# Machine Learning

# Intro to NN and Representation learning

Dan Goldwasser

dgoldwas@purdue.edu

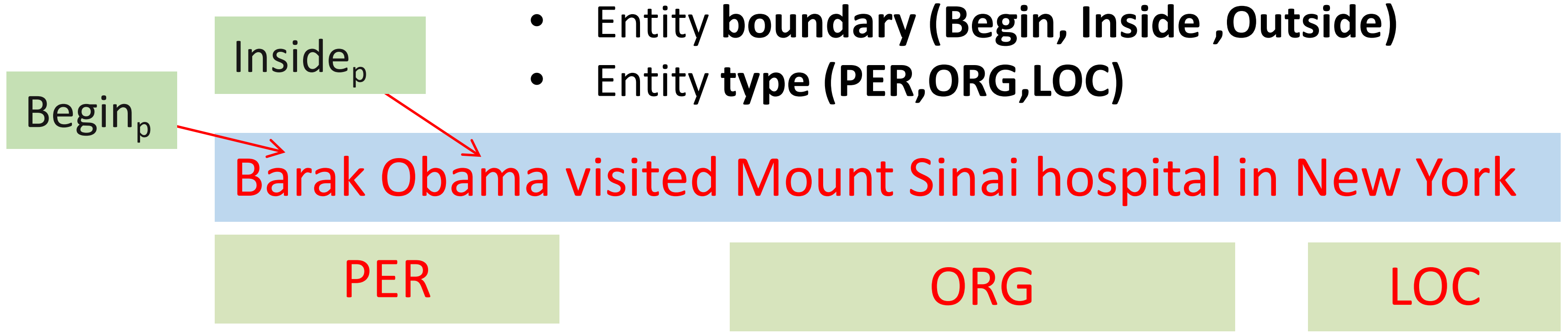
Some slides based on M. Gormley
NN Example slides

# Dealing with Structures

- Structured prediction – dealing with multiple decisions at the same time. Modeling the interactions between decisions is the key challenge.

Conceptually two considerations:

- Entity **boundary (Begin, Inside ,Outside)**
- Entity **type (PER,ORG,LOC)**



**Many of the predictions are context dependent:** e.g., Mount Sinai is a LOC, while Mount Sinai hospital is an ORG.

**How can you capture it using the machinery we currently have? At what cost?**

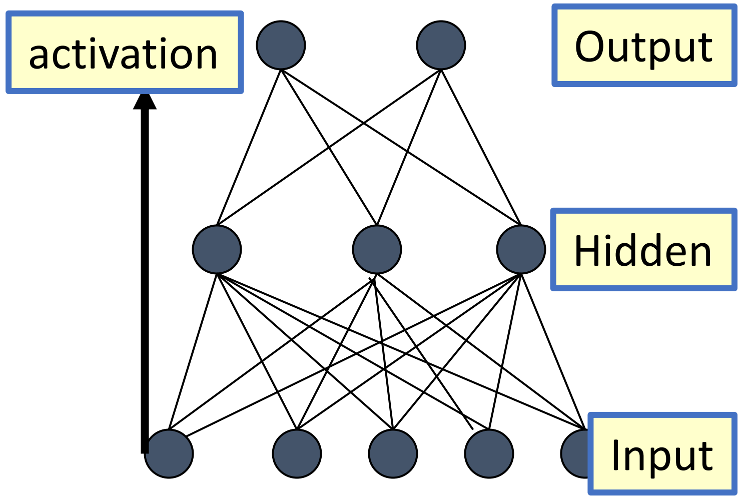
# Multi Layer NN: forward computation

- 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

  - **Prediction:** $$\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w^2_{kj} h_j)$$
    - **Categories**: winner take all
    - **Vector**: take all output values
    - **Binary outputs**: Round to nearest 0-1 value



## 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?

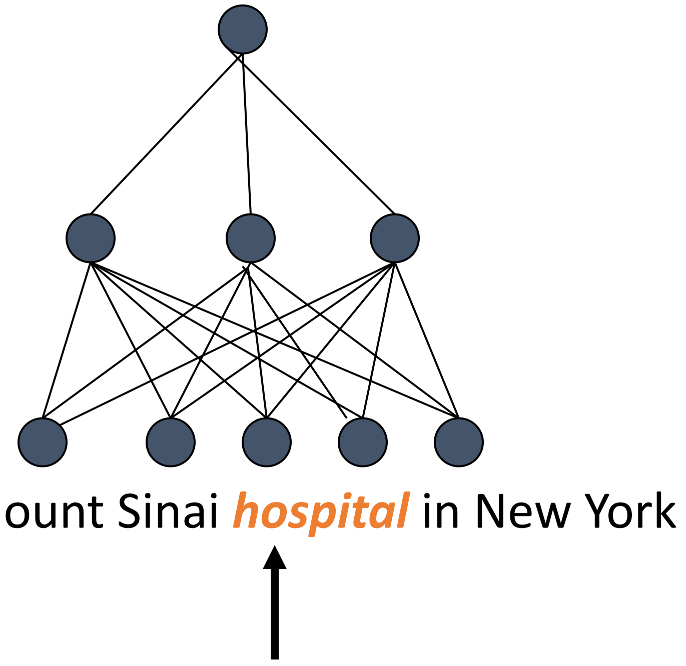
# Let’s build our own NN!

- 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

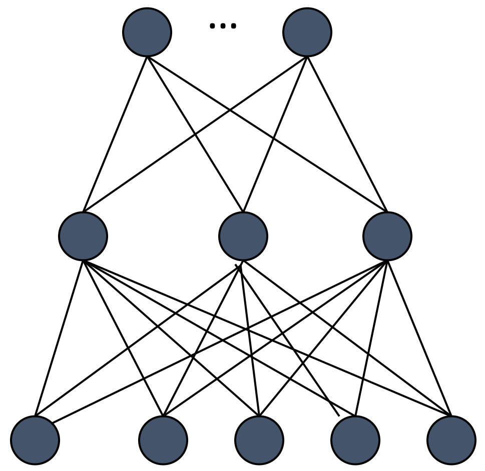
**Binary case**:
Single output unit, can be interpreted as a threshold function



Is the center word a Named Entity?

# NN for NER

**Multiclass case:** winner-takes-all



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

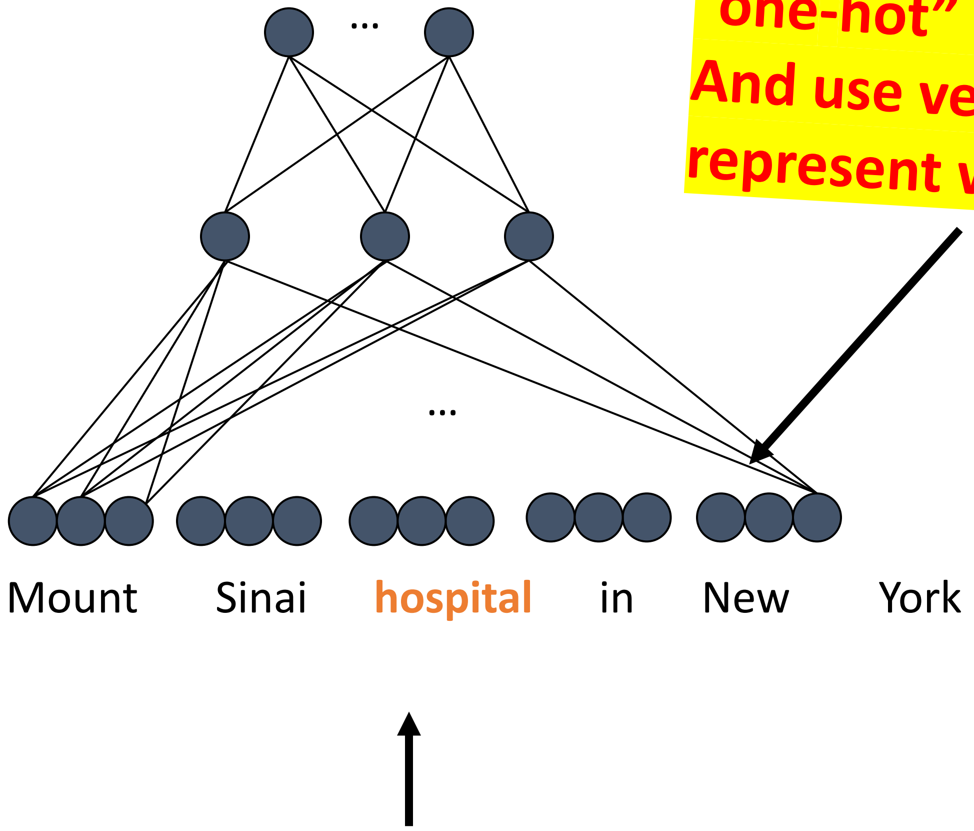
↑

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 (short version)

- **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?**

# NN for NER

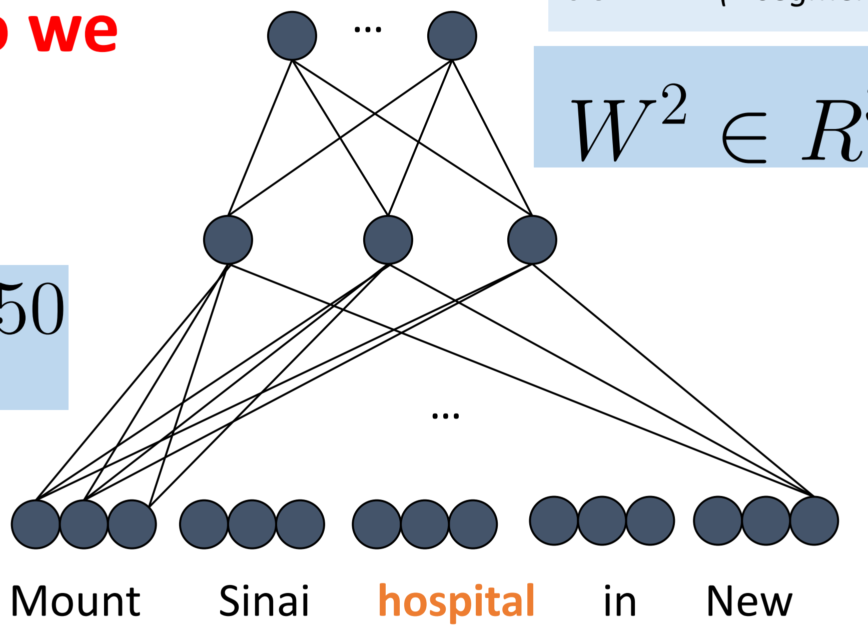
**How many parameters do we have to train?**

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

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

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

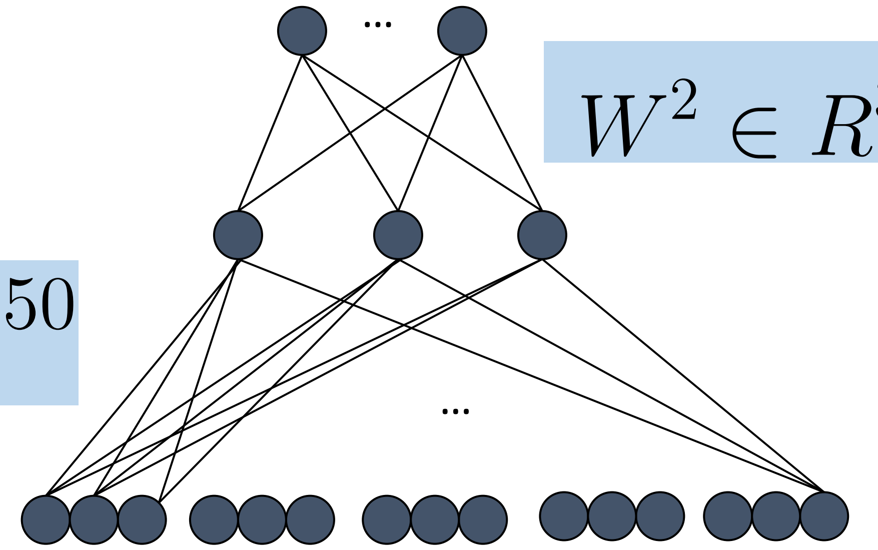
*Let's assume our word embedding are in $R^{50}$*



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

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

# NN for NER



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

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

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

**Forward computation:**

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

**Hidden layer**

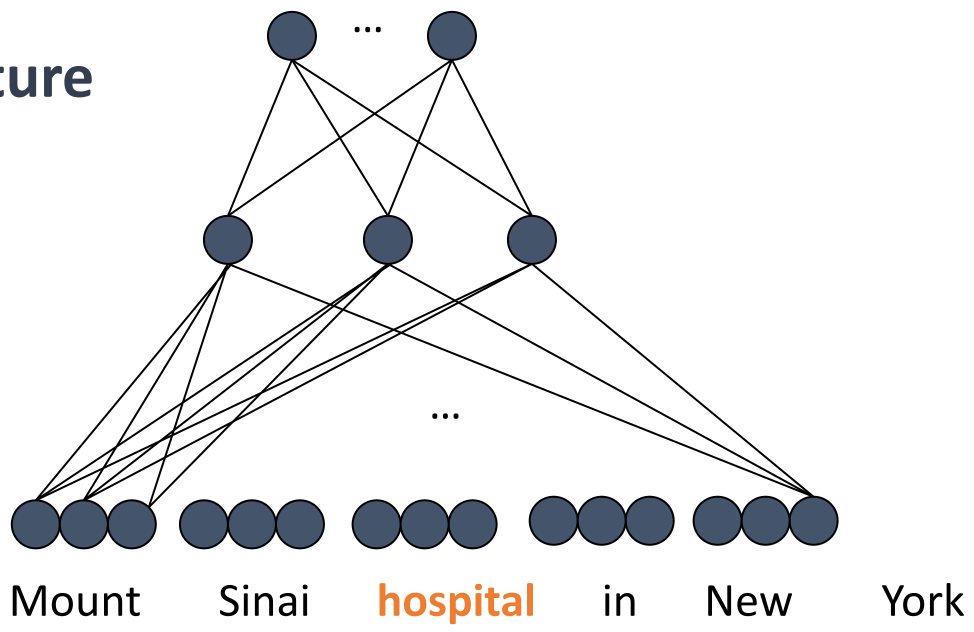
$$\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w^2_{kj} h_j)$$

**output layer**

# NN for NER

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

*Why not just use the word vectors directly?*



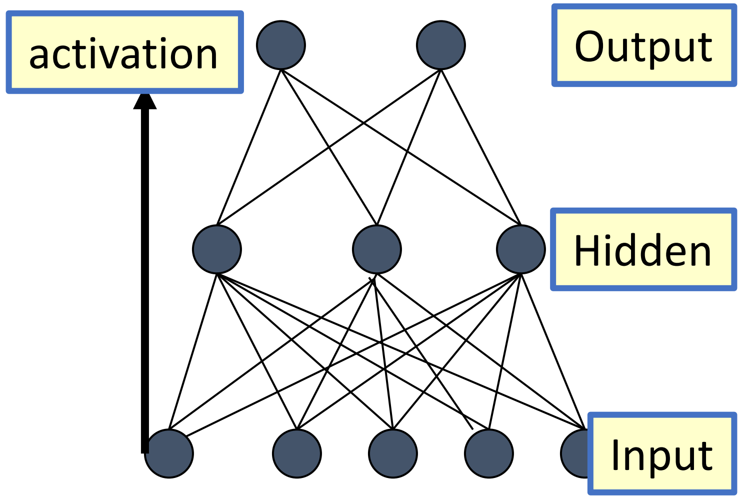
**What do we get by adding an *extra* layer?**

## Training Neural Nets

- Learning so far – derive the gradient of the loss function, and use GD/SGD to optimize.
- Neural nets learn in the same way, however the process has to account for structure of the network.
- Mathematically – NN correspond to multiple steps of **function composition.**

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

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



# 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**

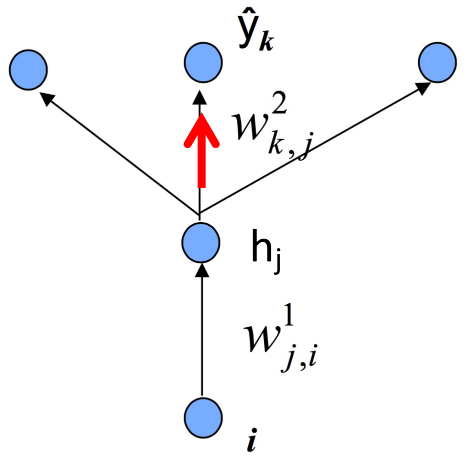
$$z = 3y$$

$$y = x^2$$

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

## Training ML NN: Backpropagation

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



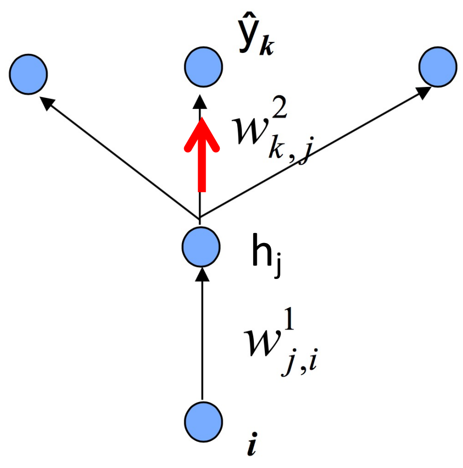
**Chain rule:**

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

# Training ML NN: Backpropagation

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

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



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

Loss Function:

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

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**:

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

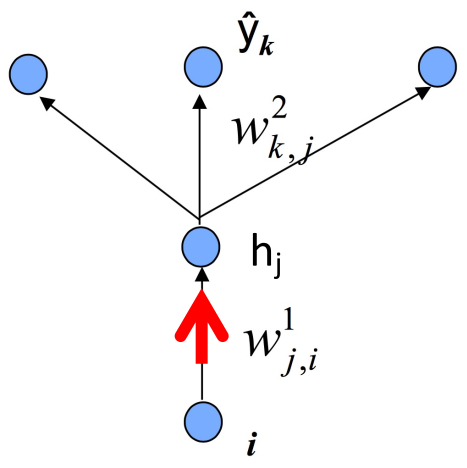
**Chain rule:**
$$\frac{\partial}{\partial v_{w_i}} f(z(v_{wi})) = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial v_{w_i}}$$

# Training ML NN: Backpropagation

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

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

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



$$\frac{\partial J}{\partial w_{ji}^1} = \sum_{k'} -2(y_{k'} - \hat{y}_{k'})(\partial \hat{y}_{k'})$$
$$= \sum_{k'} -2(y_{k'} - \hat{y}_{k'})\ \sigma'(s_k)\ w_{kj}\ \partial h_j$$
$$= \sum_{k'} -2(y_{k'} - \hat{y}_{k'})\ \sigma'(s_k)\ w_{kj}\ \sigma(t_j)\ x_i$$

Loss Function:

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

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

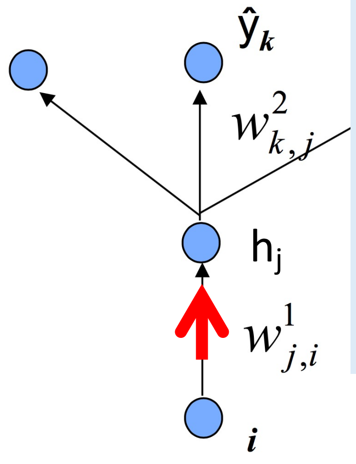
*Note that we can reuse ("back-propagate") the computation*

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

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

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

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



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 :

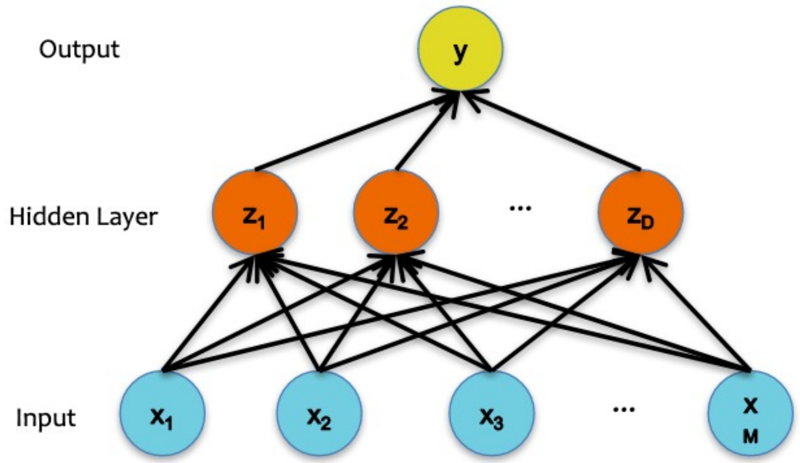
$$h_j = \sigma(t_j) = \sigma(\sum_i w^1_{ji}\ x_i)$$

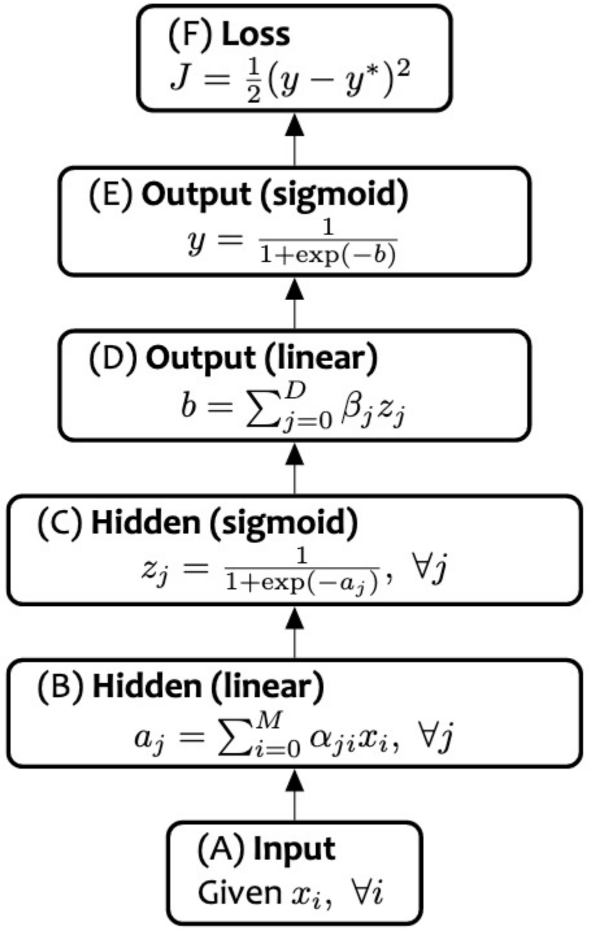
Recall:

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

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

# Computation Graph




(F) **Loss**
$J = \frac{1}{2}(y - y^*)^2$
(E) **Output (sigmoid)**
$y = \frac{1}{1+\exp(-b)}$
(D) **Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$
(C) **Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$
(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$
(A) **Input**
Given $x_i, \ \forall i$

# Backpropagation

Forward Computation

1. Write an **algorithm** for evaluating the function y = f(**x**). The algorithm defines a **directed acyclic graph,** where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable $u_i$ with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

Backward Computation

1. **Initialize** all partial derivatives $dy/du_j$ to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1, \ldots, v_N)$
   a. We already know $dy/du_i$
   b. Increment $dy/dv_j$ by $(dy/du_i)(du_i/dv_j)$
      (Choice of algorithm ensures computing $(du_i/dv_j)$ is easy)

**Return** partial derivatives $dy/du_i$ for all variables

# Backpropagation

| | Forward | Backward |
|---|---|---|
| Module 5 | $J = y^* \log y + (1 - y^*) \log(1 - y)$ | $\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$ |
| Module 4 | $y = \frac{1}{1 + \exp(-b)}$ | $\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$ |
| Module 3 | $b = \sum_{j=0}^{D} \beta_j z_j$ | $\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$ |
| | | $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$ |
| Module 2 | $z_j = \frac{1}{1 + \exp(-a_j)}$ | $\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$ |
| Module 1 | $a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$ | $\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$ |
| | | $\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$ |

# 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.

# Multi Layer NN: forward computation

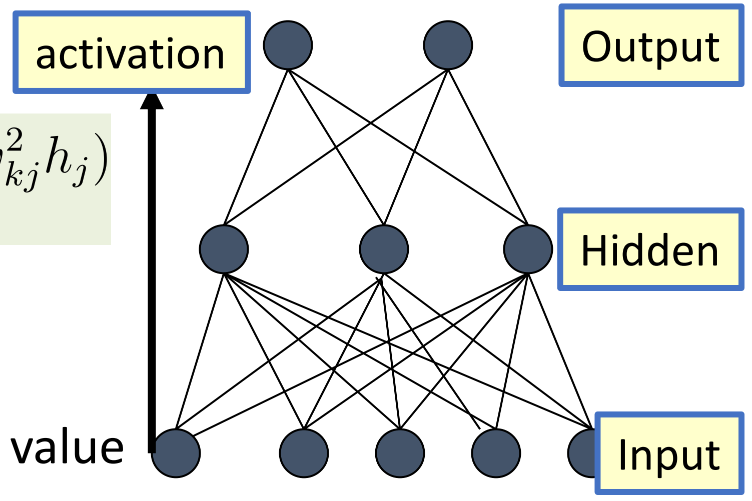
- Observe an input vector x
- *Push x through the network:*
  - For each **hidden unit** compute the activation value

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

  - 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



## 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

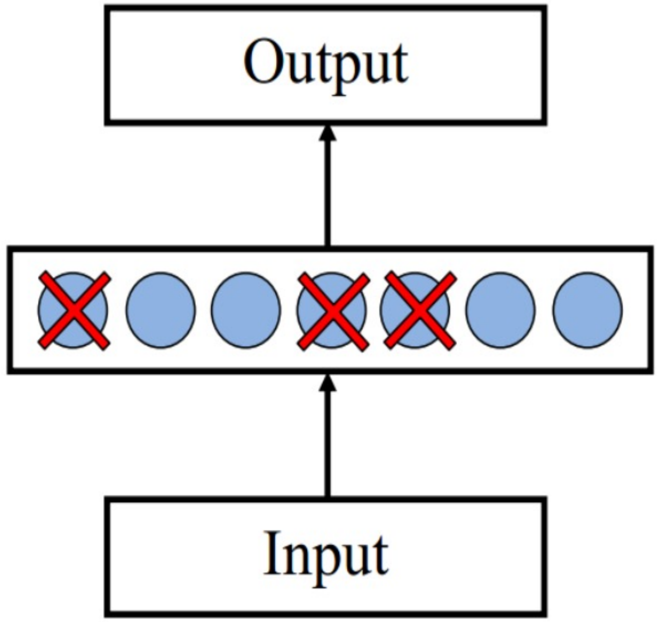
- Running too many epochs may over-train the network and result in over-fitting
  - Keep a hold-out validation set and test accuracy after every epoch
  - Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.
  - ***Why 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: **use a regularizer**

## Dropout Training

- Proposed by (Hinton et-al 2012)



**Prevent feature co-adaptation**

*Encourage "independent contributions"*

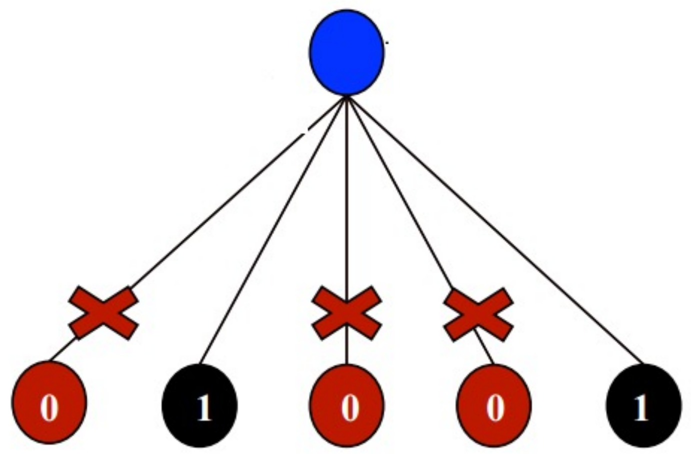
*From different features*

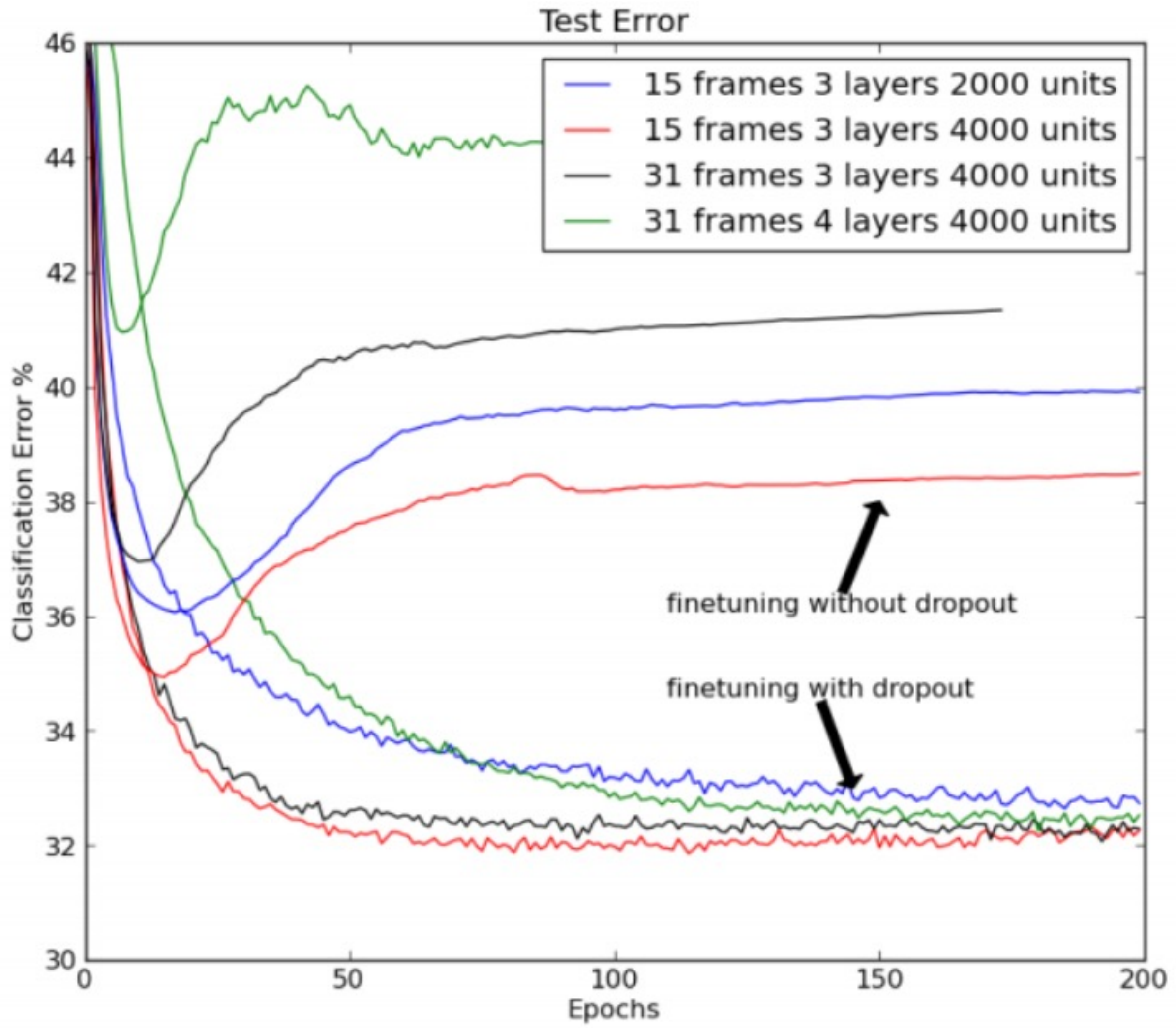
- At each training step, decide whether to delete one hidden unit with some probability p

# 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!

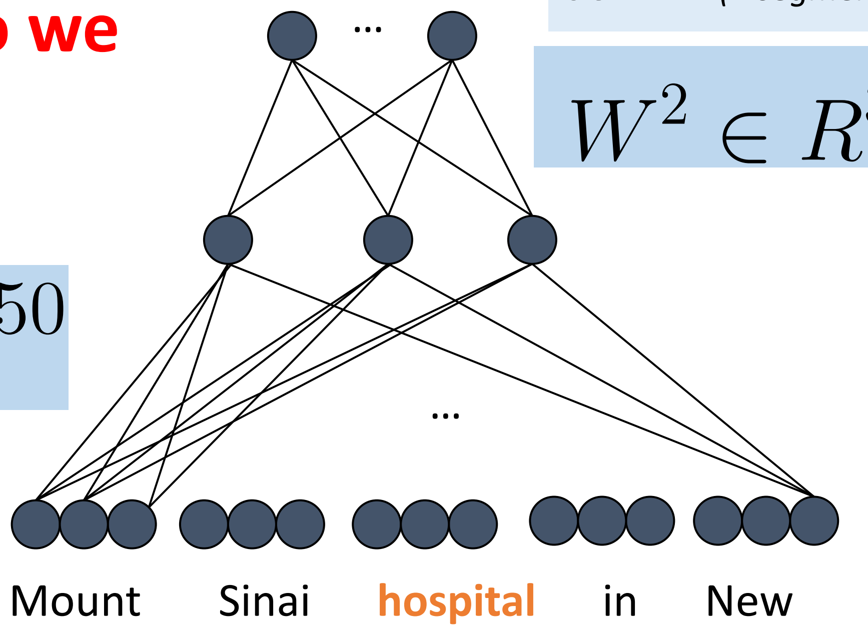




- Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

# Let’s recall our NER problem

**How many parameters do we have to train?**



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

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

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

*Let’s assume our word embedding are in R*$^{50}$

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

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

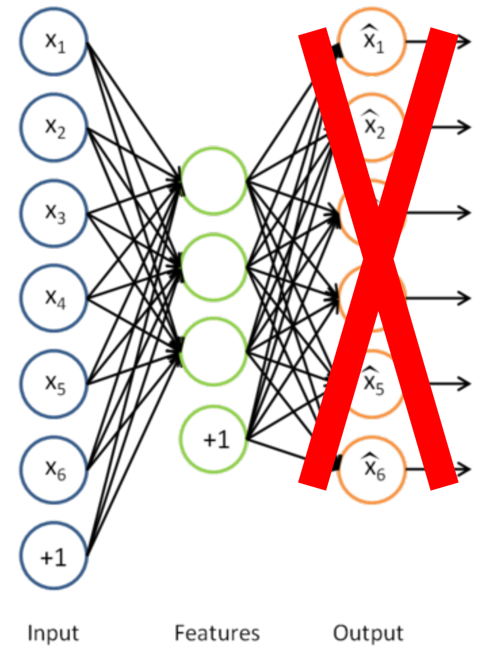
## Learning Hidden Layer Representation

- **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*

# Sparse Auto-Encoder

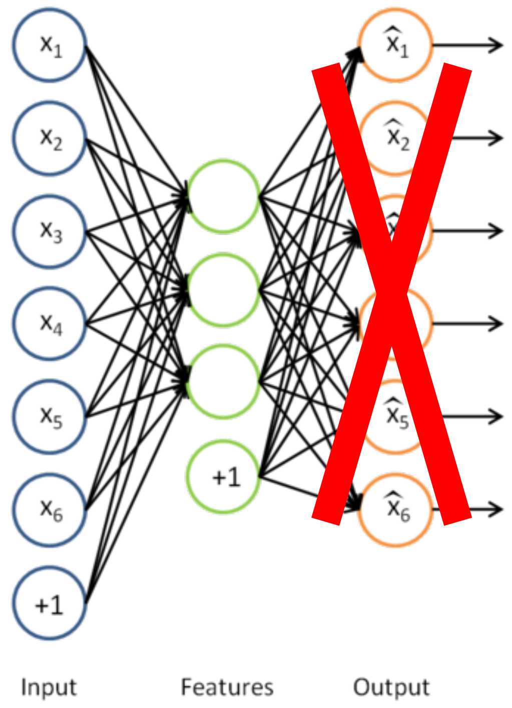
**Goal**: perfect reconstruction of the input vector x, by the output x'

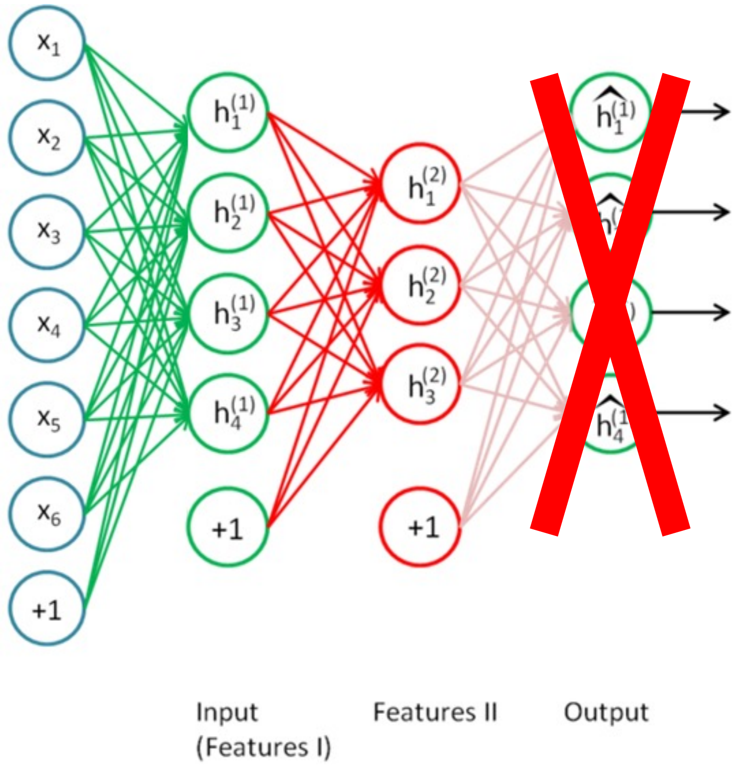
- **Simple approach**:
  - Minimize the error function l(h(x),x)
  - After optimization:
    - Drop the reconstruction layer



# Stacking Auto Encoder

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





## Representation Learning

- Learning Deep Structured Semantic Models for Web Search using Clickthrough Data. Huang et-al. CIKM'13
- **IR goal:** *find relevant documents given a short query*
  - **Challenge:** *how can a short query capture the intent and information need of the user?*
- **Settings**: *Massive clickthrough log data, consisting of documents and search queries.*

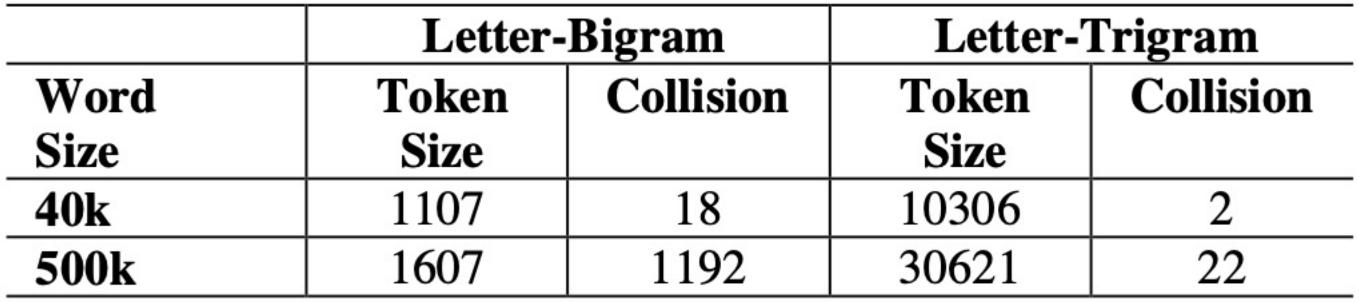
## Paper Review

- Our goal is to learn a representation for queries such that they match documents selected by users.
- **What is the simplest way to do that given the data?**
- Given a document and query, are the two a good match or not?
- How should we encode each element?

Learning Deep Structured Semantic Models for Web Search using Clickthrough Data. Huang et-al. CIKM'13

# Word Hashing

- One of the key challenges: map long documents and queries to the same space.
- Key idea: break words into ”bag of ngrams”
  - e.g. *#good#* ➔ *{ #go, goo, ood, od# }*

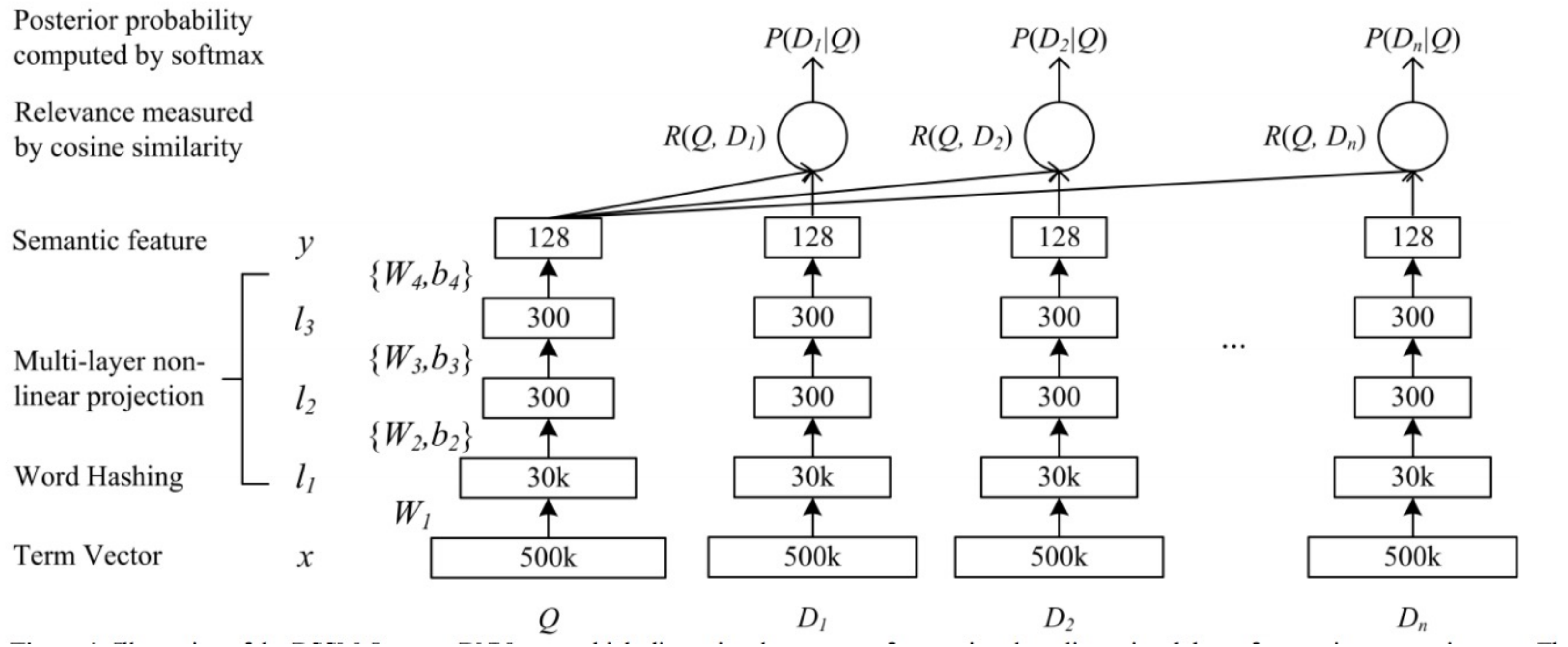


|  | Letter-Bigram |  | Letter-Trigram |  |
|---|---|---|---|---|
| **Word Size** | **Token Size** | **Collision** | **Token Size** | **Collision** |
| **40k** | 1107 | 18 | 10306 | 2 |
| **500k** | 1607 | 1192 | 30621 | 22 |

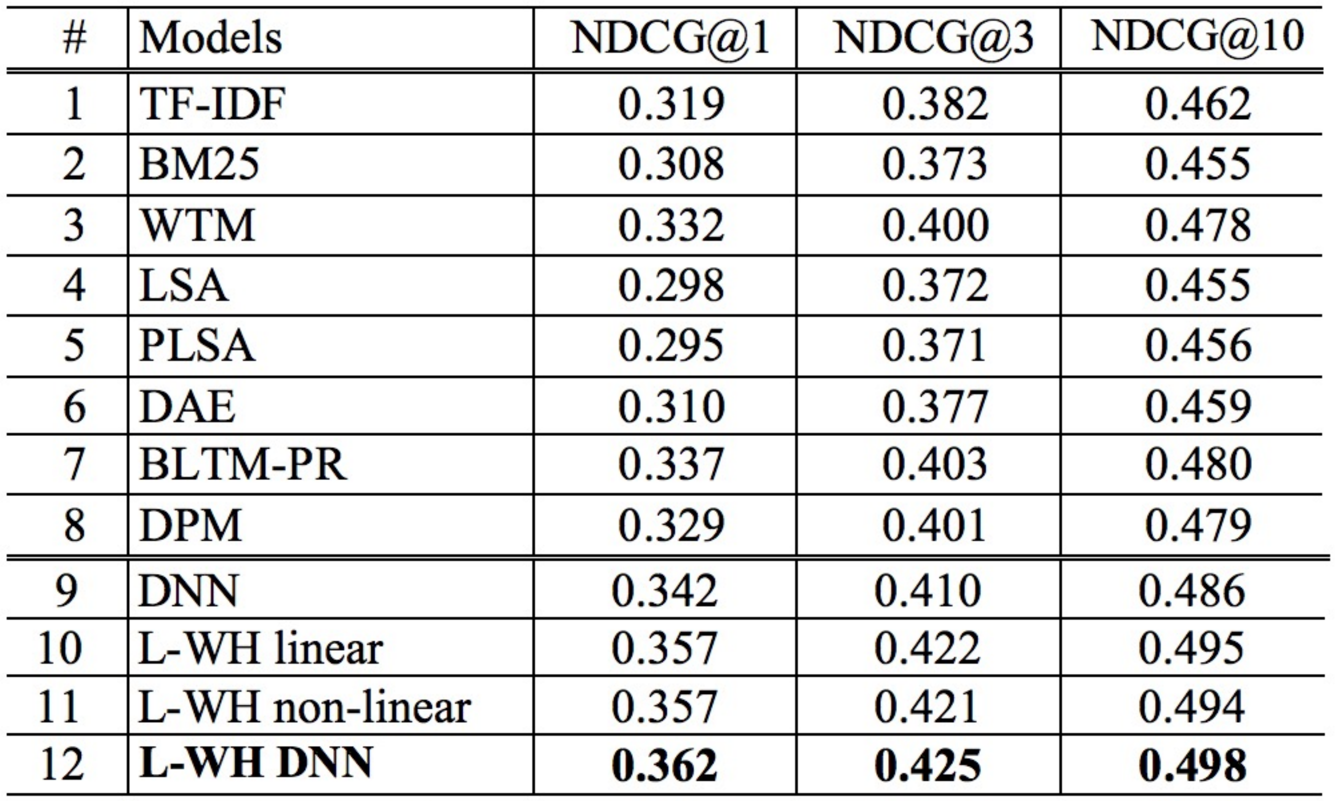
**Table 1:** Word hashing token size and collision numbers as a function of the vocabulary size and the type of letter ngrams.

## Objective Function

- Define the relationship between two document vectors: $R(Q, D) = \text{cosine}(y_Q, y_D) = \frac{{y_Q}^T y_D}{\|y_Q\| \|y_D\|}$
- Objective: maximize the prob. of relevant docs, give query: $P(D|Q) = \frac{\exp(\gamma R(Q, D))}{\sum_{D' \in \mathbf{D}} \exp(\gamma R(Q, D'))}$
  - In practice, sample a few negative examples to represent **D**



# Paper Review



| # | Models | NDCG@1 | NDCG@3 | NDCG@10 |
|---|---|---|---|---|
| 1 | TF-IDF | 0.319 | 0.382 | 0.462 |
| 2 | BM25 | 0.308 | 0.373 | 0.455 |
| 3 | WTM | 0.332 | 0.400 | 0.478 |
| 4 | LSA | 0.298 | 0.372 | 0.455 |
| 5 | PLSA | 0.295 | 0.371 | 0.456 |
| 6 | DAE | 0.310 | 0.377 | 0.459 |
| 7 | BLTM-PR | 0.337 | 0.403 | 0.480 |
| 8 | DPM | 0.329 | 0.401 | 0.479 |
| 9 | DNN | 0.342 | 0.410 | 0.486 |
| 10 | L-WH linear | 0.357 | 0.422 | 0.495 |
| 11 | L-WH non-linear | 0.357 | 0.421 | 0.494 |
| 12 | **L-WH DNN** | **0.362** | **0.425** | **0.498** |

**Table 2:** Comparative results with the previous state of the art approaches and various settings of DSSM.

# 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?**