Lecture 4: Nonlinear Classifiers

Instructor: Jackie CK Cheung & David Adelani

COMP-550

Readings: Eisenstein Ch. 3

J&M Ch 7 – 7.3 (3rd ed)

Announcements

Assignment 1 to be released this week

- Theme: **Text classification**

TA Office hours start next week

- We'll post schedule on Ed

Tutorials coming in the next few weeks

- Using NLP Python packages
- Probability basics and review

Classification

Map input x to output y:

$$y = f(x)$$

Classification: y is a discrete outcome

- Genre of the document (news text, novel, …?)
- Overall topic of the document
- Spam vs. non-spam
- Identity, gender, native language, etc. of author
- Positive vs. negative movie review
- Other examples?

Steps

1. Define problem and collect data set
2. Extract features from documents
3. **Train a classifier on a training set [today, again]**
4. **Apply classifier on test data [more on this too]**

Logistic Regression

Linear regression:

$$y = a_1x_1 + a_2x_2 + \dots + a_nx_n + b$$

Intuition: Linear regression gives as continuous values in [-∞, ∞] —let's squish the values to be in [0, 1]!

Function that does this: logit function

$$P(y|\vec{x}) = \frac{1}{Z}e^{a_1x_1 + a_2x_2 + \dots + a_nx_n + b}$$

This Z is a normalizing constant to ensure this is a probability distribution.

(a.k.a., maximum entropy or MaxEnt classifier)

N.B.: Don't be confused by name—this method is most often used to solve classification problems.

Linear Model

Logistic regression, support vector machines, etc. are examples of **linear models**.

$$P(y|\vec{x}) = \frac{1}{Z} e^{\underbrace{a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + b}_{\text{Linear combination of feature weights and values}}}$$

Cannot learn complex, non-linear functions from input features to output labels (without adding features)

e.g., Starts with a capital AND not at beginning of sentence -> proper noun

Linear Model

Another way to express logistic regression is to pass all argument to a **sigmoid function**.

Suppose $\mathrm{u} = a_1 x_1 \ + \ a_2 x_2 \ + \ \ldots + \ a_n x_n \ + b$

$$P(y|\vec{x}) = \frac{1}{1+e^{-u}} = \frac{e^u}{e^u+1}$$

$$P(y|\vec{x}) \ = \frac{1}{Z} e^u$$

$$P(y|\vec{x}) = \frac{1}{Z} e^{a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + b}$$

This Z is a normalizing constant to ensure this is a probability distribution.

(Artificial) Neural Networks

A kind of learning model which automatically learns non-linear functions from input to output

Biologically inspired metaphor:

- Network of computational units called neurons
- Each neuron takes scalar inputs, and produces a scalar output, very much like a logistic regression model

$$\text{Neuron}(\vec{x}) = g(a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + b)$$

As a whole, the network can theoretically compute any computable function, given enough neurons. (These notions can be formalized.)

Responsible For:

Atari game-playing bot (Google) (2015)

Generative AI (e.g., ChatGPT, Midjourney)

State of the art in:

- Language modeling
- Speech recognition
- Machine translation
- Object detection
- Most NLP tasks

Feedforward Neural Networks

All connections flow forward (no loops); each layer of hidden units is fully connected to the next.

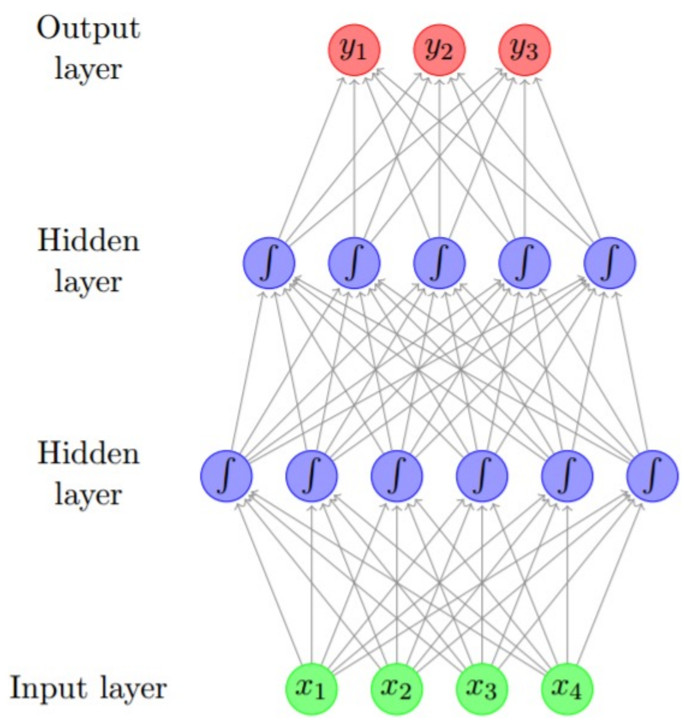

Figure 2: Feed-forward neural network with two hidden layers.

Figure from Goldberg (2015)

Inference in a FF Neural Network

Perform computations forwads through the graph:

$$\mathbf{h^1} = g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})$$
$$\mathbf{h^2} = g^2(\mathbf{h^1}\mathbf{W^2} + \mathbf{b^2})$$
$$\mathbf{y} = \mathbf{h^2}\mathbf{W^3}$$

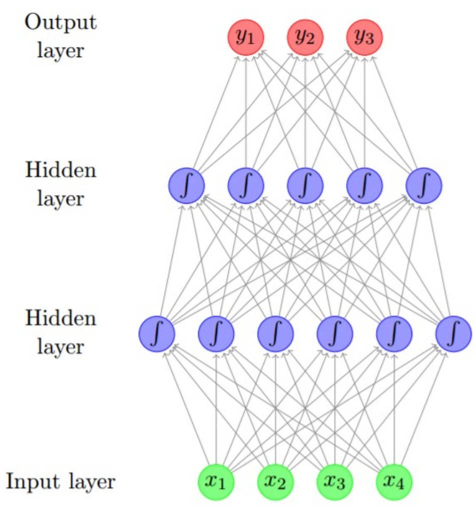

Figure 2: Feed-forward neural network with two hidden layers.

Note that we are now representing each layer as a vector; combining all of the weights in a layer across the units into a weight matrix

Activation Function

In one unit:

Linear combination of inputs and weight values → non-linearity

$$\mathbf{h^1} = g^1(\mathbf{xW^1} + \mathbf{b^1})$$

Popular choices:

Sigmoid function (just like logistic regression!)

tanh function

Rectifier/ramp function: $g(x) = \max(0, x)$

Why do we need the non-linearity?

Softmax Layer

In NLP, we often care about discrete outcomes

- e.g., words, POS tags, topic label

Output layer can be constructed such that the output values sum to one:

$$\text{Let } \mathbf{x} = x_1 \ldots x_k$$

$$softmax(x_i) = \frac{\exp(x_i)}{\sum_j^k \exp(x_j)}$$

Interpretation: unit x_i represents probability that outcome is i.

Essentially, the last layer is like a *multi-class logistic regression*

Loss Function

A neural network is optimized with respect to a **loss function**, which measures how much error it is making on predictions:

$\mathbf{y}$: correct, gold-standard distribution over class labels

$\hat{\mathbf{y}}$: system predicted distribution over class labels

$L(\mathbf{y}, \hat{\mathbf{y}})$: loss function between the two

Popular choice for classification (usually with a softmax output layer) – **cross entropy**:

$$L_{ce}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log(\widehat{y_i})$$

Training Neural Networks

Typically done by **gradient descent**

- Find gradient of loss function wrt parameters of the network (i.e., the weights of each layer); “travel along in that direction”.

Network has very many parameters!

Efficient algorithm to compute the gradient with respect to all parameters: **backpropagation** (Rumelhart et al., 1986)

- Boils down to an efficient way to use the chain rule of derivatives to propagate the error signal from the loss function backwards through the network back to the inputs

Gradient Descent Summary

Descent vs ascent

Convention: think about the problem as a minimization problem

Minimize the *loss function*

- $\theta \leftarrow \theta - \gamma(\nabla L(\theta))$

Initialize $\theta = \{\theta_1, \theta_2, \dots, \theta_k\}$ randomly

Do for a while:

Compute $\nabla L(\theta)$, [by doing calculus]
$\theta \leftarrow \theta - \gamma \nabla L(\theta)$

Stochastic Gradient Descent (SGD)

In the standard version of the algorithm, the gradient is computed over the entire training corpus.

- Sum over all samples in training corpus
- Weight update *once* per iteration through training corpus.

Alternative: calculate gradient over a small mini-batch of the training corpus and update weights

SGD is when mini-batch size is one.

- Many weight updates per iteration through training corpus
- Usually results in much faster convergence to final solution, without loss in performance

SGD Overview

Inputs:

- Function computed by neural network, $f(\mathbf{x}; \theta)$
- Training samples $\{\mathbf{x^k}, \mathbf{y^k}\}$
- Loss function L

Repeat for a while:

Sample a training case, $\mathbf{x^k}, \mathbf{y^k}$

Compute loss $L(f(\mathbf{x^k}; \theta), \mathbf{y^k})$

Forward pass

Compute gradient $\nabla L(\mathbf{x^k})$ wrt the parameters θ

Update $\theta \leftarrow \theta - \eta \nabla L(\mathbf{x^k})$

In neural networks, by backpropagation

Return θ

Example: Forward Pass

$\mathbf{h^1} = g^1(\mathbf{xW^1} + \mathbf{b^1})$
$\mathbf{h^2} = g^2(\mathbf{h^1W^2} + \mathbf{b^2})$
$f(\mathbf{x}) = \mathbf{y} = g^3(\mathbf{h^2}) = \mathbf{h^2W^3}$

Loss function: $L(\mathbf{y}, \mathbf{y}^{gold})$

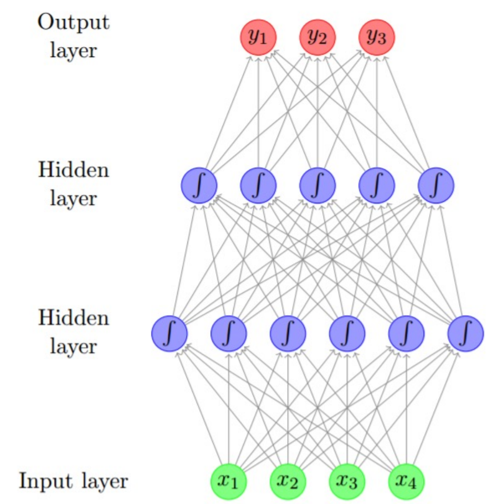

Figure 2: Feed-forward neural network with two hidden layers.

Save the values for $\mathbf{h^1}$, $\mathbf{h^2}$, $\mathbf{y}$ too!

Example Cont'd: Backpropagation

$$f(\mathbf{x}) = g^3(g^2(g^1(\mathbf{x})))$$

Need to compute: $\frac{\partial L}{\partial \mathbf{W}^3}, \frac{\partial L}{\partial \mathbf{W}^2}, \frac{\partial L}{\partial \mathbf{W}^1}$

By calculus and chain rule:

- $\frac{\partial L}{\partial \mathbf{W}^3} = \frac{\partial L}{\partial g^3} \frac{\partial g^3}{\partial \mathbf{W}^3}$
- $\frac{\partial L}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial g^3} \frac{\partial g^3}{\partial g^2} \frac{\partial g^2}{\partial \mathbf{W}^2}$
- $\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial g^3} \frac{\partial g^3}{\partial g^2} \frac{\partial g^2}{\partial g^1} \frac{\partial g^1}{\partial \mathbf{W}^1}$

Notice the overlapping computations? Be sure to do this in a smart order to avoid redundant computations!

Word Representations

We need to represent words in a document as a feature vector. What we've done so far:

w1	w2	w3	w4
[count_1	count_2	count_3	count_4]

More typical choice is to associate each word type with a fixed-dimensional vector e.g. 300d (like Word2Vec), which are model parameters:

w1 [0.3, 0.5, 0.6]

w2 [-0.1, 0.2, 0.7]

Sentence Representations

To represent an input sentence or document, need to combine the input word vector representations:

Simple vector addition

this	is	a	sentence	
v^{this}	v^{is}	v^{a}	$v^{sentence}$	look-up layer

$$s = v^{this} + v^{is} + v^{a} + v^{sentence}$$

Component-wise vector multiplication

this	is	a	sentence	
v^{this}	v^{is}	v^{a}	$v^{sentence}$	look-up layer

$$s = v^{this} \odot v^{is} \odot v^{a} \odot v^{sentence}$$

Much more sophisticated options are possible e.g. concatenation.

Modern Trends in NNs for NLP

Transformer architecture (we'll return to this later)

- Learn how word vectors interact with each other in a passage, rather than applying a simple pooling operator

Multi-stage training on large text corpus

Stage 1 – Pretraining: predict which words are in the context of which other words

this is a [**blank**] bank

Stage 2 – Finetuning: change the output head to one that is geared towards end task (e.g., classification task) on a dataset of interest

Hardware for NNs

Common operations in inference and learning:

- Matrix multiplication
- Component-wise operations (e.g., activation functions)

This operation is **highly parallelizable**!

Graphical processing units (**GPUs**) are specifically designed to perform this type of computation efficiently

Summary: Advantages of NNs

Learn relationships between inputs and outputs:

- Complex features and dependencies between inputs and states over long ranges
- Reduces need for feature engineering
- More efficient use of input data via weight sharing

Highly flexible, generic architecture

- **Multi-task learning**: jointly train model that solves multiple tasks simultaneously
- **Transfer learning**: Take part of a neural network used for an initial task, use that as an initialization for a second, related task e.g. model trained on imagenet (Vision) or English Wikipedia (NLP)

Summary: Challenges of NNs

Complex models may need a lot of training data, though pre-training helps to an extent

Many hyperparameters to tune:

- Learning rate, number of hidden units, number of hidden layers, how to connect units, non-linearity, loss function, how to sample data, training procedure, etc.

Can be difficult to interpret the output of a system

- *Why* did the model predict a certain label? Have to examine weights in the network.
- Important to convince people to act on the outputs of the model!

NNs for NLP

Some interesting open research questions:

- How to use linguistic structure (e.g., word senses, parses, other resources) with NNs, either as input or output?
- When is linguistic feature engineering a good idea, rather than just throwing more data with a simple representation for the NN to learn the features?
- Multitask and transfer learning for NLP
- Defining and solving new, challenging NLP tasks

Steps in Building a Text Classifier

1. Define problem and collect data set
2. Extract features from documents
3. Train a classifier on a training set
4. **Apply classifier on test data**

Evaluations Measures

How to measure performance of your classifier?

Simplest option: **accuracy**

- # correct / # samples in test set

This is not always a good idea! Consider the following case:

- Suppose that only one of every twenty e-mails is a *spam* e-mail. What would the accuracy of a classifier that always predicts *non-spam* be?

Precision and Recall

Other commonly used measures:

Precision

correct / # predicted

Recall

correct / # of that class

Above can be class-specific:

e.g., Recall among spam class:

correctly identified spam / # spam in test set

Combining Precision and Recall

F1 is the harmonic mean of precision and recall

F1 = 2 * P * R / (P + R)

P: precision R: recall

Can combine P, R, F1 for each class:

- **Macro-average**: take the average *after* computing P, R, F1 for each class.
  - Weights each class equally. Good if all classes are equally important. E.g.
- **Micro-average**: take the sum of the counts first, then compute P, R, F1
  - Weights each sample equally.

Confusion Matrix

It is often helpful to visualize the performance of a classifier using a confusion matrix:

Predicted class

Actual class

	C1	C2	C3	C4
C1	count	count	count	count
C2	count	count	count	count
C3	count	count	count	count
C4	count	count	count	count

Hopefully, most of the cases will fall into the diagonal entries!

Exercise

Predicted

Actual

	Spam	Non-spam
Spam	5	10
Non-spam	15	20

- Compute the precision, recall, and F1 for each class from the confusion matrix.
- Compute the macro-averaged and micro-averaged P, R, and F1
- Compute the accuracy.