# MIT Introduction to Deep Learning | 6.S191

56 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=II4GIR4VOOO
https://www.youtube.com/watch?v=II4giR4vOOo

SUMMARY

*Alexander Amini introduces MIT's 6.S191 Introduction to Deep Learning, a one-week boot camp covering the evolution and current state of deep learning technologies, including neural networks and large language models (LLMs). The course emphasizes understanding both foundational concepts and recent advancements, with hands-on labs and project competitions to enhance learning.*

- *The course covers the history and rapid advancements in deep learning, including the evolution from neural networks to LLMs like GPT-4.*
- *Students will learn about the architecture of neural networks, starting from single neurons to deep neural networks.*
- *Key concepts include forward propagation, activation functions, and the importance of non-linearities in modeling complex data.*
- *The course will also address practical aspects of training neural networks, such as loss functions, gradient descent, and backpropagation.*
- *Regularization techniques like dropout and early stopping will be discussed to prevent overfitting.*
- *Students will participate in hands-on labs using TensorFlow and PyTorch, focusing on music generation, computer vision, and language modeling.*
- *The course culminates in a project pitch competition, with opportunities for winners to present at TEDxMIT talks.*

Okay. Good morning everyone, or actually good afternoon everyone. My name is Alexander Amini and thank you all for joining us today in MIT 6.S191 Introduction to Deep Learning. We're super excited to welcome you to this class. Uh my name is Alexander. I'll be your instructor along with Ava, uh who will you'll hear from later today as well with the second lecture. This is a one-week boot camp on everything deep learning, right? So, you'll go from the very beginnings of neural networks all the way to some of uh the the most recent LLM advances that we've had uh even up to last week even.

We'll be covering the the field. The pace of this field is really truly remark- remarkable, right? And that's one thing that we'll keep touching back on throughout the course in a lot of detail that we want to see actually not just what the current state of the art is, but also remember how we got there. And I think it's really easy for us to become desensitized actually with a lot of the state of the art with AI because there are so many amazing things happening and we really are on this exponential trend. It's really hard for us to actually remember where we were just a few years ago. So, I think what better way to to do that than to see it.

So, exactly one decade ago, uh we started teaching the class. We created this class nine years ago. We started

teaching it nine years ago. One decade ago, in 2015, this was the state of the art in terms of facial image generation. So, generating facial images. Uh fast forward a couple years to 2018, you can already see a massive leap in quality. And then fast forward a couple more years and you see that extend into a new dimension of time into generating videos as well. In fact, this was a video that we generated for the class in 2020 to actually introduce the class. If you guys haven't seen this video, you should watch it online. It's on YouTube. It went a bit viral a few years ago when we started the class.

Now, in the past few years, we've seen this revolution expand also into natural language as well. So, in 2022, we had ChatGPT-3 released and launched to the public followed shortly by GPT-4 in 2023. Now, the jump between 3.5 and 4 was often times regarded as this massive leap in AI capabilities when we went from GPT-3.5 to GPT-4 because really everybody felt that the the capabilities were vastly improved between these two models. And everybody could feel it.

Even non-technical folks could really feel the capabilities come through. Now, what you couldn't really see or what you couldn't feel was what was happening behind all of those giant models, right? So, GPT-4 was this incredible model, but this is what you don't see. Powering GPT-4 are these massive and 5, of course, as well, are these massive data centers, GPU data centers. They cost hundreds of millions of dollars to build and tons of infrastructure to maintain and to to manage these different data centers.

Now, these are for large language models. We've also had this this this revolution of AI computing capability on small form factors as well, right? So, just like we have large language models powered in the cloud, we've also started to see small language models go directly down onto edge devices. Now, traditionally, this gap between large language models and small language models has been very large in a capability sense. We couldn't really accomplish huge jumps in capability in small language world uh that we've seen in the large language world. But, let me show you actually what's happening right now. So, I'm going to switch over my screen to this screen, which is actually running Let's see here. So, this is actually live on this phone running right here.

You can see everything that's happening on the phone. So, what I'll do is I'll show you actually a language model running entirely on this phone right now. So, we can ask it questions. This should look very familiar to what you have seen with GPTs. So, how are you doing? We can ask it very simple questions to start. Now, the thing that you'll notice here, if I swipe down from the top, this model is running entirely on airplane mode.

There's no Wi-Fi connected to this phone. This is entirely offline. Everything that's being computed is running on the device itself. Now, of course, we can ask it more complex questions as well. So, here's a question about finding the zeros in this equation. It looks simple at first glance, but actually, if you read it, it's actually asking about it in Z7 space. Does anybody know what Z7 space is? Anybody? Yes. If it's integers but like modulo 7.

Exactly. Exactly. So, it's in modulo 7, the finite space of uh of modulo 7. We can ask it this question. It can immediately know the answer. You can check the answer. It is indeed correct. We can also ask it to maybe like give me more explanation. Right. Okay. So, it it's able to do this. Now, let me go back out, and let me actually show you another demonstration. Now, these models are also permeating beyond text language as well. We can

ask questions directly in audio language. Now, these are models that are not converting from to audio audio to text. These are pure audio models that take audio in and directly output audio tokens out. So, I can ask you a question like "Can you give me a step-by-step guide on how to give a really compelling talk?"

Sure. Start by clarifying your main message and who you're speaking to. Then, sketch a simple outline with an intro, a few key points, and a strong conclusion. Next, craft a few opening and closing lines that grab attention and leave a lasting impression. And practice your delivery so you can adjust your pace and body language. Finally, rehearse your talk, get feedback, and tweak any weak spots before the event. Right. So, what have we seen? Right. So, this is an end-to-end audio model. It means that the latency is extremely low is what you'll probably notice there because number one, you don't send anything to the cloud, so you don't have to wait for internet times. Everything runs on device.

And you're directly processing audio. So, it's extremely responsive. These are models that can actually understand your emotion because they don't convert audio to text. When you convert to text, you lose emotion. So, you directly capture the emotion in the speech and you can directly convey the emotion back to the user. Now, this was a model actually developed by Liquid AI. It released about 1 week ago. It's multimodal. It can process all of these different modalities, text, audio, and vision.

But, there have been a ton of these model releases in the past uh few years actually since GPT first launched. And this problem that traditionally was having between large language models, the gap between large language models and small language models has always been around, right? This was actually a very special release for the community because of this context on the left-hand side. So, I'll read it out just for everybody to see. So, it's pretty crazy when you think about it. One of the biggest wow moments of GPT was the update from 3.5 to 4. And now this open-source model that anybody can download and put on their phone with only 2 billion parameters is better than the original GPT-4 model in terms of every benchmark basically that comes out, right? And this is a model again that you can run entirely on your phone. It's open-source. You can fine-tune it on your own prem.

Right? So that's the pace of the field. That's why this class is such an happening at such an exciting time. Um so let's dive into it, right? This is what we'll learn in the class. Let's dive in by first of all laying some foundation of what exactly is deep learning. And to do that we need to first understand what is intelligence because deep learning at the heart of deep learning is AI. Right? So what is AI first of all? AI is the practice of building artificial algorithms to process information in order to inform some future decisions, right? At its core, that's what we call intelligence in fact even. And artificial intelligence is just the ability for a machine to do that same ability. Process information to inform a future decision.

Now machine learning is nothing more than a subset of AI that specifically does this without explicitly programming a computer on the steps of how to process that information. It's able to learn this from data, right? That's the differentiating point between AI and machine learning. And now deep learning is nothing more than a subset of machine learning which focuses on the use of neural networks, specifically deep neural networks, to do this task of learning from data to inform future decisions and other tasks.

Right? So we'll we'll circle around this theme of teaching computers to learn a task directly from raw data. That really is what this whole class is all about. And we'll learn about different models to do that, different ways and different algorithms to accomplish that main objective. This class is split between both technical lectures as well as hands-on project software labs. We'll have several new updates this year, especially towards the second half of the week where we start to cover a lot more of the new advances in AI and deep learning. And we'll conclude also with some guest lectures and some project presentation competitions that will give everybody the opportunity to win some awesome prizes.

Yes, also we John is graciously offered to host the top winners of the project presentation for TEDxMIT talks. So, who has not given a TEDxMIT talk? Probably most of this audience. So, this is your opportunity to win a spot on the big stage and give a TEDxMIT talk as part of the project pitch competitions. Um the software labs are an excellent way to get hands-on and actually learn how to deploy and build what we learn about in the technical lectures directly with your own code and programming. We offer software labs in both TensorFlow and PyTorch.

We'll talk more about this later today. Okay, so how do the projects labs work? Right, so every day we have a dedicated software lab that helps mirror exactly what you learn in the technical lectures. Also, you learn this in the project labs as well. And every day covers on both sides. Right, so starting with lab one today, where where you'll be learning how to build a music generation model that can learn how to generate a certain style of music.

Lab two, which is tomorrow, is going to be focused on computer vision and facial detection systems. And then lab three is a lab dedicated on large language modeling. You'll fine-tune your own large language model. You'll evaluate it. You'll see the whole process end to end from training all the way to evaluation and deployment. And finally, on the last day of this class, we'll cover a final project pitch competition. Three to five minutes, very Shark Tank-like style. You'll get on stage. You can work individually or in groups. We'll talk more about the details for this a bit later throughout the class.

This class has many amazing resources to help throughout the week. Uh please post to Piazza. I think everybody here should have received the link for the Piazza, but please uh reach out to us if you haven't. Um we also have some incredible TAs who you'll meet throughout the week and will help support if there's any questions. All of these slides, by the way, are online on the website, so feel free to um grab any of these links or details from online as well.

And of course, uh we have an incredible team. Myself and Ava are your main instructors, but then we also have uh an incredible team of TAs and we also have some incredible guest lecturers lined up for Thursday and Friday as well that you'll definitely not want to miss. And finally, just a huge thanks and uh call out to our sponsors who, without their support, this class is not possible at all. Okay. So, now let's start with uh some of the fun stuff. And let's start by asking ourselves, first of all, a question that uh probably is pretty self-explanatory at this point, right? So, which is why do we actually care about deep learning so much, right? Why are you all here? Why are you so excited about this topic?

And the answer really boils down to what I believe is is one thing, right? Traditional machine learning algorithms really require a level of hand engineering, human human-guided hand engineering to teach computers how to perform a particular task. Deep learning is so exciting because it enables us to learn those rules that traditionally are driven by human engineering, now by computers and by data, right? So, the key idea here is to learn those features directly from data and to do that in a hierarchical fashion, right? So if we want to learn how to detect faces, for example, right?

You may do that. As a human, how would you detect faces, right? And think from a first principles way. You would probably start by looking at an image and first detecting the lines in the image and composing those lines together to see if there are certain corners in certain parts of the image that resemble a face and then putting those corners in different uh configurations together to see if you can configure an eye or a nose or a mouth. And if you detect any of those things, then it's even more inkling that you're probably uh looking at a face, right? Deep learning algorithms are extremely hierarchical in the same way. They build up features from very low-level features like lines and edges to higher-level shapes like corners and curves all the way up to mid-level objects and higher-level objects as well.

Now, applying the fundamental building blocks of how to learn those hierarchical representations is actually something that is not new and not very recent at all. A lot of these core algorithms have existed since the 1950s, 1970s, they really took off a lot. So why are we really now seeing this resurgence? And there are really three answers to this. One is big data. The data in the world has never been more prevalent than today and these algorithms live off of data. This is what they are feeding off of, right? So they really need and require massive amounts of data that the world has now been able to provide. The second is hardware. Companies like AMD and Nvidia, of course, as well are providing new GPUs that are accelerating parallel computing and accelerating the development and the deployment of these uh algorithms. And finally, software that you'll get hands-on experience throughout this class that it actually democratizes the ability to create, train, and deploy any of these models even at relatively large scales.

Okay, so let's start with the fundamental, the most fundamental building block of what makes up pretty much every neural network, uh and that is the single neuron, right? Also known as the perceptron. Right? So, this is really the most basic building block that's very important to understand when starting deep learning. Now, what is the idea of a single perceptron? At its core, it's actually extremely simple. Let's start by first talking about the forward propagation of information through this neuron. So, if you have information and you're passing it through a neuron, we can define a set of the inputs to that neuron as X1 to Xn.

Or here we call it Xm, excuse me. Okay, so we have m inputs, one to m, and this one neuron takes all of these inputs and it has to create some output. This is the goal of this neuron. Now, how do we do this? How does this neuron actually take those m inputs and compute its output? It does this by taking each of those inputs, multiplying every input with a corresponding weight, or it's called the weight. Well, every input has a corresponding weight, so X1 corresponds to weight one.

X2 corresponds to weight two. You multiply every weight with its corresponding input, then you add up all of

the answers, and then you pass this through to the output. With one more exception, which is that you also pass it through what's called an activation function. Here it's denoted as g. So, you multiply your inputs with your weights, add up everything into one number, and then pass it through some transformation, which we'll talk about in a second, and that is your final answer. That's one number that comes out of this neuron from m inputs that go in.

Okay. There's one more detail I left out of this, which is that we can also have a what's called a bias term to this neuron as well. So, in addition to our M inputs on the left, we also have one more weight W0, which is called our bias. Now, our bias is what can allow us to shift this this function up or down, right? And I'll show a graphical example of this in a second. Right? But the equation is still the same. We take every input, multiply by corresponding weights, add a bias, pass through our non-linearity.

So, now that X and W are actually just lists of numbers, each of which are M dimensional, we can actually write this in vector uh form, right? In linear algebra form. So, the output now, Y, is just obtained by taking the dot product between X and W, adding this bias, and passing through a non-linearity. Right? This simplifies the equation at least uh notation-wise quite a lot, right? From the previous slide.

Now, you might be wondering about this activation function that I mentioned previously. I said this is a non-linear function, right? What does that mean? It just means that you're transforming something from the X axis that can be any real number, anything from negative infinity to infinity, into a new real number that sometimes is bounded, sometimes not, but that transformation is not linear. So, here's one example of a non-linear activation function. This is called the sigmoid activation function. It transforms any any number on the X axis into a new number between 0 and 1.

Right? In fact, there are many types of non-linear activation functions that you'll get exposure with throughout this class. The sigmoid activation, which you see in the left-hand side, is just one example of very common activation functions in neural networks. The the central theme that links all of these together though is that they're all non-linear. Right? The activation function with sigmoid on the left is very commonly used for things like probabilities because it always outputs between zero and one, which is the space that probabilities live in.

On the right-hand side, you have activation functions that output between zero and positive infinity, which is great for introducing non-negativity constraints into your models as well. And it's very simple because it's actually just two linear functions with a non-linearity between them. Okay. So, now why do we actually even need activation functions, right? Why not just directly use the dot product that comes from our weights and our inputs? The point of an activation function is precisely to introduce non-linearities into our model.

Right? And why do we care about this? It's because in real life, real life is highly non-linear. It's highly complex and dynamic. And let me show you an example. So, here's a here's an example of a data set, just a two-dimensional data set, not even extremely complex. But if I asked you to draw a line separating the green points from the red points in this data set, it would actually be really hard. In fact, there's no line that perfectly separates the green points from the red points, and that's because this data is non-linear. Even though it's just in

two-dimensional space, it's not in high-dimensional space, but it's still non-linear and very complex. Now, imagine you're dealing with real-world data that's extremely high-dimensional and also non-linear.

Definitely, you need non-linearities in your model to handle this. If I tell you you can make a curved line, this problem becomes very easy, right? And that's what it means to have non-linear activation functions. We're trying to learn a mapping, a decision, to classify between red points and green points, but we allow our model to draw curves in the edges in this decision space between the two points. Let's understand this with maybe a simple example, and we can walk through it together. So, let's assume now that we have a neural network that was trained with two weights.

So, W1 and W2 is 3 and -2, just as an example. We trained it, this is the result that we got. And we want to pass in a new input to this model. So, we have two inputs to this model, X1 and X2. We obtain the output of this uh neuron by as we saw before, computing a dot product between our X and our W. Right? Our W's already trained, so we can plug in those two weights directly into this vector here.

Adding our bias, and then passing through a non-linearity. Now, if you look at this, what's inside of the activation function, this is nothing more than a two-dimensional line as a function of X1 and X2, our two inputs. We can plot this line, right? This is a trained neural network, or it's a trained neuron at least. We can plot this line and observe what this decision uh boundary looks like in the space between X1 and X2. So, for any new point X1 and X2, if I want to pass in an input to this neuron, I can actually plot this point. So, let's say I I pass in a new input, -1, +2 is the XY coordinate of this point. I can plot it in this space, and I can see that it actually lies on the left side of this decision boundary. What does it mean to lie on the left side of the decision boundary? Well, let's actually compute it mathematically first, and then we'll see. So, if we plug in those numbers to the equation, we'll get 1 + 3 * -1, which is -3, 2 * 2, which is -4.

Right? Add up all those together, you get -6. Right? That's a negative number. That means that we're on the left side of the decision boundary. When we pass that through our non-linearity, which is our sigmoid function, being a very negative number, that means that we're going to be further on the left on the under 50 under .5 on the sigmoid function, so where it's a very small number close to zero, .00. In fact, we can actually draw a hyperplane between these two parts of the space and say anything that lies on the left side of the decision boundary will always be negative, right? It will always be negative before the activation function, it will always be less than .5 after the activation function if it's sigmoid activation, and anything on the right is going to be positive before the activation function.

Now, this is really nice and convenient that we can visualize this trained neuron in this way here that for any input, we can directly visualize exactly where it lives in this decision space. But of course, this is only a two-dimensional space. It has only two neurons in our neuron Excuse me, two parameters in our neuron in our neuron, right? It's extremely small, right? Modern neural networks have billions of parameters. So, imagine this plot now with a billion-dimensional space. This is what we would be looking at.

Um so, of course, that's not possible, right? So, this is why this is a helpful exercise to explain, but of course, we

need uh to actually scale without having this exact uh visual, but building up more mathematical intuition. So, let's do exactly that. Let's scale this idea now into larger networks, and let's start by taking that one neuron and actually building a neural network out of it to see how this all comes together, right? So, this is our original neuron picture, okay? Now, if there's a few things that you remember from this class, this is probably the most important thing is that you should remember how a single neuron works, and that's by doing a dot product, adding a bias, and applying a non-linearity. It's really three steps: dot product, add a bias, and apply a non-linearity. That equation is the core to so many different parts of deep learning.

Now, let's simplify the diagram a little bit now that we got that foundation settled, right? I'll remove all of the weight labels, and we'll just have every line you can remember that every line will have both an input coming in as well as a weight associated to that line. Okay. Now, Z here, what's shown as Z, is the result of that dot product plus a bias, right? This is what's happening before the non-linearity, right? So, the final output is simply Y of G, which is our non-linearity, of Z.

Okay. Now, let's define a multi-output neural network now. To do this, we just have two neurons instead of one. They will both see the same inputs, but now they will have their own independent weights. So, each neuron has its own three weights. It sees the same three inputs, but it now can output two different answers because it has two different sets of weights. Okay. So, now with this mathematical understanding, let's see if we can now build our first layer neural network layer entirely from scratch.

So, let's do this uh by first initializing a matrix called W. This is going to be a matrix that contains all of our weights for all of our neurons in that one layer. Okay. So, we can create now this matrix with uh of weights, which is going to be dimensionality of the input space by the dimensionality of our output space, right? Now, the bias is also important to initialize here. And then when we create our call function, our forward pass function, this is going to show actually how we can multiply, do the dot product of our inputs with our weights.

Okay. Right here. Add the bias. Apply the non-linearity. And that's our answer, right? It's the same equation again. Okay, we can do this in TensorFlow. We can also do it in PyTorch. And you'll notice actually between these two it's very very similar type of language. Uh, a lot of parallels. Basically just different names for a lot of things, but the overall frameworks are extremely similar. Same foundations. We'll initialize the weights and the biases.

In the forward pass, we'll compute the output of this layer by computing a matrix multiplication of our inputs with our weights, adding a bias, and then applying our non-linearity. Okay. Now luckily both TensorFlow and PyTorch have actually defined that exact layer for us already. So you don't actually need to to write that code yourself. You would just call the layer. In TensorFlow it's called the dense layer. In PyTorch it's called a linear layer. They do the exact same thing. They do the matrix multiply, add a bias, apply a non-linearity.

So we can just call it instead, right? We call it with the number of units or the number of neurons in each layer. Okay, let's keep building up this abstraction step by step. That's a single layer neural network. This is one where we can have a single hidden layer now. So this is actually two layers. Now we can see how we can do this

transformation from input space to the hidden layer space with one layer, and then hidden layer space to output space with another layer. So now we actually have two weight matrices, right? The first weight matrix on the left-hand side W1 converts the inputs to the hidden layer.

Second weight matrix is W2 which goes to the output. Right? Each of these are separate weight matrices, and they actually have different sizes as well because your output shape is different than your hidden layer shape. Now, if we look at a single unit, let's zoom back in now to a single unit within the hidden layer. Right? Let's take for example this one, Z2. This is just the second unit in our hidden layer of this model. This is the same perceptron that we saw before, nothing special here. We compute its output Z2 by taking a dot product of all inputs with the weights corresponding to that neuron, adding its bias, and applying a non-linearity.

If we took a different node, like Z3, for example, it would look also the exact same, except its weights would be different than Z2's weights. The inputs are the same, but they see different weights. Okay, so this picture looks a bit messy, so let's abstract even further so we can keep building up this this intuition. So now I'm going to remove all the lines and just put these symbols here in cases wherever we have these fully connected layers. Everything that connects from input to output with the matrix multiply in between.

And again, we can actually stack these fully connected or linear layers on top of each other with non-linearities between them to make sure that we have the non-linearities across our network as well. And here's an example of how to do this. You basically stack the same layers that we saw before now in these sequential blocks. Sequential just basically allows you to stack them. Okay, now finally the the point that we've been waiting for is how to go from these neural networks to deep neural networks. There's nothing more to this than just stacking a bunch of linear layers on top of each other. A deep neural network is nothing more than a neural network that has more than usually three layers is what people can conceptually say.

Right? So this is a network where the final output is computed by going deeper and deeper and deeper across each of these layers to compute and its final output at the end of the network. And again, to no surprise, this is done in the same abstraction code as before, just having a sequential block and a bunch of linear layers within that block. Okay, this is awesome. So now we have this good intuition of how to build up from a single neuron all the way to a layer to a neural network and to compose these in the forward pass.

Let's take a quick look at how we can apply this to a particular problem that it's it's a good starting problem for everybody in this class, which is should will I pass this class, right? So this is a question, it has a yes no answer and there's a lot of data for this question actually because we've taught this class for a long time. So we have a lot of data from past students who have taken this class. Here's an example. So we have an X and a Y axis.

X is the number of lectures that you attend and the Y axis is the number of hours that you spend on the final project. And we also have individual data points which represent past students who have taken this class. You can see where they live in this space. And then we have you. You fall right here at 4 5. Right? You've attended four lectures and you spent five hours on your final project and the question is will you pass this class based off of all of the other data that you've seen, right? Now, how can we do this?

We have a neural network that can do this. Let's try and actually feed it into a neural network. We have two inputs which are the exact two axes that you saw before, number of class number of lectures and number of hours. One input is four, the other input is five. We can feed these two inputs into our neural network and we can see that it predicts an answer of .1 or 10% probability that you'll pass this class.

Very bad, I know. Okay, who knows why the the network is wrong in this case. Yes. The four Exactly. Okay, so this is a randomly initialized network. It is effectively the same as a brand new baby, right? It's never seen the world before. It has never seen any of the data before. So, the step that's missing here is that we've just done a forward pass through the model. Now, we have to actually teach the model, right?

And the key part of this is that every time that we have a prediction, the model predicts .1, there's also a ground truth answer for every prediction that we have in our training set. And the ground truth answer here is is true, right? It did actually You did actually pass this class, right? So, there's a deviation here. There's a difference between the .1 that the model predicted and the one that it should be answering. Now, to train this network, we need to show it this difference. We need to tell it when it gets the answer wrong so it can learn to move closer towards the correct answer the next time it sees a four or five, right? If it sees four or five again, it should now improve on its answer and should predict something closer to a one.

We do this by quantifying what's called a loss. A loss is much nothing more than just the error, this deviation term between the prediction and the ground truth. Smaller losses means that the model is is is outputting a higher quality answer. Larger losses means that the answer is more erroneous. So, let's assume of course that the data is not just from one student, but we have a data set of many, many students, right? This is coming from now a loss over the average of each of those losses that we have in our data set, right? So, when training a neural network, often times we're not going to actually minimize the loss on a single data point, but minimize it across the entire data set, right? And that's to have higher quality outputs.

Now, if we look at this problem in the case of classification, a yes-no type of problem, then this is where the answers will typically predict a probability, a probability of being a yes versus a probability of being a no, and that's why we predict between zero and one. This ties us back to the sigmoid function from before, right? Because those outputs are zero and one outputs, so we can train it in this way. Right? And we can use losses like the cross-entropy loss that actually support matching those two distributions of probability against each other.

Now, you might be asking yourselves, "Okay, what if you didn't want to predict a probability distribution, like a yes-no question, but rather a continuous value, like what's the temperature going to be, or what score will actually get on the class rather than will I just pass or fail it?" In those cases, you care about predicting continuous values instead, not probability distributions. So, you actually want to output and have losses that encourage deviations that are also continuous. In those cases, you may consider things like mean squared errors or L1 errors, things like this that simply compare the predicted value minus the uh ground truth value, and just minimizing that absolute difference.

Okay, now let's put all of this loss information together into the problem of actually finding the neural network

weights that are optimally suited for any particular data set or any particular task. So, we know that we want to find a neural network that looks roughly like this form, right? It we want it to minimize the loss on our entire training data set, and this means that we want to find the W's, the weights, that minimize uh the the the the loss here, the J, the J of W, right? J of W is our empirical average loss across our entire data set of N data points.

Okay. So, remember again that W is nothing more than just a collection of many, many weights across all of the different layers in your neural network. Everything from layer zero, the first layer, all the way to layer N, which is your last layer. Now, our loss function is also nothing more than just a function that outputs a single scalar at the end, right? We take the average loss, the average error across every data point, compute its average, and that is our loss. It's just one number at the end of the day. And that one number is a function of our weights.

So, on on here, you can see a visualization of let's say again, a two parameter neural network, just two parameters, W0 and W1. For any configuration of those two parameters, we have a particular value of the loss, which is the height. Okay? There are some weights that encourage a really good loss, very low. There are other weights that are really bad losses, very high on the curve here. Right? And the goal is that we want to find the neural network that has the two weights, W0 and W1, that have the lowest loss.

Now, how do we do that? So, this is called training the neural network. We start by training the neural network. We just randomly initialize it. We'll pick a W0 and a W1 to start with. Let's say we start here at this black point. We compute its loss. And then we compute the gradient. The gradient tells us the direction of the slope at that location, right? Doesn't tell us the slope everywhere, just tells us at this very local location, which way is pointing up.

But of course, we want to go down, so we actually take the negative of the gradient and take a small step down. And then we repeat this process. We compute the new gradient at the new point, and we take a negative step in the opposite direction of which way the gradient is pointing, until we finally get to the bottom. And this will actually be guaranteed to converge to a bottom. May not be the lowest bottom, right? Because it actually depends on where you start. You may not find yourself in the absolute lowest, but you will find a bottom, and you'll eventually converge there.

So, we can summarize this algorithm. This is known as gradient descent, and we can write it roughly in pseudocode. It looks like this. We start by initializing our weights randomly, and then we repeat the following process until convergence. We compute the gradient at the weights that we're currently in, and then we take a small negative step of our gradient with our weights. So, we change the gradients to step in the opposite way that the gradient tells us, and then we keep repeating this until there's convergence, until our weights are not really changing anymore, and then finally we call those final weights the trained network of the model.

Um in code, right? This also is is is doable in code. We can do this while looping in a while loop, where we compute the loss. We can compute the gradient of our loss with respect to our weights, and then compute that same update formula again. Now, a very important line here is to look at this term right here, the gradient, right? Now, I glossed over this this point, but actually the gradient, this is a key part of this algorithm, because

computing this tells us exactly how to update our weights on every part of the iteration, but I never actually told you how to compute this. So, let's talk about actually computing the gradient for a neural network, and this is a process known as backpropagation. We already learned about forward propagation of information through the model. Now, we'll learn about backpropagation, which is how to actually compute the gradient of the model.

We'll start with a very simple neural network uh to illustrate this example, and this is actually probably the simplest neural network that you can have, which is just a single neuron with a single output as well. So, X goes to a single neuron, goes to a single output, goes to a loss. So, let's visualize what this would look like if we wanted to compute the gradient of our loss with respect to our weight W2, which tells us how a small change in our uh in W2 affects our loss J.

Okay, so let's write it out as a derivative. So, this is the derivative of J with respect to W2. Right? Now, to compute this, we can use the chain rule. Right? We can decompose dJ dW2 into two parts. The first part being dJ dY times dY dW2. Right? This is just uh an application of the chain rule in linear algebra. Now, this is only possible because Y is only dependent on the previous layer.

Right? Let's suppose now we wanted to compute the gradients of the weight before that, which is W1, not W2. So, let's actually change this to uh dJ of d W1. Now, here, this is dY dW1. We all We again need to apply a chain rule again to compute this. Right? So, we expand it out one more time. That part turns into dY dZ1 followed by dZ1 dW1. Right? And if you had a deeper neural network, you would just keep repeating this process as you go deeper and deeper into the model.

And and uh just repeats like this, right? You can repeat this process by propagating those gradients that you compute from the end, from the output, all the way back in in the in the topology of the neural network, all the way back to the input. And this will allow you to determine how every single weight impacts your final loss. Right? That's exactly what the gradient shows you. It shows you if you move in some direction, how will the loss change? Will it go up or down?

Now, that's the backprop algorithm. In theory, it's just a application of the chain rule. Right? It's nothing more than the chain rule. In practice, this is a lot more complex because computing the gradient of neural networks does not look as clean as what we saw in that previous slide. Right? In practice, neural network loss topologies look highly non-convex. Right? There are a lot of different minimum and it's highly dependent on the initializations that you pick in these models and the regularizations that you apply to make sure that you can find actually this minimum.

So, this is a really cool paper from uh many years ago, 2017, where they actually tried to visualize what these high-dimensional neural network loss landscapes actually look like. Of course, this is a two-dimensional representation of something that is, you know, millions of dimensions. So, it's you need to take it with a grain of salt, but even projecting it down into two dimensions, you can see that this is a highly nonlinear landscape. Now, recall the update equation that we defined during gradient descent and remember this this parameter that we also didn't talk so much about. Right? I said we take a small step in the opposite direction of the gradient. What I

didn't mention is, you know, how big is the step? This step here is referred to as eta, right? This small symbol here. This is called the learning rate.

This is the rate at which you follow the gradient. In practice, setting this learning rate is just a number that you set. In practice, setting this number is also quite difficult, right? Because if you set something that is too small, then you actually never really reach some of the the big minimums. Right? This is a minimum, but the model gets stuck in it even though it's not the biggest one. Right? You would actually and if you set something too large, you can overshoot and your loss can also explode and you never find the minimum, either minimum.

So, ideally, you want to use learning rates that are basically just large enough that you can skip over these kind of fake local minimums, but get stuck in these global or close to global minimums uh to converge there. So, how do we deal with this? How do we actually set the learning rate? One option is just to try a bunch of different learning rates and see what works best. And actually, this is not as crazy of an option as it sounds like. Um but can you do something smarter than this? Uh how about we design adaptive learning rates that adapt to the shape of our loss landscape. Um that actually is a much smarter idea because it means that we can now increase or decrease our learning rate depending on how large the gradient is at that location. When you have a very large gradient, maybe you want to take a bit of a smaller step. If you have a small gradient, maybe you want to adapt or have some momentum that can carry you through these these local minimums and pass over them.

And that's exactly what a lot of different optimizers have come out with. So, we saw the most basic version of stochastic gradient descent previously, but there are a bunch of other variations of gradient descent algorithms called Adam, AdaDelta, AdaGrad, RMSProp, etc. that you'll get a lot of practice with throughout your software labs. And in general, these are actually much better types of optimizers for the reasons especially around setting learning rates and making sure that you can actually find these minimum.

Um and these are widely studied in practice, right? You In general, you will very rarely use the vanilla stochastic gradient descent SGD algorithm. You'd probably almost always use and change this line to something like Adam or another adaptive learning rate scheduler or optimizer. Awesome. Okay, I'll continue now and just talking about some more practical considerations, especially now towards the the end of the technical part of the lecture, this first lecture. I want to talk about some of the practical considerations that you should have when actually training neural networks.

So, let's go back to the gradient descent algorithm for a second. We just saw this this algorithm here. We saw how we can compute the gradient, but remember this is done through backpropagation, and this is a really expensive process because you have to go entirely back and compute derivatives for every single parameter in your network going from the output all the way back to the input, and you have to do this for every single data point in your data set, right? Not just one data point, of course. Now, in most real-life problems, it's not feasible to do this on every training iteration, right? You simply would not want to do that over every data point in your data set.

So, instead, we're going to define what's called stochastic gradient descent, SGD. So, instead of computing the

gradient over your entire data set, let's compute the gradient over just one point. Pick a random point and compute the gradient with respect to that one point and step in the opposite direction of that one point. Now, that is much, much faster to compute, obviously, on big data sets. Computing a gradient on one point is going to be way more efficient than the entire data set, but it's also going to be way more noisy. So, there's a trade-off there, of course.

Now, the middle ground, what is the middle ground? Is that you do this over what's called a mini-batch. So, you compute some batch size of data points, and you compute the gradient with respect to that batch size. Batch sizes can range from anything very small to like a few data points all the way up to, you know, uh like most commonly people use things like 32 is a very common batch size. You can scale it up. In large language models, we have batch sizes of, you know, millions, right? It depends on the type of data sets that you're using.

But in general, you always want to pick something much smaller than the size of your entire data set because you don't want to be going through your entire data set every time. Now, B is is normally not that large for most problems. Like I said, this is usually on the order of tens, hundreds, and even this gives you a pretty good estimate of the gradient for most problems. This increases the accuracy that's going to be much more accurate than SGD, stochastic gradient descent, that's only looking at one point, but it'll also be much more efficient than computing it on the whole data set. So, it's this nice balance and you can actually control the balance yourself by controlling the size of the batch.

Now, the last topic I want to touch through is overfitting, right? This is known a very well-known problem in the field of machine learning, but it extends also, obviously, into deep learning as well. Now, what is overfitting? Overfitting is nothing more than the the process of learning too much into your data, right? Looking too much into it and overfitting on the details of that data set that do not generalize outside of the particular training data that you've seen. So, here's a good example, right? If you start on the right-hand side, and here's your training data set, the optimal line that fits this data is not going to follow all of the intricate details of the data.

But on the same side, on the other Excuse me, on the other side, there's also underfitting as well, where you don't learn a function that's expressive enough that captures all of the intricacies of the data. So, for example, a linear function on the left side is going to be underfitting on this problem because the problem is a nonlinear type of relationship, but on the right-hand side, it's also too high dim the the the learned solution on the right-hand side is too high dimensional and too overfitting to to capture the generalization of the of the problem. The ideal is somewhere in the middle, right? Where you don't have too much complexity and you don't read into too much other details, so that when you see brand new data in a test set or in deployment, you can actually extend your solution to have accurate predictions in that regime as well.

So, to address this problem, how do you go and toggle between these two sides of the coin? It's called regularization, right? Regularization is the technique that allows us to constrain how much we want to underfit versus overfit on a particular problem. And this is effectively allowing us to discourage the learning of very complex models. This is how we're able to learn billions of parameter models, you know, on actually data sets, but not actually memorizing all of the details in the data sets or allowing us to generalize beyond those those

data sets.

And this will allow us to improve the ability of our model to perform even on unseen data. The most popular form of regularization, there are many different ways to regularize depending on the type of model, but I'll talk about one of the most popular ways. This is the idea of dropout, right? Dropout is is a stochastic method. It means that it's probabilistic. Let's revisit this this chart, this schematic that we saw earlier in the lecture from a neural network.

Right, in dropout, all we do is that we randomly drop out some activations of the hidden layers with some probability that we define. Okay, so let's assume that we set a probability like .5, 50%. All this will mean is that we will randomly pick 50% of the neurons in our two hidden layers and randomly kill the outputs from coming out of those neurons. So, what does that mean? That this neuron here has to be resilient to sometimes receiving inputs from this neuron, but sometimes not receiving it from this neuron, right? And then on other iterations, it may receive it from the other set of neurons. So, what this encourages is actually the ability that on every iteration, every neuron is seeing different pathways through the network. This is forcing it to not rely on any one pathway and not memorize any one pathway too much through its forward path propagation. And that's actually able to allow it to generalize as well because it has to find these relationships that span across different pathways. Yes. Are the same activations set to zero for the entire mini batch?

Same activations are set to zero for the entire mini batch, but then on the next mini batch, you have a new set of activations and new set of neurons that are set to zero. So, you reset on every batch. Awesome. Okay. So, you repeat that on every iteration. Let me show you one more technique for regularization as well that goes beyond dropout. Dropout is a architectural level of regularization. This is going to be beyond architectural level. So, let's assume for any architecture, you can have the ability to stop training once you start to overfit. Sounds easy. Let's see how you might actually do this, right? This is called early stopping.

So, we already have this definition of overfitting, which is simply that we start to perform better on our training set than we are on our testing set, right? That's effectively what it means to overfit. So, we're no longer generalizing our learnings from training into testing, right? Our training is continuing to improve, but our testing data is just getting worse, right? So, we can do this by actually having two separate data sets, one that we train on and one that we test on. And constantly throughout training, we're evaluating both data sets. Now, in the beginning, both lines just decrease together, which is excellent. It means our model is learning, right? It's moving from that underfit regime to a more fit regime. And it continues to have this trend. Eventually, though, the network loss will start to plateau, and the testing loss will start to increase.

Right? This is actually the regime where we start to see overfitting. And this pattern typically will continue for the rest of training. Now, the question here is where would you actually you train the model for all of these training iterations, you save the model at these checkpoints across the x-axis. You can now take this checkpoint, right? You've saved all of the checkpoints. You look at the curve, and you actually say, "Okay, this is the checkpoint that I want to use because after this checkpoint, yes, my training loss continues to get better, but it's not actually translating to my testing data set. So, this is the one that I actually should be using."

And we can see that, you know, after the early stopping checkpoint, things do get worse on your testing data. So, you wouldn't actually want to apply those checkpoints. This is such a very powerful idea because it's very general, this approach. You can really apply this to any type of model, and you can really track these these curves, right? It doesn't require architectural modifications. Uh you just monitor the loss of your neural network over time on two different data sets, one that you train on, one that you test on.

Awesome. Okay, I'll conclude the first lecture now. Just summarize quickly uh what we've covered. So, we started with the most fundamental building blocks of neural networks, which was just a single neuron. We scaled that up to single layers and multi-layer perceptrons, multi-layered neural networks. And we've learned how we could build complex, hierarchical learning machines that go from input to output across a hierarchy of abstractions. And finally, we addressed a lot of the practical sides of actually training these models, evaluating them, picking the best model, and so on.

In the next lecture, we're going to be hearing from Alva on covering sequence models, right? This is a really exciting lecture, especially in today's world, because sequence models power uh GPTs, right? Everything a lot of things are sequences in today's world, right? You think of text is a sequence of words, audio is a sequence of waveforms, video is a sequence of images, right? A lot of things are sequences, and what we saw today is only covering non-sequential data so far in my lecture. In the next lecture, we'll see how we can extend that into sequences of data.

Uh so, we'll take a 5-minute break, and then we'll just set up, and then uh we'll continue from there. Thank you.