Skymind Wiki: Part 3

skymind

The Math Behind Neural Networks

Chris V. Nicholson

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. To an outsider, a neural network may appear to be a magical black box capable of human-level cognition.

Under the surface, however, neural networks contain a structured sequence of math and code. To understand how neural networks make predictions, you must understand the methodology, structure, and mathematics behind each architecture – the foundations of which are just linear algebra and calculus.

INTRODUCTION 1

BACKPROPAGATION IN NEURAL NETWORKS 2

- Backpropagation 2
- Gradient Descent 2

EIGENVECTORS, EIGENVALUES, PCA, COVARIANCE AND ENTROPY 3

- Linear Transformations 3
- Principal Component Analysis (PCA) 6
- Covariance Matrix 7
- Change of Basis 9
- Entropy & Information Gain 9

GRAPH ANALYTICS AND DEEP LEARNING 10

- Concrete Examples of Graph Data Structures 10
- Difficulties of Graph Data: Size and Structure 11
- Representing and Traversing Graphs for Machine Learning 12
- Further Resources on Graph Data Structures and Deep Learning 12

FOOTNOTES 17

A BEGINNERS GUIDE TO BACKPROPAGATION IN NEURAL NETWORKS

Backpropagation is the central mechanism by which neural networks learn. It is the messenger telling the network whether or not the net made a mistake when it made a prediction.

To propagate is to transmit something (light, sound, motion or information) in a particular direction or through a particular medium. When we discuss backpropagation in deep learning, we are talking about the transmission of information, and that information relates to the error produced by the neural network when they make a guess about data.

Untrained neural network models are like new-born babies: They are created ignorant of the world, and it is only through exposure to the world, experiencing it, that their ignorance is slowly revised. Algorithms experience the world through data. So by training a neural network on a relevant dataset, we seek to decrease its ignorance. The way we measure progress is by monitoring the error produced by the network.

The knowledge of a neural network with regard to the world is captured by its weights, the parameters that alter input data as its signal flows through the neural network towards the net's final layer that will make a decision about that input. Those decisions are often wrong, because the parameters transforming the signal into a decision are poorly calibrated; they haven't learned enough yet. A data instance flowing through a network's parameters toward the prediction at the end is forward propagation. Once that prediction is made, its distance from the ground truth (error) can be measured.

So the parameters of the neural network have a relationship with the error the net produces, and when the parameters change, the error does, too. We change the parameters using an optimization algorithm called gradient descent, which is useful for finding the minimum of a function. We are seeking to minimize the error, which is also known as the **loss function** or the **objective function**.

Backpropogation

A neural network propagates the signal of the input data forward through its parameters towards the moment of decision, and then **backpropagates** information about the error through the network so that it can alter the parameters one step at a time.

You could compare a neural network to a large piece of artillery that is attempting to strike a distant object with a shell. When the neural network makes a guess about an instance of data, it fires, a cloud of dust rises on the horizon, and the gunner tries to make out where the shell struck, and how far it was from the target. That distance from the target is the measure of error. The measure of error is then applied to the angle of and direction of the gun (parameters), before it takes another shot.

Backpropagation takes the error associated with a wrong guess by a neural network, and uses that error to adjust the neural network's parameters in the direction of less error. How does it know the direction of less error?

Gradient Descent

A gradient is a slope whose angle we can measure. Like all slopes, it can be expressed as a relationship between two variables: "y over x", or **rise over run**. In this case, the y is the error produced by the neural network, and x is the parameter of the neural network. The parameter has a relationship to the error, and by changing the parameter, we can increase or decrease the error. So the gradient tells us the change we can expect in y with regard to x.

To obtain this information, we must use differential calculus, which enables us to measure **instantaneous rates of change**, which in this case is the tangent of a changing slope expressed the relationship of the parameter to the neural network's error.

Obviously, a neural network has many parameters, so what we're really measuring are the **partial derivatives** of each parameter's contribution to the total change in error.

What's more, neural networks have parameters that process the input data sequentially, one after another. Therefore, backpropagation establishes the relationship between the neural network's error and the parameters of the net's last layer; then it establishes the relationship between the parameters of the neural net's last layer those the parameters of the second-to-last layer, and so forth, in an application of the **chain rule of calculus**.

It is of interest to note that backpropagation in artificial neural networks has echoes in the functioning of biological neurons, which respond to rewards such as dopamine to reinforce how they fire; i.e. how they interpret the world. Dopaminergenic behavior tends to strengthen the ties between the neurons involved, and helps those ties last longer.

Further Reading About Backpropogation

- Backpropagation
- 3BLUE1BROWN VIDEO: What is backpropagation really doing?
- Dendritic cortical microcircuits approximate the backpropagation algorithm

A BEGINNER'S GUIDE TO EIGENVECTORS, EIGENVALUES, PCA, COVARIANCE AND ENTROPY

This section introduces eigenvectors and their relationship to matrices in plain language and without a great deal of math. It builds on those ideas to explain covariance, principal component analysis, and information entropy.

The **eigen** in eigenvector comes from German, and it means something like "very own." For example, in German, "mein eigenes Auto" means "my very own car." So eigen denotes a special relationship between two things. Something particular, characteristic and definitive. This car, or this vector, is mine and not someone else's.

Matrices, in linear algebra, are simply rectangular arrays of numbers, a collection of scalar values between brackets, like a spreadsheet. All square matrices (e.g. 2 x 2 or 3 x 3) have eigenvectors, and they have a very special relationship with them, a bit like Germans have with their cars.

Linear Transformations

We'll define that relationship after a brief detour into what matrices do, and how they relate to other numbers.

Matrices are useful because you can do things with them like add and multiply. If you multiply a vector **v** by a matrix **A**, you get another vector **b**, and you could say that the matrix performed a linear transformation on the input vector.

Av = b

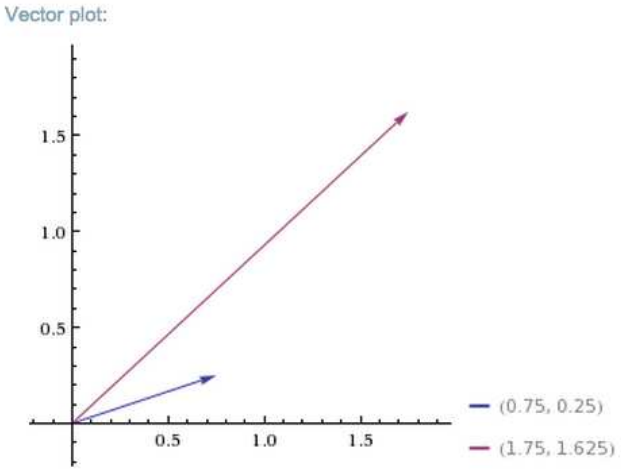
It maps one vector **v** to another, **b**.

We'll illustrate with a concrete example. (You can see how this type of matrix multiply, called a dot product, is performed here.)

$$\begin{matrix} A & & v & & b \end{matrix}$$

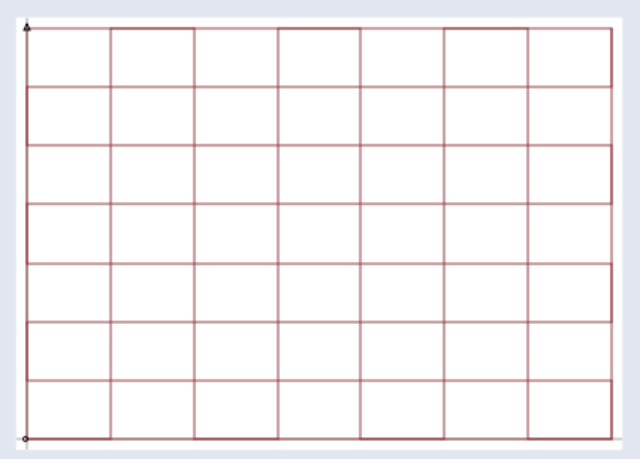
$$\begin{bmatrix} 2 & 1 \\ 1.5 & 2 \end{bmatrix} * \begin{matrix} 0.75 \\ 0.25 \end{matrix} = \begin{matrix} 1.75 \\ 1.625 \end{matrix}$$

So **A** turned **v** into **b**. In the graph below, we see how the matrix mapped the short, low line **v**, to the long, high one, **b**.

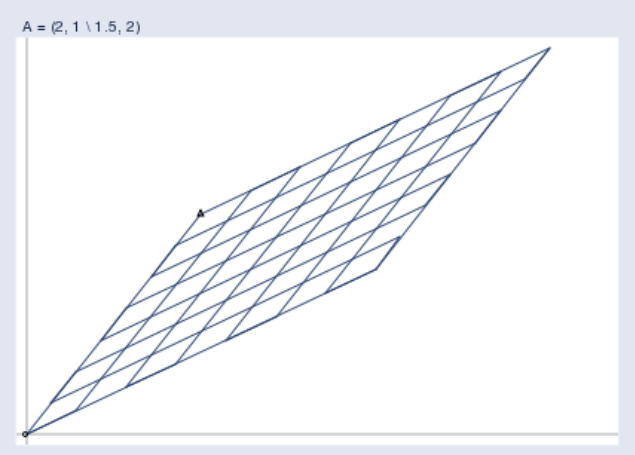


You could feed one positive vector after another into matrix A, and each would be projected onto a new space that stretches higher and farther to the right.

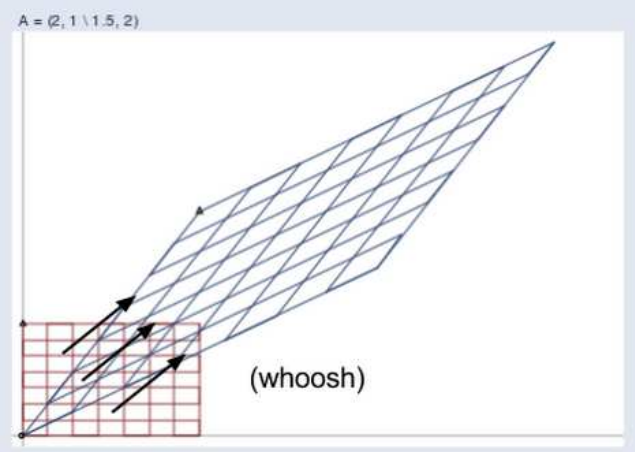
Imagine that all the input vectors **v** live in a normal grid, like this:



And the matrix projects them all into a new space like the one below, which holds the output vectors **b**:

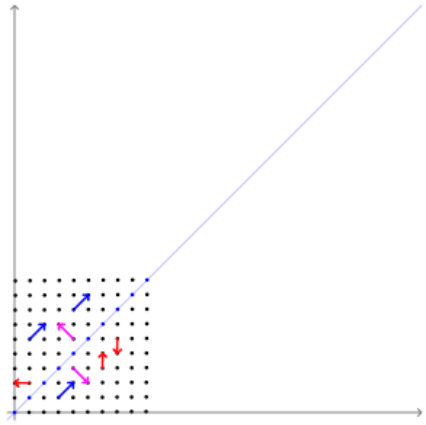


Here you can see the two spaces juxtaposed:



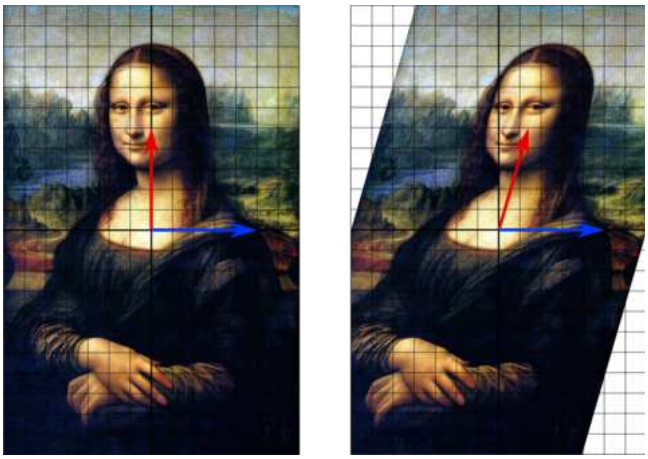
(Credit: William Gould, Stata Blog)

And here's an animation that shows the matrix's work transforming one space to another:



The blue lines are eigenvectors.

You can imagine a matrix like a gust of wind, an invisible force that produces a visible result. And a gust of wind must blow in a certain direction. The eigenvector tells you the direction the matrix is blowing in. You can imagine a matrix like a gust of wind, an invisible force that produces a visible result. And a gust of wind must blow in a certain direction. The eigenvector tells you the direction the matrix is blowing in.



Credit: Wikipedia

So out of all the vectors affected by a matrix blowing through one space, which one is the eigenvector? It's the one that that changes length but not direction; that is, the eigenvector is already pointing in the same direction that the matrix is pushing all vectors toward. An eigenvector is like a weathervane. An eigenvane, as it were.

The definition of an eigenvector, therefore, is a vector that responds to a matrix as though that matrix were a scalar coefficient. In this equation, A is the matrix, x the vector, and lambda the scalar coefficient, a number like 5 or 37 or pi.

$$Ax = \lambda x$$

You might also say that eigenvectors are axes along which linear transformation acts, stretching or compressing input vectors. They are the lines of change that represent the action of the larger matrix, the very "line" in linear transformation.

Notice we're using the plural – axes and lines. Just as a German may have a Volkswagen for grocery shopping, a Mercedes for business travel, and a Porsche for joy rides (each serving a distinct purpose), square matrices can have as many eigenvectors as they have dimensions; i.e. a 2 x 2 matrix could have two eigenvectors, a 3 x 3 matrix three, and an n x n matrix could have n eigenvectors, each one representing its line of action in one dimension.[1]

Because eigenvectors distill the axes of principal force that a matrix moves input along, they are useful in matrix decomposition; i.e. the diagonalization of a matrix along its eigenvectors. Because those eigenvectors are representative of the matrix, they perform the same task as the autoencoders employed by deep neural networks.

To quote Yoshua Bengio:

> **Many mathematical objects can be understood better by breaking them into constituent parts, or finding some properties of them that are universal, not caused by the way we choose to represent them.**
>
> **For example, integers can be decomposed into prime factors. The way we represent the number 12 will change depending on whether we write it in base ten or in binary, but it will always be true that 12 = 2 × 2 × 3.**
>
> **From this representation we can conclude useful properties, such as that 12 is not divisible by 5, or that any integer multiple of 12 will be divisible by 3.**

> **Much as we can discover something about the true nature of an integer by decomposing it into prime factors, we can also decompose matrices in ways that show us information about their functional properties that is not obvious from the representation of the matrix as an array of elements.**
>
> **One of the most widely used kinds of matrix decomposition is called eigen-decomposition, in which we decompose a matrix into a set of eigenvectors and eigenvalues.**

Principal Component Analysis (PCA)

PCA is a tool for finding patterns in high-dimensional data such as images. Machine-learning practitioners sometimes use PCA to preprocess data for their neural networks. By centering, rotating and scaling data, PCA prioritizes dimensionality (allowing you to drop some low-variance dimensions) and can improve the neural network's convergence speed and the overall quality of results.

To get to PCA, we're going to quickly define some basic statistical ideas – **mean**, **standard deviation**, **variance** and **covariance** – so we can weave them together later. Their equations are closely related.

Mean is simply the average value of all **x**'s in the set X, which is found by dividing the sum of all data points by the number of data points, **n**.

$$\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n}$$

Standard deviation, as fun as that sounds, is simply the square root of the average square distance of data points to the mean. In the equation below, the numerator contains the sum of the differences between each datapoint and the mean, and the denominator is simply the number of data points (minus one), producing the average distance.

$$s = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \bar{X})^2}{(n-1)}}$$

Variance is the measure of the data's spread. If I take a team of Dutch basketball players and measure their height, those measurements won't have a lot of variance. They'll all be grouped above six feet.

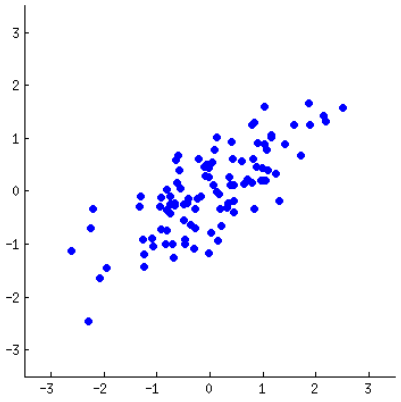
But if I throw the Dutch basketball team into a classroom of psychotic kindergartners, then the combined group's height measurements will have a lot of variance. Variance is the spread, or the amount of difference that data expresses.

Variance is simply standard deviation squared, and is often expressed as **sˆ2**.

$$var(X) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})}{(n-1)}$$

For both variance and standard deviation, squaring the differences between data points and the mean makes them positive, so that values above and below the mean don't cancel each other out.

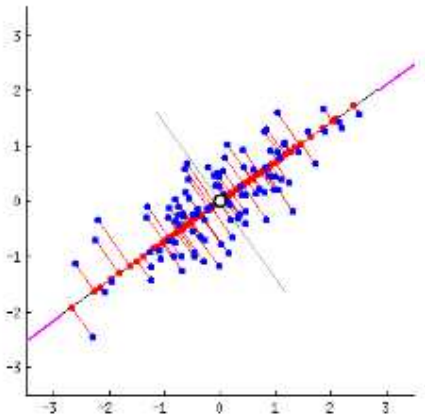
Let's assume you plotted the age (x axis) and height (y axis) of those individuals (setting the mean to zero) and came up with an oblong scatterplot:



PCA attempts to draw straight, explanatory lines through data, like linear regression.

Each straight line represents a "principal component," or a relationship between an independent and dependent variable. While there are as many principal components as there are dimensions in the data, PCA's role is to prioritize them.

The first principal component bisects a scatterplot with a straight line in a way that explains the most variance; that is, it follows the longest dimension of the data. (This happens to coincide with the least error, as expressed by the red lines...) In the graph below, it slices down the length of the "baguette."



The second principal component cuts through the data perpendicular to the first, fitting the errors produced by the first. There are only two principal components in the graph above, but if it were three-dimensional, the third component would fit the errors from the first and second principal components, and so forth.

Covariance Matrix

While we introduced matrices as something that transformed one set of vectors into another, another way to think about them is as a description of data that captures the forces at work upon it, the forces by which two variables might relate to each other as expressed by their variance and covariance.

Imagine that we compose a square matrix of numbers that describe the variance of the data, and the covariance among variables. This is the **covariance matrix**. It is an empirical description of data we observe.

Finding the eigenvectors and eigenvalues of the covariance matrix is the equivalent of fitting those straight, principal-component lines to the variance of the data. Why? Because eigenvectors trace the **principal lines of force**, and the axes of greatest variance and covariance illustrate where the data is most susceptible to change.

Think of it like this: If a variable changes, it is being acted upon by a force known or unknown. If two variables change together, in all likelihood that is either because one is acting upon the other, or they are both subject to the same hidden and unnamed force.

When a matrix performs a linear transformation, eigenvectors trace the lines of force it applies to input; when a matrix is populated with the variance and covariance of the data, eigenvectors reflect the forces that have been applied to the given. One applies force and the other reflects it.

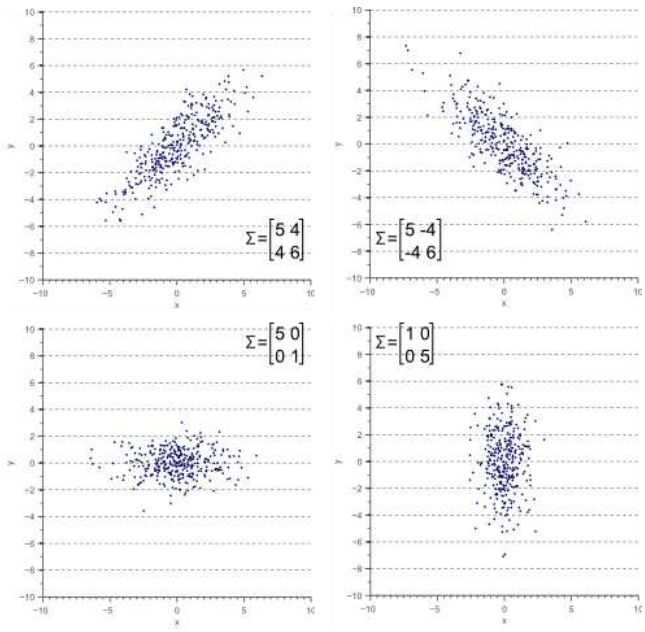
Eigenvalues are simply the coefficients attached to eigenvectors, which give the axes magnitude. In this case, they are the measure of the data's covariance. By ranking your eigenvectors in order of their eigenvalues, highest to lowest, you get the principal components in order of significance.

For a 2 x 2 matrix, a covariance matrix might look like this:

$$\begin{bmatrix} 1.07 & 0.63 \\ 0.63 & 0.64 \end{bmatrix}$$

The numbers on the upper left and lower right represent the variance of the x and y variables, respectively, while the identical numbers on the lower left and upper right represent the covariance between x and y. Because of that identity, such matrices are known as symmetrical. As you can see, the covariance is positive, since the graph near the top of the PCA section points up and to the right.

If two variables increase and decrease together (a line going up and to the right), they have a positive covariance, and if one decreases while the other increases, they have a negative covariance (a line going down and to the right).



(Credit: Vincent Spruyt)

Notice that when one variable or the other doesn't move at all, and the graph shows no diagonal motion, there is no covariance whatsoever. Covariance answers the question: do these two variables dance together? If one remains null while the other moves, the answer is no.

Also, in the equation below, you'll notice that there is only a small difference between covariance and variance.

$$cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{(n-1)}$$

vs.

$$var(X) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})}{(n-1)}$$

The great thing about calculating covariance is that, in a high-dimensional space where you can't eyeball intervariable relationships, you can know how two variables move together by the positive, negative or non-existent character of their covariance. (**Correlation** is a kind of normalized covariance, with a value between -1 and 1.)

To sum up, the covariance matrix defines the shape of the data. Diagonal spread along eigenvectors is expressed by the covariance, while x-and-y-axis-aligned spread is expressed by the variance.

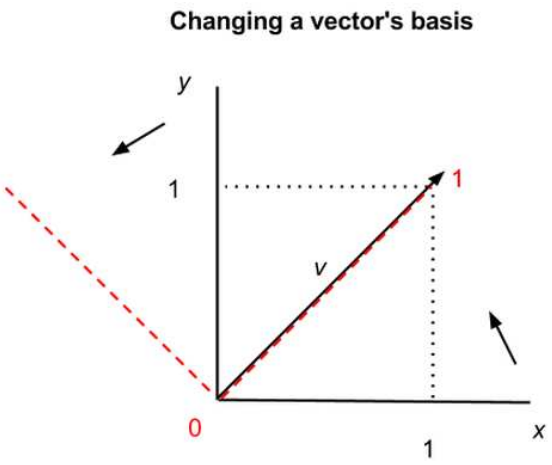
Causality has a bad name in statistics, so take this with a grain of salt:

While not entirely accurate, it may help to think of each component as a causal force in the Dutch basketball player example above, with the first principal component being age; the second possibly gender; the third nationality (implying nations' differing healthcare systems), and each of those occupying its own dimension in relation to height. Each acts on height to different degrees. You can read covariance as traces of possible cause.

Change of Basis

Because the eigenvectors of the covariance matrix are orthogonal to each other, they can be used to reorient the data from the x and y axes to the axes represented by the principal components. You re-base the coordinate system for the dataset in a new space defined by its lines of greatest variance.

The x and y axes we've shown above are what's called the basis of a matrix; that is, they provide the points of the matrix with x, y coordinates. But it is possible to recast a matrix along other axes; for example, the eigenvectors of a matrix can serve as the foundation of a new set of coordinates for the same matrix. Matrices and vectors are animals in themselves, independent of the numbers linked to a specific coordinate system like x and y.



In the graph above, we show how the same vector v can be situated differently in two coordinate systems, the x-y axes in black, and the two other axes shown by the red dashes. In the first coordinate system, v = (1,1), and in the second, v = (1,0), but v itself has not changed. Vectors and matrices can therefore be abstracted from the numbers that appear inside the brackets.

This has profound and almost spiritual implications, one of which is that there exists no natural coordinate system, and mathematical objects in n-dimensional space are subject to multiple descriptions. (Changing matrices' bases also makes them easier to manipulate.)

A change of basis for vectors is roughly analogous to changing the base for numbers; i.e. the quantity nine can be described as 9 in base ten, as 1001 in binary, and as 100 in base three (i.e. 1, 2, 10, 11, 12, 20, 21, 22, 100 <– that is "nine"). Same quantity, different symbols; same vector, different coordinates.

Entropy & Information Gain

In information theory, the term **entropy** refers to information we don't have (normally people define "information" as what they know, and jargon has triumphed once again in turning plain language on its head to the detriment of the uninitiated). The information we don't have about a system, its entropy, is related to its unpredictability: how much it can surprise us.

If you know that a certain coin has heads embossed on both sides, then flipping the coin gives you absolutely no information, because it will be heads every time. You don't have to flip it to know. We would say that two-headed coin contains no information, because it has no way to surprise you.

A balanced, two-sided coin does contain an element of surprise with each coin toss. And a six-sided die, by the same argument, contains even more surprise with each roll, which could produce any one of six results with equal frequency. Both those objects contain **information** in the technical sense.

Now let's imagine the die is loaded, it comes up "three" on five out of six rolls, and we figure out the game is rigged. Suddenly the amount of surprise produced with each roll by this die is greatly reduced. We understand a trend in the die's behavior that gives us greater predictive capacity.

Understanding the die is loaded is analogous to finding a principal component in a dataset. You simply identify an underlying pattern.

That transfer of information, from **what we don't know** about the system to what we know, represents a change in entropy. Insight decreases the entropy of the system. Get information, reduce entropy. This is information gain. And yes, this type of entropy is subjective, in that it depends on what we know about the system at hand. (Fwiw, information gain is synonymous with Kullback-Leibler divergence, which we explored briefly in this tutorial on restricted Boltzmann machines.)

So each principal component cutting through the scatterplot represents a decrease in the system's entropy, in its unpredictability.

It so happens that explaining the shape of the data one principal component at a time, beginning with the component that accounts for the most variance, is similar to walking data through a decision tree. The first component of PCA, like the first if-then-else split in a properly formed decision tree, will be along the dimension that reduces unpredictability the most.

A Beginner's Guide to Graph Analytics and Deep Learning

Graphs are data structures that can be ingested by various algorithms, notably neural nets, learning to perform tasks such as classification, clustering and regression.

TL;DR: here's one way to make graph data ingestable for the algorithms:

Data (graph, words) -> Real number vector -> Deep neural network

Algorithms can "embed" each node of a graph into a real vector (similar to the embedding of a word). The result will be vector representation of each node in the graph with some information preserved. Once you have the real number vector, you can feed it to the neural network.

Concrete Examples of Graph Data Structures

The simplest definition of a graph is "a collection of items connected by edges." There are many problems where it's helpful to think of things as graphs.[2] The items are often called **nodes** or **points** and the edges are often called vertices, the plural of **vertex**. Here are a few concrete examples of a graph:

- Cities are nodes and highways are edges
- Humans are nodes and relationships between them are edges (in a social network)
- States are nodes and the transitions between them are edges (for more on states, see our post on deep reinforcement learning). For example, a video game is a graph of states connected by actions that lead from one state to the next...
- Atoms are nodes and chemical bonds are edges (in a molecule)
- Web pages are nodes and hyperlinks are edges (Hello, Google)
- A thought is a graph of synaptic firings (edges) between neurons (nodes)
- A neural network is a graph ... that makes predictions about other graphs. The nodes are places where computation happens and the edges are the paths by which signal flows through the mathematical operations

Any ontology, or knowledge graph, charts the interrelationship of entities (combining symbolic AI with the graph structure):

- Taxonomies of animal species
- Diseases that share etiologies and symptoms
- Medications that share ingredients

Difficulties of Graph Data: Size and Structure

Applying neural networks and other machine-learning techniques to graph data can de difficult.

The first question to answer is: What kind of graph are you dealing with?

Let's say you have a finite state machine, where each state is a node in the graph. You can give each state-node a unique ID, maybe a number. Then you give all the rows the names of the states, and you give all the columns the same names, so that the matrix contains an element for every state to intersect with every other state. Then you could mark those elements with a 1 or 0 to indicate whether the two states were connected in the graph, or even use weighted nodes (a continuous number) to indicate the likelihood of a transition from one state to the next. (The transition matrix below represents a finite state machine for the weather.)

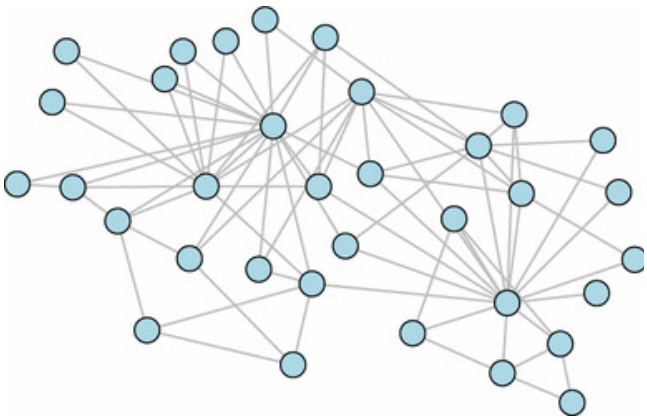
$$
\begin{array}{cc} & \textit{Today} \\ \textit{Yesterday} & \begin{array}{c} \\ \text{sun} \\ \text{cloud} \\ \text{rain} \end{array} \begin{bmatrix} \text{sun} & \text{cloud} & \text{rain} \\ 0.50 & 0.375 & 0.125 \\ 0.25 & 0.125 & 0.625 \\ 0.25 & 0.375 & 0.375 \end{bmatrix} \end{array}
$$

That seems simple enough, but many graphs, like social network graphs with billions of nodes (where each member is a node and each connection to another member is an edge), are simply too large to be computed. **Size** is one problem that graphs present as a data structure. In other words, you can't efficiently store a large social network in a tensor. They don't compute.

Neural nets do well on vectors and tensors; data types like images (which have structure embedded in them via pixel proximity – they have fixed size and spatiality); and sequences such as text and time series (which display structure in one direction, forward in time).

Graphs have an **arbitrary structure**: they are collections of things without a location in space, or with an arbitrary location. They have no proper beginning and no end, and two nodes connected to each other are not necessarily "close".

You usually don't feed whole graphs into neural networks, for example. They would have to be the same shape and size, and you'd have to line up your graph nodes with your network's input nodes. But the whole point of graph-structured input is to not know or have that order. There's no first, there's no last.



The second question when dealing with graphs is: What kind of question are you trying to answer by applying machine learning to them? In social networks, you're usually trying to make a decision about what kind person you're looking at, represented by the node, or what kind of friends and interactions does that person have. So you're making predictions about the node itself or its edges.

Since that's the case, you can address the uncomputable size of a Facebook-scale graph by looking at a node and its neighbors maybe 1-3 degrees away; i.e. a subgraph. The immediate neighborhood of the node, taking k steps down the graph in all directions, probably captures most of the information you care about. You're filtering out the giant graph's overwhelming size.

Representing and Traversing Graphs for Machine Learning

Let's say you decide to give each node an arbitrary representation vector, like a low-dimensional word embedding, each node's vector being the same length. The next step would be to traverse the graph, and that traversal could be represented by arranging the node vectors next to each other in a matrix. You could then feed that matrix representing the graph to a recurrent neural net. That's basically DeepWalk (see below), which treats truncated random walks across a large graph as sentences.

If you turn each node into an embedding, much like word2vec does with words, then you can force a neural net model to learn representations for each node, which can then be helpful in making downstream predictions about them. (How close is this node to other things we care about?)

Another more recent approach is a **graph convolutional network**, which very similar to convolutional networks: it passes a node filter over a graph much as you would pass a convolutional filter over an image, registering each time it sees a certain kind of node. The readings taken by the filters are stacked and passed to a maxpooling layer, which discards all but the strongest signal, before we return to a filter-passing convolutional layer.

One interesting aspect of graph is so-called side information, or the attributes and features associated with each node. For example, each node could have an image associated to it, in which case an algorithm attempting to make a decision about that graph might have a CNN subroutine embedded in it for those image nodes. Or the side data could be text, and the graph could be a tree (the leaves are words, intermediate nodes are phrases combining the words) over which we run a recursive neural net, an algorithm popolarized by Richard Socher.

Finally, you can compute derivative functions such as graph Laplacians from the tensors that represent the graphs, much like you might perform an eigen analysis on a tensor. These functions will tell you things about the graph that may help you classify or cluster it. (See below for more information.)

Further Resources on Graph Data Structures and Deep Learning

Below are a few papers discussing how neural nets can be applied to data in graphs.

Graph Matching Networks for Learning the Similarity of Graph Structured Objects

This paper addresses the challenging problem of retrieval and matching of graph structured objects, and makes two key contributions. First, we demonstrate how Graph Neural Networks (GNN), which have emerged as an effective model for various supervised prediction problems defined on structured data, can be trained to produce embedding of graphs in vector spaces that enables efficient similarity reasoning. Second, we propose a novel Graph Matching Network model that, given a pair of graphs as input, computes a similarity score between them by jointly reasoning on the pair through a new cross-graph attention-based matching mechanism. We demonstrate the effectiveness of our models on different domains including the challenging problem of control-flow-graph based function similarity search that plays an important role in the detection of vulnerabilities in software systems. The experimental analysis demonstrates that our models are not only able to exploit structure in the context of similarity learning but they can also outperform domain-specific baseline systems that have been carefully hand-engineered for these problems.

A Comprehensive Survey on Graph Neural Networks

by Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, Philip S. Yu

Deep learning has revolutionized many machine learning tasks in recent years, ranging from image classification and video processing to speech recognition and natural language understanding. The data in these tasks are typically represented in the Euclidean space. However, there is an increasing number of applications where data are generated from non-Euclidean domains and are represented as graphs with complex relationships and interdependency between objects. The complexity of graph data has imposed significant challenges on existing machine learning algorithms. Recently, many studies on extending deep learning approaches for graph data have emerged.

In this survey, we provide a comprehensive overview of graph neural networks (GNNs) in data mining and machine learning fields. We propose a new taxonomy to divide the state-of-the-art graph neural networks into different categories. With a focus on graph convolutional networks, we review alternative architectures that have recently been developed; these learning paradigms include graph attention networks, graph autoencoders, graph generative networks, and graph spatial-temporal networks. We further discuss the applications of graph neural networks across various domains and summarize the open source codes and benchmarks of the existing algorithms on different learning tasks. Finally, we propose potential research directions in this fast-growing field.

Representation Learning on Graphs: Methods and Applications (2017)

by William Hamilton, Rex Ying and Jure Leskovec

Machine learning on graphs is an important and ubiquitous task with applications ranging from drug design to friendship recommendation in social networks. The primary challenge in this domain is finding a way to represent, or encode, graph structure so that it can be easily exploited by machine learning models. Traditionally, machine learning approaches relied on user-defined heuristics to extract features encoding structural information about a graph (e.g., degree statistics or kernel functions). However, recent years have seen a surge in approaches that automatically learn to encode graph structure into low-dimensional embeddings, using techniques based on deep learning and nonlinear dimensionality reduction. Here we provide a conceptual review of key advancements in this area of representation learning on graphs, including matrix factorization-based methods, random-walk based algorithms, and graph convolutional networks. We review methods to embed individual nodes as well as approaches to embed entire (sub)graphs. In doing so, we develop a unified framework to describe these recent approaches, and we highlight a number of important applications and directions for future work.

A Short Tutorial on Graph Laplacians, Laplacian Embedding, and Spectral Clustering
by Radu Horaud

Community Detection with Graph Neural Networks (2017)
by Joan Bruna and Xiang Li

DeepWalk: Online Learning of Social Representations (2014)
by Bryan Perozzi, Rami Al-Rfou and Steven Skiena

We present DeepWalk, a novel approach for learning latent representations of vertices in a network. These latent representations encode social relations in a continuous vector space, which is easily exploited by statistical models. DeepWalk generalizes recent advancements in language modeling and unsupervised feature learning (or deep learning) from sequences of words to graphs. DeepWalk uses local information obtained from truncated random walks to learn latent representations by treating walks as the equivalent of sentences. We demonstrate DeepWalk's latent representations on several multi-label network classification tasks for social networks such as BlogCatalog, Flickr, and YouTube. Our results show that DeepWalk outperforms challenging baselines which are allowed a global view of the network, especially in the presence of missing information. DeepWalk's representations can provide F1 scores up to 10% higher than competing methods when labeled data is sparse. In some experiments, DeepWalk's representations are able to outperform all baseline methods while using 60% less training data. DeepWalk is also scalable. It is an online learning algorithm which builds useful incremental results, and is trivially parallelizable. These qualities make it suitable for a broad class of real world applications such as network classification, and anomaly detection.

DeepWalk is implemented in Deeplearning4j.

Deep Neural Networks for Learning Graph Representations (2016)
by Shaosheng Cao, Wei Lu and Qiongkai Xu

In this paper, we propose a novel model for learning graph representations, which generates a low-dimensional vector representation for each vertex by capturing the graph structural information. Different from other previous research efforts, we adopt a random surfing model to capture graph structural information directly, instead of using the samplingbased method for generating linear sequences proposed by Perozzi et al. (2014). The advantages of our approach will be illustrated from both theorical and empirical perspectives. We also give a new perspective for the matrix factorization method proposed by Levy and Goldberg (2014), in which the pointwise mutual information (PMI) matrix is considered as an analytical solution to the objective function of the skipgram model with negative sampling proposed by Mikolov et al. (2013). Unlike their approach which involves the use of the SVD for finding the low-dimensitonal projections from the PMI matrix, however, the stacked denoising autoencoder is introduced in our model to extract complex features and model non-linearities. To demonstrate the effectiveness of our model, we conduct experiments on clustering and visualization tasks, employing the learned vertex representations as features. Empirical results on datasets of varying sizes show that our model outperforms other state-of-the-art models in such tasks.

Deep Feature Learning for Graphs
by Ryan A. Rossi, Rong Zhou, Nesreen K. AhmedDeep

Learning multi-faceted representations of individuals from heterogeneous evidence using neural networks (2015)
by Jiwei Li, Alan Ritter and Dan Jurafsky

Inferring latent attributes of people online is an important social computing task, but requires integrating the many heterogeneous sources of information available on the web. We propose learning individual representations of people using neural nets to integrate rich linguistic and network evidence gathered from social media. The algorithm is able to combine diverse cues, such as the text a person writes, their attributes (e.g. gender, employer, education, location) and social relations to other people. We show that by integrating both textual and network evidence, these representations offer improved performance at four important tasks in social media inference on Twitter: predicting (1) gender, (2) occupation, (3) location, and (4) friendships for users. Our approach scales to large datasets and the learned representations can be used as general features in and have the potential to benefit a large number of downstream tasks including link prediction, community detection, or probabilistic reasoning over social networks.

node2vec: Scalable Feature Learning for Networks (Stanford, 2016)
by Aditya Grover and Jure Leskovec

Prediction tasks over nodes and edges in networks require careful effort in engineering features used by learning algorithms. Recent research in the broader field of representation learning has led to significant progress in automating prediction by learning the features themselves. However, present feature learning approaches are not expressive enough to capture the diversity of connectivity patterns observed in networks. Here we propose node2vec, an algorithmic framework for learning continuous feature representations for nodes in networks. In node2vec, we learn a mapping of nodes to a low-dimensional space of features that maximizes the likelihood of preserving network neighborhoods of nodes. We define a flexible notion of a node's network neighborhood and design a biased random walk procedure, which efficiently explores diverse neighborhoods. Our algorithm generalizes prior work which is based on rigid notions of network neighborhoods, and we argue that the added flexibility in exploring neighborhoods is the key to learning richer representations. We demonstrate the efficacy of node2vec over existing state-of-the-art techniques on multi-label classification and link prediction in several real-world networks from diverse domains. Taken together, our work represents a new way for efficiently learning state-of-the-art task-independent representations in complex networks.

Gated Graph Sequence Neural Networks (Toronto and Microsoft, 2017)
by Yujia Li, Daniel Tarlow, Marc Brockschmidt and Richard Zemel

Graph-structured data appears frequently in domains including chemistry, natural language semantics, social networks, and knowledge bases. In this work, we study feature learning techniques for graph-structured inputs. Our starting point is previous work on Graph Neural Networks (Scarselli et al., 2009), which we modify to use gated recurrent units and modern optimization techniques and then extend to output sequences. The result is a flexible and broadly useful class of neural network models that has favorable inductive biases relative to purely sequence-based models (e.g., LSTMs) when the problem is graph-structured. We demonstrate the capabilities on some simple AI (bAbI) and graph algorithm learning tasks. We then show it achieves state-of-the-art performance on a problem from program verification, in which subgraphs need to be matched to abstract data structures.

Graph Classification with 2D Convolutional Neural Networks

Deep Learning on Graphs: A Survey (December 2018)

Viewing Matrices & Probability as Graphs

Graph Convolutional Networks, by Kipf

Diffusion in Networks: An Interactive Essay

Matrices as Tensor Network Diagrams

Viewing Matrices & Probability as Graphs

FOOTNOTES

1. In some cases, matrices may not have a full set of eigenvectors; they can have at most as many linearly independent eigenvectors as their respective order, or number of dimensions.

2. In a weird meta way it's just graphs all the way down, not turtles. A human scientist whose head is full of firing synapses (graph) is both embedded in a larger social network (graph) and engaged in constructing ontologies of knowledge (graph) and making predictions about data with neural nets (graph).

FURTHER READING

- A Recipe for Training Neural Networks, by Andrej Karpathy