Natural Language Processing

CS 3216/UG, AI 5203/PG

Week-6

Neural Networks & Neural Language Models

Mahindra™
University
Global Thinkers. Engaged Leaders.

Recap

- NLP
- Applications
- Regular expressions
- Tokenization
- Stemming
    - Porter Stemmer
- Lemmatization
- Normalization
- Stopwords
- Bag-of-Words
- TF-IDF
- NER
- POS tagging
- Semantics, Distributional semantics, Word2vec
- Language models

Last Lecture

- Language models
- N-grams
- Evaluation

Smoothing in N-grams

- Like many statistical models, the N-gram probabilistic language model is dependent on the training corpus.
- One practical issue with this is that some word sequences and phrases appear in practice (or in the test set), may not also occur in the training set.
- SPARSITY AND STORAGE PROBLEMS
- Important to train robust models that generalize well to handle the unseen words and zero probabilities

Add one Smoothing

- Also called Laplace smoothing
- Pretend we saw each word one more time than we did
- Just add one to all the counts!

MLE estimate:

$$P_{MLE}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

Add-1 estimate:

$$P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$

Credits: Slide adapted from [2]

Recall the Language modeling task

Input: sequence of words, $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, \ldots\ldots\ldots\ldots x^{(t)}$

Output: probability distribution of the next word: $P(x^{(t+1)} | x^{(t)}, \ldots\ldots\ldots x^{(1)})$

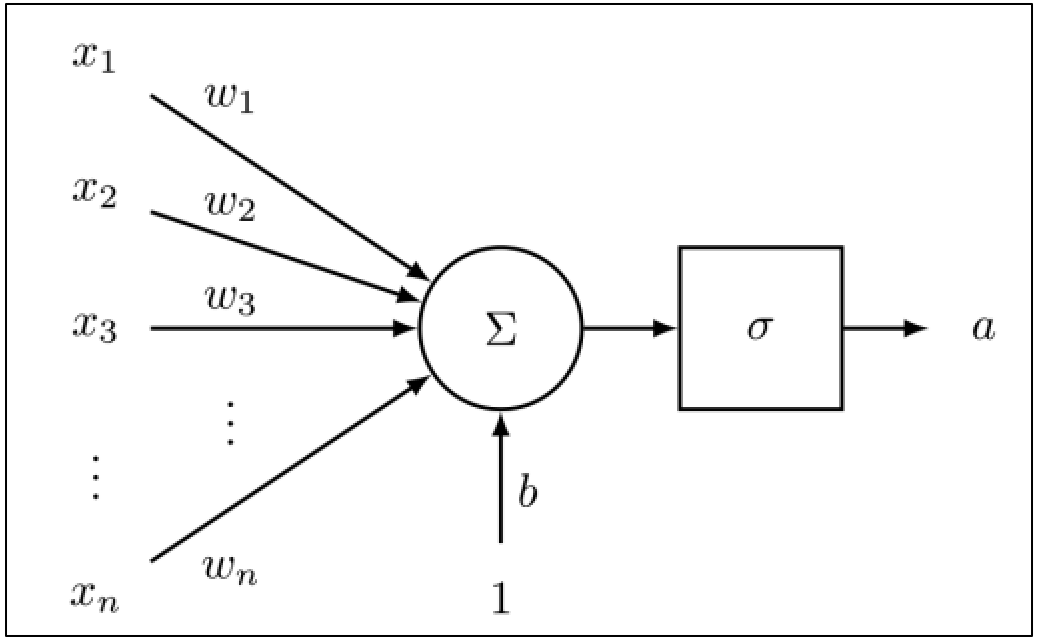
How about a window-based neural model?

Before building Neural language models

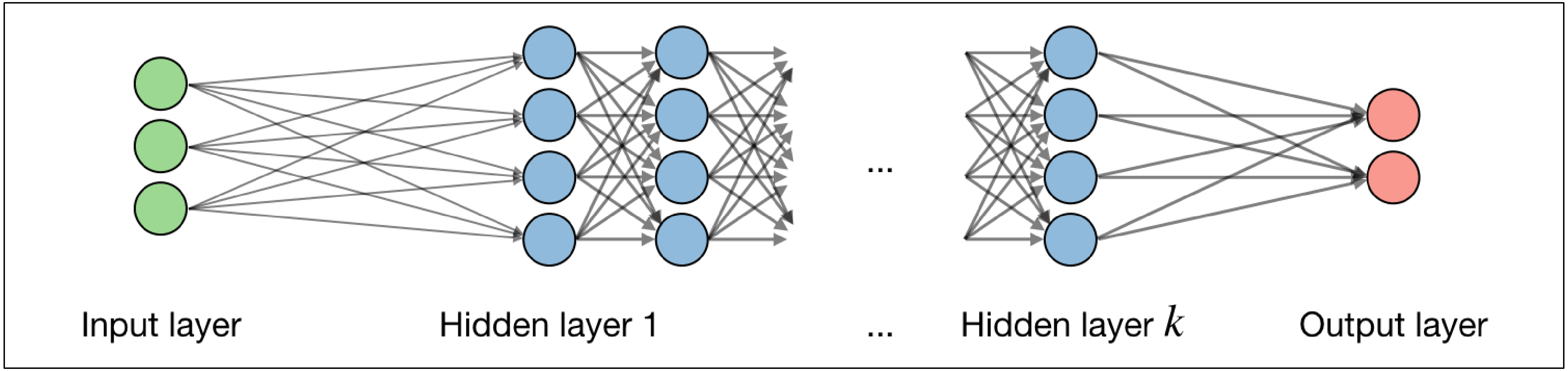
Let's first look into basics of Neural Network.........

Neurons

A neuron is the fundamental building block of neural networks



Neural Networks/Feed- Forward Neural Network



$$z_j^{[i]} = {w_j^{[i]}}^T x + b_j^{[i]}$$

where we denote, *w, b, z* as the weight, bias and output respectively.

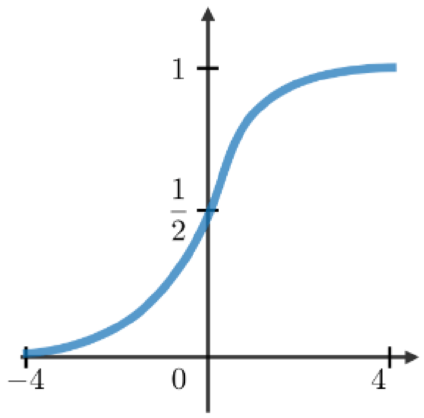
https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning#Neural%20Networks

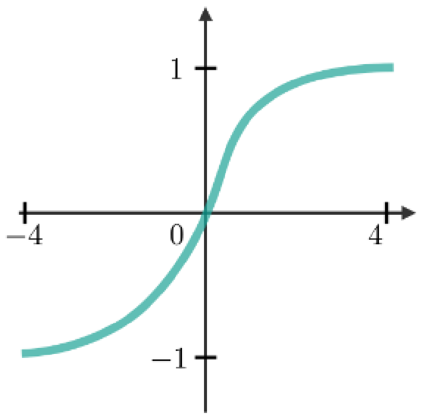
Activation functions

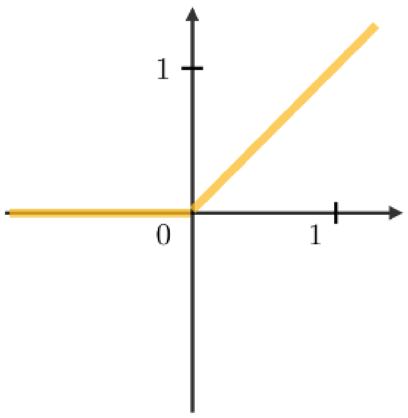
Activation functions are used at the end of a hidden unit to introduce non-linear complexities to the model.

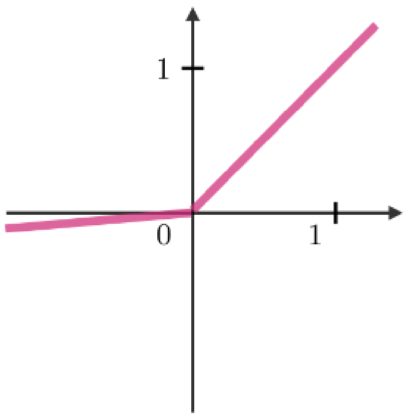
Here are the most common ones:

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$









Backpropagation

- Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output.
- The derivative with respect to weight w is computed using **chain rule** and is of the following form:

$$\boxed{\frac{\partial L(z,y)}{\partial w} = \frac{\partial L(z,y)}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w}}$$

- Weight is updated as follows:

$$\boxed{w \longleftarrow w - \alpha \frac{\partial L(z,y)}{\partial w}}$$

Backpropagation Visualization

Chain Rule!!!!

https://docs.google.com/presentation/d/1pmstGvQColwIDP9fJkVLDlG4BNRIfJu8cJTDcokUjrM/edit#slide=id.g27be483e10_0_0

Learning rate

- **Learning rate:** α or sometimes η, indicates at which pace the weights get updated.

- This can be fixed or adaptively changed.

- The current most popular method is called Adam, which is a method that adapts the learning rate.

https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning#Neural%20Networks

Training Neural Networks

In a neural network, weights are updated as follows:

Step 1: Take a batch of training data.

Step 2: Perform forward propagation to obtain the corresponding loss.

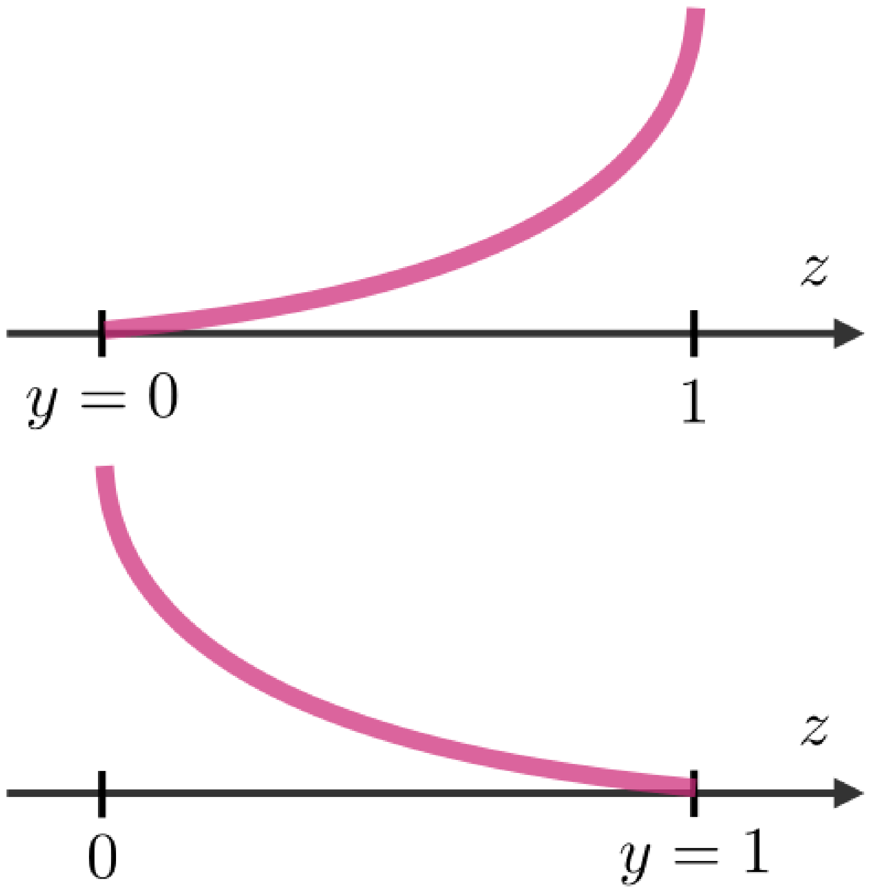
Step 3: Backpropagate the loss to get the gradients.

Step 4: Use the gradients to update the weights of the network.

https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning#Neural%20Networks

Neural Network Loss function (Cross-entropy loss)

$$-\left[y \log(z) + (1-y)\log(1-z)\right]$$



Interpreting logits: Sigmoid

- *Neural networks* are capable of producing raw output scores for each of the classes.
- How do we convert output scores into probabilities?
- Sigmoid function: $\sigma : R \rightarrow [0,1]$

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

  - **Class A** (also called the positive class)
  - **Not Class A** (complement of Class A or also called the negative class)

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

Interpreting logits: Softmax

Presenting the **softmax** function $S : \mathbf{R}^C \rightarrow [0,1]^C$

$$S(\mathbf{z})_i = \frac{e^{\mathbf{z}_i}}{\sum_{j=1}^{C} e^{\mathbf{z}_j}} = \frac{e^{\mathbf{z}_i}}{e^{\mathbf{z}_1} + \ldots + e^{\mathbf{z}_j} + \ldots + e^{\mathbf{z}_C}}$$

This function takes a vector of real-values and converts each of them into corresponding probabilities. In a C-class classification where $k \in \{1, 2, \ldots, C\}$, it naturally lends the interpretation

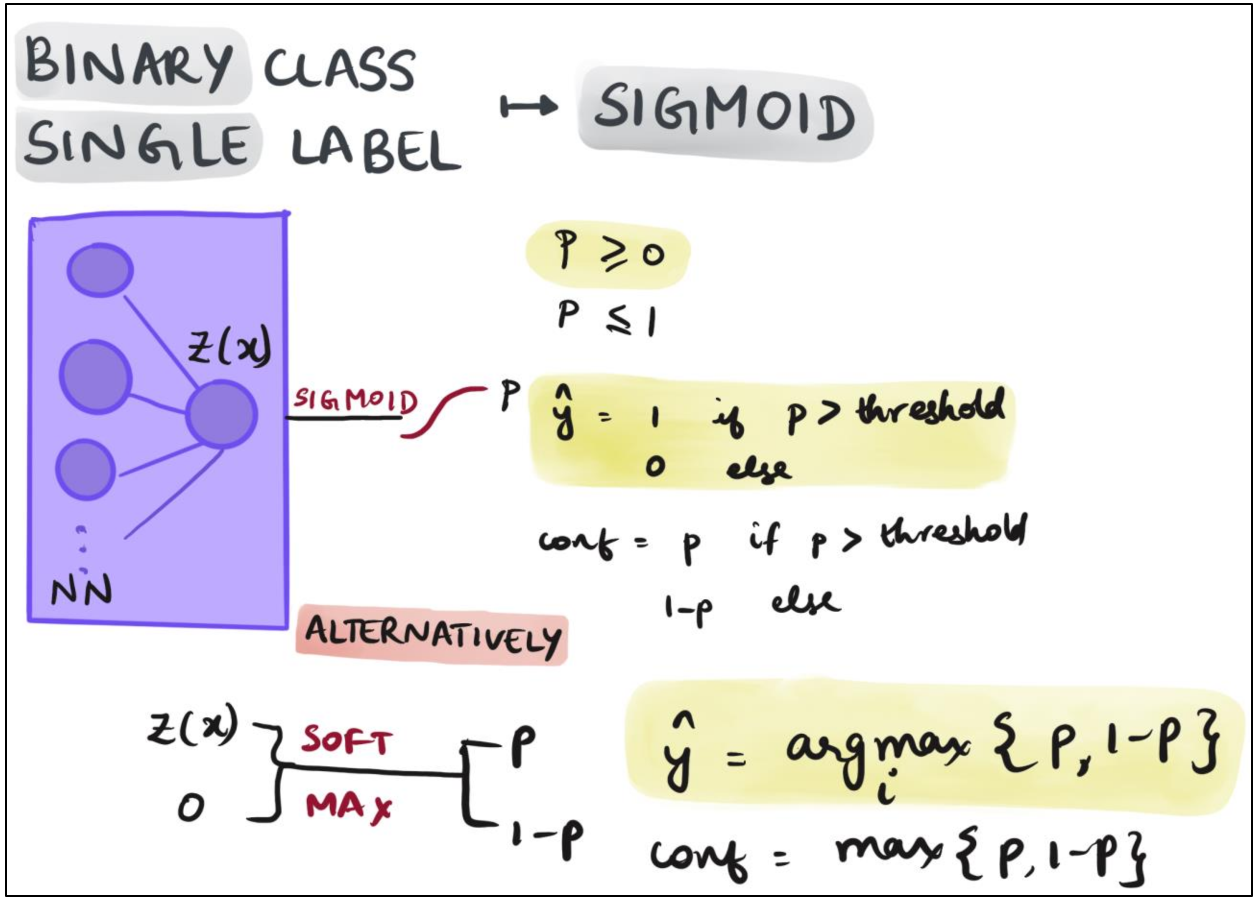
$$\mathbf{prob}(y = k|\mathbf{x}) = \frac{e^{\mathbf{z}(\mathbf{x})_k}}{\sum_{j=1}^{C} e^{\mathbf{z}(\mathbf{x})_j}}$$

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

SoftMax Classifier Implementation

https://docs.google.com/presentation/d/1dFbY-Buku18xhHHbFVNPNuc6vsPeR-LbE8KQXzvuzZg/edit#slide=id.g27be483e10_0_0

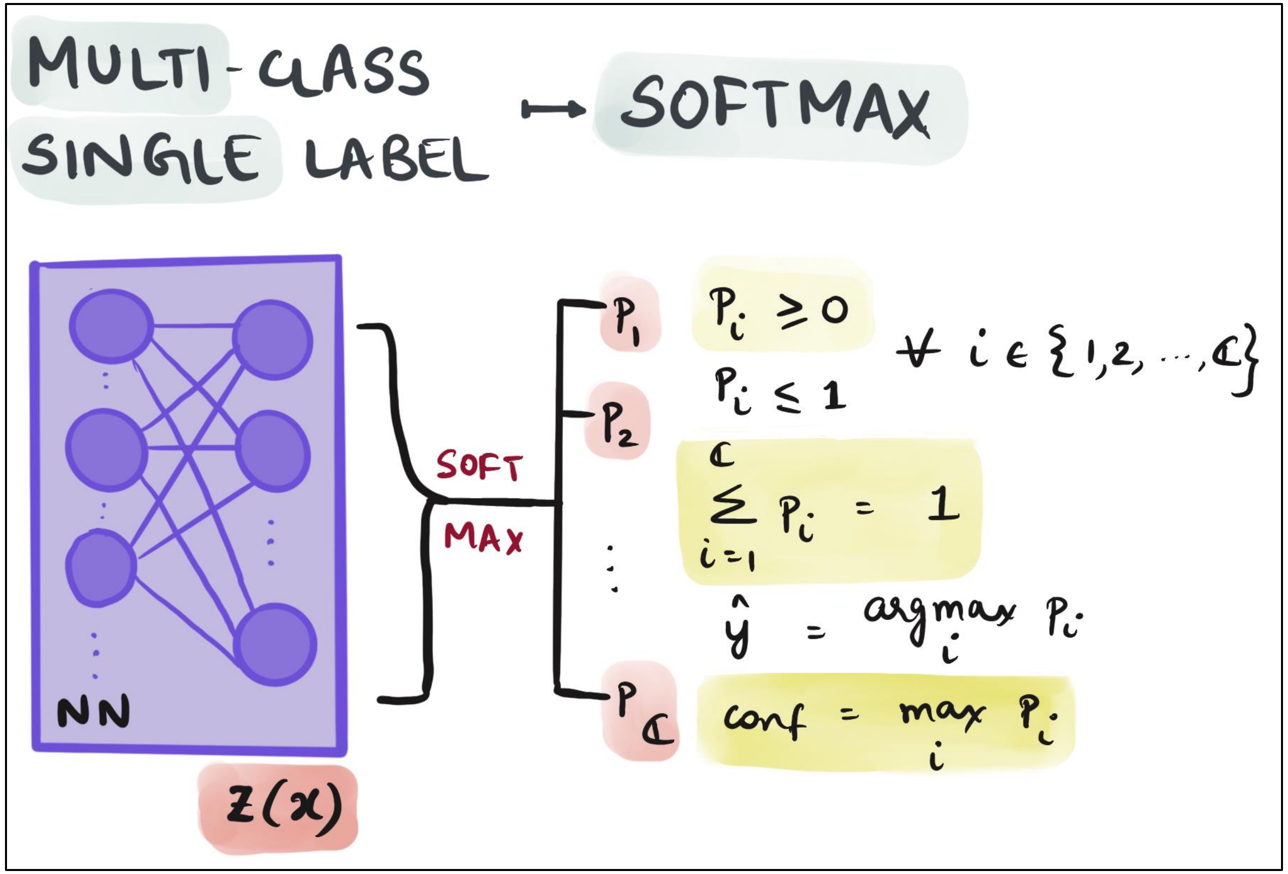
Binary Classification/Single Label



Mutually exclusive and exhaustive, i.e., an input instance can belong to either class, but not both and their **probabilities sum to 1.**

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

Multi-class Classification/Single Label

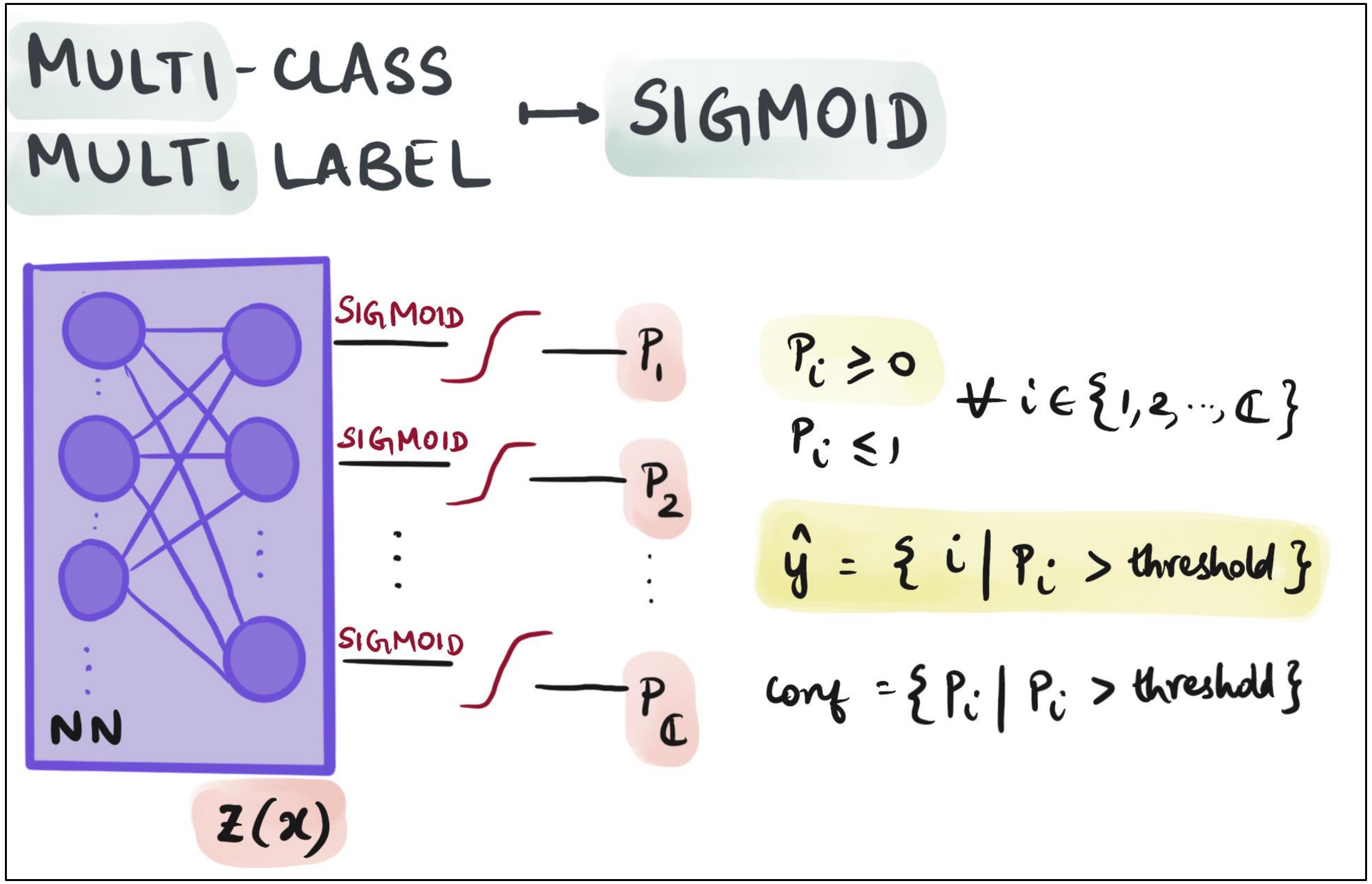


Mutually exclusive and exhaustive, i.e., an input instance can belong to either class, but not both and **their probabilities sum to 1.**

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

Multi-class Classification/Multi Label

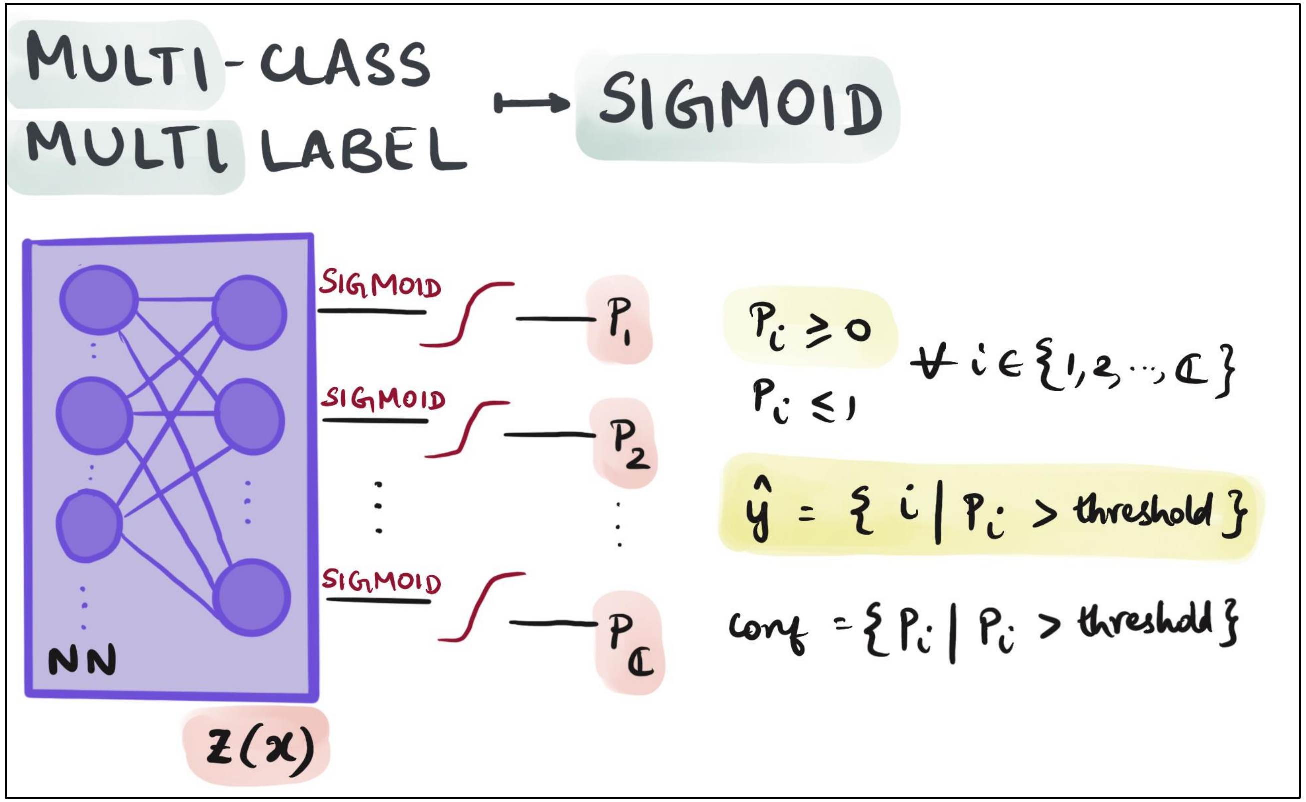
If input data can belong to more than one class in a multi-class classification problem?



For instance, Genre **classification of movies (a movie can fall into multiple genres) or classification of chest x-rays (a given chest x-ray can have more than one disease).**

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

Multi-class Classification/Multi Label



Here Classes are NOT mutually exclusive. **i.e., train a binary classifier independently for each class.** This can be done easily by just applying **sigmoid function** to each of raw scores.

Note that the output probabilities will NOT sum to 1.

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

Implementation of SoftMax/Sigmoid

```
import torch

def getSoftmaxScores(inputs, dimen):
        ''' Get the softmax scores '''
        print('---Softmax---')
        print('---Dim = ' + str(dimen) + '---')
        softmaxFunc = torch.nn.Softmax(dim = dimen)
        softmaxScores = softmaxFunc(inputs)
        print('Softmax Scores: \n', softmaxScores)
        sums_0 = torch.sum(softmaxScores, dim=0)
        sums_1 = torch.sum(softmaxScores, dim=1)
        print('Sum over dimension 0: \n', sums_0)
        print('Sum over dimension 1: \n', sums_1)

def getSigmoidScores(inputs):
        ''' Get the sigmoid scores: they are element-wise '''
        print('---Sigmoid---')
        sigmoidScores = torch.sigmoid(inputs)
        print('Sigmoid Scores: \n', sigmoidScores)

logits = torch.randn(2, 3)*10 - 5
print('Logits: ', logits)
```

https://web.stanford.edu/~nanbhas/blog/sigmoid-softmax/#binary-classification

A fixed-window Neural Language model

output distribution

$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$

hidden layer

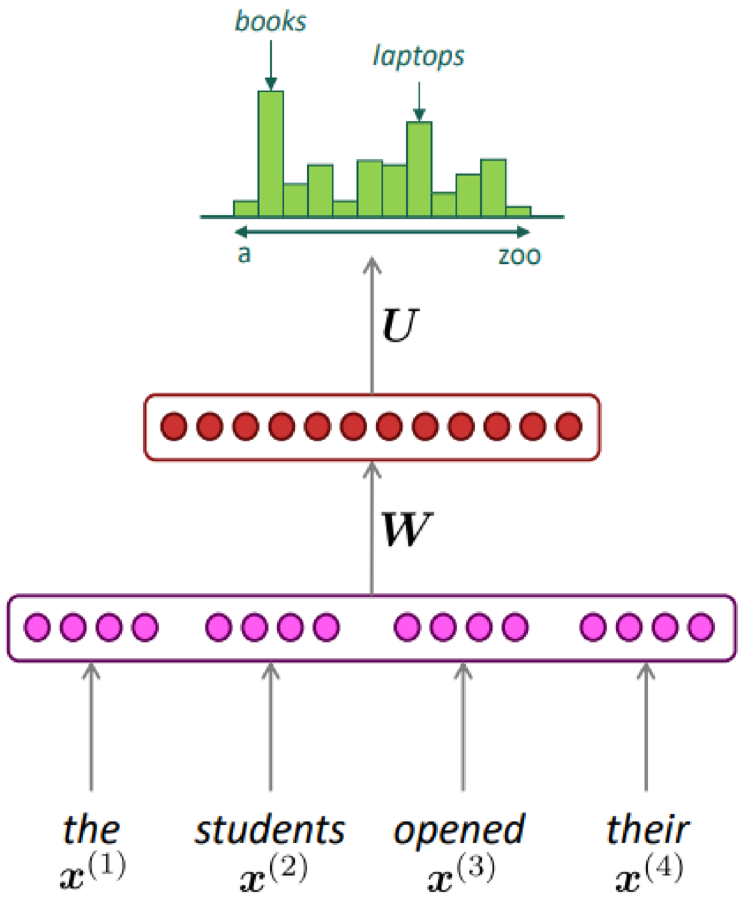
$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$

concatenated word embeddings

$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$

words / one-hot vectors

$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$



Credits: Slide adapted from [1]

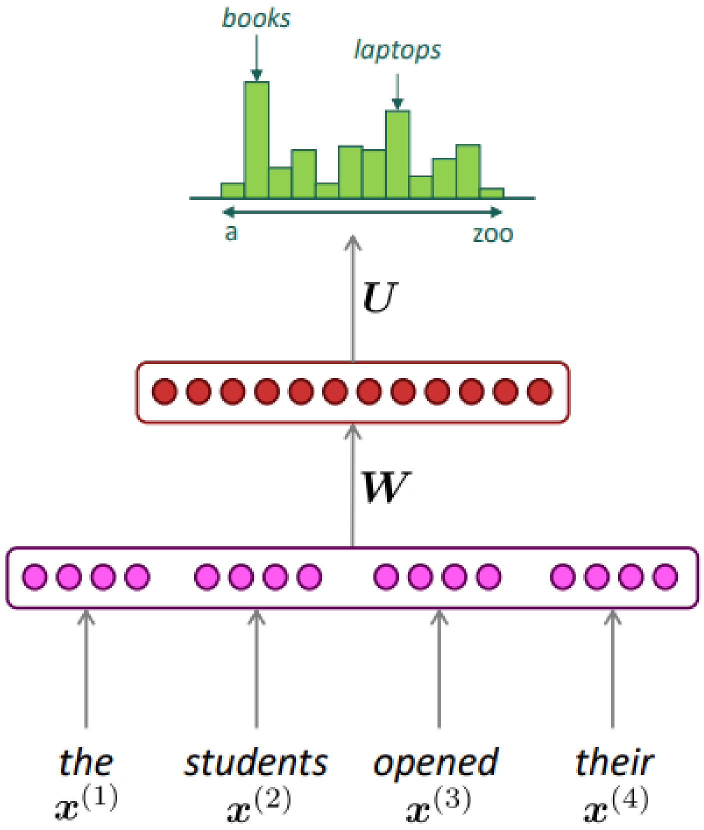
A fixed-window Neural Language Model

Approximately: Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Improvements over *n*-gram LM:
- No sparsity problem
- Don't need to store all observed *n*-grams

Remaining **problems**:
- Fixed window is too small
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W. No symmetry in how the inputs are processed.

We need a neural architecture that can process *any length input*



Credits: Slide adapted from [1]

References

[1] https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1234/slides/cs224n-2023-lecture05-rnnlm.pdf

[2] https://web.stanford.edu/~jurafsky/slp3/slides/LM_4.pdf

Reference materials

- https://vlanc-lab.github.io/mu-nlp-course/
- Lecture notes
- (A) Speech and Language Processing by Daniel Jurafsky and James H. Martin
- (B) Natural Language Processing with Python. (updated edition based on Python 3 and NLTK 3) Steven Bird et al. O'Reilly Media