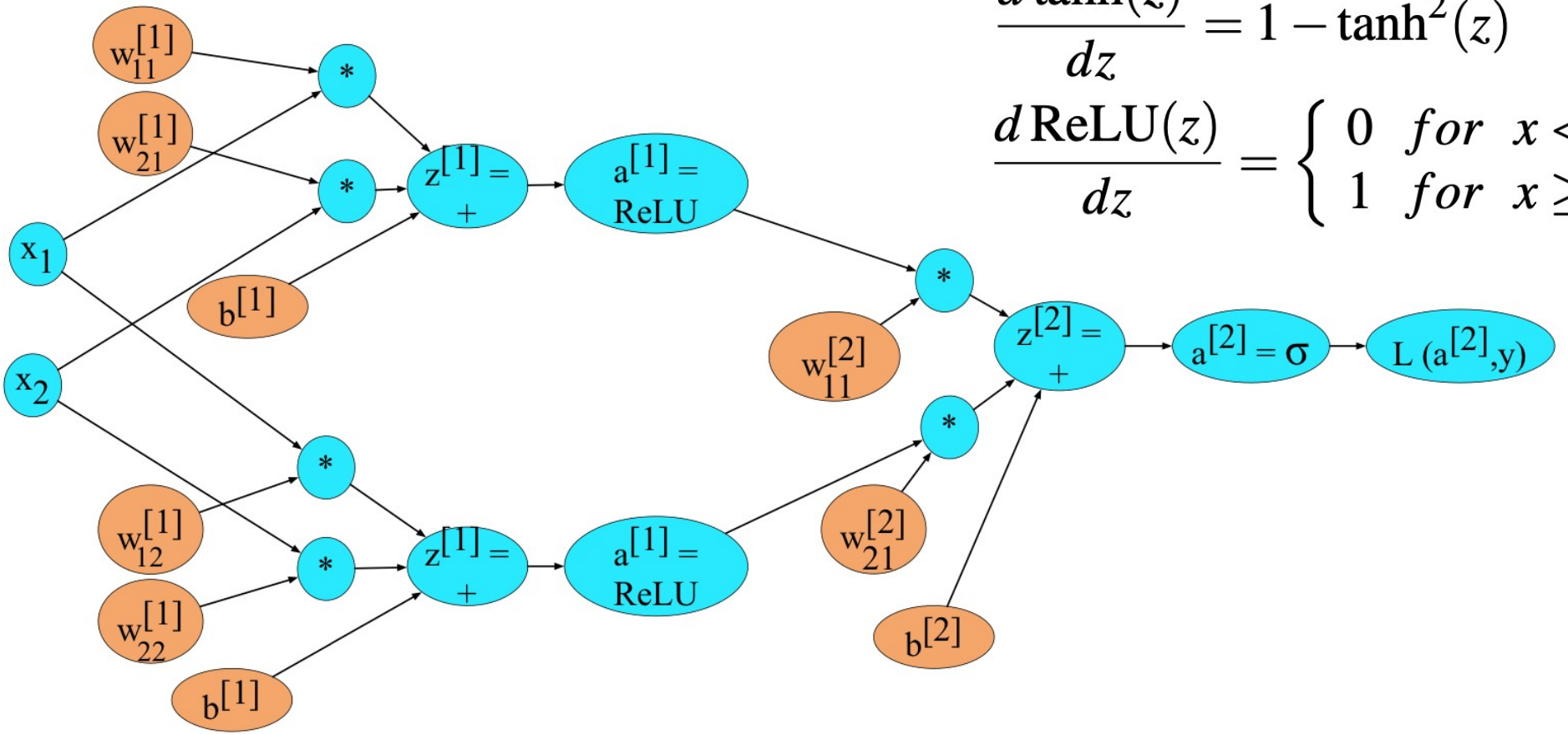
Computation graphs for real neural networks are much more complex

- Computation graph for a 2-layer neural network:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z)$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad x<0 \\ 1 & for \quad x\geq 0 \end{cases}$$



Source: https://web.stanford.edu/~jurafsky/slp3/7.pdf

Regularization

- Endeavors to reduce generalization error
- Make models that originally overfit better solely reflect the true data-generating process
- Often the models that generalize the best are large models that are properly regularized

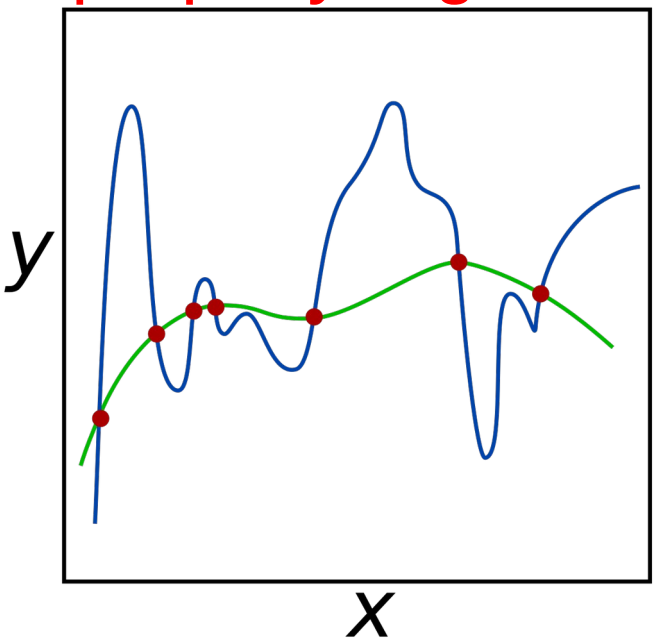


Image: https://en.wikipedia.org/wiki/Regularization_(mathematics)
Adopted from: 2nd CCIS summer school, Umass Lowel: https://github.com/text-machine-lab/ciss2_materials

Regularization

Parameter Norm Penalties

$$\hat{J}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) = J + \alpha \Omega(\boldsymbol{\theta})$$

- L2 Regularization: $\Omega(\boldsymbol{\theta}) = (\frac{1}{2})||\boldsymbol{w}||_2^2$
- L1 Regularization: $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1$
  - Enforcing a sparsity constraint on w can lead to simpler and more interpretable models

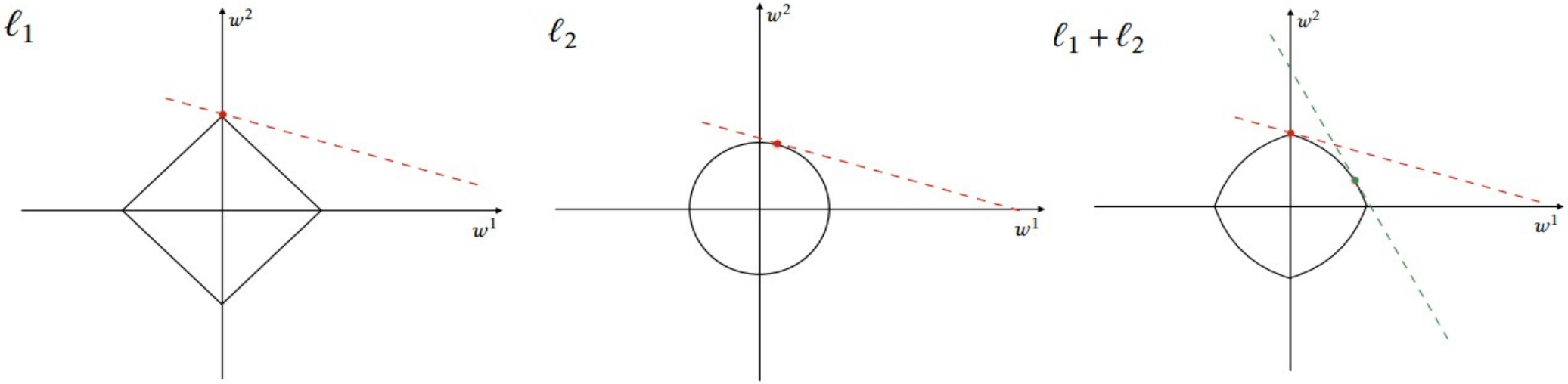


Image: https://en.wikipedia.org/wiki/Regularization_(mathematics)

Early-Stopping

- Can be considered also as a kind of regularization

Idealized training and validation error curves. Vertical: errors; Horizontal: time.

Image from: Prechelt, Lutz. "Early stopping-but when?." Neural Networks: Tricks of the trade. Springer, Berlin, Heidelberg, 1998. 55-69.

Early-Stopping

- Validation loss does not always continue to decrease with training loss
- Train model until loss does not improve on validation set
- Requires a portion of training data to be held out
  - Data can be reintegrated for training by restarting training, or continuing training with validation set added

Source: 2nd CCIS summer school, Umass Lowel: https://github.com/text-machine-lab/ciss2_materials

Early-Stopping

A real validation error curve.

- Vertical: validation set error.
- Horizontal: time (in training epochs)

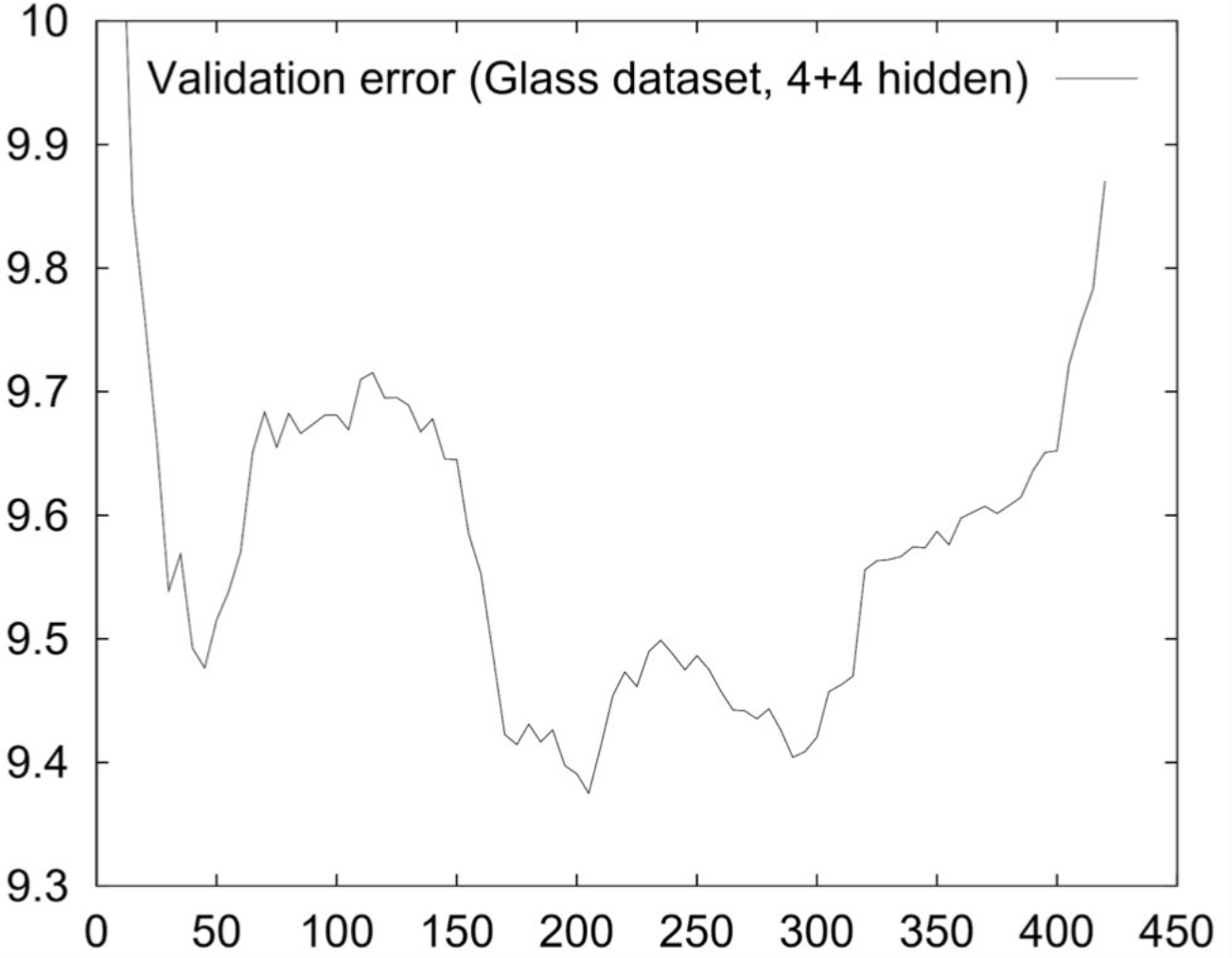


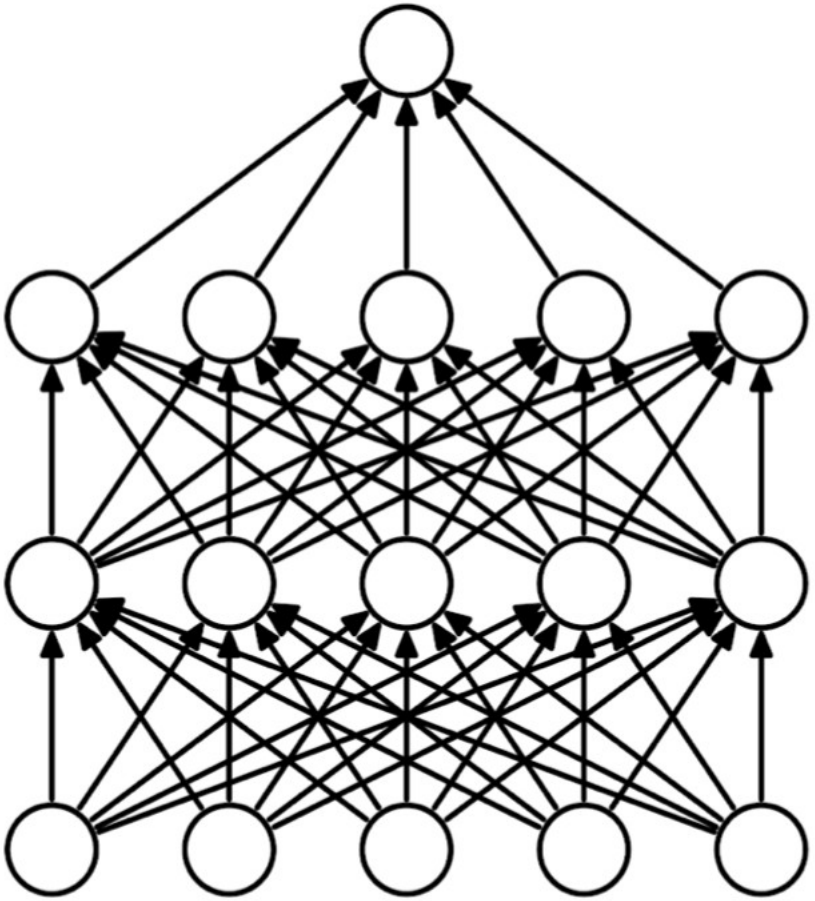
Image from: Prechelt, Lutz. "Early stopping-but when?." Neural Networks: Tricks of the trade. Springer, Berlin, Heidelberg, 1998. 55-69.

Dropout

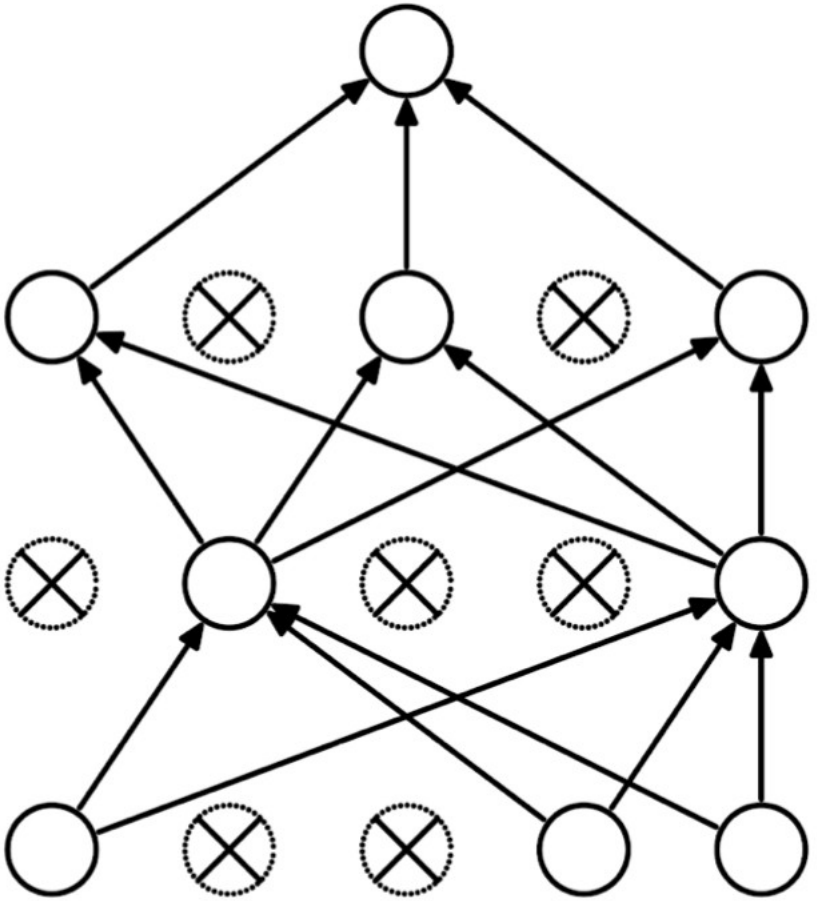
- In practice, ensembling the predictions of many pretrained models at test time often leads to better performance
- Dropout can be seen as training the approximation of all possible subsets of configurations of model parameters
- For each layer in the network, a binary mask is sampled (based on fixed probability p) to determine which neurons' output is kept
- To ensure proper predictions at inference, divide hidden states by p during training

Source: 2nd CCIS summer school, Umass Lowel: https://github.com/text-machine-lab/ciss2_materials

Dropout



(a) Standard Neural Net



(b) After applying dropout.

Image from: Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.