[SQUEAKING]

[CLICKING]

[RUSTLING]

[CLICKING]

[TAPPING]

PHILLIP ISOLA: Good. So welcome to today's lecture. And today, we're going to talk about representation learning. So we're moving into the next third of the class. So we covered basic approximation and generalization properties of MLPs. And then we talked about different architectures, transformers, and RNNs, and so forth.

And now we're going to come to a kind of a new perspective on deep learning, which is the idea of representation learning and generative modeling. And this is a perspective, which is associated mostly with deep neural networks, but actually could be construed to be broader than this.

You could do representation learning and generative modeling with other architectures that are maybe not neural networks, but most of the techniques have been developed intimately in conjunction with neural networks. And this field of representation learning usually goes hand in hand with deep learning.

So we'll talk about the idea that deep nets learn representations. What does that even mean? We'll ask, why should we learn representations? And then we'll look at one kind of representation learner called the autoencoder.

We'll look at the relationship to clustering and what's called vector quantization. And we'll look at the idea of learning representations via prediction and what's called self-supervised learning. And then, in the next lecture, we'll talk about a different type of representation learning, which is called metric learning or contrastive learning, where you learn about similarities and distances.

So this is going to be the new picture for deep learning for the next few weeks. We have data that comes into a system on the bottom. And then layer by layer, we change the changed the representation of that data, until we get to an output at the top. And we'll call the output an embedding or representation.

But actually, each layer is a representation of the data. So layer by layer, the deep net takes your raw data x and turns it into f of x, and then f of f of x, and so forth. And each layer is a different representation, so the data will be differently distributed. It'll be transformed somehow.

And the forward direction through a neural network will be called the encoding direction. So it's going from data to maybe simpler, better, more abstract representations. And that's the direction in which representation learning operates. It tries to find a good mapping from data to embeddings.

And the backwards direction is not going to be backprop. Now backprop tries to find the gradient. The backwards direction is more like the inverse of the forward direction in this context.

And that's called generative modeling. It's going from abstract representations and latent variables to data. And we'll talk about generative modeling in a few weeks.

OK. So these two things can roughly be thought of as inverses of each other. And we'll clarify that connection as we go. But today, we're just going to talk about the first direction going forward through a network and view that as learning representations.

And one label for this idea is data to vec, so x2vec. X is your data. And people have developed x2vec for all different modalities. So the most famous early version of x2vec was called word2vec.

You want to go from words in English or some other language to representations that are vectors. But you can do this for images. You can do this for sounds. You can do this for molecules, whatever you want.

And here's an example of an image. We take our data as input. And then, now, we're thinking, layer by layer, we get a vector representation or maybe a tensor representation of the data. And let's investigate what that representation is, what each of these is indicating a neuron or mathematically just a scalar dimension of a vector.

And we can investigate what those actually represent. So we'll talk about the layer 3 representation of this image is if I go through three layers of neural processing, three convolutional layers, for example, what is the tensor of activations I get out? That's the representation at layer 3. And then we can talk about other layers, too.

So the representations will be a vector or tensor of neural activations. And in this example, they might be representing that there's a car, or there's a ground, or so forth, a building, a road. So we're going to try to see how representations might map onto these semantic representations that we think about in our head. OK, question?

AUDIENCE: With this family of two vectors, are they feature extractions?

PHILLIP ISOLA: Feature extraction is the same-- is another name for the same thing, yeah. So x2vec can also be called extracting features from input data x, and the features in the form of vectors or tensors in this case. We have one more question.

AUDIENCE: Was x2vec, traditionally, only getting from bigger data sets to smaller representations? [INAUDIBLE]

PHILLIP ISOLA: So the question is, is x2vec always going to smaller representations, lower dimensional representations? Well, that is one of the classic ways of doing it. Often, the representation you want to be a smaller object, a lower dimensional vector than the input, but that's not the only way of doing it. You don't have to do it that way.

OK, the intuition here is-- I'll use these cartoons a few times over the next few weeks, or we'll use them in a few lectures. So the data will be this blue blob. It's some complicated object, high-dimensional, weird topology. That's like the space of images. It's this weird manifold.

And the representation will be this orange or reddish circle. And it's a circle because we're going to make the point that the representation is, oftentimes, a simpler space. So the distribution over embeddings is often going to be a very simple object.

In fact, it will typically be a Gaussian distribution. So Gaussian is a circle in high dimensions, or Gaussian in high dimensions looks like a circle

OK. So this is the picture where somehow learning embeddings are simpler. They have a simpler distribution than the data distribution. The input blob is like the data distribution. And we call the mapping from data to representation an encoder.

So I'm going to now show some visuals. And I want to introduce a different way of thinking about functions than what you might have learned about in grade school. So when you first encountered functions, we learned this way of representing them.

You have on the x-axis the input and on the y-axis the output to your function. But that's not really the only way. That's basically telling you, what is the mapping from Xin to Xout, or from x to y, traditionally. But in this class, we usually talk about Xin to Xout for some layer.

So this is just telling me that if I'm at some point on the x-axis, where do I map to on the y-axis? It's telling me something about the mapping. So we're just going to rotate the y-axis to be at the top. And now we'll look at this mapping as mapping a set of points on the input dimension to the set of points on the output dimension.

And the identity mapping now looks really simple, right? It's just straight lines. Nothing changed. So I don't know why we decided to go with this way of representing functions as opposed to this way, but I think both of them have some interesting properties and advantages in terms of clarifying what's going on.

But the second way will help us to visualize some of the mappings in neural networks. Actually, in some of the earlier slides, we already showed a few examples of this. So here are a few different neural layers for a scalar input to a scalar output. So this is the width of the network is 1, in this case.

And what does a linear layer do? A linear layer can-- sorry, if the linear layer has weight 2, it will just expand. It'll take your input data, and it will make it scaled up. So it'll just spread it out.

What does a ReLU do? OK, ReLU is kind of interesting. A ReLU takes all of your data in the negative half space and maps it to 0. So this ReLU has this property that the density of data that will get mapped to 0.

You'll usually have a spike of density at 0, which we'll see on the next slide. And the sigmoid will map most of your inputs to either 0 or 1, but in a soft way.

So I like thinking about functions this way because I think it helps to understand representations and distributional transformations. So think of your functions as taking a set, a distribution of input data points, and doing these geometric transformations, squishing them, and skewing them, and remapping them.

And so here are some of the functions that we've seen, some of the basic neural networks layers that we've seen in this class. So we have the linear, the ReLU, the normalization layers, softmax layers.

And activations are going to be colored in this red font. And parameters are colored in the blue font, just to clarify what is actually being learned. So the only learnable parameters are the weights and biases in these layers.

So here's the wiring diagrams. So let's look at the mapping diagrams for these layers, just to build up this intuition. So we're always mapping from Xin to Xout. Now one nice property of this other way of plotting functions is I don't have to only plot scalar to scalar functions. I can plot 2D to 2D functions.

I don't have to plot 3D to 3D. That's where it breaks down. But at least I can go to 2D to 2D, which you can't do with traditional plotting.

So here's the mapping. I'm showing in two different ways. On the left, I'm showing how each point on a grid of input values maps to the output. And on the right, I'm showing how each point in a Gaussian cloud over the input space maps to the output.

So 2D data goes in. And what does the linear layer do? It's going to be an affine transformation. And geometrically, it will just look like rotating, squishing, scaling, anything like this. So it's like a geometric transformation of the input data. That's all it's doing.

What does a ReLU do? So a ReLU looks like this funny shape. So there's two important properties to notice.

So one is the ReLU maps all data to the positive orthant, in this case. So in high dimensions, it's just going to be the subspace where all the dimensions are positive because anything that's negative gets mapped to 0.

And two is you can see how most of the points get mapped to these axes. You get this what's called a sparse representation. So ReLU has this property of making things sparser because most of the inputs or in each dimension, half of the values will tend to be negative and half will tend to be positive.

And the negative values will get mapped to the axes here. And there'll be a lot of density right at the origin because that will correspond to anything in this strictly negative orthant. And in high dimensions, I think this effect becomes even more extreme.

So what does L2 norm look like? And this is what I think I showed in one of the last lectures, but I didn't really fully explain it. What L2 norm will look like-- and same with RMSE norm, and same with LayerNorm, which is a variation on this-- is it will take all of your data, and it will map it to unitary vectors. It will map it to vectors that have norm 1.

And so in this particular case, with Euclidean norm, it will map it to points on the circle. And high dimensions will map it to points on the hypersphere. So that's what these layers do.

And that's nice for various reasons, but one of them is that we know that the numerics are going to be bounded somehow. The vectors will not go to infinity or not go to 0. They'll all get into a unitary range.

So what is softmax? Now I'm going to ask you, can anybody explain, why does softmax look like that? What is that line that's showing up in the softmax graph?

So I have input data, two-dimensional data, and I'm doing softmax of those two dimensions. Let's let me see if somebody else-- yeah?

AUDIENCE: x1 plus x2 equals [INAUDIBLE]

PHILLIP ISOLA: Yes, exactly. So the answer was that's a line that x1 plus x2 equals 1, or x plus y equals 1, if you think of one dimension as y and one dimension as x. And that's the simplex.

Is that the one dimensional simplex or the zero-- I think that's the one-dimensional simplex. I can't remember how to define simplex. But anyway, the simplex is the set of points in Rd, where the dimensions sum to 1.

And the output of a softmax is going to be a point in the simplex. So that's what that looks like. This is, basically, going to be your probability is high of one class, if you're at that point, and low for that class if you're at the other point.

So this is what neural networks are doing. Those are our basic layers. It's a little more complicated with attention and other kinds of layers, but these are the basic ones. So let's now look at what an MLP looks like plotted this way.

Let's see. This is a three-layer MLP because there's three linear layers, so our convention is three layers. And the input data is going to be a cloud of points at the origin with label red and a hemisphere of points around with label blue. So we're going to try to do a binary classification.

So this will be training with cross-entropy loss doing softmax regression. And we're going to look at what training actually looks like. So this is a non-linear classification problem because there's no linear hyperplane that separates the red cloud from the blue cloud.

So layer by layer, the representation of the distribution of data gets transformed. And what you're going to see happen is that the red points will move away from the blue points. And this is what it looks like.

So after a few steps of gradient descent, they start to spread out like this. And at this layer we're shifting things over. And then the ReLU snaps it back onto the axes. And then we skew things in a different direction. And eventually, we get things to spread out on a line. And then the softmax clamps that back down to this simplex.

And so the red points are all heading toward the 0.10. And the blue points are all heading toward the 0.01. And those are the one-hot labels for your data. So this optimization is actually working at segregating the blue points to be classified as class II and the red points to be classified as class 1.

So let's look at that animated. So this is what it looks like. This is every single layer has width 2. So we can plot everything here. There's nothing hidden.

It's not like I'm doing dimensionality reduction or visualizing. It's just the raw activation values at each of these layers. And each of the layers now can be understood as a different representation of the data distribution and a better and better representation for solving the class of segregating the red points from the blue points, which is the classification task.

OK. Oh, yeah, just for fun, I didn't really have a deep insight to take from this. But just for fun, you can compare the optimization using SGD, gradient descent, to the optimization using the spectral descent method that you worked out in your problem set 2.

So in the spectral descent method, you are normalizing the updates by their spectral norm. And you can see there's some interesting different dynamics. It's a little bit hard to read too much into this. Well, the spectral descent is descending faster, but that depends on the learning rate, so there's some caveats. But it is getting to a lower loss.

The loss is tiny, little font up here, is actually segregating the red and blue points better. So you can understand that different optimization schemes actually do perform differently. And there was some reason for working out what you worked out in your problem set.

I think something that might be interesting here is it looks like this first linear layer is almost like a rotation, which is a orthogonal transformation. And if your updates are orthogonalized, then maybe that somehow relates to this first layer finding this orthogonal transformation. It's just a rotation, but it's also scaling. I think it might be fun to see if you can get insights from this, but it's a little complicated.

So we can also try visualizing more advanced nets in this way. And this is where the power of the ideas of representation learning really come in. Not when I'm just having an MLP over two-dimensional embeddings, but when I am having a very high-dimensional feature space.

So this is the CLIP neural network. It's a computer vision system that is quite good at extracting good features from images. We'll talk about that a little bit later in the course.

And now the embeddings are not two-dimensional, but I am doing a dimensionality reduction technique, PCA, to show what they look like. So it's a bit of a cartoon rather than just showing you the real data.

But the point is that layer by layer-- and these are vision transformer layers or a block of three-vision transformer blocks. In the transforming, you have attention, and then LayerNorm, and then softmax, and then this and that. So this is an entire block of sequence of layers.

But what happens is as you go deeper and deeper into the network, you end up segregating these different colors. And the colors are corresponding to the class of images in a computer vision data set. So is it a cat? Is it a dog? And you see that at the top of the network, they're all very separated from each other. And that's because this network was trained for classification.

So it's trained to find a way to disentangle all of these different classes and have their representation in some high-dimensional space get all separated so that then, at the very end, I can just take a softmax of that. And they'll be cleanly separated to the vertices of the simplex, which is the one-hot labels for these classes. So it's a little bit hard to see because, again, it's doing PCA, so it's not actually showing the simplex. But it's something kind of like that.

So the whole point is just to build up the intuition of how we go from entangled, complicated data at the input to a neural network to, layer by layer, gradually morphing, disentangling it through these geometric transformations, until you get clean separation of your semantics of interest that can be read off into a prediction, for example, classification. And the output space is a simpler object than the input space. That's the point I'm trying to make.

OK. Cool. So any questions about the general idea of representation learning, and what these neural networks are doing, and what these visualizations represent? OK. Cool. So one of the initial inspirations for deep learning, of course, was to understand the human brain. And in neuroscience, people have studied how representations are learned in the brain.

So I wanted to make this connection. So this is a map of the visual cortex, a cartoon of the visual cortex of the human brain. And it goes through a sequence of layers.

The layers are just like layers in our neural networks. And they are linear followed by pointwise nonlinear operations, plus some extra complexities that biologists debate over whether or not these extra complexities are important or not. But for us, it's just linear ReLU, linear ReLU.

And in visual cortex, there's about six of these layers or five or six of these layers. And then there's a final layer called IT, where that's thought to be where the semantics are segregated, where object recognition has occurred. And so we have first layers extract, do filtering to find edges.

And the next layers look for conjunctions of edges, and disjunctions, and all kinds of combinations until you get pattern recognizers. And if you look at the neurons, and you poke into the brain, and you measure what they're responsive to, you'll see that as you go deeper into the brain, they become responsive selectively for more and more complex patterns.

So let's see, do deep nets artificial deep nets do the same thing? What do they learn internally? OK, so let's say I train a deep net to do classification of images. Now I'm going to try to probe and interpret the representation that's learned in order to perform that classification task. And we'll do it just like the neuroscientists do.

So we will stick artificial probes into the neurons, and measure their value as a function of the inputs, and see what inputs these neurons are responsive to. So in the brain, we would stick an electrode in and see when the neuron fires. That's when the ReLU is above 0.

But it might not be exactly ReLU in the brain. It might be some other threshold function. And we're going to do the same thing. So for which inputs does the neuron turn on? And for which inputs does it turn off?

So people have been doing this for a while. One of the modern names for this type of work is interpretability or mechanistic interpretability sometimes. So it's a really interesting area if you've come across those terms, but it's an old problem.

And I'll show you some results from this Zeiler and Fergus 2014 paper on trying to interpret the units within a CNN. OK. So the first layer in the CNN that they were working with, this was just a vanilla convolutional network with five or six layers.

Well, the first layer has a set of filters. So the filters will all output a feature map. So each filter will output some spatially varying set of activations. And we can look at what input patches in the image are most strongly activating one of the filters to fire. So what is the filter looking for?

So here are nine filters in this montage of nine subgrids. And on each of these sub areas, we have the top nine input patches that most activated that particular filter. So this is channel 1 filter. And what do you think it's doing? So what is this a filter for, this top left one?

AUDIENCE: [INAUDIBLE]

PHILLIP ISOLA: Yeah, it's like an edge detector. It's looking for an oriented line, a bright followed by a dark transition, so, right, filter. What is the idea of a filter? A filter is trying to filter out something from the data, find the thing it's looking for and throw away everything else.

That's what the word filter means, the English word filter. So that's what it's doing. It's just it's responding to some things and not to other things. And what it responds to is edges.

OK. And then we have other-- this is a different orientation of the edge. And this is like a greenness detector. It fires selectively when it sees green. OK. So it looks like at the first layer of processing in a deep neural net, you get these simple feature detectors

So the next layer, what is that going to look like? So it turns out not so different than what the neuroscientists came up with when they were studying animal brains. So the second layer looks like it's having neurons filters that are selective not for edges, but for some conjunction of edges, higher-order patterns.

So if I have two channels that detect two different orientations of edges, the next layer of my network could look for a conjunction of those two orientations, like a cross. And that turns out to map to maybe even more complicated things, like gradients, or circles, and so forth. So then as we go layer by layer, we're going to get more and more complex pattern detectors.

So here are the image patches that most strongly activate nine different filters on layer 3 of this network. And already you can start to see, just layer 3, we get selectivity for faces right. And it's not because it's really recognizing humans necessarily, but maybe it's because it's recognizing little circle of darkness around a bright background with a broader torso thing underneath it, so some blobbiness detector. But it happens to be an OK template for a human.

And then as we go deeper, we get much more selectivity. And eventually, we start actually detecting, selectively segregating dog faces from human faces, for example. And again, this is 2014. This is the fifth layer of a convolutional network from that era.

If you run the same types of methods on modern networks, which are dozens or hundreds of layers, you will get very precise detectors deep in the network that code for very specific things. And that's this area of mechanistic interpretability, where people are trying to understand, what are the emergent detectors in these systems? And you can do it for images, or language models, or other things as well.

We can say, is there a neuron within a language model that codes for the sentiment? There was a famous paper called "The Sentiment Neuron" that found a neuron that codes for sentiment. But now this is well known, that this just occurs in all of our modern deep nets.

OK. So the rough story is that deep nets and the neuroscience models of the brain are quite in alignment here. The first layer of the human visual system or monkey visual system, to be more precise, will have these edge detectors. And the next layer will have shape and pattern detectors. And then, finally, at IT at the sixth layer or so, we'll have dog face recognition.

One difference-- so there are some differences between brains and machines. One interesting difference is that the brain seems to only have about seven-- it depends how you count, but maybe like seven layers of convolutional filters. And our modern architectures have hundreds of layers of filters and other operations. But the brain has recurrence, so it uses those seven layers over and over again.

So let's now get a little bit more formal and define what a representation is for the context of this class. So we're going to, basically, restrict ourselves to representation as a vector embedding, is a vector. Or it could be a more general tensor object. It could have multiple dimensions, but most canonically, it will be a vector.

And there's two different ways we're going to use the word representation. So the first way is we'll say a representation of some data domain x or maybe a data distribution p of x, probability of x, is a function that maps from data in that domain to embeddings in Rd, vector embeddings.

That function f is called the encoder. And it will assign a feature vector to every element in the input domain. And we'll say that function f, that's the representation of that data domain.

So what is actually parameterizing f? It's the weights and biases. So if I have a neural network, the representation, by this definition, is its weights and biases. That's a learned representation of the data distribution.

Another use of the word representation will be the representation not of the domain or the distribution, but of a single point. And that will just be a vector z in Rd, where z is equal to the encoder applied to f. So people use both of these definitions of representation. If you want to be clear, you can say the representation of a data point versus the representation of a data domain.

But I wanted to point out that distinction because we'll talk about the learned representation. And when we say the learned representation, we usually mean f, OK? So that's the learned representation is the weights and biases that fit the data.

So that's the formalism we'll work with for the next few lectures. So why learn representations? So there's a lot of answers you could come up with.

But maybe the simplest is to do more learning, right? So we learn to learn. It's interesting. It seems a little bit circular, but this is usually the context in which we're going to use representations.

We're going to use them to accelerate future learning. We'll have a few lectures on transfer learning, where we get into more details. But I have to introduce a little bit of this now because the main use of trained representations is to accelerate future learning. So quote in Goodfellow, "Generally speaking, a good"-- or the *Deep Learning* book-- "a good representation is one that makes subsequent learning easier."

And I think this was a strange misconception in the early days of deep learning. It was that we will make deep networks that act as blank slates. They come at new problems with no pretrained prior information.

And people would complain about deep learning as being very data hungry. It would take massive data to apply deep net to your problem. And they would say, well, humans are much more sample efficient. We only take a little bit of data to solve a new problem But it wasn't apples to apples.

So humans are born. And you in this class are coming to learn new information not as a blank slate. You've gone through billions of years of evolution, tens of billions of years of the physical evolution of the universe before that, before even biological evolution. And you're also having 20 plus years of pre-training in your lifetime.

You've learned very good internal skills, and knowledge, and representations, so you're not blank slates. And deep nets also should generally not be used as blank slates. This is something that's changed over the last decade.

It used to be we thought deep nets require a lot of data because we'll start from scratch. We'll initialize them randomly, and they'll be blank slates. But these days, this is not really what we do. We start with pretrained representations.

So let's look at that idea. So very often, we will have-- let's imagine that you're working for some music company at Spotify or something. And you will train your neural network to classify the genre of the music.

So in two parts of your network, you'll have some encoder f, and then you might have some extra readout operation at the end to do your classification. We'll think of w as just being like a linear layer on top of a representation z. But we can actually select any layer in the network to be called the representation, any prior layers are the encoder, and any subsequent layers are like the readout, OK?

But often, what we're going to be tested on is not going to be the same as what we've trained on. Maybe we invested $1 billion to train the Spotify sound recognition system. We don't want to use $1 billion to solve a new task because maybe the next year, we realize, actually, we also need to predict if users the music or don't like the music, so we can give them recommendations.

So the trick is that we will just use our same encoder up to some layer in the network, and we will only retrain the final readout. And that might, in this case, be a linear operation, which would be called a linear probe, or it could be MLP, or something else.

So we're not going to retrain from scratch. We're going to initialize our network with the pre-trained representation. And in this case, we might just freeze that pre-trained representation, no longer update it, and just learn a linear mapping onto our new prediction problem.

So these are the two phases. Now they have new names for these, pre-training and post-training. But we used to call it-- there's a lot of different names for this, but you pre-train and then you post-train.

So another thing you can do is not just train this read-off head. You can actually update the parameters of your encoder, as well, but initialize from the prior f. So you just continue to do backprop to update to find some fine-tuned perturbation of your parameters that will solve the new problem.

So this is called fine-tuning. I'll talk about that on the next slide as well. So we initialize F prime not with random weights, but we initialize it with the same weights as f, and then we keep training. That's the whole idea of fine-tuning.

And here's the transfer learning paradigm. This is what is typically done for real-world problems. For toy problems, you can train from scratch. But for real problems, you generally can't.

So you have three phases. You have now not just train and test. You have pre-train, which can be thought of as learning a representation. You do that on a lot of data. And typically, you can do that on a lot of data. You can afford to do that on a lot of data because the pre-training phase, the representation learning phase, doesn't require as much knowledge about what the final task is going to be.

We'll see that autoencoders and contrastive learning are these pre-training methods that don't require knowing what the final task is going to be. But then you adapt to your actual task of interest. And you can do that on just a little data because you already have learned a good representation of the domain f, of the data domain x through this function f. F is the representation. And then you test right. That's just regular. You deploy your system out in the real world.

So I think a lot of you have probably done this because this is just a practical thing. If you're ever going to play with deep nets, you'll download a pre-trained system, and you'll fine-tune it, right? So a lot of you I'm sure, have done this, but it's worth understanding why and how it works.

So fine-tuning-- extremely simple recipe. Pre-train a network on task A resulting in parameters w and b, weights and biases. Those weights and biases are your pre-trained representation of the data domain.

Then you initialize a second network using some or all of the weights and biases. You could initialize some weights from scratch, some from your pre-trained weights. You can do all kinds of interesting network surgery on that.

And then you'll train the second network to resolve some parameters, W prime and beta prime. And this is often called fine-tuning because we assume the W prime is just a minor modification of W, because we only fine-tune it on a little bit of data.

So the point here is a lot of people think of deep learning as the thing you do when you have a ton of data. But I think the real point of deep learning is it's the thing you do that enables learning from little data. So deep nets and deep learning, to me, is really all about how to learn from little data.

And the way you learn from little data is by pre-training on massive data. So, yes, you have to pre-train on massive data, but once you do, you get a very sample efficient method, which is just update the representation a little bit to solve the new task. And this is the best known way right now for a lot of problems to do learning from little data.

To do, learning from little data, you have to pre-train on massive data. That's a little trick that happened that people didn't quite expect. I think for a while, people thought there might be some algorithm that's just better at learning. You don't have to pre-train that algorithm, but this works better. Yeah, question in the back?

AUDIENCE: When you talk about massive data versus little data, roughly, what's the scale or magnitude of the amount?

PHILLIP ISOLA: Yeah, so what's the scale for massive, and what's the scale for little? So in practice, the ratio is always becoming more extreme, I would say. But there's not going to be a precise answer.

But I would say, in practice, each year, the pre-trained representations get trained on more and more data. So that will be a language model trained on 10 trillion tokens or words on the internet. And the fine-tuning is typically something that you can do on CoLab yourself in a few minutes or hours.

Now it depends what kind of fine-tuning you're doing. So we'll talk about linear probes and low-rank fine-tuning methods, but that will come in a future lecture. But it would be a trillion pre-training tokens and a million or less fine-tuning tokens, so the ratio is many orders of magnitude. There's a question in the way back.

AUDIENCE: Yeah, for a lot of applications, there are, obviously, pre-trained models on specific types of data, like language and images. But let's say you're in some scientific field and there are not like [INAUDIBLE] models that are not trained [INAUDIBLE] use of, for example, [INAUDIBLE]

PHILLIP ISOLA: Yeah, very good question. So if I can rephrase the question, what if you're in a domain where you don't have a lot of pre-training data for that domain? What if you are in a scientific domain where it's very specialized measurements? Are you allowed to use a language models pre-training that's trained on internet talking and text?

And the empirical answer is, yes. That often actually works decently well. So you don't have to pre-train on the same type of data or even the same data distribution as you're going to fine-tune on. There's often a lot of shared information between one data modality and one distribution and other ones that might look very disparate, but your mileage may vary.

If you have a really, really big gap between the type of data you pre-train on and the type you fine-tune on, then it might not work as well. But I would say, in practice, the surprise has been if you fine-tune a language model trained on the internet on almost anything, it will help. That's kind of been the current finding.

It might not be the best way of doing things, but it will be better than initializing the network from scratch. Yeah, question?

AUDIENCE: Yeah, I want to ask, what's the theory behind this fine-tune framework? There is no guarantee for similarity between task A or task B. And what's the guarantee for this kind of paradigm?

PHILLIP ISOLA: So the question is, I think, what is the theoretical guarantees about fine-tuning? So the big question in this area and transfer learning-- we'll have more on this later-- but the big question is, when and why does training on task A on date A help on task B and data B? What is the property of A and B that has to be alike for this to actually work?

So empirically, it just generally works pretty well, more than people maybe thought. Theoretically, I would say it's very much an open question. Maybe I'll point you to Sanjeev Arora has some papers and some talks that phrase it the way I just gave it. When does training on task A help on task B? And he tries to develop some theory out of this, but I would say it's in its early days.

OK. Let me move on. We'll come back to question more questions later. So that's really just all motivation setting up the concept, why we actually care about this. But now let's get into a few algorithms and see how representation learning actually can work.

So just, first, what are properties of a good representation? The optional reading for today gives some ideas. So let me give two or three, and then I'll ask if you have any others. So what are properties of good representation? What are we looking for?

So I said that I can pre-train somehow, but how should I pre-train to get the best representation? What are the properties I'm looking for? So one property is that you often want your representation to be somehow small.

You want it to be something that is compact so that-- there's two reasons for this. One is that this will just use less memory. You can store low-dimensional, compact features on your disk with less memory than the raw data itself. That's the most obvious one.

And the other one is an Occam's razor generalization theory type of analysis, where you would say that the more compact representations will be somehow simpler and generalize better, or learn more abstractions that generalize better. But you don't just want your representation to be compact.

You also want it to be explanatory. And one of the technical terms for this is sufficient. You want minimal and sufficient representations.

Minimal, they're very low-dimensional or low-information, but sufficient, that they are sufficient statistics for solving your tasks. They actually are informative toward the tasks of interest that you care about.

So this cartoon on the right is meant to be like a minimal sufficient representation for semantic understanding of a scene. Just three things and where they are, but not all the weird, photometric details that might not actually matter for driving a car or for the semantic problems of interest.

Another one might be that your representation has disentangled the interesting factors of variation in your data into independent dimensions, so axis-aligned representation. We'll talk a little bit about that. I have two more, but I want to query the class now.

What are your ideas? What would you want in a good representation of data? What else is there besides this, minimal, compact, sufficient, disentangled? Yeah? Let's go here.

AUDIENCE: Somehow human interpretable?

PHILLIP ISOLA: Yes, interpretable. You got exactly the one I was thinking of next. So, yeah, there's other considerations. We're going to have to interface with these representations. OK, another one?

AUDIENCE: Similar dimensional size, each dimension, [INAUDIBLE] something.

PHILLIP ISOLA: Yeah, you maybe want your vector embeddings to be unit variance or have some nice numerical properties. That's a good one. I didn't think of that.

AUDIENCE: [INAUDIBLE] between explanatory and interpretable. It seems similar.

PHILLIP ISOLA: Explanatory and interpretable-- by explanatory, I meant that it is explanatory toward the automated tasks of interest, like the machine will be able to use that representation to solve problems and make accurate predictions. By interpretable, I meant a human can use that representation to solve problems and understand things. They're definitely related, but that was the distinction I was trying to make. Back here?

AUDIENCE: Context aware.

PHILLIP ISOLA: Context aware, yeah. You want your representation to know as much as possible about the universe when it's making a decision. Another one?

AUDIENCE: Something similar, I was just going to say resilient.

PHILLIP ISOLA: Resilient, robust, yeah. So you want representations that are not going to be fragile and vary a lot. When the input data changes, you want the representation to be stable. And you don't want it to be you can just add a little noise to the data and, suddenly, the representation will go crazy.

And so those are all good. I just had this one final-- yeah, good representation, fundamentally, it's just something that makes subsequent problem-solving easy. Think of the 4DA transform. It's a representation of data that makes convolution really easy.

It makes convolution just into a product. So that's a classic representation, but all representations have this property or should have this property. You can think of more.

So how do we learn representations? So one way is you just train your neural network on whatever task you want, like music prediction, and you hope that it learned a good representation. And that's OK. That works sometimes. That's just like supervised learning. And it will find an f, and maybe that f is a good representation.

But we're now going to talk about methods that don't target some supervised task, they just try to learn good representations generically. And we sometimes call this unsupervised learning or self-supervised learning, where we have input data, which is not xy pairs, but it's just x. There's no target, but I want to still learn a good representation.

And so what would a good representation be? So maybe it's some embedding vectors. I said that's the main one we'll focus on. But there's other things it could be, too.

It could be clusters. It could be distance metric. You could learn a proper distance metric for your data domain x.

A metric is a function of x1 and x2, so it's a bivariate function as opposed to an embedding, which is a univariate function. It just is a function of x. We'll talk about metrics in the next lectures.

So we're going to talk about the vector embeddings. And in my opinion, there's two general principles for how to learn a good vector embedding without having an explicit supervised task. So the one principle is compression, just find a way to find a good compression of your data.

And the other principle is prediction, be able to predict missing data when you hold that data out. We'll talk about both of these. And in the reading, there's a catalog of some of the methods we'll see and how at least, in my view, they can all be understood as compression or prediction.

And if you want to, you can actually even see the compression prediction as fundamentally the same. I'm not sure if there's really a difference, but at least it's useful, intuitively, to think about compression versus prediction. So let's see both of these. That's a question for you to think about. Is there actually a connection between compression and prediction?

So learning via compression will work as follows. We'll take our data, and we'll try to find that compact mental representation. But what is that going to actually do mechanistically?

So we're going to take our data. We're going to find a vector embedding, and then we're going to find a lower-dimensional vector embedding and a lower-dimensional vector embedding. And we'll just do that by constructing a neural network, where the width decreases as a function of depth. So each new layer will be lower width.

So that is all fine. I'll learn a low dimensional compressed representation, but I also need it to be sufficient in explanatory of the data. I don't want to just map my input to 0, right? I want to map it to something that actually still can reconstruct the data.

So we call that embedding vector-- canonically, we'll call it z. And we also want to be able to decode the image from the compressed representation. I'm using a lot of image examples, just because I like images. I often work on images. But think of image as just a placeholder for whatever data you want to put there, whatever domain you care about.

So what is this thing called? It's called an autoencoder. And this is representation learning algorithm 101, just the most basic vanilla one, and also my favorite, and probably the best one. And I think it will just win out in the end. So many things in machine learning can be viewed as some kind of autoencoder or some variation on an autoencoder. So this is a very fundamental concept.

Just map the data to a simpler form such that you can decode the original data from that simpler form. Autoencoders is just such a beautiful idea. This is not the original reference for autoencoders. But Hinton, who just won the physics Nobel Prize, of course, was one of the people that really helped develop autoencoders in his early work.

I'm not sure there's a single inventor of autoencoders. That would probably be-- yeah, it's an idea that's existed for centuries in some form or another. So that's an autoencoder. So let's look at it mathematically.

So we have our input data distribution, which is all complicated. We map it to a compressed, lower-dimensional representation. So I'm drawing it as a circle because it's a smaller object. It's a lower-dimensional thing.

It might not actually be like a Gaussian object. In order for it to actually be a Gaussian circle, we'll have to do something called variational autoencoders. We'll come to that later.

But just keep in mind that autoencoders do have a way of actually mapping to the circle. But we'll talk about just vanilla autoencoders for now. And then we'll train the decoder, g, which will predict the data in such a way that we minimize the reconstruction error between the reconstructed predicted data and the input data.

So here's our objective function. We're just going to say that encode x into a lower-dimensional format, decode it back to the original dimensionality should be an identity. G of f should be an identity. And that should just be reconstructing the raw data x in expectation over the training set. So that's an autoencoder.

So here's a learning diagram of what the autoencoder objective and hypothesis spaces. So the critical thing to note here is that the autoencoder objective just says, I'm going to define big F as being the composition of the decoder and the encoder. So it's encode and then decode becomes this one big function F.

So the funny thing about the autoencoder is that this big function F should ideally just be an identity. It should do nothing. So the objective is trivial. The objective is you just don't do anything to your data, just learn identity.

If I had no constraints on the architecture, on the hypothesis space, the autoencoder would be a trivial thing to learn, and it would have no utility. It would just spit out an exact copy of the data. But the key thing is the autoencoder works because you put constraints over the architecture. In our formal language, we'll call it the hypothesis space.

So the key thing is that we're mapping from a high-dimensional space to a lower-dimensional embedding and back. So typically, m should be less than n. Now there are variations on autoencoders where you don't need that. You can impose simplicity on the embeddings in other ways than dimensionality reduction.

But for vanilla autoencoders, m is less than n. And so for an image, it looks like this. We have a low-dimensional representation z. So question for the audience.

What if f and g are both linear? So I learn a linear transformation to a low-dimensional space, and then I learn a linear transformation that tries to invert that. What do you think that is going to result in?

Does anybody know? It's a model that I've encountered in previous machine learning or statistics. Yeah?

AUDIENCE: So will it be another linear?

PHILLIP ISOLA: That will be another linear function. Yeah, that's right. OK, yeah, that's one good response. Any others? Let's go over here.

AUDIENCE: [INAUDIBLE] the equivalent of PCA, but--

PHILLIP ISOLA: Yeah, equivalent of PCA. OK, right. So you're on to this. What is PCA doing? PCA can be understood as trying to maximize the variance I'm capturing in the signal via some linear orthogonal transformation of the vector.

And if I'm maximizing the variance, I'm able to best reconstruct. And that actually is exactly equivalent to the L2 reconstruction objective. So PCA can be understood as doing essentially the same thing.

So if they're both linear, we'll just replace it with the encoders of matrix W. The decoder is a matrix Wg, and the encoders matrix Wf. And we're trying to say, if I pass encode, decode, I will obtain my original data matrix again. That's the objective of an autoencoder written with linear f and g.

And PCA is a variant of this, where we assume that the encoder and decoder are the same matrix W. And they have this property that it's an orthogonal matrix, so that's the constraint. It's a slightly different setting, but it's almost the same.

And so this is the objective that PCA tries to minimize. It tries to find W such that this transformation will result in decoding back to the original data. So you can work through the math.

If you're more familiar with PCA maximize the variance in the projected space, then you can work out that this thing under this condition is equivalent to minimize the variance in x minus the variance in the transformed version of x. And then we can maximize and change the sign to be plus instead of minus. And then we can remove this term because this term doesn't depend on W, and we're maximizing over w.

And then this is just maximize the variance captured in the reconstructions. And that might be more familiar PCA that you're used to. And what can be proven is that the embeddings learned by a linear autoencoder span the same subspace as embeddings learned by PCA. So this is the rough the rough argument there.

So vanilla, most intro statistical model that you can think of, the most standard one that's been around for hundreds of years, I believe, or quite a long time, PCA. Well, an autoencoder, the very best representation learning method in the family of reconstruction based representation learning is just nonlinear PCA. That's a generalization of PCA to nonlinear representations.

By the way, I was trying to work out these equations. And I found that ChatGPT has gotten incredibly good at this. So if you're a little bit confused about-- I didn't put in every single step here.

But if you're a little confused, asked ChatGPT, what's the connection between PCA and linear autoencoders? And it will just give this wonderful derivation of everything. I think it's correct. I did check. I thought about it.

But my point is that these tools are getting a lot better. So if you're confused about a concept in this class, not a homework question, but a concept, just ask ChatGPT, or Claude, or whatever language model you prefer. They're really good at explaining concepts.

Anyway, of course, they can be wrong. You have to be very aware of that. But I'm just finding it to be quite a nice resource for teaching. And I'd recommend it for students, too.

So let's do a little experiment now. Are autoencoder is actually learning good representations? Or it sounds nice. I learn a low-dimensional, compressed representation. But is there any point to doing this besides just compression and saving memory?

OK, obviously, I save memory by doing this. I have a smaller object to put on my disk, but do I get any other advantage out? We said that we think we might get representation learning, where we actually get a meaningful representation, so do we?

So here's an experiment. You'll do some of these experiments on this same type of data on your p set 3. So it's a data set of colored shapes. We have three different shapes-- triangle, circle, and square. And we have nine or so different colors. And we're going to try to train an autoencoder on this data.

So we'll encode some x, and we'll visualize. To understand the representation, we're going to do this particular probe called a nearest neighbor probe. We're going to look in the representation space, so we're going to embed x into z. And then we're going to look at the nearest neighbors to z.

We're going to look at the images whose z vectors are the nearest neighbors to our query image. So in the representation space z, the nearest neighbors to this triangle are going to be these other triangles that are roughly the same color. And the nearest neighbor to this square will be blue squares.

So what is this saying? This is saying that our representation space has organized the data in a way that seems meaningful and seems to match human perception. So nearby embeddings in representation space, actually, are similar images, according to their color and their shape. So it looks like the autoencoder did learn a meaningful representation that encodes properties of both color and shape.

Another thing that we can do is we can look at where that representation becomes effective. So we can now ask-- we'll measure the quality of the nearest neighbors by using them to do classification. So we'll do a KNN or, actually, one nearest-neighbor classifier.

So what is a one nearest-neighbor classifier? That just says, is my first nearest neighbor in a data set the same class as the query? So is it the same color class, and is it the same shape class?

And I'm plotting accuracy of the one nearest neighbor classifier for the shape class and the color class, and I'm plotting that at every layer of a encoder. This is a convolutional encoder applied to this data. So what's going on here?

The accuracy at shape is getting better, which kind of makes sense. I'm learning a better and better representation so that similar shapes are embedded near each other in representation space. Why is the accuracy at color getting worse? Anyone have an idea? Yeah?

AUDIENCE: In a very crude way, it's perhaps learning the principal components of shape and separating it from the principal component of color.

PHILLIP ISOLA: Yeah, so it's learning something like the one principal component dimension is shape and the other is maybe color. That would be if I had the linear autoencoder, that might be exactly what it does. This one's non-linear, but it's going to be something kind of like that.

So anyone have an idea for why color did not improve as I go deeper into the network? I mean, it got worse. But let's just ask the question, why didn't it get better? Yeah?

AUDIENCE: You don't need that much stuff to--

PHILLIP ISOLA: You don't need that much stuff to decode color. That's the answer? OK, right. So in pixel space, pixel space is a representation of the data. It's just the raw format that we take the data in. And color is very superficial and explicit in pixel space.

And the nearest-neighbor image in pixel space to this query is going to have the same color. Color is explicitly represented in that space. Shape is not explicitly represented in pixel space.

If I do L2 distance between two images represented as pixels, just a vector of pixels, it won't put similar shapes near each other, but it will put similar colors near each other. You can think about why that is, but this is the general-- the general point is that every representation is good at some things and bad at other things, and there's trade-offs.

So in this case, the pixels are very good representation of color, but a bad representation of shape. And the embeddings are a good representation of shape, but a worse representation of color, in fact, because the pixels were already, I guess, the best you could do in this case.

So autoencoding doesn't strictly result in better representations. It results in different representations. That's a fundamental principle. All representation learning is making trade-offs to reformat the data into a way that makes some tasks easier, shape classification and other tasks harder, color classification in this example.

So I said autoencoders are the deep learning version of PCA. And now I'm going to say that another model is the deep learning version of k-means. So PCA and k-means are the two most important models, and everything else is just a generalization of them. That's just some opinionated statement. But that's one way of thinking about it.

So let's look at clustering now. So clustering is the problem of taking data and assigning each data point to a cluster. So a representation-learning lens on clustering is you're learning an encoder that doesn't output a vector. It outputs an integer. So for every data point, I output an integer, which is the cluster assignment. It still is an encoder. It's still representation learning.

So at inference time, if I just apply my learned encoder that outputs integers to some data points, it will tell me, this is the third class. And I'll color that red. And that has identified clusters So clustering is learning an encoder to integers.

So why is clustering a good representation? So from the representation-learning angle, clustering is learning a function f that outputs an integer, will represent the integers of one-hot code, in this example here. So we can think of it as a vector embedding, but it's just a one-hot vector embedding that's isomorphic with the integers, for a finite set of integers.

So. what's the best representation that-- this is another subjective statement. What's the best representation that humans have come up with so far? What do you think? What's the best representation that the human brain has discovered? Yeah?

AUDIENCE: Words.

PHILLIP ISOLA: Words, OK. That was my answer. You can disagree. I don't know if this is true. Traditionally, I come from a computer vision background, and so this is kind of blasphemy, I suppose. But I think language is just the best representation of the world that humans have discovered.

There's other representations that are good and have their trade-offs, but language is like the pinnacle. And words are the atoms of language. There's grammar and there's structures between words, but words are the important component of language.

And clustering is the problem of making up new words for things. That's one way of understanding it, right? A word is like a symbol that at least-- there might be some linguists that would quibble about this, but one rough definition of word is it's just like it is a symbol that denotes a set, and that's what clustering is doing.

So this is like saying, oh, I will map every image like this to this cluster, and I can just label it arbitrarily with a word. Language has more structure beyond the words, but this is but words are an important part of it. So clustering is the representation learning problem of finding new words for concepts.

Let's talk about k-means. So this should be review. I hope you've seen PCA and k-means before. So if you haven't, go back and review those. Because as I said, deep learning is just generalizing those, representation learning is generalizing these concepts.

So what does k-means do? k-means finds k different clusters in your data. And each cluster is represented with what's called the mean, which is just going to be the average of all the data points in that cluster. So it maps data points to integers. And it does that in such a way that each data point is as close as possible to the mean of the data points in the cluster it is assigned to.

So here's the representation learning view of k-means. I'm going to train an encoder that will output one-hot codes. And then my decoder will just be-- think of it like a lookup table. It takes in a one-hot code, and it outputs a vector. And it's trying to output the vector that will reconstruct the input as best as possible.

If I only can output a single vector for every item in a cluster, every item in the cluster goes to the same one-hot code. So now I can decode that into just a single vector. Think of g as a matrix W applied to a one-hot code. It selects a row of that matrix.

What is going to be the decoder that minimizes the Euclidean distance reconstruction error, the L2 reconstruction error? What is the name for the vector that minimizes the distance to all points in a set, the L2 distance? Yeah?

AUDIENCE: Mean.

PHILLIP ISOLA: The mean, OK. I think a lot of that, maybe some of you don't. But, yes, the vector that minimizes the L2 distance to a set of other points is the mean of those other points. You can show that.

So that means that k-means is an L2 autoencoder. The only difference from the other autoencoder that I showed you is that the hypothesis space doesn't have a low-dimensional bottleneck. It has an integer bottleneck, OK?

So that's just a mapping of k-means onto autoencoders. They're the same model with a different hypothesis space. k-means, this hypothesis space is not differentiable. It has some properties that can be exploited, some structure that can be exploited.

So we optimize it with a different algorithm than SGD. And that's where you might have encountered the expectation maximization method for doing k-means and so forth. And, yeah, that's just because this problem has some special structure.

And then the deep learning version of k-means is called a vector quantized autoencoder. So a vector quantized autoencoder is exactly the same as k-means, except that f is non-linear instead of linear. And there's a whole bunch of variations on VQ models. There's VQGAN, VQVAE, et cetera, et cetera. These have bells and whistles, but this is the gist of it.

I'm trying to learn an encoding into a set of integers and a decoding that minimizes reconstruction error. And f will be a deep network, or little f in little g are both going to be deep neural networks. So you can read up on exactly how that method works, but it's just k-means with deep nets. Yeah?

AUDIENCE: In this k-means analogy, does the autoencoder learn to find the best k?

PHILLIP ISOLA: No. So does the autoencoder learn to find the best k's? So k is a hyperparameter and, typically, the user has to define it. Yeah, you can have methods that try to automatically determine k, but that's beyond the scope of these vanilla methods.

So data compression-- that was the first way of learning representations. And it shows up in PCA, and k-means, and VQVAEs, and VQGANs, and autoencoders, and so forth. There's another thing you can do, and that will be the prediction method.

So now rather than learning representations by compressing data, we're going to try to learn representations by predicting held-out data. And this is actually the kind I started this lecture with. I said that we could just pre-train a network on music classification. That's a prediction problem, and it induces an OK representation for music classification.

But we're instead going to do something else, which is we'll say, we didn't want labels. We don't want to be we don't want a representation that is tied to a specific task. So we don't want to have to have humans label some narrow task. We do something much more generic that doesn't require annotations or task specificity.

And one way to do that is we say, we're not going to predict labels. We're going to predict the raw data itself. And this is called self-supervised learning. So it's called self-supervised learning because it's using the machinery of supervised learning, meaning predict y from x, except that we define y as being some part of the raw data as opposed to some label.

So this is like an autoencoder, except I'm predicting half of the data from the other half of the data. And interestingly, this works really well. This tends to work a lot better than autoencoders.

So here's an example of this setup. This is the colorization problem I showed you before. Try to predict the colors from a black and white image?

And this is a self-supervised method that tries to predict part of the raw data, the color channels, from another part of the raw data, the black and white channel. So we can now take that model that's just trained to do this self-supervised prediction of the missing colors.

Now it's free labels because color images have the colors built in. I just took the color image. I split it into the luminance channel and the color channels. So it's cheap. It's easy to run this on just raw, unlabeled data.

Now I can investigate, did this system actually learn a meaningful representation? So we'll do the deep net electrophysiology. I'll poke my probe in there and ask, what are the neurons sensitive to?

So here, I guess I'll ask the class. What do you think? Remember when I talked about the Zeiler and Fergus model, I said that they were like face selective neurons, and dog face selective neurons, and other things. What do you think the neurons will be sensitive to for predicting colors?

And let's look at layer five of a network, so deep inside the network. What do you all think? Yeah, over here?

AUDIENCE: Textures.

PHILLIP ISOLA: Textures, OK, good. Yeah, textures seem to be-- color is kind of low level. It seems like it should be something that's a photometric property, like texture. Any other ideas? Yeah, in the back.

AUDIENCE: Objects.

PHILLIP ISOLA: Objects, OK. Yeah, so why objects?

AUDIENCE: Different classes of objects have different colors like [INAUDIBLE]

PHILLIP ISOLA: Yeah, different classes of objects have different colors. If I know something's strawberry, I can say it's probably red. So knowing the object category tells me a lot about the color. Anything else? So, yes, it turns out objects is one of the answers.

So here are our three different neurons. And this is looking at the feature maps. And we're just shading-- we're blacking out all the parts of the feature maps where the activations are below a threshold. So this is like, where are the neurons firing at some convolutional layer 5 of this network?

And, yeah, on lower layers, it is textures and other things like this. But the interesting thing is that it doesn't really matter how you train these networks. If you train them to predict classes, if you train them to predict colors, if you train them to inpaint missing pixels, just pick the right half of the image from the left half, the units that carve the world at its joints, that are predictive of everything, turn out to be objects and semantics and the words that humans have.

So it's like words are not arbitrary. We have the words we have because they're very predictive statistically of missing data. So this is discovery of semantic words without any semantic labels.

And this has been the big finding over the last decade that led to this revolution in how we do deep learning, which was the move from supervised learning to self-supervised learning. So the self-supervised learning is-- we used to try to do what's called unsupervised learning, which is just learn from raw, unlabeled data. But then we switch it into the mathematical machinery of supervised learning by just predicting some fake labels, which are just raw data, which you call pretext labels or pretext tasks from the data itself.

And there's so many different ways of doing this learning by prediction. So you could just predict class labels. That would be called supervised learning of representations. You could predict the next frame in a video, the future. You could predict the next pixel in a sequence and the same thing for any other data modality, too.

So the two on the right are self-supervised because no human had to provide the label target. It's the next frame in the video, the raw data. It's the next pixel in a sequence.

And as you probably know, the way that language models work is they predict the next word in a sequence, so they're mostly in the family of this type of learning. And most people still call language models self-supervised because they're just predicting the raw data. It happens the raw data is semantic and is words, but in that context, it's not like they're predicting the sentiment. They're just predicting the next word.

So all of these self-supervised tasks can be understood as something we call imputation, which just means take your data-- it could be a matrix, a tensor, some data object. A video would be like time by x by y. A sentence would be time by the content at that time, the word at that time.

And mask part of it, and put that into an encoder. Decode it to predict the masked part of that input. So masked prediction or imputation is the standard pretext tasks that people like to use these days to learn representations.

So spatial imputation, just predict the next pixel from the previous pixel. Temporal imputation-- predict the next frame from the previous frame. Channel imputation-- predict the colors from the black and white and, again, on other modalities, you can do the same.

Yeah, you can do this in a way that frames it just like an autoencoder, again, but it will be called now a masked autoencoder. So masked autoencoder is you take your data, you mask random chunks of it. And the little interesting trick is that in masked autoencoder is applied to images.

You're using a vision transformer. And the vision transformer already tokenized the image into the first-- remember, we talked about vision transform. We said that we are going to create a set of tokens by chopping the image into patches and then mapping those patches to vector embeddings or tokens that then get processed. And so they just said, well, what if I just remove some of the tokens, and then I predict the missing tokens?

And the really interesting thing about the attention architecture is that I can mask different ratios. I can only keep four of these tokens, but then the attention mechanism will scale in a way that is proportional to the number of tokens. Each token will attend to each other token.

So if I have four, it will be four attending to four. And it will output predictions for four. If I have eight, it will be eight attending to eight. So it has this nice kind of architectural invariance to the number of tokens you put in.

And then I decode. I just have some blank tokens I put into another transformer. And then I have trained it so that those blank tokens get filled in with the prediction of the pixels in the missing tokens, or the prediction of the data in the missing tokens.

And masked autoencoders are just a new name for another model which was very popular called BERT. So there are differences. Engineering this to work on language versus vision is a very important difference but, conceptually, they're almost the same thing.

So BERT was a language model that was very popular some years ago. It's gone a bit out of fashion, but BERT is roughly the same thing on text. So I just take my text. I tokenize it. I mask some of the tokens. I run it through a transformer, and I predict all the tokens. So I'm now doing masked prediction with language.

And the autoregressive models that try to predict the next word in a sentence are just the same, except they're only masking the final word as opposed to interleaving words. And that has some advantages in that I can decode in sequential order as opposed to in out-of-time order. So, yeah, question?

AUDIENCE: Why is BERT getting out of fashion?

PHILLIP ISOLA: Why is BERT getting out of fashion? So I think it's because masking the final token, naturally, can be used for generating sentences autoregressively, which we talked about before. If I mask the interleaving, then it's like I'm generating words out of temporal order.

So if I'm talking with a human, time is an axis that is important and not symmetric with other axes. I have to answer the question after the question has been asked by the person I'm talking to. And so having these causal attention mechanisms for the masking is in causal order. I'm only masking the future in conditioning on the past. It just fits into language models, which operate in causal order.

And once that became popular, the biggest models were all masking only the tokens in the future, given the tokens in the past. And because those were the biggest models, they just worked the best. But if I want to learn a sentence embedding, I bet the BERT method is still going to work better if scaled the same amount.

OK, so here's an interesting empirical finding. This is going back to that colorization paper, but it's been shown in a lot of work, which is that masked prediction just tends to always work better than autoencoding.

So if I I'm going to look at the accuracy, I'm going to look at the representations at each layer of an autoencoder versus a masked prediction network, for colorization in this case. And I'm going to assess performance seeing how well I can linearly classify given the representation on that layer.

Imagenet categories, in this case, is like a probe, just like we were doing with the shapes, where we had the nearest neighbor probe. Now we have a linear classifier probe. So here's what happens.

As I go deeper in the network, you get this separation, where autoencoding learns an OK representation, but masked prediction learns a representation which is more semantic. It linearly decodes ImageNet categories better. And this is something that I thought I had a good explanation for, and then I realized I don't.

So I'm going to call it ongoing science and leave it as a puzzle for the class. And maybe this would be a good final project. Why does masked prediction work better than autoencoding? So here are three hypotheses to think about.

So one is that autoencoding controls compression via dimensional bottleneck. Masked prediction controls compression by the non-overlap between the outputs you're predicting and the inputs you're conditioning on. So the only thing that's useful for predicting the outputs is the mutual information between the inputs and the outputs. And if these are separate, then I will just forget all the stuff that's specific to the inputs that's not relevant for the output. So that's how it does compression.

So it could be that just controlling compression via dimensional bottlenecks is really hard to make work. It's very finicky. It requires an architecture that has constraints on the dimensionality of the embeddings.

And we know that low-dimensional things, anything with low dimension and deep learning just is hard. It interacts with optimization in weird ways and interacts with BatchNorm and LayerNorm in weird ways. So low-dimensional embeddings have some bad properties, and maybe it's just really hard to use dimensionality of the embeddings as the way to learn a good representation.

Hypothesis two is autoencoders are learning shortcuts. So in order for me to reconstruct the signal-- imagine this. What if I had an autoencoder with skip connections, with residual connections?

Would that be a good idea? What would be wrong with that, autoencoder f, g, but they just have some residual connections in between f and g?

AUDIENCE: Not forcing this low-dimensional representation.

PHILLIP ISOLA: Yeah. It skips the bottleneck. So there's all these little gotchas like that. And it could be that autoencoders just have this tendency to copy the local information that they're processing. And that will be a decent solution, even though it would be better for them to capture global properties in order to reconstruct the entire signal.

And then maybe math prediction is closer to the prediction problems we actually care about. So somehow it's closer to the actual use case. We're going to try to predict the future from the past or something like this.

So I don't know. I asked Kaiming, who's a professor here who was the first author of "The Masked Autoencoder" paper. And he said, well, at the end of the day, it's just empiricism. So mass prediction works better than autoencoding. And I think it's a really interesting question, theoretically why that should be the case. Yeah?

AUDIENCE: [INAUDIBLE] beginning of the lecture, you said that in the scope of representation learning, you think this framework of [INAUDIBLE] is simplest and will win out. I was wondering if you could talk about why you think that is?

PHILLIP ISOLA: Yeah, so ultimately, it just goes back to some first principle argument that I can't really prove. But there are arguments along the lines of Occam's razor and formalisms of that the compression is somehow equivalent to prediction. And the most compressed representation will make the most accurate predictions about the future.

And autoencoders are just a simple embodiment of the idea of take your data, compress it as much as possible, and remove all redundancies. And it just seems, from first principles, if compression is all you need, then autoencoders are all you need. I think there's more to say about that. I don't have time right now.

So, yeah, still an open question. I'll end with I think this is a nice idea that Yann LeCun put out some years ago, which is that intelligence is like this cake. A lot of you may have seen this, where the bulk of the cake is representation learning.

You spend a lot of data and a lot of time coming up with a really good representation of the world in a self-supervised way or an unsupervised way, not tied to any specific task. So self-supervision is about not having labels, but it's also about not being narrow and tied to a task. It's just generally compress the universe into something that's predictive of the future and is compact.

And then the icing and the cherry are all the rest of machine learning-- supervised learning, adaptation, post-training, reinforcement learning. So he's making the point that representation learning is the bulk of intelligence, and I agree with that point. So I will end there. Summary-- you can read online.