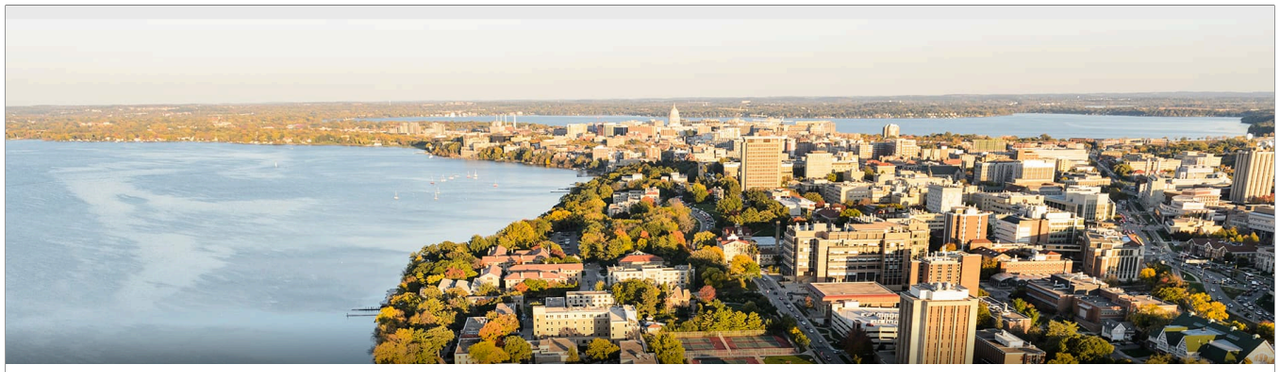


CS 540 Introduction to Artificial Intelligence

Neural Networks (III)

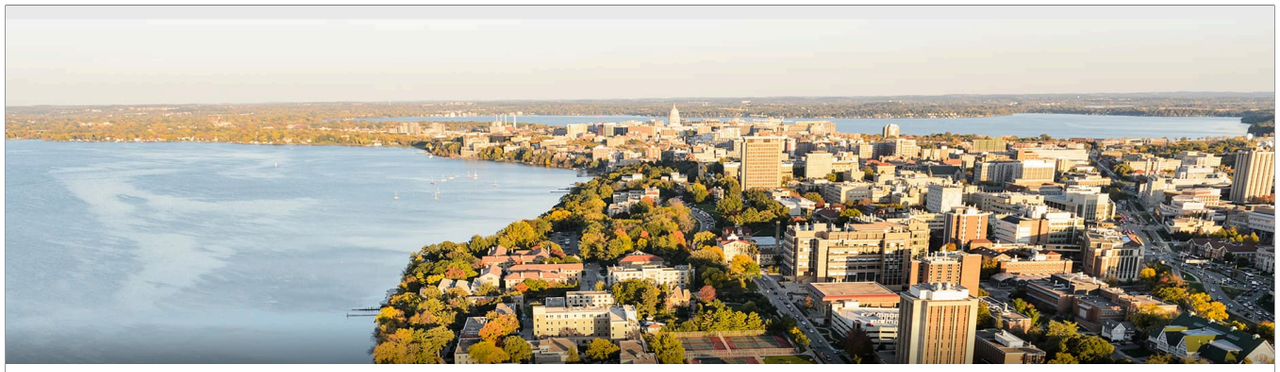
Yingyu Liang
University of Wisconsin-Madison

Oct 26, 2021

Slides created by Sharon Li [modified by Yingyu Liang]

Today's outline

- Deep neural networks
  - Computational graph (forward and backward propagation)
- Numerical stability in training
  - Gradient vanishing/exploding
- Generalization and regularization
  - Overfitting, underfitting
  - Weight decay and dropout



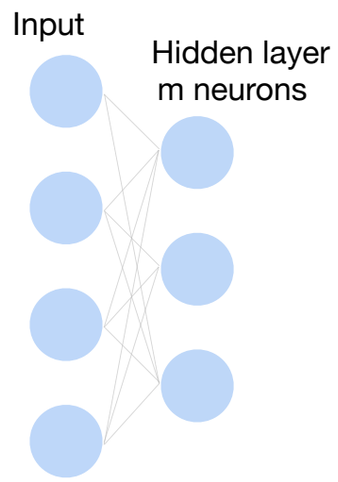
Part I: Neural Networks as a Computational Graph

Review: neural networks with one hidden layer

- Input $\mathbf{x} \in \mathbb{R}^d$
- Hidden $\mathbf{W}^{(1)} \in \mathbb{R}^{m \times d}, \mathbf{b} \in \mathbb{R}^m$
- Intermediate output

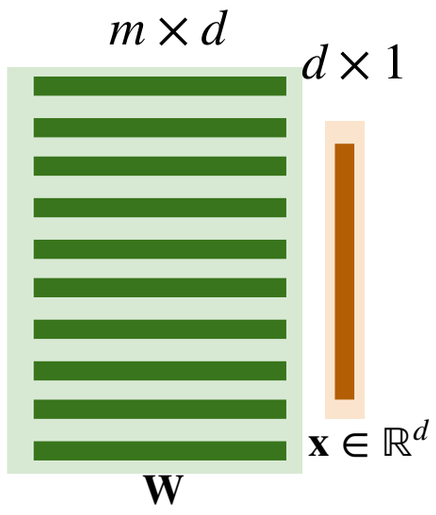
$$\mathbf{h} = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b})$$

$$\mathbf{h} \in \mathbb{R}^m$$



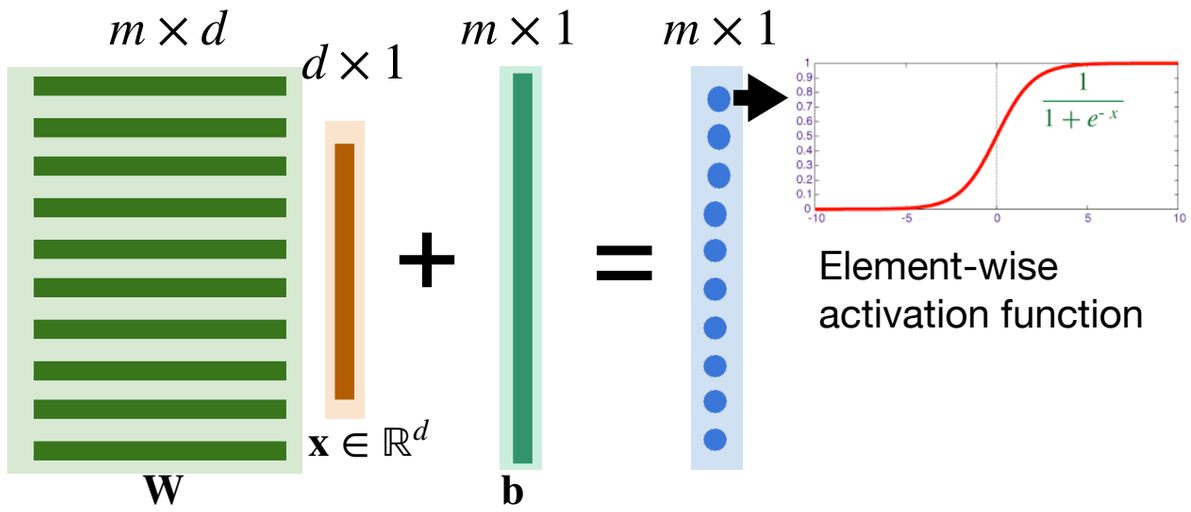
Each hidden neuron is a perceptron (but can be with a different activation function than the step function). Viewing the outputs of the hidden neurons in the layer as a vector, then the layer just first does a linear transformation on the input vector and then applies an activation function on each element of the vector obtained by the linear transformation. Note that here, we apply the activation function \sigma on the vector Wx+b; it simply means applying \sigma on each element of the vector.

Review: neural networks with one hidden layer



Review: neural networks with one hidden layer

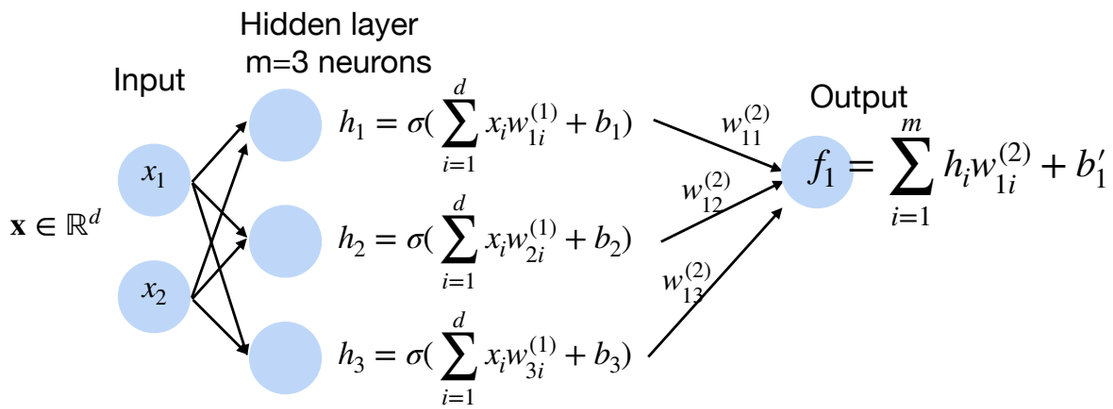
Key elements: linear operations + Nonlinear activations



This is an illustration of the operations in a layer. Note the dimensions.

Review: Neural network for k-way classification

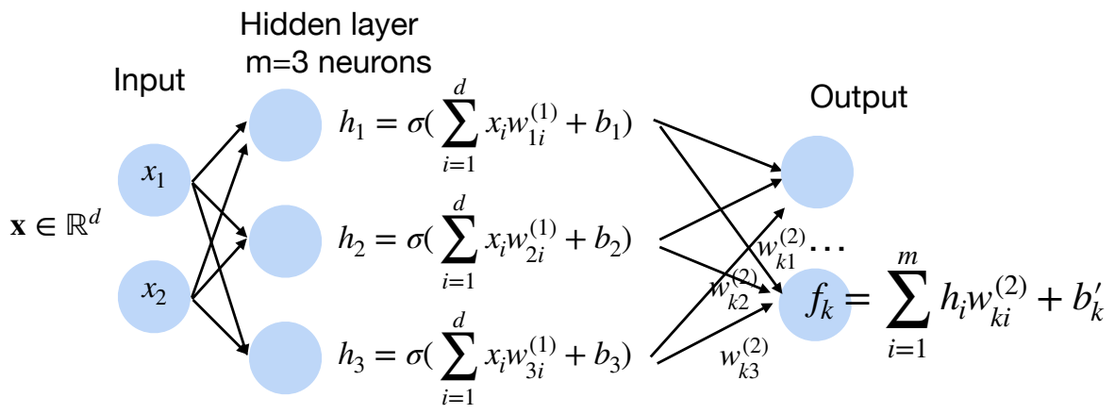
•K outputs in the final layer



Review: Neural network for k-way classification

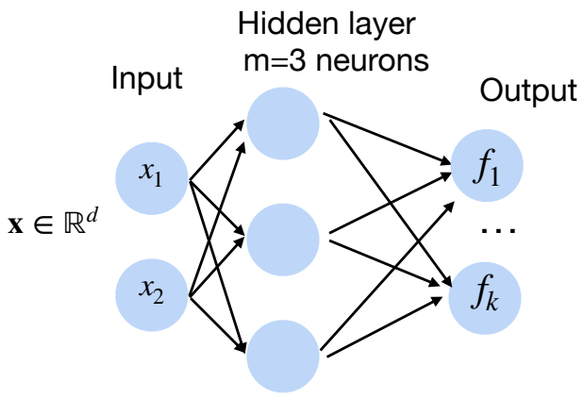
•K outputs units in the final layer

Multi-class classification (e.g., ImageNet with k=1000)



Review: Softmax

Turns outputs f into probabilities (sum up to 1 across k classes)



$$p(y \mid \mathbf{x}) = \text{softmax}(f)$$
$$= \frac{\exp f_y(x)}{\sum_i^k \exp f_i(x)}$$

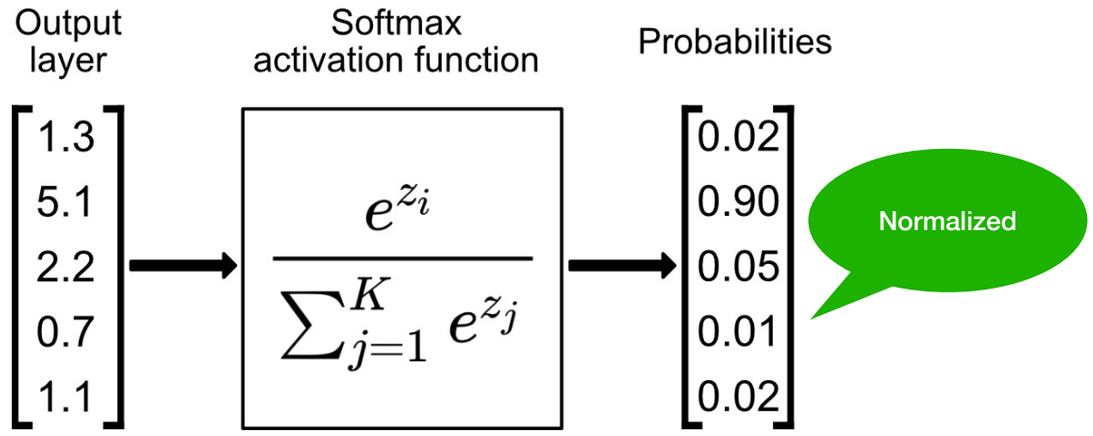
For multiple-class classification, the output layer first applies a linear transformation to get a k dim vector f=(f_1,…,f_k). Then apply the softmax activation to turn f into a probabilistic vector (a distribution) over the k classes. This is typically used to modeled the conditional probabilities of y over the k classes conditioned on the input x.

Note that

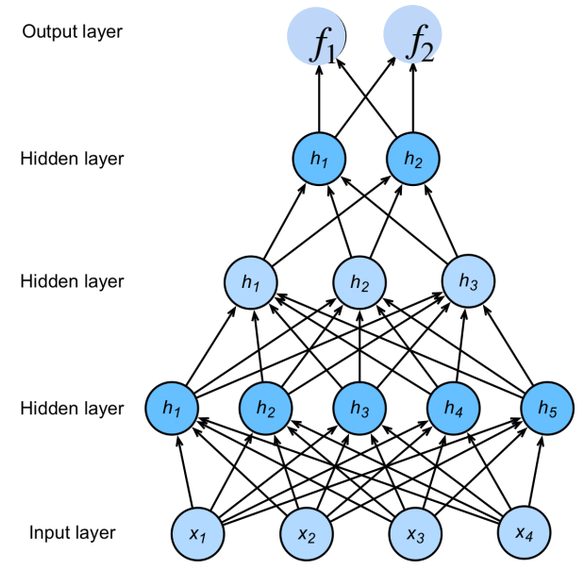
1. after softmax, the entries are nonnegative and have sum 1, ie, the vector after softmax is a probabilistic vector.
2. Larger the f_i, larger the corresponding entry after softmax.
3. Softmax can be viewed as an activation function but it’s not element-wise as each output entry depends on all entries in f.

Softmax

Turns outputs f into probabilities (sum up to 1 across k classes)



Deep neural networks (DNNs)



$$\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$
$$\mathbf{f} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4$$
$$\mathbf{y} = \text{softmax}(\mathbf{f})$$

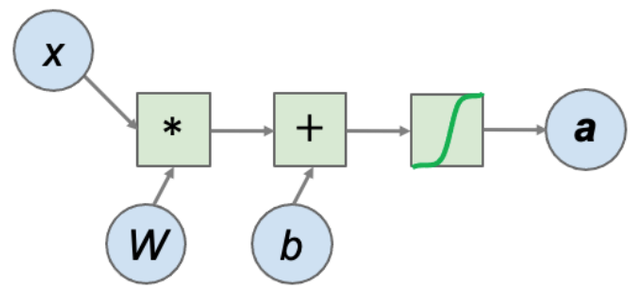
NNs are composition of nonlinear functions

A standard deep network is simply stacking multiple layers of computation; each layer is linear transformation+activation (typically nonlinear). For regression, we can simply use the linear transformation result f in the output layer as the final output of the network. For classification, we apply softmax on f to get the probabilities over k classes (or apply sigmoid for binary classification).

Neural networks as variables + operations

$$\mathbf{a} = \text{sigmoid}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- Decompose functions into atomic operations
- Separate data (variables) and computing (operations)
- Known as a **computational graph**



In general, we can draw a computational graph (introducing intermediate variables after each computational operation).

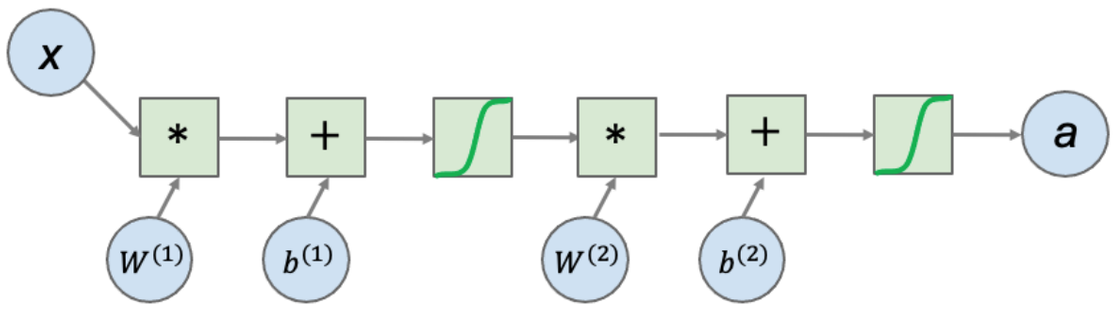
Backpropagtion is going from the output backward to the input. Each time we go backward one step and compute the derivative of the loss w.r.t. some variable; apply chain rule so that we can reuse the derivative computed in previous steps.

Note: Here the notations like \partial L /\partial W are vector/matrix notations. You can think of \partial L /\partial W as concatenating the derivative of the loss w.r.t. each parameter. Another example is \partial z_1 /\partial W: you can think of it as a matrix, where the rows correspond to dimension in z_1 and columns corresponds to parameters in W, and each entry is the derivative of the corresponding dimension in z_1 w.r.t. the corresponding parameter in W. So the chain rule now is a product of multiple matrices (it can contain vectors which can also be viewed as a special kind of matrices).

In this course we don't require to handle chain rule in this matrix form. We present it here only for
1. demonstrating how backpropagation is done in general (the matrix form can compactly represent this).
2. showing the reason for vanishing gradients: when each matrix in the chain is small (w.r.t. certain "size" measurement of the matrix) then the product result (the gradient) will be very small.

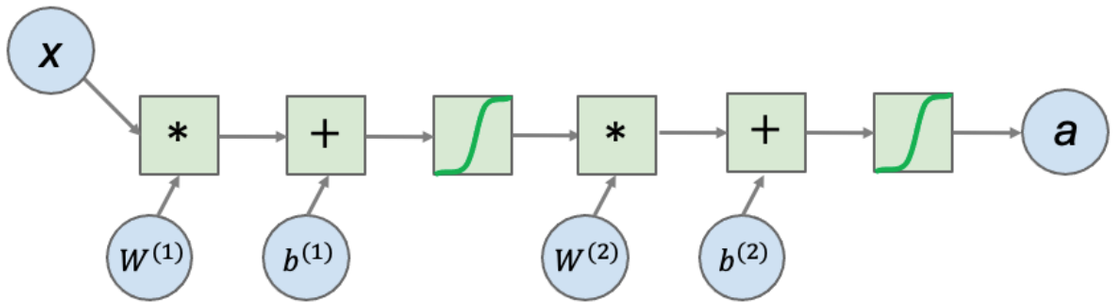
Neural networks as a computational graph

- A two-layer neural network



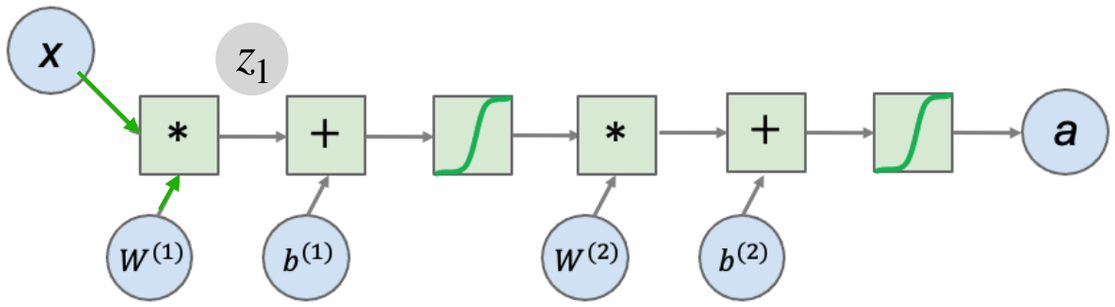
Neural networks as a computational graph

- A two-layer neural network
- Forward propagation vs. backward propagation



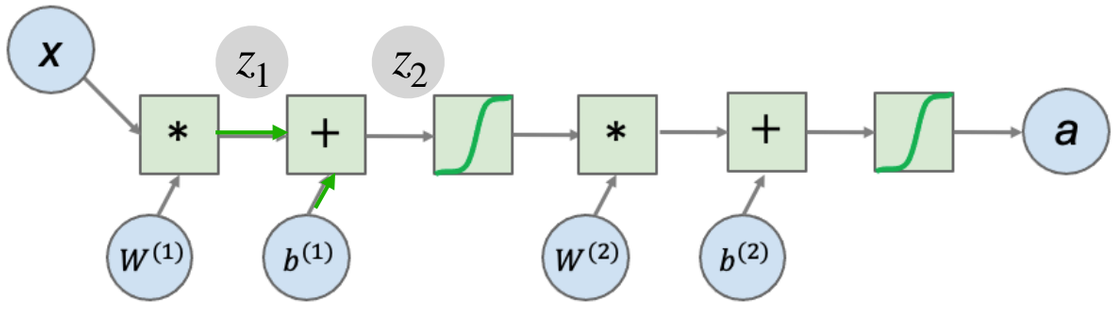
Neural networks: forward propagation

- A two-layer neural network
- Intermediate variables Z



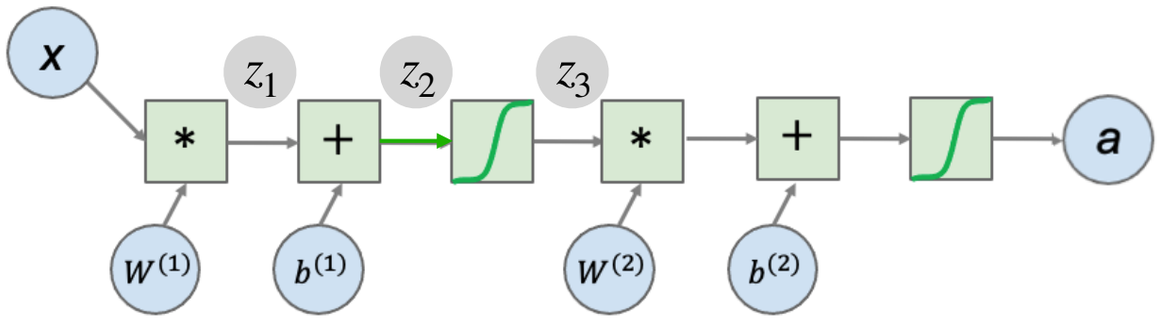
Neural networks: forward propagation

- A two-layer neural network
- Intermediate variables Z



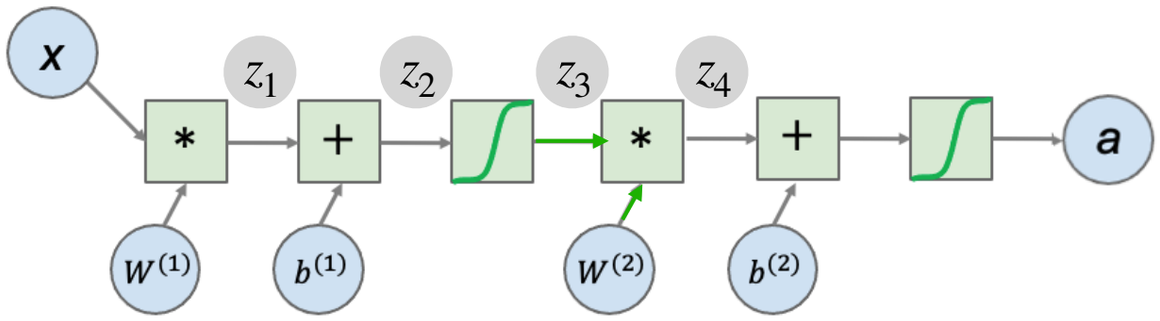
Neural networks: forward propagation

- A two-layer neural network
- Intermediate variables Z



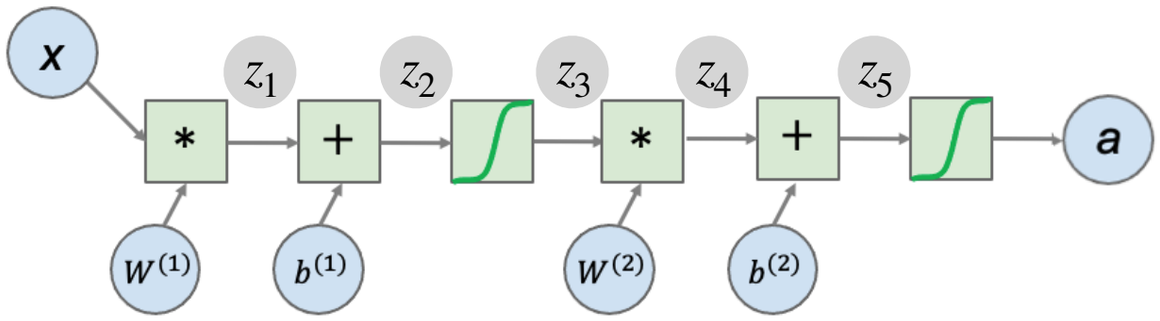
Neural networks: forward propagation

- A two-layer neural network
- Intermediate variables Z



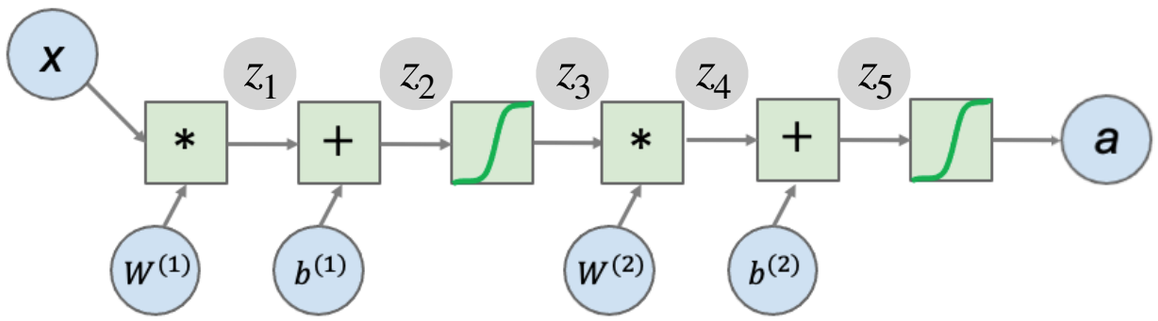
Neural networks: forward propagation

- A two-layer neural network
- Intermediate variables Z



Neural networks: backward propagation

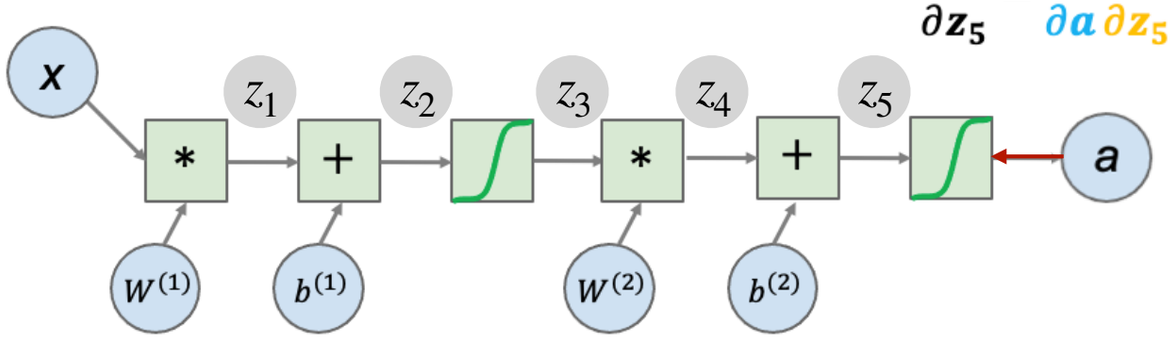
- A two-layer neural network
- Assuming forward propagation is done
- Minimize a **loss function** L



Neural networks: backward propagation

- A two-layer neural network
- Assuming forward propagation is done
- Minimize a **loss function** L

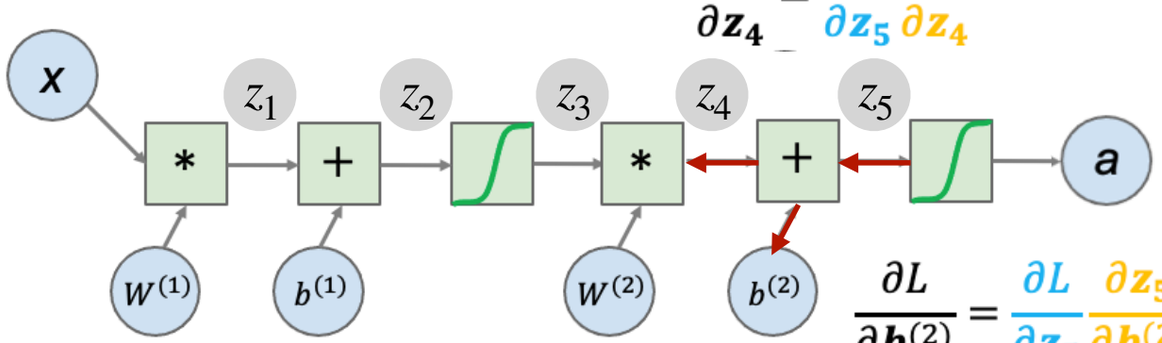
$$\frac{\partial L}{\partial \boldsymbol{z_5}} = \frac{\partial L}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z_5}}$$



Neural networks: backward propagation

- A two-layer neural network
- Assuming forward propagation is done
- Minimize a **loss function** L

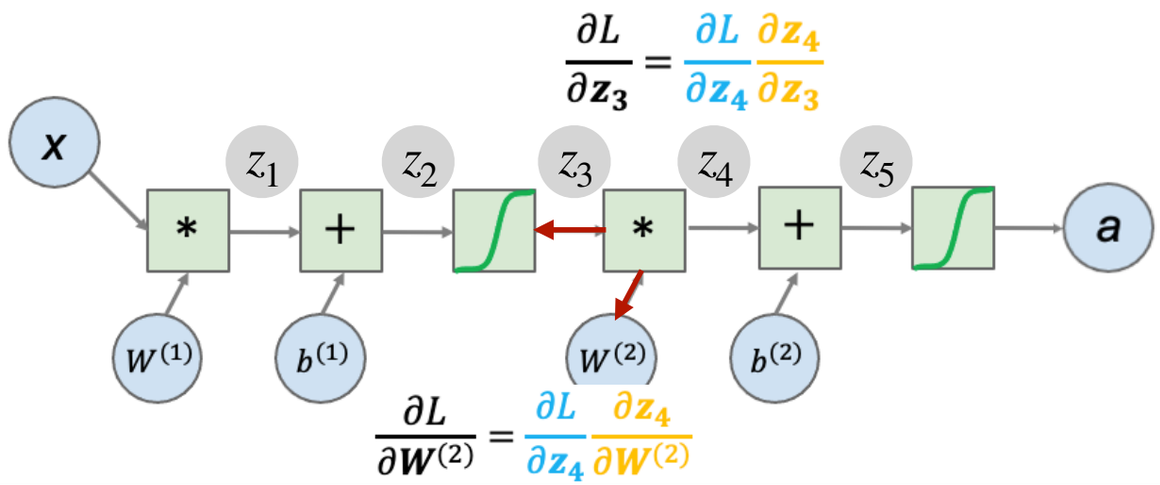
$$\frac{\partial L}{\partial \boldsymbol{z_4}} = \frac{\partial L}{\partial \boldsymbol{z_5}} \frac{\partial \boldsymbol{z_5}}{\partial \boldsymbol{z_4}}$$



$$\frac{\partial L}{\partial \boldsymbol{b}^{(2)}} = \frac{\partial L}{\partial \boldsymbol{z_5}} \frac{\partial \boldsymbol{z_5}}{\partial \boldsymbol{b}^{(2)}}$$

Neural networks: backward propagation

- A two-layer neural network
- Assuming forward propagation is done



Backward propagation: A modern treatment

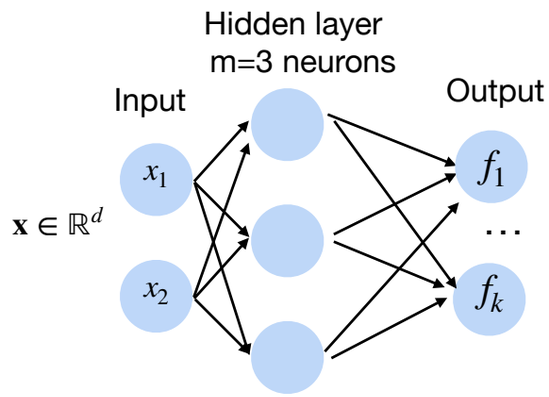
- Define a neural network as a computational graph
- Must be a directed graph
- Nodes as variables and operations
- All operations must be **differentiable**

Q1. Suppose we want to solve the following k-class classification problem with cross entropy loss $\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{k} y_j \log \hat{y}_j$, where the ground truth and predicted probabilities $\mathbf{y}, \hat{\mathbf{y}} \in \mathbb{R}^k$. Recall that the softmax function turns output into probabilities: $\hat{y}_j = \frac{\exp f_j(x)}{\sum_i^k \exp f_i(x)}$. What is the partial derivative $\partial_{f_j} \ell(\mathbf{y}, \hat{\mathbf{y}})$?

A. $\hat{y}_j - y_j$

B. $\exp(y_j) - y_j$

C. $y_j - \hat{y}_j$

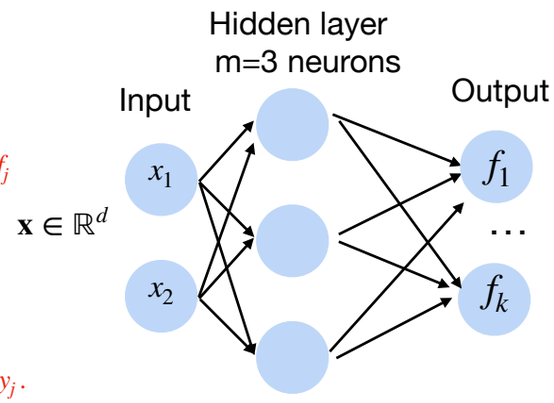


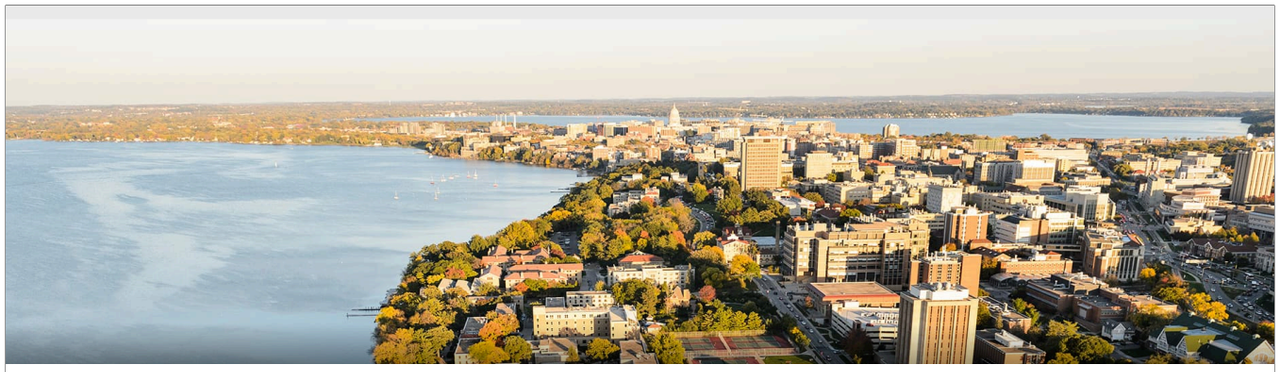
- For notational simplicity, we use y_i to denote $\mathbf{1}\{y_i = 1\}$, and $\hat{y}_i$ as $p(y_i = 1 \mid \mathbf{x}; \theta)$

Q1. Suppose we want to solve the following k-class classification problem with cross entropy loss $\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{k} y_j \log \hat{y}_j$, where $\mathbf{y}, \hat{\mathbf{y}} \in \mathbb{R}^k$. Recall that the softmax function turns output into probabilities: $\hat{y}_j = \frac{\exp f_j(x)}{\sum_i^k \exp f_i(x)}$. What is the partial derivative $\partial_{f_j} \ell(\mathbf{y}, \hat{\mathbf{y}})$?

Rewrite
$$\begin{aligned}\ell(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{k} y_j \log \frac{\exp(f_j)}{\sum_{i=1}^{k} \exp(f_i)} \\ &= \sum_{j=1}^{k} y_j \log \sum_{i=1}^{k} \exp(f_i) - \sum_{j=1}^{k} y_j f_j \\ &= \log \sum_{i=1}^{k} \exp(f_i) - \sum_{j=1}^{k} y_j f_j .\end{aligned}$$

We have
$$\partial_{f_j} \ell(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(f_j)}{\sum_{i=1}^{k} \exp(f_k)} - y_j = \hat{y}_j - y_j .$$





Part II: Numerical Stability

Gradients for Neural Networks

- Compute the gradient of the loss ℓ w.r.t. $\mathbf{W}_t$

$$\frac{\partial \ell}{\partial \mathbf{W}^t} = \frac{\partial \ell}{\partial \mathbf{h}^d} \underbrace{\frac{\partial \mathbf{h}^d}{\partial \mathbf{h}^{d-1}} \cdots \frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^t}} \frac{\partial \mathbf{h}^t}{\partial \mathbf{W}^t}$$

Multiplication of *many* matrices

Wikipedia

Two Issues for Deep Neural Networks

$$\prod_{i=t}^{d-1} \frac{\partial \mathbf{h}^{i+1}}{\partial \mathbf{h}^{i}}$$

Gradient Exploding

$1.5^{100} \approx 4 \times 10^{17}$

Gradient Vanishing

$0.8^{100} \approx 2 \times 10^{-10}$

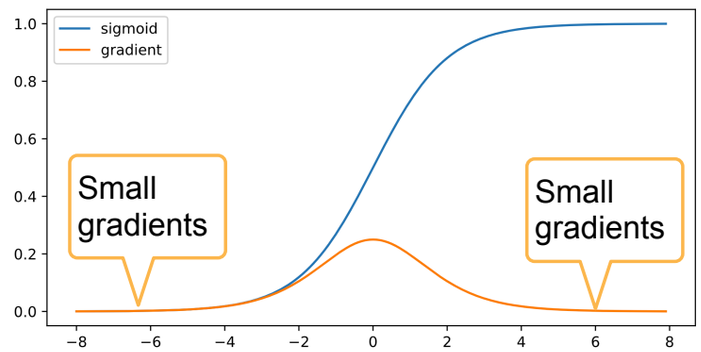
Issues with Gradient Exploding

- Value out of range: infinity value (NaN)
- Sensitive to learning rate (LR)
  - Not small enough LR -> larger gradients
  - Too small LR -> No progress
  - May need to change LR dramatically during training

Gradient Vanishing

- Use sigmoid as the activation function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \sigma'(x) = \sigma(x)(1 - \sigma(x))$$



Issues with Gradient Vanishing

- Gradients with value 0
- No progress in training
  - No matter how to choose learning rate
- Severe with bottom layers
  - Only top layers are well trained
  - No benefit to make networks deeper

How to stabilize training?

Stabilize Training: Practical Considerations

- Goal: make sure gradient values are in a proper range
  - E.g. in [1e-6, 1e3]
- Multiplication -> plus
  - Architecture change (e.g., ResNet)
- Normalize
  - Batch Normalization, Gradient clipping
- Proper activation functions

Q2. Let’s compare sigmoid with rectified linear unit (ReLU). Which of the following statement is NOT true?

A. Sigmoid function is more expensive to compute

B. ReLU has non-zero gradient everywhere

C. The gradient of Sigmoid is always less than 0.3

D. The gradient of ReLU is constant for positive input

Q2. Let's compare sigmoid with rectified linear unit (ReLU). Which of the following statement is NOT true?

A. Sigmoid function is more expensive to compute

B. ReLU has non-zero gradient everywhere

C. The gradient of Sigmoid is always less than 0.3

D. The gradient of ReLU is constant for positive input

Q3. A Leaky ReLU is defined as *f(x)=max(0.1x, x).* Let f'(0)=1. Does it have non-zero gradient everywhere??

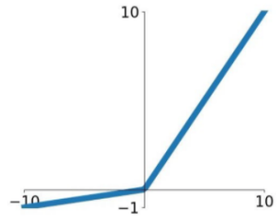
A. Yes

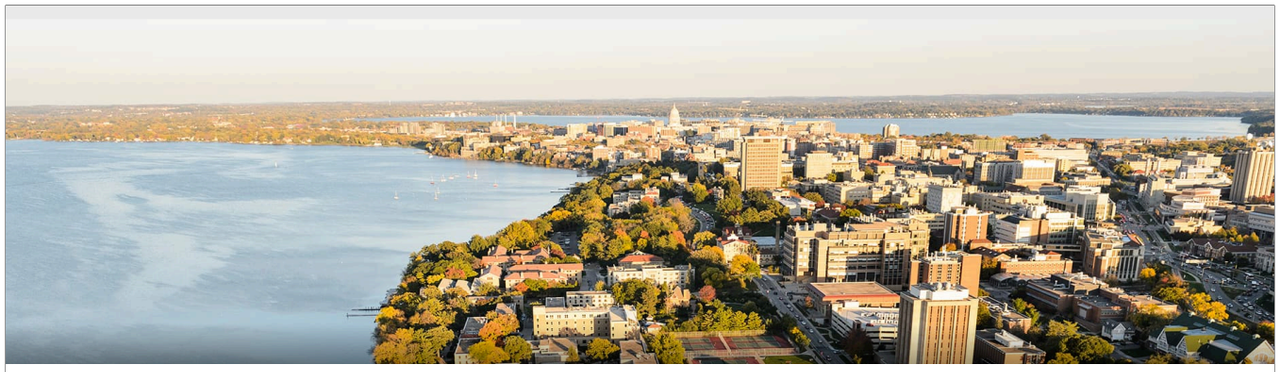
B. No

Q3. A Leaky ReLU is defined as *f(x)=max(0.1x, x).* Let f'(0)=1. Does it have non-zero gradient everywhere??

A. Yes

B. No





Part III: Generalization & Regularization

How good are the models?

Training Error and Generalization Error

- Training error: model error on the training data
- **Generalization error**: model error on new data
- Example: practice a future exam with past exams
  - Doing well on past exams (training error) doesn't guarantee a good score on the future exam (generalization error)

What we care about ultimately is generalization error, while what we know at training time is the training error/loss.

Underfitting
Overfitting

Image credit: hackernoon.com

Model Capacity

- The ability to fit variety of functions
- Low capacity models struggles to fit training set
  - Underfitting
- High capacity models can memorize the training set
  - Overfitting

Intuitively: if we use a too large family of functions to choose model from (high capacity models), then it's easy to find a model in that family to fit the data, including fitting the noise/spurious correlation. This leads to overfitting.

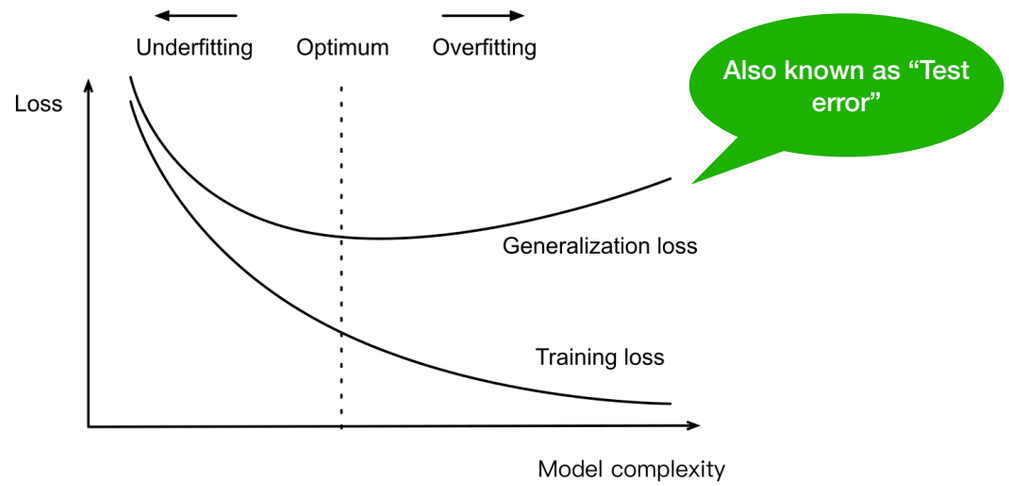
If we use a too small family of functions to choose model from (low capacity models), then there may to be a model in that family to fit the data. This leads to underfitting.

We should choose the proper capacity.

Underfitting and Overfitting

Model capacity	Data complexity: Simple	Data complexity: Complex
Low	Normal	Underfitting
High	Overfitting	Normal

Influence of Model Complexity



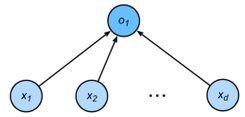
Estimate Neural Network Capacity

- It’s hard to compare complexity between different algorithms
  - e.g. tree vs neural network

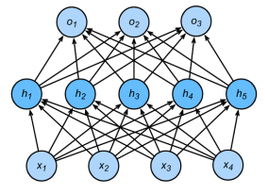
Estimate Neural Network Capacity

- It's hard to compare complexity between different algorithms
  - e.g. tree vs neural network
- Given an algorithm family, two main factors matter:
  - The number of parameters
  - The values taken by each parameter

$d+1$



$(d+1)m+(m+1)k$

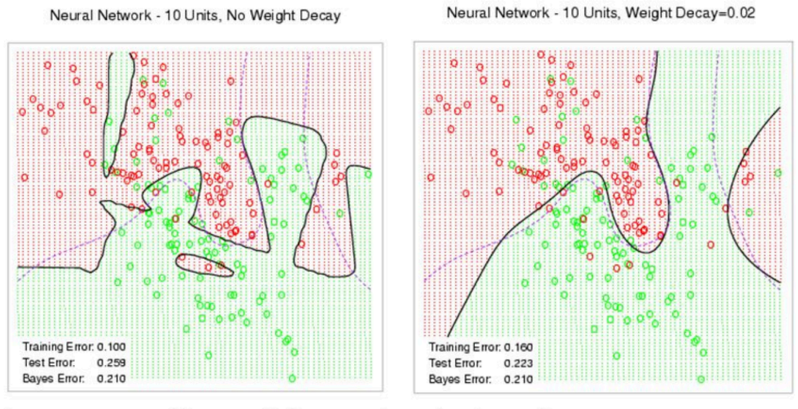


Data Complexity

- Multiple factors matters
  - # of examples
  - # of features in each example
  - time/space structure
  - # of labels

How to regularize the model for better generalization?

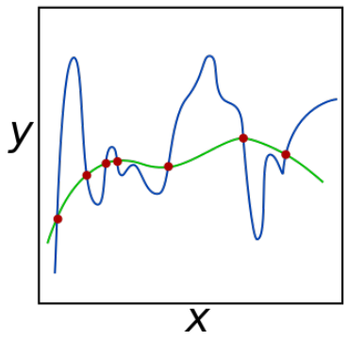
Weight Decay



Squared Norm Regularization as Hard Constraint

- Reduce model complexity by limiting value range

$$\min \ \ell(\mathbf{w}, b) \quad \text{subject to} \quad \|\mathbf{w}\|^2 \leq \theta$$



  - Often do not regularize bias *b*
    - Doing or not doing has little difference in practice
  - A small θ means more regularization

Squared Norm Regularization as Soft Constraint

- We can rewrite the hard constraint version as

$$\min \ \ell(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

L2 or squared norm regularization is also called weight decay. This is because the gradient of the squared norm is 2w, so in gradient descent we will subtract \lambda*step_size *w, ie, shrink the w.

Squared Norm Regularization as Soft Constraint

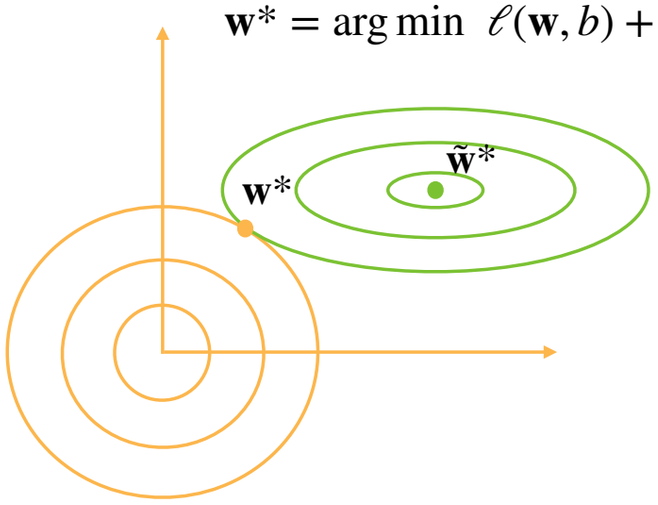
- We can rewrite the hard constraint version as

$$\min \ \ell(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

  - Hyper-parameter λ controls regularization importance
  - $\lambda = 0$: no effect
  - $\lambda \to \infty, \mathbf{w}^* \to \mathbf{0}$

Illustrate the Effect on Optimal Solutions

$$\mathbf{w}^* = \arg\min \ \ell(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$



$$\tilde{\mathbf{w}}^* = \arg\min \ \ell(\mathbf{w}, b)$$

The green curve: the level set of the loss, ie, the w parameters on the same curve will have the same loss
The yellow curve: the level set of the squared norm

Consider the green curve on which the new optimizer w* after regularization lies, ie, the level set with l(w*,b)
Claim: w* will be on the touching point with some yellow curve.
Proof: If not, it will have a larger squared norm than the touch point. Contradictory to the definition of the optimizer w*.

With this claim we can see that compared to the old optimizer \tilde{w}* before regularization, w* will move towards the original: the norm of w* shrinks.

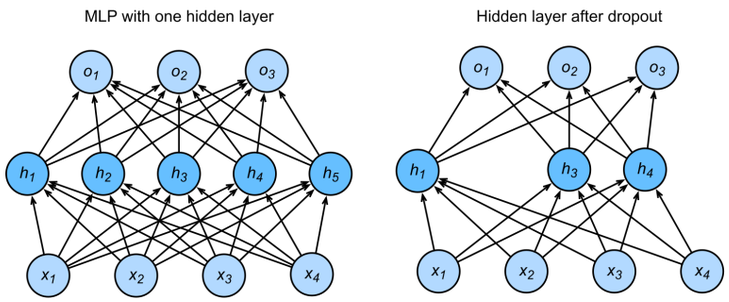
Dropout

Hinton et al.

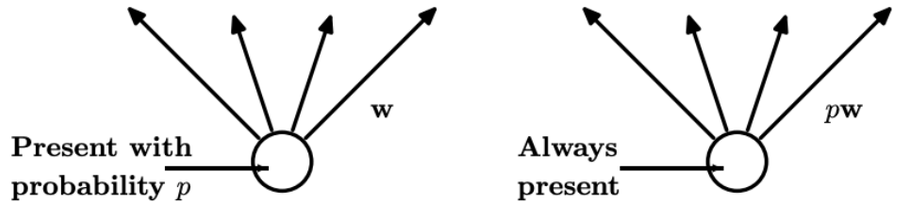
Apply Dropout

- Often apply dropout on the output of hidden fully-connected layers

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}' = \text{dropout}(\mathbf{h})$$
$$\mathbf{o} = \mathbf{W}_2\mathbf{h}' + \mathbf{b}_2$$
$$\mathbf{y} = \text{softmax}(o)$$



Dropout



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

p: non-dropout probability
At training time, randomly and independently drop the neurons with probability 1-p.
At test time, always keep the neurons but scale the weight by p

Dropout

Hinton et al.

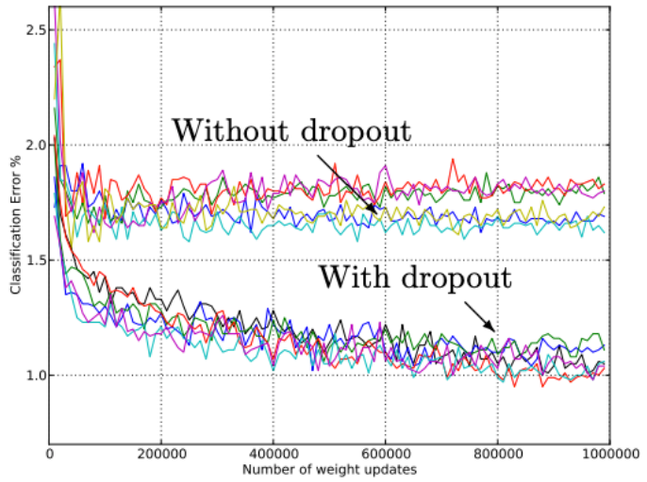


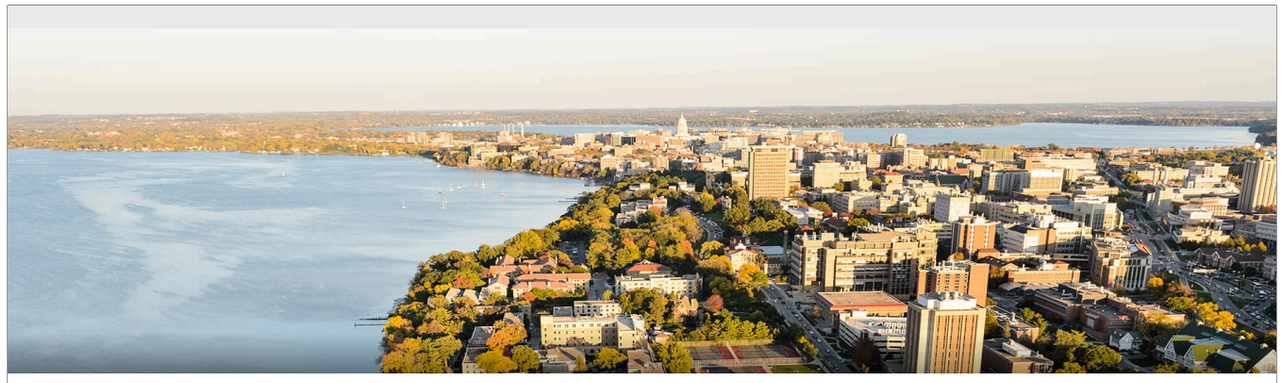
Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

It leads to much improved performance.

In recent days, we use batch normalization more, instead of dropout.

What we've learned today…

- Deep neural networks
  - Computational graph (forward and backward propagation)
- Numerical stability in training
  - Gradient vanishing/exploding
- Generalization and regularization
  - Overfitting, underfitting
  - Weight decay and dropout



Thanks!