Deep Learning for Natural Language Processing

Cornelia Caragea

Computer Science
University of Illinois at Chicago

Credits for slides: Manning, Socher

Neural Networks and Word Window Classification

Today

Lecture Structure (Part 1):

- Classification review/introduction
- Updating word vectors for classification
- Neural networks introduction
- Word2Vec as neural nets

Lecture Structure (Part 2):

- Named Entity Recognition
- Binary true vs. corrupted word window classification
- Backpropagation

Classification setup and notation

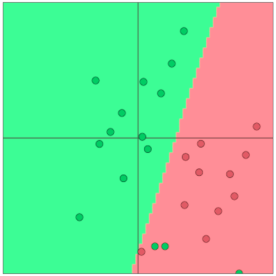
- Generally we have a training dataset consisting of samples

$$\{x_i, y_i\}_{i=1}^{N}$$

- x_i are inputs, e.g. words (indices or vectors!), context windows, sentences, documents, etc.
  - Dimension d
- y_i are labels (one of C classes) we try to predict, for example:
  - classes: sentiment, named entities, buy/sell decision
  - other words

Classification intuition

- Training data: $\{x_i, y_i\}_{i=1}^{N}$
- Simple illustration case: $\longrightarrow$
  - Fixed 2D word vectors to classify
  - Using softmax/logistic regression
  - Linear decision boundary



Visualizations with ConvNetJS by Karpathy!
http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

- **Traditional ML/Stats approach**: assume x_i are fixed, train (i.e., set) softmax/logistic regression weights $W \in \mathbb{R}^{C \times d}$ to determine a decision boundary (hyperplane) as in the picture
- **Method:** For each x, predict:

$$p(y|x) = \frac{\exp(W_{y}.x)}{\sum_{c=1}^{C} \exp(W_{c}.x)}$$

Details of the softmax classifier

$$p(y|x) = \frac{\exp(W_{y\cdot}x)}{\sum_{c=1}^{C} \exp(W_{c\cdot}x)}$$

We can tease apart the prediction function into two steps:

1. Take the y^{th} row of W and multiply that row with x:

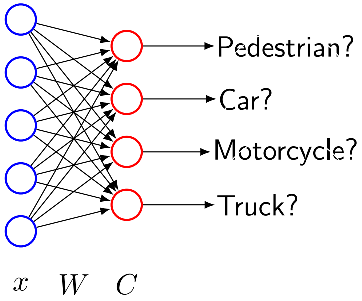
$$W_{y\cdot}x = \sum_{j=1}^{d} W_{yj}x_j = f_y$$

   Compute all f_c for $c = 1, \cdots, C$

2. Apply softmax function to get normalized probability:

$$p(y|x) = \frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)} = softmax(f_y)$$

Multiple Classes



$y \in R^4$

Want $h_W(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ when pedestrian, $h_W(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, when car, etc.

Training with softmax and cross-entropy loss

- For each training example (x, y), our objective is to maximize the probability of the correct class y

- Or we can minimize the negative log probability of that class:

$$-\log p(y|x) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C}\exp(f_c)}\right)$$

Background: What is "cross entropy" loss/error?

- Concept of "cross entropy" is from information theory
- Let the true probability distribution be p
- Let our computed model probability be q
- The cross entropy is:

$$H(p,q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: $p = [0, ..., 0, 1, 0, ...0]$ then:
- **Because of one-hot p, the only term left is the negative log probability of the true class**

Classification over a full dataset

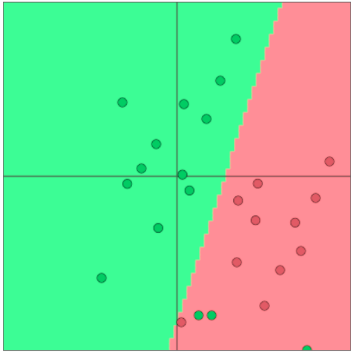
- Cross entropy loss function over full dataset $\{x_i, y_i\}_{i=1}^{N}$

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} - \log \left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}} \right)$$

- Instead of

$$f_y = f_y(x) = W_{y\cdot}x = \sum_{j=1}^{d} W_{yj} x_j$$

We will write f in matrix notation: $f = Wx$



Visualizations with ConvNetJS by Karpathy! http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

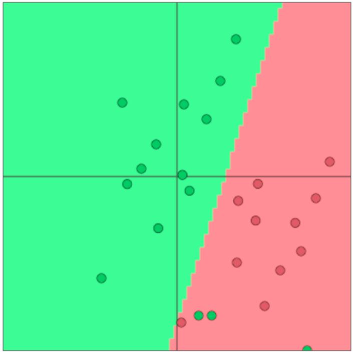
Traditional ML optimization

- For general machine learning θ usually only consists of columns of W:

$$\theta = \begin{bmatrix} W_{\cdot 1} \\ \vdots \\ W_{\cdot d} \end{bmatrix} = W(:) \in \mathbb{R}^{Cd}$$

- So we only update the decision boundary via

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \end{bmatrix} \in \mathbb{R}^{Cd}$$



Visualizations with ConvNetJS by Karpathy!
http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Classification difference with word vectors

Commonly in NLP deep learning:

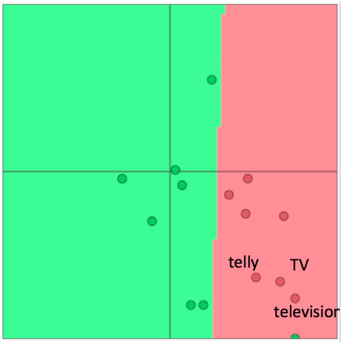
- We learn **both** W and word vectors x
- We learn **both** conventional parameters and representations

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd + \boxed{Vd}}$$

Very large number of parameters!

Losing generalization by re-training word vectors

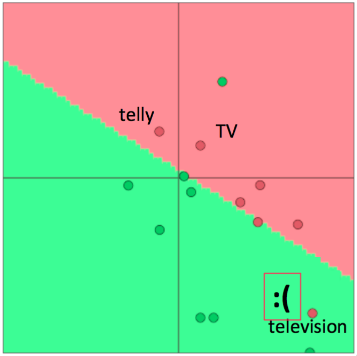
- Setting: Training logistic regression for movie review sentiment and in the training data we have the words
  - "TV" and "telly"
- In the testing data we have
  - "television"
- Originally they were all similar (from pre-training word vectors)
- What happens when we train the word vectors?



Visualizations with ConvNetJS by Karpathy!

Losing generalization by re-training word vectors

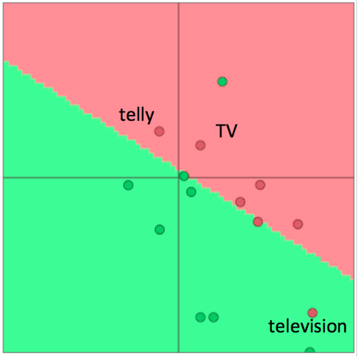
- What happens when we train the word vectors?
  - Those that are in the training data move around
  - Words from pre-training that do NOT appear in training stay
- Example:
  - In training data: “TV” and “telly”
  - In testing data only: “television”



Visualizations with ConvNetJS by Karpathy!

Losing generalization by re-training word vectors

- Take home message:
  - If you only have a small training data set, do not train the word vectors.
  - If you have a very large dataset, it may work better to train word vectors to the task.

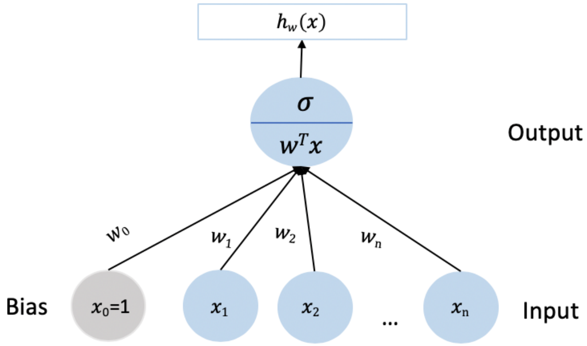


Visualizations with ConvNetJS by Karpathy!

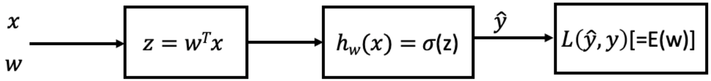
Recall: Logistic Regression for Binary Classification

- Hypothesis Representation for Logistic Regression:
  - $h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$
- Hypothesis Output Interpretation:
  - $h_{\mathbf{w}}(\mathbf{x}) = P(y = 1|\mathbf{x}, \mathbf{w})$ - the confidence in the predicted label
  - $P(y = 0|\mathbf{x}, \mathbf{w}) = 1 - P(y = 1|\mathbf{x}, \mathbf{w})$
- Cost/Loss Function: $E(\mathbf{w}) = L(y, h_{\mathbf{w}}(x)) = -\frac{1}{N}\left[\sum_{i=1}^{N} y^{(i)} \log(h_{\mathbf{w}}(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\mathbf{w}}(\mathbf{x}^{(i)}))\right]$
- To fit parameters $\mathbf{w}$: $\min_{\mathbf{w}} E(\mathbf{w})$
- To make a prediction given a new $\mathbf{x}$: Calculate $h_{\mathbf{w}}(\mathbf{x})$ and compare the value obtained with 0.5.

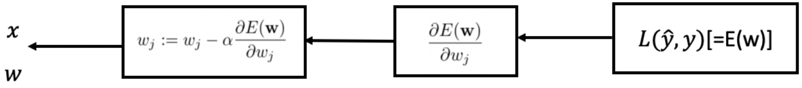
Logistic Regression as a Neural Network



Forward Propagation

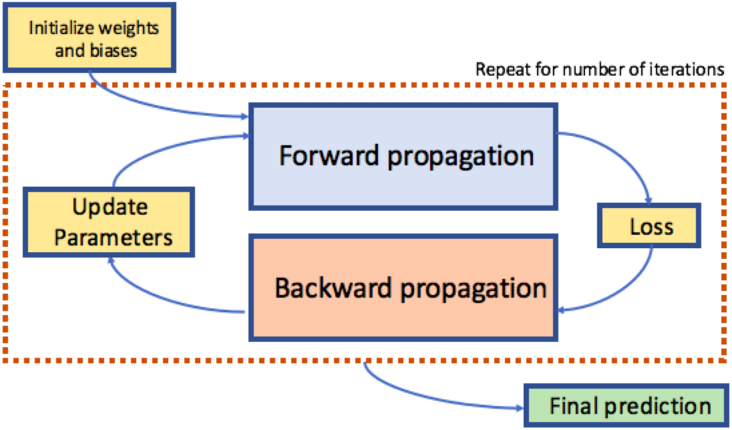


Backward Propagation



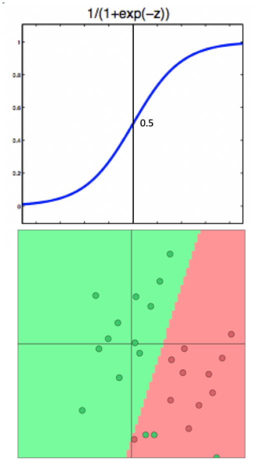
x
w
$w_j := w_j - \alpha \frac{\partial E(\mathbf{w})}{\partial w_j}$
$\frac{\partial E(\mathbf{w})}{\partial w_j}$
$L(\hat{y}, y)$[=E(w)]

Gradient Descent in Neural Network Terminology



Logistic Regression: Decision Boundary

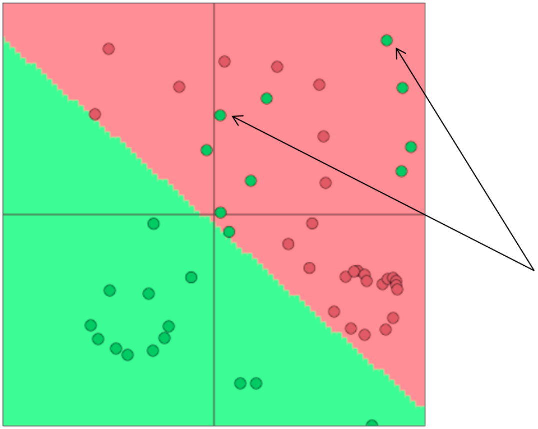
- How does the decision boundary look like for Logistic Regression?
    - Suppose predict $y = 1$ if $h_{\mathbf{w}}(\mathbf{x}) \geq 0.5 \Leftrightarrow \mathbf{w}^T\mathbf{x} \geq 0$
    - Predict $y = 0$ if $h_{\mathbf{w}}(\mathbf{x}) < 0.5 \Leftrightarrow \mathbf{w}^T\mathbf{x} < 0$
- Decision boundary: $\mathbf{w}^T\mathbf{x} = 0$.
    - Hence, the decision boundary is linear, i.e. Logistic Regression is a linear classifier (note: it is possible to kernelize and make it nonlinear)



Visualization with ConvNetJS by Karpathy

Neural Network Classifiers

- Softmax ($\approx$ logistic regression) alone not very powerful
- Softmax gives only linear decision boundaries



This can be quite limiting

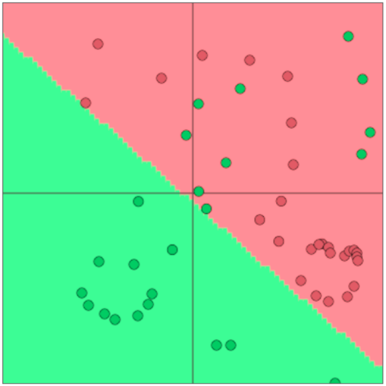
$\rightarrow$ Unhelpful when a problem is complex

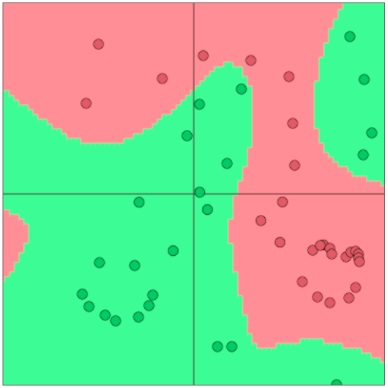
Wouldn't it be cool to get these correct?

Visualizations with ConvNetJS by Karpathy! http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Neural Nets for the Win!

- Neural networks can learn much more complex functions and nonlinear decision boundaries!
    - In original space

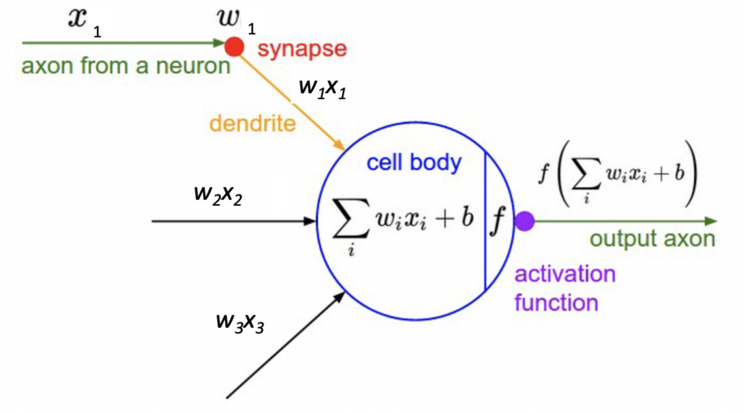




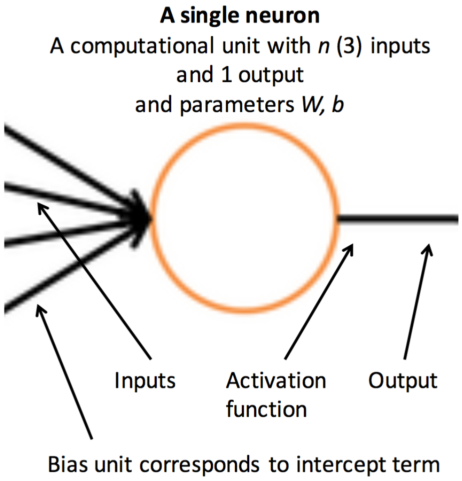
From logistic regression to neural nets

An artificial neuron

- Neural networks come with their own terminology
- But if you understand how softmax models work, then you can easily understand the operation of a neuron!



An artificial neuron - in a nutshell



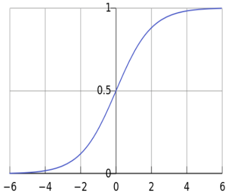
A neuron can be a binary logistic regression unit

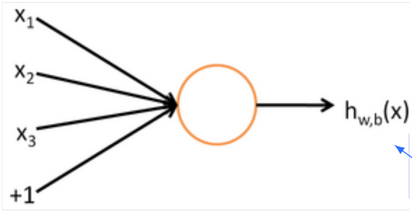
f = nonlinear activation fct. (e.g. sigmoid), w = weights, b = bias, h = hidden, x = inputs

$$h_{w,b}(x) = f(w^T x + b)$$

b: We can have an "always on" feature, which gives a class prior, or separate it out, as a bias term

$$f(z) = \frac{1}{1+e-z}$$

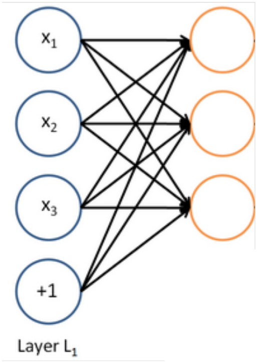




w, b are the parameters of this neuron, i.e., this logistic regression model

A neural network = running several logistic regressions at the same time

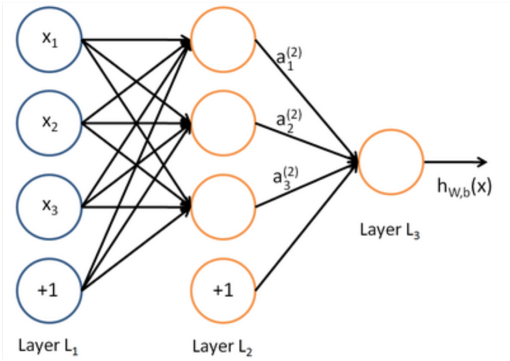
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...



But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network = running several logistic regressions at the same time

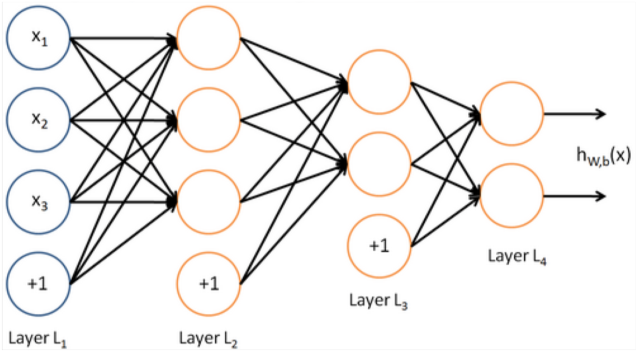
... which we can feed into another logistic regression function



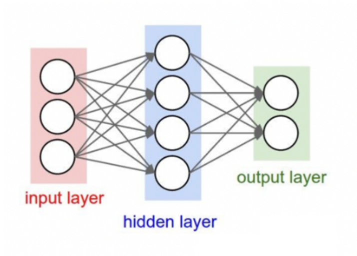
It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

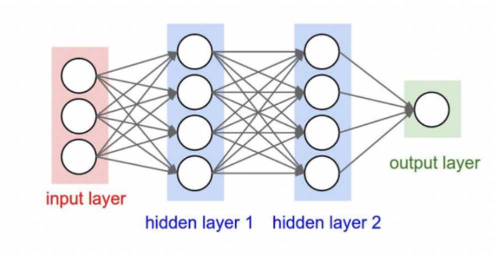
A neural network = running several logistic regressions at the same time

Before we know it, we have a multilayer neural network....



Neural Network Architectures





"2-layer Neural Net" or
"1-hidden-layer Neural Net"

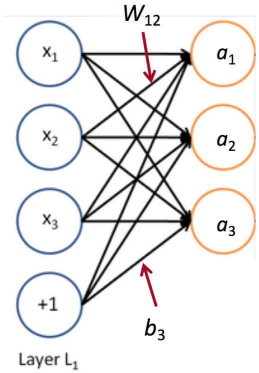
"3-layer Neural Net", or
"2-hidden-layer Neural Net"

Matrix notation for a layer

We have
$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$
$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_1)$
etc.

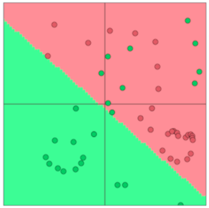
In matrix notation
$z = Wx + b$
$a = f(z)$

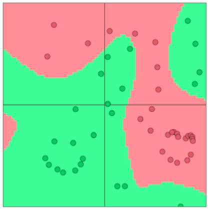
Activation f is applied element-wise:
$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$



Non-linearities ("f"): Why they're needed

- Example: function approximation, e.g., classification
  - Without non-linearities, deep neural networks cannot do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = W x$
  - With more layers, they can approximate more complex functions!

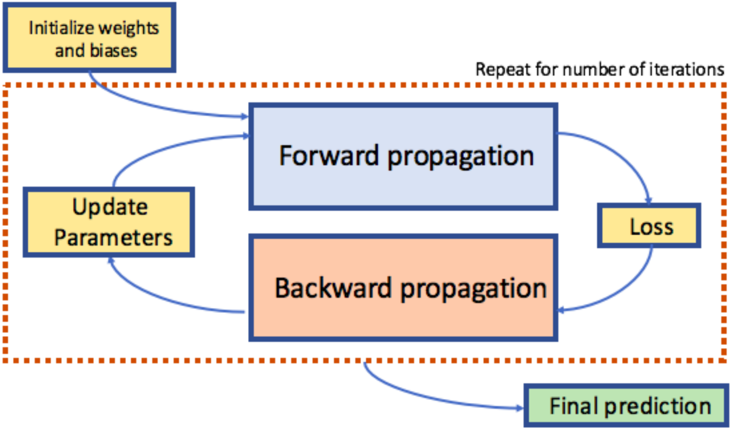




Training a Neural Network

- Given training data $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), \cdots, (\mathbf{x}^{(N)}, y^{(N)})\}$
- Inputs $\mathbf{x}^{(i)}$: n-dimensional vectors (R^n), targets $y^{(i)}$: scalars (R)
- The hypothesis representation $h_{\mathbf{W},\mathbf{b}}$ corresponds to the neural network to be trained
- $\mathbf{W}$ and $\mathbf{b}$ are network parameters, specifically weights and biases.
- As for linear regression and logistic regression, we define a cost/loss function $L(h_{\mathbf{W},\mathbf{b}}(x), y) = L(\hat{y}, y)$
- How to choose network parameters **W** and **b**?
- We minimize the loss $L(\hat{y}, y)$ with respect to **W** and **b**.
- How to minimize the loss?
- Gradient descent!

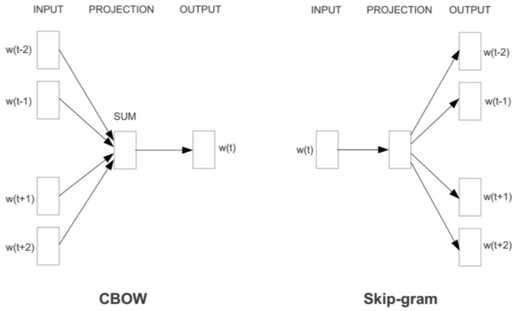
Gradient Descent in Neural Network Terminology

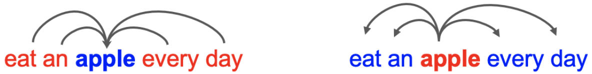


Word2Vec as Neural Nets

Word2vec: neural network models

- Continuous Bag of Word (CBOW): use a window of words (context/outside) to predict the middle word (center)
- Skip-gram (SG): use a word (center) to predict the surrounding ones (context/outside) in the window

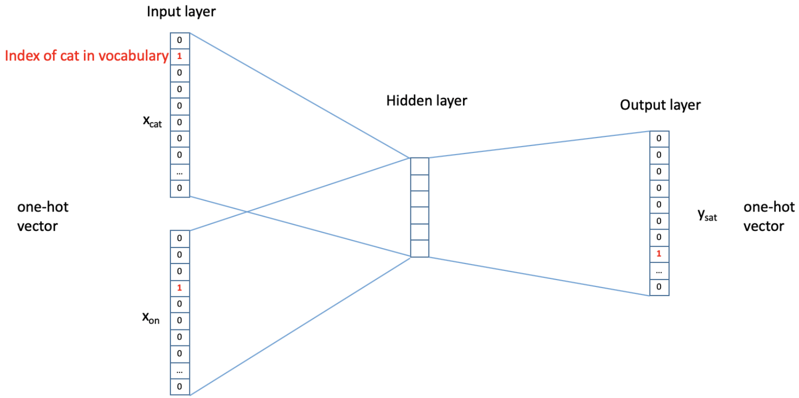




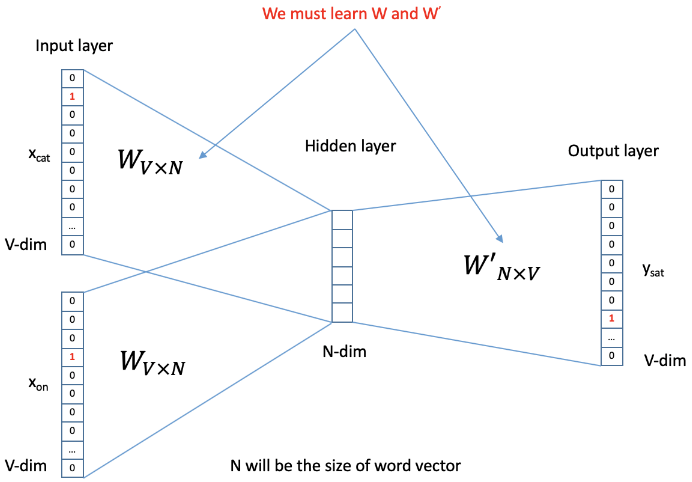
Word2vec: CBOW

Example:

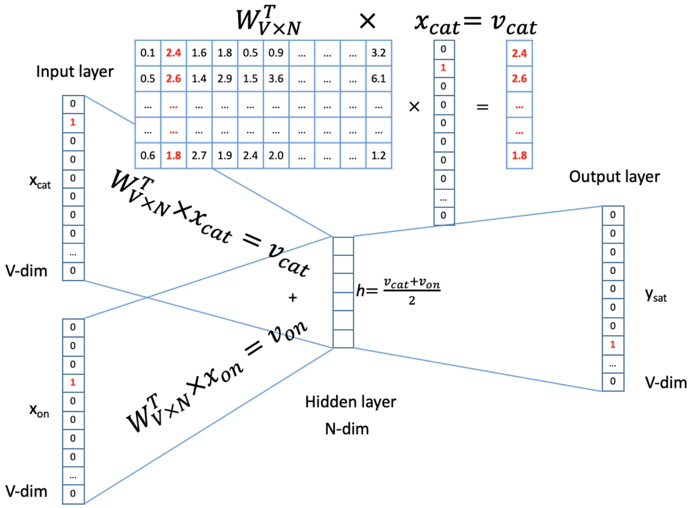
- "The cat sat on the floor"
- Window size is 1 (i.e., 1 word on each side)



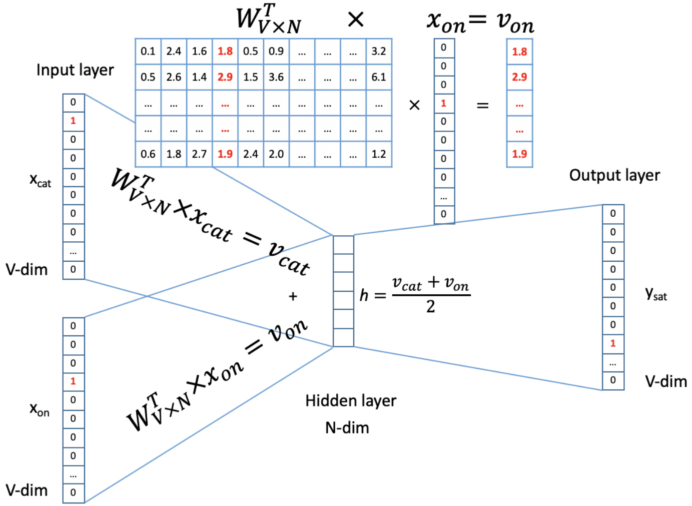
Word2vec: CBOW



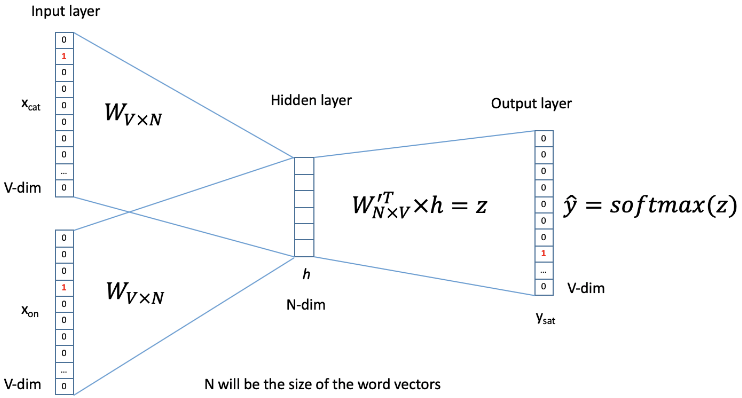
Word2vec: CBOW



Word2vec: CBOW

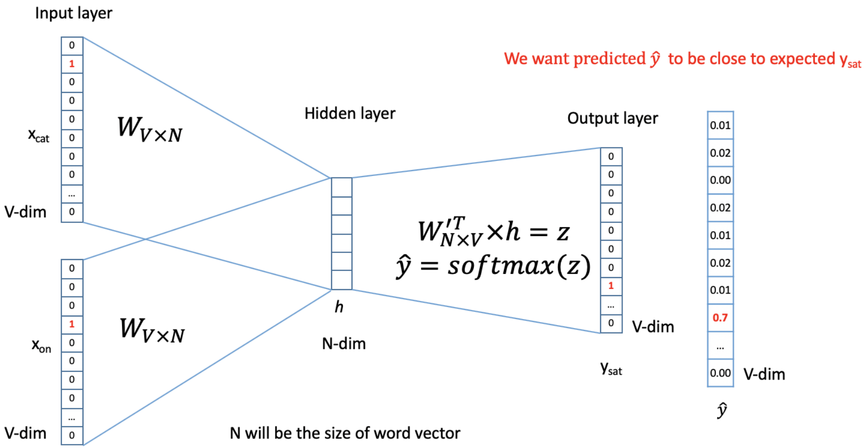


Word2vec: CBOW

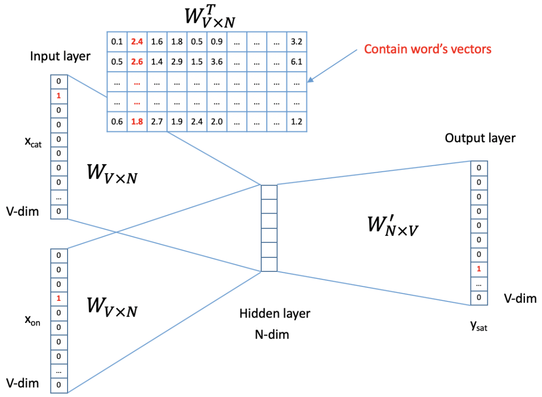


N will be the size of the word vectors

Word2vec: CBOW



Word2vec: CBOW



0.1	2.4	1.6	1.8	0.5	0.9	...	...	...	3.2
0.5	2.6	1.4	2.9	1.5	3.6	...	...	...	6.1
...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
0.6	1.8	2.7	1.9	2.4	2.0	...	...	...	1.2

- Matrix W corresponds to input word vectors, while matrix W' corresponds to output word vectors.
- We can use either W or W' for word embeddings, or even the element-wise average of the two matrices.

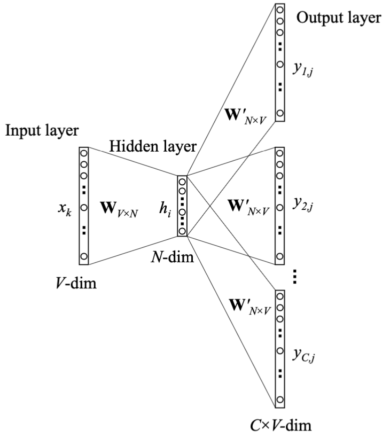
Softmax function

- Softmax function $\mathbb{R}^V \rightarrow \mathbb{R}^V$

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{n} exp(z_j)} = \hat{y}_i$$

- The softmax function maps values z to a probability distribution $\hat{y}$ (where each z_i is mapped to a probability $\hat{y}_i$)
  - "max" because amplifies probability of largest z_i
  - "soft" because still assigns some probability to smaller z_i
  - Frequently used in Deep Learning

Skip-gram model (nearby 1 word)



- Objective: Predict context words $w_{o_1}, \cdots, w_{o_C}$ (one at a time) given a center word w_c. We denote by $y_1, \cdots, y_C$ the one-hot representations of target context words $w_{o_1}, \cdots, w_{o_C}$.
- For each target context word w_{o_j}, we compute a softmax distribution over the vocabulary V, using the same W' matrix.

Skip-gram: additional efficiency

- **Skip-grams (SG)**
  Predict context ("outside") words (position independent) given center word
- Uses **softmax** (simpler, but expensive training method)
- The normalization factor is too computationally expensive.

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Additional efficiency in training:

- Negative sampling

Skip-gram model with negative sampling

- In standard word2vec implementation, the skip-gram model uses negative sampling
- Main idea: train binary logistic regressions for a true pair (center word and word in its context window) versus several noise pairs (the center word paired with a random word)
- **Example**: `I want a glass of orange juice to go along with my cereal.`

center	**outside**	**label**
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

Skip-gram model with negative sampling

- Softmax:

$$P(w_o|w_c) = \frac{\exp(\mathbf{u}_{w_o}^T \mathbf{v}_{w_c})}{\sum_{j=1}^{V} \exp(\mathbf{u}_{w_j}^T \cdot \mathbf{v}_{w_c})}$$

- Logistic regression:

$$P(1|w_o, w_c) = \sigma(\mathbf{u}_{w_o}^T \mathbf{v}_{w_c})$$

- Instead of training softmax with V classes, we train V logistic regression classifiers.
- For each 1 positive and k negative samples (sampled using word probabilities), we train $k+1$ binary logistic regression classifiers on each iterations
- Maximize probability that real outside word appears,
  minimize probability that random words appear around center word

Skip-gram model with negative sampling

- Minimize the cross-entropy loss:

$$J_{neg\_sample}(u_o, v_c, U) = -\log(\sigma(u_o^T v_c)) - \sum_{k=1}^{K} \log(\sigma(-u_k^T v_c))$$

- Maximize probability that real outside word appears,
  minimize probability that random words appear around center word

- We take k negative samples (using word probabilities)

- $P(w) = U(w)^{3/4}/Z$,
  the unigram distribution $U(w)$ raised to the $3/4$ power, Z is the normalizing factor.
- The power makes less frequent words be sampled more often (e.g., $0.9^{3/4} = 0.92$, $0.01^{3/4} = 0.032$).