1 What is Deep Learning

Welcome to this deep learning lecture. My name is Vincent, and I'm a research scientist at Google working on machine learning and artificial intelligence. Deep learning is an exciting branch of machine learning that uses data, lots of data, to teach computers how to do things only humans were capable of before. Myself, I am very interested in solving the problem of perception. Recognizing what's in an image, what people are saying when they are talking on their phone, helping robots explore the world and interact with it. Deep learning is emerge as a central tool to solve perception problems in recent years. It's the state of the art having to do with computer vision and speech recognition. But there's more. Increasingly, people are finding that deep learning is a much better tool to solve problems like discovering new medicines, understanding natural language, understanding documents, and for example, ranking them for search. So let's take a look at what you'll be doing during this class.

2 Course Overview

This course has four parts. Just now, you're going to lay the foundations by training your first simple model entirely end-to-end. For that, we'll need to talk about logistic classification, stochastic optimization and general data practices to train models. In the next session, we're going to go deeper. You'll train your first deep network and you'll also learn to apply regularization techniques to train even bigger models. The third session will be a deep dive into images and convolutional models. The fourth session, all about text and sequences in general. We'll train embeddings and recurrent models. This is going to be a very hands-on course. The assignments are IPython notebooks that you can read and modify easily. Play with them and try things out. We will use them alongside the lectures to make everything you will hear about very concrete. All the code will be written in TensorFlow, a very simple Python-based deep learning toolkit. By the time you are done with these lectures, you'll be able to apply all this knowledge to new problems entirely by yourself with a handful of lines of code.

3 Solving Problems - Big and Small

Many companies today have made deep learning a central part of their machine learning toolkit. Facebook, Baidu, Microsoft, and Google are all using deep learning in their products and pushing their research forward. It's easy to understand why. Deep learning shines wherever there is lots of data and complex problems to solve. And all these companies are facing lots of complicated problems, like understanding what's in an image to help you find it, or translating a document into another language that you can speak. In this class, we'll explore a continuum of complexity, from very simple models to very large ones that you'll still be able to train in minutes on your personal computer to do very elaborate tasks. Like predicting the meaning of words or classifying images. One of the nice things about deep learning is that it's really a family of techniques that adapts to all sorts of data and all sorts of problems, all using a common infrastructure and a common language to describe things. When I began working in machine learning, I was doing research on speech recognition. Whenever I met someone working on computer vision or natural language processing, we had very little in common to talk about. We used different acronyms, different techniques, and collaboration was pretty much impossible. Deep learning really changed that. The basic things that you will learn in this class are applicable to almost everything, and will give you a foundation that you can apply to many, many different fields. You'll be part of a fast growing community of researchers, engineers, and data scientists who share a common, very powerful set of tools.

4 Lets Get Started

A lot of the important work on neural networks happened in the 80's and in the 90's but back then computers were slow and datasets very tiny. The research didn't really find many applications in the real world. As a result, in the first decade of the 21st century neural networks have completely disappeared from the world of machine learning. Working on neural networks was definitely fringe. It's only in the last few years, first seeing speech recognition around 2009, and then in computer vision around 2012, that neural networks made a big comeback. What changed? Lots of data and cheap and fast DBU's. Today, neural networks are everywhere. So if you're doing anything with data, analytics or prediction, they're definitely something that you want to get familiar with. So let's get started.

5 Supervised Classification

This entire course, I'm going to focus on the problem of classification. Classification is the task of taking an input, like this letter, and giving it a label that says, this is a B. The typical setting is that you have a lot of examples, called the training set, that I've already been sorted in. This is an A, this is a B, and so on. Now, here's a completely new example. And your goal is going to be to figure out which of those classes it belongs to. There is a lot more to machine learning that just classification. But classification, or marginally prediction, is the central building block of machine learning. Once you know how to classify things, it's very easy, for example, to learn how to detect them, or to rank them.

6 Classification For Detection Quiz

Let's take an example, the task of detection. Imagine you have a camera in your car looking at the street ahead. You want to detect where pedestrians are in front of you, so that you don't hit them. How would you use a classifier to do that? Type in what approach you would take in the box provided.

7 Classification For Ranking Quiz

Here is another example. Web search ranking. Imagine you have a search query, and you want to find all the web pages on the web, that are relevant for that query. How would you use a classifier for that? Type in what approach you would take, in the box provided.

8 Classification For Ranking Solution

Here's a way to do it. Make the classifier take the pair query and webpage, and output one of two classes, either relevant or not relevant. Of course, it would be a lot of webpages to look at if you ran that classifier on the whole web. But search engines take shortcuts, and only try to classify promising candidates.

9 Lets make a deal

So let's learn about classification. Here is the plan. I'm going to tell you just enough, so that you can put together a very simple, but very important type of classifier, called the logistic classifier. With that, you'll be able to run through the first assignment which will have you download and pre-process some images for classification. And then run an actual logistic classifier on that data. Once you've connected the bit of math that's coming up and the code in that first assignment, everything else will just be building up from there. We will be building together all of the pruning on top of this piece, block by block.

10 Training Your Logistic Classifier

So let's get started training a logistic classifier. A logistic classifier is what's called the linear classifier. It takes the input, for example, the pixels in an image, and applies a linear function to them to generate its predictions. A linear function is just a giant matrix multiply. It take all the inputs as a big vector, that will denote X, and multiplies them with a matrix to generate its predictions, one per output class. Throughout, we'll denote the inputs by X, the weights by W, and the biased term by b. The weights of the matrix and the bias, is where the machine learning comes in. We're going to train that model. That means we're going to try to find the values for the weights and bias, which are good at performing those predictions. How are we going to use the scores to perform the classification? Well, let's recap our task. Each image, that we have as an input can have one and only one possible label. So, we're going to turn the scores into probabilities. We're going to want the probability of the correct class to be very close to one and the probability for every other class to be close to zero. The way to turn scores into probabilities is to use a softmax function, which I'll denote here by S. This is what it looks like. But beyond the formula, what's important to know about it is that it can take any kind of scores and turn them into proper probabilities. Proper probabilities, sum to 1, and they will be large when the scores are large and small when the scores are comparatively smaller. Scores in the context of logistic regression are often also called logits.

11 Softmax Q

Let's play with this Softmax function for a little bit. Let's say you have a classifier which outputs three scores for three classes. My question is, what probability does the Softmax function give you? Define the function softmax(x) to compute the softmax probability values given this set of scores. Let's plot how the probabilities vary as we modify the scores, for example, of the first class. Note that the scores that are passed through the softmax here can be an numpy array with one row for each score, three in this case. And some arbitrary number of columns, one for each sample. Your function should be able to handle such input and return a numpy array of the same shape.

12 Softmax S

Here is an example solution. We take the exponential of the scores and we divide by the sum of the exponential of the scores across the other categories. Let's say testra. Here is the result. Notice that the probabilities do sum to one. Let's add some legends to make this a little bit more clear. As you can see, the probability of the class one increases with the score x. It starts near zero and it ends close to one. At the same time, the probabilities of the other classes start pretty high, but then go vanishingly down to 0. Feel free to go back and play with the softmax function. For example you could draw a bar chart to convince yourself that the probabilities do sum to one for every value of X.

13 Softmax Quiz 2

Here is something to keep in mind about the Softmax. Say you take all the scores and multiply them by 10, what happens? Do the probabilities get close to either 0.0 or 1.0? Or do the probabilities goes close to the uniform distribution?

14 Softmax Solution 2

If you multiple the scores by 10, the scores get either very close to 1.0, or very, very small. Therefore, when you multiply the scores by 10, the first answer is correct.

15 Softmax Quiz 3

Now divide the scores by 10. What happens? Does the probabilities then get close to either 0 or 1? Or do they get close to the uniform distribution?

16 Softmax Solution 3

When you divide the scores by 10 the probabilities become very close to the uniform. One-third, one-third, one-third. When you divide the scores by 10 the second answer is correct.

17 One-Hot Encoding

In other words, if you increase the size of your outputs, your classifier becomes very confident about its predictions. But if you reduce the size of your outputs, your classifier becomes very unsure. Keep this in mind for later. We'll want our classifier to not be too sure of itself in the beginning. And then over time, it will gain confidence as it learns. Next, we need a way to represent our labels mathematically. We just said, let's have the probabilities for the correct class be close to 1, and the probability for all the others be close to 0. We can write exactly with that. Each label will be represented by a vector that is as long as there are classes and it has the value 1.0 for the correct class and 0 every where else. This is often called one-hot encoding.

18 One-Hot Encoding Quiz

Here's some class labels, and they're partially filled out when hot hang putting vectors. Type in the remaining values so that they are consistent, you can leave the zeros blank. Image that your classifier is giving you these predictions. Now, which class is the most likely according to this? Enter your answer here.

19 One-Hot Encoding Solution

For a consistent one hut encoding, we need to pick ones such that each class gets a unique position on the vector. Here's what I picked. Given this encoding the most likely classes is C.

20 Cross Entropy

When hot encoding thing including works very well from most problems until you get into situations where you have tens of thousands, or even millions of classes. In that case, your vector becomes really, really large and has mostly zeros everywhere and that becomes very inefficient. You'll see later how we can deal with these problems using embeddings. What's nice about this approach is that we can now measure how well we're doing by simply comparing two vectors. One that comes out of your classifiers and contains the probabilities of your classes and the one hot encoded vector that corresponds to your labels. Let's see how we can do this in practice. The natural way to measure the distance between those two probability vectors is called the cross-entropy. I'll denote it by D here for distance. Math, it looks like this. Be careful, the cross-entropy is not symmetric and you have a nasty log in there. So you have to make sure that your labels and your distributions are in the right place. Your labels, because they're one hot encoded, will have a lot of zeroes in them and you don't want to take the log of zeroes. For your distribution, the softmax will always guarantee that you have a little bit of probability going everywhere, so you never really take a log of zero. So lets recap, because we have a lot of pieces already. So we have an input, it's going to be turned into logits using a linear model, which is basically your matrix multiply and a bias. We're then going to feed the logits, which are scores, into a softmax to turn them into probabilities. And then we're going to compare those probabilities to the one hot encoded labels using the cross entropy function. This entire setting is often called multinomial logistic classification.

21 Minimizing Cross Entropy

Okay, so now we have all the pieces of our puzzle. The question of course is how we're going to find those weights w and those biases b that will get our classifier to do what we want it to do. That is, have a low distance for the correct class but have a high distance for the incorrect class. One thing you can do is measure that distance averaged over the entire training sets for all the inputs and all the labels that you have available. That's called the training loss. This loss, which is the average cross-entropy over your entire training set, Is one humongous function. Every example in your training set gets multiplied by this one big matrix W. And then they get all added up in one big sum. We want all the distances to be small, which would mean we're doing a good job at classifying every example in the training data. So we want the loss to be small. The loss is a function of the weights and the biases. So we are simply going to try and minimize that function. Imagine that the loss is a function of two weights. Weight one and weight two. Just for the sake of argument. It's going to be a function which will be large in some areas, and small in others. We're going to try the weights which cause this loss to be the smallest. We've just turned the machine learning problem into one of numerical optimization. And there's lots of ways to solve a numerical optimization problem. The simplest way is one you've probably encountered before, gradient descent. Take the derivative of your loss, with respect to your parameters, and follow that derivative by taking a step backwards and repeat until you get to the bottom. Gradient descent is relatively simple, especially when you have powerful numerical tools that compute the derivatives for you. Remember, I'm showing you the derivative for a function of just two parameters here, but for a typical problem it could be a function of thousands, millions or even billions of parameters.

22 Transition into Practical Aspects of Learning

In the coming lectures, I'll talk about these tools that compute the derivatives for you, and a lot about what's good and bad about grading descent. For the moment, though, we'll assume that I give you the optimizer as a black box that you can simply use. There are two last practical things that stand in your way of training your first model. First is how do you fill image pixels to this classifier and then where do you initialize the optimization? Let's look into this.

23 Lather Rinse Repeat

We've just gone over the essential things that you need to be able to train a simple linear classification model. Now, it's your turn. How well does this work? Pretty well, as you will see in your first assignment. The first part of it will have you download and pre-process an image data set that you will be reusing throughout the lectures, and that you can use yourself to experiment with. The second part will have you go through an exact re-implementation, piece by piece, of all the components of the logistic classifier that we've discussed so far. Make sure that you carefully step through the code before coming back to the videos. Run it, break it, tinker with it. We'll be building on top of it for the rest of the lecture. You'll also quickly see some of the problems with the simple approach. It can be very slow. Even for a simple model like this linear model on a very small amount of training data. Scaling this up is what deep learning is all about.

24 Measuring Performance

Now that you have trained your first model. There is something very important I want to discuss. You might have seen in the assignment that we had a training set as well as a validation set and a test set. What is that all about? Don't skip that part, it has to do with measuring how well you're doing, without accidentally shooting yourself in the foot. And it is a lot more subtle than you might initially think. It's also very important because as we'll discover later, once you know how to measure your performance on a problem you've already solved half of it. Let me explain why measuring performance is subtle. Let's go back to our classification task. You've got a whole lot of images with labels. You could say, okay, I'm going to run my classifier on those images and see how many I got right. That's my error measure, and then you go out and use your classifier on new images, images that you've never seen in the past. And you measure how many you get right, and your performance gets worse, the classifier doesn't do as well. So what happened? Well imagine I construct a classifier that simply compares the new image to any of the other images that I've already seen in my training set, and just return the label. By the measure we defined earlier, it's a great classifier. It would get 100on the training sets. But as soon as it sees a new image it's lost, it has no idea what to do. It's not a great classifier. The problem is that your classifier has memorized the training set and it fails to generalize to new examples. It's not just a theoretical problem. Every classifier that you will build will tend to try and memorize the training sets. And it will usually do that very, very well. Your job, though, is to help it generalize to new data instead. So, how do we measure the generalization instead of measuring how well the classifier memorized the data? The simplest way is to take a small subset of the training set. Not use it in training and measure the error on that test data. Problem solved, now your classifier cannot cheat because it never sees the test data so it can't memorize it. But there is still a problem because training a classifier is usually a process of trial and error. You try a classifier, you measure its performance, and then you try another one, and you measure again. And another, and another. You tweak the model, you explore the parameters, you measure, and finally, you have what you think is the prefect classifier. And then after all this care you've taken to separate your test data from your training data and only measuring your performance on the test data, now you deploy your system in a real production environment. And you get more data. And you score your performance on that new data, and it doesn't do nearly as well. What can possibly have happened? What happened is that your classifier has seen your test data indirectly through your own eyes. Every time you made a decision about which classifier to use, which parameter to tune, you actually gave information to your classifier about the test set. Just a tiny bit, but it adds up. So over time, as you run many and many experiments, your test data bleeds into your training data. So what can you do? There are many ways to deal with this. I'll give you the simplest one. Take another chunk of your training sets and hide it under a rock. Never look at it until you have made your final decision. You can use your validation set to measure your actual error and maybe the validation set will bleed into the training set. But that's okay because you're always have this test set that you can rely on to actually measure your real performance

25 The Kaggle Challenge - Higgs Boson

Does splitting your data into three separate data sets sound overly complicated? Let's look at this in action in the real world. Kaggle is a competition platform for machine learning. People compete on classification tasks, and whoever gets the highest performance wins. Here, I'm showing the example of a scientific competition on data that relates to the Higgs Boson. Kaggle also has three data sets, the training data, the public validation data set, and a private data set that is not revealed to the competitors. Here Kaggle shows you the performance of the top competitors when measured on the private test sets. The green and red arrows show how different the ranking was compared to the ranking on the public set. Let's look at the rankings. The top competitors were doing well on the public validation data and they remain at the top once their private data was revealed. If you go further down the leaderboard, however, it's a real bloodshed. Many competitors who thought they were doing well, were not doing well at all in the private data sets. As a result, their ranks went down dramatically once the private data set was revealed. Why is that? Maybe they had a good model that did well in validation just by chance. What's more likely however, is that by validating themselves over and over dozens of times on the validation set, they ended up over fitting to the validation set and failing to generalize. The top competitors] however, had good experimental design. They were not misled into thinking that they were doing well. They probably took out some of the training sets to validate their algorithm or used more sophisticated methods like cross validation and didn't make many decisions based on the public data set scores.

26 Transition Overfitting - Dataset Size

I'm not going to talk about cross validation here, but if you've never encountered it in your curriculum, I'd strongly recommend that you learn about it. I am spending time on this because it's essential to deep learning in particular. Deep learning has many knobs that you can tweak, and you will be tempted to tweak them over and over. You have to be very careful about overfitting on your test set. Use the validation set. How big does your validation and test sets need to be? It depends. The bigger your validation set the more precise your numbers will be.

27 Validation and Test Set Size

Imagine that your valuation set has just six examples with an accuracy of 66Now you tweak your model and your performance goes from 66is this something you can trust? No of course, this is only a change of label for a single example. It could just be noise. The bigger you test set, the less noisy the accuracy measurement will be. Here is a useful rule of thumb. And if you're a statistician, feel free to cover your ears right now. A change that affects 30 examples in your validation set, one way or another, is usually statistically significant, and typically can be trusted.

28 Validation Set Size

Let's do some back of the envelope calculations. Imagine you have 3,000 examples in your variation set, and assume you trust my hand wavey rule of 30. Which level of accuracy improvement can you trust to not be in the notice? A difference from 80a difference from 80to 80.5

29 Validation Set Size

If your accuracy changes from 80that's only at most three examples changing their labels. It's 0.1 x 3000 divided by 100. That's very few. It could just be noise. And it definitely doesn't meet my rule of thumb of 30 examples minimum. Same thing going from 80At worst, only 15 examples are changing then. When you get an improvement of 1going from 80a more robust 30 examples that are going from incorrect to correct. That's a stronger signal that whatever you're doing is indeed improving your accuracy.

30 Validation Test Set Size Continued

This is why for most classification tasks people tend to hold back more than 30,000 examples for validation. This makes accuracy figures significant to the first decimal place and gives you enough resolution to see small improvements. If your classes are not well balanced, for example, if some important classes are very rare, his heuristic is no longer good. Bad news, you're only going to need much more data. Now, holding back even 30,000 examples can be a lot of data if you have a small training set. Cross-validation, which I've mentioned before, is one possible way to mitigate the issue. But cross-validation can be a slow process, so getting more data is often the right solution.

31 Optimizing a Logistic Classifier

With that out of the way, let's go back to training models. Training logistic regression using gradient descent is great. For one thing, you're directly optimizing the error measure that you care about. That's always a great idea. And that's why in practice, a lot of machine learning research is about designing the right last function to optimize. But as you might experienced if you've run the model in the assignments, it's got problems. The biggest one is that it's very difficult to scale.

32 Stochastic Gradient Descent

The problem with scaling gradient descent is simple. You need to compute this gradient. Here is another rule of thumb. If computing your loss takes n floating point operations, computing its gradient takes about three times that compute. As we saw earlier, this last function is huge. It depends on every single element in your training set. That can be a lot of compute if your data set is big. And we want to be able to train lots of data because in practice on real problems you will always get more gains the more data you use. And because gradient descent is intuitive, you have to do that for many steps. That means going through your data tens or hundreds of times. That's not good, so instead, we're going to cheat. Instead of computing the loss, we're going to compute an estimate of it, a very bad estimate, a terrible estimate in fact. That estimate is going to be so bad, you might wonder why it works at all. And you would be right, because we're going to also have to spend some time making it less terrible. The estimate we're going to use is simply computing the average loss for a very small random fraction of the training data. Think between 1 and 1000 training samples each time. I say random because it's very important. If the way you pick your samples isn't random enough, It no longer works at all. So, we're going to take a very small sliver of the training data, compute the loss for that sample, compute the derivative for that sample and pretend that that derivative is the right direction to use to do gradient descent. It is not at all the right direction, and in fact at times it might increase the real loss, not reduce it. But we're going to compensate by doing this many, many times, taking very, very small steps each time. So each step is a lot cheaper to compute. But we pay a price. We have to take many more smaller steps, instead of one large step. On balance, though, we win by a lot. In fact, as you'll see in the assignments, doing this is vastly more efficient than doing gradient descent. This technique is called stochastic gradient descent and is at the core of deep learning. That's because stochastic gradient descent scales well with both data and model size, and we want both big data and big models. Stochastic gradient descent, SGD for short, is nice and scalable. But because it is fundamentally a pretty bad optimizer, that happens to be the only one that's fast enough, it comes with a lot of issues in practice.

33 Momentum and Learning Rate Decay

You've already seen some of those tricks. I asked you to make your inputs zero mean and equal variance earlier. It's very important for SGD. I also told you to initialize with random weights that have relatively small variance, same thing. I'm going to talk about a few more of those important tricks, and that should cover all you really need to worry about to implement SGD. The first one is momentum. Remember that at each step, we're taking a very small step in a random direction, but that on aggregate, those steps take us toward the minimum of the loss. We can take advantage of the knowledge that we've accumulated from previous steps about where we should be headed. A cheap way to do that is to keep a running average of the gradients, and to use that running average instead of the direction of the current batch of the data. This momentum technique works very well and often leads to better convergence. The second one is learning rate decay. Remember, when replacing gradient descent with SGD, I said that we were going to take smaller, noisier steps towards our objective. How small should that step be? That's a whole area of research as well. One thing that's always the case, however, is that it's beneficial to make that step smaller and smaller as you train. Some like to apply an exponential decay to the learning rate some like to make it smaller every time the loss reaches a plateau, there are lots of ways to go about it, but lowering it over time is the key thing to remember.

34 Parameter Hyperspace

Learning rate tuning can be very strange. For example, you might think that using a higher learning rate means you learn more or that you learn faster. That's just not true. In fact, you can often take a model, lower the learning rate and get to a better model faster. It gets even worse. You might be tempted to look at the curve that shows the loss over time to see how quickly you learn. Here the higher learning rate starts faster, but then it plateaus, when the lower learning rate keeps on going and gets better. It is a very familiar picture for anyone who's trained neural networks. Never trust how quickly you learn. It has often little to do with how well you train. This is where SGD gets its reputation for being black magic. You have many, many hyper-parameters that you could play with. Initialization parameters, learning rate parameters, decay, momentum. And you have to get them right. In practice, it's not that bad. But if you have to remember just one thing, it's that when things don't work, always try to lower your learning rate first. There are lots of good solutions for small models. But sadly, none that's completely satisfactory so far for the very large models that we really care about. I'll mention one approach called AdaGrad that makes things a little bit easier. AdaGrad is a modification of SGD which implicitly does momentum and learning rate decay for you. Using AdaGrad often makes learning less sensitive to hyper-parameters. But it often tends to be a little worse than precisely tuned SDG with momentum. It's still a very good option, though, if you're just trying to get things to work. So, let's recap. We have this very simple linear model which emits probabilities which we can use to classify things. We now know how to optimize its parameters on lots and lots of data using SGD and its variants. It's still a linear, shallow model, though, but now we have all the tools that we need. It's time to go deeper.