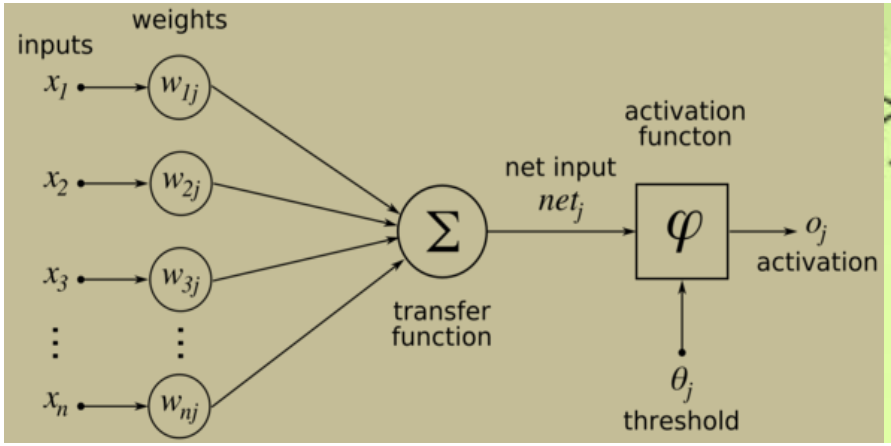
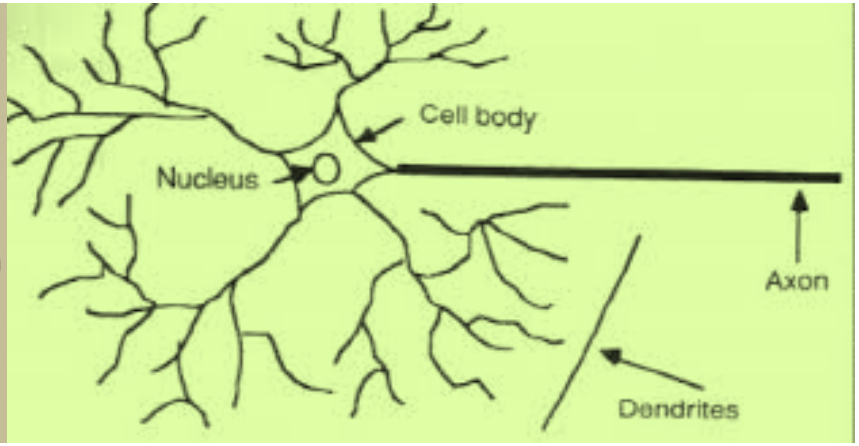
ML4NLP

Introduction to Deep Learning for NLP





Dan Goldwasser

Purdue University

dgoldwas@purdue.edu

Deep Learning in NLP

So far we used **linear models**

- They work fine, but we made some assumptions

The problems are linear, or we are willing to work in order to make them linear

Feature engineering is a form of expressing domain knowledge

We are fine working with **very very high** dimensional data

- It's easy to get there.
- Everything is mostly linear at that point.

What could go wrong?

Deep Learning Architectures for NLP

- The key question we follow – how can you **build complex (compositional?) meaning representation**, for **larger units than words**, to support **advanced classification tasks**?

- We will look at several popular architectures.
  - We will build on a *"recently introduced model from the 70's"*
    - NN made a come-back in the last 5 years.

Neural Networks

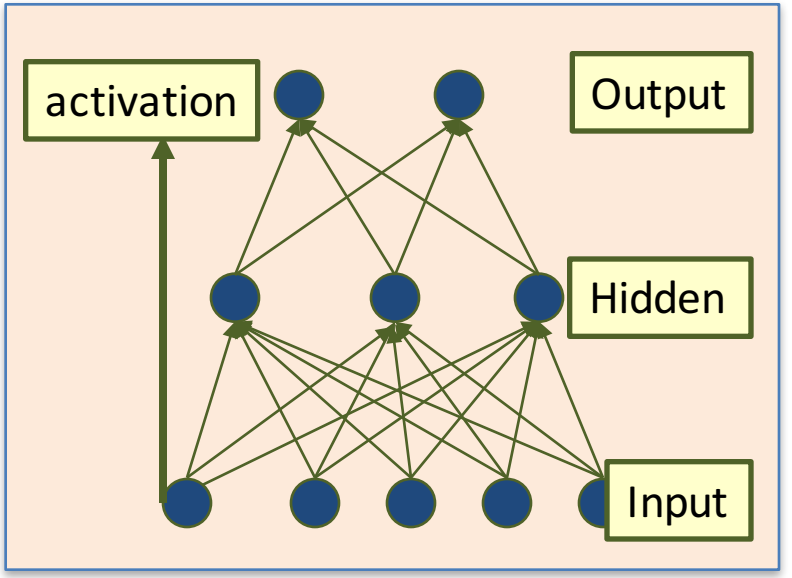
- Robust approach for approximating functions
  - *Functions can be real-valued, discrete or vector valued*
- One of the most effective general purpose learning methods
  - A lot of attention in the 90's, making a comeback!
- Especially useful for complex problems, where the input data is hard to interpret
  - Sensory data (speech, vision, etc)
- Many successful application domains
- **Interesting spin**: *Learning input representation*
  - *So far we thought about the feature representation as being fixed*

Neural Network

- Simply put, NN's are functions **f: X→Y**
  - *f is a **non-linear** function*
  - *X is a **vector** of continuous or discrete variables*
  - *Y is a **vector** of continuous or discrete variables*
- **Very expressive classifier**
  - In fact, NN can be used to represent any function
- The function f is represented using a network of logistic units

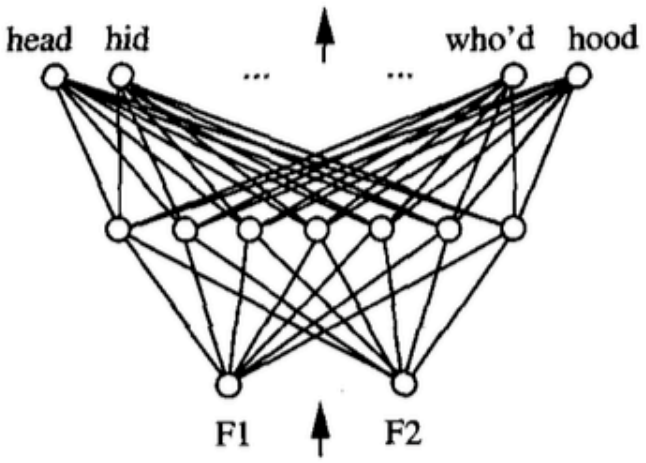
Multi Layer Neural Networks

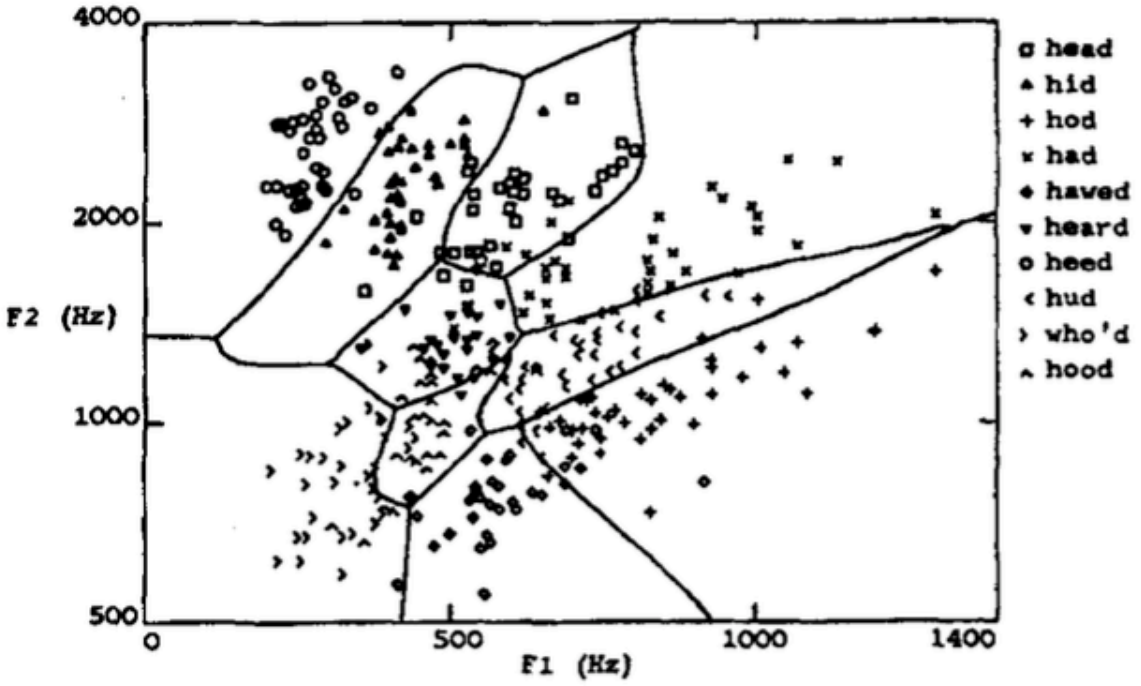
- Multi-layer network were designed to overcome the computational (**expressivity**) limitation of a single threshold element.
- The idea is to **stack** several layers of threshold elements, *each layer using the output of the previous layer as input*



Multi-layer networks **can represent arbitrary functions**, but building effective learning methods for such network was [thought to be] difficult.

Example: NN for speech vowel recognition



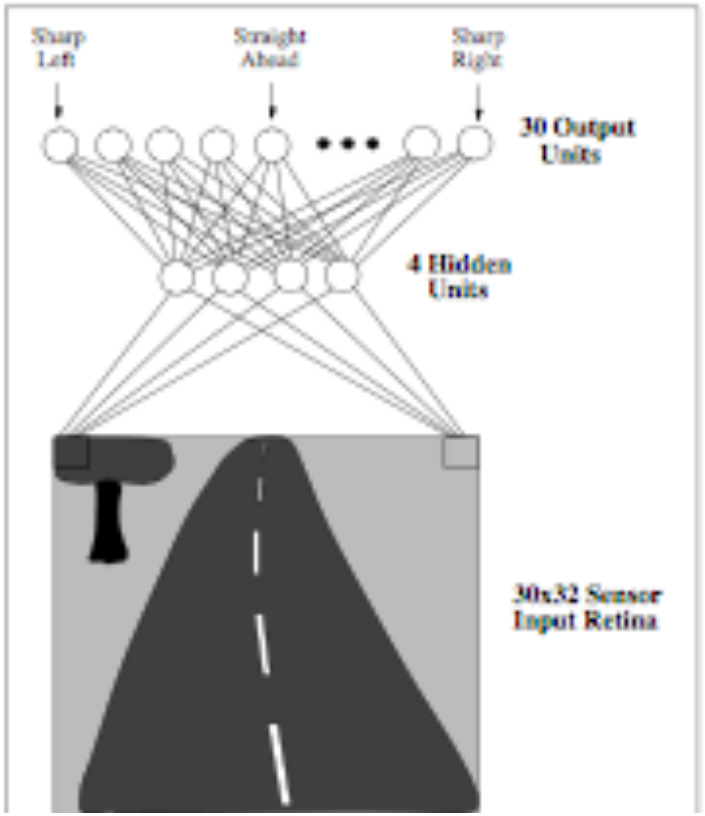


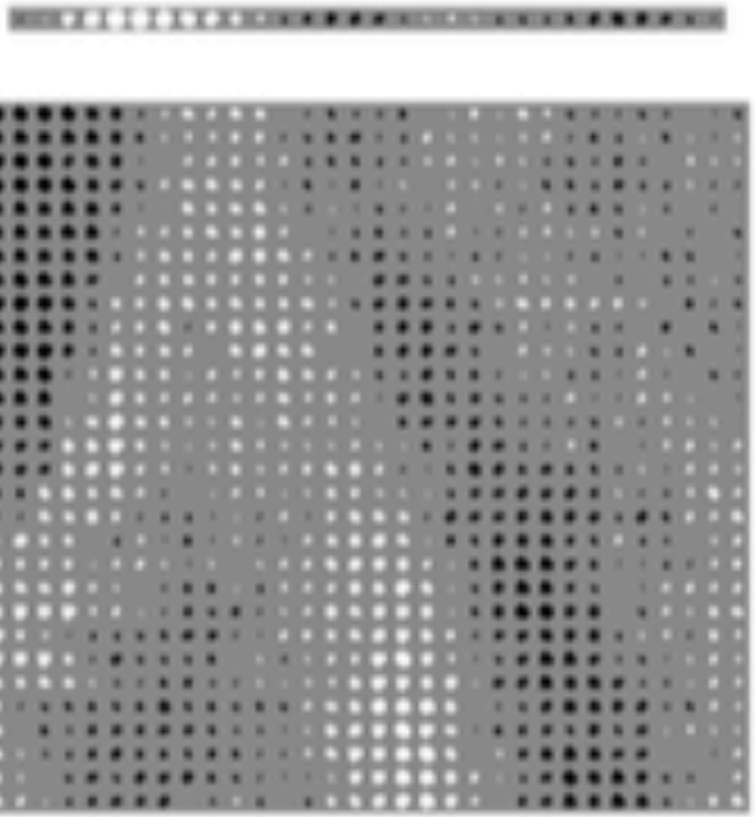
ALVINN: autonomous land vehicle in a NN

Figure 3: NAVLAB, the CMU autonomous navigation test vehicle.

Pomerleau ‘89

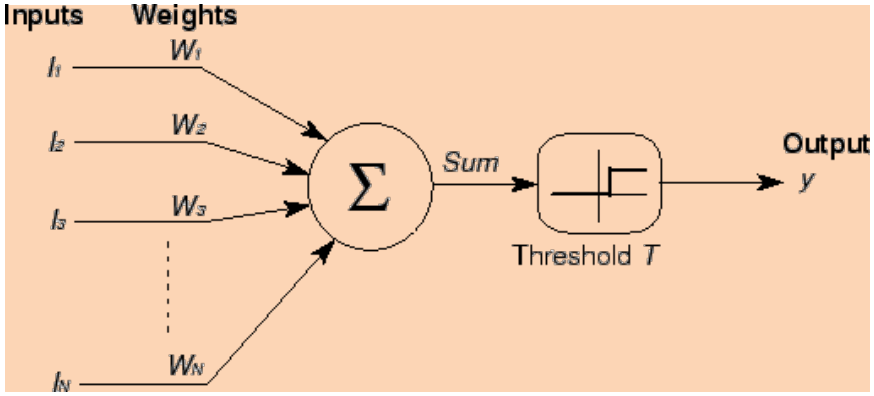
ALVINN: autonomous land vehicle in a NN

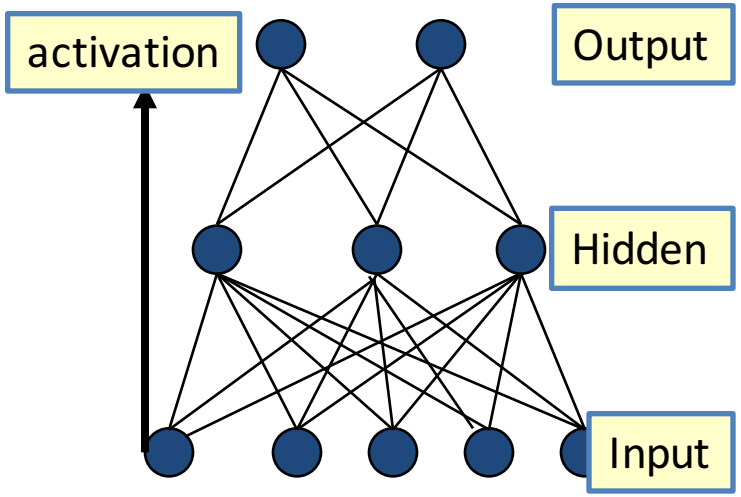




Basic Units in Multi-Layer NN

- 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)

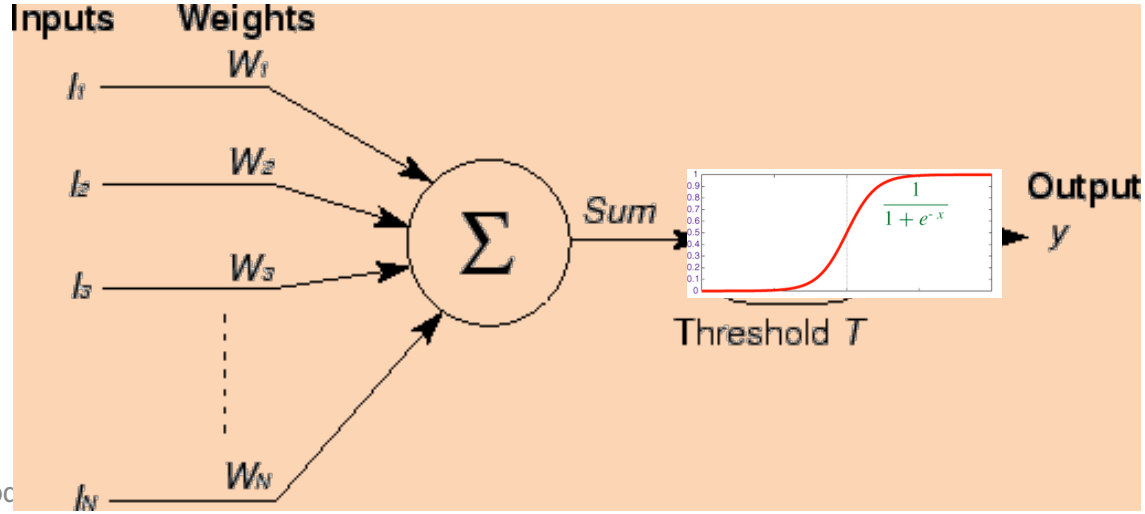




Basic Units in Multi-Layer NN

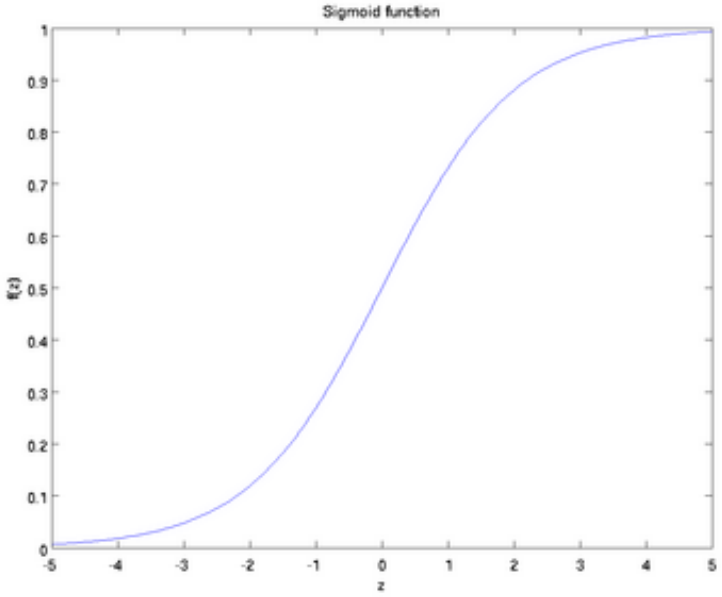
- Basic element: **sigmoid unit**
  - Input to a unit *j* is defined as: $\Sigma w_{ij} x_i$
  - Output is defined as : $\sigma (\Sigma w_{ij} x_i)$
    - σ is simply the logistic function: $\frac{1}{1+e^{-x}}$

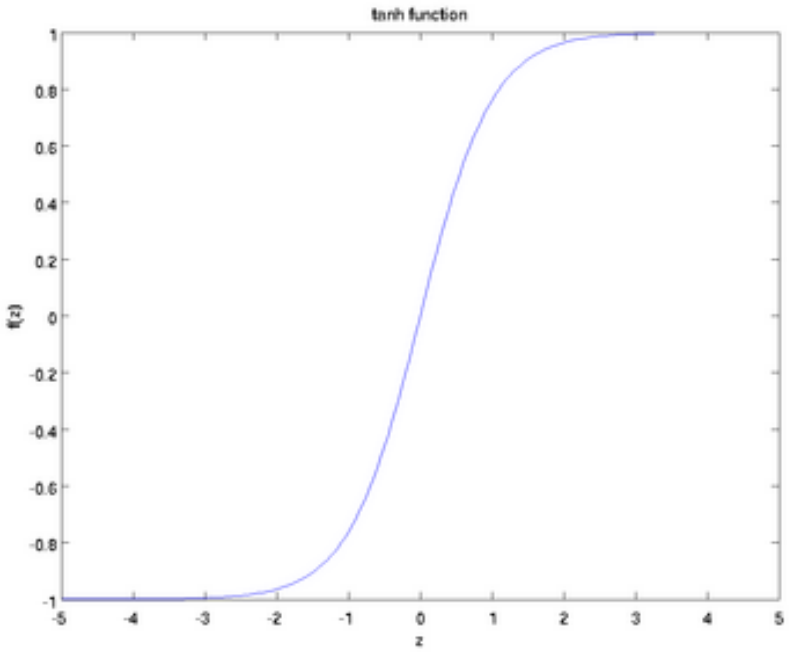
Note: similar to previous algorithms, We encode the bias/threshold, as a "fake" Feature that is always active



Basic Units in Multi-Layer NN

- Basic element: **sigmoid unit**
  - You can also replace the logistic function with other smooth activation functions





Widrow-Hoff Rule (LMS)

- Incremental update rule, providing an approximation to the goal

$$Err(\vec{w}^{(j)}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

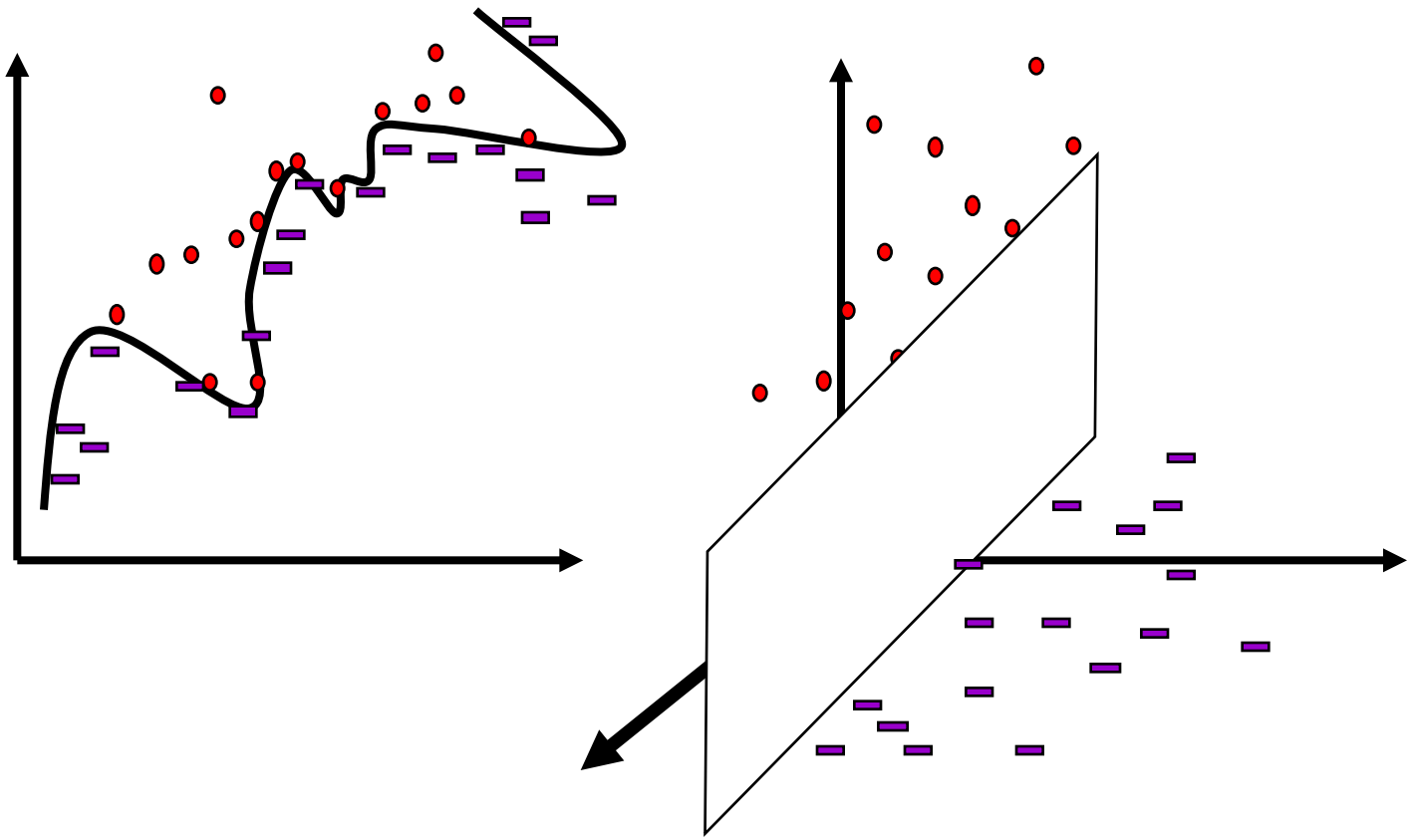
- Where: $o_d = \sum_i w_{ij}.x_i = \vec{w}^{(j)}.\vec{x}$

- And t_d is the target output for example d

Basic Units in Multi-Layer NN

- **Key issue**: limited expressivity!
  - Minsky and Papert (1969) published an influential books showing what cannot be learned using perceptron
- These observation discouraged research on NN for several years
- But.. we really like linear functions!
- How did we deal with these issues so far?

Basic Units in Multi-Layer NN



In fact , Rosenblatt (1959) asked: *"What pattern recognition problems can be transformed so as to become linearly separable"*

Multi Layer NN

- Another approach for increasing expressivity:
  Stacking multiple sigmoid units to form a network
- Compute the output of the network using a 'feed-forward' computation
- Learn the parameters of the network using the backpropagation algorithm
- Any Boolean function can be represented using a two layer network
- Any bounded continuous function can be approximated using a two layer network

Multi Layer NN: forward computation

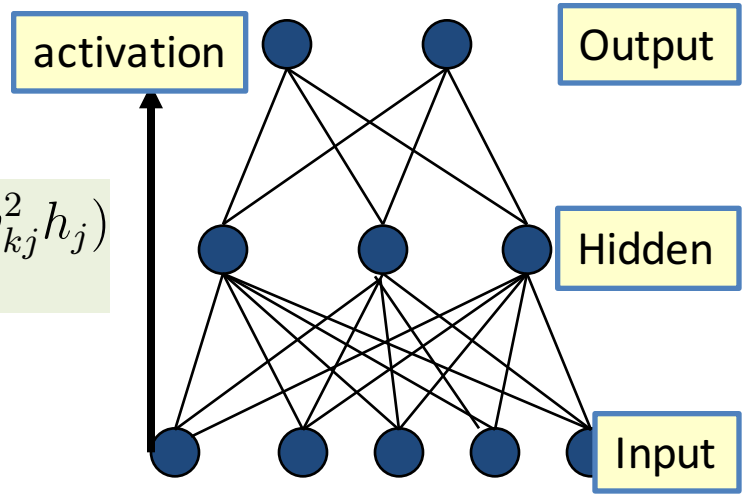
- Observe an input vector x
- *Push x through the network:*
  - For each **hidden unit** compute the activation value

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

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

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

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



- Write a linear function that represent an AND function over two Boolean variables
- Can you write a linear function to represent XOR?
- Describe a NN that represents the XOR function

Our old nemesis – **XOR**!

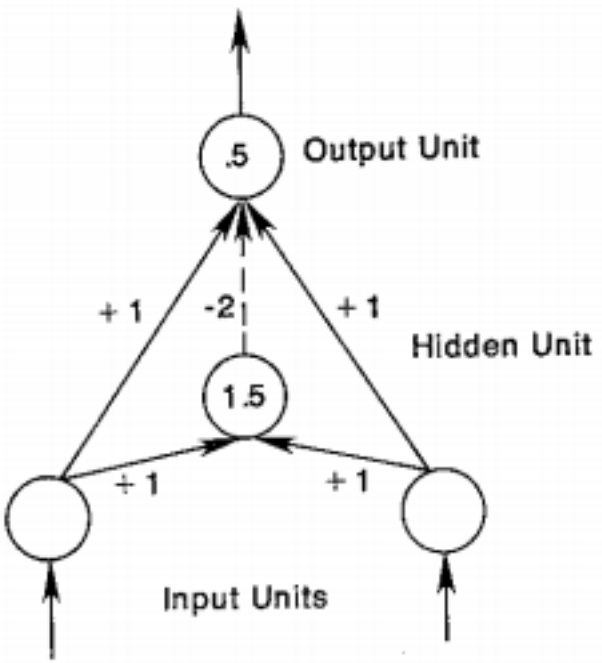


FIGURE 2. A simple XOR network with one hidden unit. See text for explanation.

Learning internal representations by error propagation
D. E. Rumelhart G. E. Hinton R. J. Williams. 1986

Let's build our own NN!

- So far we have seen an architecture for predicting a word, given it's context.
  - How many input units?
  - How many hidden layers?
  - How many output units?

- **What was the point of the hidden units for this classification problem?**

Let's build our own NN!

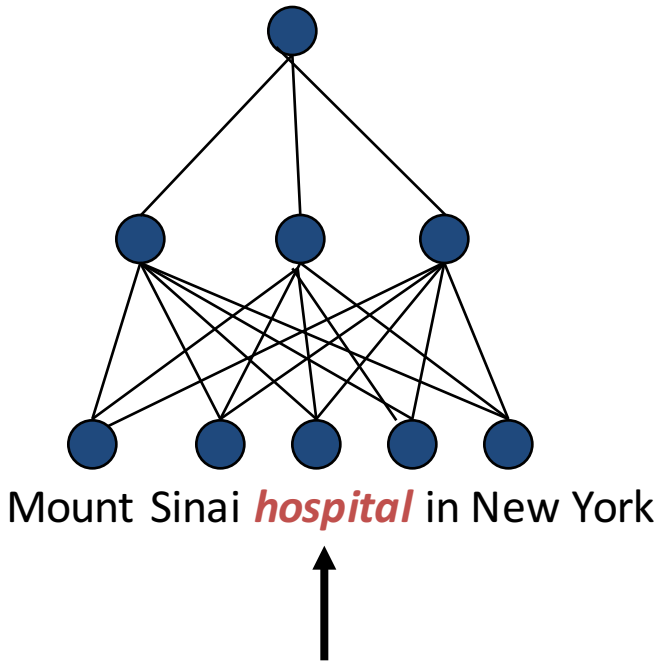
- Let's revisit a familiar problem, and design a NN
  - **Named Entity Recognition**
    - **Binary case**: given a sentence, decide which words are NE and which are not
    - **Multi-class case**: decide if a word is a Loc, Per, Org, None

- **What is the right architecture?**

NN for NER

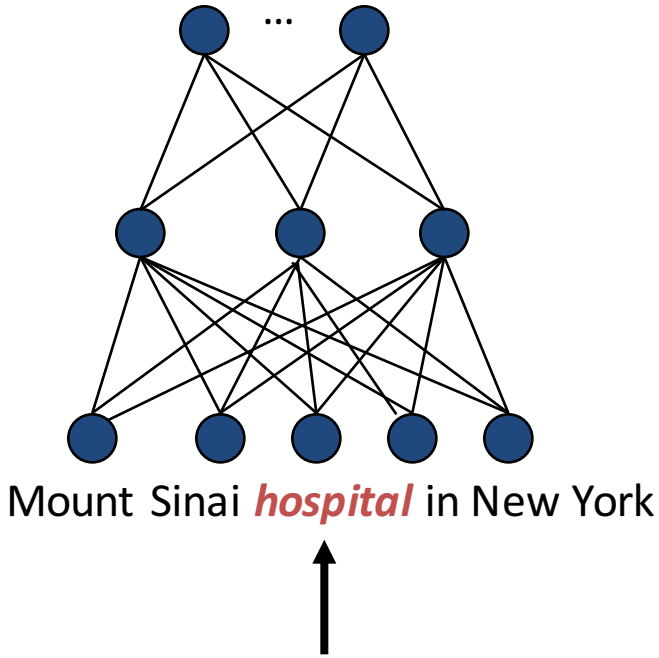
Binary case:

Single output unit, can be interpreted as a threshold function



NN for NER

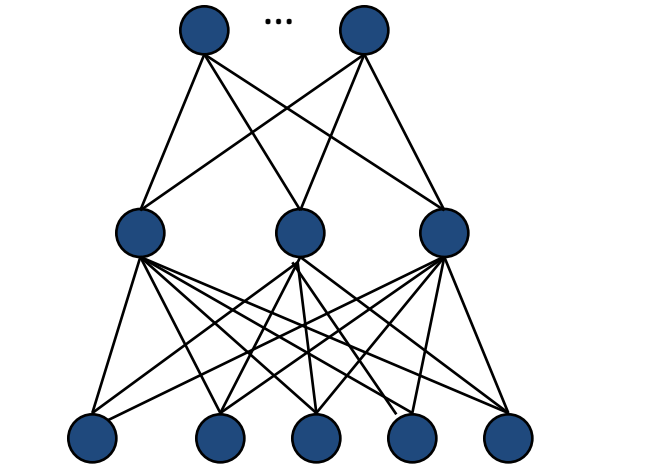
Multiclass case: winner-takes-all



- Recall another old nemesis of linear models –
  - NEGATION!
  - *"I like rice but not beans" "I like beans but not rice"*
- Why would BoW models have difficulties?
  - What was the solution for linear models?
  - What was the "price" we had to pay? (I want a number!)
- How would a 1-hidden layer network help?
  - What is the price we pay now? (I want a number!)

NN for NER

Multiclass case: winner-takes-all

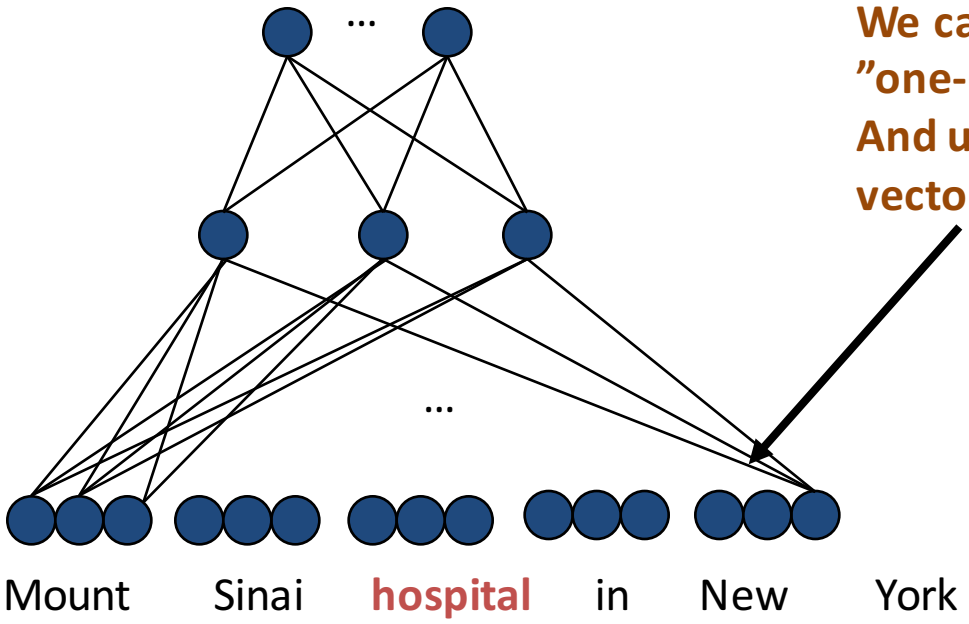


So far we assumed BoW features (binary, frequency) , but NN lend themselves nicely to continuous representations of input objects

Example – Word Embedding

NN for NER

Multiclass case: winner-takes-all



We can also replace the ”one-hot” representation, And use pre-trained word vectors!

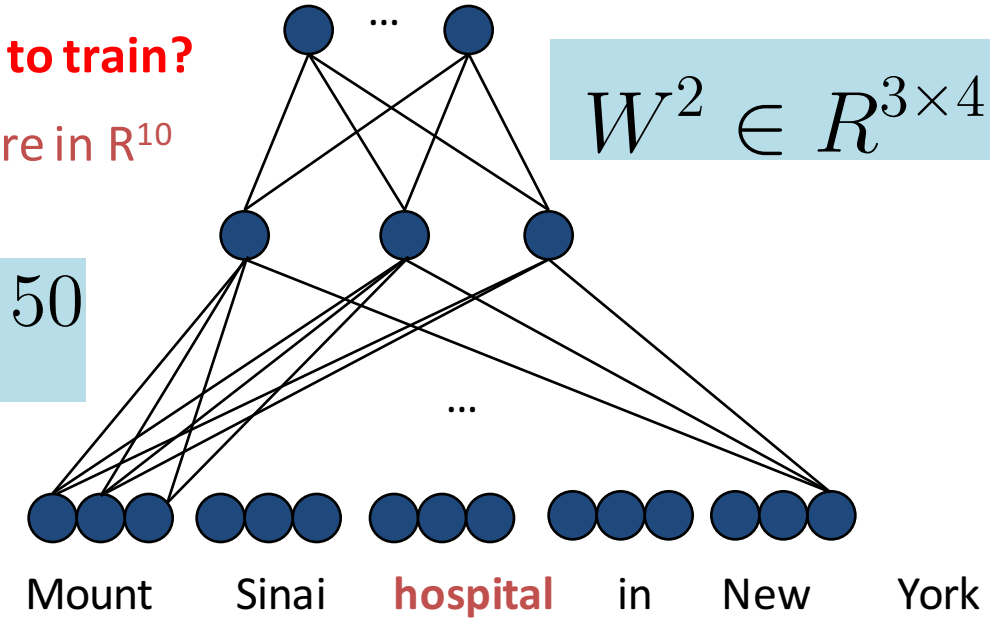
Is the center word a Named Entity?

NN for NER

How many parameters do we have to train?

Let's assume our word embedding are in R^{10}

$$W^1 \in R^{3\times 50}$$



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

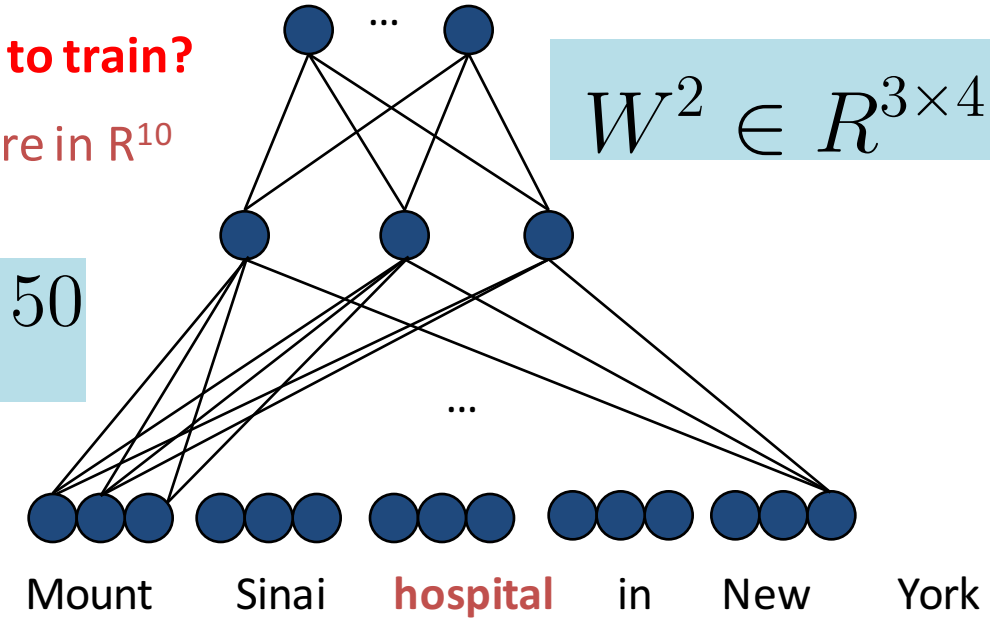
$$x \in R^{50}$$

NN for NER

How many parameters do we have to train?

Let's assume our word embedding are in R^{10}

$$W^1 \in R^{3\times 50}$$



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

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

Forward computation:

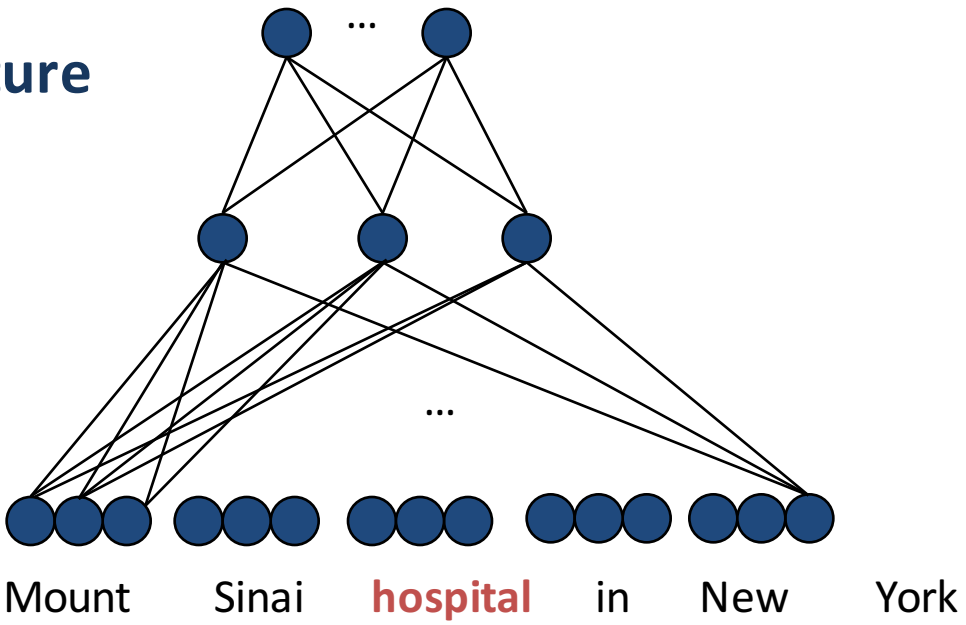
$$h_j = \sigma(t_j) = \sigma(\sum_i w^1_{ji}\ x_i)$$

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

NN for NER

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

Why not just use the word vectors directly?



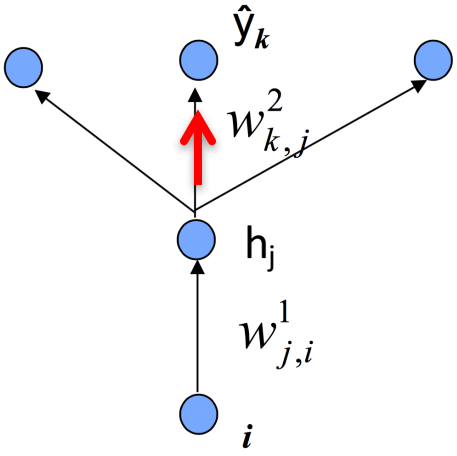
What do we get by adding an extra layer?

Training ML NN: Backpropagation

Backpropagation = **Gradient descent** + **chain rule** (*applied to the architecture of the network*)

We'll compute the gradient wrt each of the models parameters:

$$\frac{\partial J}{\partial w_{kj}^2} = -2\sum_{k'}(y_{k'} - \hat{y}_{k'})(\partial \hat{y}_{k'}) = -2(y_k - \hat{y}_k)\sigma'(s_k)h_j$$



Note: this is just logistic mean squared error regression, treating 'h' as input features

Loss Function:

$$J_i(\mathbf{w}) = \sum_k (y^i - \hat{y}_k^i)^2$$

Output Layer :

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

Hidden Layer :

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

Recall:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

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

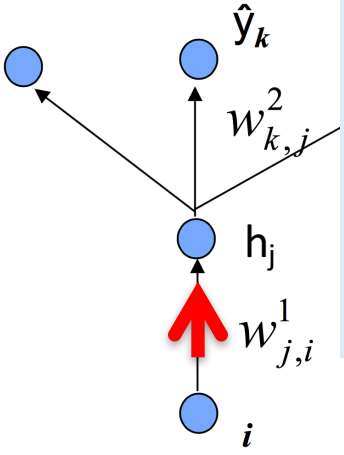
Training ML NN: Backpropagation

Backpropagation = **Gradient descent** + **chain rule** (*applied to the architecture of the network*)

We'll compute the gradient wrt each of the models parameters:

Now, let's look at weights at the first layer

$$\frac{\partial J}{\partial w^2_{kj}} = -2\sum_{k'}(y_{k'} - \hat{y}_{k'})(\partial \hat{y}_{k'}) = -2(y_k - \hat{y}_k)\sigma'(s_k)h_j$$



$$\frac{\partial J}{\partial w^1_{ji}} = \sum_{k'} -2(y_{k'} - \hat{y}_{k'})(\partial \hat{y}_{k'})$$

$$= \sum_{k'} -2(y_{k'} - \hat{y}_{k'})\ \sigma'(s_k)\ w_{kj}\ \partial h_j$$

$$= \sum_{k'} -2(y_{k'} - \hat{y}_{k'})\ \sigma'(s_k)\ w_{kj}\ \sigma'(t_j)\ x_i$$

Loss Function:

$$J_i(\mathbf{w}) = \sum_k (y^i - \hat{y}^i_k)^2$$

Output Layer :

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

Hidden Layer :

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

Recall:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

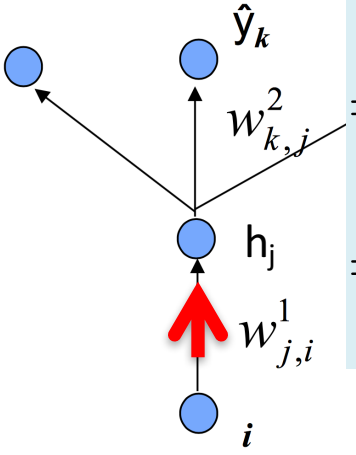
Chain rule: $\frac{\partial}{\partial v_{w_i}} f(z(v_{wi})) = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial v_{w_i}}$

Training ML NN: Backpropagation

Backpropagation = **Gradient descent** + **chain rule** (*applied to the architecture of the network*)

Note that we can reuse ("back-propagate") the computation

$$\frac{\partial J}{\partial w_{kj}^2} = -2\sum_{k'}(y_{k'} - \hat{y}_{k'})(\partial \hat{y}_{k'}) = -2(y_k - \hat{y}_k)\sigma'(s_k)h_j$$



$$\frac{\partial J}{\partial w_{ji}^1} = \sum_{k'} -2(y_{k'} - \hat{y}_{k'})(\partial \hat{y}_{k'})$$

$$= \sum_{k'} -2(y_{k'} - \hat{y}_{k'})\ \sigma'(s_k)\ w_{kj}\ \partial h_j$$

$$= \sum_{k'} -2(y_k - \hat{y}_k)\sigma'(s_k)\ w_{kj}\ \sigma'(t_j)\ x_i$$

Loss Function:

$$J_i(\mathbf{w}) = \sum_k (y^i - \hat{y}_k^i)^2$$

Output Layer :

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

Hidden Layer :

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

Recall:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

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

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 (tens of thousands) 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.

Over-training Prevention

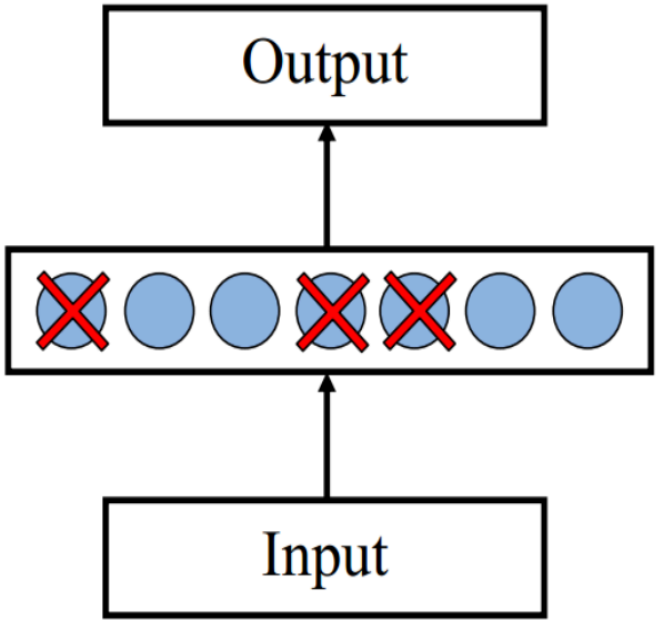
- Running too many epochs may over-train the network and result in over-fitting
  - Keep a hold-out validation set and test accuracy after every epoch
  - Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.
  - *Why not 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

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: change Error function to include a regularizer

Dropout Training

- Proposed by (Hinton et-al 2012)



Prevent feature co-adaptation

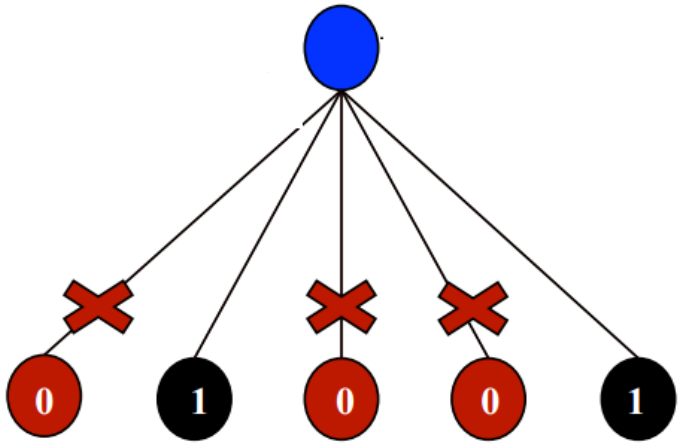
Encourage "independent contributions"

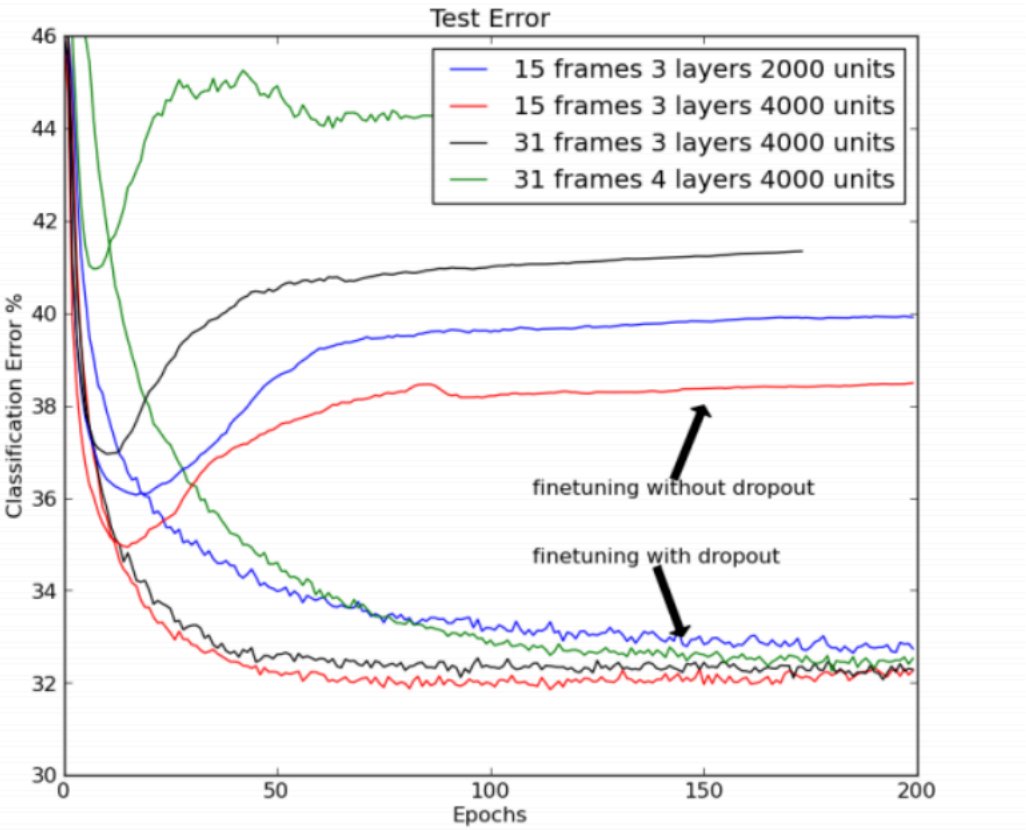
From different features

- 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!



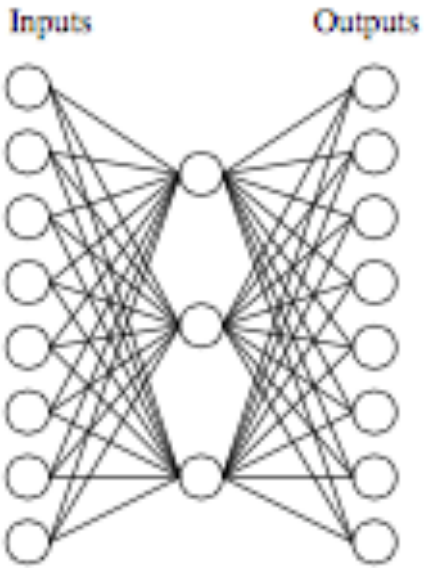


- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

Learning Hidden Layer Representation

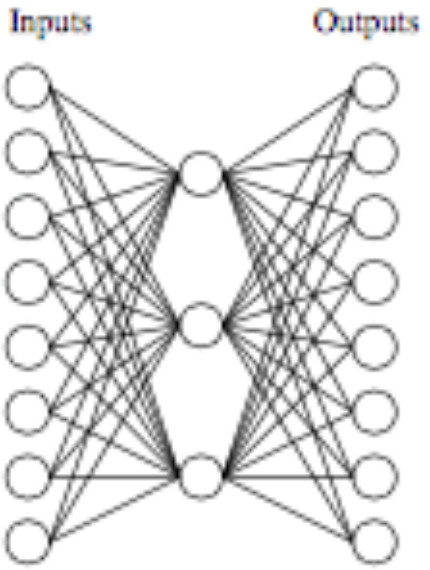
- **NN can be seen as a way to learn a feature representation**
  - Weight-tuning sets weights that define hidden units representation most effective at minimizing the error
- Backpropagation can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.
- *Trained hidden units can be seen as newly constructed features that re-represent the examples so that they are linearly separable*

Auto-associative Network



Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Auto-associative Network

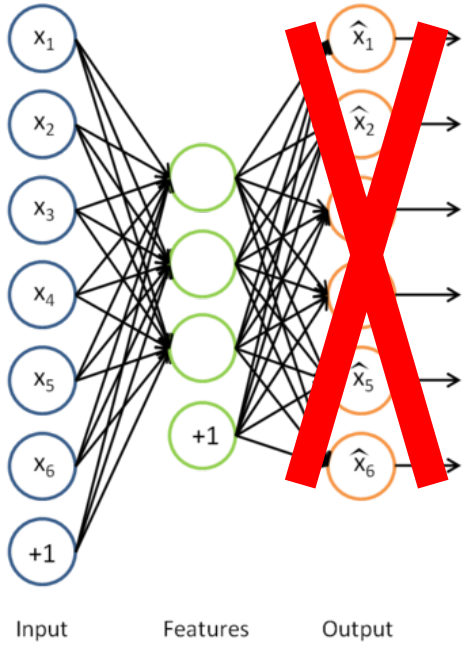


Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Sparse Auto-Encoder

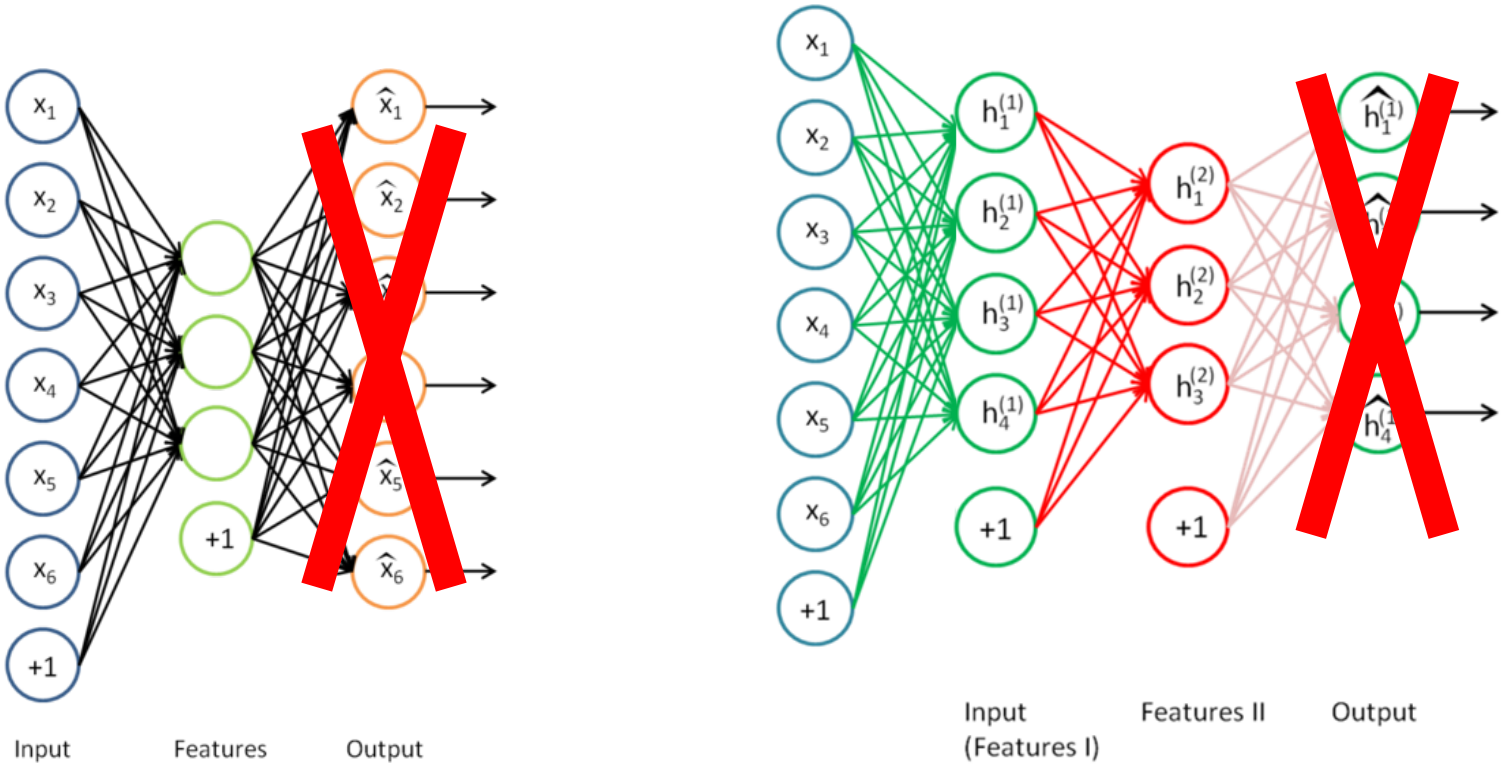
Goal: perfect reconstruction of the input vector x, by the output x'

- **Simple approach**:
  - Minimize the error function l(h(x),x)
  - After optimization:
    - Drop the reconstruction layer



Stacking Auto Encoder

- Add a new layer, and a reconstruction layer for it.
- Repeat.



10,000 feet view

- Neural networks are an extremely flexible way to define complex prediction models.
  - **Simple update rule:** propagate the error on the architecture of the network (essentially DAG).
  - All deep learning models share this property, **just different DAGs**!
- **Key issues**:
  - Preventing over fitting
  - Representation learning, pre-training with minimal supervision

10,000 feet view

- So far, we looked at simple classification problems.
  - Assume a word window, that provides fixed sized inputs.
    - In case you "run out of input" – zero padding.
- What can you do if the size of the input is not fixed?
  - Some notion of compositionality is needed
  - Simplest approach: sum up word vectors
    - Document structure is lost.. **Can we do better?**