Natural Language Processing

Lecture 6: *Introduction to Neural Nets (continued)*

Dan Goldwasser

The 10000 feet view

- **Text classification**: useful formulation of NLP tasks
  - Break text interpretation into a set of categories
- **Learning** : optimize an objective function over the training data.
- **Linear** classification vs. **Non-Linear** classification.
- So far, we've talked about linear models (LR, NB, etc.)
  - Simple (=convex) but require feature engineering effort (**why**?)
- **Questions for this lecture:**
  - Do we need to move beyond linear classifiers?
  - How can we learn representation from data, rather manual engineer them?

History

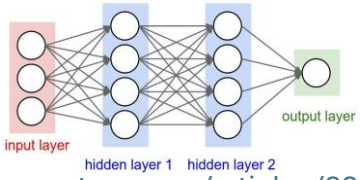
In ML, there were multiple waves of interest in neural nets

- **1940s-1960s,** mathematical models and physical implementations of single neurons (before computers were widely available)

- **1969** critical article by Minsky & Papert halted most work on neural nets. People focused on statistical and logical approaches for a while instead

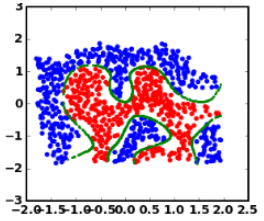
$$a \leftarrow b \wedge c$$
$$b \leftarrow d \wedge e$$
$$b \leftarrow e \wedge \neg f$$
$$c \leftarrow f \wedge g$$

- **1985** A technique for training multilayer neural networks (backpropagation) gave new life to neural networks, and people were again excited about them for a while



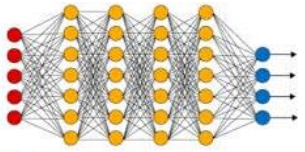
https://www.nature.com/articles/323533a0

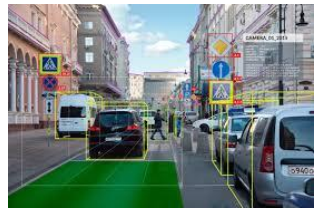
- **1998** Support Vector Machines / kernel methods arrived, seemed a much simpler, better understood solution than neural networks. (airplane wings vs bird wings debate)

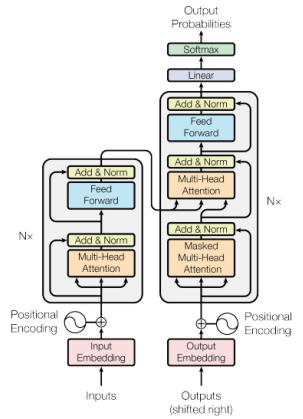


History

- **2010s-2020** New benefits observed for using "deep" neural networks (networks with many layers), leading a renewed wave of interest in neural networks.
- **2021-present** Based on the transformer architecture a new wave of models is popularized: LLM. "General purpose" AI based on massive pre-training efforts.



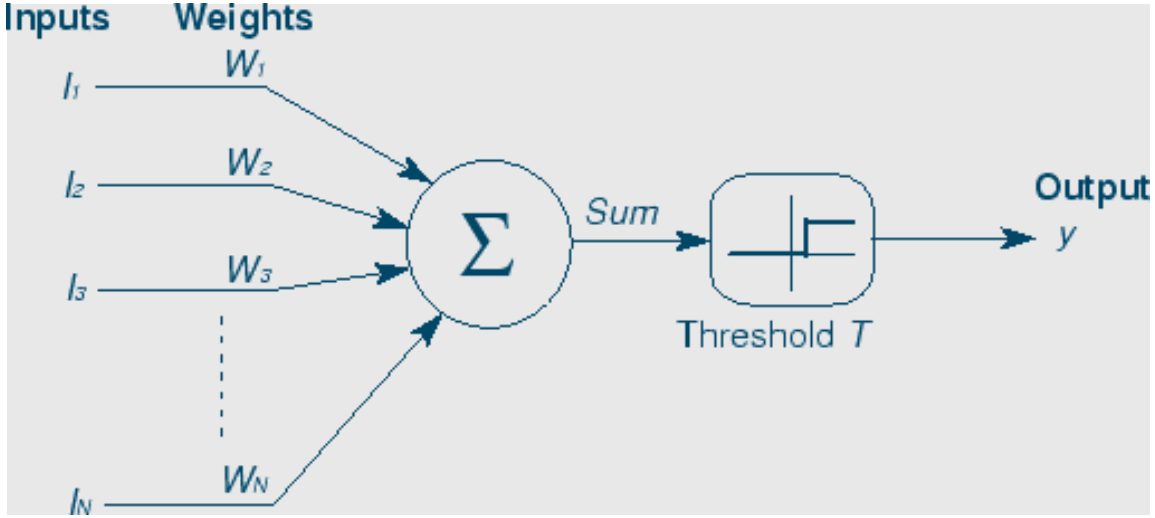


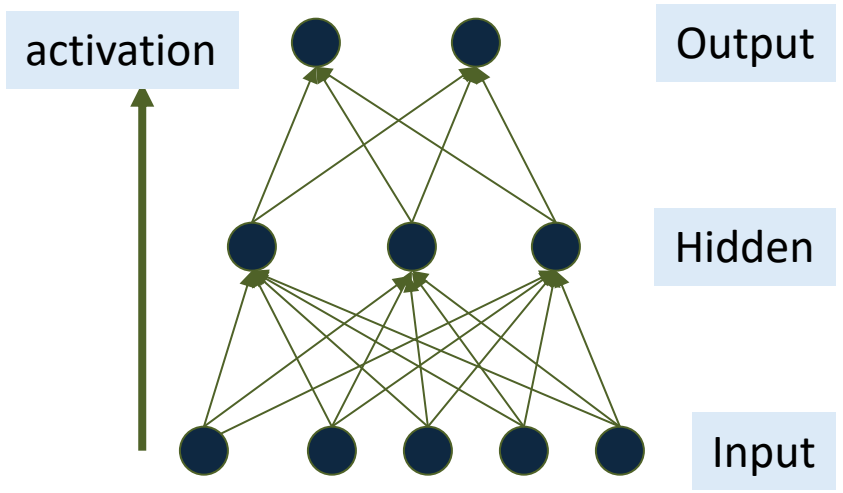


- Each wave sparked a lot of hope that AI is solved, and was followed by "AI winter"
- Current wave: massive pre-training rather than task-specific training.

Basic Units in Multi-Layer NN

- Basic element: **linear unit**
  - But, we would like to represent nonlinear functions
    - Multiple layers of linear functions are still linear functions
  - Threshold units are not smooth (we would like to use gradient-based algorithms)

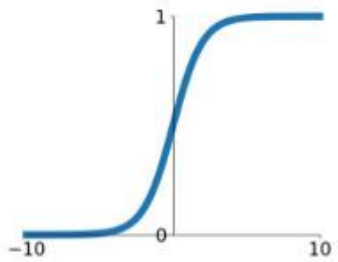




Activation Functions

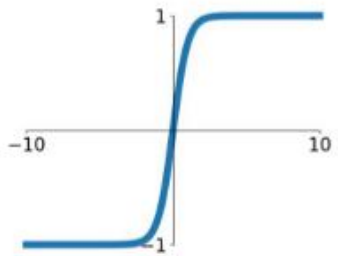
Sigmoid

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



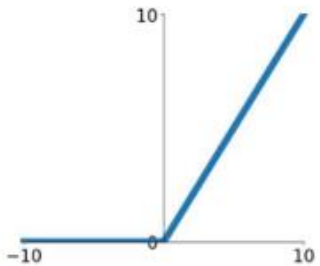
tanh

$$\tanh(x)$$



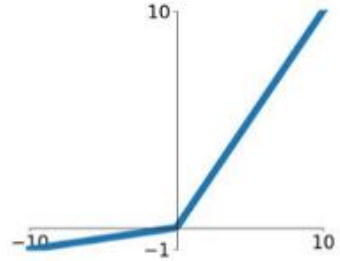
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

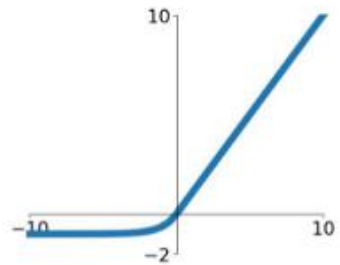


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Multi Layer NN: forward computation

Push x through the network:

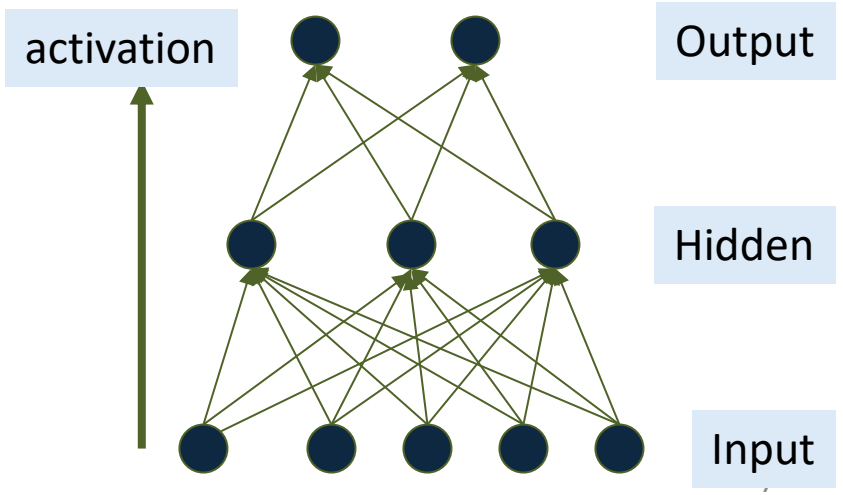
- compute the activation value of **hidden units**

$$h_j = \sigma(t_j) = \sigma(\sum_i w_{ji}^1 \, x_i)$$

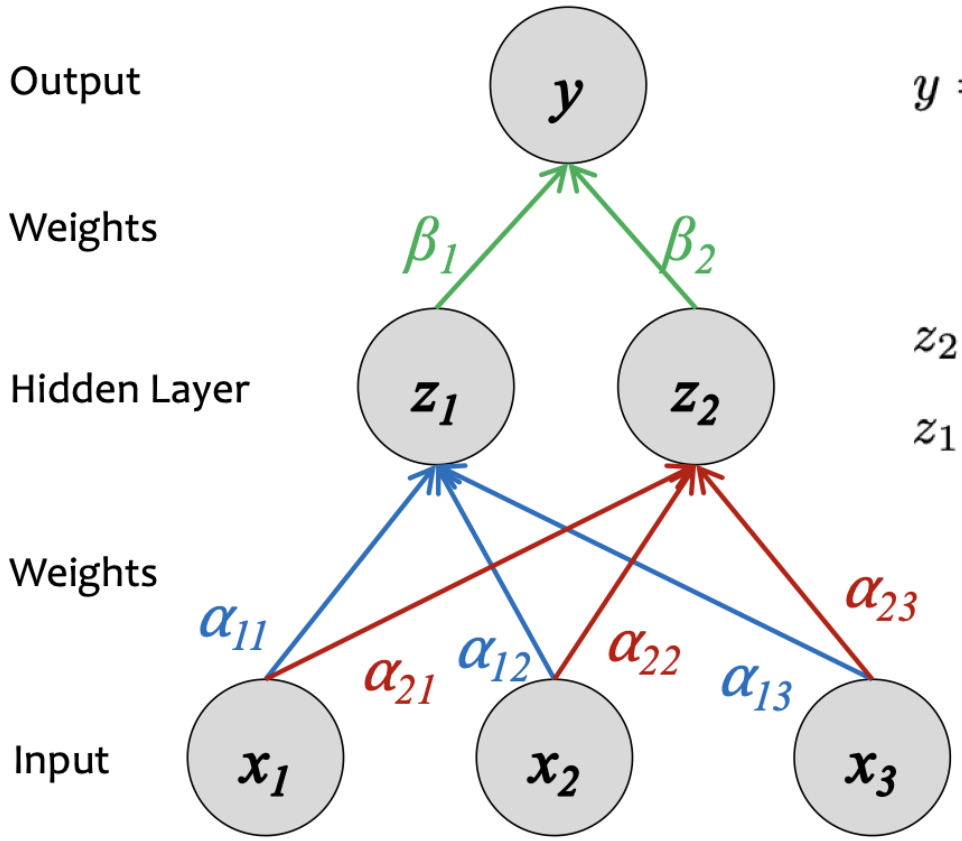
- For each **output value**, compute the activation value coming from the hidden units

$$\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w_{kj}^2 h_j)$$

- **Prediction**:
  - **Categories**: winner take all
  - **Vector**: take all output values
  - **Binary outputs**: Round to nearest 0-1 value



Forward Computation

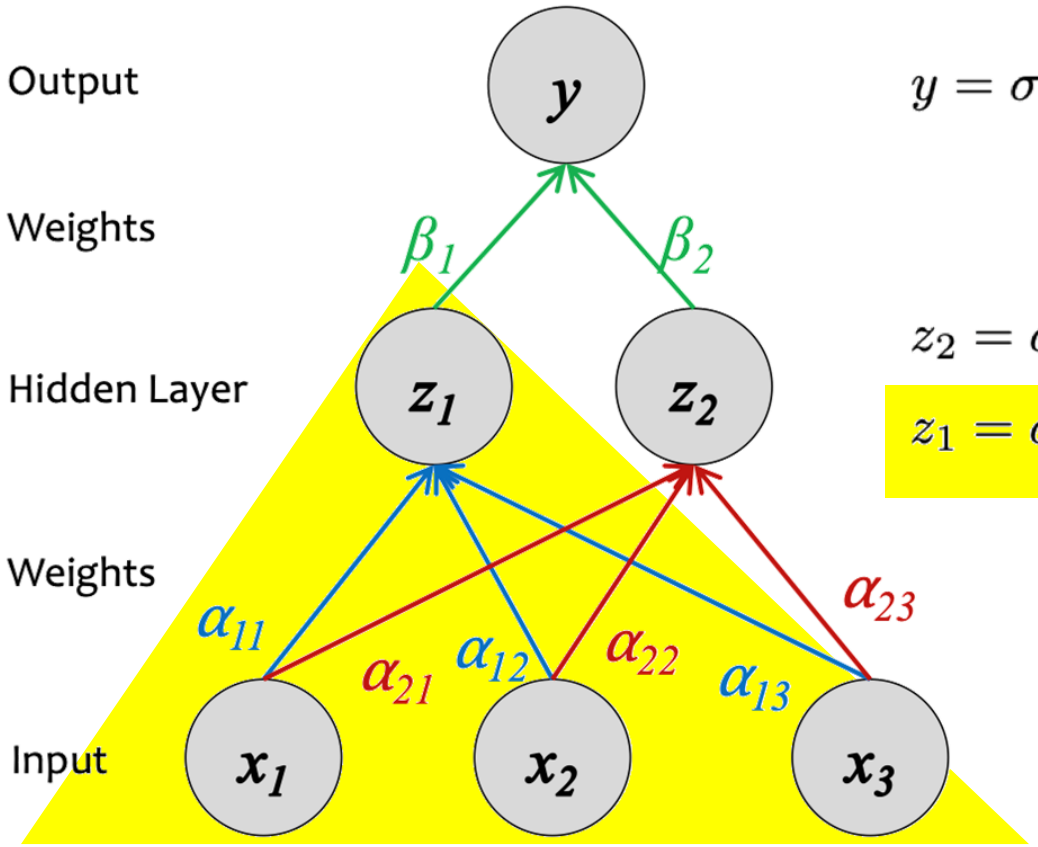


$$y = \sigma(\beta_1 z_1 + \beta_2 z_2)$$

$$z_2 = \sigma(\alpha_{21} x_1 + \alpha_{22} x_2 + \alpha_{23} x_3)$$

$$z_1 = \sigma(\alpha_{11} x_1 + \alpha_{12} x_2 + \alpha_{13} x_3)$$

Forward Computation

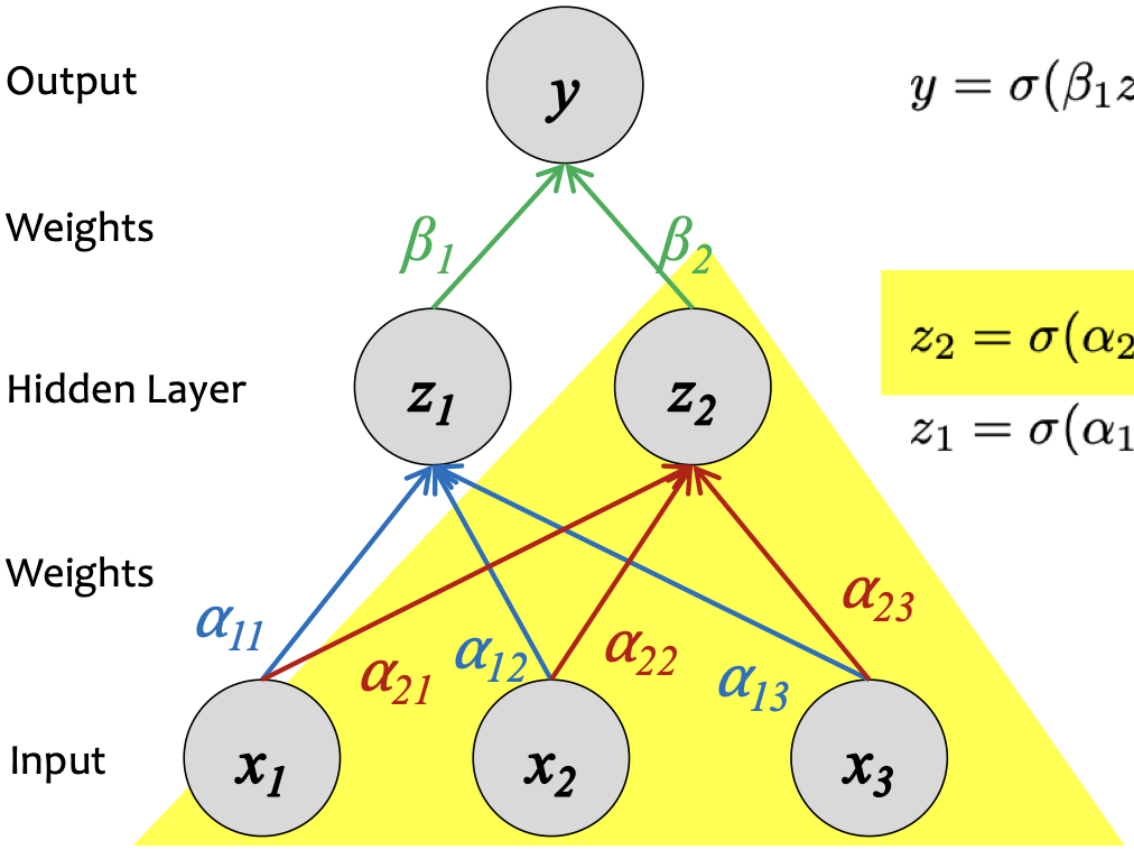


$$y = \sigma(\beta_1 z_1 + \beta_2 z_2)$$

$$z_2 = \sigma(\alpha_{21} x_1 + \alpha_{22} x_2 + \alpha_{23} x_3)$$

$$z_1 = \sigma(\alpha_{11} x_1 + \alpha_{12} x_2 + \alpha_{13} x_3)$$

Forward Computation

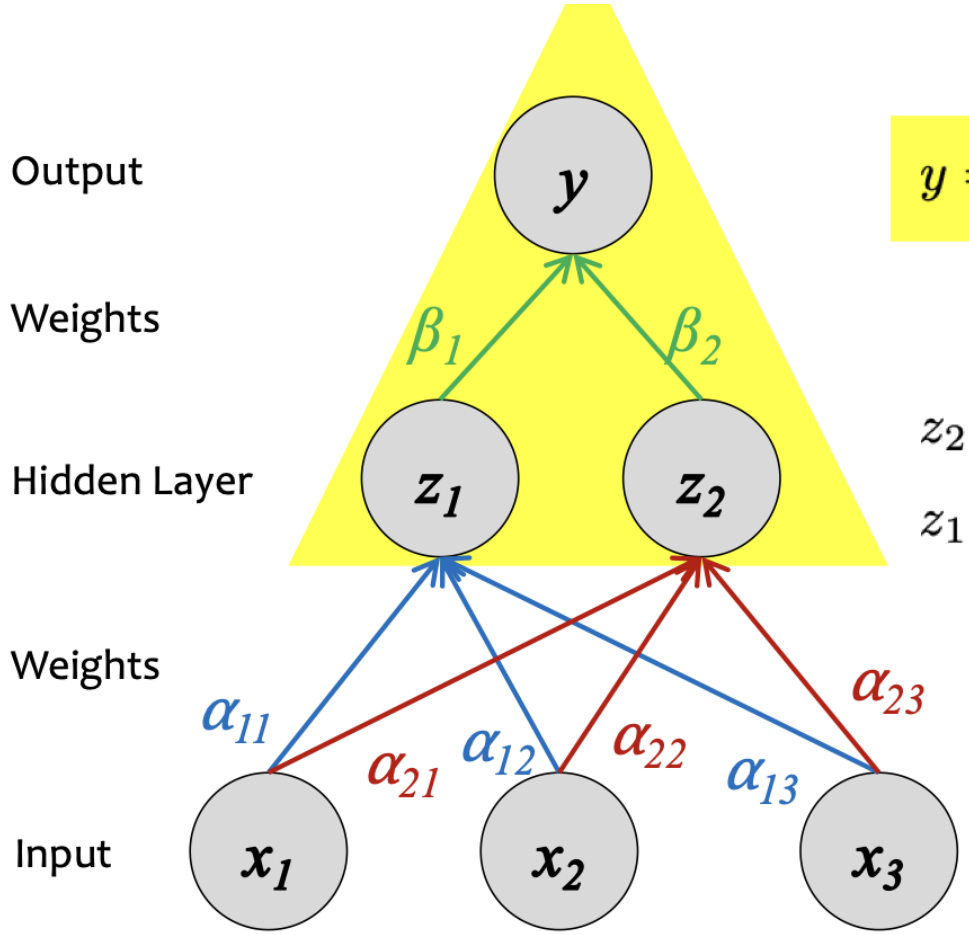


$$y = \sigma(\beta_1 z_1 + \beta_2 z_2)$$

$$z_2 = \sigma(\alpha_{21} x_1 + \alpha_{22} x_2 + \alpha_{23} x_3)$$

$$z_1 = \sigma(\alpha_{11} x_1 + \alpha_{12} x_2 + \alpha_{13} x_3)$$

Forward Computation



$$y = \sigma(\beta_1 z_1 + \beta_2 z_2)$$

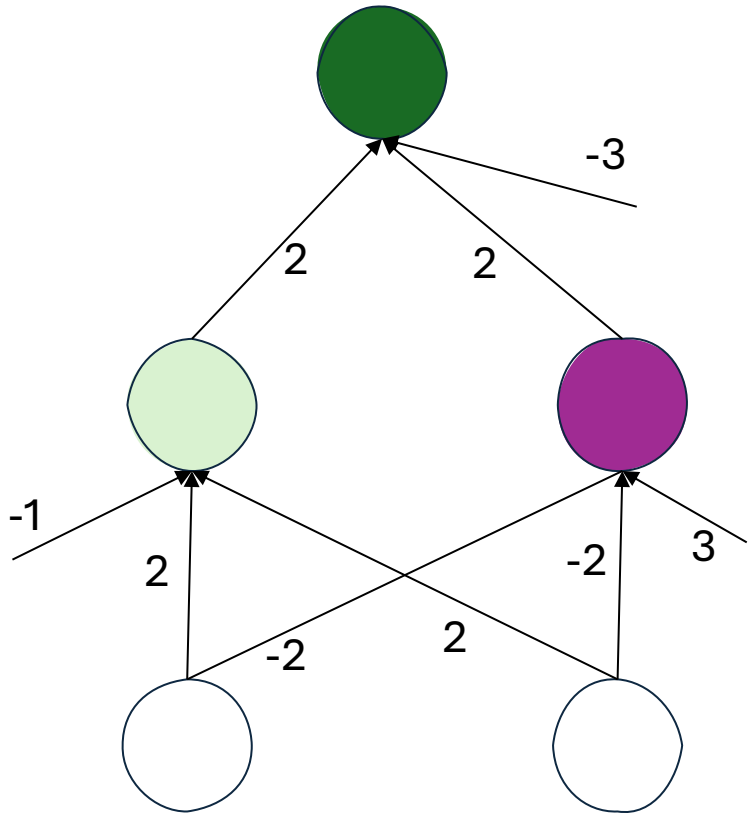
$$z_2 = \sigma(\alpha_{21} x_1 + \alpha_{22} x_2 + \alpha_{23} x_3)$$

$$z_1 = \sigma(\alpha_{11} x_1 + \alpha_{12} x_2 + \alpha_{13} x_3)$$

Solving XoR

- We can easily find an assignment to the weights to represent XoR.
- Each of the latent nodes has a prescribed Boolean ”meaning”

- **Realistic applications require us to think about how to represent complex attributes and their dependencies.**



Feature Encoding

- Nominal Features: represented as 1-of-k encoding (e.g., BoW)

$$\text{table}=\begin{bmatrix}1\\0\\0\\0\end{bmatrix} \qquad \text{chair}=\begin{bmatrix}0\\1\\0\\0\end{bmatrix} \qquad \text{sofa}=\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

- Ordinal features represented as thermometer encoding

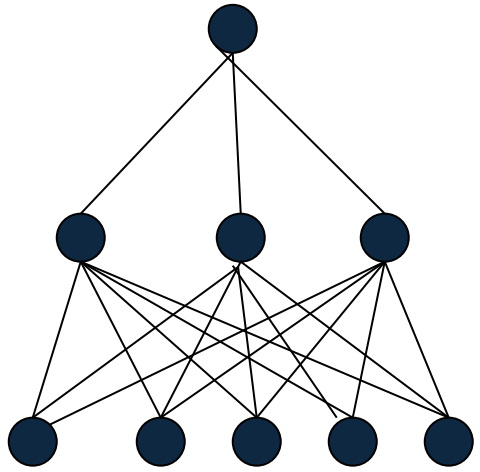
$$\text{good}=\begin{bmatrix}1\\0\\0\end{bmatrix} \qquad \text{great}=\begin{bmatrix}1\\1\\0\end{bmatrix} \qquad \text{GOAT}=\begin{bmatrix}1\\1\\1\end{bmatrix}$$

- Real Valued features as individual measurements
  - Normalizing the values is a good idea!

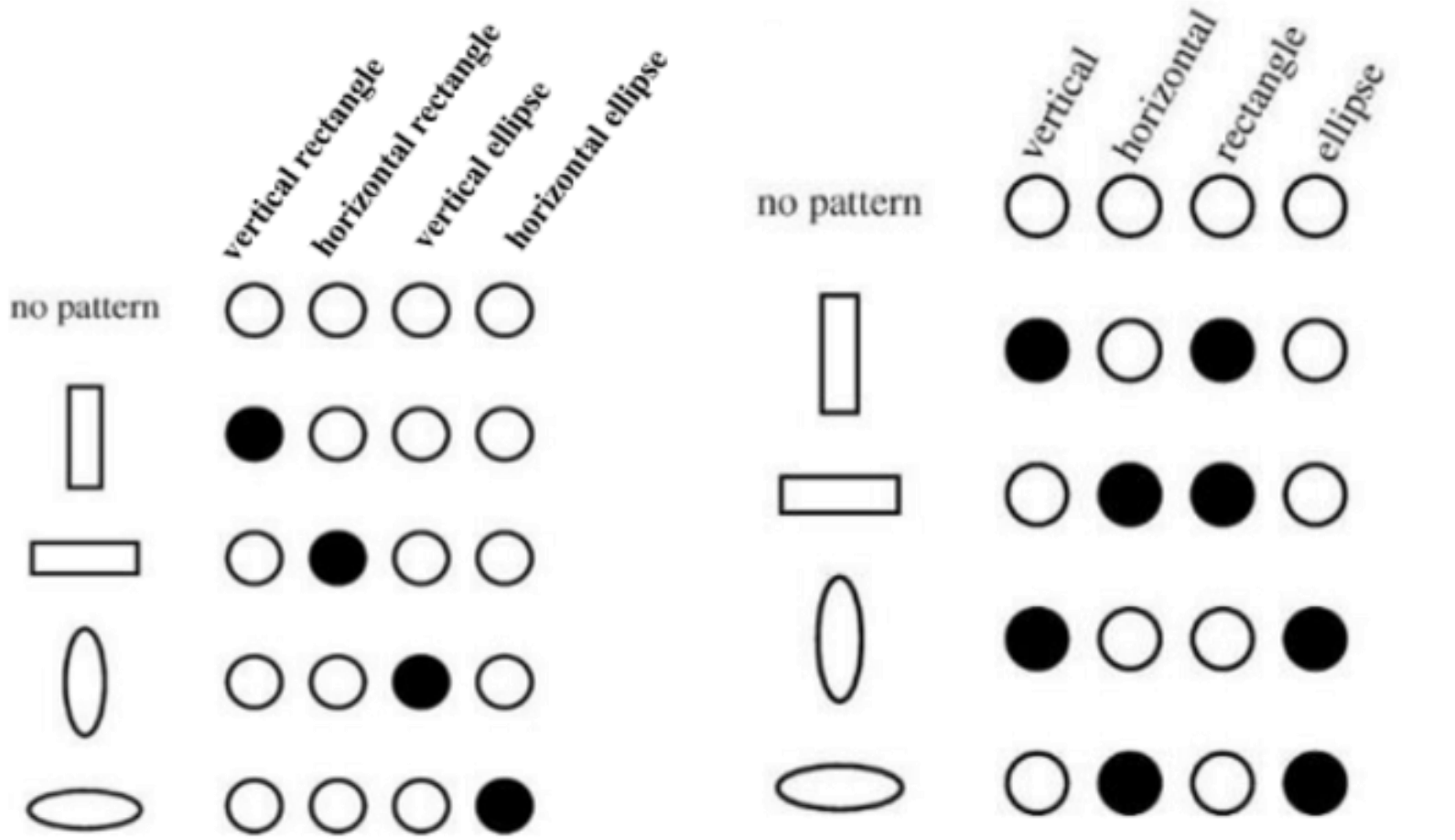
$$\text{temprature}=[78]$$

Feature Encoding

- **Note**: visual depictions of NN can be confusing, multiple input nodes can be required to represent a single feature.
  - We want to predict “buy car” based on attributes: max speed (mph), license type (3 categories), comfort level (basic, regular, premium), has cup holders (yes/no) **How many input nodes?**



Distributed Representation

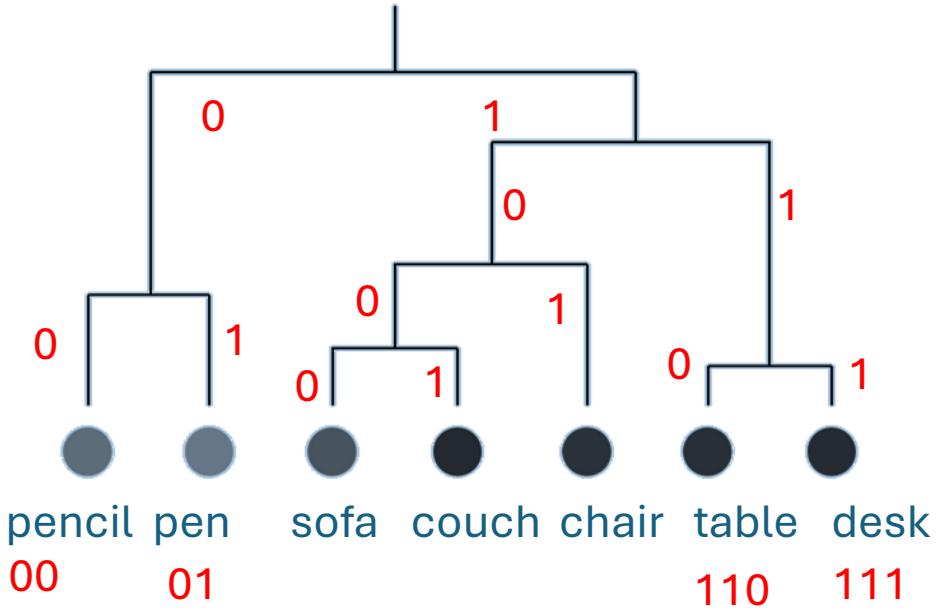


One-hot

Distributed

Data Driven Representation

- Instead of using fixed set of attributes to create distributed representation we can create data driven representations.
- Much more concise representation (i.e., low dim)
  - **The values capture similarity in the vector space (more later!)**



Famous example - Word representation using **Brown clusters.**
Binary encoding of words based on the similarity in the context they appear in.

Data Driven Representation

- Instead of using fixed set of attributes to create distributed representation we can create data driven representations.

lawyer	1000001101000
newspaperman	100000110100100
stewardess	100000110100101
toxicologist	10000011010011
slang	1000001101010
babysitter	100000110101100
conspirator	1000001101011010
...	
Nike	10110111001001010111100
Maytag	10110111001001010111010
Generali	10110111001001010111011
Gap	10110111001001010111110
Harley-Davidson	101101110010010101111110
Enfield	101101110010010101111110
genus	101101110010010101111111
Microsoft	10110111001001011000
Ventritex	10110111001001010010

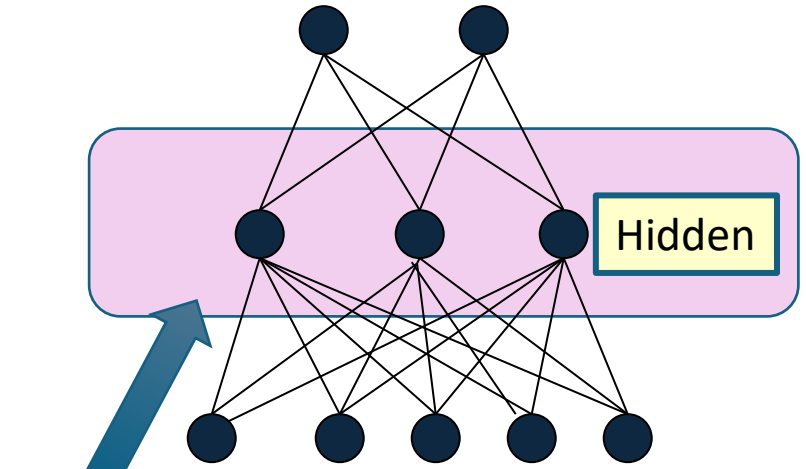
John	1011100100000000000
Consuelo	1011100100000000001
Jeffrey	1011100100000000010
Kenneth	1011100100000000001100
Phillip	1011100100000000011010

Famous example - Word representation using **Brown clusters.**

- Can you identify the "name" property?

Distributed Representation (so far)

- **Very often we use a low dimensional real vector to represent the input.**
  - Latent properties rather than well defined
  - Real values rather than 0/1 encoding
- Can result in a much more concise representation (i.e., low dim)
- **The values can capture similarity in the vector space**
  - Potentially more interesting properties (more on that later..)



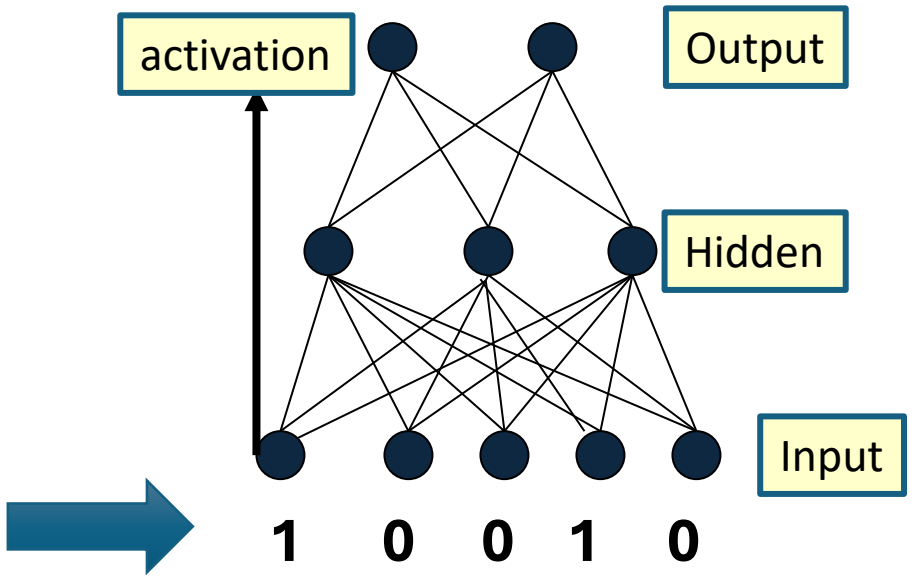
Intuitively, the hidden layer captures latent properties by combining multiple attributes of the previous layer.

Dealing with Structured Data

- Until now, we implicitly assumed that the input layer consists of individual properties. **However** for many domains this is not the case!
  - Image, text, financial data, climate,…
- It opens up a broader discussion about ***how to represent structure.***
- Let's start simple, with sequential data (i.e., chain)
  - **1-Hot BoW using different n-grams size**
  - BoW using distributed representation
  - Concatenation and zero padding.

"I like carrots but not lettuce"

Like 0 0 0 1 0
Carrots 1 0 0 0 0

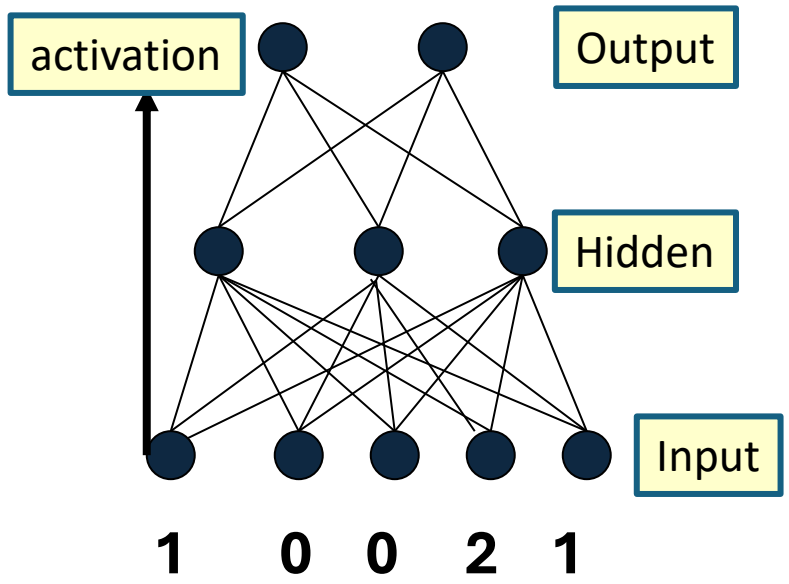


Dealing with Structured Data

- Until now, we implicitly assumed that the input layer consists of individual properties. **However** for many domains this is not the case!
  - Image, text, financial data, climate,…
- It opens up a broader discussion about ***how to represent structure.***
- Let's start simple, with sequential data (i.e., chain)
  - 1-Hot BoW using different n-grams size
  - **BoW using distributed representation**
  - Concatenation and zero padding.

"I like carrots but not lettuce"

Like 0 0 0 1 1
Carrots 1 0 0 1 0



Dealing with Structured Data

- Until now, we implicitly assumed that the input layer consists of individual properties. **However** for many domains this is not the case!
  - Image, text, financial data, climate,…
- It opens up a broader discussion about ***how to represent structure.***
- Let's start simple, with sequential data (i.e., chain)
  - 1-Hot BoW using different n-grams size
  - BoW using distributed representation
  - **Concatenation and zero padding.**

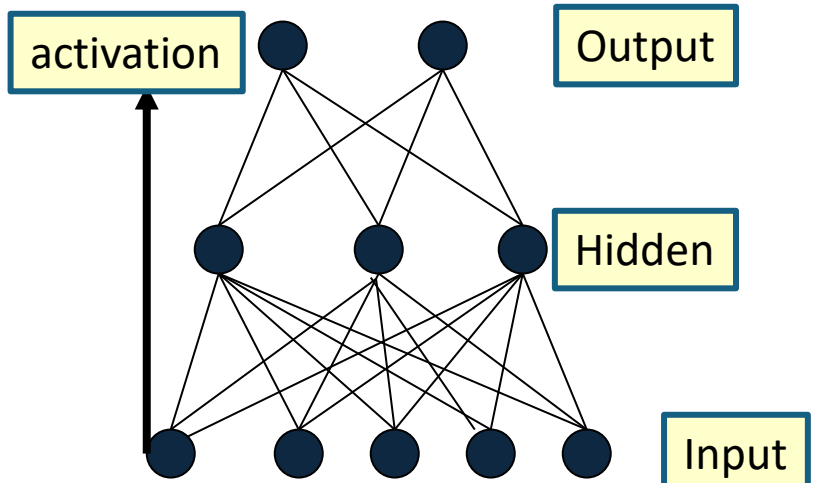
Carrots Like 1 0 0 0 0 0 0 0 1 0

Vs

Like Carrots 0 0 0 1 0 1 0 0 0 0

Like 0 0 0 1 0

Carrots 1 0 0 0 0



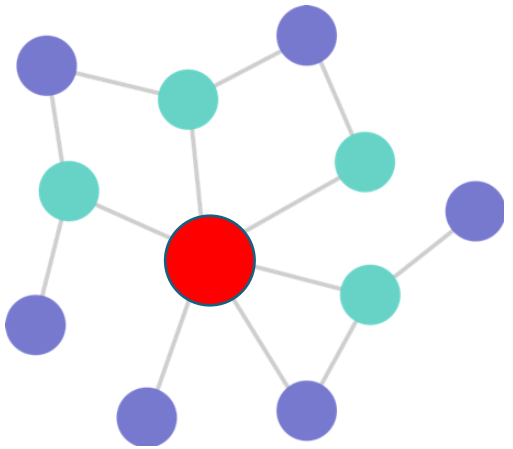
0 0 0 1 0 1 0 0 0 0

Dealing with Structured Data

- This discussion is not limited to text problems.

CACGTGGAG

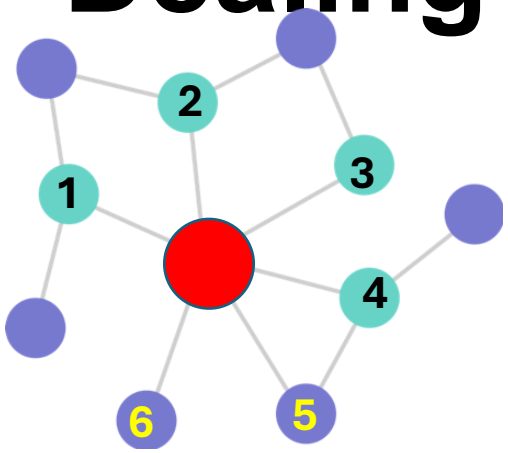
We can think about graphs as a generalization of sequences (i.e., graphs in which each node has at most one child and one parent).



In a graph, each node can have a rich set of properties, and interacts with other nodes (potentially typed edges)

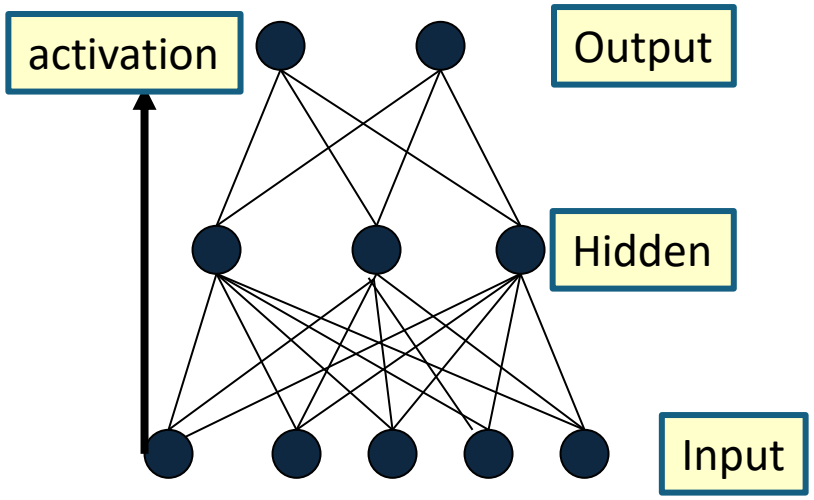
What is the right architecture for a node classification problem?

Dealing with Structured Data



What is the right architecture for a node classification problem?

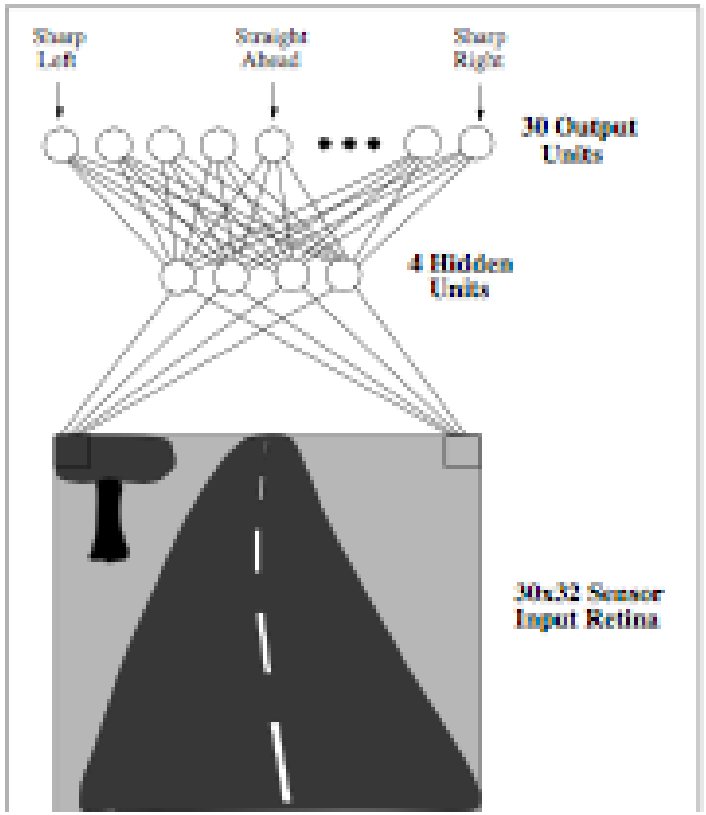
- **Node Representation:**
  - How is each node represented?
- **Node neighborhood:**
  - What is a neighborhood? direct neighbors? Beyond?
  - Does the order of nodes matter? If so, how should we decide it?
  - What if it doesn't (i.e., graph doesn't have any meaningful edge properties that would allow us to assign order), what should we do?



[target node] [N_1].. [N_{last}]

Dealing with Structured Data

- ALVINN Self driving car from the 90's (Pomerleau NeurIPS'88)
- NN trained to control wheel based on road image. Linearized the order of pixels (similar to concatenating words in a sentence).
- **What would be a better representation for images?**



Let’s build our own NN!

- For a given problem, we have to decide:
  - How many input units?
  - How many hidden layers?
  - How many output units?

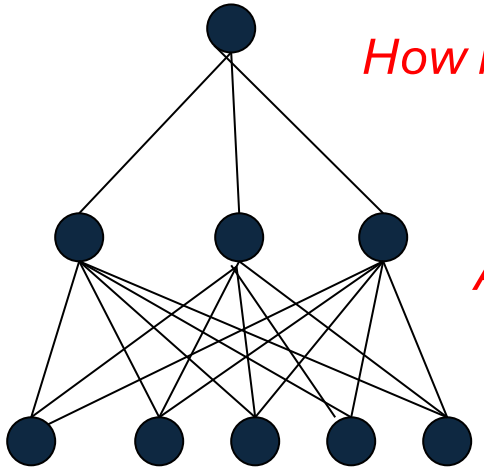
- What are the considerations we should have when deciding the number of hidden units?
  - Is there a “right” number?

NN Tradeoffs

- Our goal is to build an apple-sentiment classifier from text. Here are two examples that the classifier should work well on.

I like apples but not oranges I like oranges but not apples

- What should our network architecture look like to deal with this problem? Assume that we can love_apples, hate-apples or be OK_with_apples.



How many output units?

Architecture?

Input Representation?

Named Entity Recognition (NER)

- The problem of identifying named entities in text.
- Identify if word spans are **loc/org/per** or **none**.
- Classification problem: world level decisions
- **Type**: **loc/org/per** or **none**.
- **Segmentation**: Begin/Inside/Outside

BOrg IOrg IOrg O BLoc ILoc

[Mount Sinai hospital] in [New York]

Can you identify the potential issue here?

Let’s build our own NN!

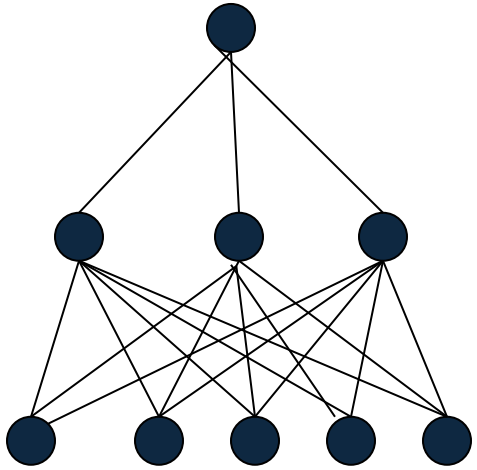
- Let’s design a NN for …
  - **Named Entity Recognition**
    - **Binary case**: given a sentence, decide which words are NE and which are not
    - **Multi-class case**: decide if a word is a Loc, Per, Org, None

- **What is the right architecture?**

NN for NER

Binary case:

Single output unit, can be interpreted as a threshold function



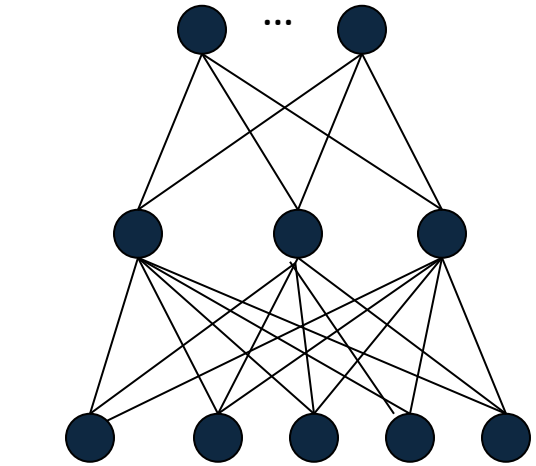
Note: each word position is a vector of Dim V

Mount Sinai *hospital* in New York

Is the center word a Named Entity?

NN for NER

Multiclass case: winner-takes-all

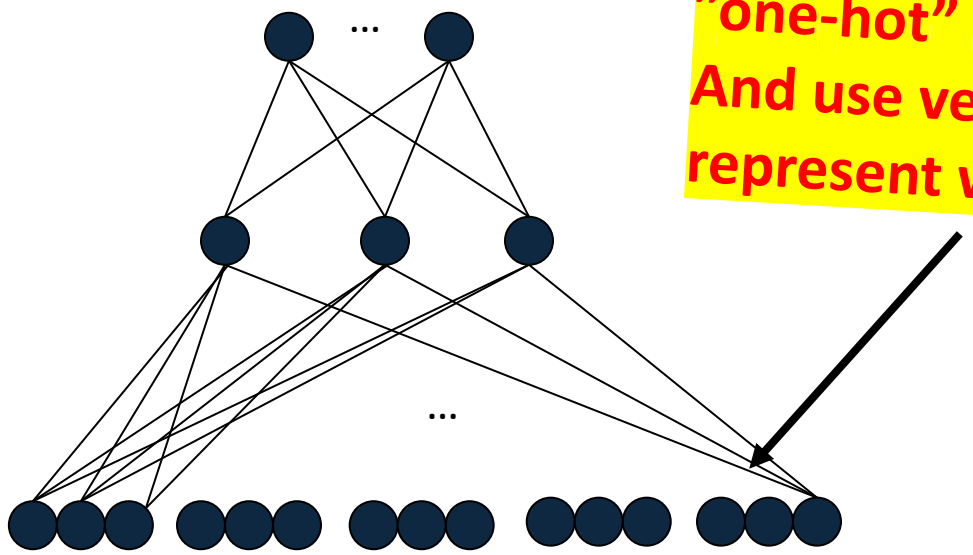


Mount Sinai ***hospital*** in New York

Is the center word a Named Entity?

NN for NER

Multiclass case: winner-takes-all



We can also replace the "one-hot" representation, And use vectors to represent words!

Is the center word a Named Entity?

Word Vectors (reminder)

Recall:

- **BoW representation is also known as 1-hot**
  - Points to the fact that it is an incredibly sparse representation.
- **Dense vectors are an alternative** – each word is represented in a continuous space, using dense (i.e., not sparse) vectors
  - Can help reduce the feature space
- **Where do these vectors come from?**
  - Let's assume we can learn these representations from data
    - Sometimes referred to as an *embedding function,* mapping an input to a vector

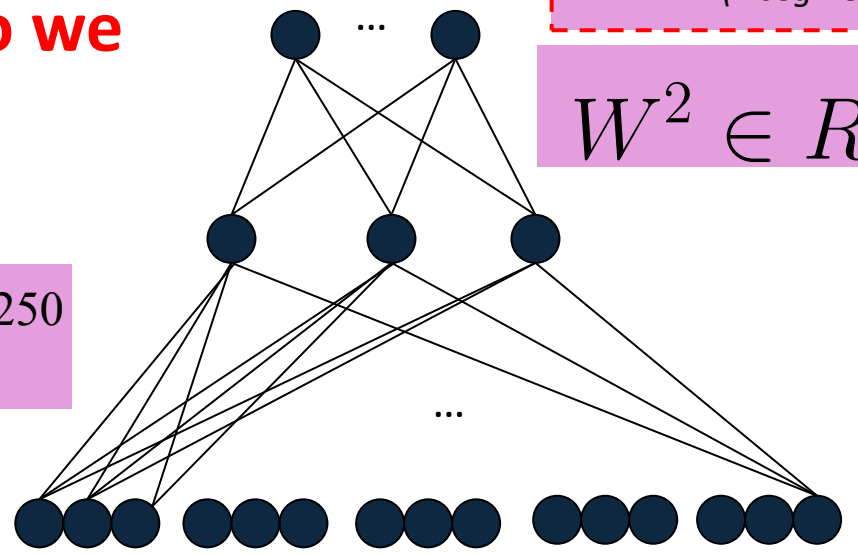
NN for NER

How many parameters do we have to train?

$R^{3 \times 7}$ *(+ segmentation)*

$W^2 \in R^{3 \times 4}$

$W^1 \in R^{3 \times 250}$



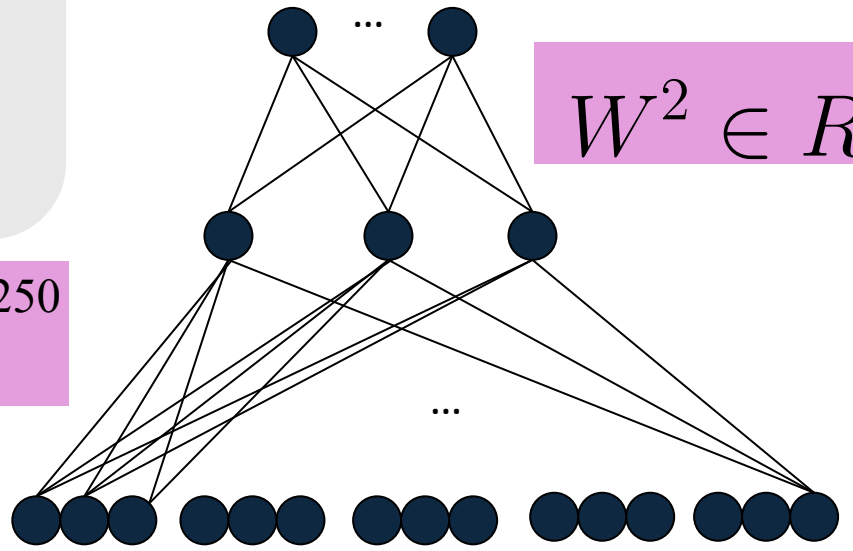
Let's assume our word embedding are in R^{50}

$x \in R^{50}$

Question: *How would this architecture change if we wanted to add NE segmentation as well?*

important point about NNs:
representation sharing, i.e., the input transformation (hidden layer activations) is used by all output classes. The class-specific "specialization" happens at the final layer.

$$W^1 \in R^{3 \times 250}$$



$$W^2 \in R^{3 \times 4}$$

This is the increase in the number of parameters when moving from 4 class to 7 class classification

$R^{3 \times 7}$ *(+ segmentation)*

$$x \in R^{50}$$

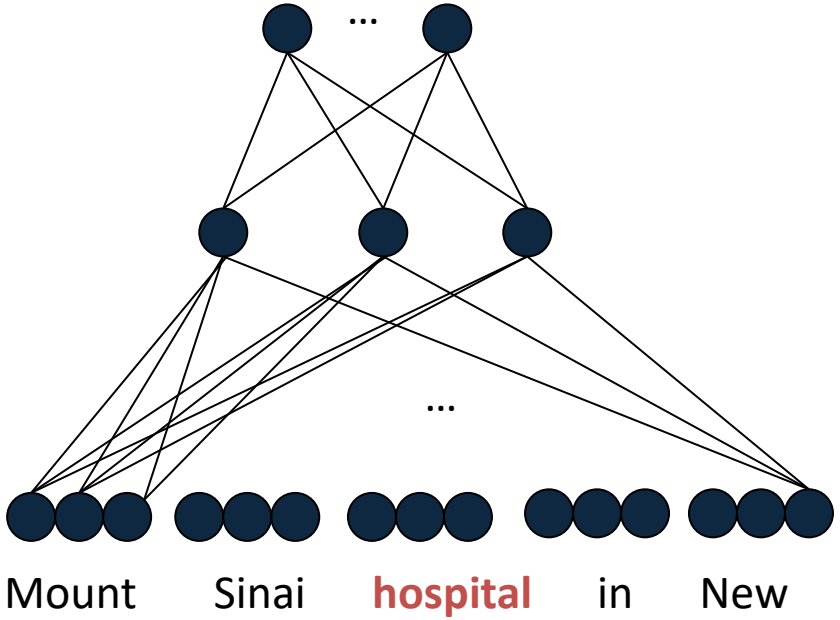
Given the same feature representation when training a multiclass logistic regression model – **what would the increase in the number of parameters be? Explain!**

Representation Sharing

important point about NNs: representation sharing, i.e., the input transformation (hidden layer activations) is used by all output classes. The class-specific "specialization" happens at the final layer.

Intuition

B_Per, B_Org, B_Loc, are labels that share information. They capture the **beginning** of a named entity, which for English has similar properties. **Sharing representation** via the hidden layers can help improve the prediction of label_A using data annotated for Label_B

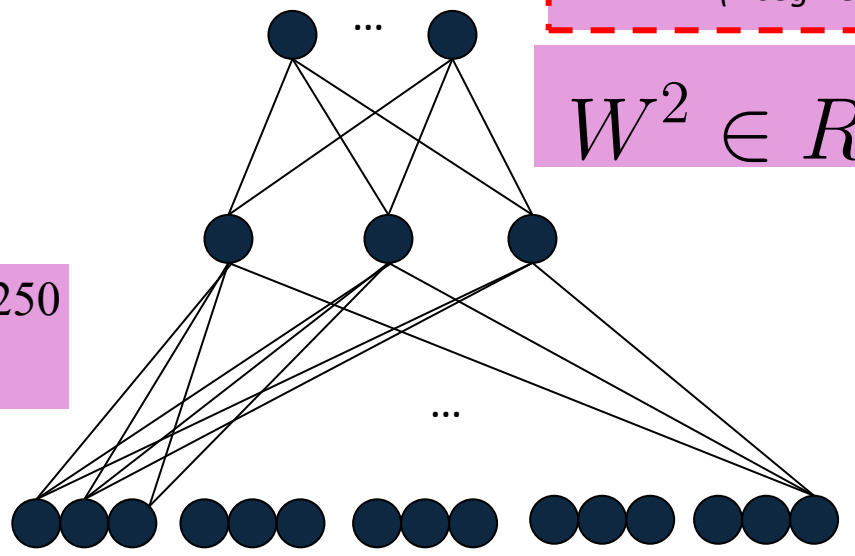


NN for NER

$R^{3 \times 7}$ *(+ segmentation)*

$W^2 \in R^{3 \times 4}$

$W^1 \in R^{3 \times 250}$



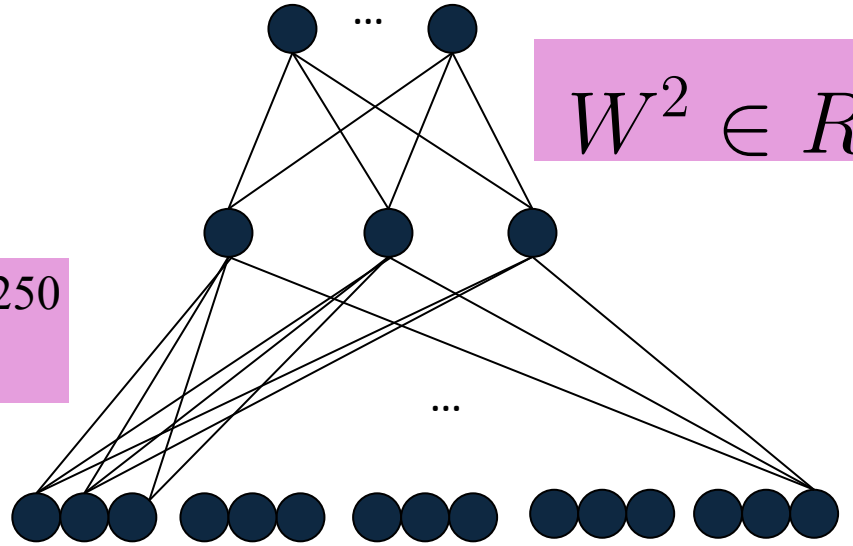
Let's assume our word embedding are in R^{50}

$x \in R^{50}$

What would happen if we switch to a Bag-of-Words representation over word embeddings?

NN for NER

$$W^2 \in R^{3\times 4}$$



$$W^1 \in R^{3 \times 250}$$

$$x \in R^{50}$$

Forward computation:

$$h_j = \sigma(t_j) = \sigma(\sum_i w^1_{ji}\, x_i)$$

Hidden layer

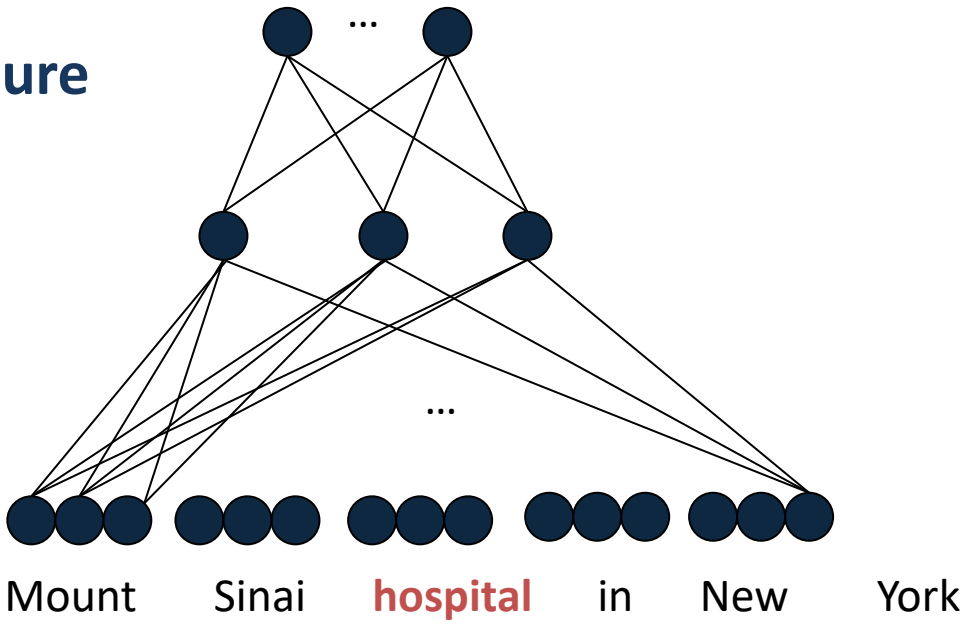
$$\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w^2_{kj} h_j)$$

output layer

NN for NER

Can we simplify this architecture and *not* use a hidden layer?

Why not just use the word vectors directly?



What do we get by adding an *extra* layer?

Training Neural Nets

- We can use the **chain rule** to get the gradient:

Chain rule:

$$\frac{\partial}{\partial v_{w_i}} f(z(v_{wi})) = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial v_{w_i}}$$

Simple Example

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = (3)(2x) = 6x$$

Note: to compute $\frac{dz}{dx}$ we first have to compute $\frac{dz}{dy}$. If we need to compute both, it's better to start with $\frac{dz}{dy}$ and reuse it later.. (the idea behind backprop implementation)

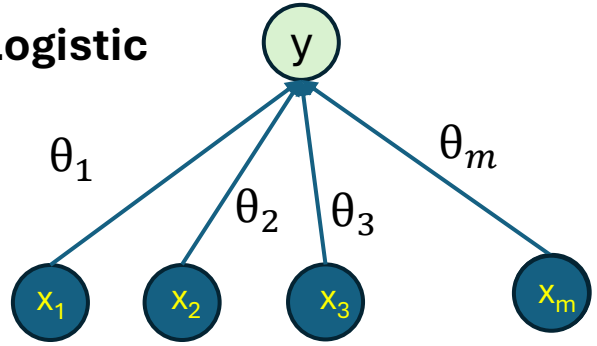
Backprop Intuition

- **Backpropagation** = **Gradient descent** + **chain rule** (*applied to the NN architecture)*
- *Example, compute* $J = \cos(\sin(x^2) + 3x^2)$ *on forward pass and the derivate* $\frac{dJ}{dx}$ *on the backward pass*

Forward	Backward
$J = cos(u)$	$\frac{dJ}{du}$ += $-sin(u)$
$u = u_1 + u_2$	$\frac{dJ}{du_1}$ += $\frac{dJ}{du}\frac{du}{du_1}$, $\frac{du}{du_1} = 1$ $\quad$ $\frac{dJ}{du_2}$ += $\frac{dJ}{du}\frac{du}{du_2}$, $\frac{du}{du_2} = 1$
$u_1 = sin(t)$	$\frac{dJ}{dt}$ += $\frac{dJ}{du_1}\frac{du_1}{dt}$, $\frac{du_1}{dt} = \cos(t)$
$u_2 = 3t$	$\frac{dJ}{dt}$ += $\frac{dJ}{du_2}\frac{du_2}{dt}$, $\frac{du_2}{dt} = 3$
$t = x^2$	$\frac{dJ}{dx}$ += $\frac{dJ}{dt}\frac{dt}{dx}$, $\frac{dt}{dx} = 2x$

Backprop Intuition

Easy case: **Logistic Regression**



Forward

$$J = y^* \log y + (1 + y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{m} \theta_j x_j$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

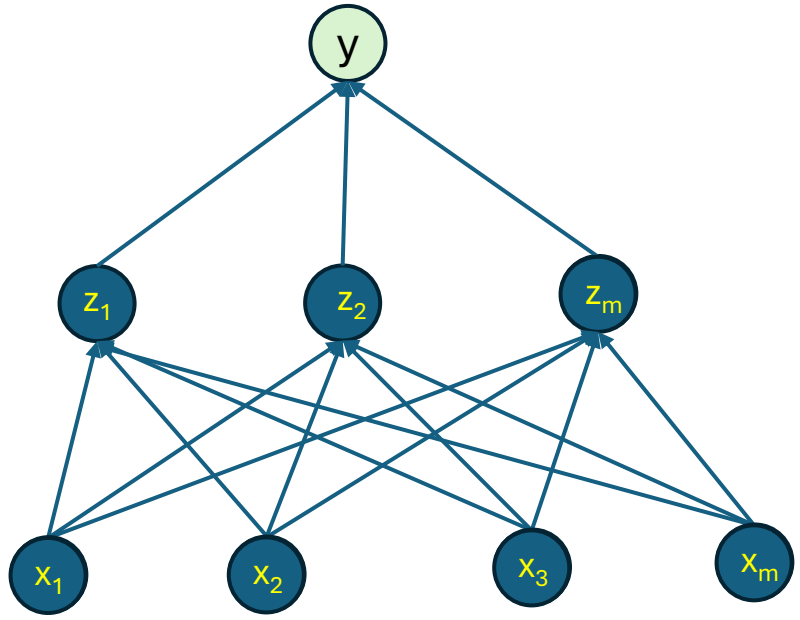
$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \qquad \frac{dy}{da} = \frac{exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \qquad \frac{da}{d\theta_j} = x_j$$

$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \qquad \frac{da}{dx_j} = \theta_j$$

Backprop

One hidden layer,
forward computation



Loss function: $J = y^* \log y + (1 + y^*) \log(1 - y)$

Output: $y = \frac{1}{1 + \exp(-b)}$

Output (raw): $b = \sum_{j=0}^{D} \beta_j z_j$

Hidden layer: $z_j = \frac{1}{1 + \exp(-a_j)}, \forall j$

Hidden layer(raw): $a_j = \sum_{i=0}^{m} \alpha_{ji} x_i, \ \forall j$

Backprop

Loss function:

$$J = y^* \log y + (1 + y^*) \log(1 - y)$$

Output: $$y = \frac{1}{1 + \exp(-b)}$$

Output (raw): $$b = \sum_{j=0}^{D} \beta_j z_j$$

Hidden layer: $$z_j = \frac{1}{1 + \exp(-a_j)}, \forall j$$

Hidden layer(raw): $$a_j = \sum_{i=0}^{m} \alpha_{ji} x_i, \ \forall j$$

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \qquad \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \qquad \frac{db}{d\beta_j} = z_j$$

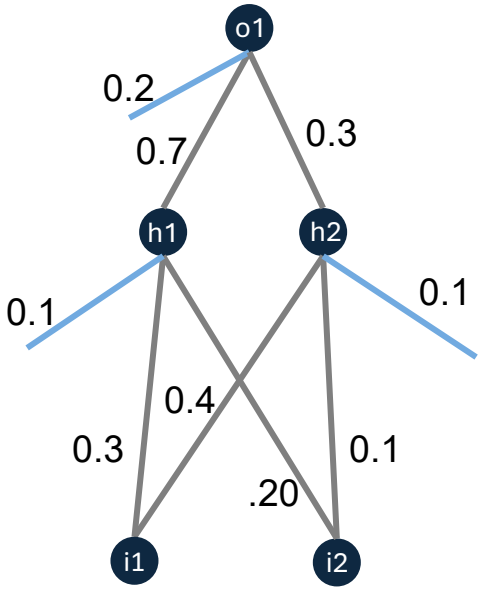
$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \qquad \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \qquad \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \qquad \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \qquad \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$$

Forward / Backward Prop Example



Build the network:
2 input units: i1, i2
2 hidden units: h1, h2

Activation function:

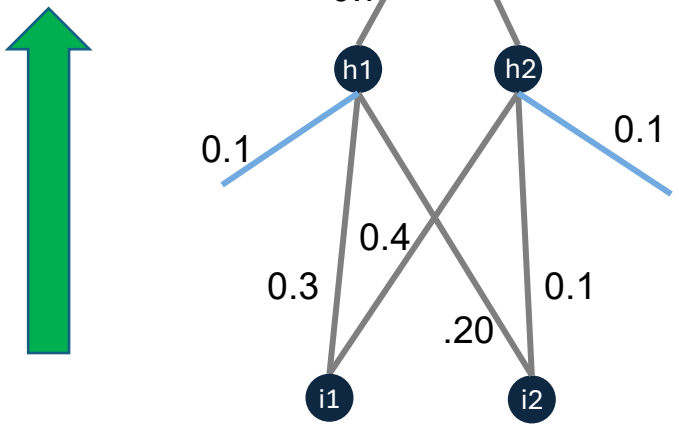
sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$

Loss function:

Mean Squared Error (MSE): $L = \frac{1}{2}(y - \hat{y})^2$

Input example:
x=(0.6, 0.1) Label: y=1

Forward / Backward Prop Example



Forward Computation:

Input example:

x=(0.6, 0.1) Label: y=1

Raw Activations of hidden units:

$$a_1 = 0.3(0.6) + 0.2(0.1) + 0.1 = 0.18 + 0.02 + 0.1 = 0.30$$

$$a_2 = 0.4(0.6) + 0.1(0.1) + 0.1 = 0.24 + 0.01 + 0.1 = 0.35$$

Now, apply activation function:

h1= $\sigma(0.3)$ =0.574

h2= $\sigma(0.35)$=0.587

Raw Activation of output unit:

$$\mathrm{b} = W^2\boldsymbol{h} + b^2 = 0.7(0.574) + 0.3(0.587) + 0.2 = 0.402 + 0.176 + 0.2 = 0.778$$

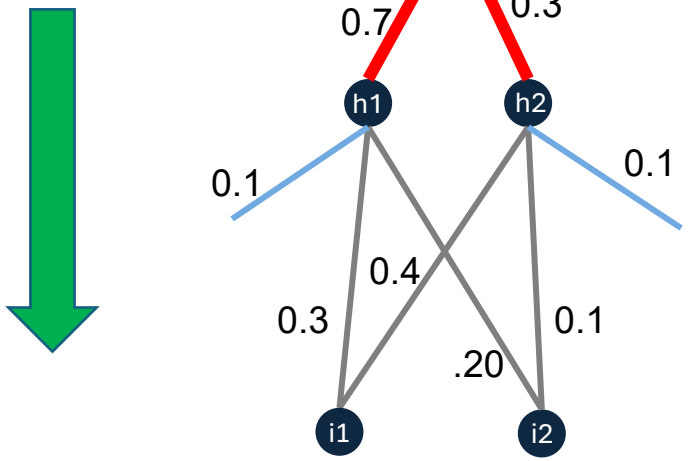
Now, apply activation function:

$$\hat{y} = \sigma(b) = \sigma(0.778) = 0.685$$

Compute the loss:

$\frac{1}{2}(y - \hat{y})^2$ = ½*(1-0.685)^2 = 0.0496125

Forward / Backward Prop Example



Backwards Pass:

$$\frac{\partial Err}{\partial W^2} = \frac{\partial Err}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial b} \times \frac{\partial b}{\partial W^2}$$

$$\frac{\partial Err}{\partial \hat{y}} = \frac{1}{2}\, 2(\hat{y} - y) = (\hat{y} - y) = 0.685 - 1 = -0.315$$

$$\frac{\partial \hat{y}}{\partial b} = \sigma'(b) = \hat{y}(1 - \hat{y}) = 0.685\ (1-0.685) = 0.685\ (0.315) = 0.216$$

Recall: $\frac{\partial \hat{y}}{\partial b} = \sigma(b)(1 - \sigma(b))$

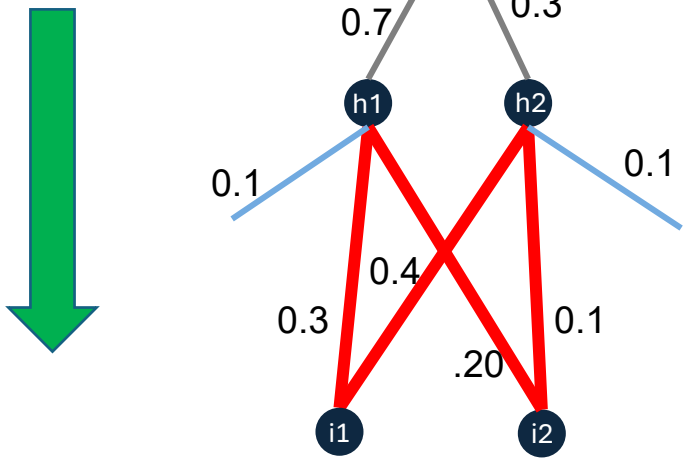
$$\frac{\partial b}{\partial W^2} = \boldsymbol{h}$$

Recall: $b = W^2 h + b$

$$\frac{\partial Err}{\partial W^2} = (-0.315) * (0.216) * [0.574, 0.587] = (-0.068) * [0.574, 0.587] = [-0.039, -0.040]$$

$$\frac{\partial Err}{\partial bias^2} = (-0.068)$$

Forward / Backward Prop Example



Backwards Pass:

$$\frac{\partial Err}{\partial W^1} = \frac{\partial Err}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial b} \times \frac{\partial b}{\partial h} \times \frac{\partial h}{\partial a} \times \frac{\partial a}{\partial W^1}$$

$$\frac{\partial Err}{\partial \hat{y}} = \frac{1}{2}\, 2(\hat{y} - y) = (\hat{y} - y) = 0.685 - 1 = -0.315$$

$$\frac{\partial \hat{y}}{\partial b} = \sigma'(b) = \hat{y}(1 - \hat{y}) = 0.685\,(1-0.685) = 0.685\,(0.315) = 0.216$$

Already computed!

$$\frac{\partial h}{\partial a} = \boldsymbol{W^2} = [0.7, 0.3]$$

$$\frac{\partial h}{\partial a} = h(1-h) = 0.5744*(1-0.5744),\ 0.5866*(1-0.5866)] = [0.2445,\ 0.2425]$$

Recall: $h = \sigma(a)$
$\sigma'(a) = \sigma(a)(1 - \sigma(a))$

$$\frac{\partial a}{\partial W^1} = x = [0.6,\ 0.1]$$

$$\frac{\partial Err}{\partial W^1} = \begin{matrix} -0.00696 & -0.00116 \\ -0.00294 & -0.00049 \end{matrix}$$

$$\frac{\partial Err}{\partial bias1} = [-0.0116, -0.0049]$$

Back-Prop Comments

- **No guarantee of convergence;** may oscillate or reach a local minima.
  - *In practice, many large networks can be trained on large amounts of data for realistic problems.*
  - **To avoid local minima**: several trials with different random initial weights with majority or voting techniques
- Many epochs (10Ks) may be needed for adequate training.
  - Large data sets may require many hours (days) of CPU
- **Termination criteria**: Number of epochs; Threshold on training set error; No decrease in error; Increased error on a validation set.

Overfitting!

- Neural networks are highly expressive.
- They also define a non-linear error surface.
- **As a result it's hard to train them!**
- Several "tricks" can help control overfitting
  - Regularization
  - Adding noise
  - Data augmentation
  - Dropout
  - ….

Over-training Prevention

- Running too many epochs may over-train the network and result in over-fitting
  - Keep a hold-out validation set and test accuracy after every epoch
  - Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.
  - ***Why not just stop once validation error starts increasing?***
- To avoid losing training data to validation:
  - Use 10-fold cross-validation to determine the average number of epochs that optimizes validation performance
  - Train on the full data set using this many epochs to produce the final results

Early stop

- **Idea**: don't train the network to too small training error
- When training, also output validation error
  - Every time validation error improved, store a copy of the weights
  - When validation error not improved for some time, stop
  - Return the copy of the weights stored

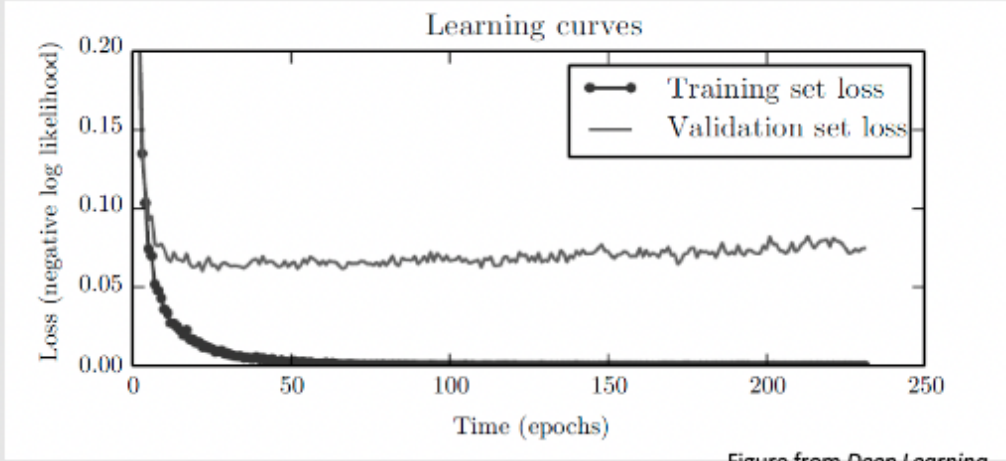
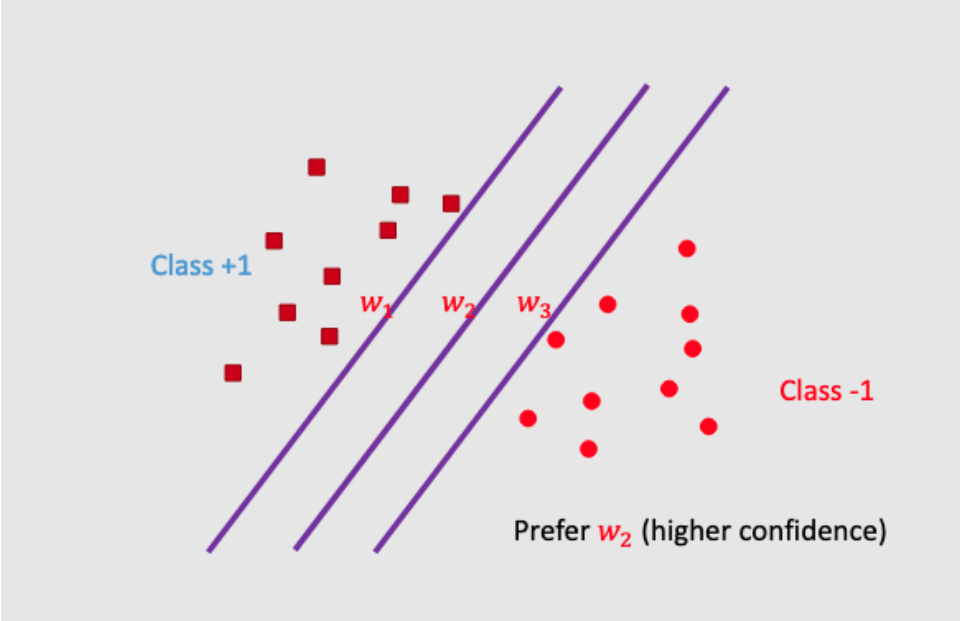


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

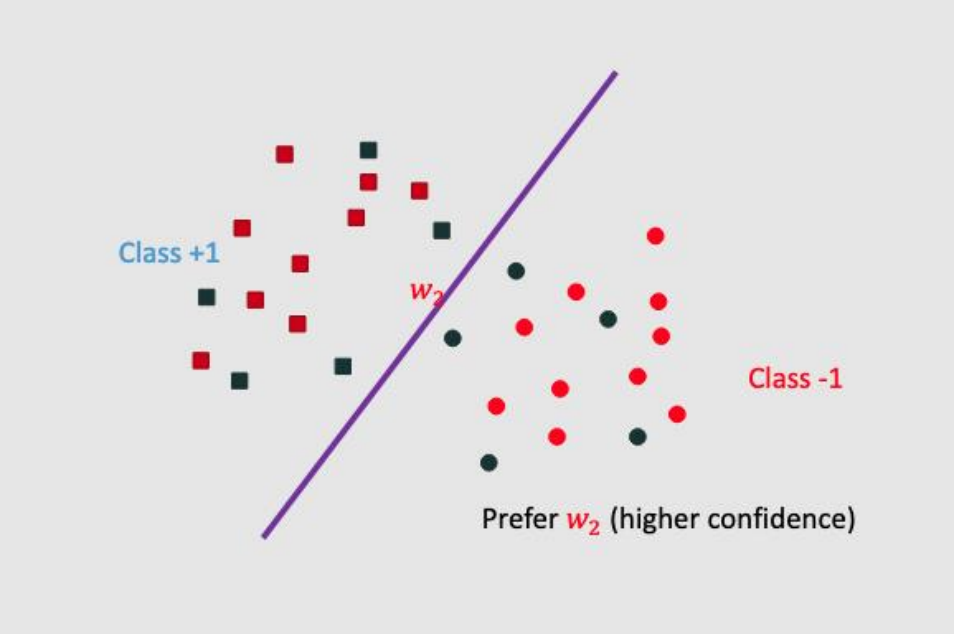
Over-training Prevention

- Too few hidden units prevent the system from adequately fitting the data and learning the concept.
- Using too many hidden units leads to over-fitting.
- **Similar cross-validation method can be used to determine an appropriate number of hidden units.**
- Another approach to prevent over-fitting is **weight- decay**: all weights are multiplied by some fraction in (0,1) after every epoch.
  - Encourages smaller weights and less complex hypothesis
- Equivalently: **use a regularizer**

Add Noise to Input



This approach is similar to a regularizer (e.g., weight decay)



Data Augmentation

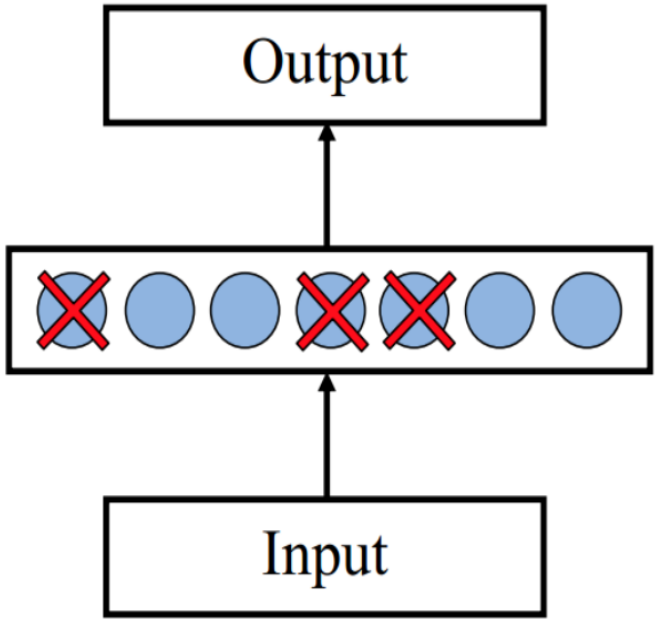
- Augment the dataset automatically

Figure from *Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7,* by Keven Wang

- Adding noise: special case of data augmentation

Dropout Training

- Proposed by (Hinton et-al 2012)



Prevent feature co-adaptation

Encourage "independent contributions"

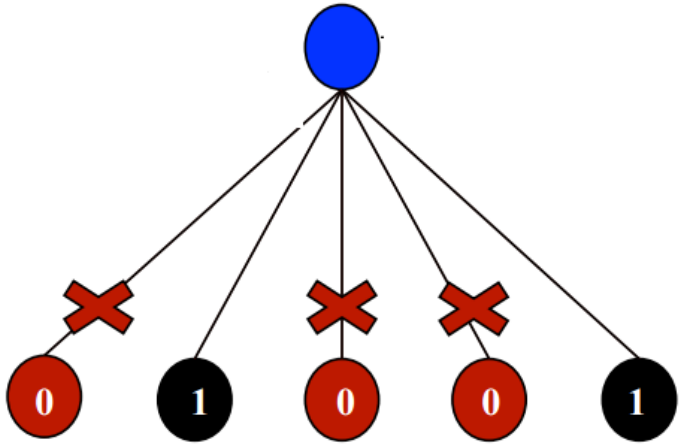
From different features

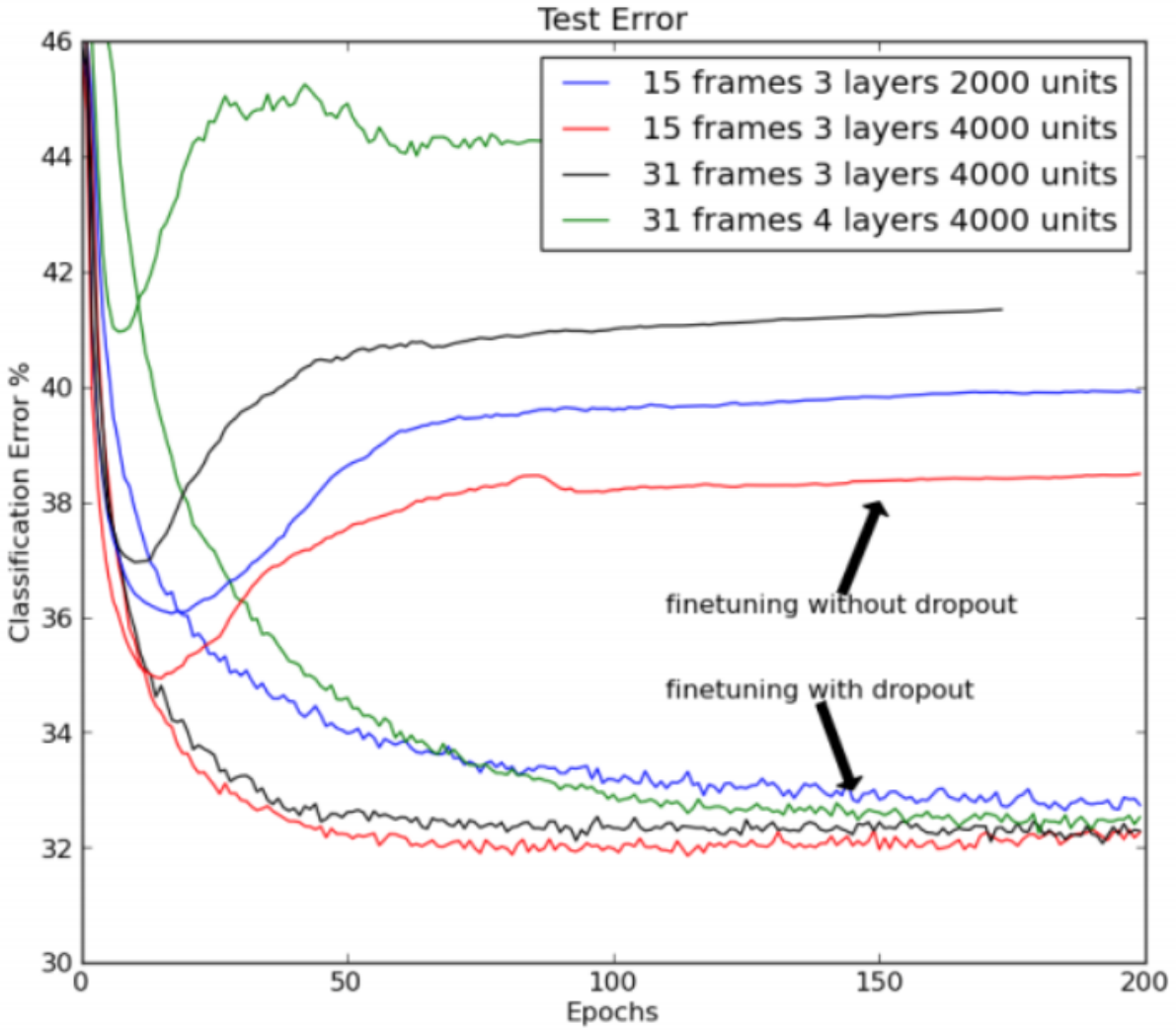
- At each training step, decide whether to delete one hidden unit with some probability p

Dropout Training

- **Model averaging effect**
  - Average the results of multiple NN
  - Each NN has a different initialization point, resulting in a different model
  - Extremely computationally intensive for NNs!
- Much stronger than the known regularizer

- **What about the input space?**
  - Do the same thing!





- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)