Neural Network Exploration

First Author[1] and Second Author[2]

[1]**Address of first author**
[2]**Address of second author**

ABSTRACT

Please provide an abstract of no more than 300 words. Your abstract should explain the main contributions of your article, and should not contain any material that is not included in the main text.

Keywords: Keyword1, Keyword2, Keyword3

INTRODUCTION

When you look at all these polls, there's a lot of them, but there's only so many organizations, copanies, universities, whatever, that're actually running polls. So after a while, you start to get a little used to the same names coming over and over again. One of them is the L.A.

Thanks for using Overleaf to write your article. Your introduction goes here! Some examples of commonly used commands and features are listed below, to help you get started.

METHODS AND MATERIALS

Guidelines can be included for standard research article sections, such as this one.

SOME LaTeX EXAMPLES

Use section and subsection commands to organize your document. LaTeX handles all the formatting and numbering automatically. Use ref and label commands for cross-references.

Figures and Tables

Use the table and tabular commands for basic tables — see Table 1, for example. You can upload a figure (JPEG, PNG or PDF) using the project menu. To include it in your document, use the includegraphics command as in the code for Figure 1 below.

Item	Quantity
Candles	4
Fork handles	?

Table 1. An example table.

Citations

LaTeX formats citations and references automatically using the bibliography records in your .bib file, which you can edit via the project menu. Use the cite command for an inline citation, like Lees-Miller et al. (2010), and the citep command for a citation in parentheses (Lees-Miller et al., 2010).

Figure 1. An example image of a frog.

Mathematics

LaTeX is great at typesetting mathematics. Let $X_1, X_2, \ldots, X_n$ be a sequence of independent and identically distributed random variables with $\mathrm{E}[X_i] = \mu$ and $\mathrm{Var}[X_i] = \sigma^2 < \infty$, and let

$$S_n = \frac{X_1 + X_2 + \cdots + X_n}{n} = \frac{1}{n}\sum_{i}^{n} X_i$$

denote their mean. Then as n approaches infinity, the random variables $\sqrt{n}(S_n - \mu)$ converge in distribution to a normal $\mathcal{N}(0, \sigma^2)$.

Lists

You can make lists with automatic numbering ...

1. Like this,
2. and like this.

... or bullet points ...

- Like this,
- and like this.

... or with words and descriptions ...

Word Definition

Concept Explanation

Idea Text

ACKNOWLEDGMENTS

Additional information can be given in the template, such as to not include funder information in the acknowledgments section.

REFERENCES

Lees-Miller, J., Hammersley, J., and Wilson, R. (2010). Theoretical maximum capacity as benchmark for empty vehicle redistribution in personal rapid transit. *Transportation Research Record: Journal of the Transportation Research Board*, (2146):76–83.

LESSON 8: EIGENVECTORS

One of the nicer way (according to Shewchuk) to look at and understand eigenvectors of symmetric matrices is by looking at a quadratic curve called the quadratic form that comes out of it. To remind you, here's the definition of a eigenvector:

Definition 0.1. Eigenvector Given matrix A, if $Av = \lambda v$, for some vector $v \neq 0$ and λ is a scalar, then v is an eigenvector of A.

So this is the definition you learned in your Linear Algebra class or whatever. But what does it really mean? So what that $Av = \lambda v$ means is there's a special vector v that when I multiply it by A, it still points at the same direction or in the opposite direction, but the line through that vector has not changed at all.

Let's look at a couple examples. I'm gonna draw four coordinate systems here, and it these coordinate systems, I'm gonna put a eigenvector v whose eigenvalue is 2.

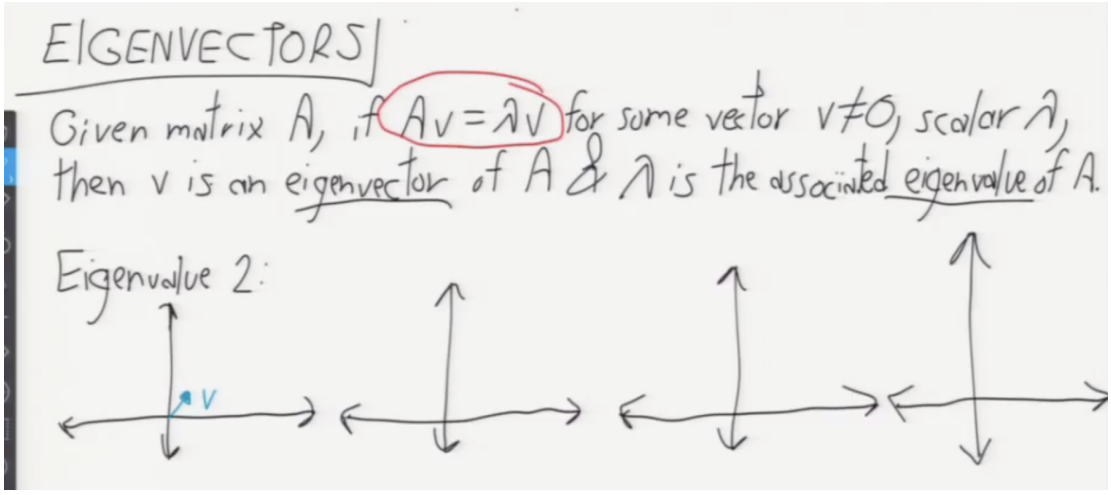

Figure 2. Eigenvector-01

So if I take that vector v and I multiply it by A, since its eigenvalue is 2, I get another eigenvector v that is twice as long and points in the same direction.

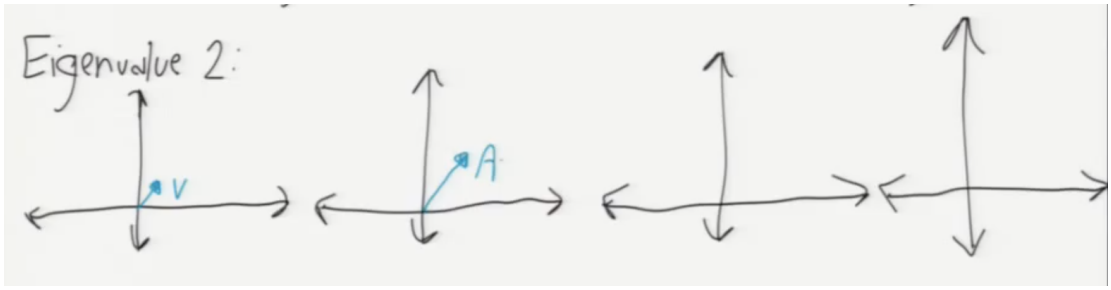

Figure 3. Eigenvector-02

And we can do that again, so if I multiply it by A again, so I get an vector that's twice as long, so I get another vector that points at the same direction. So now it's A^2v:

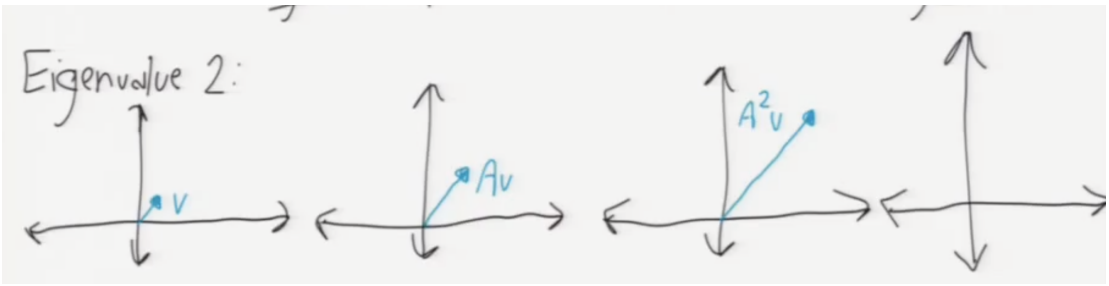

Figure 4. Eigenvector-03

And it I do it again, I get another vector that's twice as long, and points in the same direction:

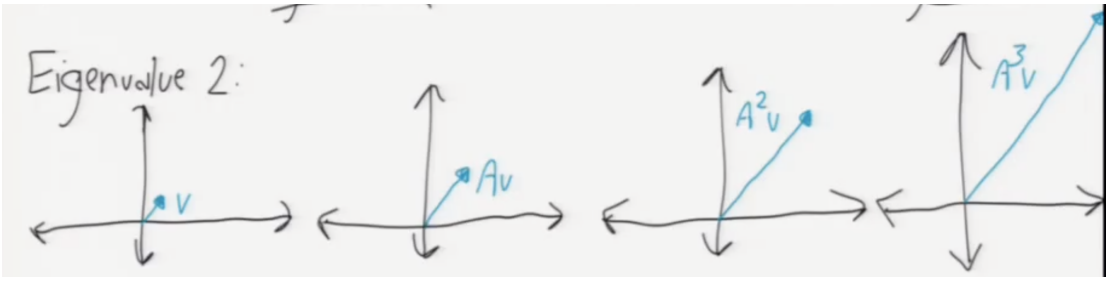

Figure 5. Eigenvector-04

Obviously you can do this as many times as you want. And as an counterpoint, you can consider another example, where the eigenvalue is $-\frac{1}{2}$, cause we can see a few more things that's going on here. One thing is what an negative eigenvalue do and the other being what an eigenvector less that one do. Now let's suppose I got this vector here, which I'll call w, and let's say w is an eigenvector with an eigenvalue that's $-\frac{1}{2}$:

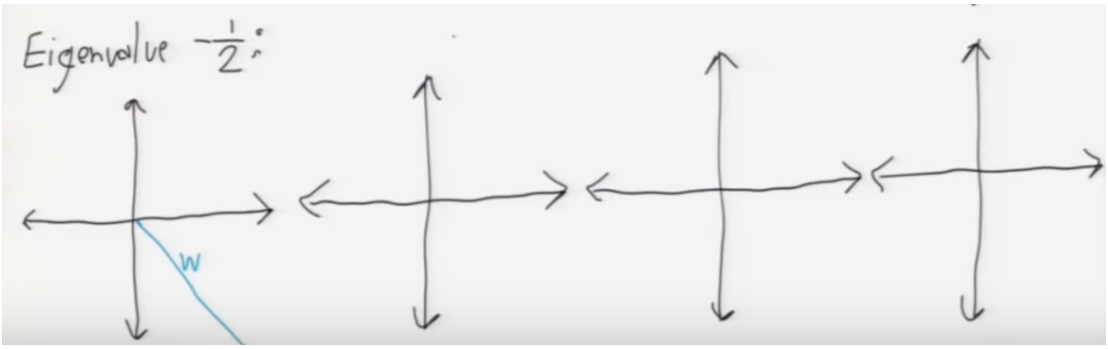

Figure 6. Eigenvector-05

So if I multiply it by A, I get an vector that's sort of pointing at the opposite direction and only half as long (Aw):

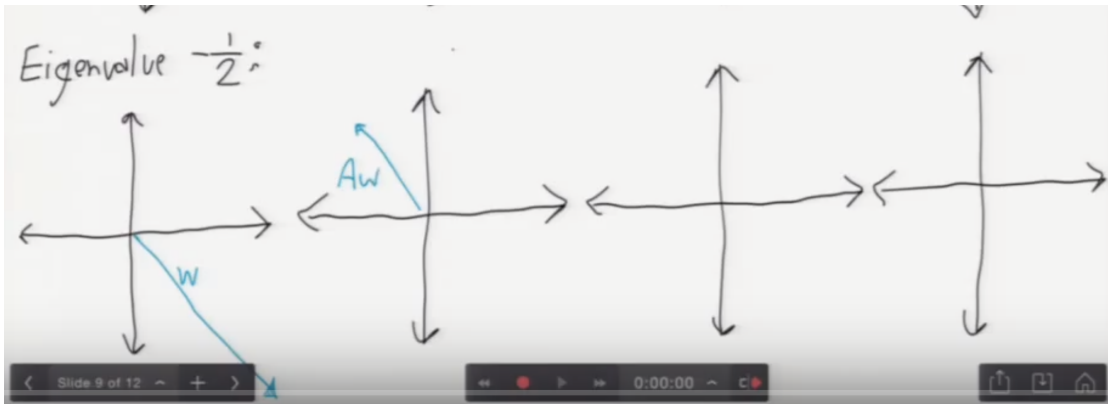

Figure 7. Eigenvector-06

And if I do it again, I get another vector that's only half as long and pointing at the opposite direction (A^2w):

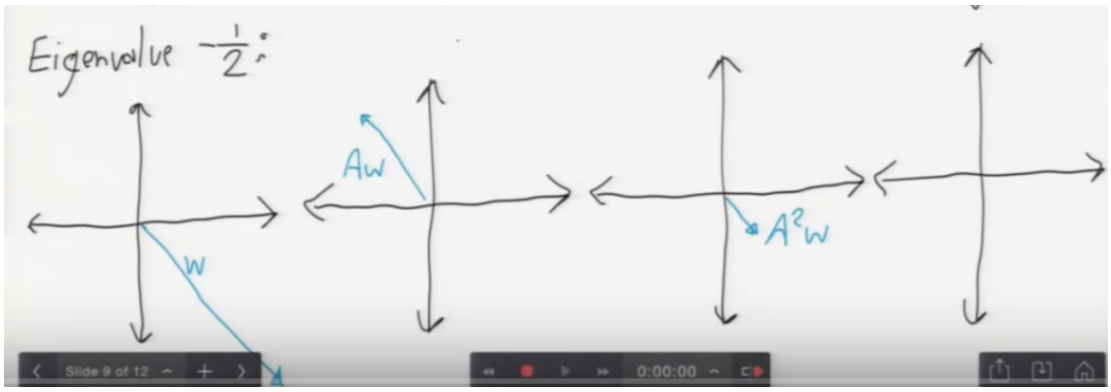

Figure 8. Eigenvector-07

And I can do it again to get an even shorter vector and in the opposite direction (A^3w):

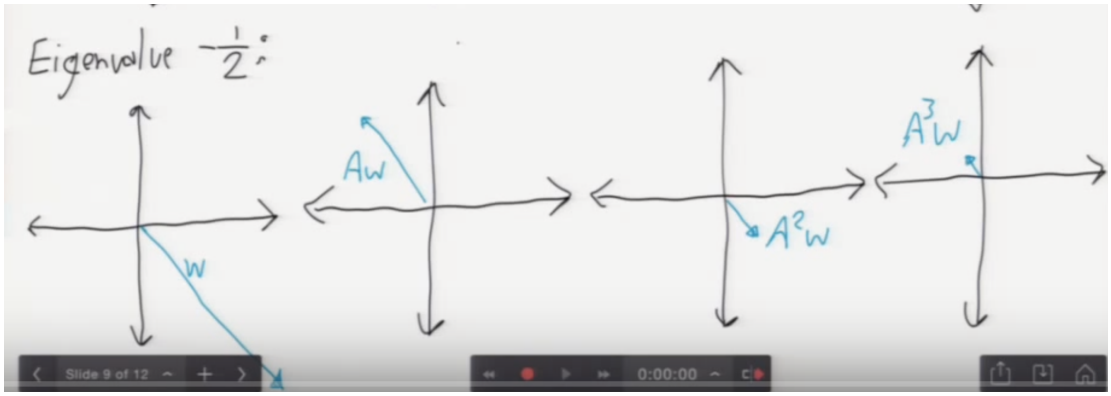

Figure 9. Eigenvector-08

Now for most matrices, most vectors do not have this special property, so they are not eigenvectors. Eigenvectors in vectors are special and rare. And it's clear to show if an vector is an eigenvector, when you scale it, it's still an eigenvector. So when we talk about eigenvectors we only care about their directions. We usually generally don't care too much about the length. That's not implicit in the definition of the eigenvector.

So let's look at a few consequences of this fact:

Theorem 1. If v is an eigenvector of A w/eigenvector λ, then v is an eigenvector of A^k w/eigenvalue λ^k.

Proof. $A^2v = A(\lambda v) = \lambda^2 v$, etc. (Induction) □

And the same thing works for matrix inversion:

Theorem 2. Moreover, if A is invertible, then v is an eigenvector of A^{-1} w/eigenvalue $\frac{1}{\lambda}$.

Proof.

$$\begin{aligned} A^{-1}v &= (\frac{1}{\lambda})(A^{-1})(A)v \quad (\because v = \frac{A}{\lambda}v) \\ &= \frac{1}{\lambda}v \end{aligned}$$

□

There's an interesting thing coming out of these two theorems, which is that I do these things to matrices, I square them, I cubed them, I invert them, the eigenvectors don't change. The eigenvalues change, they get squared, they get cubed, they get inverted, but not the eigenvectors. This is really handy.

Another interesting way to look at the last theorem is to look at the previous diagrams from right to left instead of left to right: I apply A^{-1} from the fourth vector I get the third, I apply it again and I get the second.

The following theorem will be very useful in understanding the effect of an symmetric matrix on a vector that is not an eigenvector. I think you all are familiar with what's called the Spectrum Theorem, which applies only to symmetric matrices:

Theorem 3. Spectrum Theorem: Every symmetric n x n matrix has n eigenvectors that are mutually orthogonal: $v_i^T v_j = 0 \quad \forall i \neq j$.

This is great, this means they can form a coordinate system. The proof for the Spectrum Theorem could take up a whole page so I'm not gonna do it here.

One minor detail I should mention is that you can have situations, for instance, where a symmetric matrix has two different eigenvectors with the same eigenvalue. When that happens, every linear combination of

those two eigenvectors is also an eigenvector with the same eigenvalue. So you wind up with a matrix that has infinite number of eigenvectors, we don't want them all though, those two eigenvectors, they span a plane, all of the eigenvectors that go with them span that same plane. We just need to pick two eigenvectors in that plane that are mutually orthogonal to each other, and that's what we do in practice, cause that's what's useful.

By contrast, for any matrix, its eigenvalues are uniquely defined, the eigenvectors aren't always uniquely defined, sometimes you have to pick and choose, the eigenvalues are always uniquely defined for any matrix, including the multiplicity of the eigenvalues, some eigenvalues may appear more than once. Anyway, the upshot of the Spectrum Theorem is that **when you are dealing with a symmetric matrix, you can use its eigenvalues as a basis for space**. So now I ask what happens to an vector that is not an eigenvector when you apply an symmetric matrix A to it?

When we wanna understand the effect of an symmetric matrix on an ordinary vector that is not an eigenvector, the way we do it is we break it up into a linear combination of eigenvectors.

Recall the two eigenvectors we looked at in previous diagrams, and what I'm drawing here is another vector that's a linear combination of those two and call it the vector x ($x = v + w$):

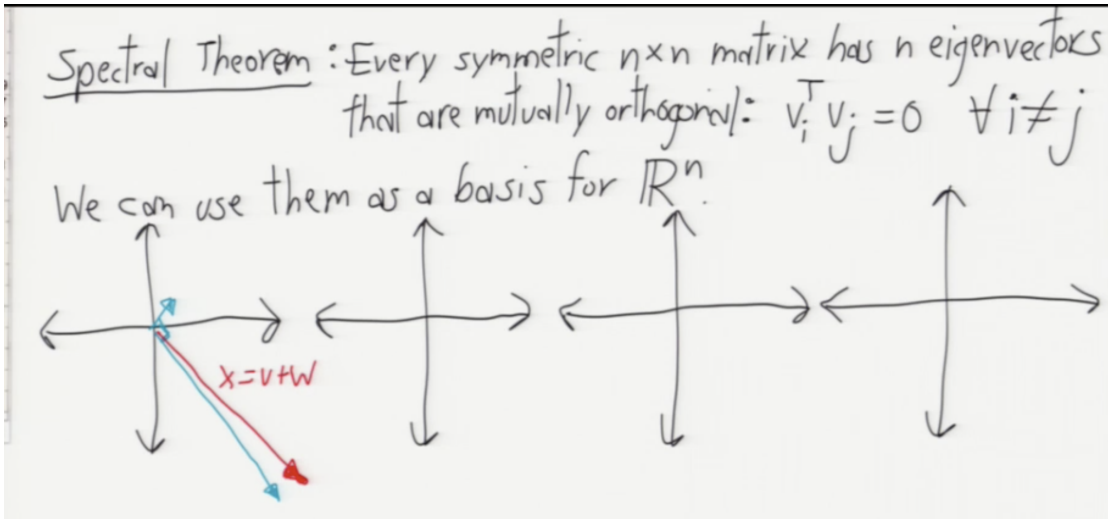

Figure 10. Eigenvector-09

So what happens when I apply the matrix A to this vector x? Well, I decompose it into those two eigenvectors, I look at what A does to each of the two eigenvectors and then I add it up. We've already seen what happens to those eigenvectors when I apply A:

One of them is doubled.

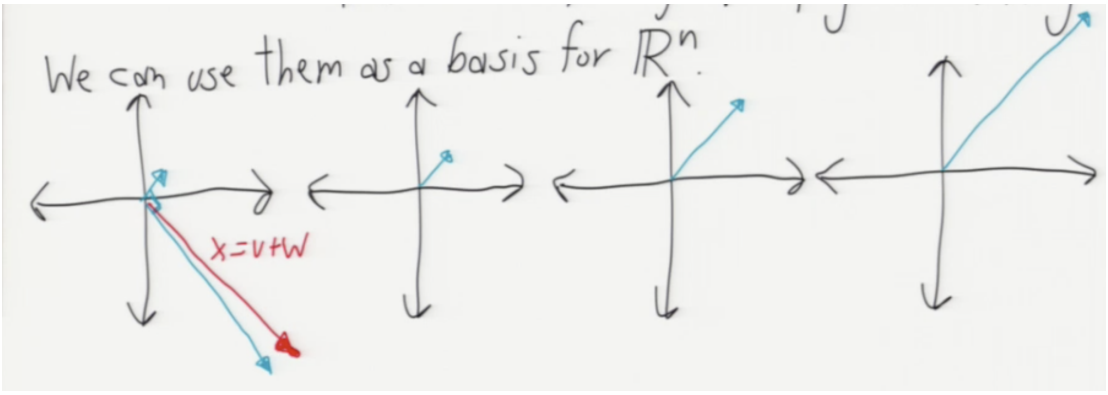

Figure 11. Eigenvector-10

The other one kinda gets half in length and flops back and forth like a dying fish.

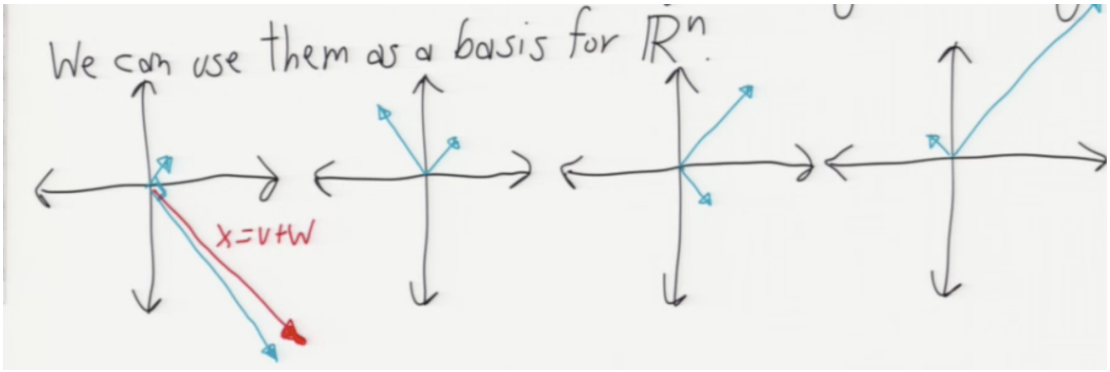

Figure 12. Eigenvector-11

And the vector x, the one that we really care about, is just gonna be the sum of these two components:

And to write down we just saw:

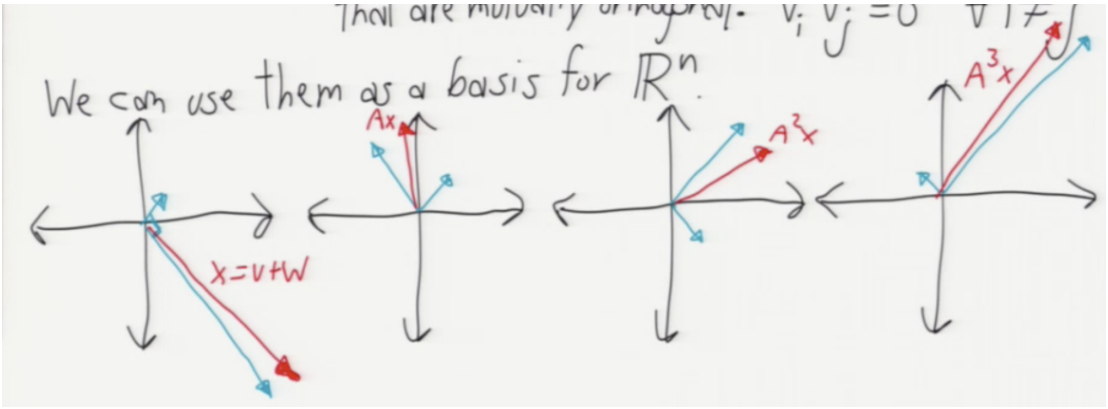

Figure 13. Eigenvector-12

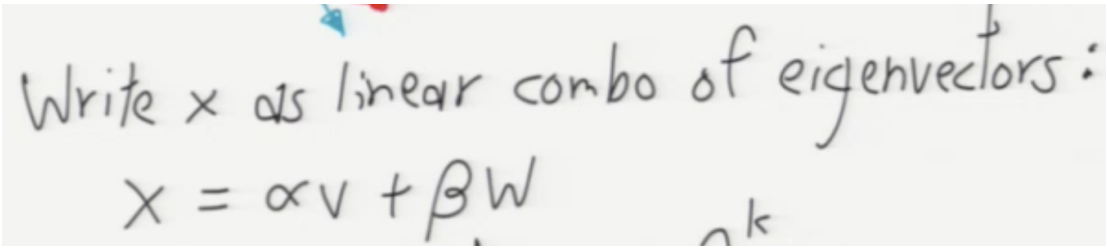

Figure 14. Eigenvector-13

Remember v and w form a basis for a coordinate system. You can think of α and β as the coordinates of x in the coordinate system of the eigenvectors. So when we apply A k times to x, we just need to find out what effect does A^k have on v and w, and so that's what we get.

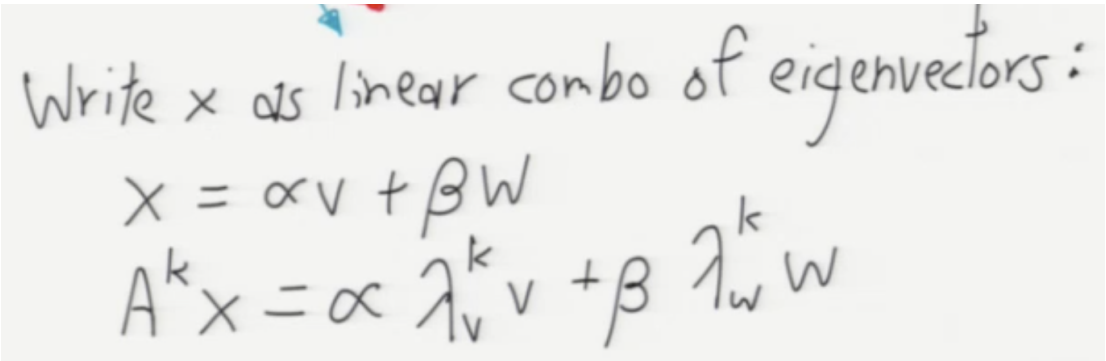

Figure 15. Eigenvector-14

MACHINE LEARNING

Machine learning is all about finding patterns from data. If you wanna describe machine learning in four words, that's probably the best I can do. And we don't just find patterns for the sake of saying "Wow, cool pattern dude", we actually use them to make predictions. See now, the reason why the NSA is going through all of your credit receipts right now is because they want to know if you're going to bomb this classroom tomorrow, and this is how you predict it.

Now, kinda a big part of the core of machine learning is models and statistics, which are the things that help us understand patterns. Models tend to be a vague word, but it has a very specific in machine learning, which I'll try to flash our normally in a few lectures. So, a model does not mean it's how things look like. Data, if they are totally random, has no structure, but the data we're gonna look at in the real world almost always has some structure, almost always has some sort of smoothness to it. If it didn't, it wouldn't be predictable and there'll nothing for us to do. But, the question is, what is the structure of that smoothness, what do I mean when I say it's smooth or that it's regular, there're many different ways that real-world data can exhibit smoothness or regularity. And a model is a particular way your data exhibits smoothness or regularity. It's a lot easier though once you have examples of models, so take a few classes of what models usually mean. But, what you do is you sub out a bunch of models and say I believe the thing I'm trying to model is a smooth quadratic curve, for instance, or close enough to a quadratic curve with a little noise thrown in. And then, you use statistics to try to backward guess what is the most likely smooth curve that would be responsible for the data that you see.

And the way we select a particular pattern that fits the model that we have is usually through techniques such as numerical optimization. And so we have numerical optimization algorithms that could learn the patterns, most commonly.

So one thing to understand machine learning is that we change by the exposure of data, the data is super important, the data is the most important part in machine learning. The data drives everything else. You cannot learn much if you don't have enough data. You can guess things about the world, you usually guess wrong. And also, if you have lots of data, but your data sucks because it's really noisy, or it's just wrong for some reason, like systematic errors, then you can't learn much. but, if you have lots of and lots of and lots of good data, it's amazing what you can learn, and part of what's been changing machine learning and making it a very exciting field in the last few years is that suddenly with the Internet we have huge streams of data at our finger tips. So you can go download millions of photographs, you can download a million of parrots, if you want, and there're now people who are creating 3D street views of Paris, just by assembling millions and millions of photographs together that they found on the Internet and using them, when you have enough data, you can make a very accurate model of all kinds of things.

Another thing I should mention is that there were some machine learning techniques that have gone in and out of style, and probably the most important one is neural nets. Neural nets have gone through three phases of great popularity, separated by two phases where they were just ignored, and now we're in the third phase of great neural net popularity, specifically because we've discovered that if you have really really large datasets, and you pound those datasets with fast compute servers and parallel cloud distributors that we are computing, you can do amazing and amazing things that you couldn't do before.

So, I want to look at an example of a machine learning task. This particular task obtained a lot of fame during the second wave of popularity of neural networks when a very good neural network was trained to recognition of digits, written on postal codes, for the U.S. Post Office. And, so, eventually it became a time when 15% of the mails sent through the U.S. the post code were read by the automatic postal code readers, based on neural nets. And so the problem here is, people write digits, there's 10 digits we usually care about and we wanna distinguish them from each other. FOr the sake of discussion right now, we just simplify the task and pretend there's only two digits: 7 and 1. And so if you have a bunch of hand-written sevesn, the top row, and you have a bunch of hand-written ones, then how do you learn to distinguish them from each other?

So, the first step is collecting lots and lots of training images, there they are. Often that's the hardest of the machine learning task by the way. but we'll move on to the more interesting parts.

So, the digits that we're given, they are pixels, in some small box. Let's pretend it's a really small box,

and each of those pixels has some intensity. Right now, we're just looking at black and white images, so it'll be a grayscale intensity. ANd so, you can think of the input for one digit is looking like this:

Obviously it's gonna be bigger than four-by-four, but that's what I can draw right now. What we're gonna do with these digits is for now, this may seem like a really dumb thing to do, but we're just gonna express it as a vector, we'll be writing all the pixel values vertically. So there we go, we've take our digit, we have converted it into a big one-dimensional array of one-numbers. And when we do that, we think of that grid of digits as a point in 16-dimensional space. So, as a general rule in machine learning, you have some data, you take that data, and you imagine it to be some point in high-dimensional space. I know it's hard to think of high-dimensional spaces sometimes but that's what we're gonna do a lot in this class.

So we now have a bunch of points in 16-dimensional space and you want to know is there some structure, like can we build the wall, so that all the sevens are at one side of the wall, and the ones at the other side of the wall. If we can do that, we can distinguish sevens from ones.

So this brings us to the idea of classification, now I just got points in 16-dimensional space, I can't draw 16-dimensional space. So I'm gonna start you off with an example in an 2-dimensional space. The idea here is that instead of digits,

NEURAL NETWORK

In machine learning, what we usually deal with are two kinds of problems: classification and regression. And neural networks can do both. Neural networks tie a lot of different ideas together like perceptrons, logistic regression, ensemble of learners, and of course, stochastic gradient descent to train them. You can lift samples from a lower dimension feature-space to a higher dimensional space. But neural network has a great twist, which is they can actually learn features on their own if you can think of good ones. Maybe a neural net will find the best features for you.

I wanna to begin by reminding you of the story I told you at the beginning of the semester, about Frank Rosenblatt's invention of perceptrons in 1957. Remember that he held a press conference where he predicted that perceptrons would be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Perceptron research continued until something monumental happened in 1969. Marvin Minsky, one of the founding fathers of AI, and Seymour Papert published a book called "Perceptrons." Sounds good, right? Well, part of the book was devoted to things perceptrons can't do. And one of those things is XOR.

If we look at our traditional exclusive OR table:

		x_1	
	XOR	0	1
x_2	0	0	1
	1	1	0

Figure 16. XOR Table

You got two inputs x_1 and x_2, and the output, you can see them as points in a 2-dimensional space. So these aren't just numbers I'm drawing here, these are four sample points in a 2-dimensional space: two of them are of Class 0, and two of them are of Class 1. And remember perceptrons can only classify sample points that are linearly separable.

Is this linearly separable? Can we divide it with this line?

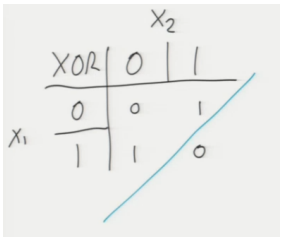

Figure 17. XOR Table Line 1

Nope. No good.

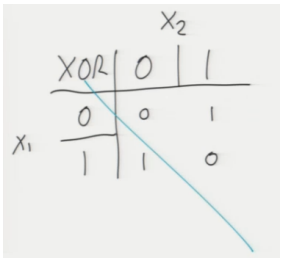

Figure 18. XOR Table Line 2

What about this line?

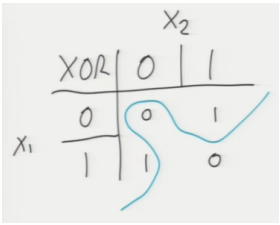

Figure 19. XOR Table Line 3

There we go. But that's not a line. These four points are not linearly separable. So, this is one of the most basic things you can think of about logic gates, but if you can't do that, how can you convince me you can solve more complicated problems, let alone mimicking how a human think or classifying images or something like that? You are telling me this machine is gonna be conscious of its own existence but it can't do XOR? This book has a devastating effect on the field. After its publication, almost no research was done on neural nets and similar ideas for about an decade. And that's a time we now call the AI Winter. And shortly after the book was published, Frank Rosenblatt died. Officially his cause of death was a boating accident, but we all know he died of a broken heart.

Looking at this problem, one thing I can't quite understand though is why at the time why didn't some people point out some obvious ways of getting around this problem. Here's the easiest way to get around this problem:

If you add one new feature to bring our points to a 3-dimensional space, we're gonna do a quadratic feature, the one that particularly helps in this situation is the one where you multiply the two inputs x_1x_2 together. And having done that, XOR is linearly separable in 3-dimensional space. And this is how it could look like:

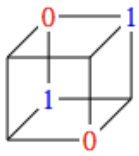

Figure 20. XOR Cube

So that's one way to solve the XOR problem, but there's an even more powerful way to do XOR which is what led us to the idea of neural nets. And the idea is:

> ***Idea:*** *let's design some classifiers who outputs are inputs for other linear classifiers.*

Let's picture the perceptrons are like this.

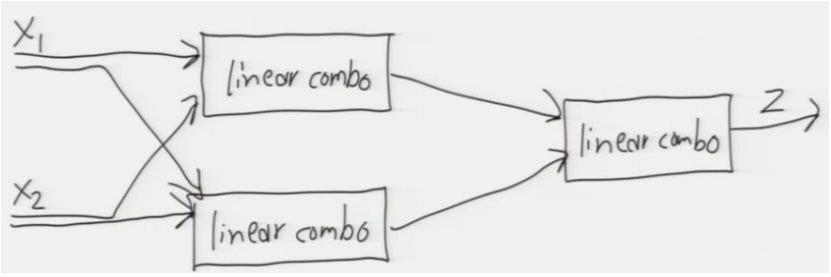

Figure 21. Linear Combination Boxes

We have a box which represents a linear combination of its inputs x_1 and x_2. And then I do a second box that computes a different linear combination of the same things as inputs. And then I'll take the outputs of those and feed them into another linear combination. And there we have an output z. So I ask you: let's suppose that our interpret rule for the output is if z is positive we're gonna call it a binary 1, if z is negative, we're gonna call it a binary 0. So can we do XOR with this setup?

No, because it's the same thing as before. There's a problem here, which is that a linear combination of a bunch of linear combinations is just a linear combination of the original inputs. And so that means this cannot work for XOR, because it can only work for things that are linearly separable. And XOR is not linearly separable.

So yeah, this doesn't work. But, fortunately, this idea is really close to something that CAN work. We need one more idea to make neural nets work. And the idea is we have add some sort of non-linearity between first two boxes and third box on the right. And by putting the right kind of non-linearity, we break this "a linear combo of other linear combos" and this allows us to start doing more interesting things that are not linearly separable. And in fact, they can get arbitrarily complicated because a little non-linearity in the middle is gonna allow us to build arbitrarily boolean circuits, for instance.

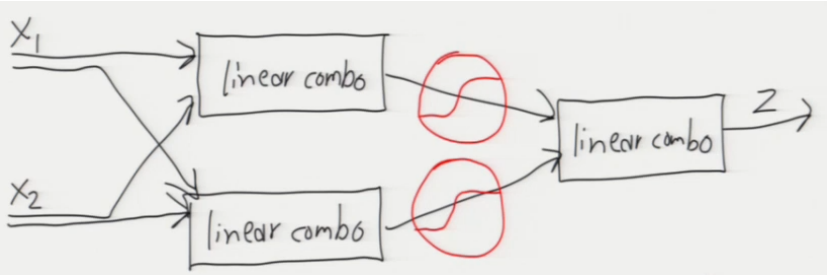

Figure 22. Linear Combination Boxes with Nonlinearity

So what I'm proposing is that there needs to be some sort of modifiers here that takes the input and maps it to a slightly different output before we go to the next linear combination box. And there's more than one function we can choose to introduce that non-linearity. But by far, the most popular is the logistic function, the logistic function seems to work really well in logistic regression for classification, so why not see if it'll work well here. So what that means is the logistic function is this S-shaped function, the output always varies between 0 and 1, however big or small the input is, and that's gonna introduce the right kind of non-linearity to allow us to do very general things.

By the way, it doesn't have to be this complicated non-linearity to work, an non-linearity is simple as clamping the output so it cannot go below zero, can work too. And in fact, it's also used in practice a lot. but we'll start with the sigmoid and move on to other possibilities later.

So, the logistic function has a couple of nice characteristics. One is that since it's always bounded between 0 and 1, it's not going to get two big and over-saturate the other boxes or neurons, as we call them, that its output is going into. So it's nice that it can't get too big, it doesn't get too over-saturate the next neuron down the line. Another nice thing about it is that it's continuous smooth differentiable, well-defined derivatives. And that's helpful because it means we can do gradient descent relatively easily. And indeed, we're gonna train these things with gradient descent.

So if we use this logistic function, then here's how we can do XOR, with the three boxes I just drawn, but this time I'll add logistic function to make this work. And the idea is if I have inputs with column X and Y for simplicity, and I write the following linear combination, $30 - 20x - 20y$, and then we run the output through the logistic function, that actually acts an awful lot like an NAND gate, providing the inputs x and y are in some reasonable range like 0 to 1. And, I'm gonna add a similar gate at the bottom, which acts like an OR gate by taking the linear combination $20x + 20y - 10$, and run it through the logistic function. So we take the output of those two gates and I can devise an AND gate by taking $20v + 20w - 30$ with the logistic function. And the output is essentially x XOR y.

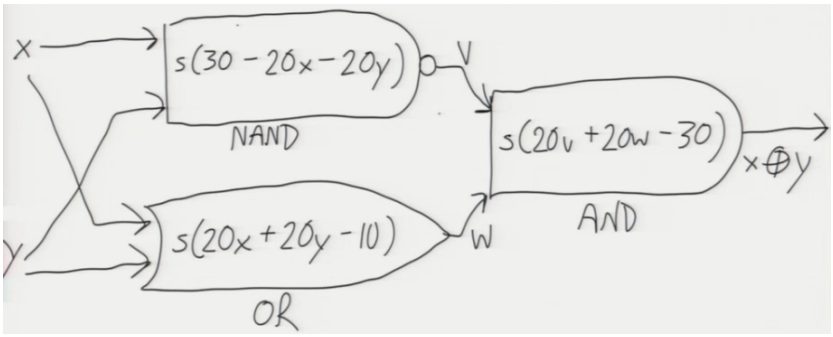

Figure 23. XOR with Nonlinearity

Of course it's not exactly x XOR y, because the function never outputs exactly 0 or 1, but it gets pretty close to 0 or to 1, so what we do is for a value that is lower than half, we count it as a 0 and for a value that is more than half, we count it as a 1.

And the reason why I chose these large constants here (20, 20 and 10) is just to make sure it's large enough to drive the gate to saturation. Either the linear combination is gonna be ten or higher, or it's gonna be -10 or lower, providing the input being 0 or 1. So then once you apply the logistic function, the output is gonna be pretty close to the 0, 1. And that's how you do XOR in a neural net.

Network with 1 Hidden Layer

The next thing I wanna do is setup a specific neural net and some notation that we can work with so we can start deriving algorithms and in particular, figure out how to do gradient descent.

I'm gonna start with a neural network that has one hidden layer of neurons or units. There are three layers in total because neural nets generally have an input layer, the X you'll put into your classification algorithm, and then you'll have an output layer, which is Y, and sometimes we might have more than one output, so we may have more than one Y.

So again, the input layer is the usual usual, we have a vector x that is our input vector, and it has d components. I'm also going to have a convention that every input vector has a hidden $d+1$ component, which is 1, the usual fictitious dimension trick.

We're gonna have a layer of hidden units, and their outputs are gonna be h_1 through h_m where m is the number of hidden units, or the number of hidden neurons if you prefer. And again, we're gonna use the fictitious dimension trick as well, because we want to be able to have all linear combinations and a linear combination has a constant 1.

And we'll have an output layer, which I'm gonna call these Zs, the predicted outputs and there will be up to k of those.

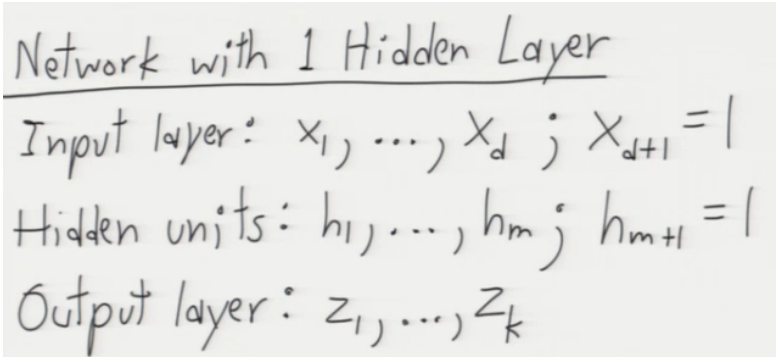

Figure 24. Network with 1 Hidden Layer

Now, through out this semester we've been doing various classification algorithms. We take an X in, and your output is usually just like one thing, like in classification it's one class, in regressions the output is one number. Now we're gonna sometimes have more than one output, what's special about that?

So, generally if we're using an algorithm like SVM, or logistic regression, if you want to do more than one output, that's just like doing more than one classifier. So you train every classifier separately, once for each output. With neural nets it actually makes sense for us to group outputs together and the reason why is sometimes you get unexpected benefits by having all your outputs come from the same set of hidden units, because what can happen sometimes is that a hidden unit that's being trained to support one of these outputs suddenly becomes unexpectedly useful for correctly predicting another one of the outputs, and you wind up predicting that other output better than you predicted it with just training that one output by itself. So by training outputs, sometimes you get better classifiers in some of those outputs than you training them apart because they're sharing particularly intelligent hidden units.

We need to have a notation for the weights for those linear combinations. For layer 1 weights, We're just gonna express them as a matrix, called V. And the reason why it is a matrix is because every input is gonna contribute to every hidden unit. So we have a matrix that says how much of these inputs contribute to this particular hidden unit, what is the coefficient of the linear combination that links them together. And I'll use the notation V_i for row i of the matrix. We also have another layer of weights, I'll call it layer 2. They connect these hidden units to the output. And I'll call that matrix W, again W_i represents the ith row of that matrix, which is important to that particular output, row i corresponds to output i.

We'll be using the logistic function and so, let me just remind you what that is. Now again, as I said before, other non-linear functions can be used and are often used. I'll probably talk about them on Monday.

And I want to apply s not just to a scalar, but to vectors, and when I apply it to a vector, it's just applying it component by component, nothing special. So, when we have a vector v, and I write $s(v)$, that just means applying s to each of the component of the vector.

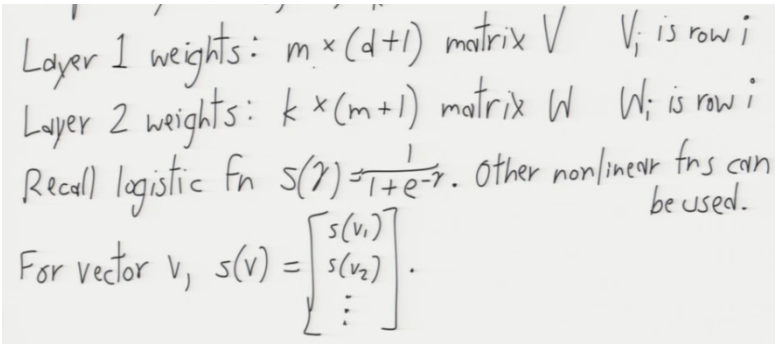

Figure 25. Network with 1 Hidden Layer Notations

Alright, so that's the notation that we'll use. Now let me draw these things so you can see how everything relates. I'm gonna have three hidden units in the middle, I'm gonna have two input units on the left, and two output units on the right. For this particular example, of course, neural nets are usually bigger than these, but we want to keep our drawing within bounds. And every input has a connection to every hidden unit, and every hidden unit has a connection to every output. And each of these edges has one weight. These edges on the left, their weights are stored in the matrix V. These edges on the right, their weights are stored in the matrix W. So for instance, this edge here is V_{11}, this edge here is V_{21}, you put first the number of neuron it's going to and then the number of neuron it's coming from. So V_{21} means we're going from X_1 to h_2, cause the matrix calculation works out right that way. Likewise, on the output layer, we'll have weights in the W matrix, and they are numbered in the same way.

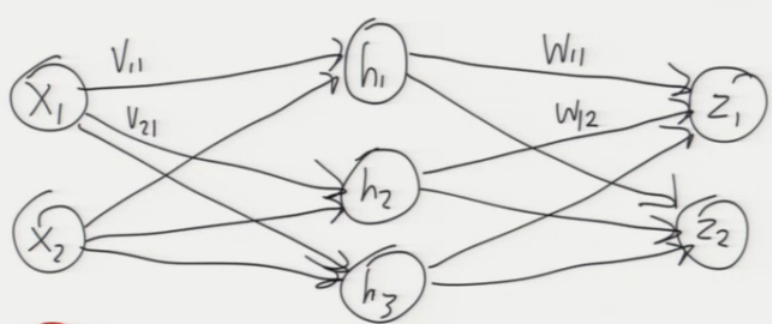

Figure 26. Network with 1 Hidden Layer

I'm also including a few ghost variables that are equal to one. So, remember X_3 is equal to one, h_4 is equal to one, we want to do any linear combinations, so we need a constant term for every linear combination. So, imagine the arrows are here, you know, we won't actually have a unit per se, in your code, for these two nodes, but we do have these red edges. Those are the bias weight, the constant weights in your linear combinations. And so, they are important too.

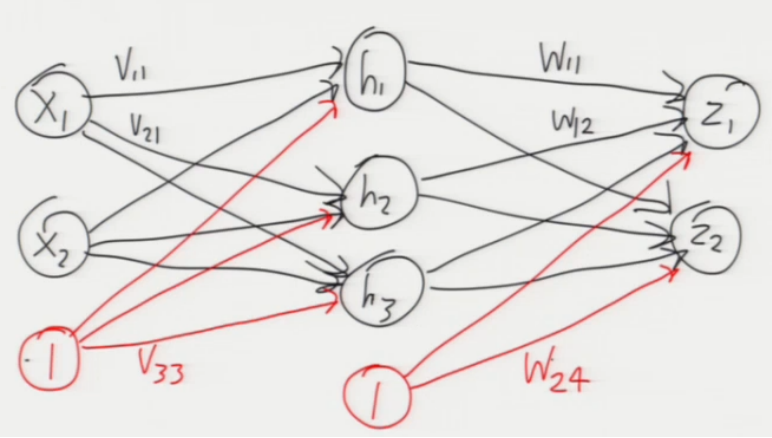

Figure 27. Network with 1 Hidden Layer with Bias

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_d & 1 \end{bmatrix}^T, \quad \mathbf{h} = \begin{bmatrix} h_1 & h_2 & \dots & h_m & 1 \end{bmatrix}^T, \quad \mathbf{z} = \begin{bmatrix} z_1 & z_2 & \dots & z_k \end{bmatrix}^T$$

$$\mathbf{V} = \begin{bmatrix} V_{1,1} & V_{1,2} & \dots & V_{1,d+1} \\ V_{2,1} & V_{2,2} & \dots & V_{2,d+1} \\ \dots & \dots & \dots & \dots \\ V_{m,1} & V_{m,2} & \dots & V_{m,d+1} \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} W_{1,1} & V_{1,2} & \dots & V_{1,m+1} \\ W_{2,1} & V_{2,2} & \dots & V_{2,m+1} \\ \dots & \dots & \dots & \dots \\ W_{k,1} & V_{m,2} & \dots & V_{k,m+1} \end{bmatrix},$$

Now we have everything, and we can start writing the formula here:

So, for each hidden unit, that's gonna be equal to the logistic function applied to the linear combination. So we sum over all of these input edges, the weight of that edge gimes the input value:

$$h_i = s(\sum_{j=1}^{3} V_{ij} x_j)$$

And of course, we can write them shorter using the matrix notation:

$$h = s(V \cdot x)$$

So h is equal to applying the logistic function s to every component of the vector $V \cdot x$.

And we do the same thing to compute z from the hidden units h.

$$z = s(W \cdot h)$$

z is gonna be applying s to W multiplied by h, so... weight matrix times hidden unit vector and then you apply the logistic function, component by component, to this vector $W \cdot h$.

And if you want to write this in composition, you can write the outputs in terms of the inputs:

$$z = s(W \cdot h) = s(W \cdot s_1(V \cdot x))$$

Here, I write s_1 just to remind myself to add 1 to the end of the vector $V \cdot x$ so before we can do the next matrix multiplication.

Training: Gradient Descent

Now, what we want to do is to train them, and we'll train them using gradient descent. I would say that using stochastic gradient descent, meaning one sample point at a time is the most common way to train them. But, batch gradient descent exists too.

In order to train, we first always have to decide on what is the function we want to optimize, so we need to pick a **loss function**, $L(z,y)$, where z is the neural net's vector of predictions, that's the output. And the y vector, is the true values of what we really want to fit. Note that y can be a vector or scalar.

For now, we'll just use the square error as the loss function:

$$L(z,y) = |z-y|^2$$

Remember this is an interchangeable part of a neural net. you can swap this out and swap in some other loss function pretty easily. Once you have a loss function, then you need a **cost function** that looks not just up one input x, but all of the training samples. So for the cost function, we're just gonna use the simplest thing that we usually do, which is we're gonna take the sum over all of the training points of the loss function. Therefore, the cost function is

$$J(h) = \sum_{i=1}^{n} L(h(X_i), Y_i)$$

I'm using capital Y because I'm now assuming there's an input matrix Y that has one row for each sample point and one column for each output of the neural net, i.e., assuming we have n samples,

$$\mathbf{Y} = \begin{bmatrix} Y_{1,1} & Y_{1,2} & \dots & Y_{1,d} \\ Y_{2,1} & Y_{2,2} & \dots & Y_{2,d} \\ \dots & \dots & \dots & \dots \\ Y_{n,1} & Y_{n,2} & \dots & Y_{n,d} \end{bmatrix}$$

Up till now, we always use a vector y because we only have one output, now we can have more than one input, so let's use a matrix.

Now, one of the big differences between neural nets and logistic regressions, even though neural nets look a lot like logistic regressions, is that there's no longer just one local minimum. When you do logistic regression, we have a **convex cost function**, which is great, because that means there's only one minimum, which is also the global minimum, once we're there we're happy. Neural nets are like the opposite of that, there's probably billions and billions of local minima when you do a really large training set with a large network. And there's no guarantee that you're gonna find the best one, you can just hope you can find a good one. And part of the complications we're gonna deal with is how do you find a good one, but right now, let's just worry about how to find a local minimum at all.

Let's suppose we have this **cost function**, and we wanna do **gradient descent**, and so we start with all the weights set to zero, and we start doing gradient descent on this cost function. What is going to go wrong?

Looking at the diagram of the neural net, if you set all these weights to zero, there is a symmetry between these weights in that what is the difference between x_1 and x_2, what is the difference between h_1 and h_2 and h_3, why should anyone of those weights come out differently than any of the other weights? What's gonna happen is if all those weights start the same, then the direction of steepest descent is going to try to change them all by exactly the same amount, and they're always gonna stay the same. You have a symmetry wrecking problem here. You have to make different hidden units, focus on different parts of the problem, and the way we do that is pretty trivial, we just start with random weights rather than starting with zero weights, and so far that seems to be working reasonably well in practice.

Alright, just to write down the gradient descent algorithm, it's pretty simple.

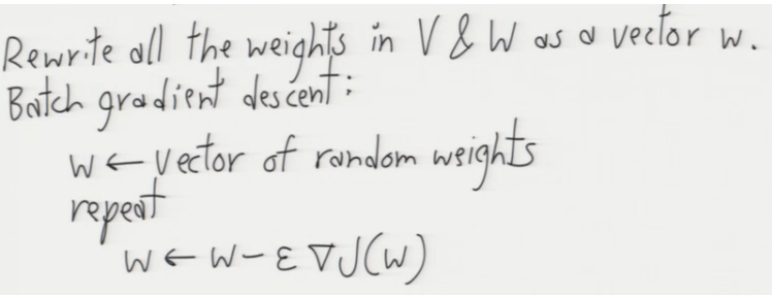

Figure 28. Gradient Descent Algorithm

- Let's remember, we are storing these weights in two matrices, V and W, so let's rewrite those as a vector, w. Then we do batch gradient descent in the usual manner.
- we will start with w being a vector of random weights, to break symmetry, and then we just repeat the gradient descent step.
- Looking at $w \leftarrow w - \varepsilon \nabla J(w)$, we're descending on the whole cost function. We could do stochastic gradient descent instead, which requires us making this to be the **loss function** for one sample point. And we choose that sample point randomly or choose that sample point by shuffling all the sample points and going through in that shuffled order.
- The important thing is that this is still the same old gradient descent algorithm you know and love, but now the weights, we think of them as being stored in two matrices, that doesn't change how gradient descent works, you just have to remember that we have to treat them as vectors in gradient descent.
- It's important that you don't pick the random weights too big, because if the input to the neurons are big, the neurons will saturate, that means our output is very close to 1 or 0, and once a neuron is saturated, it's very hard to get it to stop being saturated. You can actually get into situations like the neurons are kinda stuck, and stuck means for every input example point, the neuron is always very close to 1 or 0. And when a neuron gets stuck like that, it's hard for it to get unstuck because the gradient is gonna be close to zero as well, so you wanna start with random weights that are not too big.

So the hard part of this algorithm is gonna be computing the gradient $\nabla J(w)$ here. That's actually quite tricky to do fast. It's possible to write out the formula, and if you write out the formula, what you'll find is that for any one weight, the gradient formula involves lots and lots of other neuron values and actually takes linear time to compute, linear to the number of neurons. So again, the derivative for just one weight is gonna be $O(number of neurons in the network)$. And then you'll have to do that for each weight, that means the number of neurons times the number of weights.

Fortunately, there's a clever way of getting around this, it's called **backpropagation**. Probably a lot of you have heard of the backpropagation algorithm which is kinda considered as the center of neural nets, or at least of how you train a neural net. And what that does is it gets the running time down to linear, specifically linear to the number of edges (weights), cause you have to do that to every edge at least once,

to change its weight. But this is a hugely important way of making neural net practical, that you can compute the gradient in linear time instead of essentially quadratic time. It still can be quite slow, but at least it makes neural nets trainable. So we're gonna spend quite a bit of time understanding how to do this.

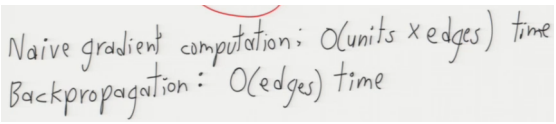

Figure 29. Big O

Backpropagation

One way to think about backpropagation is it's a glorified version of the chain rule in very large expressions. So what we're gonna do is we're gonna have to go through several phases of learning before we get to backprop. Let's start with computing gradients for an arithmetic expression.

Computing Gradients for Arithmetic Expressions

Let's suppose we have a general arithmetic expression, and we write it out as a circuit. How do you compute a gradient for that?

And we're gonna do that by looking at the following expression:

- We're gonna take the sum of numbers a and b and call the output d.
- And the output d we're gonna multiply by another number c
- And then we're gonna take all of that (call it e) and we're gonna square it. So that output is f.

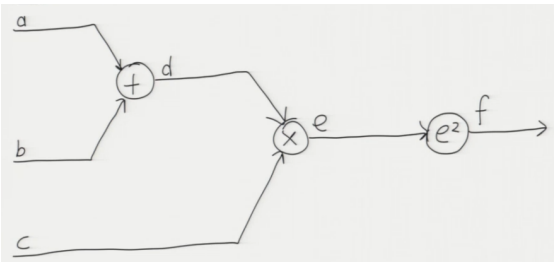

Figure 30. Arithmetic Expressions

The goal I wanna do here is to compute the gradient of f. And when I say the gradient of f it not with respect to all the variables here, of course we can compute all five of them, but right now the ones I really want is I want the gradient of f with respect to the inputs a, b, and c:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial a} \\ \\ \frac{\partial f}{\partial b} \\ \\ \frac{\partial f}{\partial c} \end{bmatrix}$$

So the question now is how do I do that?

Let's starting by going from left to right. At a, what we want is we want the partial derivative of f w.r.t a. It looks kinda complicated because there're all these levels of computations between a and f. But what I can do is I can apply the chain rule. So I can rewrite this partial derivative in terms of a by throwing d into the mix.

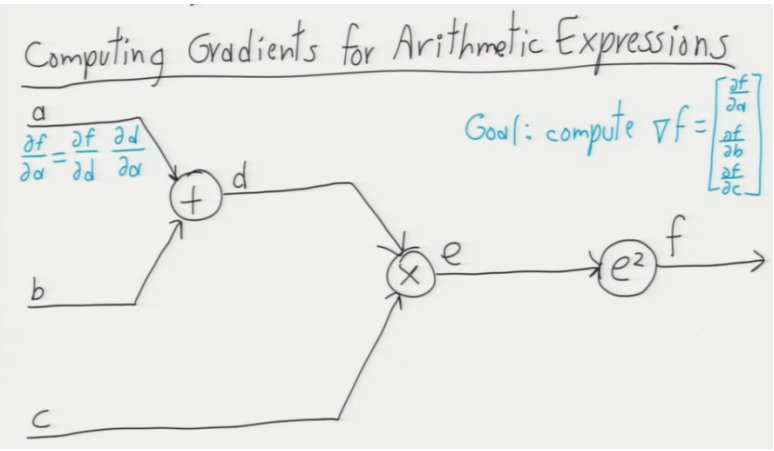

Figure 31. Arithmetic Expressions

And I'm gonna do the same thing to the other variables here.

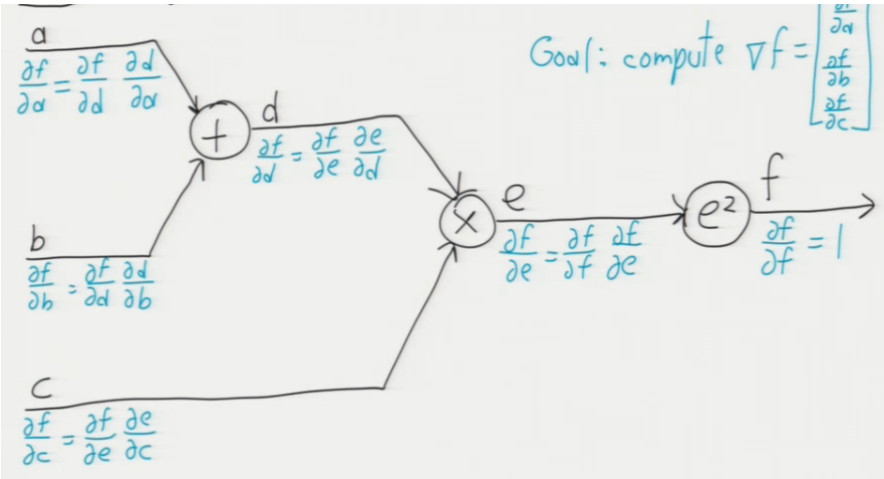

Figure 32. Arithmetic Expressions

So now we got all these partial derivatives we need to compute. And we can compute some of them immediately as the numbers are moving forward through the circuit, but not all of them. Let's look at which ones we can compute. So, we have here $d = a + b$, and let's figure out what the partial derivatives are:

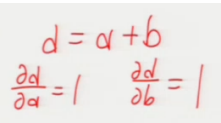

Figure 33. Arithmetic Expressions

With these information we can substitute them in the diagram:

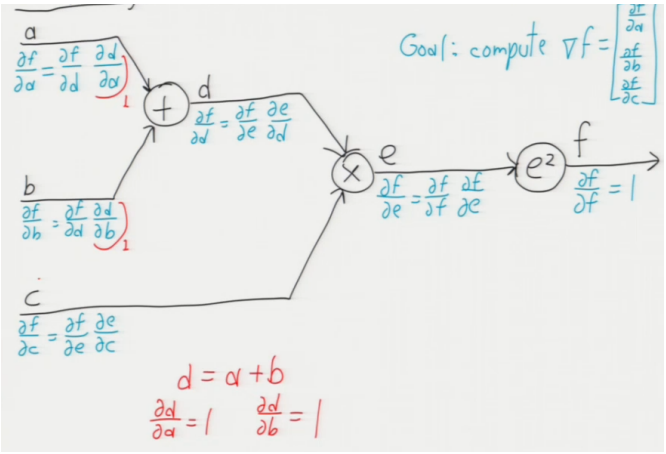

Figure 34. Arithmetic Expressions

I'm gonna do the same thing for e now, $e = cd$, so we have:

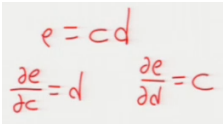

Figure 35. Arithmetic Expressions

Again, we're gonna rewrite those in their proper places:

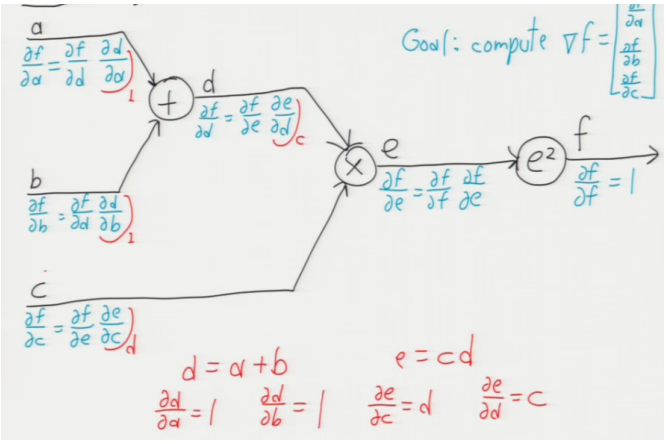

Figure 36. Arithmetic Expressions

And lastly, we have $f = e^2$. And we do the same thing.

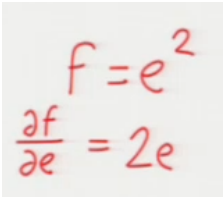

Figure 37. Arithmetic Expressions

So now the diagram looks like:

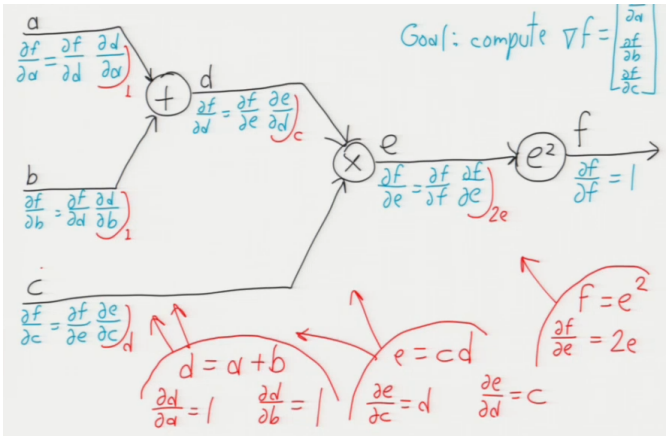

Figure 38. Arithmetic Expressions

As you can see, basically I got one half of all these products done. And so, let's simplify these:

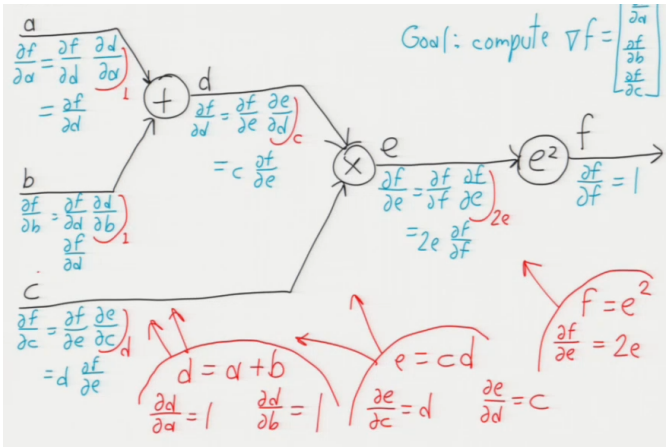

Figure 39. Arithmetic Expressions

So that is as much information as I can figure out about these derivatives on the **forward-pass**. but we're gonna fill in the rest of these missing values on the **backward-pass**. And that's the part we call **backpropagation**.

So, the idea is we start at the end of the circuit, and then we work backward. So, we got $\frac{\partial f}{\partial f} = 1$, we take that and we substitute it in $\frac{\partial f}{\partial e}$, and since we now know what $\frac{\partial f}{\partial e}$ is, we can fill in $\frac{\partial f}{\partial d}$ and $\frac{\partial f}{\partial c}$, and lastly we can fill in $\frac{\partial f}{\partial a}$ and $\frac{\partial f}{\partial b}$.

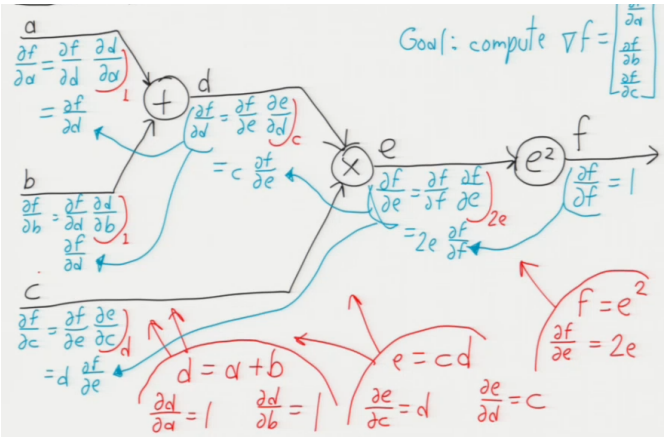

Figure 40. Arithmetic Expressions

And so we now have computed the gradient of f. And so one thing I really wanna emphasize is as you can see here these arrows pointing to the left, that is not you write on the paper and do the substitution and figure out what the big long expression is over there and type that expression in your code. No, what happens is at runtime, you're training your neural network, you're gonna compute this number at runtime, then you're gonna go backward and your code will look at the inputs and substitute the appropriate values there, and take that value and substitute it into a and b. And this is what allows us to get these derivatives in linear time rather than quadratic time, because of the fact that as we go from right to left, we're computing values in a kinda dynamic-programming-like way where first you're solving this subproblem here, which allows you to solve the subproblem there, which allows you to solve the subproblem there, which looks really much like dynamic programming on a tree.

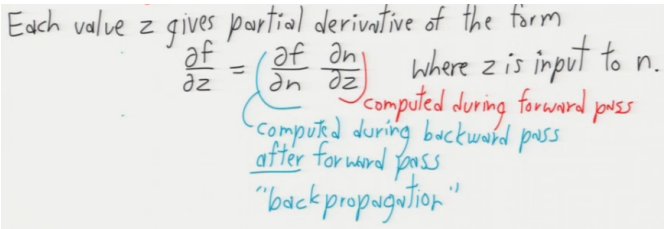

Figure 41. Arithmetic Expressions

So for every value our neural net produces, including intermediate values, we have a partial derivative of this form: the derivative of the loss function or the risk function f over our variable z, we ask: well, what is z input to? If it's an input to another variable n, then we compute $\frac{\partial f}{\partial n}$ and $\frac{\partial n}{\partial z}$, and we multiply those two things together, it's always the same pattern, and it's always possible to compute $\frac{\partial n}{\partial z}$ during the forward pass. On the other hand, we can backward pass this later on to get $\frac{\partial f}{\partial n}$. And this whole idea is called "backpropagation".

More than one unit

So I want to give you one more example before we go to backpropagation proper because what if a unit's output goes to more than one unit?

I want to do an example that looks more like a neural net. So let's have a unit that actually computes the linear combination just like a neuron does, we're not gonna do the sigmoid yet, we're just gonna do the linear combination part. So I'll have two hidden units that just do linear combinations here. And their inputs are column x_1 and x_2. And their outputs will go toward a loss function., a square-loss function.

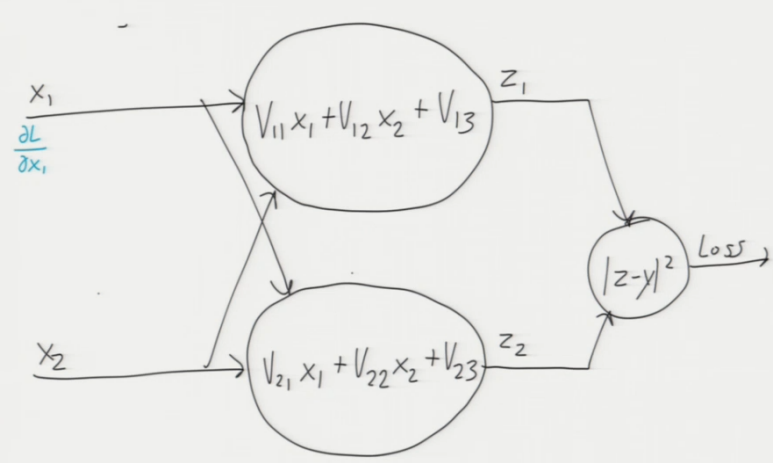

Figure 42. Arithmetic Expressions

So when you do backpropagation for this there's one additional trick that comes up with partial derivatives, which is that our partial derivatives are now more complicated than you'd like them to be because what is the derivative of loss function over x_1. in the last example I gave you, the inputs only went into one operator, but now x_1 is going into two operators, and so you have to know the rules of calculus and partial derivatives for when L depends on x_1 in more than one way. L depends on x_1 in two ways through the top neuron and the bottom neuron. So, when you write allthe partial derivative of L w.r.t x_1, it's actually going to be a sum of two terms. First, you need the derivative of loss w.r.t to z_1 time the derivative of z_1 w.r.t x_1, but you also need the derivative of the loss w.r.t to the second neuron as well, i.e. z_2, and of course, x_2 is gonna follow the same rules.

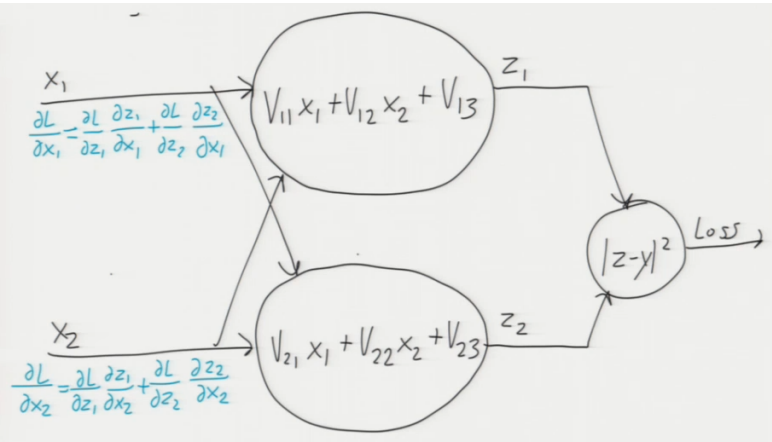

Figure 43. Arithmetic Expressions

Now I can simplify the first one by actually taking the derivatives of z_1 w.r.t x_1, which is pretty simple, cause that's just a constant.

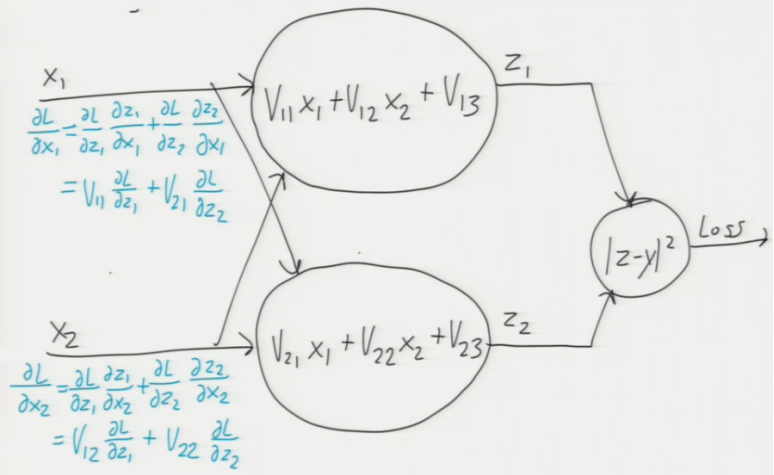

Figure 44. Arithmetic Expressions

So this is as close as I can get to those derivatives during forward propagation. To get the rest of the missing derivatives here, I'm gonna have to go up and do that part of the circuit and then back-prop those derivatives. So at the front end of the circuit, we found out that the derivative of the loss function w.r.t the first hidden unit is $2(z_1 - y_1)$, and the derivative of the loss w.r.t the second hidden unit is $2(z_2 - y_2)$. And having done that I can back-prop those values to where they are needed.

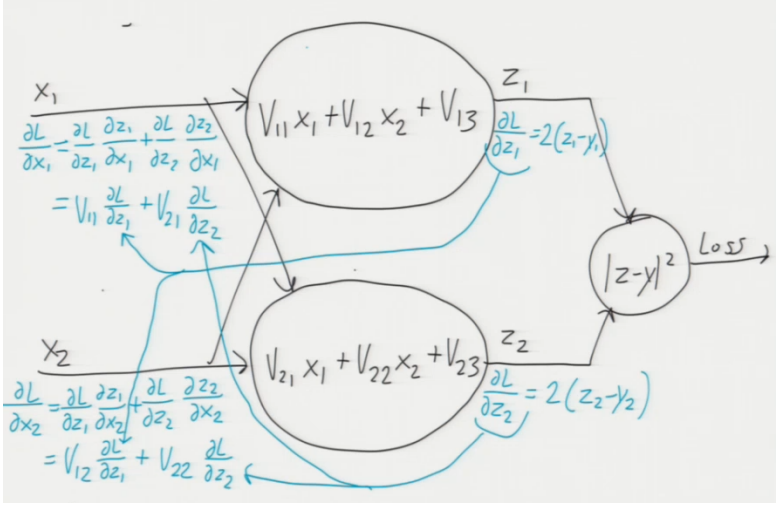

Figure 45. Arithmetic Expressions

So one more time, on the forward pass we compute x_1, x_2, z_1, z_2, and the loss. We also have these weights here, which we already know the values of. On the backpropagation step, we take these derivatives here, we backpropagate them to x_1 and x_2, so we can figure out the gradient of the loss function w.r.t. x.

Backprogation on Neural Net

Okay, we're ready to do an actual neural net now. I want to quickly recall a few things. We know the gradient of the logistic function, we're gonna need that. Again, we need to pick a loss function, so we'll use $|z-y|^2$. And I want you to remember from the neural net setup that the hidden unit values are computed by $h_i = s(V_i \cdot x)$, hidden unit i is computed by applying the logistic to row i of the matrix V, dot producted with x. Therefore, what I can do is I can take the gradient of h_i w.r.t the row of the matrix V that matters here. This will look a little weird here, I'm taking w.r.t one row of a matrix, so we take the row that the hidden unit i depends on, and we're gonna do a gradient w.r.t that row, which tells us how that row should change during gradient descent. By using chain rule, we take the gradient of the logistic function, and then chain rule says you multiply it by the vector x itself. So this gradient is gonna be proportional to the vector x.

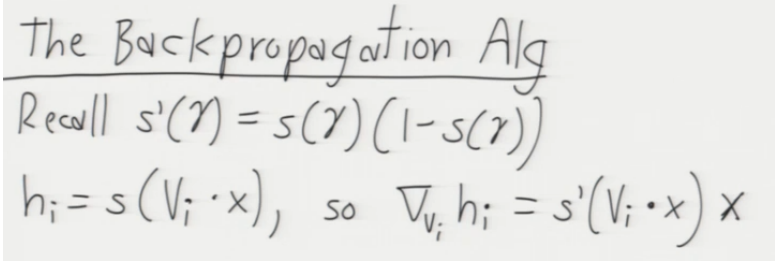

Figure 46. Arithmetic Expressions

So now, let's look at what is the actual back-prop look like as expressions. Once you reduce things to matrix form, it's not all that complicated a circuit. Now you notice this does not look like the neural net I drew you before, the previous neural net form I cared about the numbers, now I care about how the weights change, so now I'm going to draw the weights as the inputs, forget about the X's, for the purpose of training, the X's are constants, so I'm not gonna draw them as inputs cause we don't care about gradients w.r.t X. All we care about is the gradient w.r.t the weights: the weight matrix V and the weight matrix W. So from the weights' point of view, this is what the neural net looks like:

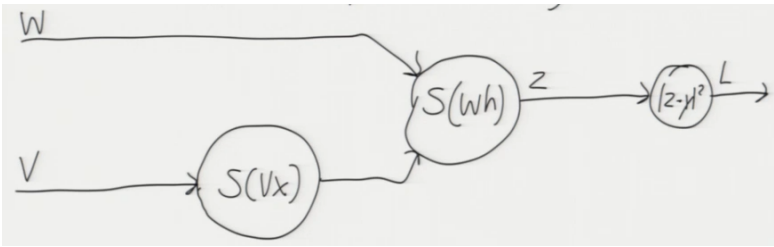

Figure 47. Arithmetic Expressions

This is what the neural net looks like now, that's the whole neural net. It helps that W and V are whole matrices, we can deal with a bunch of variables at once, and now, let's compute the gradients of these things. So, the gradient w.r.t row i of matrix W of the loss function is the derivative of L w.r.t output z_i, times the gradient w.r.t w_i of z_i. So, we're looking the output of one hidden unit at a time here, and this equation holds for every value of i, once for each output unit, and the same thing holds true for V.

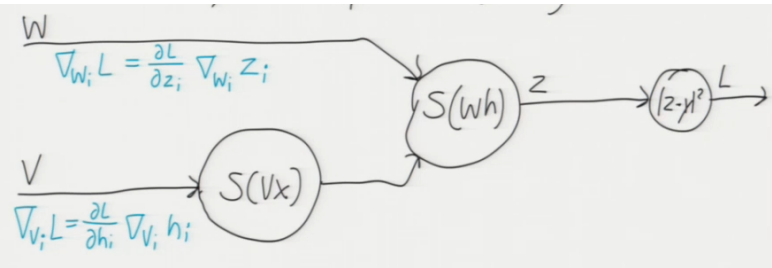

Figure 48. Arithmetic Expressions

So the first one $\nabla_{w_i} L$, we know the second gradient, we just don't know the first derivative. And likewise, taking the derivative of L w.r.t the weight V, I know half of this formula, and I'm missing the other half, which we'll get from backpropagation.

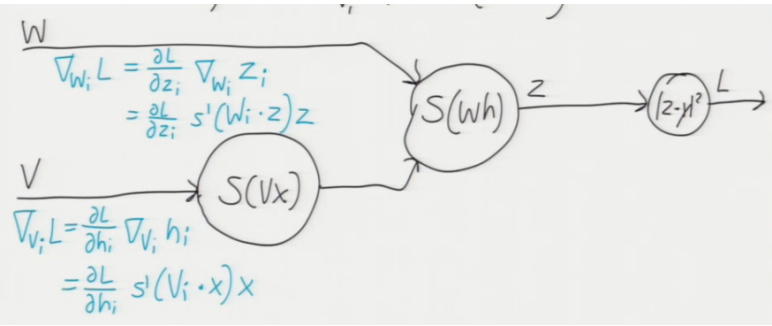

Figure 49. Arithmetic Expressions

I forgot to label it, these are the hidden units h. And we need their gradients too: the derivative of the loss function w.r.t the hidden unit i is that summation you recall from the previous section. Now I'll do the summation over all of the output units because every output unit is gonna have an influence on every hidden unit. So this derivative is a sum of all of these products. And again, we can fill in half of those.

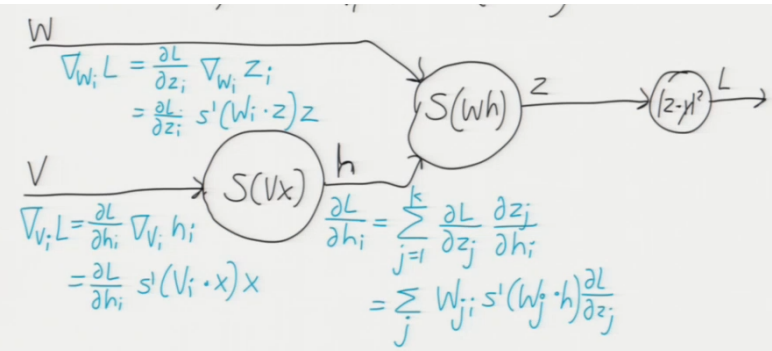

Figure 50. Arithmetic Expressions

And lastly, we need the derivative of the last function in terms of each of the output units. So let's look at one of the output unit j, that's the easiest, that just $2(z_j - y_j)$.

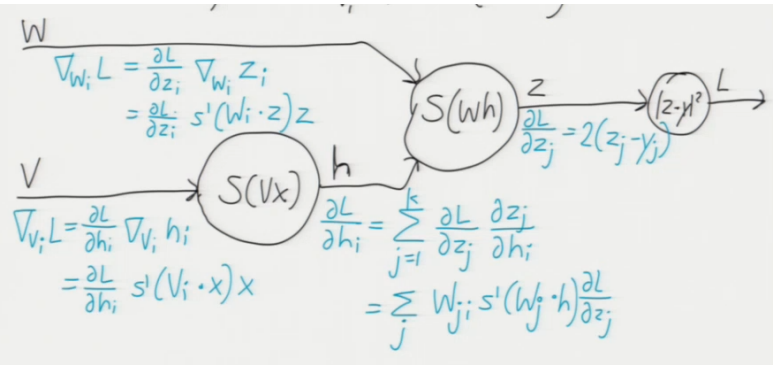

Figure 51. Arithmetic Expressions

And now, having figured out what these derivatives are, we have half of each derivative when you do backpropagate the other half. Again, we start at the end of the network, where we have the derivative of the loss function w.r.t any particular output neuron, and we propagate that down to here, and we also propagate back to W, and, we now also have the derivate of the loss function w.r.t each of our hidden units. And so we take that and we backpropagate that to V to get the derivatives of the weights in the matrix V.

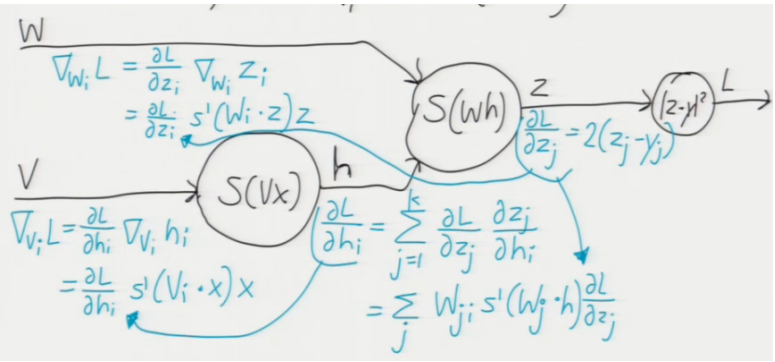

Figure 52. Arithmetic Expressions

And even though I've done this with just one layer of hidden units, the exact same pattern just repeats itself, if you say like 5 layers of hidden units. That doesn't change at all. So you can take this idea here, and just extend it by adding additional stages of hidden units, and it's all the same.

NEURAL NETWORK: A SIMPLER EXPLANATION

Let's say we're now looking at a neural network like this:

Backpropagation

Idea: update each of the weights in the network so that they cause the actual output to be closer to the target output, thereby minimizing the error for each output neuron and the network as a whole

Output Layer

Consider w_5, we wanna know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$. And by chain rule, we have

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \cdot \frac{\partial out_{o_1}}{\partial net_{o_1}} \cdot \frac{\partial net_{o_1}}{\partial w_5}$$

Let's start with $\frac{\partial E_{total}}{\partial out_{o_1}}$, i.e. we wanna find how much does the total error change w.r.t to the output.

$$E_{total} = \frac{1}{2}(target_{o_1} - out_{o_1})^2 + \frac{1}{2}(target_{o_2} - out_{o_2})^2$$

So we have

$$\begin{aligned}\frac{\partial \mathbf{E_{total}}}{\partial \mathbf{out_{o_1}}} &= 2 \cdot \frac{1}{2}(target_{o_1} - out_{o_1}) \cdot (-1) + 0 \\ &= -target_{o_1} + out_{o_1} \\ &= -(0.01 - 0.75136507) \\ &= 0.74136507\end{aligned}$$

As for $\frac{\partial out_{o_1}}{\partial net_{o_1}}$, we have

$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}}$$

And recall for sigmoid function $r(x) = \frac{1}{1+e^{-x}}$,

$$r'(x) = r(x)(1 - r(x)))$$

So,

$$\begin{aligned}\frac{\partial \mathbf{out_{o_1}}}{\partial \mathbf{net_{o_1}}} &= out_{o_1}(1 - out_{o_1}) \\ &= 0.75136505 \cdot (1 - 0.75136507) \\ &= 0.186815602\end{aligned}$$

Lastly, for $\frac{\partial net_{o_1}}{\partial w_5}$, we have

$$net_{o_1} = w_5 \cdot out_{h_1} + w_6 \cdot out_{h_2}$$

So,

$$\begin{aligned}\frac{\partial \mathbf{net_{o_1}}}{\partial \mathbf{w_5}} &= out_{h_1} \\ &= 0.593269992\end{aligned}$$

Putting them altogether, we have 0.082167041.

To decrease the error, subtract this value from the current weight:

$$\begin{aligned}\mathbf{w_5^+} &= w_5 - \alpha \cdot \frac{\partial E_{total}}{\partial w_5} \\ &= 0.4 - 0.5 * 0.082167041 \quad = 0.35891648\end{aligned}$$

Repeat for w_6^+, w_7^+ and w_8^+, we have:

$$w_6^+ = 0.408666186$$
$$w_7^+ = 0.511301270$$
$$w_8^+ = 0.561370121$$

Hidden Layer

Big picture, here's what we need to figure out:

$$net_{o_1} = w_5 * h1 + w_6 * h2 + b_2 * 1$$

$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}}$$

$$E_{total} = \frac{1}{2}(target_{o_1} - out_{o_1})^2 + \frac{1}{2}(target_{o_2} - out_{o_2})^2$$

CONVOLUTIONAL NEURAL NETWORK

Convolutional Neural Networks are very similar to ordinary Neural Networks:

- they are made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.
- The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.
- And they still have a loss function (e.g. SVM/Softmax) on teh last layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

So what does change? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Architecture Overview

Recall: ***Regular Neural Nets***. As we saw in the previous chapter, Neural Networks receive an input (a single vector), and transform it through a series of ***hidden layers***. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores.

Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3=120000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

3D volumes of neurons. Convolutional Neural Networks take advantage of the fact that the inputs consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**. (Note that the word ***depth*** here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully- connected manner. Moreover, the final input layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension.Here is a visualization:

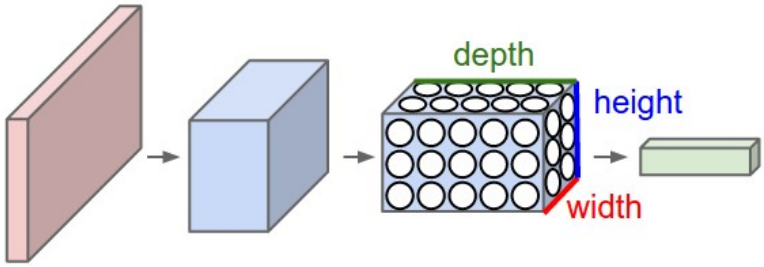

Figure 53. A regular 3-layer Neural Network.

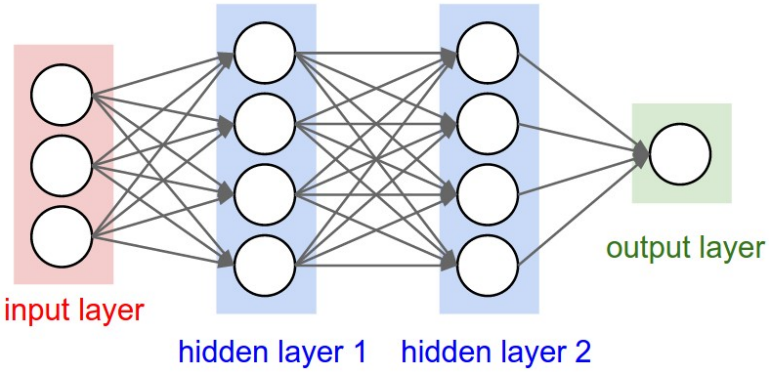

Figure 54. A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue Channels).

> A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.

Layer used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures:

1. Convolutional Layer
2. Pooling Layer
3. Fully-Connected Layer (exactly as seen in regular Neural Networks)

We will stack these layers to form a full ConvNet architecture.

Example Architecture Overview: We will go into details below, but a simple ConvNet for CIFAR-10 classification could have the architecture **[INPUT-CONV-RELU-POOL-FC]**. In more detail:

- **INPUT [32x32x3]** will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R, G, B.
- **CONV layer** will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as **[32x32x12]** if we decided to use 12 filters.
- **RELU layer** will apply an elementwise activation function, such as the $\mathbf{max}(\mathbf{0}, \mathbf{x})$ thresholding at zero. This leaves the size of the volume unchanged (**[32x32x12]**).
- **POOL layer** will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as **[16x16x12]**.
- **FC (i.e. fully-connected) layer** will compute the class scores, resulting in volume of size **[1x1x10]**, where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

In summary:

- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transform it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

We now describe the individual layers and the details of their hyperparameters and their connectivities.

Covolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.

Overview and intuition without brain stuff. Let's first discuss what the CONV layer computes without brain/neuron analogies. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of

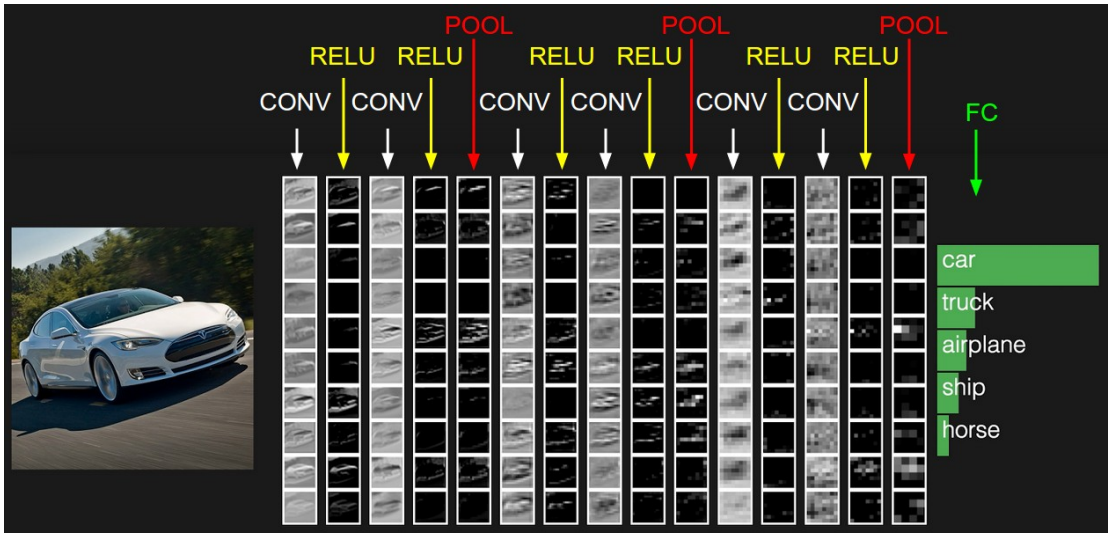

Figure 55. The activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's dificult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one. The full web-based demo is shown in the header of our website. The architecture shown here is a tiny VG net, which we will discuss later.

that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honey comb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

The brain view. If you're a fan of the brain/neuron analogies, every entry in the 3D output volume can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter). We now discuss the details of the neuron connectivities, their arrangement in space, and their parameter sharing scheme.

Local Connectivity. When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.

Example 1. For example, suppose that the input volume has size **[32x32x3]**, (e,g, an RGB CIFAR-10 image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3=75 weights (and +1 bias parameter). Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.

Example 2. Suppose an input volume has size **[16x16x20]**. Then using an example receptive field size of 3x3, every neuron in the Conv Layer would now have a total of 3*3*20=180 connections to the input volume. Notice that, again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

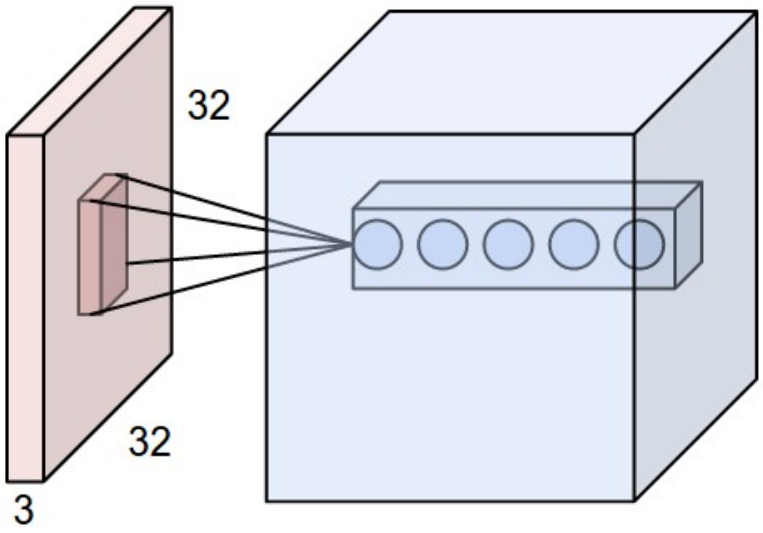

Figure 56. An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below.

Spatial arrangement. We have explained the connectivity of each neuron in the Conv Layer to the input volume, but we haven't yet discussed how many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume: the **depth**, **stride** and **zero-padding**. We discuss these next:

1. First, the **depth** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input.

CONVOLUTIONAL NEURAL NETWORK (UDEMY)

Translational Invariance

Pooling

Another import operation we'll need before we build the convolutional neural network is **downsampling**.

First, importing all the packages needed.

```
# Import the stuff
import os
import numpy as np
import theano
import theano.tensor as T
import matplotlib.pyplot as plt

from theano.tensor.nnet import conv2d
from theano.tensor.signal import pool

from scipy.io import loadmat
from sklearn.utils import shuffle

from datetime import datetime
```

Listing 1. Importing

The first method we're looking at reads the data.

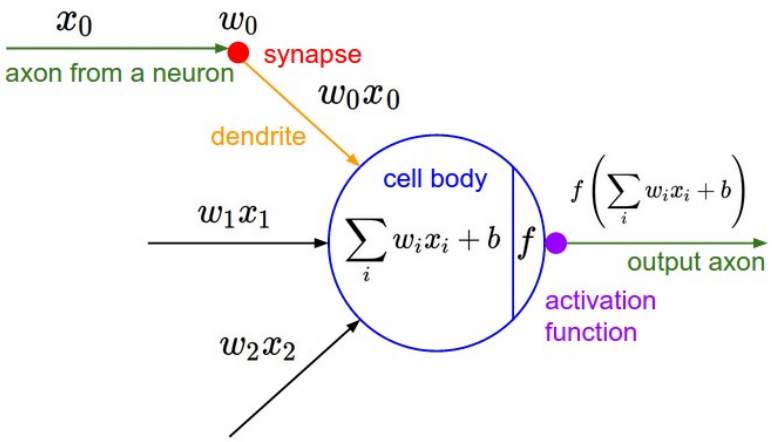

Figure 57. The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

```
def get_data():
    """Reads dataset

    Retrieves the dataset

    Returns:
        A tuple containing the training set (dict) and the testing set (dict)
    """
    if not os.path.exists('../large_files/House_Numbers/train_32x32.mat')
        print('Looking for ../large_files/House_Numbers/train_32x32.mat')
        print('You have not downloaded the data and/or not placed the files in the
            correct location.')
        print('Please get the data from: http://ufldl.stanford.edu/housenumbers')
        print('Place train_32x32.mat and test_32x32.mat in the folder large_files
            adjacent to the class folder')
        exit()

    train = loadmat('../large_files/House_Numbers/train_32x32.mat')
    test  = loadmat('../large_files/House_Numbers/test_32x32.mat')
    return train, test
```

Listing 2. Getting Data

This method rearranges the dimensions of data for Theano.

```
def rearrange(X):
    """Rearranges the matrix

    Rearranges the matrix into a different dimension. For example:

    Input is (32, 32, 3, N)
    Output is (N, 3, 32, 32)

    In this case, N is the number of color images (3 color channels) with
    a size of 32 x 32. Instead of using a matrix of size 32 x 32 x 3 x N,
    we rearrange it as N x 3 x 32 x 32

    Args:
        X: a numpy.ndarray matrix with 4 dimensions

    Returns:
        A rearranged numnpy.ndarray matrix
    """
    return (X.transpose(3, 2, 0, 1) / 255).astype(np.float32)
```

Listing 3. Rearranging Dimensions

The next method we're looking at initiates a kernel filter.

```
def init_filter(shape, poolsz):
    """Initializes a filter

    Takes in the shape and pooling size and generates a filter
    with the same dimensions as `shape`

    Args:
        shape: a tuple specifying the dimension
        poolsz: a tuple specigying the pooling size
    Returns:
        A numpy.ndarray containing the filter
    """
    w = np.random.randn(*shape) * np.sqrt(2.0 / np.prod(shape[]1:))
    return w.astype(np.float32)
```

Listing 4. Initing an Filter

Then we define our RELU function.

```
def relu(a):
    return a * (a > 0)
```

Listing 5. RELU

Next, we pipe Convolution and Pooling altogether.

```
def convpool(X, W, b, poolsize=(2,2)):
    conv_out = conv2d(input=X, filters=W)

    # downsample each feature map individually, using maxpooling
    pooled_out = pool.pool_2d(
        input=conv_out,
        ws=poolsize,
        ignore_border=True
    )

    # add the bias term. Since the bias is a vector (1D array), we first
    #
```

Listing 6. Convolution and Pooling

```


def main():
    # Overview of the network
    """
    Input: 32 x 32 x 3 x N

    """

    # Step 1
    # Load the data, transform as needed
    train, test = get_data()

    # Need to scale! Don't leave as 0..255
    # Y is a N x 1 matrix with values 1..10 (Matlab indexes by 1)
    # So flatten it and make it 0..9
    # Also need indicator matrix for cost calculation
    Xtrain = rearrange(train['X']) # (N, 3, 32, 32)
    Ytrain = test['y'].flatten() - 1
    del train
    Xtrain, Ytrain = shuffle(Xtrain, Ytrain)
```

```
22
23      Xtest = rearrange(test['X'])
24      Ytest = test['y'].flatten() - 1
25      del test
26
27      max_iter = 20
28      print_period = 10
29
30      # learning rate
31      lr = np.float32(1e-2)
32      # regularization = 0.01
33      # momentum
34      mu = np.float32(0.99)
35
36      N = Xtrain.shape[0]
37      batch_sz = 500
38      n_batches = N // batch_sz
39
40      # at our last stage, the fully connected NN layer has 500 hidden units
41      M = 500
42      # and 10 outputs
43      K = 10
44      poolsz = (2, 2)
```

Listing 7. Theano example

SIGNALS AND SYSTEMS

MIT Link YouTube Playlist

Lecture 1: Introduction

Mathematically, the continuous sinusoidal signal is expressed as:

$$x(t) = A\cos(\omega_0 t + \phi)$$

where A is the amplitude, ω is the frequency and ϕ is the phase.

Graphically, the sinusoidal signal has the form as show here:

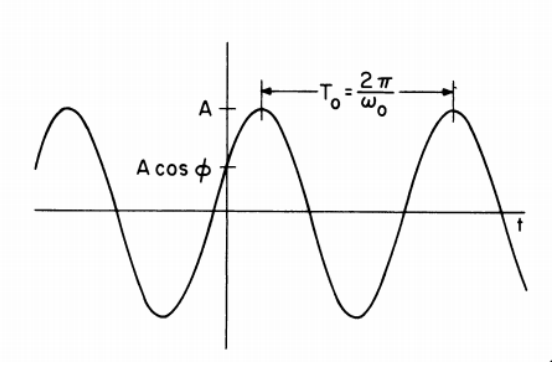

Figure 58. Continuous Sinusoidal Signal

One of the properties of sinusoidal signal is it's periodic, i.e., under an appropriate time shift, T_0, the signal replicates or repeats itself:

$$x(t) = x(t + T_0) \quad \text{period} \triangleq \text{small}$$

To demonstrate that, we substitute T_0 into the expression:

Proof.

$$\begin{aligned} x(t + T_0) &= A\cos[\omega_0(t + T_0) + \phi] \\ &= A\cos[\omega_0 t + \omega_0 T_0 + \phi] \end{aligned}$$

□

One of the things we know about sinusoidal functions is that if we change the argument by any integer multiple of 2π, then the function has the same value. So we can exploit that and if:

$$\omega_0 T_0 = 2\pi m$$

then the right hand side of the equation will be the same as the original signal. So with

$$T_0 = \frac{2\pi m}{\omega_0}$$

the signal becomes the same and we can say the period is $\frac{2\pi}{\omega_0}$.

In addition, another useful property of the sinusoidal signal is that a time shift of a sinusoid is equivalent to a phase change. If we put the sinusoidal signal under a time shift of small t_0, and expands it out, we get:

$$A\cos[\omega_0(t + t_0)] = A\cos[\omega_0 t + \omega_0 t_0]$$

and we can see it's equivalent to a change in phase. And not only is a time shift generating a phase change, but in fact, if we insert a phase change, there's always a t_0 which would correspond to an equivalent time shift, i.e., if we take $\omega_0 t_0$ and think of that as a change in phase, for any change in phase, we can solve this equation for time shift:

$$\omega_0 t_0 = \phi$$

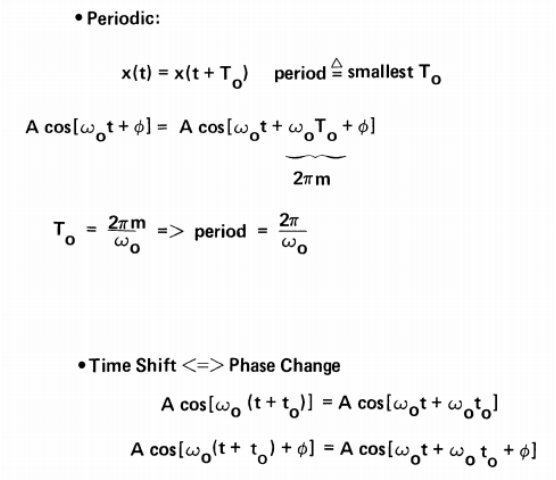

Figure 59. Continuous Sinusoidal Signal Relationships

And an important thing to recognize about this statement is that not only is a time shift generating a phase change, but, in fact, if we inserted a phase change, there is always a value of t_0 which would correspond to an equivalent time shift. Said another way, if we take $\omega_0 t_0$ and think of that as our change in phase, for any change in phase, we can solve this equation for a time shift, or conversely for any value of time shift, that represents an appropriate phase.

So a time shift corresponds to a phase change, and a phase change, likewise, corresponds to time shift. And so for example, if we look at the general sinusoidal signal that we saw previously, in effect, changing the phase corresponds to moving this signal in time one way or the other. For example, if we look at the sinusoidal signal with a phase equal to 0 that corresponds to locating the time origin at this peak. And I've indicated that on the following graph. (Look at Figure 60)

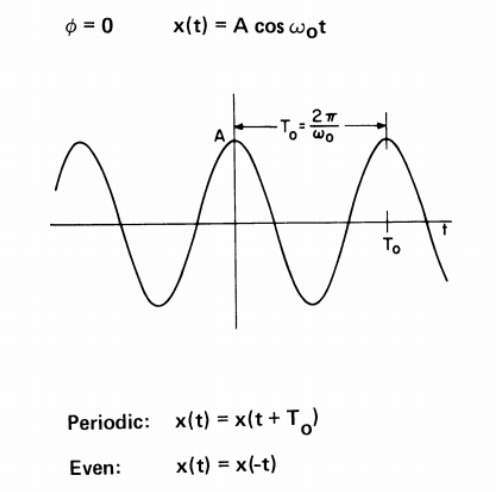

Figure 60. Illustration of the signal $Acosw_o t$ as an even signal.

So here we have illustrated a sinusoid with 0 phase, or a cosine with 0 phase, corresponding to taking our general picture and shifting it. Shifting it appropriately as I've indicated here. This, of course, still has the property that it's a periodic function, since we simply displaced it in time.

And by looking at the graph, what we see is that it has another very important property, a property referred to as even. And that's a property that we'll find useful, in general, to refer to in relation to signals. A signal is said to be even if, when we reflect it about the origin, it looks exactly the same. So it's symmetric about the origin.

And looking at this sinusoid, that, in fact, has that property. And mathematically, the statement that it's even is equivalent to the statement that if we replace the time argument by its negative, the function itself doesn't change.

Now this corresponded to a phase shift of 0 in our original cosine expression. If instead, we had chosen a phase shift of, let's say, $-\pi/2$, then instead of a cosinusoidal signal, what we would regenerate is a sinusoid with the appropriate phase. Or, said another way, if we take our original cosine and substitute in for the phase $-\pi/2$, then of course we have this mathematical expression. (Look at Figure 61)

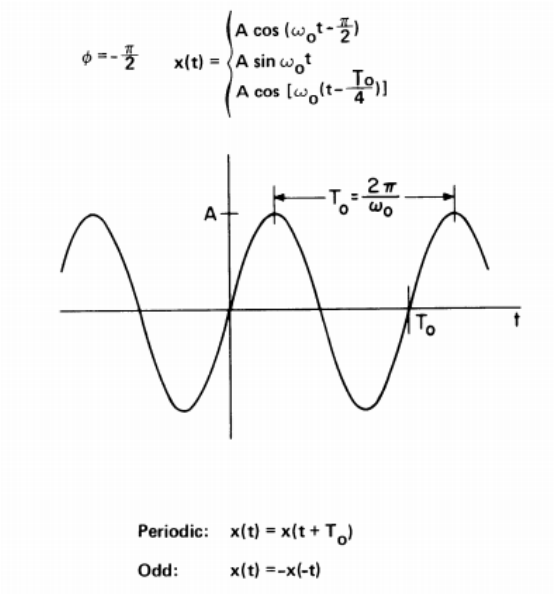

Figure 61. Illustration of the signal $Asinw_o t$ as an odd signal.

Using just straightforward trigonometric identities, we can express that alternately as $sin(\omega_0 * t)$. The frequency and amplitude, of course, haven't changed. And that, you can convince yourself, also is equivalent to shifting the cosine by an amount in time that I've indicated here, namely a quarter of a period.

So illustrated below (Still in Figure 61) is the graph now, when we have a phase of $-\pi/2$ in our cosine, which is a sinusoidal signal. Of course, it's still periodic. It's periodic with a period of $2\pi/\omega_0$ again, because all that we've done by introducing a phase change is introduced the time shift.

Now, when we look at the sinusoid in comparison with the cosine, namely with this particular choice of phase, this has a different symmetry, and that symmetry is referred to odd. What odd symmetry means, graphically, is that when we flip the signal about the time origin, we also multiply it by a minus sign. So that's, in effect, anti-symmetric. It's not the mirror image, but it's the mirror image flipped over. And we'll find many occasions, not only to refer to signals more general than sinusoidal signals, as even in some cases and odd in other cases. And in general, mathematically, an odd signal is one which satisfies the algebraic expression, $x(t)$. When you replace t by its negative, is equal to $-x(-t)$. So replacing the argument by its negative corresponds to an algebraic sign reversal.

OK. So this is the class of continuous-time sinusoids. We'll have a little more to say about it later. But I'd

now like to turn to discrete-time sinusoids. What we'll see is that discrete-time sinusoids are very much like continuous-time ones, but also with some very important differences. And we want to focus, not only on the similarities, but also on the differences.

Well, let's begin with the mathematical expression. A discrete-time sinusoidal signal, mathematically, is as I've indicated here, $Acos(\omega_0 n + \phi)$. And just as in the continuous-time case, the parameter A is what we'll refer to as the amplitude, $omega_0$ as the frequency, and ϕ as the phase. (Look at Figure 62)

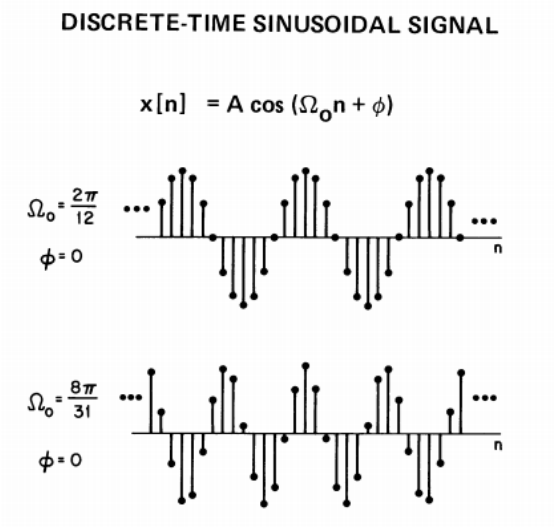

Figure 62. Illustration of discrete-time sinusoidal signals.

And I've illustrated here several discrete-time sinusoidal signals. And they kind of look similar. In fact, if you track what you might think of as the envelope, it looks very much like what a continuous-time sinusoid might look like. But keep in mind that the independent variable, in this case, n, is an integer variable. And so the sequence only takes on values at integer values of the argument. And we'll see that has a very important implication, and we'll see that shortly.

Now, one of the issues that we addressed in the continuous-time case was periodicity. And I want to return to that shortly, because that is one of the areas where there is an important distinction. Let's first, though, examine the statement similar to the one that we examined for continuous time, namely the relationship between a time shift and a phase change.

Now, in continuous time, of course, we saw that a time shift corresponds to a phase change, and vice versa. Let's first look at the relationship between shifting time and generating a change in phase. In particular for discrete time, if I implement a time shift that generates a phase change, and we can see that easily by simply inserting a time shift, $n + n_0$. And if we expand out this argument, we have $\omega_0 n + \omega_0 n_0$. (Look at Figure 63)

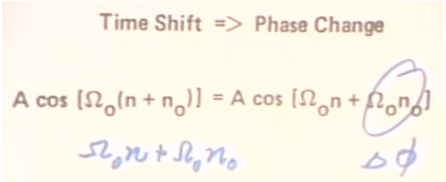

Figure 63. Relationship between a time shift and a phase change for discrete-time sinusoidal signals. In discrete time, a time shift always implies a phase change.

And so I've done that on the right-hand side of the equation here. And the $\omega_0 n_0$, then, simply corresponds to a change in phase. So clearly, a shift in time generates a change in phase.

And for example, if we take a particular sinusoidal signal, let's say we take the cosine signal at a particular

frequency, and with a phase equal to 0, a sequence that we might generate is one that I've illustrated here. So what I'm illustrating here is the cosine signal with 0 phase. And it has a particular behavior to it, which will depend somewhat on the frequency. (Look at Figure 64)

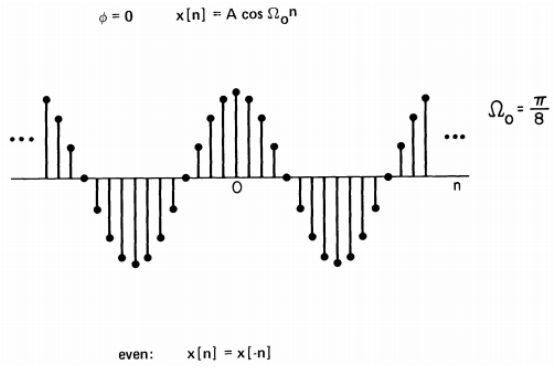

Figure 64. Relationship between a time shift and a phase change for discrete-time sinusoidal signals. In discrete time, a time shift always implies a phase change.

If I now take this same sequence and shift it so that the time origin is shifted a quarter of a period away, then you can convince yourself– and it's straightforward to work out– that that time shift corresponds to a phase shift of $\pi/2$. So in that case, with the cosine with a phase of $-\pi/2$, that will correspond to the expression that I have here.

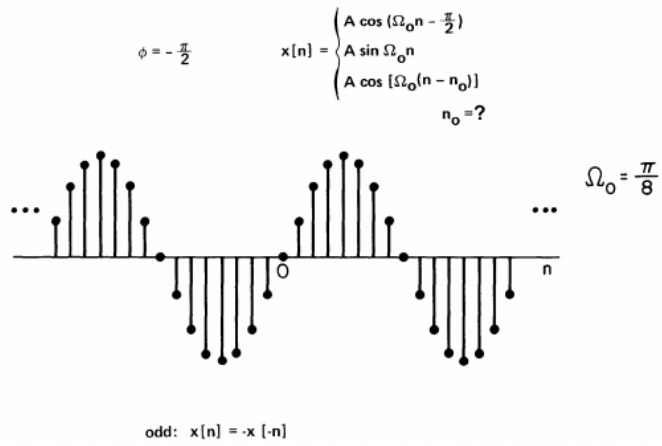

Figure 65. The sequence $A sin \omega_0 n$ illustrating the antisymmetric property of an odd sequence.

We could alternately write that, using again a trigonometric identity, as a sine function. And that, I've stated, is equivalent to a time shift. Namely, this shift of $\pi/2$ is equal to a certain time shift, and the time shift for this particular example is a quarter of a period.

So here, we have the sinusoid. Previously we had the cosine. The cosine was exactly the same sequence, but with the origin located here. And in fact, that's exactly the way we drew this graph. Namely, we just simply took the same values and changed the time origin.

Now, looking at this sequence, which is the sinusoidal sequence, the phase of $-\pi/2$, that has a certain symmetry. And in fact, what we see is that it has an odd symmetry, just as in the continuous-time case. Namely, if we take that sequence, flip it about the axis, and flip it over in sign, that we get the same sequence back again. Whereas with 0 phase corresponding to the cosine that I showed previously, that has an even symmetry. Namely, if I flip it about the time origin and don't do a sign reversal, then the sequence is maintained.

So here, we have an odd symmetry, expressed mathematically as I've indicated. Namely, replacing the independent variable by its negative attaches a negative sign to the whole sequence. Whereas in the

previous case, what we have is 0 phase and an even symmetry. And that's expressed mathematically as $x[n] = x[-n]$.

Now, one of the things I've said so far about discrete-time sinusoids is that a time shift corresponds to a phase change. And we can then ask whether the reverse statement is also true, and we knew that the reverse statement was true in continuous time. Specifically, is it true that a phase change always corresponds to a time shift? Now, we know that that is true, namely, that this statement works both ways in continuous time. Does it in discrete time?

Well, the answer, somewhat interestingly or surprisingly until you sit down and think about it, is no. It is not necessarily true in discrete time that any phase change can be interpreted as a simple time shift of the sequence. And let me just indicate what the problem is.

If we look at the relationship between the left side and the right side of this equation, expanding this out as we did previously, we have that $\omega_0 n + \omega_0 n_0$ must correspond to $\omega_0 n + \phi$. And so $\omega_0 n_0$ must correspond to the phase change. Now, what you can see pretty clearly is that depending on the relationship between ϕ and ω_0, n_0 may or may not come out to be an integer.

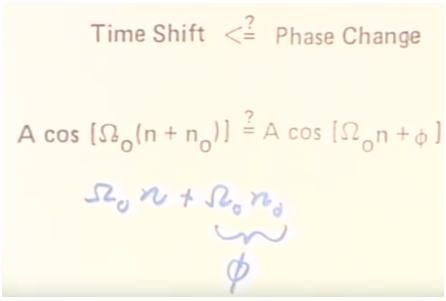

Figure 66. For a discrete-time sinusoidal sequence a time shift always implies a change in phase, but a change in phase might not imply a time shift.

Now, in continuous time, the amount of time shift did not have to be an integer amount. In discrete time, when we talk about a time shift, the amount of time shift– obviously, because of the nature of discrete time signals– must be an integer. So the phase changes related to time shifts must satisfy this particular relationship. Namely, that $\omega_0 n_0$, where n_0 is an integer, is equal to the change in phase.

OK. Now, that's one distinction between continuous time and discrete time. Let's now focus on another one, namely the issue of periodicity. And what we'll see is that again, whereas in continuous time, all continuous-time sinusoids are periodic, in the discrete-time case that is not necessarily true.

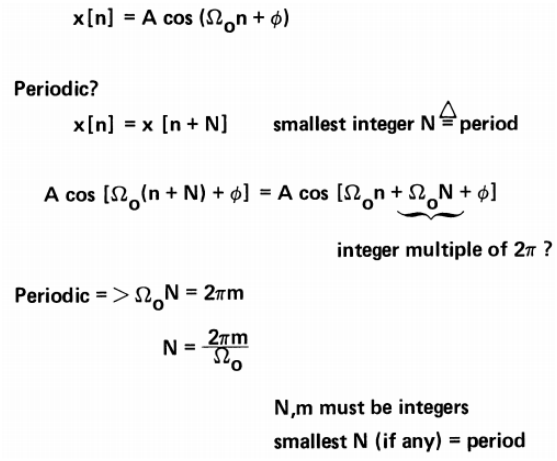

Figure 67. The requirement on ω_0 for a discrete-time sinusoidal signal to be periodic.

To explore that a little more carefully, let's look at the expression, again, for a general sinusoidal signal

with an arbitrary amplitude, frequency, and phase. And for this to be periodic, what we require is that there be some value, N, under which, when we shift the sequence by that amount, we get the same sequence back again. And the smallest-value N is what we've defined as the period.

Now, when we try that on a sinusoid, we of course substitute in for n, $n+N$. And when we expand out the argument here, we'll get the argument that I have on the right-hand side. And in order for this to repeat, in other words, in order for us to discard this term, $\omega_0 N$, where N is the period, must be an integer multiple of 2π. And in that case, it's periodic as long as $\omega_0 N$, N being the period, is 2π times an integer. Just simply dividing this out, we have N, the period, is $2\pi m/\omega_0$.

Well, you could say, OK what's the big deal? Whatever N happens to come out to be when we do that little bit of algebra, that's the period. But in fact, N, or $2\pi m/\omega_0$, may not ever come out to be an integer. Or it may not come out to be the one that you thought it might.

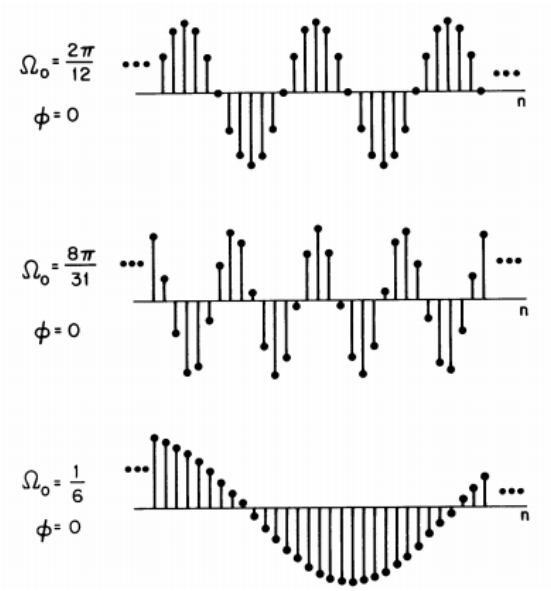

Figure 68. Several sinusoidal sequences illustrating the issue of periodicity.

For example, let's look at some particular sinusoidal signals. Let's see. We have the first one here, which is a sinusoid, as I've shown. And it has a frequency, what I've referred to as the frequency, $\omega_0 = 2\pi/12$. And what we'd like to look at is $2\pi/\omega_0$, then find an integer to multiply that by in order to get another integer.

Let's just try that here. If we look at $2\pi/\omega_0$, $2\pi/\omega_0$, for this case, is equal to 12. Well, that's fine. 12 is an integer. So what that says is that this sinusoidal signal is periodic. And in fact, it's periodic with a period of 12.

Let's look at the next one. The next one, we would have $2\pi/\omega_0$ again. And that's equal to $31/4$. So what that says is that the period is $31/4$. But wait a minute. $31/4$ isn't an integer. We have to multiply that by an integer to get another integer. Well, we'd multiply that by 4, so $(2\pi/\omega_0)$ times 4 is 31, 31 is an integer. And so what that says is this is periodic, not with a period of $2\pi/\omega_0$, but with a period of $(2\pi/\omega_0)$ times 4, namely with a period of 31.

Finally, let's take the example where *omega*$_0$ is equal to $1/6$, as I've shown here. That actually looks, if you track it with your eye, like it's periodic. $2\pi/\omega_0$, in that case, is equal to 12 pi. Well, what integer can I multiply 12 pi by and get another integer? The answer is none, because π is an irrational number.

So in fact, what that says is that if you look at this sinusoidal signal, it's not periodic at all, even though you might fool yourself into thinking it is simply because the envelope looks periodic. Namely, the continuous-time equivalent of this is periodic, the discrete-time sequence is not.

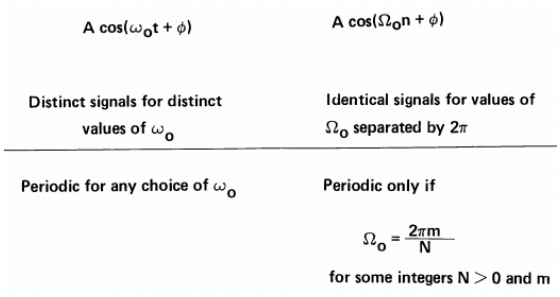

Figure 69. Some important distinctions between continuous-time and discrete-time sinusoidal signals.

OK. Well, we've seen, then, some important distinctions between continuous-time sinusoidal signals and discrete-time sinusoidal signals. The first one is the fact that in the continuous-time case, a time shift and phase change are always equivalent. Whereas in the discrete-time case, in effect, it works one way but not the other way.

We've also seen that for a continuous-time signal, the continuous-time signal is always periodic, whereas the discrete-time signal is not necessarily. In particular, for the continuous-time case, if we have a general expression for the sinusoidal signal that I've indicated here, that's periodic for any choice of ω_0. Whereas in the discrete-time case, it's periodic only if $2\pi/\omega_0$ can be multiplied by an integer to get another integer.

Now, another important and, as it turns out, useful distinction between the continuous-time and discrete-time case is the fact that in the discrete-time case, as we vary what I've called the frequency ω_0, we only see distinct signals as ω_0 varies over a 2π interval. And if we let ω_0 vary outside the range of, let's say, $-\pi$ to π, or 0 to 2π, we'll see the same sequences all over again, even though at first glance, the mathematical expression might look different.

So in the discrete-time case, this class of signals is identical for values of ω_0 separated by 2π, whereas in the continuous-time case, that is not true. In particular, if I consider these sinusoidal continuous-time signals, as I vary ω_0, what will happen is that I will always see different sinusoidal signals. Namely, these won't be equal.

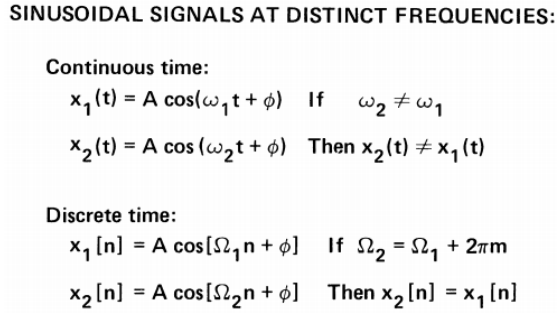

Figure 70. Continuous-time sinusoidal signals are distinct at distinct frequencies. Discrete-time sinusoidal signals are distinct only over a frequency range of 2π

And in effect, we can justify that statement algebraically. And I won't take the time to do it carefully. But let's look, first of all, at the discrete-time case. And the statement that I'm making is that if I have two discrete-time sinusoidal signals at two different frequencies, and if these frequencies are separated by an integer multiple of 2π– namely if ω_2 is equal to $\omega_1 + 2\pi$ times an integer m– when I substitute this into this expression, because of the fact that n is also an integer, I'll have $m * n$ as an integer multiple of 2π. And that term, of course, will disappear because of the periodicity of the sinusoid, and these two sequences will be equal.

On the other hand in the continuous-time case, since t is not restricted to be an integer variable, for different values of ω_1 and ω_2, these sinusoidal signals will always be different.

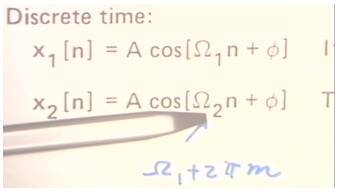

Figure 71

OK. Now, many of the issues that I've raised so far, in relation to sinusoidal signals, are elaborated on in more detail in the text. And of course, you'll have an opportunity to exercise some of this as you work through the video course manual. Let me stress that sinusoidal signals will play an extremely important role for us as building blocks for general signals and descriptions of systems, and leads to the whole concept Fourier analysis, which is very heavily exploited throughout the course.

What I'd now like to turn to is another class of important building blocks. And in fact, we'll see that under certain conditions, these relate strongly to sinusoidal signals, namely the class of real and complex exponentials.

Let me begin, first of all, with the real exponential, and in particular, in the continuous-time case. A real continuous-time exponential is mathematically expressed, as I indicate here, $x(t) = Ce^{at}$, where for the real exponential, C and a are real numbers. And that's what we mean by the real exponential. Shortly, we'll also consider complex exponentials, where these numbers can then become complex.

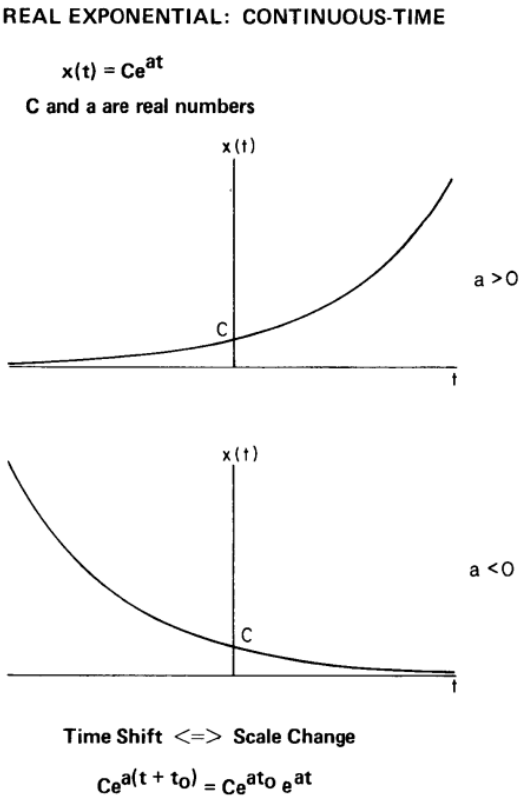

Figure 72. Illustration of continuous-time real exponential signals.

So this is an exponential function. And for example, if the parameter a is positive, that means that we have a growing exponential function. If the parameter a is negative, then that means that we have a decaying exponential function.

Now, somewhat as an aside, it's kind of interesting to note that for exponentials, a time shift corresponds to a scale change, which is somewhat different than what happens with sinusoids. In the sinusoidal case, we saw that a time shift corresponded to a phase change. With the real exponential, a time shift, as it turns out, corresponds to simply changing the scale.

There's nothing particularly crucial or exciting about that. And in fact, perhaps stressing it is a little misleading. For general functions, of course, about all that you can say about what happens when you implement a time shift is that it implements a time shift.

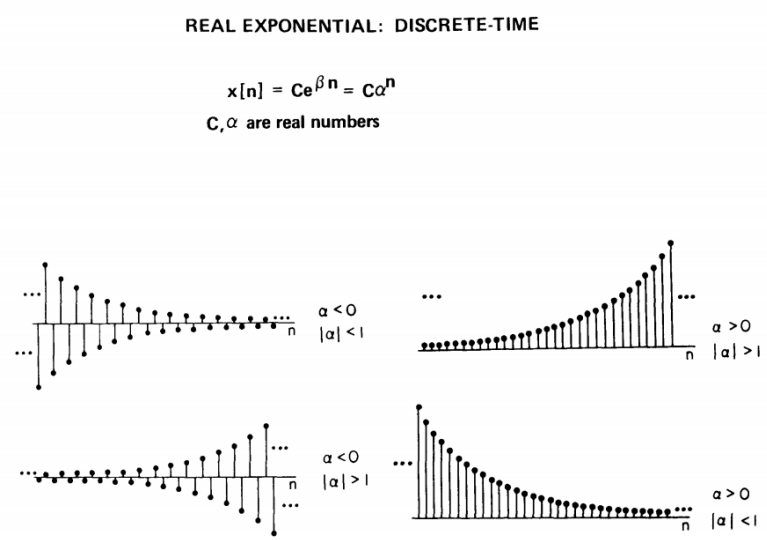

Figure 73. Illustration of discrete-time real exponential sequences.

OK. So here's the real exponential. Just Ce^{at}. Let's look at the real exponential, now, in the discrete-time case. And in the discrete-time case, we have several alternate ways of expressing it. We can express the real exponential in the form $Ce^{\beta n}$, or as we'll find more convenient, in part for a reason at I'll indicate shortly, we can rewrite this as $C\alpha^n$, where of course, $\alpha = e^{\beta}$. More typically in the discrete-time case, we'll express the exponential as $C\alpha^n$.

So for example, this becomes, essentially, a geometric series or progression as n continues for certain values of alpha. Here for example, we have for α greater than 0, first of all on the top, the case where the magnitude of α is greater than 1, so that the sequence is exponentially or geometrically growing. On the bottom, again with α positive, but now with its magnitude less than 1, we have a geometric progression that is exponentially or geometrically decaying.

OK. So this, in both of these cases, is with α greater than 0. Now the function that we're talking about is α^n. And of course, what you can see is that if α is negative instead of positive, then when n is even, that minus sign is going to disappear. When n is odd, there will be a minus sign. And so for α negative, the sequence is going to alternate positive and negative values.

So for example, here we have α negative, with its magnitude less than 1. And you can see that, again, its envelope decays geometrically, and the values alternate in sign. And here we have the magnitude of α greater than 1, with α negative. Again, they alternate in sign, and of course it's growing geometrically.

Now, if you think about α positive and go back to the expression that I have at the top, namely $C\alpha^n$. With α positive, you can see a straightforward relationship between α and β. Namely, β is the natural logarithm of α.

Something to think about is what happens if α is negative? Which is, of course, a very important and useful class of real discrete-time exponentials also. Well, it turns out that with α negative, if you try to express it as $Ce^{\beta n}$, then β comes out to be an imaginary number. And that is one, but not the only reason why, in the discrete-time case, it's often most convenient to phrase real exponentials in the form α^n, rather than $e^{\beta n}$. In other words, to express them in this form rather than in this form.

Those are real exponentials, continuous-time and discrete-time. Now let's look at the continuous-time complex exponential. And what I mean by a complex exponential, again, is an exponential of the form Ce^{at}. But in this case, we allow the parameters C and a to be complex numbers.

COMPLEX EXPONENTIAL: CONTINUOUS-TIME

$$x(t) = Ce^{at}$$

C and a are complex numbers

$$C = |C|\, e^{j\theta}$$

$$a = r + j\omega_o$$

$$x(t) = |C|\, e^{j\theta}\, e^{(r + j\omega_o)t}$$

$$= |C|\, e^{rt}\, e^{j(\omega_o t + \theta)}$$

Euler's Relation: $\cos(\omega_o t + \theta) + j \sin(\omega_o t + \theta) = e^{j(\omega_o t + \theta)}$

$$x(t) = |C|\, e^{rt} \cos(\omega_o t + \theta) + j\, |C|\, e^{rt} \sin(\omega_o t + \theta)$$

Figure 74. Continuous-time complex exponential signals and their relationship to sinusoidal signals

And let's just track this through algebraically. If C and a are complex numbers, let's write C in polar form, so it has a magnitude and an angle. Let's write a in rectangular form, so it has a real part and an imaginary part. And when we substitute these two in here, combine some things together– well actually, I haven't combined yet. I have this ($|C|e^{j\theta}$) for the amplitude factor, and this ($e^{(r+j\omega_0 t)}$) for the exponential factor. I can now pull out of this the term corresponding to e^{rt}, and combine the imaginary parts together. And I come down to the expression that I have here.

So following this further, an exponential of this form, $e^{j\omega}$ or $e^{j\phi}$, using Euler's relation, can be expressed as the sum of a cosine plus j times a sine. And so that corresponds to this factor ($e^{j(\omega_0 t+\theta)}$). And then there is this time-varying amplitude factor ($|C|e^{rt}$) on top of it.

Finally putting those together, we end up with the expression that I show on the bottom. And what this corresponds to are two sinusoidal signals, 90 degrees out of phase, as indicated by the fact that there's a cosine and a sine. So there's a real part and an imaginary part, with sinusoidal components 90 degrees out of phase, and a time-varying amplitude factor, which is a real exponential. So it's a sinusoid multiplied by a real exponential in both the real part and the imaginary part.

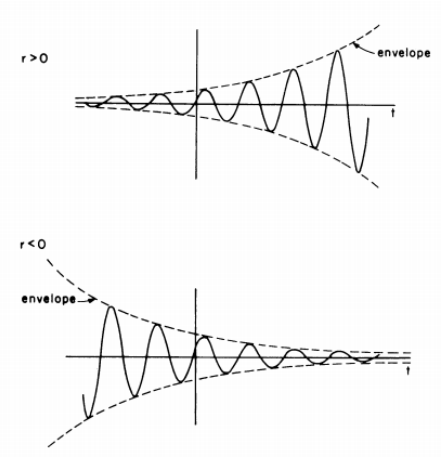

Figure 75. Sinusoidal signals with exponentially growing and exponentially decaying envelopes

And let's just see what one of those terms might look like. What I've indicated at the top is a sinusoidal signal with a time-varying exponential envelope, or an envelope which is a real exponential, and in particular which is growing, namely with r greater than 0. And on the bottom, I've indicated the same thing with r less than 0.

And this kind of sinusoidal signal, by the way, is typically referred to as a damped sinusoid. So with r negative, what we have in the real and imaginary parts are damped sinusoids. And the sinusoidal components of that are 90 degrees out of phase, in the real part and in the imaginary part.

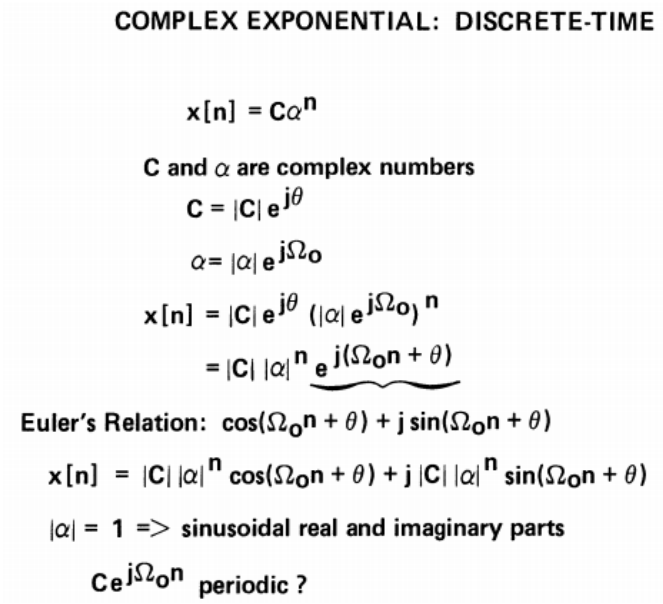

Figure 76. Discrete-time complex exponential signals and their relationship to sinusoidal signals.

OK. Now, in the discrete-time case, we have more or less the same kind of outcome. In particular we'll make reference to our complex exponentials in the discrete-time case. The expression for the complex exponential looks very much like the expression for the real exponential, except that now we have complex factors. So C and α are complex numbers.

And again, if we track through the algebra, and get to a point where we have a real exponential multiplied by a factor which is a purely imaginary exponential, apply Euler's relationship to this, we then finally come down to a sequence, which has a real exponential amplitude multiplying one sinusoid in the real part. And in the imaginary part, exactly the same kind of exponential multiplying a sinusoid that's 90 degrees out of phase from that.

And so if we look at what one of these factors might look like (Figure 77), it's what we would expect given the analogy with the continuous-time case. Namely, it's a sinusoidal sequence with a real exponential envelope. In the case where α is positive, then it's a growing envelope. In the case where α is negative– I'm sorry– where the magnitude of α is greater than 1, it's a growing exponential envelope. Where the magnitude of α is less than 1, it's a decaying exponential envelope.

And so I've illustrated that here. Here we have the magnitude of α greater than 1. And here we have the magnitude of α less than 1. In both cases, sinusoidal sequences underneath the envelope, and then an envelope that is dictated by what the magnitude of α is.

OK. Now, in the discrete-time case, then, we have results similar to the continuous-time case. Namely, components ($x[n] = |C||\alpha|^n cos(\omega n + \theta) + j|C||\alpha|^n sin(\omega|0n + \theta)$) in a real and imaginary part that have a real exponential factor times a sinusoid. Of course, if the magnitude of α is equal to 1, then this factor ($|\alpha|^n$) disappears, or is equal to 1. And this factor ($|\alpha|^n$) is equal to 1. And so we have sinusoids in both the real and imaginary parts.

Now, one can ask whether, in general, the complex exponential with the magnitude of α equal to 1 is

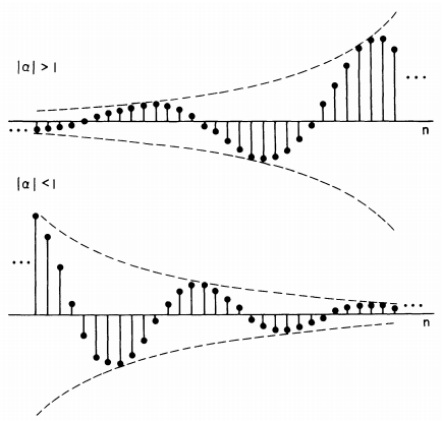

Figure 77. Sinusoidal sequences with geometrically growing and geometrically decaying envelops

periodic or not periodic. And the clue to that can be inferred by examining this expression. In particular, in the discrete-time case with the magnitude of α equal to 1, we have pure sinusoids in the real part and the imaginary part. And in fact, in a continuous-time case with r equal to 0, we have sinusoids in the real part and the imaginary part.

In a continuous-time case when we have a pure complex exponential, so that the terms aren't exponentially growing or decaying, those exponentials are always periodic. Because, of course, the real and imaginary sinusoidal components are periodic. In the discrete-time case, we know that the sinusoids may or may not be periodic, depending on the value of ω_0. And so in fact, in the discrete-time case, the exponential $e^{j\omega_0 n}$, that I've indicated here, may or may not be periodic depending on what the value of ω_0 is.

OK. Now, to summarize, in this lecture I've introduced and discussed a number of important basic signals. In particular, sinusoids and real and complex exponentials. One of the important outcomes of the discussion, emphasized further in the text, is that there are some very important similarities between them. But there are also some very important differences. And these differences will surface when we exploit sinusoids and complex exponentials as basic building blocks for more general continuous-time and discrete-time signals.

In the next lecture, what I'll discuss are some other very important building blocks, namely, what are referred to as step signals and impulse signals. And those, together with the sinusoidal signals and exponentials as we've talked about today, will really form the cornerstone for, essentially, all of the signal and system analysis that we'll be dealing with for the remainder of course. Thank you.

Four Ways to Think About Matrix Multiplication

There are four ways to think about and to implement standard matrix multiplication.

All four methods give exactly the same result, but they provide a different perspective on what matrix multiplication means and how it works. It's really useful to understand all four of these perspectives, because they provide distinct insights into matrix computations in different contexts and for different problems.

Dot-Products

So first of all, you can think about matrix multiplication as an ordered collection of dot-products. I call this the "**element perspective**" because you think about matrix multiplication one element at a time. This is the main way and sometimes, unfortunately in my opinion, the only way that teachers introduce matrix multiplication.

So, **each element in this matrix is the dot product between rows of this matrix and columns of this matrix**.

The idea is to think about this left matrix as comprising rows and this right matrix as comprising columns. Then, each element of the dot product comes from the corresponding row and the corresponding column.

Here's an example.

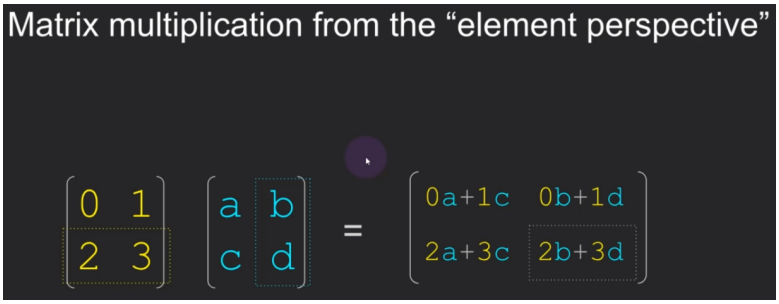

Figure 78. Multiplication 01

So, this first element in the product matrix is defined as the dot product between this – the first row and the first column – and it's the first row in the first column because this is element $(1,1)$.

Now we can think about this element here. This is the second row and the first column in this product matrix. So this element is defined as the dot product between the second row in the left matrix and the first column of the right matrix. Then we go here. This again is the first row and the second column of the product matrix, so this is the dot product between the first row of the left matrix and the second column of the right matrix.

And finally, we get to the last element, which is second row and second column. That's the dot product between the second row and the second column of these matrices.

So what I've done here is build up the product matrix one individual element at a time.

That's why this is called the **element perspective**, because you are thinking about this matrix as a collection of individual elements.

This is one way to conceptualize standard matrix multiplication. The second way to think about matrix multiplication is to build the product matrix one "layer" at a time.

Layer Perspective

I call this the "**layer perspective**." Each layer in the matrix is the same size as the product matrix, but it's a rank-1 matrix.

I'll have a separate video on rank in a later section, but for now you can think of a rank-1 matrix as containing only 1 column's worth of information. All the other columns are linearly dependent on the first column. So the idea of the layer perspective is to compute and sum a series of rank-1 matrices, and the end result is going to be the product matrix.

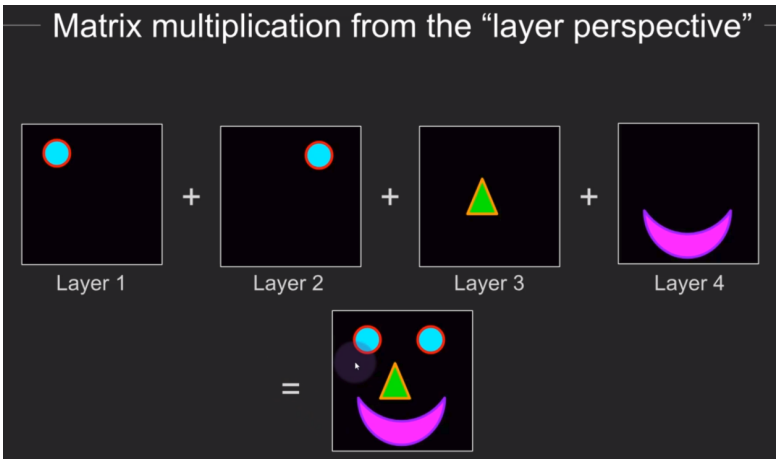

Figure 79. Analogy

Here's an analogy that you can use to think about. Imagine someone is creating a picture in Photoshop or Inkscape by layering transparent shapes on top of each other.

Each shape contains a different part of the image. And when you put them all together, when you sum all of these layers together, you get the final result. So this result is not present in any individual layer, but each of these layers provides a different piece. And then you put them all together and these layers are all the same size as the result.

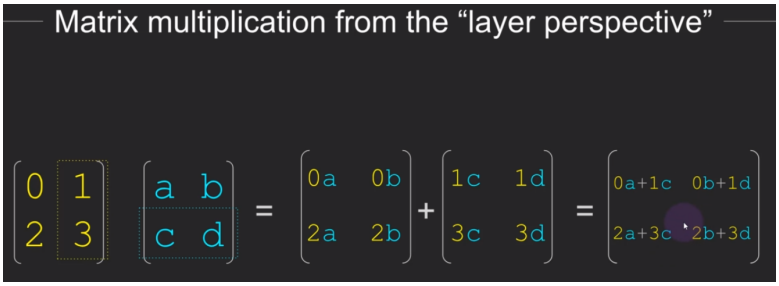

Figure 80. Layer Perspective

So the mechanism of computing the layer perspective is to think about this **left matrix** as **comprising columns**. Previously we thought about this as rows. Now we think about this as columns, and we think about the **right matrix** as **comprising rows**. Then you compute all of the corresponding outer products. In this case it's only 2, so that's going to give us two matrices, two **outer products**.

So this first matrix here is the outer product of the first column in the last matrix and the first row in the right matrix. And you can see that the columns are linearly dependent on each other, as are the rows. So this is the outer product.

And now we get to this one. This is also an outer product, but it's from the second column and the second row. And if these were larger matrices, you would continue this to have more layers. And then what you do is add these two matrices, element by element, and that gives you this result.

And I encourage you to pause the video and confirm with your notes that this resulting matrix is exactly the same as the previous product matrix from the element perspective, using dot products.

And now we use outer products to compute the **layer perspective**.

Another way to think about this is that each of these rank-1 matrices is like a single color, and the matrix is the rainbow. So we put all these single colors together and then we get this beautiful rainbow.

This is an important and elegant idea, and it forms the basis for the **singular value decomposition**, which you will learn about in a later section of this course.

Column Perspective

The third way to think about standard matrix multiplication is what I call the "**column perspective**."

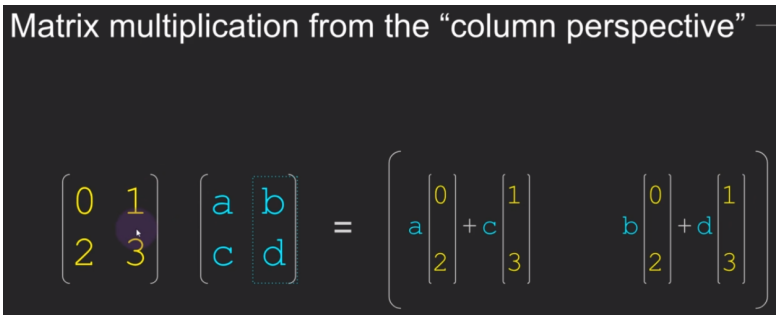

Figure 81. Column Perspective

Here you think about the product matrix as being a **linear weighted combination of the columns of the left matrix**, where the weights – or scalars to combine these columns – come from this right matrix.

So let's see what that would look like. The left column – I should say, the first column of the product matrix – is the sum of the two columns of the left matrix, so $[0,2]^T$, plus the column $[1,3]^T$, and those columns are not just added; they are weighted and then they're added. And the weights come from the right matrix, so $[a,c]^T$. And now the second column in the product matrix again it comes from exactly the same columns of the left matrix.

But now we're combining them by weighting them according to the second column in the right matrix so it's b times the first column of the left matrix plus b times the second column in the right matrix. And of course this would expand for however many columns you have in these matrices.

So you'll notice that in the element perspective, I was building up the product matrix one element at a time; in the layer perspective, I was building up the product matrix one layer at a time. And now I'm building up the product matrix one column at a time.

This interpretation – this column perspective – turns out to be really useful in statistics, because the columns of the left matrix will contain a set of regressors, which is a simplified model of a data set, and the right matrix will contain the coefficients. And the idea in statistics is that the coefficients encode the importance of each vector, and the goal of model fitting in statistics is to find the best coefficients, such that the weighted combination of regressors best matches the data.

So you will see this column perspective coming up again in a later section of this course about linear least squares modeling and statistics.

Row Perspective

I'm sure you can guess what the fourth perspective of matrix multiplication will be: of course it's the **row perspective**. Again, the result will be exactly the same as the previous three methods I showed you; this is just a different way of thinking about the same operation. So now we build up the product matrix one row at a time.

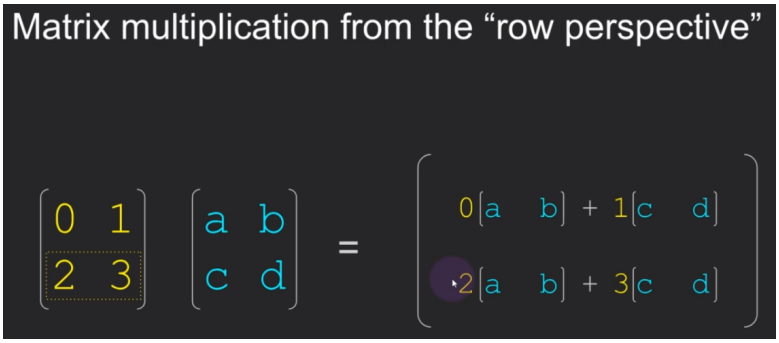

Figure 82. Row Perspective

So let’s see how this works. So the first row in the product matrix comes from a sum of both rows of the right matrix, except we’re not just summing the rows of this right matrix, we are weighting them and taking the weighted combination of these rows. The weightings come from the elements of the first row in the left matrix. And same story for the second row. We take again the sum of all the rows of the second matrix, but now instead of being weighted by the first row the left matrix, they’re weighted by the second row of the left matrix.

OK. And of course all of these examples that I’ve been showing are for two by two matrices but that’s really just in the interest of brevity. The same perspectives apply for any size matrices, where the matrix multiplication is valid.

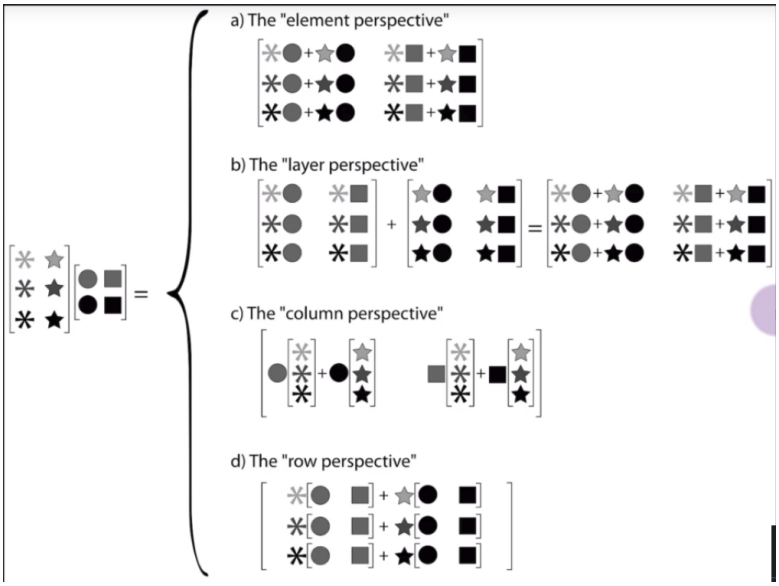

Figure 83. Summary

So let’s recap. In this video I showed four different ways of thinking about, and also four different ways of implementing, matrix multiplication between two matrices.

I know it’s a lot to take in, and matrix multiplication is a bit of a weird thing at first, but it is very important. I like to teach these four ways to think about the same operation, because this increases your understanding in the flexibility of this really fundamental operation. And the more comfortable you are with matrix multiplication, the easier it will be to learn more advanced topics in linear algebra.

Matrix Rank

Rank: Concepts, Terms and Applications

The rank of a matrix is a single number that provides insight into the amount of information that is contained in the matrix. That's different from the dimensionality of the matrix. And in this video I'm going to discuss the ideas and the uses of matrix rank, so you can have a sense of why it's such an important number in linear algebra.

Indeed, the rank of a matrix is something that comes up all the time in applied linear algebra. So this is not just some esoteric, abstract concept that you have to learn in class and never use again. So let's get started.

Figure 84. Rank

We'll start with terminology. The rank of a matrix is typically indicated by the lowercase letter *r*. Or sometimes by the word **rank** and then the matrix in parentheses. The **rank** is a single number that characterizes the entire matrix. It applies to matrix of any size; the matrix can be square or rectangular.

Even vectors have a rank. Rank is a non-negative integer meaning it can be 0, 1, 2, 3, and so on. There is no such thing as negative rank, and there is no such thing as a rank of 1.3. That's because rank is related to the dimensionality of a matrix and in linear algebra we work only with integer dimensions.

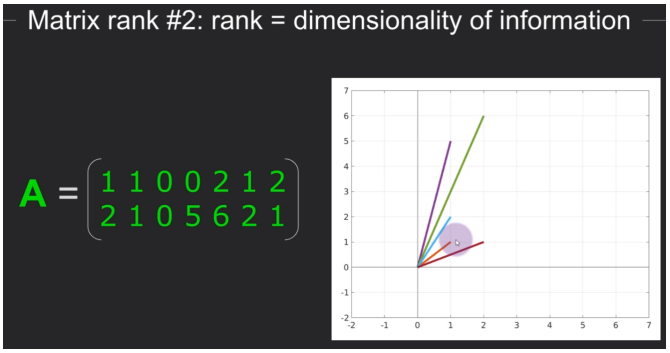

Figure 85. Rank

Second, the rank of a matrix indicates the number of dimensions of information that are coded inside the matrix. That's not the same thing as the total number of columns or the total number of rows.

Consider this matrix for example. We can interpret this algebraically or geometrically. Algebraically, it's clear that the columns of this matrix form a linearly dependent set. The subspace that is spanned by the columns, which include all of $\mathbb{R}^2$ with only two independent vectors.

And you can certainly find at least two columns in this matrix that are linearly independent – that would form a linearly independent set. The rest of the columns don't provide any new information that cannot be obtained by combining the other columns in some way.

You can also think of this matrix geometrically. You can think of each column being the endpoint of a vector from the origin. And when plotting these vectors, you can see that they all lie in a plane it's all in $\mathbb{R}^2$.

So again, any two vectors that are distinct can provide unique geometric information but one vector is a line and the second vector makes a plane, and additional vectors in that plane don't provide any new

geometric dimensions. So the rank of this matrix is two.

Here's another example.

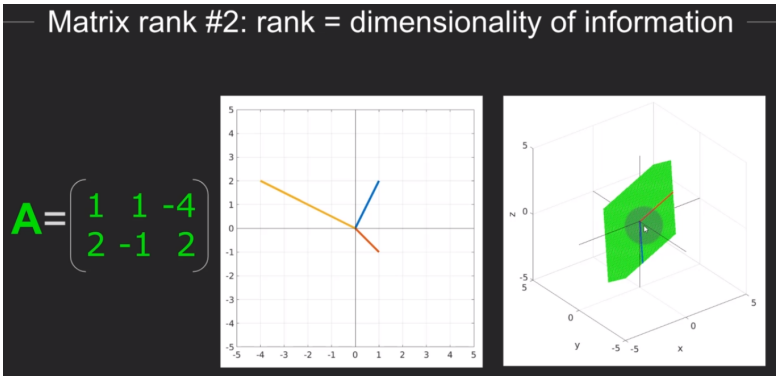

Figure 86. Rank

This is a two by three matrix. And here are the three vectors that are created by the columns.

So again geometrically, this makes us think that we have a rank 2 matrix because we have two independent vectors that span all of $\mathbb{R}^2$. And then there's a third vector that provides no new geometric dimensional information.

But I'm just talking about the columns. Maybe you are looking at the rows and you say, well the rows are actually in $\mathbb{R}^3$, So maybe the rank of the rows is three. But by plotting these vectors – the rows – you can see that this is only two vectors in $\mathbb{R}^3$, and two vectors in $\mathbb{R}^3$ that are linearly distinct can create a plane, which is a two dimensional subspace.

So in fact, the rank of this matrix is still two, regardless of whether you think about the columns or you think about the rows.

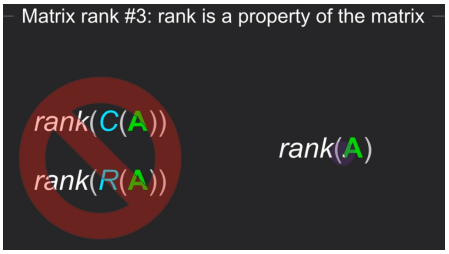

Figure 87. Rank

And in fact this is not just a unique property of this particular matrix; rank is a property of the matrix. It doesn't matter whether you're looking at the columns of the matrix or whether you're looking at the rows of the matrix. There is no such thing as the rank of a column space or the rank of the row space. There is just the rank of the matrix. And that is the third important thing that you need to know about rank.

The idea is that the rank of the matrix tells you about the number of dimensions of information that are contained in the matrix, which might be a subspace of the total dimensionality of the matrix.

So a 3 by 3 matrix lives in $\mathbb{R}^3$. It's three dimensional. But it might have rank-2 if all the information in the matrix is contained inside a plane, which is a subspace of $\mathbb{R}^3$. Again, here with rank you see the concept of the distinction between the dimensionality of a subspace versus the ambient dimensionality. This is a concept that I introduced in the section on vectors.

So here's a list so far of important things to know about rank. It's indicated with r sometimes the word **rank**.

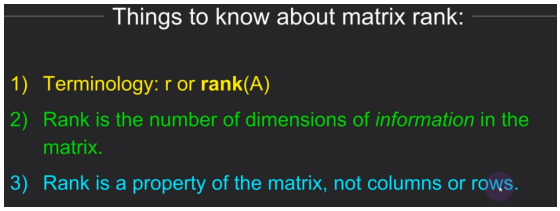

Figure 88. Rank

It corresponds to the number of dimensions of information in the matrix, and it's a property of the matrix, it doesn't matter whether you're looking at columns or rows.

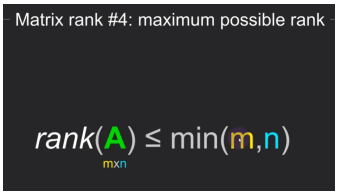

Figure 89. Rank

Now let's move on to point four. The fourth important thing to know about the rank of a matrix is that the maximum possible rank for each matrix is the smaller of the number of rows or the number of columns.

So if you have the size of matrix *A* as *M* by *N*. Then the rank cannot be bigger than the smaller, the minimum, and *m* and *n*. Keep in mind that this number – whatever is the minimum of M or N – that is a theoretical maximum. You don't necessarily have to get a rank that is this high.

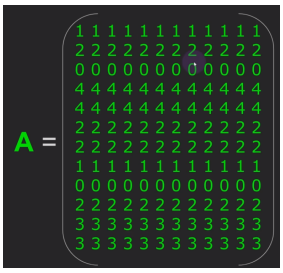

Figure 90. Rank

Consider this matrix (Figure 90), for example.

It's a 10 by 12 matrix, meaning the maximum possible rank is 10. But if you look carefully, you see that all the rows are repeated. So every row in this matrix is a scaled version of a different row in the matrix. So really, all of the information in this entire matrix is just a line, geometrically speaking. I have no idea what that line looks like, but it's still just a line. So the rank of this matrix is one.

Now consider this matrix (Figure 91). I changed a few numbers here but still if you look carefully you will see that there isn't a whole lot of information. So most of these rows are still linear combinations of other rows. So the rank of this matrix is still well below 10 or 12.

So here's a little bit more about terminology and rank. If a matrix is square and it has the maximum possible rank – so the matrix is *M* by *M* and the rank is *M*, then we call this a full-rank matrix.

If the matrix is rectangular, and the rank is equal to the number of columns, so the matrix is *M* by *N*, where *M* is greater than N, so there's more rows and columns, but the rank of the matrix is equal to the number of columns in the matrix, then this is called **full column rank**.

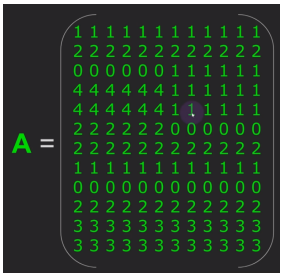

Figure 91. Rank

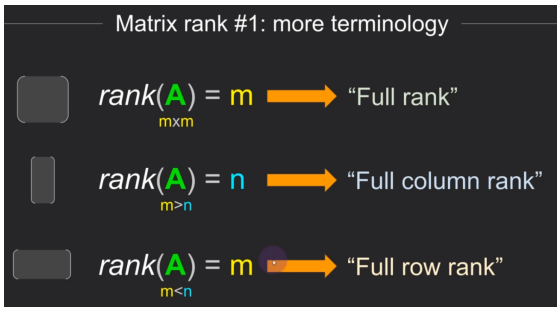

Figure 92. Rank

And you can probably guess that the converse – if the rank is equal to the number of rows for a rectangular matrix – then it's called a **full row rank matrix**.

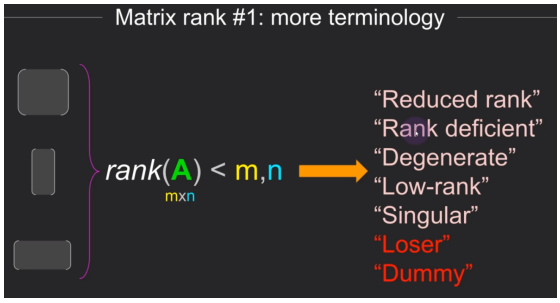

Figure 93. Rank

If the matrix rank is less than the maximum possible rank, it's given a bunch different names. You can call it **reduced-rank** or **rank-deficient** or sometimes called **degenerate** or **low rank**.

If it's a square matrix then you can also call it a singular or a non-invertible matrix.

You'll learn in the section on computing the matrix inverse why it's called a singular or a non-invertible matrix.

And if you're having a bad day and you're not really in a good mood, you can also call a reduced rank matrices a **loser matrix**, or a **dummy matrix**, or something like that. But these are the more accepted terms.

Now this leads me to the fifth thing to know about matrix rank, which you might have already guessed by now. The rank of a matrix corresponds to the number of linearly independent columns, and that turns out to be exactly the same as the number of linearly independent rows.

And in fact this is one definition of matrix rank. I'd like to stress here that I said "one" definition not "the" definition. There are actually several ways of defining and computing the rank of the matrix. I'll more to

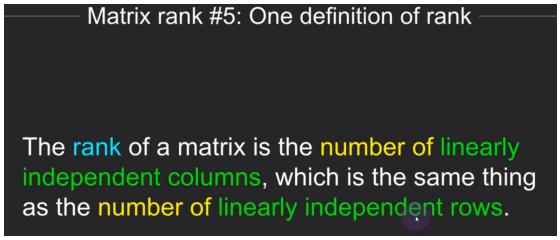

Figure 94. Rank

say about this issue in a separate video. But for now, at this point in the course, you can think about the rank as a measure of the number of linearly independent columns, or rows, in a matrix.

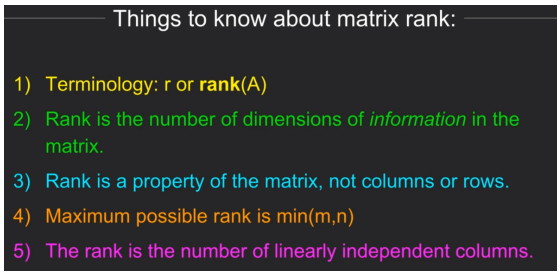

Figure 95. Rank

So there you go. There are the 5 important things to know about the rank of a matrix. Matrix rank has several applications:

It's used to test whether a matrix has an inverse, because only full-rank matrices are invertible; rank is used in multivariate data analyses such as principal components analysis or factor analysis, because you start with a big data set and you want to know how much information, or what is the dimensionality not of the matrix, but the important information that's contained in the matrix. And that corresponds to rank. Rank is also used in compression, because you can create a so-called low-rank representation of a matrix or of a data set. And I'll talk more about this on the section on eigendecompetition and singular value decomposition. I use rank often in my own research on multivariate brain activity dynamics, because the rank of the data tells me how much noise I'm able to project out of the data, and I use the rank to inform my decisions about the best ways to analyze multivariate data sets.

Computing Rank: Theory and Practive

One of the things that I find so compelling and so insightful about mathematics is when you arrive at the same answer from very different starting points.

Matrix rank is an example of this phenomenon. I mentioned in the previous video that there are several ways to compute the rank of a matrix.

Most of the ways to compute the rank of the matrix rely on linear algebra concepts and procedures that you haven't yet learned in this course, although I will cover them in later sections. In this video, I want to introduce you to the concept of different ways of computing the rank of a matrix.

Actually, they might seem like very different methods at first, but you will learn by the end of this course that these different methods are actually quite related to each other.

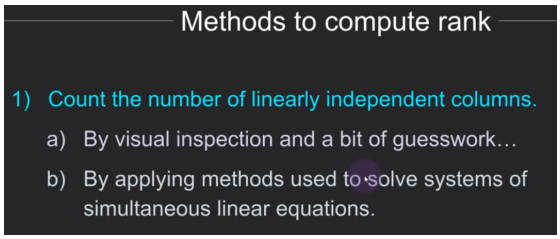

Figure 96. Rank

So I'll start with the definition of rank that I gave in the previous video, which is that the rank of a matrix corresponds to the number of linearly independent columns.

So one way to determine the rank of a matrix is by **figuring out how many linearly independent columns there are**, or sometimes it's easier to do it row-wise, because of course the rank is a property of the matrix, it doesn't matter whether you're looking at the columns or the rows. That, of course, leads to the natural question of how do you figure out the number of linearly independent columns are in a matrix.

Well, for now, at this point in the course, you just have to look at the matrix, do a little bit of arithmetic and a bit of insight and guesswork. That works fine for small matrices with relatively simple entries, like integers or maybe a few simple fractions, but of course this is not really a scalable method.

Certainly, this is not a technique you would use in practice. Soon in this course, you will learn about solving simultaneous equations using linear methods, and at that point you will learn a more algorithmic approach to determining the number of linearly independent columns.

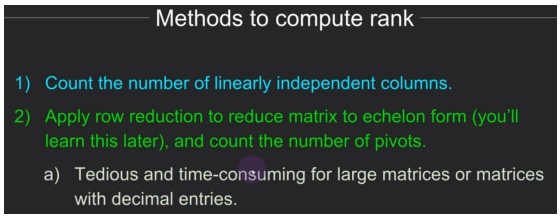

Figure 97. Rank

Another way to compute the rank of a matrix is to apply something called **row-reduction** to get the matrix into its so-called **echelon form**. Then you just count the number of **pivots**.

I will talk more in a later section about what this means but I just want to introduce you now to the ideas so you can keep it in mind. The problem is that applying a row reduction and in particular reduced row reduction is quite tedious and time consuming. It's basically impossible to do for large matrices or matrices with more difficult entries.

A third way to compute the rank of a matrix is to apply a decomposition called the **singular value decomposition**. And then you count the number of nonzero singular values of the matrix. The rank of

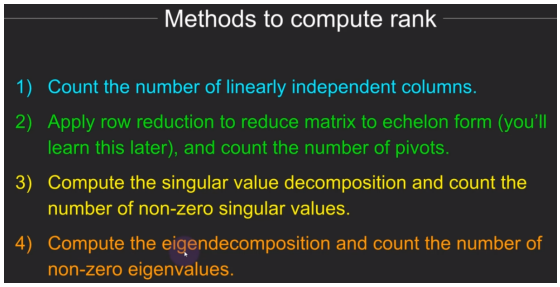

Figure 98. Rank

the matrix then corresponds to the number of nonzero singular values.

This is actually the way that Matlab and many other matrix-computing software programs will compute the rank of a matrix. And when you learn about the SVD, you will understand why the number of nonzero singular values corresponds to the rank.

A complimentary procedure is to compute the eigendecomposition of the matrix. This would work for a square matrix or if it's a rectangular matrix you can compute the eigendecomposition of the matrix times its transpose, and again count the number of non-zero eigenvalues. The number of non-zero eigenvalues corresponds to the rank of the matrix.

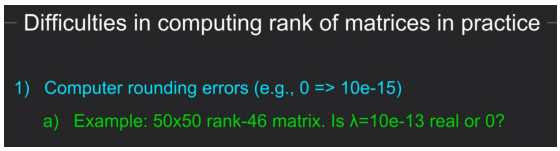

Figure 99. Rank

When you are working with little toy examples in the context of learning linear algebra in a first semester linear algebra course, particularly if you're doing a lot of paper and pencil work, then the rank is unambiguous. You compute the rank. You get a number. And that's the answer. End of story.

But in practice, and particularly when dealing with large matrices, the rank is not so trivial to compute. And it may depend on some characteristics of the matrix. It may depend on some arbitrary thresholding that you have to pick, and it also may depend on the specific algorithm that you decide to use to compute the rank. And this really has to do with noise and with machine rounding errors.

So you've seen this before, if you've taken a programming course, or if you've taken other computer-based mathematics courses. Computers often have a hard time representing exact zero. So sometimes you get an answer that you know should be zero, but in fact it's something like 10 to the minus 15 or 10 to the minus 19 depending on your computer precision. So there are just these very tiny computer rounding errors.

Imagine that you have a 50 by 50 matrix and you know somehow that the true rank of the matrix is 46. It's a little bit smaller than the full size of the matrix. Now when you do an eigendecomposition of this matrix you get 50 eigenvalues.

Now in theory, you should have 46 non-zero eigenvalues and four eigenvalues that are exactly equal to zero. The reason why that should be the case will become clear to you in the section on eigendecomposition. But for now, you can just assume that I'm telling you the truth. Let's say because of computer rounding errors, you don't see any eigenvalues that are exactly exactly equal to zero. Instead, you see a couple of eigenvalues that are somewhere around 10 to the minus 13.

So what do you do? When can you assume that such a tiny number is a real eigenvalue or a real non-zero eigenvalue that increases the rank of the matrix? Or when do you assume that this is really zero, and it's just a little bit of computer rounding error?

In practice, what you do is come up with some threshold, and then you would ignore any eigenvalues that

are less than that threshold. But of course thresholds can be arbitrary and it's difficult to know what the appropriate or the correct threshold is.

I don't want to go into a lot of the details about what I'm talking about now; again, you'll learn more about that in the section on eigendecomposition. I just want to give you an idea of where the ambiguity comes from, when determining the matrix rank.

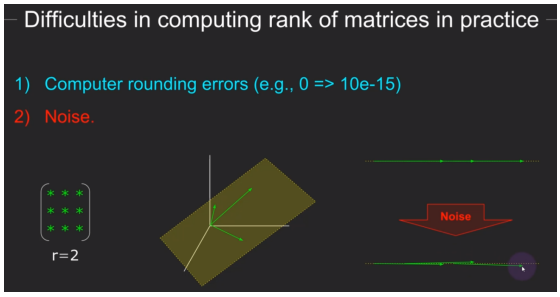

Figure 100. Rank

In practice, a related source of difficulty comes from noise. So let's say you have this 3 by 3 matrix of rank 2. So this is a matrix that the columns are in R3, but the rank is 2. So the information that's contained in this matrix exists on a 2D plane that's embedded in an ambient three dimensional space. So that's fine; there's nothing weird or wrong about this situation.

So here's this plane. Imagine you're looking at the edge of the plane; you're looking through the plane. So it looks flat like this, so this would be the plane going this way. So all three of these vectors, which are these columns in the matrix, are all in this plane. So when you're looking into the plane the three vectors would look something like this. But now imagine that these are empirically measured data, let's say from a satellite. And the measurement sensors on a satellite are a tiny bit noisy and not perfectly perfectly calibrated.

So although this is really three vectors lying in a two dimensional plane, if you have a tiny bit of noise it's possible that the vectors will point off the plane ever so slightly. An overly sensitive way of computing rank might actually call this a rank 3 matrix even though we know that is just a rank 2 matrix so the information is just in two dimensions. So again, you need to have some kind of threshold that decides when this little bit of projection off the plane should be considered an additional source of information in the matrix, which would knock up the rank to 3.

Rank of Added and Multiplied Matrices

In this video I'm going to tell you about the boundaries for the rank of summed matrices, and multiplied matrices. The question here is: if you know the rank of A, and you know the rank of B, what can you infer about the rank of $A+B$ and the rank of AB?

Now the thing is, you cannot know a priori exactly what the rank of A plus B or A times B will be, even if you know the ranks of A and B, because it depends on the specific numbers, the specific elements. However, there are rules for the upper boundary of what the maximum possible rank could be in this case and in this case. In this video, you will learn what those rules are.

Before explaining what the rule is and saying something about why it must be true, I want to start with giving a few examples. Let's start with the rank of A plus B.

Here I have four sets of matrix equations.

If you're up for a bit of a challenge, and some practice in computing rank, I encourage you to pause the

video here and compute the rank of each of these 12 matrices.

It seems like a lot but it's actually not so bad.

I think the rank is relatively straightforward to compute by visual inspection and a little bit of guesswork

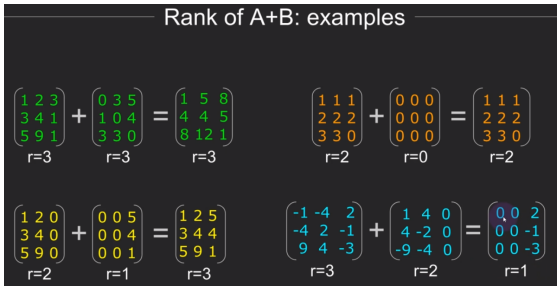

Figure 101. Rank

in all 12 of these cases.

Now I go through each one of these in turn.

So here we have three matrices, all of them have rank 3.

So A has rank-3, matrix B has rank 3, and their sum as also rank 3.

So these are all full-rank matrices.

Here you have an example where this matrix has rank 2.

So this is rank deficient.

This matrix has rank 1, it's also rank deficient.

And yet the two of these matrices added together turns out to be a full rank matrix, so rank 3 matrix.

So that's a pretty interesting case. In this example,

we have this matrix with rank 2, this matrix with rank two – uhh, sorry, with rank zero – and the resulting sum has rank 2.

Finally, we get to the fourth case that I'm illustrating here. Here we start with the matrix of rank 3.

So this is a full rank matrix.

This is a reduced rank matrix because there is one column of zeros, so this is rank 2, and the resulting matrix

has a rank of 1.

And that happened because basically these first two columns here are the negatives of each other

for these two matrices, so you end up with a reduced rank matrix.

So I've illustrated several different possibilities of the rank: Here the rank of the summed matrix

is the same as the rank of the two matrices; here,

this is a funny case where the rank is actually larger – the summed matrix has a larger rank than either of the individual matrices,

and here you see reduced-rank cases.

Now let me tell you the rule about the boundary of the rank of A plus B.

The rule is that the rank of matrix A plus B cannot be larger than the sum of the individual ranks.

So if you know the rank of A and you know the rank of B, you add those together, and the rank of A plus B cannot be bigger than this.

Actually there is an additional bound on top of this, which is that the rank can also not be larger than the smaller of M or N, so the smaller of the number of rows or the number of columns. We've already established that rule in a previous video.

I didn't mention it explicitly here.

And you can see that in this first case here.

So rank 3 and rank 3 is a three by three matrix.

But the sum of these two numbers is six. Obviously, a three by three Matrix cannot have a rank of six.

One way to think about this rule is that by adding matrices, you are simply linearly pooling information element by element, and by pooling across different stores of information, the total amount of information can increase, because the pooling is linear.

It's additive.

You cannot create any new information out of thin air.

You can only work with what you already have.

So here I have a little bit of non-overlapping information in these two matrices: this one here and this one here.

So the non-overlapping bits will increase the total information in the matrix.

But there's just no way to create new information here in this third column because we're not starting with any information in these original columns.

This is the way that I like to think about this role.

And needless to say I didn't say this explicitly but of course this rule makes sense only when A and B can be added together, which means that A and B are the same size.

The second thing I'd like to point out here is that scaling a matrix by some scalar, so scalar-matrix multiplication, doesn't change the rank of the matrix.

So that means you can multiply B by any arbitrary scalar.

You could multiply by a scalar of -1 and that would turn this addition and subtraction, and that doesn't change the rule at all.

OK so this is about the rank of A plus B.

Now let's talk about multiplication and the rank of A times B.

Again, I'll first start by showing a few examples and then I'll explain what the rule is.

So here again we have these matrices that we are working with.

But now instead of adding these matrices together, I am multiplying them together.

And now the resulting rank is a little bit different in some cases, compared to what it was for the addition case.

In this case, we start off with two full-rank matrices, and we happened to also end up with a full rank matrix,

in this case rank 3. That's not necessarily the case,

that's just how it happened to work out with these specific numbers.

Here we have our rank 2 and rank 1 matrix. You'll remember, or you can go back a few slides, that when we added these two matrices together, the resulting matrix was full rank.

So this was rank 2, this was rank 1. When we added these two matrices together, their result was actually rank 3, because all of these zeros get added with unique, new information. However, here when we do the product, we end up with a rank-1 matrix. In this case, we are multiplying by the zero matrix. The zero matrix

has rank zero.

And of course any matrix times the zero matrix is the zero matrix.

So the product is zero.

And in this case we end up with a rank 2 matrix, and one column of zeros.

So with these examples in mind, here is the rule for matrix multiplication.

The rule is that the rank of the product matrix of A times B cannot be any greater than the smaller of the ranks of either matrix individually.

So if you know the rank of A and you know the rank of B, then whichever rank is smaller sets the maximum

possible boundary of the rank of A times B. The rank of A times B could actually be smaller than whatever is the minimum of these two, but it couldn't be larger than the minimum of these two.

And this rule we can see in these different examples here.

So here the smallest rank is three.

And the resulting product matrix happens to be three. Here the smallest rank is one and the product matrix has rank 1. 0 and 0, and 2 and 2.

There are several proofs to this statement.

I will show you one proof.

This is the proof that I find intuitive.

It has to do with the column space of a matrix.

The column space is a concept that you will learn more about in the next section, but basically the column space is the subspace that is spanned by the columns of a matrix.

So let's start by calling this product matrix C.

So this is the rank of C.

If you think about each column in C as being the matrix-vector product of the matrix A and the corresponding

column of B,

so the jth column in C is the jth column in B,

post-multiplying the matrix A, then basically each column of C is a linear combination of the columns of A.

This means that the subspace spanned by the columns of C is at most the same dimensionality as the subspace

spanned by the columns of A.

Now it could be smaller dimensionality, depending on the elements in B, but it certainly couldn't be larger.

And you'll remember I mentioned in a previous video that the dimensionality of a subspace spanned by the

columns of a matrix is one of the definitions of rank.

Therefore the rank of C cannot be bigger than the rank of A.

It could be smaller but it couldn't be bigger.

Or, you can make the same argument for the rows of matrix C coming from the rows of matrix A times matrix B.

And there you would come to the conclusion that the rank of matrix C cannot be any bigger than the rank of matrix B, because the rows in matrix C come from the rows of B weighted according to the elements in A.

So here you see both of these rules on the same page.

Now to be honest, these are not rules that you really have to use in practice in applied linear algebra.

So I don't know if you really need to worry about committing these rules to memory, except if you are taking a college course in linear algebra and you would need to recite these rules in an exam for example.

Nonetheless, matrix rank is such an important property and you use the matrix rank very often in practice.

So the more comfortable you are with thinking about matrix rank and working with matrix rank, the easier it will be able to learn the more advanced topics. The more general point from this video is that

the rank of a summed matrix or a product matrix is not necessarily the same as the rank of either of the individual matrices.

One final point I'd like to make here is that in the examples I shown in this video all the matrices were square matrices, but these two rules apply equally well to rectangular matrices,

as long as the addition or the multiplication is valid. In the exercises for this section, you will have the opportunity to explore these rules using different sized matrices.