MICHALE FEE: OK. All right, let's go ahead and get started. OK, so we're going to continue talking about the topic of neural networks. Last time, we introduced a new framework for thinking about neural network interactions, using a rate model to describe the interactions of neurons and develop a mathematical framework for how to combine collections of neurons to study their behavior.

So, last time, we introduced the notion of a perceptron as a way of building a neural network that can classify its inputs. And we started talking about the notion of a perceptron learning rule, and we're going to flesh that idea out in more detail today. We're going to then talk about the idea of using networks to perform logic with neurons. We're going to talk about the idea of linear separability and invariance.

Then we're going to introduce more complex feed-forward networks, where instead of having a single output neuron, we have multiple output neurons. Then we're going to turn to a more fully developed view of the math that we use to describe neural networks, and matrix operations become extremely important in neural network theory. And then, finally, we're going to turn to some of the kinds of transformations that are performed by matrix multiplication and by the kinds of-- by feed-forward neural networks.

OK, so we've been considering a kind of neural network called a rate model that uses firing rates rather than spike trains. So we introduced the idea that we have an output neuron with firing rate v that receives input from an input neuron that has firing rate u. The input neuron synapses onto the output neuron with a synapse of weight w. And we described how we can think of the input neuron producing a synaptic input into the output neuron that has a magnitude of the firing rate times the strength of the synaptic connection.

So the input to the output neuron here is w times u. And then we talked about how we can convert that input current, let's say, into our output neuron into a firing rate of the output neuron through some function f, which is what's called the F-I curve of the neuron that relates the input to the firing rate of the neuron. And we talked about several different kinds of F-I firing rate versus input functions that can be useful.

We then extended our network from a single input neuron synapsing onto a single output neuron by having multiple input neurons. Again, the output neuron has a firing rate, and our input neurons have a vector of firing rates now-- u1, u2, u3, u4, and so on-- that we can combine together into a vector, u. Each one of those input neurons has a synaptic strength w onto our output neuron. So we have a vector of synaptic strengths.

And now we can write down the input current to our output neuron as a sum of the contributions from each of those input neurons-- so w1, u1 plus w2, u2, plus w3, u3, and so on. So we can now write the input current to our output neuron as a sum of contributions that we can then write as a dot product-- w dot u. OK, any questions about that?

And so, in general, we have the firing rate of our output neuron is just this F-I function, this input-output function of our output neuron acting on the total input, which is w dot u. And then we talked about different kinds of functions that are useful computationally for this function f. So in the context of the integrate and fire neuron, we talked about F-I curves that are zero below some threshold and then are linear above that threshold current.

We talked last time about a binary threshold known that has zero firing rate below some threshold and then steps up abruptly to a constant output firing rate one. And then we also introduced, last time, the notion of a linear neuron, whose firing rate is just proportional to the input current and has positive and negative firing rates.

And we talked about the idea that although it's biophysically implausible to have neurons that have negative firing rates, that this is a particularly useful simplification of neurons. Because we can just use linear algebra to describe the properties of networks of linear neurons. And we can do some really interesting things with that kind of mathematical simplification. We're going to get to some of that today. And that allows you to really build an intuition for what neural networks can do.

OK, so let's come back to what perceptron is and introduce this perceptron learning role. So we talked about the idea that a perceptron carries out a classification of its inputs that represent different features. So we talked about classifying animals into

dogs and non-dogs based on two features of animals. We talked about the fact that you can't make that classification between dogs and non-dogs just on the basis of one of those features, because these two categories overlap in this feature and in this feature.

And so in order to properly separate those categories, you need a decision boundary that's actually a combination of those two features. And we talked about how you can implement that using a simple network, called a perceptron, that has an output neuron and two input neurons. Each one of those input neurons represents the magnitude of those two different features for each object that you're trying to classify.

So u1 here and u2 are the dimensions on which we're performing this classification. And so we talked about the fact that that decision boundary between those two classifications is determined by this weight matrix w. And then we used a binary threshold neuron for making the actual decision. Binary threshold neurons are great for making decisions, because unlike a linear neuron-- so a linear neuron just responds more if its input is larger, and it responds less if its input is smaller. Binary threshold neurons have a very clear threshold below which the neuron doesn't spike and above which the neuron does spike.

So, in this case, this network, this output neuron here, will fire, will have a firing rate of one, for any input that's on this side of the decision boundary and will have a firing rate of zero for any input that's on this side of the decision boundary, OK? All right, so we talked about how we can, in two dimensions, just write down a decision boundary that will separate, let's say, green objects from red objects. So you can see that if you sat down and you looked at this drawing of green dots and red dots, that it would be very simple to just look at that picture and see that if you put a decision boundary right there, that you would be able to separate the green dots from the red dots.

How would you actually calculate the weight vector that that corresponds to in a perceptron? Well, it's very simple. You can just look at where that decision boundary crosses the axes-- so you can see here, that decision boundary crosses the u1 axis at point A, crosses the u2 axis at, I should say, a value of B. And then we can use those numbers to actually calculate the w.

So, remember, u is the input space. w is a weight vector that we're trying to calculate in order to place the decision boundary at that point. Is that clear what we're trying to do here? OK, so we can calculate that weight vector. We assume that just data is some number. Let's just call it one. We have an equation for a line-- w dot u equals theta. That's the equation for that decision boundary.

We have two knowns, the two points on the decision boundary that we can just read off by eye. And we have two unknowns-- the synaptic weights, w1 and w2. And so we have two equations-- ua dot w equals theta, ub dot w equals theta. And we can just solve for w1 and w2, and that's what you got, OK?

So the weight vector that gives you that decision boundary is 1 over a and 1 over b, OK? Those are the two weights. Any questions about that? OK.

So in two dimensions, that's very easy to do, right? You can just look at that cloud of points, decide where to draw a line that best separates the two categories that you're interested in separating. But in higher dimensions, that's really hard. So in high dimensions, for example, we're trying to separate images, for example. So we can have a bunch of images of dogs, a bunch of images of cats. Each pixel in that image corresponds to a different input to our classification unit.

And now how do you decide what all of those weights should be from all of those different pixels onto our output neuron that separates images of one class from images of another class? So there's just no way to do that by eye in high dimensions. So you need an algorithm that helps you choose that set of weights that allows you to separate different classes-- you know, a bunch of images of one class from a bunch of images of another class.

And so we're going to introduce a method called the perceptron learning rule that is a category of learning rules called supervised learning rules that allow you to take a bunch of objects that you know-- so if you have a bunch of pictures of dogs, you know that they're dogs. If you have a bunch of pictures of cats, you know they're cats. So you label those images. You feed those inputs, those images, into your network, and you tell the network what the answer was.

And through an iterative process, it finds all of the weights that optimally separate

those two different categories. So that's called the perceptron learning rule. So let me just set up how that actually works. So you have a bunch of observations of the input. So in this case, I'm drawing these in two dimensions, but you should think about each one of these dots as being, let's say, an image of a dog in very high dimensions, where instead of just u1 and u2, you have u1 through u1000, where each one of those is the value of a different pixel in your image.

So you have a bunch of images. Each one of those corresponds to an image of a dog. Each one of those corresponds to an image of a cat. And we have a whole bunch of different observations or images of those different categories. Any questions about that?

All right, so we have n of those observations. And for each one of those observations, we say that the input is equal to one of those observations for one iteration of this learning process, OK? And so with each observation, we're told whether this input corresponds to one category or another, so a dog or a non-dog. And our output, we're asking-- we want to choose this set of weights such that the output of our network is equal to some known value.

So t sub i, where if it's a dog, then the answer is one for yes. If it's a non-dog, the answer is no for that's not a dog. And we have n of those answers. We have n images and labels that tell us what category that image belongs to. So for all of these, t equals one. For all of these, t equals zero.

And we want to find a set of weights such that when we take the dot product of that weight factor into each one of those observations minus theta that we get an answer that is equal to t for each observation. Does that make sense? So how do we do that?

All right, so each observation, we have two things-- the input and the desired output. And that gives us information that we can use to construct this weight vector. So, again, that's called supervised learning. And we're going to use an update rule, or a learning rule, that allows us to change the weight vector on as a result of each estimate, depending on whether we got the answer right or not. So how do we do this?

What we're going to do is we're going to start with a random set of weights, w1 and

w2, OK? And we're going to put in an input. So there's a space of inputs. We're going to start with some random weight, and I started with some random vector in this direction. You can see that that gives you a classification boundary here. And you can see that that classification boundary is not very good for separating the green dots from the red dots.

Why? Because it will assign a one to everything on this side of that decision boundary and a zero to everything on that side. But you can see that that does not correspond to the assignment of green and red to each of those dots, OK? So how do we update that w in order to get the right answer? So what we're going to do is we're going to put in one of these inputs on each iteration and ask whether the network got the answer right or not.

So we're going to put in one of those inputs. So let's pick that input right there. We're going to put that into our network. And we see that the answer we get from the network is one, because it's on the positive side of the decision boundary. And so one was the right answer in this case. So what do we do? We don't do anything. We say the change in weight is going to be zero if we already get the right answer.

So if we got lucky and our initial weight vector was in the right direction, so our perceptron already classified the answer, then the weight vector is never going to change, because it was already the right answer. OK, so let's put it in another input-- a red input. You can see that the correct answer is a zero. The network gave us a zero, because it's on the negative side of the weight vector of the decision boundary. And so, again, delta w is zero.

But let's put in another input now such that we get the wrong answer. So let's put in this input right here. So you can see that the answer here, the correct answer is one, but the network is going to give us a zero. So what do we do to update that weight vector?

So if the output is not equal to the correct answer, then we're wrong. So now we update w. And the perceptron learning rule is very simple. We introduce a change in w that looks like this. It's a little change, so eps eta is a learning rate. It's generally going to be smaller than one. So we're going to put in a small change in w that's in the direction of the input that was wrong if the correct answer is a one.

We're going to make a small change to w in the opposite direction of that input if the correct answer was zero. Does that make sense? So we're going to change w in a way that depends on what the input was and what the correct answer was.

So let's walk through this. So we put it in an input here. The correct answer is a one, and we got the answer wrong. The network gave us a zero, but the correct answer is a one. So we're in this region here. The answer was incorrect, so we're going to update w. The correct answer was a one, so we're going to change delta-- we're going to change w in the direction of that input.

So that input is there. So we're going to add a little bit to w in this direction. So if we add that little bit of vector to the w, it's going to move the w vector in this direction, right? So let's do that. So there's our new w. Our new w is the old plus delta w, which is in the direction of this incorrectly classified input. So there's our new decision boundary, all right?

And let's put in another input-- let's say this one right here. You can see that this input is also incorrectly classified, because the correct answer is a zero. It's a red dot. But the network since it's on the positive side of the decision boundary. So the network classifies it as a one. OK, good. So the network classified it as a one and the correct answer was a zero, so we were wrong. So we're going to update w, and we're going to update it in the opposite direction of the input if the correct answer was zero, which is the case.

So we're going to update w. And that's the input xi. Minus xi is in this direction. So we're going to update w in that direction. So we're going to add those two vectors to get our new w. And when we do that, that's what we get. There's our new w. There's our new decision boundary. And you can see that that decision boundary is now perfectly oriented to separate the red and the green dots. So that's Rosenblatt's perceptron learning rule. Yes, Rebecca?

AUDIENCE: How do you change the learning rate? Because what if it's too big? You'll sort of get not helpful [INAUDIBLE].

MICHALE FEE: Yeah, that's right. So if the learning rate were too big, you could see this first correction. So let's say that we corrected w but made a correction that was too far

in this direction. So now the new w would point up here. And that would give us, again, the wrong answer. What happens, generally, is that if your learning rate is too high, then your weight vector bounces around. It oscillates around.

So it'll jump too far this way, and then it'll get an error over here, and it'll jump too far that way. And then you'll get an error over there, and it'll just keep bouncing back and forth. So you generally choose learning rates that-- the process of choosing learning rates can be a little tricky Basically, the answer is start small and increase it until it breaks. OK, any questions about that?

So you can see it's a very simple algorithm that provides a way of changing w that is guaranteed to converge toward the best answer in separating these two classes of inputs. All right, so let's go a little bit further into single layer binary networks and see what they can do. So these kinds of networks are very good for actually implementing logic operations.

So you can see that-- let's say that we have a perceptron that looks like this. Let's give it a threshold of 0.5 and give it a weight vector that's 1 and 1. So you can see that this perceptron gives an answer of zero. The output neuron has zero firing rate for an input that's zero. But any input that's on the other side of the decision boundary produces an output firing rate of one.

What that means is that if the input a, or u1, is a 1, 0, then the output neuron will fire. If the input is 0, 1, the output neuron will fire. And if the input is 1, 1, the output neuron will fire. So, basically, any input above some threshold will make the output neuron fire. So this perceptron implements an OR gate. If it's input a or input b, the output neuron spikes, as long as those inputs are above some threshold value. So that's very much like a logical OR gate.

Now let's see if we can implement an AND gate. So it turns out that implementing an AND gate is almost exactly like an OR gate. We just need-- what would we change about this network to implement an AND gate?

AUDIENCE: A larger [INAUDIBLE].

MICHALE FEE: What's that?

AUDIENCE: A larger theta?

MICHALE FEE: Yeah, a larger theta. So all we have to do is move this line up to here. And now one of those inputs is not enough to make the output neuron fire. The other input is not enough to make the output neuron fire. Only when you have both. So that implements an AND gate. We just increase the threshold a little bit.

Does that make sense? So we just increase the threshold here to 1.5. And now when either input is on at a value of one, that's not enough to make the output neuron fire. If this input's on, it's not enough. If that output is on, it's not enough. Only when both inputs are on do you get enough input to this output neuron to make it have a non-zero firing rate, to get it above threshold.

Now, there's another very common logic operation that cannot be solved by a simple perceptron. That's called an exclusive OR, where this neuron, this network, we want it to fire only if input a is on or input b is on, but not both. Why is it that that can't be solved by the kind of perceptron that we've been describing? Anybody have some intuition about that?

AUDIENCE: I mean, it's obviously [INAUDIBLE] separable.

MICHALE FEE: Yeah, that's right. The keyword there is separable. If you look at this set of dots, there's no single line, there's no single boundary that separates all the red dots from off the green dots, OK? And so that set of inputs is called non-separable. And sets of inputs that are not separable cannot be classified correctly by a simple perceptron of the type we've been talking about.

So how do you solve that problem? So this is a set of inputs that's non-separable. You can see that you can solve this problem now if you have two separate perceptrons. So watch this. We can build one perceptive one that fires, that has a positive output when this input is on. We can have a separate perceptron that is active when that input is on.

And then what would we do? If we had one neuron that's active if that input is on another input that's active when that input is on? We would or them together, that's right. So this is what's known as a multi-layer perceptron. We have two inputs, one that represents activity in a, another that represents activity in b. And we have one neuron in what's called the intermediate layer of our perceptron that has a weight

vector of 1 minus 1.

What that means is this neuron will be active if input a is on but not input b. This one will be active. This neuron has a different weight vector-- minus 1, 1. This neuron will be active if input b is on but not input a. And the output neuron implements an OR operation that will be active when this intermediate neuron is on or that intermediate neuron is on, OK? And so that network altogether implements this exclusive OR function. Does that make sense? Any questions about that?

So this problem of separability is extremely important in classifying inputs in general. So if you think about classifying an image, like a number or a letter, you can see that in high-dimensional space, images that are all threes, let's say, are all very similar to each other. But they're actually not separable in this linear space. And that's because in the high dimensional space they exist on what's called a manifold in this high-dimensional space, OK?

They're like all lined up on some sheet, OK? So this is an example of rotations, and you can see that all these different threes kind of sit along a manifold in this high-dimensional space that are separate from all the other numbers. So all those numbers exist on what's called an invariant transformation, OK?

Now, how would we separate those images of threes from all the other numbers or letters? How would we do that? Well, we could imagine building a multi-layer perceptron that-- so here, I'm showing that there's no single line that separates the threes on this manifold from all the other digits over here. We can solve that problem by implementing a multi-layer perceptron that while one of those perceptrons detects these objects, another perceptron detects these objects, and then we can OR those all together.

So that's a kind of network that can now detect all of these three, separate them from non-threes. Does that make sense? So we can think of objects that we recognize, like this three that we recognize, even though it has different-- we can recognize it with different rotations or transformations or scale changes. You can also think of the problem of separating images from dogs and cats as also solving this problem, that the space of dogs, of dog images, somehow lives on a manifold in the high dimensional space of inputs that we can distinguish from the set of images

of cats that's some other manifold in this high-dimensional space.

So it turns out that you need more than just a single layer perceptron. You need more than just a two-layer perceptron. In general, the kinds of networks that are good for separating different kinds of images, like dogs and cats and cars and houses and faces, look more like this. So this is work from Jim DiCarlo's lab, where they found evidence that networks in the brain that do image classification-- for example, in the visual pathway-- look a lot like very deep neural networks, where you have the retina on the left side here sending inputs to another letter in the thalamus, sending inputs to v1, to v2, to v4, and so on, up to IT.

And that we can think of this as being, essentially, many stacked layers of perceptrons that sort of unravel these manifolds in this high-dimensional space to allow neurons here at the very end to separate dogs from cats from buildings from faces. And there are learning rules that can be used to train networks like this by putting in a bunch of different images of people and other different categories that you might want to separate. And then each one of those images has a label, just like our perceptron learning rule.

And we can use the image and the correct label-- face or dog-- and train that network by projecting that information into these intermediate layers to train that network to properly classify those different stimuli, OK? This is, basically, the kind of technology that's currently being used to train-- this is being used in AI. It's being used to train driverless cars. All kinds of technological advances are based on this kind of technology here. Any questions about that? Aditi?

AUDIENCE: So in actual neurons, I assume it's not linear, right?

MICHALE FEE: Yes. These are all nonlinear neurons. They're more like these binary threshold units than they are like linear neurons. That's right.

AUDIENCE: But then do you there's, like-- because right now, I imagine that models we make have to have way more perceptron units.

MICHALE FEE: Yes.

AUDIENCE: We use our simplified [INAUDIBLE]. But then our brain is sometimes-- I mean, it's at, like, a much faster level, like way faster, right? So you think it'd be like-- if we

examine what functions neurons might be using, in a way that would let us reduce the number of units needed? Because right now, for example, [INAUDIBLE] be a bunch of lines. But maybe in the brain, there's some other function it's using, which is smoother.

MICHALE FEE: Yeah. OK, so let me just make sure I understand. You're not talking about the F-I curve of the neurons? Is that correct? You're talking about the way that you figure out these weights. Is that what you're asking about?

AUDIENCE: No. I'm asking if we use a more accurate F-I curve, we'll need less units.

MICHALE FEE: OK, so that's a good question. I don't actually know the answer to the question of how the specific choice of F-I curve affects the performance of this. The big problem that people are trying to figure out in terms of how these are trained is the challenge that in order to train these networks, you actually need thousands and thousands, maybe millions, of examples of different objects here and the answer here.

So you have to put in many thousands of example images and the answer in order to train these networks. And that's not the way people actually learn. We don't walk around the world when we're one-year-old and our mother saying, dog, cat, person, house. You know, it would be... in order to give a person as many labeled examples as you need to give these networks, you would just be doing nothing, but your parents would be pointing things out to you and telling you one-word answers of what those are.

Instead, what happens is we just observe the world and figure out kind of categories based on other sorts of learning rules that are unsupervised. We figure out, oh, that's a kind of thing, and then mom says, that's a dog. And then we know that that category is a dog. And we sometimes make mistakes, right? Like a kid might look at a bear and say, dog. And then dad says, no, no, that's not a dog, son.

So the learning by which people train their networks to do classification of inputs is quite different from the way these deep neural networks work. And that's a very important and active area of research. Yes?

AUDIENCE: Is the fact that [INAUDIBLE] use unsupervised learning, as well, to train a computer

to recognize an image of a turtle as a gun, but humans can't do that [INAUDIBLE].

MICHALE FEE: Recognize a turtle if what?

AUDIENCE: Like I saw this thing where it was like at MIT, they used an AI. They manipulated pixels in images and convinced the computer that it was something that it was not actually.

MICHALE FEE: I see. Yeah.

AUDIENCE: So like you would see a picture of a turtle, but the computer would get that picture and say it was, like, a machine gun.

MICHALE FEE: Just by manipulating a few pixels and kind of screwing with its mind.

AUDIENCE: Yes. So it's [INAUDIBLE].

MICHALE FEE: Yeah. Well, people can be tricked by different things. The answer is, yes, it's related to that. The problem is after you do this training, we actually don't really understand what's going on in the guts of this network. It's very hard to look at the inside of this network after it's trained and understand what it's doing. And so we don't know the answer why it is that you can fool one of these networks by changing a few pixels.

Something goes wrong in here, and we don't know what it is. It may very well have to do with the way it's trained, rather than building categories in an unsupervised way, which could be much more generalizable. So good question. I don't really know the answer. Yes?

AUDIENCE: Sorry, can you explain what you mean [INAUDIBLE] the neural network needs an answer? They're not categorized and then tell the user dogs?

MICHALE FEE: Yeah, so no, in order to train one of these networks, you have to give it a data set, a labeled data set. So a set of images that already has the answer that was labeled by a person.

AUDIENCE: So you can't just give it a set of photos of puppies and snakes and it'll categorize them into two groups?

MICHALE FEE: No, nobody knows how to do that. People are working on that, but it's not known yet.

Yes, Jasmine?

AUDIENCE: [INAUDIBLE] but I see [INAUDIBLE] I can't separate them and like adding an additional feature to raise it to a higher dimensional space, where it's separable?

MICHALE FEE: Sorry, I didn't quite understand. Can you say it again?

AUDIENCE: I think I remember reading somewhere about how when the scenes are nonlinearly separable--

MICHALE FEE: Yes.

AUDIENCE: --you can add in another feature to [INAUDIBLE].

MICHALE FEE: Yeah, yeah. So let me show you an example of that. So coming back to the exclusive OR. So one thing that you can do, you can see that the reason this is linearly inseparable-- it's not linearly separable-- is because all these points are in a plane. So there's no line that separates them. But one way, one sort of trick you can do, is to add noise to this. So that now, some of these points move. You can add another dimension.

So now let's say that we add noise, and we just, by chance, happen to move the green dots this way and the red dots, well, that way. And now there's a plane that will separate the red dots from the green dots. So that's advanced beyond the scope of what we're talking about here. But yes, there are tricks that you can play to get around this exclusive OR problem, this linear separability problem, OK? All right, great question. All right, let's push on.

So let's talk about more general two-layer feed-forward networks. So this is referred to as a two-layer network-- an input layer and an output layer. And in this case, we had a single input neuron and a single output neuron. We generalized that to having multiple input neurons and one output neuron. We saw that we can write down the input current to this output neuron as w, the vector of weights, dotted into the vector of input firing rates to give us an expression for the firing rate of the output neuron.

And now we can generalize that further to the case of multiple output neurons. So we have multiple input neurons, multiple output neurons. You can see that we have

a vector of firing rates of the input neurons and a vector of firing rates of the output neurons. So we used to just have one of these output neurons, and now we've got a whole bunch of them. And so we have to write down a vector of fire rates in the output layer.

And now we can write down the firing rate of our output neurons as follows. So the firing rate of this neuron here is going to be a dot product of the vector of weights onto it. So the firing rate of output neuron one is the vector of weights onto that first output neuron dotted into the vector of input firing rates. And the same for the next output neuron. The firing rate of output neuron two is dot product of the weights onto that output neuron two and onto the vector of input firing rates.

Same for neuron three. And we can write that down as follows. So the eighth output-- the firing rate of the eighth output neuron is the weight vector onto the eighth output neuron dotted into the input firing rate vector, OK? And we can write that down as follows, where we've now introduced a new thing here, which is a matrix of weights. So it's called the weight matrix. And it essentially is a matrix of all of these synaptic weights, from the input layer onto the output layer.

And now if we had a linear neuron, we can write down the firing rate of the output neuron. The firing rate vector of output neuron is just this weight matrix times the vector of input fire rates. So now, we've rewritten this problem of finding the vector of output firing rates as a matrix multiplication. And we're going to spend some time talking about what that means and what that does.

So our feed-forward network implements a matrix multiplication. All right, so let's take a closer look at what this weight matrix looks like. So we have a weight matrix w sub a comma b that looks like this. So we have four input neurons and four output neurons. We have a weight for each input neuron onto each output neuron. The columns here correspond to different input neurons. The rows correspond to different output neurons.

Remember, for a matrix, the elements are listed as w sub a, b, where a is the output neuron. b is the input neuron. On so it's w postsynaptic, presynaptic-- post, pre. Rows, columns. So the rows are the different output neurons. The columns are the different input neurons.

So it can be a little tricky to remember. I just remember that it's rows-- a matrix is labeled by rows and columns. And weight matrices are postsynaptic, presynaptic-- post, pre.

AUDIENCE: [INAUDIBLE] comment of [INAUDIBLE]?

MICHALE FEE: I think that's standard. I'm pretty sure that's very standard. If you find any exceptions let me know. OK, we can think of each row of this matrix as being the vector of weights onto one output neuron. That row is a vector of weights onto that output neuron-- that row, that output neuron; that row, that output neuron. Does that makes sense?

All right, so let's flesh out this matrix multiplication. The vector of output firing rates, we're going to write it as a column vector, where the first number is this firing rate. That number is that firing rate. That number represents that firing rate, OK? That's equal to this weight matrix times the vector of input firing rates, again, written as a column vector.

And in order to calculate the firing rate of the first output neuron, we take the dot product of the first row of the weight matrix and the column vector of input firing rates. And that gives us this first firing rate, OK? To get the second firing rate, we take the dot product of the second row of weights with the vector of firing rates, and that gives us this second firing rate. Any questions about that? Just a brief reminder of matrix multiplication.

All right, no questions? All right, so let's take a step back and go quickly through some basic matrix algebra. I know most of you have probably seen this, but many haven't, so we're just going to go through it. All right, so just as vectors are-- you can think of them as a collection of numbers that you write down. So let's say that you are making a measurement of two different things-- let's say temperature and humidity.

So you can write down a vector that represents those two quantities. So matrices you can think of as collections of vectors. So let's say we take those two measurements at different times, at three different times. So now we have a vector one, a vector two, and a vector three that measure those two quantities at three different times, all right? So we can now write all of those measurements down as a

matrix, where we collect each one of those vectors as a column in our matrix, like that. Any questions about that?

And there's a bit of MATLAB code that calculates this matrix by writing three different column vectors and then concatenating them into a matrix. All right, and you can see that in this matrix, the columns are just the original vectors, and the rows are-- you can think of those as a time series of our first measurement, let's say temperature. So that's temperature as a function of time. This is temperature and humidity at one time. Does that make sense?

All right, so, again, we can write down this matrix. Remember, this is the first measurement at time two, the first measurement at time three. We have two rows and three columns. We can also write down what's known as the transpose of a matrix that just flips the rows and columns. So we can write transpose, which is indicated by this capital super scripted t. And here, we're just flipping the rows and columns. So the first row of this matrix becomes the first column of the transposed matrix. So we have three rows and two columns.

A symmetric matrix-- I'm just defining some terms now. A symmetric matrix is a matrix where the off-diagonal elements-- so let me just define, that's the diagonal, the matrix diagonal. And a symmetric matrix has the property that the off-diagonal elements are zero. And a symmetric matrix has the property that the transpose of that matrix is equal to the matrix, OK?

That is only possible, of course, if the matrix has the same number of rows and columns, if it's what's called a square matrix. Let me just remind you, in general about matrix multiplication. We can write down the product of two matrices. And we do that multiplication by taking the dot product of each row in the first matrix with each column in the second matrix. So here's the product of matrix A and matrix B. So there's the product.

If this matrix, if matrix A, is an m by k-- m rows by k columns-- and matrix B has k rows by n columns, then the product of those two matrices will have m by n rows and columns. And you can see that in order for matrix multiplication to work, the number of columns of the first matrix equal the number of rows in the second matrix. You can see that this k has to be the same for both matrices. Does that

make sense?

So, again, in order to compute this element right here, we take the dot product of the first row of A and the first column of B. That's just 1 times 4, is 4. Plus negative 2 times 7 is minus 14. Plus 0 times minus 1 is 0. Add those up and you get minus 10. So you get this number. You multiply this row dot product this row with this column and so on.

Notice, A times B is not equal to B times A. In fact, in cases of rectangular matrices, matrices that aren't square, you can't even do this, often do this, multiplication in a different order. Mathematically, it doesn't make sense.

So let's say that we have a matrix of vectors, and we want to take the dot product of each one of those vectors x with some other vector v. So let's just write that down. The way to do that is to say the answer here, the dot product of each one of those column vectors in our matrix with this other vector v we do by taking the transpose of v, which takes a column vector and turns it into a row vector. And we can now multiply that by our data matrix x by taking the dot product of v with that column of x. And that gives us a matrix.

So this matrix here, that vector is a one by two matrix. This is a two by three matrix. The product of those is a one by three matrix. Any questions about that? OK.

We can do this a different way. Notice that the result of this multiplication here is a row vector, y. We can do this a different way. We can take dot product. We can also compute this as y equals x transpose v. So here, we've taken the transpose of the data matrix times this column vector v. And again, we take the dot product of this, this with this, and that with that. And now we get a column vector that has the same entries that we had over here.

All right, so I'm just showing you different ways that you can manipulate a vector in a matrix to compute the dot product of elements of vectors within a data matrix and other vectors that you're interested in. All right, identity matrix. So when you're multiplying numbers together, the number one has the special property that you can multiply any real number by one and get the same number back.

You have the same kind of element in matrices. So is there a matrix that when

multiplied by A gives you A? And the answer is yes. It's called the identity matrix. So it's given by the symbol I, usually. A times I equals A. What does that matrix look like?

Again, the identity matrix looks like this. It's a square matrix that has ones along the diagonal and zero everywhere else. So you can see here that if you take an arbitrary vector x, multiplied by the identity matrix, you can see that this product is x1, x2 dotted into 1, 0, which gives you x1. x1, x2 dotted into 0, 1, gives you x2. And so the answer looks like that, which is just x. So the identity matrix times an arbitrary vector x gives you x back.

Another very useful application of linear algebra, linear algebra tools, is to solve systems of equations. So let me show you what that looks like. So let's say we want to solve a simple equation, ax equals c. So, in this case, how do you solve for x? Well, you're just going to divide both sides by a, right? So if you divide both sides by a, you get that x equals 1 over a times c.

So it turns out that there is a matrix equivalent of that, that allows you to solve systems of equations. So if you have a pair of equations-- x minus 2y equals 3 and 3x plus y equals 5-- you can write this down as a matrix equation, where you have a matrix 1, minus 2, 3, 1, which correspond to the coefficients of x and y in these equations. Times a vector xy is equal to 3, 5, another vector 3, 5.

So you can write this down as ax equals c-- that's kind of nice-- where this matrix A is given by these coefficients and this vector c is given by these terms on this side of the equation, on the right side of the equation. Now, how do we solve this? Well, can we just divide both sides of that matrix equation, that vector equation, by a? So division is not really defined for matrices, but we can use another trick. We can multiply both sides of this equation by something that makes the a go away.

And so that magical thing is called the inverse of A. So we take the inverse of matrix A, denoted by A with this superscript minus 1. And that's the standard notation for identifying the inverse. It has the property that A inverse times A equals the identity matrix. So you can sort of think about this as A equals the identity matrix over A. Anyway, don't really think of it like that.

So to solve this system of equations ax equals c, we multiply both sides by that A

inverse matrix. And so that looks like this-- A inverse A times x equals A inverse c. A inverse A is just what? The identity matrix times x equals A inverse c. And we just saw before that identity matrix times x is just x. All right, so there's the solution to this system of equations. All right, any questions about that?

So how do you find the inverse of a matrix? What is this A inverse? How do you get it in real life? So in real life, what you usually do is you would just use the matrix inverse function in Matlab. Because for any matrices other than a two-by-two, it's really annoying to get a matrix inverse. But for a two-by-two matrix, it's actually pretty easy. You can almost just get the answer by looking at the matrix and writing down the inverse. It looks like this.

The inverse of a two-by-two square matrix is just given by a slight reordering of the coefficients, of the entries of that matrix, divided by what's called the determinant of A. So what you do is you flip-- in a two-by-two matrix, you flip the A and the D, and then you multiply the diagonal elements by minus 1.

Now, what is this determinant? The determinant is given by a times d minus b times c. And you can prove that that actually is the inverse, because if we take this and multiply it by A, what you find when you multiply that out is that that's just equal to the identity matrix.

So a matrix has an inverse if and only if the determinant is not equal to zero. If the determinant is equal to zero, you can see that this thing blows up, and there's no inverse. We're going to spend a little bit of time later talking about what that means when a matrix has an inverse and what the determinant actually corresponds to in a matrix multiplication context.

If the determinant is equal to zero, we say that that matrix is singular. And in that case, you can't actually find an inverse, and you can't solve this equation right here, this system of equations. All right, so let's actually go through this example. So here's our equation, ax equals c. We're going to use the same matrix we had before and the same c.

The determinant is just the product of those minus the product of those, so 1 minus negative 6. So the determinant is 7. So there is an inverse of this matrix. And we can just write that down as follows. Again, we've flipped those two and multiplied those

by minus 1.

So we can solve for x just by taking that inverse times c, A inverse times c. And if you multiply that out, you see that there's the inverse. It's just a vector. That's it. That's how you solve a system of equations, all right? Any questions about that?

So this process of solving systems of equations and using matrices and their inverses to solve systems of equations is a very important concept that we're going to use over and over again. All right, let's turn to the topic of matrix transformations. All right, so you can see from this problem of solving this system of equations that that matrix A transformed a vector x into a vector c, OK?

So we have this vector x, which was 3/7 minus 4/7 a vector. When we multiplied that by A, we got another vector, c. And the vector A inverse transforms this vector c back into vector x, right? So we can take that vector c, multiply it by A inverse, and get back to x. Does that make sense?

So, in general, a matrix A maps a set of vectors in this whole space. So if you have a two-by-two vector, it maps a set of vectors in R2 onto a different set of vectors in R2. So you can take any vector here-- a vector from the origin into here-- multiply that vector by A, and it gives you a different vector. And if you multiply that other vector by A inverse, you go back to the original vector.

So this vector A implements some kind of transformation on this space of real numbers into a different space of real numbers, OK? And you can only do this inverse if the determinant of A is not equal to zero. So I just want to show you what different kinds of matrix transformations look like.

So let's start with the simplest matrix transformation-- the identity matrix. So if we take a vector x, multiply it by the identity matrix, you get another vector y, which is equal to x. So what we're going to do is we're going to kind of riff off of a theme here, and we're going to take slight perturbations of the identity matrix and see what that new matrix does to a set of input vectors, OK?

So let me show you how we're going to do that. We're going to take it the identity matrix 1, 0, 0, 1. And we're going to add a little perturbation to the diagonal elements. And we're going to see what that does to a set of input vectors. So let me

show you what we're doing here.

We have each one of these red dots. So what I did was I generated a bunch of random numbers in a 2D space. So this is a 2D space. And I just randomly selected a bunch of numbers, a bunch of points on that plane. And each one of those is an input vector x. And then I multiplied that vector times this slightly perturbed identity matrix.

And then I get a bunch of output vectors y. Input vectors x are the red dots. The output vectors y are the other end of this blue line. Does that make sense? So for every vector x, multiplying it by this matrix gives me another vector that's over here. Does that make sense?

So you can see that what this matrix does is it takes this space, this cloud of points, and stretches them equally in all directions. So it takes any vector and just makes it longer, stretches it out. No matter which direction it's pointing, it just makes that vector slightly longer. And here's that little bit of code that I used to generate those vectors.

OK, so let's take another example. Let's say that we take the identity matrix and we just add a little perturbation to one element of the identity matrix, OK? So what does that do? It stretches the vectors out in the x direction, but it doesn't do anything to the y direction. So the vector with a component in the x direction, the x component gets increased by an by a factor 1 plus delta. The components of each of these vectors in the y direction don't change, all right?

So we're going to take this cloud of points, and we're going to stretch it in the x direction. What about this matrix here? What's that going to do?

AUDIENCE: Stretch it in the y direction.

MICHALE FEE: Good. It's going to stretch it out in the y direction. Good. So that's kind of cute. And you can see that this earlier matrix that we looked at right here stretches in the x direction and stretches in the y direction. And that's why that cloud of vectors just stretched out equally in all directions.

Out this. What is that going to do?

AUDIENCE: It would stretch in the x direction and compress in the y direction

MICHALE FEE: Right. This perturbation here is making this component, the x component larger. This perturbation here-- and delta here is small. It's less than one. Here, it's making the y component smaller. And so what that looks like is the y component of each one of these vectors gets smaller. The x component gets larger. And so we're squeezing in one direction and stretching in the other direction.

Imagine we took a block of sponge and we grabbed it and stretched it out, and it gets skinny in this direction and stretches out in that direction. All right, that's kind of cool. What is this going to do? Here, I'm not making a small perturbation of this, but I'm flipping the sign of one of those. What happens there? What is that going to do?

AUDIENCE: [INAUDIBLE]

MICHALE FEE: Good. What do we call that? There's a term for it. What do you-- yeah, it's called a mirror reflection. So every point that's on this side of the origin gets reflected over to this side of the origin. And every point that's over here-- sorry, of this axis. Every point that's on this side of the y-axis gets reflected over to this side. So that's called a mirror reflection. What is this? What is that going to do? Abiba?

AUDIENCE: Reflect it [INAUDIBLE].

MICHALE FEE: Right. It's going to reflect it through the origin, like this. So every point that's over here, on one side of the origin, is going to reflect through to the other side. That's pretty neat. Inversion of the origin. OK? So we have symmetric perturbations in the x and y components of the identity matrix. We have a stretch transformation that stretches along one axis, but not the other.

Stretch around the other axis, the y-axis, but not the x-axis. Stretch along x and compression along y. Mirror reflection through the y-axis. Inversion through the origin. These are examples of diagonal matrices, OK? So the only thing we've done so far-- we've gotten all these really cool transformations, but the only thing we've done so far are change these two diagonal elements.

So there's a lot more crazy stuff to happen if we start messing with the other

components. Oh, and I should mention that we can invert any one of these transformations that we just did by finding the inverse of this matrix. The inverse of a diagonal matrix is very simple to calculate. It's just one over those diagonal elements.

All right, how about this? What is that going to do? Anybody? When you take a vector and you multiply it by that, what's going to happen? This part is going to give you the original vector back. This part is going to take a little bit of the y component and add it to the x component. So what does that do? That produces what's known as a shear.

So points up here, we're going to take a little bit of the y component and add it to the x component. So if something has a big y component, it's going to be shifted in x. If something has a negative y component, it's going to shift this way in x. If something has a positive y component, it's going to shift this way an x. And it's going to produce what's called a shear.

So we're pushing these points this way, pushing those points this way. Shear is very important in things like the flow of liquid. So when you have liquid flowing over a surface, you have forces, frictional forces to the liquid down here that prevent it from moving. Liquid up here moves more quickly, and it produces a shear in the pattern of velocity profiles. OK, that's pretty cool.

What about this? It's going to just produce a shear along the other direction. That's right. So now components that have a-- vectors that have a large x component acquire a negative projection in y. OK, what does this look like? It's pretty cool.

We're going to get some shear in this direction, get some shear in this direction. What's it going to do?

AUDIENCE: [INAUDIBLE]

MICHALE FEE: Good. Good guess. That's exactly right, produces a rotation. Not exactly a rotation, but very close. So that's how you actually produce a rotation. So notice, for small angles theta, these are close to one, so it's close to an identity matrix. These are close to zero, but this is negative and this is positive, or the other way around.

So if we have diagonals close to one and the off-diagonals one positive and one

negative, then that produces a rotation. That, formally, is a rotation matrix. Yes?

AUDIENCE: On the previous slide, is there a reason you chose to represent the delta on the x-axis as negative?

MICHALE FEE: No. It goes either way. So if you have a rotation angle that's positive, then this is negative and this is positive. If your rotation angle is the other sign, then this is positive and this is negative. So, for example, if we want to produce a 45-degree rotation, then we have 1, 1, minus 1, 1. And of course, all those things have a square root of 2, 1 over square root of 2, in them. And so that looks like this.

So if you have, let's say, theta equals 10 degrees, we can produce a 10-degree rotation of all the vectors. If theta is 25 degrees, you can see that the rotation is further. Theta 45, that's this case right here. You can see that you get a 45-degree rotation of all of those vectors around the origin. And if theta is 90 degrees, you can see that, OK? Pretty cool, right?

OK, what is the inverse of this rotation matrix? So if we have a rotation-- oh, and I just want to point out one more thing. In this formulation of the rotation matrix, positive angles correspond to rotating counterclockwise. Negative angles correspond to rotation in the clockwise direction, OK? So there's a big hint.

What is the inverse of our rotation matrix? If we have a rotation of 10 degrees this way, what is the inverse of that?

AUDIENCE: [INAUDIBLE]

MICHALE FEE: Right.

AUDIENCE: [INAUDIBLE]

MICHALE FEE: That's right. Remember, matrix multiplication implements a transformation. The inverse of that transformation just takes you back where you were. So if you have a rotation matrix that you implemented a 20-degree rotation in the plus direction, then the inverse of that is a 20-degree rotation in the minus direction. So the inverse of this matrix you can get just by putting in a minus sign into the theta.

And you can see that cosine of minus theta is just cosine of theta. But sine of minus

theta is negative sine of theta. So the inverse of this matrix is just this. You change the sign of those diagonals, which just makes the shear go in the opposite direction, right?

OK, so a rotation by angle plus theta followed by a rotation of angle minus theta puts everything back where it was. So rotation matrix phi of minus theta times phi of theta is equal to the identity matrix. So those two are inverses of each other. And the inverse of a-- notice that the inverse of this rotation matrix is also just the transpose of the rotation matrix.

All right, so what you can see is that these different cool transformations that these matrix multiplications can do are just examples of what our feed-forward network can do. Because the feed-m forward network just implements matrix multiplication. So this feed-forward network takes a set of vectors, a set of input vectors, and transforms them into a set of output vectors, all right?

And you can understand what that transformation does just by understanding the different kinds of transformations you can get from matrix multiplication. All right, we'll continue next time.