Machine Learning Foundations
Unit 7: Use ML for Text Analysis

Table of Contents

- Watch: Introduction to Neural Networks
- Read: Linear and Nonlinear Transformations
- Watch: Designing a Neural Network
- Read: Explore Activation Functions
- Watch: Training a Neural Network
- Read: Formalize Stochastic Gradient Descent
- Activity: Design and Explore Neural Networks
- Tool: Neural Network Cheat Sheet
- Read: Python Packages for Neural Networks
- Code: Implementing a Neural Network in Keras
- Module Wrap-up: Introduction to Neural Networks

Module Introduction: Introduction to Deep Learning

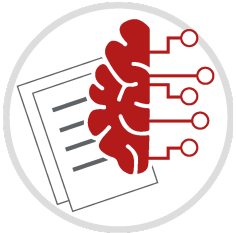
- Ask the Expert: Francesca Lazzeri on How Deep Learning Is Used in Real Life
- Read: Review Recurrent Neural Networks
- Tool: Recurrent Neural Network Cheat Sheet
- Quiz: Check Your Knowledge: Neural Networks
- Ask the Expert: Brandeis Marshall on AI Social Implications
- Assignment: Unit 7 Assignment - Using a Pipeline for Text Transformation and Classification
- Assignment: Unit 7 Assignment - Written Submission
- Module Wrap-up: Introduction to Deep Learning
- Assignment: Lab 7 Assignment

Unit 7 Overview
Video Transcript

A lot of data collected by companies comes in the form of text. From online reviews in e-commerce to posts on social media networks, data in the form of text powers a lot of the consumer products we interact with. The sub-field of machine learning that manages text is called natural language processing, or NLP for short. In this week's material, we will present an introduction to NLP. We will show a few common ways for how to convert text data to numeric data so that we can use it for machine learning applications. We'll present this in the context of different NLP applications as well. We'll also cover high-level details of advanced NLP applications that use different types of state-of-the-art neural networks. By the end of this week, you should be able to build and optimize ML models for common NLP tasks.

What you'll do:

- **Use various NLP preprocessing techniques to convert text to data suitable for machine learning**
- **Understand how word embeddings are used to convert text into numerical features without losing the underlying semantic meaning**
- **Implement ML models that make predictions from text data**
- **Explore feedforward neural networks**
- **Discover how deep neural networks are used in the NLP field**
- **Implement a feedforward neural network for sentiment analysis**



Unit Description

Natural language processing (NLP) is a branch of artificial intelligence that helps machines process and understand human language in speech and text form. In order for machine learning models to process words and blocks of text, the text must first be transformed into numerical features. There are various NLP preprocessing techniques that accomplish this.

In this unit, you will explore these techniques and the typical workflow for converting text data for NLP. You will also use a special scikit-learn utility that allows you to automate the workflow as a pipeline.

At the end of the unit, you will have the opportunity to explore neural networks, powerful ML models that are heavily used in the field of NLP. You will also discover different Python packages used to construct neural networks and see how to implement a feedforward neural network using Keras. You will then delve into deep neural networks, which are used to solve large-scale complex problems, and you will implement a deep neural network for sentiment analysis.

Back to Table of Contents

Tool: Unit 7 Glossary

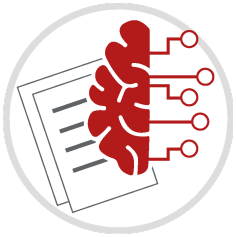
Though most of the new terms you will learn about throughout the unit are defined and described in context, there are a few vital terms that are likely new to you but undefined in the course videos. While you won't need to know these terms until later in the unit, it can be helpful to glance at them briefly now. Click on the link to the right to view and download the new terms for this unit.

Download the Tool

Use this **Unit 7 Glossary** as a reference as you work through the unit and encounter unfamiliar terms.

Back to Table of Contents

Module Introduction: Use Text as Features



Many real-life use cases of machine learning involve text data. In order for machine learning models to process text data, the data has to be transformed into a numeric format without losing its underlying meaning. In this module, Mr. D'Alessandro will discuss the workflow for converting text data into numeric features using natural language processing techniques such as tokenization and vectorization.

At the end of this module, you will walk through a demo in which you will use a vectorization technique to preprocess a book review data set for sentiment analysis, a subset of NLP that determines if text has a positive, negative, or neutral tone.

Back to Table of Contents

Watch: What Is Natural Language Processing?

In this video, Mr. D'Alessandro introduces the subject of natural language processing (NLP), which is a branch of AI that enables computers to understand language in the form of text and spoken word. Most NLP applications work with text data. Watch as Mr. D'Alessandro describes the motivation and use cases behind NLP as well as the standard pipeline for executing an NLP task that begins with raw text and ends with an NLP machine learning model.

Video Transcript

I'm going to present an overview of using text as data for machine learning applications. This topic is commonly known as natural language processing, or NLP for short. A lot of real-world data comes in the form of text, as this is a natural way for people to interact with products and systems. So it should be no surprise that we have specific methods within machine learning used to process text. Many companies create valuable products with NLP. Automated translation services are one of my particular favorites, and most social networks use the text and post to rank newsfeeds to make them more relevant to you. Further, companies that show product reviews might use NLP to highlight such reviews that might be the most useful to browsing customers. Many of these NLP applications are performed with standard machine learning methods, where the text-specific methodology happens in the preprocessing stage. I'll present a few of these applications and provide an overview of two different strategies we can use to convert text to numeric data. Beyond what I've just said. There are many other interesting applications of NLP. Many processes and systems generate text, so it is natural to find the need for NLP in many domains. This table here covers a lot of the types of common applications you'll see in use today. In each application, I list the methods most commonly used for that particular application. Most are being solved by specialized algorithms called recurrent neural networks, or RNNs for short. There are other methods like sequence-to-sequence modeling and generalized adversarial networks that are a specialized version of what we call deep neural networks. These specialized algorithms are the state of the art, and they're fairly sophisticated and beyond the scope of an introductory course on machine learning. That does not mean we cannot get very far by using just the machine learning

methods we've already taught in this course. The first two examples here can be efficiently solved using standard classification techniques. The first application, sentiment analysis, is commonly represented as a binary classification task. Topic modeling has both supervised and unsupervised variants. If we have labeled data, this is commonly approached using A multiclass classification method. When we solve NLP problems with standard machine learning techniques, the tech-specific components are in the data preprocessing. The general strategy is to follow a common pipeline, which I have shown here. The first step is called tokenization, which is where we map every word in our training data to a future position. Next, we'll do some preprocessing. We don't often want to keep the original text in its entirety or in its original forms. A lot of common words like "the" and "and" don't add much predictive value, so it is normal to remove them before doing any modeling. Some words also have similar meanings but different forms, such as the words "learn" and "learning," so we might want to convert them to the same form to get better results. After preprocessing is complete, we have to attach a numeric value to each word to make it a feature. Once this step is done, we end up with a standard data matrix that can be used in classification tasks. I'll cover each of the above pipeline steps in more detail in future lectures. I want to end, though, with a summary of two common strategies for mapping individual word tokens to a number. The first is called vectorization, which is the simple process of representing the binary presence or frequency of a word in a given text example. The other process is called word embedding. With embedding, each word can be represented by a K-dimensional vector, where those factors are commonly pre-trained and available as a lookup table for you. I will cover both methods in future lectures, so I'm leaving the descriptions at a high level for now.

Back to Table of Contents

Ask the Expert: Mehrnoosh Sameki on Natural Language Processing

Natural language processing (NLP) is broadly defined as the manipulation of a natural language, especially *at scale*, where you need to work with hundreds of books at a time, thousands of emails, or millions of product descriptions. In this video, Dr. Sameki explains its importance and why it is needed with examples.

Note: The job title listed below was held by our expert at the time of this interview.

Mehrnoosh Sameki
Product Manager, Microsoft

Dr. Mehrnoosh Sameki is a Senior Technical Program Manager at Microsoft, responsible for leading the product efforts on machine learning interpretability and fairness within the Open Source and Azure Machine Learning platforms. Dr. Sameki also co-founded Fairlearn and Responsible-AI-widgets and has been a contributor to the InterpretML offering.

Question

What is NLP and why is it needed? Can you provide examples of NLP being used in the real world?

Video Transcript

Natural language processing or NLP is a subfield of artificial intelligence. It helps machines understand and process the human language so that they can automatically perform some repetitive tasks. One thing to note is the language of humans and computers are completely different. And programming languages often exist as intermediaries between the two. When you look at humans, we speak and write in a very nuanced, and sometimes even ambiguous way. While computers are entirely logic based, following the instructions that they're programming to execute. So the difference means that traditionally, it is very hard for computers to understand human language. And so this field, natural language processing, aims to improve the way computers understand us, both when we're talking and when we're writing. And

so NLP uses techniques like artificial intelligence and machine learning, along with computational linguistics to process data, both text data and voice data, derive meaning from it, figure out intent and sentiment, and, of course form a response back to you. So let's go through some examples. Many of them are very familiar to you because you're using it on a daily basis. One of them being email filters. A spam filter specifically is probably the most well-known email filter. And something which is interesting is it's estimated that 85% of total global email traffic worldwide are essentially spams. And so behind that spam predictor filter that you have in your mailbox, there is an NLP at work. What it does is as emails come into your inbox, they're automatically scanned using text classification and keyword extraction. And so then, after that, based on the patterns that the NLP finds in those emails, it classifies them as a high chance for spam versus not. Another good example is search engine results. We all use them on a daily basis, multiple and multiple times. They no longer use just keywords to help you understand, or search your results, or figure out and reach your results. In fact now, they take your search queries, they analyze your intent when you search for information, and then through NLP, they extract and provide the best results that help cover the context that you wanted to cover. And another known one is language translation. Many languages don't allow for straight translation and have different orders of sentence structure and so traditional translation tools used to overlook a lot of these grammatical and different sentence structures. But the new ones in Bing Translate and Google Translate and other tools using NLP to translate languages way more accurately and present grammatically correct results to you.

Back to Table of Contents

Ask the Expert: Oleg Melnikov on Natural Language Processing

In these videos, Dr. Oleg Melnikov reveals several ways you already encounter NLP when you interact with various technologies. He briefly outlines the various use cases that different industries might have for incorporating NLP into their business and discusses how the building blocks of a language are combined into components that can be used in NLP.

We recommend that you watch these interview videos in full screen.

Note: The job title listed below was held by our expert at the time of this interview.

Oleg Melnikov
Visiting Lecturer of Computing and Information Science, Cornell University

Dr. Oleg Melnikov is a Visiting Lecturer of Computing and Information Science at Cornell University. He received his Ph.D. in Statistics from Rice University, advised by Dr. Katherine Ensor on the thesis topic of non-negative matrix factorization (NMF) applied to time series. He currently leads a Data Science team at ShareThis Inc. in Palo Alto, CA, and has decades of experience in various fields, including mathematical and statistical research, teaching, databases and software development, finance (portfolio management and security analysis), and adtech.

Dr. Melnikov's academic path began with a B.S. in Computer Science (and some years in pre-med); now, he holds Master's degrees in Computer Science (Machine Learning track) from Georgia Institute of Technology, Mathematics from UC Irvine (where he was in the Math Ph.D. program), and Statistics from Rice University, as well as an MBA from UCLA and a certificate in Quantitative Finance (MFE equivalent). Passionate about education and hard sciences, Dr. Melnikov has taught statistics, machine learning, data science, quantitative finance, and programming courses at eCornell, Stanford University, UC Berkeley, Rice University, and UC Irvine.

Question 1

What is natural language processing?

Video Transcript

So what is natural language? It is a form of written and verbal communication evolved naturally by humans through use. Some examples include languages like English, Russian, Japanese, Hindi, Chinese, and many dialects of these languages, as well as emojis and memes. On the other hand, we have constructed languages. These are purposefully developed by humans to have a particular grammar and vocabulary. Some examples of this include programming languages like Python and R; structured query languages, SQL. Web search queries are also in that category. Some of the spoken constructed languages are Esperanto and others. These are developed to communicate similar to natural languages, but they are specifically constructed. We use natural language processing to standardize and analyze text at scale. This includes cleaning and preprocessing text to reduce noisy typos, removing unimportant words and phrases, parsing text into meaningful words and sentences. It also includes converting text to numeric representations so we can perform statistical modeling on text more easily. It also includes computing a degree of association among documents to measure their similarity or dissimilarity. Drawing structural and semantic meaning from text and organizing documents to search them more efficiently. What about NLP in practice? What use cases do we know? There are several use cases of NLP and those include querying contents of trillions of web pages and documents, such as HTML, text, PDF files, PowerPoint presentations. Recommendation systems use NLP for recommending products and services; for example, "If you liked this particular product, you might also like this one." Language translation can be done at scale. We don't need to hire a single interpreter to help us translate from Russian to Chinese. We can translate using different translators on the web. Spelling and grammar correction can be done in the tool in the editor that we use to type documents. Question-and-answering systems, chatbots, and automated customer support can be used to answer questions for millions of customers at the same time. Text summarization and compression is used to shrink down the documents of large-sized publications, books, into just a few words or sentences, and we can visualize or read them more efficiently. There's also product categorization; that is a hierarchical structure of relationships or taxonomies between different products. This allows us to relate something like a keyboard to, say, a webcam, which are not in the same category, per se, but they both relate to a laptop.

Question 2

What is some NLP terminology?

Video Transcript

What are some basic elements of a natural language? Typically, the language starts with symbols that are hierarchically grouped to represent the largest structures. In English, we have 26 letters and some punctuation. This can be formed together into words like "one," "two," "three." More complex words can also include non-characters or non-letters, such as "Los Angeles" has a space. "Up-to-date" has hyphens or dashes in between. "5-hour" can be a single word as well with a number in it. The words can have spaces, hyphens, numbers, accent marks, and other characters. They are not just defined by letters. The 26 letters that we have can create a tremendously large number of words. Oxford Dictionary has 170,000, and there are many more short-lasting words that people create every single day through use in chat communities, for instance, or tweets, or memes, emojis, abbreviations create millions of words that live for a day. Words are combined into millions of phrases, which are assembled into trillions of sentences, which are united into paragraphs, which are clumped into chapters and articles, which are aggregated into documents which are collected into corpus, which are clustered into corpora. By the way, a corpus is just a large set of text documents. In fact, we loosely use these terms and often overload the term "documents." "Document" can be a paragraph or a sentence in NLP. Such hierarchical organization of textual elements is easier to learn, use, and maintain. All this structure is amazingly rich and practically infinite dimensional. Essentially, a document is just a long sequence of characters. Each element starts and ends with some identifier. A symbol might be just an uninterrupted stroke of a pen. There are some exceptions; there's the letter i, j, semi-colon, colon. But the words are identified by spaces or punctuation or tab characters, \t symbol. A sentence typically ends with a quotation mark or a period, a question mark, exclamation, ellipsis. Paragraphs are separated by invisible characters, such as newline character, \n, or a carriage return, \r, which is typical for Microsoft Windows systems. Documents have headers and titles and so on. By the way, the invisible characters, such as blank spaces or \n, \r, \t that I mentioned earlier, are called whitespace and are identified with a \s character. While there are algorithms that operate on long strings of characters, most classical NLP starts with tokenization. In this process, we split or parse the string into tokens, which are

contiguous groups of characters. Commonly these are words and sentences, but they can also be sub-words, characters, phrase tokens, and paragraph tokens. For example, we could have a sentence, "NLP is fun. I like it a ton" tokenized into characters, or words, or a couple of sentences.

Back to Table of Contents

Watch: Preprocessing Text With Three Methods

In this video, Mr. D'Alessandro takes a deep dive into three text preprocessing steps: lemmatization, n-gram, and stop words. He presents examples of each preprocessing method and the tradeoff when applying these methods.

Video Transcript

Data preparation for NLP starts with tokenizing text and then running applicable preprocessing techniques on the tokenized text. The first step to building tokens is to scan text in each example, parse out each individual string into a token, and create a mapping from token to feature ID. We usually apply this to individual words, but we can also apply this to individual characters or combinations of either. For now, we will focus just on words. The images here show the input and output of that process for a simple set of text examples. There is nothing special about the token-to-feature-ID mapping here either. Usually we would just assign feature IDs in the order we first encounter the particular token. Also notice that we left out the word "the." This was by design. We often leave out certain types of words because they take up memory without adding much predictive value. We'll cover this idea later. After building the token-to-feature-ID dictionary and applying preprocessing techniques, we would build a data matrix such as what we have here. The simplest approach is to assign a binary value on whether or not the specific word is in that example. This is how we get numeric features. The picture here shows exactly this for a simple example. Notice that the feature ID in the mapping corresponds to the feature column in the data matrix. At this stage, the text would be suitable for any machine learning training algorithm. I mentioned the idea of preprocessing a few times but didn't specify exactly what that meant. Three common techniques we'll cover here are stop word removal, lemmatization, and making n-grams. I'll cover each of these individually, but they all happen at the stage where we're parsing the text and creating the text-to-feature mapping. These will essentially determine what should be an individual string token and thus what should be included in the final feature set. A lemma is the canonical form of a particular word. When we do lemmatization in text processing, we are taking each word and converting it to some canonical form. Here are a few examples. The most common situation where this applies are when you're dealing

with different verb tenses or noun versions of a verb. We can also see in the middle example here that we apply it to possessive and plural forms of a noun. The motivation here is to reduce our memory footprint by reducing the feature set size without hurting model quality. In other words, can we collapse words with common meaning to be represented by the same feature? The next method in our pipeline is the process of creating n-grams. N-grams are just combinations of individual word tokens as shown in this example. The most common choice of n would be two or three, which we would then call bi- and tri-grams. This example illustrates how a sentence could be broken out into n-grams. So in the bi-gram example, the combinations "this product" or "not great" would map to a new feature. When we use n-grams, we typically add them to our feature set along with the original words. So if our text corpus has about 1,000 or 100,000 unique words, adding n-grams where n would be greater than 1 would significantly increase the number of unique tokens that become features. The motivation here is to capture more nuanced meanings and expressions that require multiple words to understand. For example, in this case I highlight the bi-gram "not great." Think about whether we wanted to determine if the sentiment of the text was positive or negative. The word "great" alone likely signifies positive sentiment. But when "great" is modified with "not," it reverses the sentiment to become negative. The bi-gram approach allows us to build in this additional nuance of the language to our feature set. Another example of a bi-gram that adds such nuance is the bi-gram "Big Apple." This is a phrase with a very specific meaning associated with a place which would be lost if we only considered the individual words "big" and "apple." The last preprocessing step to cover is this idea of removing stop words. A stop word is just a token that appears very frequently in different examples of text but also adds very little predictive value. Usually we remove them to reduce data size and to speed up computation. The easy examples are the conjunctions "and" and "or" or the articles of speech "the" and "a." Most text processing packages offer prespecified stop word lists associated with common languages. With a prespecified stop word list, the processor just removes a token if it is in the list. The other approach is to account the document frequency or the number of times a word appears across the examples in your data set. If it exceeds a certain level, we might remove it. There's another condition we can consider, which is remove the word if it appears too infrequently. This technically would not be called a stop word but it can often be done within the same process. So in addition to removing very common tokens, we remove

rare tokens. We might do this because if a given token appears and only a few documents or examples, it won't add much predictive value but it does take up space in memory. To recap, I've presented three different preprocessing methods. Within each one, we have different options we can choose. It is helpful to think of all of this as a parameterized pipeline, where the parameters include which preprocessing techniques to include in the first place and also individual parameter specifications of the specific methods. Here are two example pipelines to refer to. Choosing an appropriate pipeline is usually an empirical question. The simplest approach is to try a few different pipelines and choose the one that yields the best performance. The general goal is we want to build a feature set that captures as much predictive value as possible but isn't so large that it becomes computationally infeasible to run or becomes too likely to overfit. If you could keep that goal in mind, you should have the right intuition to design a small but reasonable set of pipelines to test and then run standard model selection techniques to find the optimal one.

Back to Table of Contents

Watch: Using Vectorizers to Convert Text to Numerical Features

After you parse and preprocess text, it is ready for vectorization. Vectorization is a process of converting a token into a numeric value. In this video, Mr. D'Alessandro introduces the three common approaches to vectorizing a token: binary, count, and term frequency-inverse document frequency (TF-IDF). Follow along as he explains how each one works in detail and how a vectorizer is implemented in scikit-learn.

Video Transcript

In this video, we will cover the common techniques for turning our word tokens into numeric features. I'll cover three simple methods that are commonly referred to as vectorization. As shown here, we'll assume that the text has already been appropriately parsed and any preprocessing has been done. If we think of tokenization and preprocessing as the steps that define what a feature is, the vectorization tells us what numeric value the feature should take on. So there are three common forms of vectorization to consider, which I present here in order of increasing sophistication. The binary version is to just use the presence of the token in a document as the numeric feature value. Remember the word "document" here is the convention we use for a single example. The next method is to use the count, so if a word appears four times in a given document, the associated numeric value would be 4. The last method I'll present is called term frequency-inverse document frequency, or TF-IDF for short, and this builds on the count method. We start by computing the term frequency of the token in the document and then divide that by the frequency in which the token appears across all documents. The TF-IDF is motivated by heuristics, but it is a very predictive and useful method. If we think through the intuition behind the numerator and denominator of the computation, we can see why it works. The numerator is the term frequency in a given document, so when a word appears a lot in a document, its importance to that document goes up. On the other hand, if that same token appears in most documents, then its relative importance to a given document goes down. There are multiple specialized Python packages we can use for text processing, but the best place to start is scikit-learn. Here's an example use of a TF-IDF vectorizer from scikit-learn. The process is similar to what we usually see in model building. We start by instantiating the object. In this, a

lot of the input parameters dictate the preprocessing that will be done. Next, we fit the vectorizer. Here we pass in the training data, and this is where the code is applying the preprocessing, the tokenization, to build the text-to-feature mapping. The last step is to transform the data. The transform step takes the text data and converts it to a data matrix. Notice that we had to both fit and transform the data here. I have also added one step to transform the test data. It is important that whatever processing and vectorization we do on the training data is exactly what we apply to the test or validation data. We can guarantee that all transformations are the same by using the exact same fitted vectorizer object when we do transformations on all three of these data sets.

Back to Table of Contents

Ask the Expert: Mehrnoosh Sameki on the NLP Pipline

In this video, Dr. Sameki describes the phases of implementing a machine learning model when you are solving an NLP problem.

Note: The job title listed below was held by our expert at the time of this interview.

Mehrnoosh Sameki
Product Manager, Microsoft

Dr. Mehrnoosh Sameki is a Senior Technical Program Manager at Microsoft, responsible for leading the product efforts on machine learning interpretability and fairness within the Open Source and Azure Machine Learning platforms. Dr. Sameki also co-founded Fairlearn and Responsible-AI-widgets and has been a contributor to the InterpretML offering.

Question

What are phases of implementing a ML model when solving an NLP problem?

Video Transcript

There are some key steps for you to train your natural language processing models. Let's go through these stages and provide some examples. The first one is always a good idea to clean your data. So let's go through data cleaning. One of the main ways for you to do data cleaning is to remove stop words. There are a few words which are very commonly used when humans interact, but they actually carry no extra value, and they don't make any sense to the computers. So if you think about the words "a," "and," "the," "at," "on," "up," some of these words necessarily are not adding much to the value, or they're not adding much value to the sentence that the computer is understanding. The other technique is making lowercase. This is required to make all the words in your sentence kind of maintain some form of uniformity. Next, we have stemming and limitization. Stemming means you're reducing the word into its very root. So for instance, if you have the word "affecting," "affects," "affection," "affective," that is bringing it back to the word "affect," which is the root of all of these. Then we have limitization, which helps to reduce the word into a single form. For instance, if

you have the words "gone," "go," "going," "went," then it's all back to "go." Then we have part of speech tagging. This helps to identify the parts of speech. So for instance, if we have the example of the dog kill the bat, "the" is definition article, "dog" is noun, "kill" is verb, "the" is definite article, and "bat" is a noun. And last but not least, under data cleaning, we have name entity recognition. This helps to identify and categorize the different groups which includes names, places, currency, et cetera. So if I give you a sentence, "Microsoft CEO Satya Nadella resides in Seattle," the name entity recognition returns to you, Microsoft is an organization, Satya Nadella is a person, Seattle is a location. After you clean your data with these techniques, you move to tokenization. This is one of the common practices while working on text data. This helps to split a phrase, sentence, or paragraph into smaller units, like words or terms. Each unit is called a token. Right after that comes vectorization or word embedding. Once cleaning and tokenizing is done, extracting features from the clean data is very important as the machine usually don't understand the words, but numbers. Vectorizations help to map the words to a vector of real numbers, which further helps the predictions. This helps extract also the most important features in your text. And so two very known techniques under vectorization is first count vectorization, which means that counting the number of times a particular word appears in the document, and then another known one is TFIDF, Term Frequency Inverse Document Frequency, which provides an overall weightage of a word in the document. And so it helps you get this weighted score. Finally, when vectorization is done and word embedding is done, you go into the last stage, which is the actual model development. And now we have either count-based or TFIDF matrix. And so you're passing it to a machine learning model and using any technique that you choose or any algorithm that you choose. Your natural language processing, which could be a text classification or any other machine learning type, is trained.

Back to Table of Contents

Code: Transforming Text Into Features for Sentiment Analysis

In this demo, you will create a binary classifier that performs sentiment classification of book reviews to determine whether a given book review is a positive or a negative one. You will discover how to use scikit-learn to convert book review text into numerical feature vectors using TF-IDF (term frequency-inverse document frequency). Using TF-IDF features when performing classification for sentiment analysis yields more accurate results than other methods, since TF-IDF consists of words that provide an understanding of the context of the textual data.

After converting the text into numerical feature vectors, you will train a logistic regression model using these feature vectors. You will also experiment with using different document-frequency values and observe how this changes the performance of a logistic regression model.

For this activity as well as other activities and assignments in this unit, you will work with a new data set that we will call the Book Review data set. It derives from the **Multi-Domain Sentiment** Data Set (Version 2.0) from "Biographies, Bollywood, Boom-boxes and Blenders: Domain Adaptation for Sentiment Classification. Association of Computational Linguistics (ACL)" and contains product reviews taken from Amazon.com for many product types, but we will only consider the book reviews.[1]

This activity will not be graded.

The full contents of this page cannot be rendered in the course transcript. Log in to the course to view it.

1. Blitzer, J., Dredze, M., and Pereira, F. "Biographies, Bollywood, Boom-boxes and Blenders: Domain Adaptation for Sentiment Classification." *Association of Computational Linguistics* (ACL), 2007. [PDF]

Back to Table of Contents

Watch: Using Scikit-learn Pipelines

The end-to-end process for building a text classifier takes many steps, all the way from text parsing to model evaluation. In this video, Mr. D'Alessandro demonstrates how scikit-learn can be utilized to string together a series of steps into a pipeline. You will see how easy it is to use scikit-learn to build an automated pipeline that contains steps from tokenization all the way to model evaluation.

Video Transcript

Building a text classifier certainly involves a lot of steps. Before we do any model selection, we need to perform a series of preprocessing steps to convert the text into numeric data that is compatible with most machine learning algorithms. When we compare this to modeling tasks that don't use text data, we can see that this preprocessing is similar to the standard data exploration and feature engineering steps we typically do. From that perspective, the preprocessing work isn't much additional work than standard data preparation. With text-only data, the sequence of preprocessing to modeling can actually be automated. Automation helps us explore a range of design options and choose what is best for the application. Scikit-learn provides packages that enable us to combine preprocessing and model selection into one convenient set of steps. The package we're going to learn here is called pipeline. A scikit-learn pipeline is a series of transformers followed by an escalator, as shown in this conceptual example. A transformer is technically any scikit-learn class with a transform method. In regular terms, it's a class that transforms data, usually from a more raw form to a form more suitable for some modeling application. The last step in the pipeline is called an estimator, and this is usually one of your typical machine learning models. This pipeline approach is very useful when we want to combine the preprocessing and model selection steps into a streamlined and automated process. Combining the steps like this makes optimizing the end-to-end process, again, fully automatable. The other advantage is that using a pipeline ensures that the preprocessing we do for training data is identical to the preprocessing we do to our test data. Here are some example pipelines that would be common in text-based modeling. In the prior conceptual example, I showed three transformers. Here we only need one. This is because a lot of the text preprocessing techniques are performed by

the same transformer. When we call the TF-IDF vectorizer in scikit-learn, we convert raw text into a data matrix where the values are determined by the TF-IDF process. This data then gets passed into a logistic regression for classification. I am showing two separate pipelines here. Notice that each uses the same type of transformer and estimator, but the input parameters are different. Remember, we have a lot of design hyperparameters to consider in this entire process. Finding the optimal configuration of preprocessing and model hyperparameters can be a time-consuming process. Luckily, with the pipeline approach, we can significantly streamline it. The first thing we need to do is set up the pipeline. Here is a demo of this using actual code. You can see here we have our import step and then one instantiation of the pipeline class. We specify the pipeline as a list of steps. Each individual item in the list is either a transformer or an estimator. But the last step always needs to be an estimator. In this specific case, we have a TF-IDF vectorizer to logistic regression pipeline. Notice we include hyperparameters of each step when we build a pipeline. Once set up, we can treat the pipeline as a standard model object. In the second block of code, we first fit the pipeline. The input here is raw text data. The raw text is converted to numbers using the TF-IDF vectorizer, and then we build a logistic regression on top of that. In the last step here, I am making a prediction against the raw text. I hope this really illustrates the simplicity of this process. With the pipeline approach, we can apply multiple types of preprocessing to our data, fit a model, and then predict with new data in just three easy steps. Now let's take the pipeline approach and combine it with model selection. This block of code represents a fully complete model building process. It is amazing to me that we can do all of this in just a few lines of code. At the top, we import the scikit-learn packages we need and then split our data. We then set up our pipeline and the instructions for cross-validation. Notice here in the pipeline we use the same steps as the prior example we just showed, but we left out the hyperparameters in each step. The section after that is where we specify all of the design hyperparameters that will be explored in cross-validation. This is a dictionary where the keys reference the pipeline steps and the parameters associated with those specific steps. In this dictionary, we have three hyperparameters, each with two options. When we pass to the grid search CV class, it is going to run a cross-validation pass on every combination of those three hyperparameters. In this case, will be exploring eight different configurations. We initiate the crossvalidation process using the FIT method. Once done, we can access the best performing model and use that to

make predictions or do evaluation. And just like that, we have a fully optimized text classifier.

Back to Table of Contents

Quiz: Check Your Knowledge: NLP

In this quiz, you will answer questions about various steps in the NLP process.

You may take this quiz up to three times.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

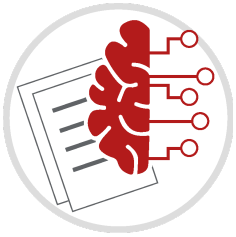
Back to Table of Contents

Module Wrap-up: Use Text as Features

In this module, Mr. D'Alessandro introduced natural language processing and the steps required to process text data for machine learning purposes. The processing steps include lemmatization, n-gram, and stop words. You explored three vectorization methods known as binary, count, and TF-IDF. Lastly, you discovered how scikit-learn can be used to conveniently string together some of these steps with model selection techniques.

Back to Table of Contents

Module Introduction: Understand Word Embeddings



An important concept for working with text data is the idea of word embeddings. Word embeddings can be utilized to convert text into numerical feature vectors without losing its underlying semantic meaning. In this module, Mr. D'Alessandro will explain how word embeddings work and how it relates to natural language processing and machine learning. You will discover how word embeddings reduce sparsity and how different pooling approaches can be used to capture different semantic concepts in the underlying text. At the end of this module, you will walk through a demonstration of using word embeddings for spam email classification.

Back to Table of Contents

Watch: Introduction to Word Embeddings

Another common approach for converting text to numerical features is with word embeddings. Word embeddings are generated by processing massive amounts of text data and determining probabilistic relationships between any given words and surrounding words. In this video, Mr. D'Alessandro introduces the idea of word embeddings and how they can be readily used off the shelf to convert text into numeric representations without losing its underlying semantic meaning. Mr. D'Alessandro will also demonstrate how to compute the semantic similarity between words using cosine similarity.

Video Transcript

In this video, I'm going to present a technique for converting text into features called word embeddings. Word embeddings are a general class of techniques that can be accomplished multiple ways. The actual computation of word embeddings requires sophisticated neural networks, which are outside the scope of this course. However, we can still leverage word embeddings in our work, because the methods have become so commonplace and popular that prebuilt versions are available to download and use. Now I will introduce us to the most popular word embedding methodology called Word2vec. I won't go deep into how Word2vec is built, but I'll share some basic intuition behind the method as well as show us how to use prebuilt versions in practice. Let's start by showing what the output of an embedding looks like. The example here shows a matrix with four words and their associated word embeddings. A word embedding is a K-dimensional vector of values that are typically in the negative to positive 1 range, though that is not an absolute requirement. The choice of K is up to you, but typical word vectors range from 50 to 300. A higher K usually means better-quality embeddings but also requires greater computational resources. Much like with other machine learning methods, a large vector size here gives you more modeling flexibility, but to use it effectively, you would need larger data sets. The values in the embeddings themselves are unit lists and the meaning of the values lies more in how they identify similarities between words. Here is another example of word embeddings associated with the words "doctor," "patient," and "cat." This time I used a real pretrained embedding that has 50 dimensions. As I just

mentioned, there isn't usually a clear meaning to the values of individual dimensions for a given word. Where we can find meaning, though, is in how two different word vectors or word embeddings relate to each other. Words that typically appear together in text or have common definitions tend to have similar word embedding vectors. Out of these three words, try to identify the two that had the most similar word vector patterns. This is naturally difficult to answer visually, but we can use a simple function to quantify similarity between two word vectors. The function we can use to compute similarity between two vectors is called the cosine similarity. Here I show the math behind the cosine similarity function as well as a Python function we can use to compute it. In the numerator, we take the dot product of two vectors, where the dot product is the sum of elementwise multiplication of the vector entries. The denominator is the product of both vector norms, where a vector norm is the square root of the dot product of a vector with itself. The cosine similarity has a range of minus 1 to 1 and has a geometric interpretation. The chart on the right shows three vectors and a reference vector, shown in black. The cosine similarity function is related to the angle between two vectors, and we are computing that between each line here and the reference vector. If the angle is less than 90 degrees, cosine similarity will be positive, and the smaller the angle, the closer the similarity is to 1. This is represented by the green vector here. Vectors that are completely orthogonal to each other, which means having a 90-degree angle between them, have a cosine similarity of zero. This is represented by the yellow vector. Vectors with angles greater than 90 degrees will have negative cosine similarities. This is represented by the red vector here. Now, to bring this back to word embeddings, words with similar semantic meanings will have word embedding vectors with high cosine similarity. Here's an example set of cosine similarities between ten words and two reference words, which are "doctor" and "cat." We can see here there are different ways to interpret semantic similarity. "Physician" has the highest similarity with "doctor," and these words are synonyms. "Patient" has the next highest similarity. These aren't synonyms but we know there is a clear relationship between "patient" and "doctor." When we look at "cat," "dog" has the highest embedding similarity; these two animals are common house pets. "Lion" also has a high similarity to "cat," and this is also a type of cat. What we can see here is that word embeddings can identify multiple types of semantic relationships between words. But a high similarity doesn't tell us exactly what type of semantic relationship

there is, so that is one limitation. Nonetheless, this is a very effective way to represent text in machine learning applications.

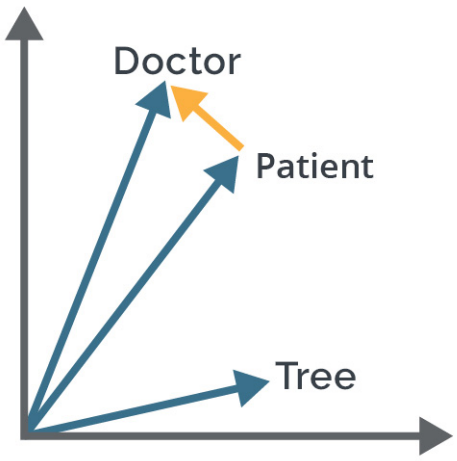
Back to Table of Contents

Read: Deeper Dive Into Word Embeddings

You just saw a high-level overview of word embeddings from Mr. D'Alessandro. Let's take a moment to delve a little deeper.

The word embeddings technique converts words to numerical vectors. For example, the word “doctor” may be mapped to a vector of $\langle 1.2, 3.0, \ldots, 4.0, 6.0 \rangle$. Word embeddings are similar to vectorizers, such as TF-IDF, in that they both convert text into numerical representations. The difference between the two is that word embeddings seek to capture the meanings of words within a body of text, while vectorizers simply convert words into numbers based on some predetermined rules.

The word embeddings technique essentially views words with similar meanings as being similar to each other mathematically. The technique learns relationships among words by analyzing a large corpus of text. It then creates numerical vectors by mapping related words in the input text, such as the words "doctor" and "patient," to similar vectors that are closely spaced in vector space. Words that are unrelated, such as "doctor" and "tree," have dissimilar vectors.



This approach encodes the semantic meanings of words since the assumption is that words with similar vectors have similar meanings. The algorithm behind how word embeddings creates the numerical vector representation is outside the scope of this

course, and as a machine learning engineer, you could simply use an an existing implementation of a word-embedding algorithm to convert words into numerical vectors. A popular one is the Word2Vec technique, which was created by Google in 2013. Word2Vec is a family of model architectures and optimizations that can be used to learn word embeddings from large data sets. It needs to be trained just like any other machine learning model, and the result of that training is vectors, one for each word in the training dictionary.

Sometimes word embeddings capture the meaning of words so well that the embeddings show interesting geometric properties, such as

$$W_{\text{king}} - W_{\text{man}} + W_{\text{woman}} \approx W_{\text{queen}}$$

To calculate the similarity between two words, cosine similarity is most commonly used. In the cosine similarity formula for two vectors V_1 and V_2 (of the same dimension), the numerator is calculated by taking the dot product of the two vectors and the denominator is calculated by taking the product of the L2 norms of the two vectors:

$$\text{Cosine Similarity} = \frac{V_1 \cdot V_2}{\|V_1\|\|V_2\|}$$

The L2 norm of a k-dimensional vector $V = \langle v_1, \ldots, v_k \rangle$ is defined by

$$\|V\| = \sqrt{\sum_{i=1}^{k} v_i^2} = \sqrt{v_1^2 + \cdots + v_k^2}$$

Geometrically, the L2 norm $\|V\|$ represents the length of V, and the cosine similarity of two vectors V_1 and V_2 is the cosine of the angle between V_1 and V_2. Vectors that are aligned in the same direction will have a cosine similarity of 1 (because the angle between the vectors is 0 degrees). When two vectors are aligned in the opposite direction, their cosine similarity will be -1 (because the angle between the vectors is 180 degrees). When the two vectors are orthogonal to each other, their cosine similarity will be 0 (because the angle between the vectors is 90 degrees).

Word embeddings are used by machine learning models, such as logistic regression, random forest, and neural networks, when analyzing new text. A machine learning model can learn the relationships between words using these embeddings and react

to a new word that it didn't learn in training based on its similarity to other words it has "learned."

How do we know when to use word embeddings vs. TF-IDF? One thing to consider is the size of the vocabulary and frequency of words. For a small vocabulary full of high-frequency words, TF-IDF is a good choice. For larger vocabularies full of low-frequency words, word embeddings are a good choice.

Back to Table of Contents

Watch: Using Word Embeddings

Word embeddings is a great tool for reducing the sparsity of the data set. While word embeddings is a convenient tool method for converting text into features, it does require preprocessing before it can be readily used by a machine learning model. In this video, Mr. D'Alessandro introduces the idea of pooling, which is an aggregation step for word embedding before it can be used by a machine learning model.

Video Transcript

I introduced us to word embeddings in a previous lecture. There we learned that word embeddings are a way to represent words as K-dimensional vectors and that word vectors with high cosine similarity tend to be semantically related. Now we'll need to learn how to put them to use in our models. But before formally introducing how to prepare your data with word embeddings, let me first provide some of our motivations for doing so. Word embeddings provide several advantages that are all related. The governing principle is that this method enables us to reduce our feature count from hundreds of thousands or millions of features to typically less than 300. As a starting point, this helps us reduce the computational and storage burden with the underlying data. Embeddings remove a common issue with standard text factorization techniques, which is data sparsity. A lot of words are rare, which makes learning patterns from them difficult. These words don't show up enough in examples for us to train a low variance signal on them. This is what I mean when I say data sparsity is an issue. Word embeddings let us pull similar words, which helps us reduce data sparsity while not throwing out any data. When we create features for word embeddings, we have to deal with one key challenge. A text example in our data is usually composed of multiple words. Since each word maps to a K-dimensional vector, each text example in our data is composed of multiple word vectors. At the same time, most training algorithms require us to use a single fixed-length vector as input for each example. So to use word embeddings in model training, we need to do some preprocessing. If we start on the left-hand side of this illustration, we first take each word in example and map it to its word embedding vector. That leaves us with output that looks like the middle column here. Then we need to leverage some aggregation method to reduce the indifferent vectors to a single vector, which is represented in

the last column here. The most common aggregation method is to take the elementwise average of each word vector dimension. As an example, if we have inwards, we take the first position of each word vector, compute the average, and then assign that to the first feature position in the final feature vector. I have this depicted in the left-hand diagram here. Notice I have written "average" or "max." We can generalize this aggregation idea to what we call a pooling layer. The pooling layer is our method for aggregating multiple vectors into one. I introduced the average function as one approach, but we can use any aggregation function. Other common aggregation functions might be max or min. The max or min aggregation techniques are more likely to capture important words that have more extreme values in any one dimension. Here, I expand upon the previous process and introduce two pooling layers as well as vector weighting. It is likely true that certain words are more influential in a given example than others. Using an average pooling layer might reduce the weight of important words in a particular example. We can overcome this limitation by using two pieces of functionality added within this diagram. We can first weight each factor before we aggregate them. This is depicted in the first step on the right-hand side, where we multiply the word vectors by a weight. One good weighting technique is to apply TF-IDF weights. We can first compute the TF-IDF weight for each word in each document and multiply the associated weights with the associated word vectors. This would reduce the impact of vectors associated with words that are more common across documents. After that, we can use two pooling layers instead of one. In this case, we are taking both an average and a max over the vectors. This process yields two vectors instead of one, though. We can then concatenate the two vectors as shown here. Instead of one final vector with a dimension of K, we might end up with one vector with a dimension of 2 times K. Having both max and average values represented here helps us capture the average meaning of the words while also highlighting meanings associated with larger word embedding values. Another important consideration is which embedding methods should you use? Like many design decisions, this becomes an empirical question that we can test. The first question, of course, is should you use embeddings in the first place? Once you go down that path, you need to think about the specific embedding model to use. There are many options available to download a pre-trained word embedding model. A key component of that decision is the size of the embeddings. Common options are 50, 100, and 300. Last, you need to think about the pooling functions and whether or not

to weight the words. There is no universally right answer on which designs to choose. Ultimately, you need to consider a few reasonable options and run empirical tests and validations to choose which is optimal for your given problem. The best approach is to start simple and iterate from there. For one thing, you can test one embedding method with a specified dimension and compare against a non-embedding method. If you find that one option is doing better, then you can iterate on that particular option until you reach a point of diminishing returns.

Back to Table of Contents

Quiz: Check Your Knowledge: Word Embeddings

In this quiz, you will test your knowledge of word embeddings.

You may take this quiz up to three times.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Code: Transforming Text Using Word Embeddings

In this demo, you will see how to use Word2Vec to transform a text document into word embeddings that can be used as features for a binary classifier that predicts whether an email is spam. You will then train a logistical regression model and evaluate its performance when predicting whether a new email is spam.

In this activity, you will work with a new data set based on the **Spam Mails Dataset** found on Kaggle. This dataset contains the subject lines of both spam and non-spam emails.

This activity will not be graded.

The full contents of this page cannot be rendered in the course transcript. Log in to the course to view it.

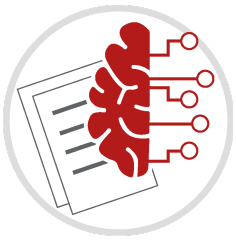
Back to Table of Contents

Module Wrap-up: Understand Word Embeddings

Word embedding significantly reduces the feature count and data sparsity, which makes it an ideal choice for machine learning. In this module, Mr. D'Alessandro showed you how word embeddings work in detail. You discovered how to use pooling to properly process word embeddings before using the embeddings in a machine learning algorithm.

Back to Table of Contents

Module Introduction: Introduction to Neural Networks



Neural networks are a class of supervised learning algorithms that can find complex patterns and relationships in data and solve complicated problems that other models cannot. While neural network-based machine learning algorithms have been around for decades, more recent architectures have exhibited impressive performance gains in large-scale machine learning problems. These architectures enable what has come to be known as a "deep learning" approach.

This module will provide you with the foundation for constructing and training a neural network in preparation for deep learning. You will also practice implementing a neural network for a classification problem using Keras, a neural network library for Python.

Back to Table of Contents

Watch: Introduction to Neural Networks

Neural networks are a type of supervised learning algorithm that can be used to tackle complex machine learning challenges, including classification and regression problems. In this video, Mr. D'Alessandro defines what a neural network is and explains what mechanisms neural networks use to solve difficult problems.

Video Transcript

The natural language processing methods we cover in this course can be thought of as beginning to intermediate. The use of vector risers and standard pre-processing techniques covers a lot of common text classification tasks such as sentiment analysis and topic classification. In this lecture, I want to introduce us to neural networks. Neural networks are machine learning models that are used in applications beyond just natural language processing. Most NLP problems today leverage what we call deep neural networks, which are specialized versions of neural networks. We'll start this intro with a focus on basic neural networks. Once you understand the fundamentals of the basic neural networks, you should be better prepared to understand how to construct and train deep neural networks. When the relationships between features and the label are linear, we can use a simple linear model to model those relationships. However, as relationships become more complex or nonlinear, we have to find another model to make predictions. The diagram here shows the difference between a linear and a nonlinear relationship between a feature and a label. This is where a neural network comes into play. A neural network is one popular class of models that can help us identify nonlinear relationships. Neural networks can learn very complex patterns in relationships amongst features and labels, and therefore they can solve very complex real-world problems. To put it simply, neural networks are a composition of simple linear and nonlinear transformations of the input data. In order to understand the neural network, you can think about it in terms of one of the linear models you have already learned, such as logistic regression. A linear model learns one weighted function or line that can be used as a model to make the predictions. Neural networks, on the other hand, extend this using many linear functions coupled with nonlinear transformations of those functions so that the final combination of them is a nonlinear relationship. This method is more flexible

because it allows data to be modeled with more complex weighted functions again, thus giving a neural network the ability to solve very complex problems dealing with nonlinear relationships between features and labels. This diagram shows the general structure of a neural network. Overall, a neural network performs the same tasks as other supervised learning methods. What we call the output layer here is usually a function that outputs either a probability or a regression estimate. What we call the input layer here is just the first set of functions that read our features in and output some transformation of them. The hidden layers perform consecutive transformations on the input from the previous layer. The process of inputting data, making transformations, and arriving at a prediction is called forward propagation. For that reason, these basic neural networks are most commonly referred to as feed forward neural networks. The transformation happens in each of the circles, which we call nodes of the network. Here is a close-up of a single node. We can think of this as a neural network that has one layer and a single node in that layer. First, the inputs are multiplied by a weight and then summed up. This is a simple linear transformation similar to logistic regression, where again, we multiply features by their corresponding weights. The weighted sum is then passed into what is called an activation function. There are different types of activation functions, and choosing them is part of the design process. The activation functions are the core mechanism that helps us capture nonlinear relationships in our data, and this is what distinguishes a neural network from something like a logistic regression. The output of the activation function is just another number that gets passed as input to the next layer. This is happening in every single node. One reason it is called an activation function is that the output is either zero or not. When not zero, we say that the node is activated. A neural network with many layers and nodes per layer is performing millions of computations that transform the data. This gives a neural network tremendous power in modeling complex problems.

Back to Table of Contents

Read: Linear and Nonlinear Transformations

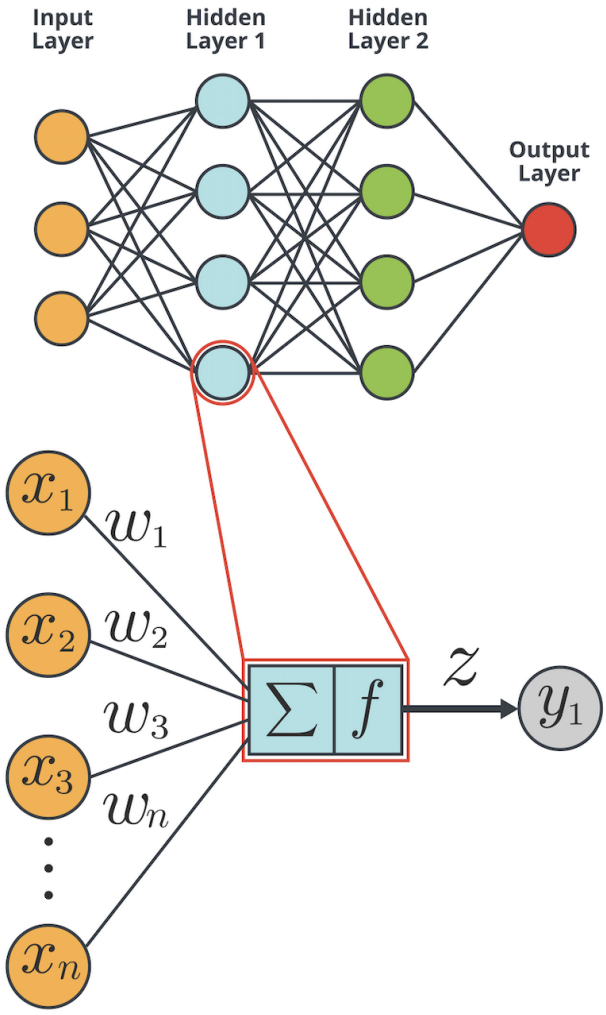
You have just observed how a neural network follows the pattern of linear models in that it computes a weighted sum of inputs, which are then learned by the model. Logistic regression learns one weighted function that can be used as a model to make predictions. Neural networks, on the other hand, use many linear functions so that the combination of them all is nonlinear. This method is more flexible because it allows data to be modeled with more complex weighted functions, thus giving a neural network the ability to solve very complex problems dealing with nonlinear relationships.

You have also discovered that in a neural network, every node in a hidden layer receives input from all nodes in the previous layer. But what happens inside these nodes, or neurons? Each neuron applies what is called an activation function to the sum of weighted inputs coming from the previous layer's neurons.

As stated, a neural network is really just a series of linear and nonlinear transformations. It involves two things: **a linear function** and a **nonlinear activation function**.

First, a linear function is applied to the input layer. Again, this corresponds to what you have implemented in other linear models: a matrix vector multiplication that involves multiplying input features by their weights and adding up the results.

The output of the linear function is then amplified (or *damped*) by a nonlinear activation function. This is where the term "firing" of a neuron in a neural network comes from. The activation function takes the output of the linear function (the summation value) and essentially determines how little or how much the linear function should contribute to the next layer in the network. It transforms the summation value accordingly. For example, it can determine that the linear function should not contribute to the next layer and therefore "transforms" the weighted sum to 0. The output of the function is the output of the neuron; in the case we just described, the neuron essentially "turns off" and passes on 0 to the next layer.



This linear and nonlinear transformation process continues for every hidden layer. All of the neurons in one hidden layer output a value. These collective values become input to every neuron in the next layer, resulting in more linear transformations, which is then followed by another application of nonlinear transition functions, and so on. It is the nonlinearity of the transition functions — combined with the size and depth of the network — that allow neural networks to model very complicated relationships.

ReLU: Nonlinear Activation Function

There are a few different activation functions you can use, and you can choose an activation function based on your problem and what you are trying to predict. One of the most commonly used activation functions is **ReLU** (rectified linear unit). ReLU is defined as $\sigma(z) = max(z, 0)$. If its input is less than 0, it outputs 0; otherwise, for any positive input value z, it outputs z. Therefore, if the weighted sum it receives from the linear function is less than 0, it can "turn off" and pass along a 0, otherwise it can simply pass along the same weighted sum it received.

Back to Table of Contents

Watch: Designing a Neural Network

There are certain choices you can make when designing a neural network architecture. In this video, Mr. D'Alessandro discusses the three core hyperparameters of a neural network that you can tune and demonstrates how to determine whether you have chosen the optimal configuration.

Video Transcript

When building a neural network, you get to play the role of architect. If you recall, all machine learning algorithms have what we call hyperparameters, and these control the overall complexity of the algorithm. In neural networks, the architectural elements act as hyperparameters. Part of the art of building a neural network that generalizes well is identifying the right architecture. In this video we'll cover the main elements of the neural network architecture. These include three core components. The number of hidden layers, the number of nodes within each layer, and the activation function we use within each node. When we train a neural network, we will use variations of the optimization algorithm, gradient descent. The gradient descent algorithm also has hyperparameters, but we do not cover that in this video. In this diagram, we illustrate example neural networks with increasing numbers of hidden layers. In this case, each network has two features, which again maps to the size of the input layer. Each hidden layer has three nodes. Again, the transformations of the last hidden layer is then passed to the output layer, which produces the final prediction. Deep neural networks are called deep because they typically have more hidden layers. There isn't a specific threshold in which a network becomes deep. The common deep network architectures can have 10 or more layers. Once we choose the number of hidden layers, we can set the size of them. This illustration shows two networks, one hidden layer, but with different numbers of nodes. Each hidden layer can also have different numbers of nodes. Last, we consider the activation function. The activation function defines how the data in each node is transformed. Shown here are three most commonly considered activation functions. I show three options, though in practice today, only one is really frequently used. This is called the rectified linear unit or ReLU for short. Note that both the tan h and sigmoid functions are nonlinear and relatively the same shape. These both approximate a binary switch which is either on or off.

Those two functions were used almost exclusively for decades. However, as neural networks started to be built with more hidden layers, researchers found that these activation functions caused problems in training. This is due to the fact that the curves become flat in their tails. The gradient descent algorithm uses gradients to update parameters. At the flat parts, the gradients become zero and this effectively stops the learning process. So now we need to wrap this up. We've already given strong advice on how to choose an activation function. For choosing the number and size of hidden layers, we need to take an empirical approach. The complexity of the network increases with the number and size of the hidden layers. There is no single rule for determining either hyperparameter. The best approach is to start simpler and iteratively test increasing levels of complexity. Let's say you start with one hidden layer with 20 nodes. We can then test and compare that against two additional networks, with one having a second hidden layer and the other having one layer with additional nodes. If we see an improvement, we can make the more complex network our default and run this test again. At this point, this process becomes standard hyperparameter optimization and model selection and the principles for those two methods apply here.

Back to Table of Contents

Read: Explore Activation Functions

An activation function is a part of neural networks that gives them the ability to switch linear components on or off and, thus, to express nonlinear functions. Without these activation functions, a neural network is only a linear model.

☆ Key Points

An activation function is the part in artificial neural networks that gives them the ability to express nonlinear functions.

ReLU, sigmoidal unit, and hyperbolic tangent are examples of activation functions.

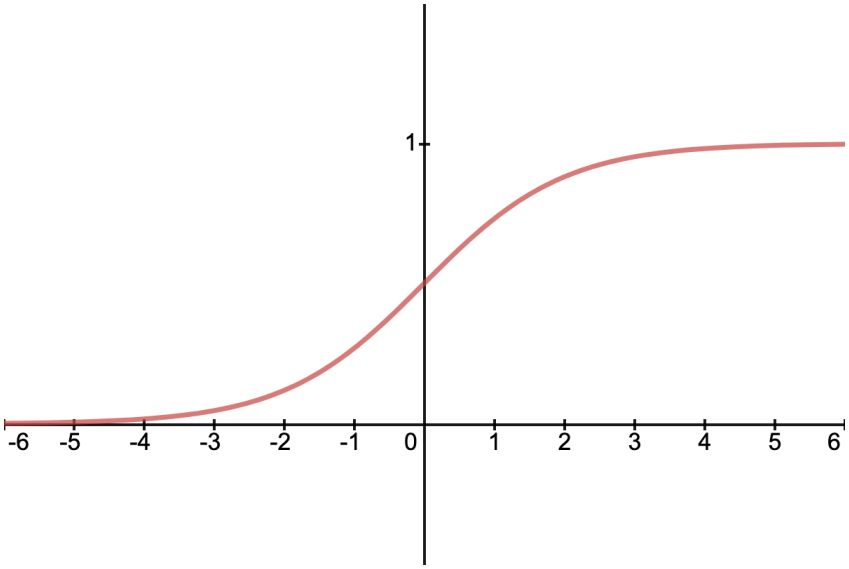
Consider a neural network with one hidden layer that is using the rectified linear unit (ReLU) activation function: $\boldsymbol{f(x) = W_2 \sigma(W_1 x)}$.

If we remove the ReLU function $\boldsymbol{\sigma}$, we essentially obtain $\boldsymbol{f(x) = W_2(W_1 x)) = Wx}$, where $\boldsymbol{W = W_2 W_1}$, and, hence, it is a linear function of $\boldsymbol{x}$.

Historically, people use the sigmoid, hyperbolic tangent, and ReLU functions as their activation functions.

Sigmoid function

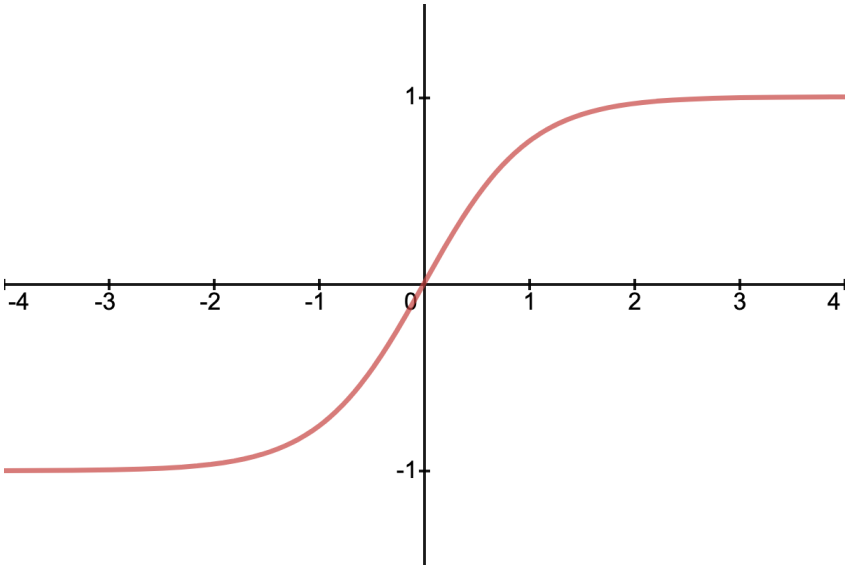
Sigmoidal unit: $\boldsymbol{\sigma(z) = \frac{1}{1+e^{-z}}}$



As shown in the graph and from its equation, the sigmoid function outputs values between 0 and 1. It is therefore popular when one wants to produce probabilities (e.g., in the very last layer). It also has a nice interpretation of a linear function being switched off (if the transition is close to 0) or switched on (if it is close to 1). Yet the two ends of a sigmoid tangent are too flat to provide a good gradient for stochastic gradient descent to learn. After a while, the values of the internal layers become too large and the weights **saturate** because the transition values are stuck in the flat regions. Note that the problem here is the symmetry of the function. Large weights will cause activations of large magnitude — either very negative or very positive, both of which lead to tiny gradients.

Hyperbolic tangent/tanh function

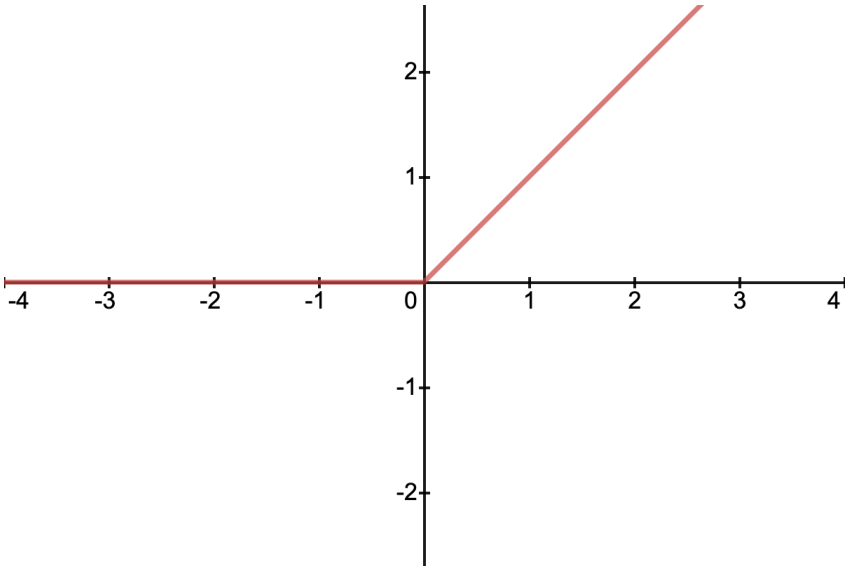
tanh: $\sigma(z) = \tanh(z)$



The tanh function has a similar shape to the sigmoid function, but its range is between -1 and 1. This means that negative values become very negative while values near 0 stay near 0. It is useful for binary classification. However, this transition function runs into the same problem as sigmoid functions due to their similar shapes.

ReLU function

ReLU: $\sigma(z) = \max(z, 0)$



The ReLU function is currently the most popular transition function as it doesn't completely fall into the same pitfall as the other two functions: having ends that are

too flat and thus not easy to optimize. Although the left part of the ReLU is flat, it is not symmetric, so when the sign of the activation changes, the gradient is very strong.

Back to Table of Contents

Watch: Training a Neural Network

The training process of a neural network is similar to the training of linear models such as logistic regression. It uses a gradient descent optimization algorithm to update model parameters to minimize a loss function and regularization can be applied to prevent overfitting.

In this video, Mr. D'Alessandro explains how a neural network is trained through the process of forward propagation to obtain an output and compute loss then back propagation using a form of gradient descent to update model parameters to minimize the loss function. Mr. D'Alessandro also discusses the loss functions used in neural networks for both regression and classification problems.

Video Transcript

Just like with logistic regression, our training goal in neural networks is to find the optimal values for our model parameters. One challenge with neural networks though, is they contain a lot of model parameters. Each hidden node in a neural network is connected to every node in prior and subsequent layers. Each of those connections is associated with the model parameter to learn. For instance, a network with 20 input features, two hidden layers and 10 nodes each, and a single output has over 300 parameters to learn, whereas a logistic regression with the same 20 features has only 20 model parameters to learn. This difference is one of the reasons why neural nets are so powerful, but it also makes training difficult. Deep neural nets can have hundreds of millions to billions of parameters, which certainly renders model training to be one of the most difficult tasks in building an effective model. Making a prediction in a neural net is called forward propagation, because we move forward from the neural network's input to the output layer. When we compute gradients to update parameters during training, we move backwards from output layers to input layers. This process is called backward propagation, and we will discuss this process in this video. We'll also show how a variation of gradient descent called stochastic gradient descent is the default approach to network training. Before we dig into backpropagation, let's discuss the role of loss functions. Similar to logistic and linear regression, the loss function in a neural net tells us how much our current prediction

deviates from the truth or label. We can use the same loss functions for neural networks that we use in logistic and linear regression. For binary classification, we commonly use log loss. For regression we commonly use mean squared error. These aren't the only loss functions we can use though. The log loss for binary classification can be generalized to multiple classes using what we call the softmax loss. A all of these loss functions, that general backpropagation algorithm is the same. We start with the forward propagation step, where features are fed into the network and we arrive at a prediction. We then calculate the loss associated with the given prediction. Backpropagation helps us compute how much each weight contributes to the loss, which then guides us in how to adjust the weight to subsequently reduce the loss. Mathematically, we need to use two tools from calculus. The first is the partial derivative, which tells us how much a function changes due to a small change in a single variable. This is also called the gradient of the function with respect to that variable. The other tool is the chain rule of derivation. Because of the nonlinear activation function used in each node, neural networks are what we call composite, or nested functions. The chain rule helps us compute the derivative of a composite function, which takes the product of each nested function in the chain. Each hidden layer of the network adds a layer of nesting. So for neural networks the size of the chain here grows with the number of network layers. We apply backpropagation to each weight in the network, which amounts to computing the partial derivative of loss function for each weight. Once we compute the partial derivative or gradient for a given weight, we can use the gradient descent method to update the weight. Computing the gradients for each parameter is actually pretty complex and beyond the scope of this lecture. Luckily, the tedious computation details are managed by customized software. As a machine learning engineer, it is more important to understand how to work with the various gradient descent algorithms that are commonly used. We typically use a variant called stochastic gradient descent, or SGD for short. This variant is highly efficient because it computes gradients using either single examples or a small batch of them at a time. And even within SGD, there are multiple variants with each one better for different properties of the data. For this class, we'll focus just on one or two fundamental variations and learn how to optimize models using just those.

Back to Table of Contents

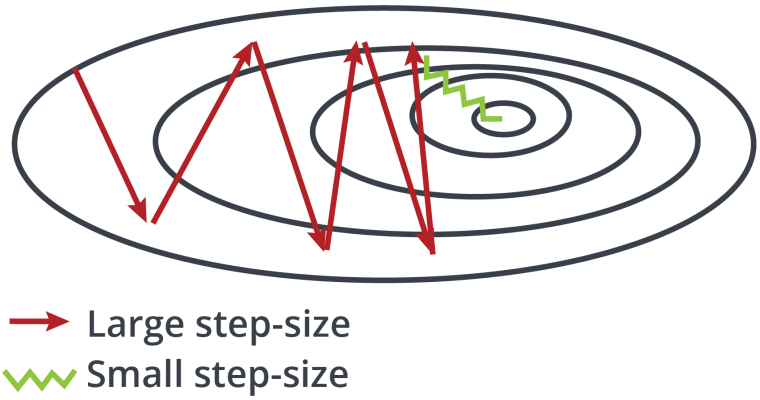
Read: Formalize Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a variant of gradient descent (GD) that is widely used to train different machine learning models, especially a neural network. In GD, a batch is the total number of training examples that are used to calculate the gradient in a single iteration. A batch may include the entire data set, potentially consisting of billions or even hundreds of billions of training examples. The data set may also contain a massive numbers of features, and such a large batch may cause one iteration to take a very long time to execute. SGD is a version of GD that mitigates this issue by selecting a subset of examples chosen at random from the data set. SGD uses a batch size (a mini-batch) that is typically between 10 and 1,000 examples.

☆ **Key Points**

Stochastic gradient descent is commonly used to train neural networks.

The modification for training deep neural networks is simple because the update is based on small batches instead of all training examples.



Let's compare SGD to GD.

Suppose we want to train our neural network using GD. To do so:

1. Evaluate the output of all training examples.
2. Calculate the loss using the loss function we are using: $\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \ell(\mathbf{x_i}, y_i)$
3. Calculate the gradient of the loss with respect to the weights:
   $\nabla \mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(\mathbf{x_i}, y_i)$
4. Take a gradient step to update the weights.
5. Repeat Steps 1 through 4 until the training loss does not decrease.

With deep neural networks, evaluating and calculating the loss and gradients of every single example is very expensive. Thus, researchers came up with SGD:

1. Sample a mini-batch of m training examples without replacement.
2. Evaluate the output of these m training examples.
3. Calculate the loss using the loss function we are using: $\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} \ell(\mathbf{x_i}, y_i)$
4. Calculate the gradient of the loss with respect to the weights:
   $\nabla \mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} \nabla \ell(\mathbf{x_i}, y_i)$
5. Take a gradient step to update the weights.
6. Repeat Steps 1 through 4 until the training loss does not decrease.

The modification is rather simple. Instead of each update being based on all training examples, they are based on a small batch of size m = 64, 128, 256, etc. We can do this because the loss and gradient can be expressed as a sum of terms. Sampling a small set of examples from the training set then evaluating the loss and gradient on the small batch can be viewed as estimating the true training loss and gradient. This is from where the term "stochastic" derives.

Back to Table of Contents

Activity: Design and Explore Neural Networks

Explore the interactive below, which gives you the ability to change the type of data, the features used, the size and depth of your network, and parameters such as problem type. When you click the **Play** button, you will see the number of epochs (training cycles) increase and you will see an output of both the training and test loss above the output.

Using the interactive, answer these six questions:

1. Experiment with different data sets and different network architectures; how does changing the activation functions change the output?
2. Can you construct a network that can properly learn each data set (training loss below 0.1)?
3. Can you learn the spiral classification data set with a network with a single hidden layer?
4. Switch between regression and classification problems. What challenges do each seem to pose?
5. What is the effect of regularization on the output?

The full contents of this page cannot be rendered in the course transcript. Log in to the course to view it.

Created by Daniel Smilkov and Shan Carter.

Back to Table of Contents

Tool: Neural Network Cheat Sheet

Neural networks are versatile, and there is a lot to consider when implementing them. Review the algorithm and its components with this cheat sheet.

Download the Tool

Use the **Neural Network Cheat Sheet** to review how the neural network algorithm functions.

Back to Table of Contents

Read: Python Packages for Neural Networks

☆ **Key Points**

Several different Python-based packages have been developed to support neural network-based models.

We have been focused on using the scikit-learn package, which is at present the most comprehensive Python package for executing a broad array of different machine learning algorithms and for developing support code to validate models. The availability of a rich set of documentation as part of scikit-learn also makes the environment very productive for learning about machine learning algorithms.

However, there are several different Python-based packages that are more commonly used to support neural network models, especially for deep learning tasks. Python is often the language of choice in part because it is a generally popular programming language, and in part, because it provides support both for constructing complex network architectures using object-oriented programming and for creating efficient numerical computations underneath that can tackle very large data sets and training procedures. The most widely used of these Python-based deep learning packages are:

- **TensorFlow**
- **Caffe**
- **PyTorch**
- **Keras**

Each has its own particular syntax for creating, training, and applying deep learning models, as well as documentation and tutorials describing their use. Your training in using the Python programming language and ecosystem will enable you to build your own machine learning pipelines using these tools.

Back to Table of Contents

Code: Implementing a Neural Network in Keras

Many different software packages have been created to implement neural networks and deep learning. Two of these are TensorFlow and Keras, a high-level neural network API that runs on top of TensorFlow. While both packages provide high-level APIs used for easily building and training neural network models, Keras is more user friendly and has simpler APIs.

These packages are commonly used within academic research and the ML industry. Both support Python and are easily accessible through packages that you can import in your Python program. In this demo, you will discover how to construct, train, and evaluate a feedforward neural network for classification using the Keras Python package.

This activity will not be graded.

The full contents of this page cannot be rendered in the course transcript. Log in to the course to view it.

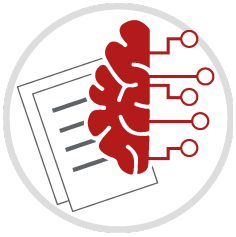
Back to Table of Contents

Module Wrap-up: Introduction to Neural Networks

In this module, Mr. D'Alessandro discussed an advanced machine learning algorithm called neural networks. You were able to examine the underlying construction of neural networks using a series of linear and nonlinear transformations. You also had the chance to explore the different Python packages for working with neural networks and observe an implementation of a neural network in Keras.

Back to Table of Contents

Module Introduction: Introduction to Deep Learning



Deep learning is a subset of machine learning that uses neural networks with many hidden layers to solve complex problems. These include image classification, text analysis, and speech recognition. Deep learning has become hugely popular in the last two decades due to its success in solving these problems, the rapid increase in computing power, and the widespread availability of big data. There are specific neural network architectures that are well suited for such applications. In this module, you will explore one deep neural network used in the field of natural language processing and discover how it takes advantage of the specifics of sequence data (i.e., sequence of words).

Back to Table of Contents

Deep Learning Neural Network Architectures

While we have been focusing thus far on feedforward neural networks, there are other neural network architectures that are better suited for very complex problems requiring many hidden layers. In this video, Mr. D'Alessandro explores an architecture used in the field of natural language processing: recurrent neural networks.

Video Transcript

Feed-forward neural networks are very powerful algorithms that can be used to approximate any mathematical function. Despite this power, they are limited in certain types of problems such as image recognition and natural language processing. In image recognition, feed-forward networks don't officially account for repetitive patterns that frequently occur in images. In NLP, feed-forward networks don't account for meaning that is embedded in longer word sequences, nor do they account for long-range dependencies. Special neural network architectures have been invented to improve performance for image recognition in NLP tasks. You've probably heard of deep learning by now, which encompasses these new and advanced architectures. Deep learning in general refers to neural networks with many hidden layers. Having more layers doesn't automatically solve the challenges we face in image recognition in NLP, though. In addition to having many layers, the core deep learning methods have additional features and architectures that are tuned to the challenges of the problem type. For instance, in image recognition, we have what we call convolutional neural networks. These neural nets apply what we call convolutions to the data, which are specialized network operations designed to capture repetitive aspects of image data. For now though, we'll focus more on deep learning methods designed for NLP applications. Feed-forward networks aren't great with sequential data, which is pretty standard in natural language text. Feed-forward neural networks have no memory of the input they receive and are bad at predicting what's coming next. Because a feed-forward network only considers the current input, It has no notion of order in time. It also isn't able to remember information from text observed in prior steps. To accommodate these issues, special deep neural networks, called recurrent neural networks have been developed. These are special because the network is designed so that the output of a given node flows back into the node, thus

the use of the name recurrent. In a recurrent neural network, or RNN for short, the information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it has received previously. RNNs are usually represented differently than standard neural networks. This diagram shows both the compressed and uncompressed versions of the RNN. The compressed shows the basic recurrence that is going on. But when we look at the uncompressed, we can get a better sense of how information is flowing through the network sequentially. There are two core pieces of input to focus on. We have the present input, which is represented by the blue circles here. This is just your data. For text, this is often individual words or tokens. Then we have the memory, which is represented by the arrows flowing between the green nodes. We can think of the memory as an embedding of all the previous information passed into the network. This is what makes these methods so powerful for text — the network here has the ability to store in its memory representation a lot of the key language context needed to draw specific conclusions about the modeling task. At each step, we also have the output, represented by the purple nodes here. Often the output here is predicted next word. A lot of our use cases we discussed abstract to different types of language generation. The RNN can take in a sequence but also generate a predicted sequence. There are many variations of recurrent neural networks. The variations include choices for the architecture of the cells, the direction of the flow of information, and the layering of the cells. There are also special architectures called encoder-decoder networks which stack two separate RNNs next to each other. These are often used for translation, where the encoder creates an embedding representation of a piece of text, and the decoder converts that encoding to plain text in another language. As I mentioned earlier, this is a specialized field in and of itself. In this course, we'll focus just on simpler tasks like sentiment classification to build the foundations to be able to tackle more complex NLP problems.

Back to Table of Contents

Ask the Expert: Francesca Lazzeri on How Deep Learning Is Used in Real Life

As you have seen, deep learning refers to neural networks with many hidden layers. It has become hugely popular in the last two decades due to its success in image classification, text analysis, speech recognition, the rapid increase in computing power, and the widespread availability of big data. Many artificial intelligence systems employ deep learning algorithms. In this video, Francesca Lazzeri gives some real-world examples of how these algorithms are used in real-world ML applications.

Note: The job title listed below was held by our expert at the time of this interview.

Francesca Lazzeri
Curriculum Committee Industry Advisor, Microsoft

Dr. Francesca Lazzeri is an experienced scientist and machine learning practitioner with both academic and industry experience as an Adjunct Professor of AI and Machine Learning at Columbia University and Principal Cloud Advocate Manager at Microsoft. She also authored the book "Machine Learning for Time Series Forecasting with Python" (Wiley) and many other publications, including technology journals and conferences. At Microsoft she leads a large international team (across the U.S., Canada, U.K., and Russia) of engineers and cloud AI developer advocates, managing a large portfolio of customers in the research and academic sectors and building intelligent automated solutions on the cloud. Before joining Microsoft, Dr. Lazzeri was a research fellow at Harvard University in the Technology and Operations Management Unit. You can find her on Twitter at @frlazzeri.

Question

What are some common algorithms used in real-world ML applications?

Video Transcipt

A very interesting algorithm and approach that a lot of these applications use is, for example, times used for casting and use for casting. It's not really like an algorithm, it's more like a family of algorithms. But as you can imagine, like for example, if you are

buying something on a website and then they're going to recommend you to buy it different but similar item based on your historical data and your historical purchases, that is the run on top of the what we call the time series forecasting algorithms, because basically they are trying to predict what you're going to buy next. Now, in this family oftentimes used forecasting algorithms, we have a few algorithms that have been used a lot in the past few years. Those are algorithms that are more from the deep learning areas. We refer to them as a unless the empathy is a long, short term memory, but also to your user. You have to think about them as a recurring natural network facility. Again, it's apology of deep learning algorithms and they have been like very, very helpful when you need them to predict something, the future specifically in terms of the sales or demand forecasting. Again, these type of algorithms are trying to predict what that person is going to buy next and as a consequence they are somehow the first step in order to predict and optimize the inventory in general. It's very important to understand that if you are happy because you buy something and you are going to receive it, that item in a couple of days because they immediately ship it to you, not even a couple of days, sometimes even just a matter of a few hours. It's because there is a great atmosphere's forecasting algorithm and as a consequence in a voting station algorithm that is trying to predict in real time what you're going to need next and to buy next.

Back to Table of Contents

Read: Review Recurrent Neural Networks

Let's take a moment to review and take a deeper dive into recurrent neural networks (RNNs). You will not practice building an RNN in this course so you may reference this page for the basic foundation of RNNs in the future. You can also reference the implementation sample from Keras to build an RNN on your own.

Feedforward neural networks are not ideal for text data because text data is inherently sequential and context dependent. Often, a word itself is not meaningful until it's put into a sequence. For instance, consider the following sentences:

- He went to the bank to withdraw some money.
- He likes to go on a stroll at the river bank after dinner.

If you only look at the single word "bank," it is hard to tell whether the bank you are referring to is a financial bank or a river bank. On the other hand, the meaning is clear if you look at the whole sentence (or the whole sequence of words). You need more complex neural networks to encode this sequential nature of text.

Special types of deep learning neural networks have been developed to deal with text data. They are used to handle different text applications, such as machine translation,

☆ Key Points

There are two components to neural networks that process text: encoders and decoders.

Encoders take in a sequence of words and output a vector (or a code) that can be viewed as a summary of the input sequence.

Decoders take in such a vector (computed from encoder) and turn it into a scalar or sequence of outputs.

A recurrent neural network is a neural network that is applied at each element of the sequence but keeps track of a hidden state vector and uses it to encode and decode sequential information.

There are different built-in Keras layers than can be used to implement an RNN.

text summarization, question answering, and so on. These models usually consist of two components:

1. **Encoder:** A neural network that takes in a sequence of words (represented using word embeddings) and outputs a vector or a code that can be viewed as a summary of the input sequence.
2. **Decoder:** A neural network that takes in the vector output of an encoder and turns it into a scalar or sequence of outputs. (These can be words represented by word embeddings or other things, depending on the application.)

Typically, the encoder and decoder are learned jointly. For example, in neural machine translation, one trains on input/output translation pairs. These could be English/German sentence pairs: One trains the encoder and decoder to map the English sentence to its German translation.

Recurrent neural networks

The goal of a sentiment analysis task is to tell whether the sentiment behind a text is positive or negative. In such cases, the ordering of the input text is important. For instance, the sentence “John enjoys eating the shark” and the sentence “The shark enjoys eating John" are very different (the second one probably has a more negative sentiment than the first). You need more complex neural networks to encode this sequential nature of text.

This is where a recurrent neural network (RNN) comes into play. An RNN works by iteratively applying the same base operation to each element in the input sequence. This is called the base operation an RNN cell. RNN cells distinguish themselves from the neurons in a feedforward neural network in that they have the concept of “memory” that helps them capture information about what has been calculated so far from previous inputs (e.g., previous words in a sequence of words). This information is called "the hidden state." In essence, an RNN keeps track of a hidden state variable and uses this hidden state variable to encode sequential information. Therefore, rather than making a decision only on a current input, an RNN is also able to consider the output it has learned from previous inputs in order to guide its decisions.

▼ **Computing the output of a recurrent neural network**

Concretely, the simplest RNN consists of two learned parameters, $\mathbf{U}$ and $\mathbf{W}$. Suppose you have a sequence of inputs $s_1, s_2, s_3, \ldots, s_n$ (this can be a sentence where each s_i is the embedding of a word in the sentence) and an initial hidden state $\mathbf{h}_0$.

A recurrent neural network essentially does the following for each input s_i in the sequence:

1. Process the previous hidden state $\mathbf{h}_{i-1}$. This is done with matrix multiplication to yield $\mathbf{W}\mathbf{h}_{i-1}$. This represents the "history" component of the sentence before the word s_i.
2. Process the current input s_i by using matrix multiplication to yield $\mathbf{U}s_i$. This represents the information from the current input s_i.
3. Add up the output of the previous two steps to yield $\mathbf{W}\mathbf{h}_{i-1} + \mathbf{U}s_i$.
4. Apply a nonlinear activation function to the output of (3) to yield the hidden state $\mathbf{h}_i = \sigma(\mathbf{W}\mathbf{h}_{i-1} + \mathbf{U}s_i)$.
5. Depending on the application, you can further process the hidden state variable o_i (optional).

The final hidden state can be viewed as the vector representation of the whole sequence.

Intuitively, think of the hidden state as a temporary representation of all the previous words; this hidden state will determine how to process the current input. Having this hidden state allows the neural network to "remember" the sequence and encode the sequential information in the sequence.

▼ Training recurrent neural networks

RNNs can be trained using stochastic gradient descent (SGD). The output of the RNN can be used in a variety of ways, which impacts the way it should be trained.

In machine translation, for instance, you want to have a model that can take in an English sentence and output a German sentence. You can use one RNN as our encoder to encode the English sentence into a vector representation $\mathbf{h}$

then have another RNN as our decoder that decodes $\mathbf{h}$ into a German sentence. The encoder is straightforward: You pass in a sequence of embedded English words and follow the above steps.

For the decoder, however, you pass in a vector (the encoded English sentence) instead of a sequence of words. How can you use an RNN if the input is a vector and not a sequence of word embeddings? The key is to use $\mathbf{h}$ as the initial hidden state and input a starting word for the sentence. This can be as simple as a non-word, or a "token," that indicates the start of a sentence. You then pass the hidden state variable through some function that picks the best German word. For example, given the hidden state $\mathbf{h}$, you predict word $i$with word vector $\mathbf{w}_i$ with probability $P(i|\mathbf{h}) = \frac{\exp(\mathbf{w}_i^\top \mathbf{h})}{\sum_k \exp(\mathbf{w}_k^\top \mathbf{h})}$. Sometimes people pick the most likely word; sometimes they sample one randomly from this word distribution.

The word vector of the generated German word is the next input passed into the RNN. The whole process can be repeated until some stopping criterion. You can then compute the loss between the ground truth translation and the prediction made by the decoder. With this loss, you can do back propagation and SGD to update the parameters of the encoder and decoder.

▼ Implementing an RNN in Keras

Keras allows you to implement RNNs in a few different ways. You have seen how to implement a feedforward neural network using Keras Dense layers. You can implement an RNN following the same pattern, but using a different type of Keras layer. Keras provides a few different built-in layers for RNNs. (For more information, take a look at the TensorFlow guide page found on the Keras website, **Recurrent Neural Networks (RNN) with Keras**.)

One of the Keras RNN layers most commonly used is the LSTM layer. This corresponds to a special type of RNN known as a long short-term memory (LSTM) network, which is often used in the real world. RNNs store previous data in their “short-term memory.” Once the memory runs out, RNNs delete some of the retained information about previous inputs. An LSTM has a more complex structure with more memory, therefore allowing information to be retained

longer, and long-term dependencies to be learned. A type of LSTM often used in NLP is the Bidirectonal LSTM (BiLSTM). A BiLSTM is capable of learning the dependencies of words from both sides of an input sequence.

The example on implementing a Bidirectional LSTM below can be found with more detail on the Keras website, **Bidirectional LSTM on IMDB**. The code example shows how to implement a two-layer bidirectional LSTM.

The code below implements the BiLSTM using two Keras layer classes:

- The Keras Bidirectional Layer is a Bidirectional wrapper for different RNNs. (Find more information on **Bidirectional Layer** on the Keras website).
- The LSTM Layer (Find more information on the **LSTM Layer** on the Keras website).

Notice that the code below also uses the Keras Embedding Layer as the first layer in the network. The Embedding Layer learns word embeddings from its input text. Before the text can be used as input into the Embedding Layer, it has to be preprocessed so that each word is represented by a unique integer. You can use a variety of techniques to preprocess the data, including using one of the Keras Tokenizers. (You can find more information on the **Embedding Layer** and Keras **Tokenizers** on the Keras website).

```
# Input for variable-length sequences of integers
inputs = keras.Input(shape=(None,), dtype="int32")
# Embed each integer in a 128-dimensional vector
x = layers.Embedding(max_features, 128)(inputs)
# Add 2 bidirectional LSTMs
x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
x = layers.Bidirectional(layers.LSTM(64))(x)
# Add a classifier
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.summary()
```

You can try implementing a BiLSTM using one of TensorFlow's preprocessed datasets such as the preprocessed IMBD dataset contained in the code example above. If you want to use the **IMDB dataset**, you can find it can be found on TensorFlow's website.

Tool: Recurrent Neural Network Cheat Sheet

This cheat sheet breaks down the functionality of recurrent neural network models. Use this cheat sheet to review how the algorithm works.

Back to Table of Contents

Download the Tool

Use this **Recurrent Neural Network Cheat Sheet** to review how sequence models function.

Quiz: Check Your Knowledge: Neural Networks

You may submit this quiz up to three times.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Ask the Expert: Brandeis Marshall on AI Social Implications

You have discovered what AI is capable of doing, such as recognizing the sentiment behind a text document, but have you thought about how AI can affect its users? Is it always good? Is it always fair? In this video, Dr. Marshall lists some social implications as well as examples of how AI can be good and bad.

Note: The job title listed below was held by our expert at the time of this interview.

Brandeis Marshall
Founder and CEO, DataedX Group

Brandeis Marshall is founder and CEO of DataedX Group, LLC. DataedX provides learning and development training to help educators, scholars and practitioners humanize their data practices. Dr. Marshall is the author of "Data Conscience: Algorithmic Siege on our Humanity" (Wiley, 2022). She speaks, writes, and consults on how to move slower and build better human-centered tech by highlighting the impact of data practices on technology and society.

Dr. Marshall has been a Stanford PACS Practitioner Fellow and Partner Research Fellow at Siegel Family Endowment. She has served as a tenure-track faculty member at Purdue University and Spelman College. Dr. Marshall's research work in data education and data science has been supported by the National Science Foundation and philanthropy organizations. She holds a Ph.D. and Master of Science in Computer Science from Rensselaer Polytechnic Institute as well as a Bachelor of Science in Computer Science from the University of Rochester.

Question

What are some social implications of artificial intelligence? Can you list some of the good and the bad?

Video Transcript

I have three technologies that I believe are good when it comes to AI. The first one is online banking. I mean, who doesn't want to get their money faster? I know I do. I'm sure you do too. Online baking actually is a very good application of AI. Second, shared

drives. Now, there was a time back in the day when all you had was pen and paper. I know it sounds crazy, but that's all we had. Let me tell you the shared drives now that you can write a document, share it with classmates, share it with colleagues, share it with friends, and then be able to add to that document and then see the edits has been game-changing. This type of AI technology is actually very useful. It helps with productivity and optimizing work. Third, and my personal favorite, happens to be GPS. Back in the day, what would happen is you wanted to go someplace and you would have to go on the Internet, go to a website, type in where you wanted to go, see the directions, then print them out because you did not have a mobile phone, and then you had to follow the directions from the piece of paper. I know, again, it sounds crazy, but this is what happened, not even 30 years ago. GPS allows us to do a couple of things. Number one, it allows us to be able to get from point A to point B without knowing, not having to print out directions. We save some paper. Secondly, it allows us to know what's happening in traffic updates. If there is an accident or there's some impedance to where you're headed, you can actually be rerouted pretty quickly. Number three, it's almost impossible to get lost. Because of the GPS and the re-routing systems of AI technology, we're able to get back on track to where we're actually trying to go. Those are three really good AI tools. Now as far as the bad ones, there are so many that I can't even enumerate them. But let me give you a bad one that I think is pretty recent. That happens to be, in my opinion, AI writing. There is AI writing tools. You've probably heard of a couple of them. Chat GPT, there's also Bard, there Jasper.ai, there's Copy.ai as well. There's many others. These tools are used in order to help generate text, sentences, whole paragraphs, and some people use them for their blogs. This has been a wonderful tool to get people started, but in the end, the content is not vetted so it might be incorrect information and number two, it does not have your voice. AI is essentially being used in order to be a shortcut. That shortcut dilutes a little bit of our humanity. Just be careful with that type of technology. Second type of example happens to be what's very well-known within the criminal justice system and facial recognition. The fact that people with melanated skin tend to be misclassified using facial recognition technology. That means that people with melanated skin tend to be stopped, pulled over, questioned and things of those nature much more than people with non-melanated skin. But that is definitely a bad component of AI. Of course, this has moved into spaces, into criminal justice spaces where criminal justice sentences have been dictated by algorithms called recidivism

algorithms. And those algorithms have unfortunately been used in order to render higher and longer sentences for those people with melanated skin. Black and brown individuals versus those with non-melanated skin get lower sentences even though the risk for re-offending is as high or higher than those that are black or brown. Again, these are just examples of just technology gone awry and gone away. Be careful out there when it comes to when and what you use AI technology tools for.

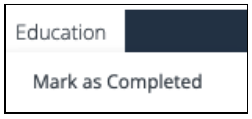
Back to Table of Contents

Assignment: Unit 7 Assignment - Using a Pipeline for Text Transformation and Classification

In this assignment, you will first transform text data into numerical feature vectors then fit a logistic regression classifier to the feature vectors. You will discover how to simplify the process by chaining these two steps together using scikit-learn pipelines. You will also learn how to find an optimal hyperparameter configuration by running a grid search on the pipeline.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window 

3. Note: This assignment will be manually graded by your facilitator. You cannot resubmit.

This assignment will be graded by your facilitator.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Assignment: Unit 7 Assignment - Written Submission

In this part of the assignment, you will answer six questions about using text as data and word embedding.

The questions will prepare you for future interviews as they relate to concepts discussed throughout the week. You've practiced these concepts in the coding activities, exercises, and coding portion of the assignment.

Completion of this assignment is a course requirement.

Instructions:

1. Download the Unit 7 Assignment document.
2. Answer the questions.
3. Save your work as one of these file types: .doc or .docx. No other file types will be accepted for submission.
4. Submit your completed Unit 7 Assignment document for review and credit.
5. Click the **Start Assignment** button on this page, attach your completed Unit 7 Assignment document, and then click **Submit Assignment** to send it to your facilitator for evaluation and credit.

Back to Table of Contents

Module Wrap-up: Introduction to Deep Learning

In this module, you explored deep learning and watched Mr. D'Alessandro introduce a deep learning architecture called a recurrent neural network. You also discovered how an RNN is constructed and why it is better suited for natural language processing problems than a feedforward neural network.

Back to Table of Contents

Lab 7 Overview

In this lab, you will perform sentiment analysis using a neural network. You will transform text data into numerical feature vectors using TF-IDF then construct and train a neural network model using these vectors as inputs. You will then compare your model's performance on the test data to its performance on the training data to determine whether your model is overfitting the training data.

You will also find ways to improve the model's ability to generalize to new data. Finally, you will experiment with different neural network configurations to observe the effect this has on the model's overall performance. You will work in a Jupyter Notebook throughout the lab.

This three-hour lab session will include:

- **10 minutes** - Icebreaker
- **30 minutes** - Week 7 Overview and Q&A
- **20 minutes** - Breakout Groups: Big-Picture Questions
- **10 minutes -** Class Discussion
- **10 minutes - Break**
- **30 minutes** - Breakout Groups: Lab Assignment Working Session 1
- **15 minutes** - Working Session 1 Debrief
- **30 minutes** - Breakout Groups: Lab Assignment Working Session 2
- **15 minutes** - Working Session 2 Debrief
- **10 minutes -** Concluding Remarks and Survey

By the end of Lab 7, you will:

- Transform features using a TF-IDF vectorizer.
- Construct and train a feedforward neural network model.
- Evaluate the model's performance on the test data and compare it to the performance on the training data.
- Experiment with ways to prevent overfitting and improve the model's ability to generalize well to new, previously unseen data.

Back to Table of Contents

Assignment: Lab 7 Assignment

In this lab, you will continue working with the "Book Reviews" data set.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window 

3. Note: This assignment will be manually graded by your facilitator. You cannot resubmit.

This lab assignment will be graded by your facilitator.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.