openSAP
Enterprise Deep Learning with TensorFlow

Week 2 Unit 1

00:00:05	Hello, and welcome back to Week 2 of "Enterprise Deep Learning with TensorFlow". Last week, we got started with deep learning, and we heard about the vision of the intelligent enterprise.
00:00:17	We heard about the development of neural networks, from the early beginnings to deep learning. And we set up Jupyter and implemented a first neural network from scratch, just in NumPy.
00:00:27	Then we introduced TensorFlow, and we saw when to use and when not to use deep learning. This week, we will get really hands-on with TensorFlow, and we'll be building a TensorFlow application end to end,
00:00:39	from data visualization to training a TensorFlow Estimator to serving the model. But before that, in this unit we will learn the nuts and bolts of how to run machine learning experiments.
00:00:54	Developing machine learning models and deep learning models is a very empirical process, and it's quite a bit different from traditional software development processes.
00:01:04	In a normal software development process, you as a developer have well-defined requirements and the outcome of the program is entirely dependent on your code that you write
00:01:13	and whether you have implemented the algorithm correctly according to the specifications. In machine learning, the development process is very empirical and experimental.
00:01:22	It involves a lot trial and error, and the reason is that the outcome of the machine learning model depends mostly on the
00:01:30	and a set of various parameters which you need to tune. The complexity of the problem is much more in the data rather than in the code that you write.
00:01:41	So when building a machine learning model, you have to go through this cycle of collecting data, building a model, evaluating the model, and then iterating to keep making the model better.
00:01:52	And in general, the faster you can go through this cycle, the faster you will make progress in your project. So, let's start by talking about the very important concept of training, validating, and testing a model.
00:02:10	Training a machine learning model means to optimize the model parameters to improve the performance on the training data. And we often refer to this learning, or this training process as the model learning from the data.
00:02:24	After we have trained the model, we evaluate the model on a held-out data set. So we have the model make predictions on new held-out data
00:02:33	and then we compare these predictions with the ground-truth labels. And this is data that the model has not seen during the training phase.
00:02:41	We call this held-out data set the "validation set" or "development set", which is often abbreviated to just "dev set".
00:02:48	We use this validation step to tune and optimize any set of hyper parameters which we need to set manually on the model or the learning process parameters.
00:02:58	Those are parameters like the learning rate, the number of layers in your model, the activation function. All things that the model cannot learn during the training process itself but that you need to

00:03:08	tune and configure when you set up the experiment and the model parameters. And then, when we have a model that really works well on both the training data and the development data,
00:03:23	then we freeze everything in the model and we do a final evaluation run of the model on another held- out data set, which we call the "test set". Evaluating a model on the test set approximates the error that we can expect when we deploy the model to production.
00:03:38	And this is, again, data that the model has never seen before, and that was also not used to tune any of its parameters.
00:03:46	And if the model then performs well on this test data, we can be pretty sure that it would work well in production and ship it to production.
00:03:55	A common beginner's mistake is to evaluate the model on the training data itself, or to fine-tune parameters directly on the test data.
00:04:02	So be very careful not to let that happen to your project, and be very careful not to have any information from the training data leak over into your test data.
00:04:17	Let's look in more detail at the process of how to best organize the data that you have available into these training, dev, and test sets. But first of all, why do we really need these held- out dev and test sets to evaluate the model?
00:04:30	Why don't we just use all the data for training – I mean, more data is always better data for training, right? Well, the reason is that we don't want the model to just memorize the training data.
00:04:41	We want the model to generalize to new, unseen data points. And remember that the dev and test sets are proxies to really estimate how well the model would do in actual production.
00:04:54	So it's very important to split the dev and test sets in a way that they are reflective of both the distribution that you see in the training data and the distribution of the data that you want the model to eventually do well on in production.
00:05:12	So you want the dev and test data to be similar to the training data, and technically, that means it should be coming from the same distribution as the training data.
00:05:21	But in some cases, that might not really be possible. So you will have cases where there is a mismatch between the training data
00:05:29	and the dev and test data that you have. So, for example, the training data that you have might consist of very well-edited formal text from newspaper and newswire agencies,
00:05:41	but the test data and the dev data that you really want to test your model on might be informal text from social media. In this case, we would call this a data mismatch problem between the training data and the dev and test data.
00:05:55	But because you might have so much newswire data, it might still be helpful to use it during the training process to improve the data availability and the model performance overall.
00:06:07	So in some cases, having a data mismatch between the training and dev sets is okay, and it can still help you improve the model,
00:06:14	but what we definitely want to avoid is that we have a data mismatch between the dev set and the test set. So the dev and test sets should always be very similar,
00:06:25	so that whatever result you achieve when tuning on the dev set carries over directly to the test set.
00:06:33	And the test set needs to be representative of the data that you expect in production. Now the question is: How much of your data should you put into training, development, and test?
00:06:47	When data sets were small, people often used splits like 60% for training, 20% for dev, and 20% for test. In some disciplines, people used 30:30:30
00:06:58	and had some kind of percentage split for each of the test sets. But in deep learning, where we often have lots of data,

00:07:06	it is often okay to use a much smaller percentage of the data for the dev and test sets. In general, you just want the dev and test sets to be large enough to allow statistically significant evaluation of your models.
00:07:18	So they need to be large enough to really estimate how well the model would do in production. But once you have dev and test sets that are large enough for that,
00:07:28	you can really use the rest of the data for training. So instead of doing a percentage split like 60:20:20,
00:07:36	you might just take, let's say, 20,000 examples for a development set and test set respectively, and just use the remaining data for training.
00:07:50	So here we see a flow chart that shows a simple process of splitting the data and doing the training, development, and test set evaluation.
00:08:00	So we take all the data that we have, which includes the inputs and the labels, so the inputs and the outputs that we want the model to train on,
00:08:09	and then we take most of that into the training data set, where we train the model – that means optimizing our neural network –
00:08:16	and then the resulting model that we get after the training process is used to make predictions on the dev set, which we then evaluate with the ground-truth labels from the dev set.
00:08:28	And here we can tune parameters, so we can change the number of neurons, the number of layers, the activation function, the learning rate,
00:08:35	until we get a good performance on the dev set. And then we do a final evaluation of the resulting model on the test set.
00:08:45	And we keep doing this training and evaluation, the tuning cycle on the dev set, until we are satisfied with the performance,
00:08:53	and then do a final test set evaluation before we ship it to production. We've stressed several times that the model needs to generalize well to new data,
00:09:07	so let's look at this very important concept in a bit more detail. Generalization, as we have said, is the ability of the neural network to perform well on unseen examples.
00:09:16	And what this means is that the model should capture the systematic underlying pattern in the data, but it shouldn't really be capturing idiosyncratic noise, or outliers.
00:09:26	So if your network is very small, it often leads to a situation that we call "underfitting", where the network is not powerful enough to really represent the patterns that are in the data.
00:09:39	If you look at the example here on the slide, you see a very simple model, just a linear model, a line,
00:09:45	and this model is not really able to separate the crosses from the circles very well. And the model would have a very high error on the training data
00:09:54	because it is not powerful enough to model this non-linear relationship between the crosses and the circle data points. On the other hand, if your network is very large and very powerful,
00:10:09	it could lead to a situation where it just memorizes every minute detail of the training data, including noise and mislabeled data.
00:10:17	Here, in the example on this slide, you see a decision boundary in green that perfectly separates the crosses from the circles, and thus it has zero training error.
00:10:28	But nobody would really think that this is a true underlying pattern in the data, and we wouldn't expect this model to generalize well to new data points.
00:10:38	This is our phenomenon that we call "overfitting". So what we really want is a decision boundary, a model, that looks like this,
00:10:48	where the network finds a systematic underlying pattern in the data without memorizing every minute detail.
00:10:55	And we need to find this right balance in practice between fitting the training data but not overfitting the data, so not fitting the noise in the training data.

00:11:05	One way to improve the ability of the network to generalize and to avoid overfitting is to add a technique that we call "regularization". Regularization techniques add an objective function to the overall optimization process
00:11:21	that makes the learning algorithms prefer simpler models and penalizes unusually large weights in the network. There are different techniques of implementing regularization.
00:11:31	Common techniques include adding the L1 or L2 Frobenius norm of the network weights as a penalty to the overall training objectives. Or more advanced techniques like dropout, which we will see in Week 4.
00:11:50	If we have a model that underfits the data, we often say that the model has "high bias", while a model that overfits the data is said to have "high variance".
00:12:01	The tradeoff between effectively capturing the patterns in the data as well as generalizing to new unseen data points is often called the bias-variance tradeoff.
00:12:13	But how do you know whether a model overfits or underfits the data? Well, when you run the training process of your network,
00:12:20	you should always monitor three essential numbers during the training process. The training error – the error of your model when predicting instances on the training set;
00:12:30	the dev error – which is the error currently made on the held-out development set; and the test error – so the error on the final test
00:12:40	So the training error should overall come down as you train the model and as it sees more data and you run more epochs over the data.
00:12:49	So hopefully the network will learn, improve, and it will approach the lower bound of the optimal error, or base error,
00:12:56	which is the lowest error any model could possibly achieve on this task, on this data set. And in many cases, you can use the performance of humans – so how well do humans do
00:13:07	predictions on these tasks – to estimate a lower bound of this optimal error rate. And although we have few tasks today where deep learning is already surpassing human error rate,
00:13:22	human error is often a good way to establish a lower bound that you want to achieve during training. If you have a large gap between the training error that your model achieves and the human error,
00:13:33	then we say that you have a bias problem and you should probably focus first on reducing the training error of your model,
00:13:40	for example by training a larger model, a larger network, or training longer, and hopefully the model will reduce the training error.
00:13:50	If you have a large gap between the training error and the dev error, then we say you have a variance problem
00:13:55	and you need to focus more on improving the generalization of your model, for example by adding regularization or dropout.
00:14:04	But you will also see situations where the dev error decreases, but then, as you keep training the model, it starts increasing again.
00:14:13	This is a situation that shows that your model is starting to overfit the training data at this point, and either you stop training at this point and don't continue with another epoch,
00:14:28	or you add additional regularization techniques to your model. And finally, if you have a low training error and a low dev error, but you still have a high test error,
00:14:38	then you've probably started to overfit on your dev and you should probably try to increase your dev set so it's more representative of the distribution that you expect in your test set.
00:14:55	So in the end, you want to have a network that has a low training error, a low dev error, and a low test error. And, of course, you want the model to work well in production.
00:15:05	So tuning a model and doing the training process and development process is like tuning different knobs, different switches, and configurations during the training process.
00:15:18	And it turns out there are different knobs and parameters that you can use to either focus on bringing down your training error, so reducing the model bias,

00:15:26	or the dev error, so reducing the model variance. So first you should always try to make sure that your training error goes down,
00:15:34	and if it doesn't, then you can try improving performance by training a larger network, training longer, optimizing parameters like the learning rate, adding techniques like batch normalization,
00:15:49	fiddling with the batch size of your gradient descent, or adding techniques like momentum to the training process.
00:16:01	We'll see some of these techniques in more detail in the following weeks. And, of course, you can always try to find a completely new network architecture that works better on the task you're trying to solve.
00:16:14	So if you have managed to get your training error down and your training error is low, but your dev error is still high, then you should add techniques like regularization or dropout,
00:16:25	or you could try to get more training data to help your model generalize better. If your dev error is now low as well, but you still have a high error on your test data,
00:16:36	then you can increase the test data size to reduce the chance of overfitting to the dev set. Or you might want to check again that the split of the data between your dev and test sets
00:16:47	was correct and that the dev and test set distributions are indeed similar. And if you follow this kind of structured approach on which knobs to tune, a bit like a tuning recipe,
00:16:59	you can often dramatically increase the rate at which you make progress in your project. Okay, so let's explore some of these concepts of tuning a model using the TensorFlow playground.
00:17:14	So this is the TensorFlow playground – we have seen this already in Week 1. It's a browser-based environment in which to play with neural networks.
00:17:23	And let's say we want to solve a classification problem like the one here on the screen. And we can just play with different model parameters and model architectures here in the browser,
00:17:33	to see how they affect the training process. So let's start by just training a very simple network with one layer, without any hidden layers,
00:17:44	with a ReLU activation function, and it should solve this classification problem. And if I start training here, you see that the model has a very high training error – close to 50% – and a very high test error,
00:17:58	and basically it doesn't learn to separate this data. And the problem is that the model here is not powerful enough to separate the blue and the orange dots,
00:18:08	which have a non-linear relationship, and so we need a more powerful model to improve performance on that.
00:18:17	So let's add a few layers. Let's maybe build a model with three layers, with the ReLU activation function.
00:18:29	And let's start again with the training process. And we see that after an initial phase, the training error quickly goes down to about 13%,
00:18:39	and the test error to about 14%, and the model seems to be separating the orange and blue dots reasonably well. And we can now try other parameters.
00:18:51	For example, we could try different learning rates. So we could try a very fast learning rate, a very high learning rate.
00:18:58	But this doesn't seem to lead to a stable convergence of the model, so this doesn't seem to work. We could try a smaller learning rate.
00:19:09	But this seems to be fairly slow and we would probably spend a long time bringing down the training error. And, of course, we can try to add techniques like regularization.
00:19:23	For example, I add an L2 norm regularization here to the model to see whether this helps the model to generalize well to the development data.
00:19:35	So this is just a toy example, but it already shows you that different parameters, like learning rate or the number of layers or regularization, can have quite a significant effect
00:19:46	on the overall training process and the training speed. Okay. Back to the slides.
00:19:56	This is the end of this unit. In the next unit we'll see Josh Gordon from Google,

00:20:03	who'll walk us through a complete end-to-end example, building a TensorFlow application, from visualizing the data to building a TensorFlow Estimator and then deploying that into production.
00:20:16	So, over to you Josh.

Week 2 Unit 2

00:00:05	Hey everyone. In these videos, I'll walk you through using the basics of TensorFlow to classify structured data.
00:00:11	And this is a common scenario, and I hope you can use this code as a template and modify it to solve a problem you care about.
00:00:17	Here's an outline of what we'll cover. First we'll download the data set we'll work with and visualize it using a tool called Facets.
00:00:24	And this is a great way to explore your data and to understand how attributes correlate with the class you'd like to predict.
00:00:29	It's also a useful way to catch errors. Next, I'll introduce TensorFlow Estimators.
00:00:35	And these provide a high-level API we use to train our models. Here we'll use a canned Estimator to train both the linear classifier
00:00:42	and a deep neural network with just a few lines of code. Next, I'll introduce techniques you can use to represent your features,
00:00:48	including bucketing, crossing, and embedding. And finally, I'll point you to resources to learn more about everything introduced here.
00:00:55	You can find the code for this video in the description. Okay, let's get started.
00:00:59	The data set we'll work with is from the US Census. And basically, it's a large CSV file with thousands of rows,
00:01:05	each of which describes an individual. And our goal is to use this data to predict whether a person's income is greater than $50,000
00:01:12	based on attributes like their age and occupation. In this screen cast, I'm working with the Jupyter notebook from the description.
00:01:19	Let me walk you through it. First I'll talk about the imports.
00:01:23	In TensorFlow, we want our code to work with both Python 2 and 3, and these lines help with compatibility.
00:01:28	Our code uses TensorFlow version 1.3, and you can see what version you have installed like this.
00:01:34	Now we'll download the data set using a utility method. It happens to already be divided into "train" and "test".
00:01:39	And if you open up these files using a text editor, you'll see that they're missing a header.
00:01:44	So we'll add one here. This will be useful later as we look up different columns when we represent our features.
00:01:50	Now we'll load the training and testing files using Pandas. And this is a great tool for working with structured data.
00:01:56	Here I'll pop up the "labels" column into a separate object, and this is for your convenience later.
00:02:01	In these lines, I'm converting the label we want to predict into true-false. It's represented as a string in the CSV files,
00:02:07	and you can see a comment in the code to learn why we're converting it. Okay.
00:02:11	Now that we have the data, let's start to understand it. Here we can see we have about 32,000 individuals in train and 16,000 in test.
00:02:18	And we can preview the first couple of lines from the training and testing data using this method. I always do this as a sanity check to make sure the data is in the format we expect.
00:02:28	Now let's visualize the data set. And to do so, we use an open source tool called Facets.
00:02:33	And conveniently, there's a live demo that runs in your browser. And when you run Facets, you can upload any CSV with a header.
00:02:39	As it happens, the census data we're using is already loaded by default. Here's Facets Dive.

00:02:45	In this window, each individual in our training set is visualized as a dot, and we can color them by target or the class we're trying to predict.
00:02:53	Now, individuals who make more than $50,000 dollars are colored in red, and less than $50,000 are colored in blue.
00:02:59	By clicking on a dot, you can see stats about the person it represents. These are the attributes we have to work with, or the columns in our CSV.
00:03:07	So why is this valuable? Well right off the bat, we can see this data set is imbalanced:
00:03:12	There are many more people making less than $50,000 than more. We can learn more about the data too.
00:03:18	For example, let's see how age correlates with income. And to do this, we can facet or partition the data by age.
00:03:24	Here we can see a number of age buckets, and each of these contains individuals that fall into that range.
00:03:30	We can also adjust the number of buckets to make it more or less granular. What can we learn from this?
00:03:35	Well from this data, it's clear that age and income are not in a linear relationship. This tells us we might want to bucket this feature in code, rather than using the raw numeric value.
00:03:45	Now let's also facet by a categorical attribute like education. And right off the bat, we can see that some groups are very likely to have a high income,
00:03:52	like individuals with a professional degree and of working age. You can also color these buckets by another attribute.
00:03:58	Here we'll use "Workclass", which basically says whether someone is employed by the government, or works in the private sector, or is self- employed.
00:04:05	By zooming into each group, you can learn more about the data. For instance, many people in this group happen to be self-employed.
00:04:11	Anyway, I like this tool a lot and find it useful to brainstorm what your features should be. Next, let's use Facets Overview to see summary stats about the data,
00:04:19	and to spot errors that might be difficult to catch later on. Here we can see we have six numeric features, and nine categorical.
00:04:27	And let's look at one example of each. For a numeric attribute like age, you can see stats about the min and max values, and so on.
00:04:34	You can also see how these compare on the train and test set.
00:04:38	Well, when you're using a toy data set like this, you should expect these values to be similar. A good concept to be aware of is "train/test skew".
00:04:46	For example, imagine if we trained a model using this data from 1990, then used it to make predictions today. Because of inflation, the percentage of people who make over $50,000 changes with time,
00:04:57	and this could lead to poor predictions. If your testing data looked different than your training data, you might be able to catch that with Facets.
00:05:04	Facets also shows the percentage of missing values for each attribute, as well as the number of values that are 0,
00:05:09	just in case that's an error for your data. Now let's look at a categorical feature like "target" or the class we want to predict.
00:05:17	Right off the bat, we can spot an error: The string for this column is different in the train and test sets,
00:05:22	and this is why we had to convert it earlier to true-false. Scrolling down, we can see that, oddly, there are many more men than women in this data set,
00:05:29	probably as an artifact of how it was sampled. Okay, now that we have a sense for our data, let's begin an experiment.

Week 2 Unit 3

00:00:05	Here we use estimators to train a linear and deep model on the census data. Estimators provide a high-level API with methods like train(), evaluate(), and predict().
00:00:13	And there are two types: canned and custom. Canned estimators, which I'll cover here, are models in a box,
00:00:19	and basically they're prebuilt implementations of common classifiers. Using one of these lets you focus on designing your experiment and coming up with good features
00:00:28	and hides the low-level implementation details. Custom estimators are appropriate when you want to define your own type of model.
00:00:35	And all estimators support advanced features like distributed training, should you need to take advantage of that in the future.
00:00:41	Estimators also enable you to export models for serving, and they provide integration with TensorBoard, as we'll see in a moment.
00:00:48	First, let's talk about how we feed data to estimators for training and testing, and to do this, we'll use an input function.
00:00:54	An input function is responsible for reading your data, say images or a CSV file, batching it up, optionally preprocessing it, and sending it to the classifier.
00:01:03	If your data happens to be in a common format, like Pandas or NumPy, you can use a prebuilt input function that handles this for you.
00:01:10	And these are convenient for datasets that fit into memory. If you have large dataset or a custom format,
00:01:16	then you should consider writing your own input function, using the new Datasets API. And you can find more on that in the description.
00:01:22	Let's write our input functions – we'll have two: one for train, and one for test. We already have data in the Pandas format, so we'll use the built-in pandas input function.
00:01:31	As you would expect, the training input function receives the training data in labels. And, of course, the testing function will receive the testing data.
00:01:38	Each time an estimator calls this input function, it will receive a batch of 32 examples and labels. And here we set num_epochs to None,
00:01:46	and this tells the input function to loop over the training data indefinitely until the estimator has finished training and stops requesting batches.
00:01:53	We'll also shuffle the data before batching it up. For the testing function, we'll use a single epoch, because we want to make one prediction for each example.
00:02:02	We won't shuffle the testing data because there's no need. Now we'll decide which features to use and how to represent them.
00:02:09	And estimators receive a list of feature columns as a parameter. Each of these tells the estimator how to work with one column from the CSV.
00:02:16	And to start, let's use a simple numeric feature like "age". To do so, we'll write a feature column.
00:02:22	And here, this string corresponds to a column in the CSV file, and the lookup will be done using the header we provided earlier.
00:02:29	We'll specify that we want to use the raw numeric value from this column, and we can append it to the list of features we
00:02:34	Now let's add a simple categorical feature like "education", and the simplest way to do this is to provide a list of the possible values.
00:02:41	How you decide to represent your features is important, and we'll cover this in more detail in the next video. But this is enough to get us started.
00:02:48	We now have everything we need to train our estimator. And here we'll create a canned estimator using a linear model.
00:02:55	We'll specify that we have two output classes we want to predict. And when we initialize the estimator, we'll provide the feature columns we just wrote as a parameter.

00:03:03	We'll also provide a directory for logging so we can visualize the results later in TensorBoard. Notice we'll write the logs for our linear and deep models to different subdirectories.
00:03:13	And here's a tip: If you run this notebook multiple times, you want to delete this directory on disk to restore everything to a clean state.
00:03:20	Training our model is as easy as calling a train method, and we'll pass our input function as a parameter.
00:03:26	And here, we decide to train for 1,000 steps. Each of these will be on a batch of data provided by the input function,
00:03:33	and the estimator will log progress as the model trains. Now once we have a train model, we can evaluate our accuracy on the test set.
00:03:40	And to do so, we'll pass our testing input function as a parameter. With our simple features, we've achieved about 75% accuracy.
00:03:48	And you can see a variety of other evaluation metrics listed here. You can also use the following code to make predictions on individual examples.
00:03:55	And one thing to be aware of is that calling predict on an estimator returns a generator object, and this is useful when you're working with a large amount of data.
00:04:03	Here I'm showing you how to iterate over the predictions that are returned, and how to compare the true label to the predicted label.
00:04:09	If you print out the prediction object itself, you'll also find probabilities for the individual classes. Now let's train a deep neural network and compare the results.
00:04:19	To do so, we'll create another canned estimator. The only additional parameter we'll need to specify is for the hidden units.
00:04:25	And here we'll say we want two hidden layers, with 256 and 128 neurons respectively. The more of these you have, the more capacity your network has,
00:04:34	meaning it can learn more sophisticated functions and model more complex data. The trade-off is it's more likely to overfit.
00:04:42	And how neural networks work is covered in more detail later in the course.

Week 2 Unit 4

00:00:05	Features are the way you represent your knowledge about the world for the classifier. And in this video, I'll introduce different techniques you can use to represent your features
00:00:13	and utilities TensorFlow provides to help. As before, we'll work with the census dataset,
00:00:18	and let's revisit a numeric attribute like age, and think about how we can use it to predict income. If you think about how age correlates with income, our intuition is that as age increases,
00:00:27	usually so does income. And the simplest way to represent this would just be to take the raw numeric value and use that as a feature.
00:00:34	Let's write a feature column for this. In code, age corresponds to a column in the CSV file.
00:00:40	A numeric column indicates we use the raw value as our feature. And now we can add it to the list of features we use to train the model.
00:00:47	Now what can go wrong with this approach? Well, if we think more closely about age, we'll realize that it's not a linear relationship with income.
00:00:54	The curve might look something like this. It's flat for children, then increases during working age, and decreases during retirement.
00:01:01	A linear classifier, for example, is unable to capture this relationship. That's because it learns a single weight for each feature.
00:01:08	To make it easier for the classifier, one thing we might do is bucket the feature. And bucketing converts a numeric feature into several categorical ones, based on the range it falls into.
00:01:18	Each of these categorical features are true/false, and they indicate whether a person's age falls into that range.
00:01:24	Now a linear model can capture the relationship by learning different weights for each bucket. How you choose the number of buckets is up to you,
00:01:31	and ideally you'd want to use your knowledge of the problem to do this well. In code, using buckets is easy.
00:01:37	To create a bucketized feature, we can wrap a numeric column from the CSV with a bucketized column like this. And here we're specifying the number and ranges of the buckets we'd like created.
00:01:47	Once this is done, you can add the bucketized feature to the list. And if you like, you can keep the raw numeric feature as well, or you can remove it.
00:01:54	It's no longer required. Now let's look at how we can represent categorical features.
00:01:59	I'll use the education column from the CSV as an example. The simplest way to represent this is just to use the raw value.
00:02:06	In code, here we'll create a feature that says education can be a single value from the list. As before, this is the column in the CSV file, and we can append it to our features list as usual.
00:02:15	Instead of specifying a list of possible values, we can also read them from a vocabulary file we prepare ahead of time.
00:02:21	If you don't feel like providing a vocabulary list, you can also use a hashed feature. A hashed feature builds up the vocabulary list automatically using a hash function.
00:02:29	Here's the idea. Under the hood, a categorical feature is represented as a one-hot encoding.
00:02:34	There's one bit for each possible value in the vocabulary. With a categorical column, this list is built in advance, using a lookup table.
00:02:42	With a hashed column, the bit for the feature is computed using a hash function. The downside is there could be collisions, meaning different items are mapped to the same value.
00:02:51	And you also lose some interpretability in your model. So why might you want to use these?

00:02:55	Well a categorical feature with many possible values is sparse, meaning most of the values are false most of the time.
00:03:02	If you have many possible values, this kind of representation can be memory intensive. And a hash column can be used to limit the maximum number of possibilities.
00:03:11	Here we'll tell TensorFlow to represent the education column from our CSV file using a hash with 1,000 possible values.
00:03:18	Now another more interesting way to represent a categorical feature is to use an embedding. An embedding is a vector that represents the meaning of a word.
00:03:26	Unlike a one-hot encoding, which is sparse, an embedding is dense. And these can be a more efficient way to represent words from a large vocabulary.
00:03:33	Let's see what an embedding looks like. Here I'm using the embedding projector which is built into TensorBoard.
00:03:39	This is an online demo you can try, and the link is in the description. Here we're exploring a dataset of word embeddings,
00:03:46	and the idea is that similar words should appear close together in this space. For example, all of the words in this cluster are cities.
00:03:53	An embedding is usually high-dimensional, so the embedding visualizer projects this down to 3D. To create an embedding, you can wrap any kind of categorical column with an embedding column.
00:04:03	And here we'll create an embedding for education with 10 dimensions. The embedding is learned automatically in the process of training a neural network.
00:04:10	And you should consider using an embedding if you have a categorical column with a large vocabulary. There are some trade-offs to be aware of, which I've listed here.
00:04:19	Now, it's relatively new to be able to use embeddings this easily, and best practices are still being learned.
00:04:25	Feature crosses are a way to generate new features that are a combination of existing ones. And crossed features can be especially helpful to linear classifiers, which can't model interactions between features.
00:04:36	To work around this, we can take our age buckets from earlier and cross them with the education feature. This generates new true/false features for every combination,
00:04:44	and now the linear classifier can learn a weight for each of them. You have to be careful as you can generate many possibilities quickly.
00:04:52	In code, generating feature crosses is as easy as writing a crossed column. Okay, I hope this was a helpful intro.
00:04:59	Feature columns can be great because they let you experiment with different representations in code and make advanced features like embeddings accessible.
00:05:06	As a next step, I'd recommend you try the code in the description and experiment with different types of features to see how they affect your accuracy.
00:05:13	Consider which features are meaningful to use, and then try to modify this code to work with a dataset you care about.
00:05:19	Thanks for watching, everyone, and happy programming!

Week 2 Unit 5

00:00:10	Hello and welcome to the last unit of this week's course of "Enterprise Deep Learning with TensorFlow." In the last few units, Josh Gordon from Google
00:00:21	showed us how to build a model for classifying structured data using TensorFlow Estimators. And in this unit, we will now dive into the topic of architectures for deep learning.
00:00:33	When people talk about architecture in the context of deep learning, they can actually refer to two different types of architectures.
00:00:41	First, architecture for deep learning often describes the architecture of the deep learning model itself. And second, architecture can refer to software architecture of how you build deep learning applications.
00:00:53	Here let's first talk briefly architecture in the sense of a network architecture. So the architecture of the neural network model, the architecture inside the model.
00:01:05	And here we can roughly categorize neural network models into three very broad classes: So first there are networks that map some form of fixed length input to a fixed length output.
00:01:17	So examples of these types of networks are supervised classifiers that we have seen in this week's unit, and models like feed-forward networks and multi-layer perceptrons.
00:01:30	And then we have a different type of model... model sequence data. These models can possess variable input sequence lengths.
00:01:39	Think of sentences or time series data. Examples of these are neural network language models
00:01:45	where you try to predict the next word in a sentence, given all the previous words. And we will see examples of these types of models in Week 3.
00:01:54	Third, we have convolutional networks where we have a very small filter, a very small part of the network that detects local features in the input data.
00:02:02	And these types of networks are commonly used in computer vision, for example. And you will see examples of these models in Week 4 of the course.
00:02:13	And there's also a distinction between what we call shallow networks and deep networks. If a model has just hidden layer, or just a few hidden layers,
00:02:23	people often refer to the networks as shallow networks. And if the model has a lot of hidden layers, this is usually called a deep network.
00:02:32	And we will see examples of the differences in performance between shallow networks and deeper networks in the next unit.
00:02:42	And you must keep in mind that deep learning research is a field that's moving very fast. And there are new architectures and new types of models being developed all the time.
00:02:52	So these three large categories are not comprehensive, they're not describing all the models that exist.
00:02:58	But it gives you a rough guide to orient yourself when you think about neural network architectures. The other meaning of "architecture" in the sense of deep learning
00:03:12	is the architecture of building a deep learning software system. So how does deep learning work together with the overall software stack
00:03:21	in your enterprise application? Let's quickly recap here that we have very different phases in the lifecycle of a model.
00:03:30	First we have the training phase, and then we have the phase where you want to use the trained model to perform inference in production.
00:03:38	And these two phases have very different workloads. Training deep learning networks is very compute and is data intensive,
00:03:45	and is also often very slow. Training large networks can sometimes take days, or in some cases even weeks.
00:03:53	And you usually have a large base of parameters that you need to explore when building the model. So there's a lot of trial and error and iterating while trying to build the model.

00:04:04	Once you have the model and you want to serve it in production, it's all about serving prediction requests that arrive at the model from the application.
00:04:13	And that is generally much faster than the training phase, where it only involves a single forward path through the network.
00:04:21	And the limiting factor in deep learning is really the huge number of matrix multiplications and tensor operations that you need to perform,
00:04:30	especially during the training phase. And it turns out that this huge amount of mathematical operations
00:04:36	is perfectly suited for modern hardware architectures like graphic processor units (GPUs). GPUs differ from the CPUs in your computer
00:04:48	through the fact that they have up to thousands of cores that are very effective in computing matrix operations and mathematical operations in an extremely parallel manner.
00:04:59	They're not as general-purpose as CPUs, which can do so many different types of instructions, but they're very good in the simple mathematical operations.
00:05:09	And so GPUs are arguably really what powers most of deep learning systems today. If you look at a company like Nvidia, they have really shifted their focus from producing graphic cards for gaming
00:05:21	to really focusing on deep learning and AI. And you will hear from some of the deep learning experts at our great partner Nvidia
00:05:29	in Week 6 of this course. There are now even a lot of developments in the industry
00:05:35	to produce special types of hardware only targeting deep learning. Google, for example, has built its own chip that is only specialized in executing deep learning instructions
00:05:44	called tensor processing units (TPUs). But now when we want to serve a model in production,
00:05:52	we often have the situation that it can run on very different architectures. So you might run it on a normal CPU, you might run it on a GPU,
00:06:02	or you might run it on completely different types of devices like a mobile phone, let's say. So with TensorFlow, we have seen that the architecture provides this device agnostic execution framework
00:06:14	that provides a layer of abstraction between the hardware-specific kernels and the various language front ends and higher- level APIs
00:06:21	that you as a developer can use to build your application. But once you have trained and tested your model,
00:06:32	the next step is really to move it to production. And this is an aspect that is not so much discussed in a lot of deep learning research papers,
00:06:40	but it is crucial for building applications and enterprise applications because here you really want to make your model available to the rest of the application,
00:06:50	and not just stop at evaluating your model in research. And this is something really great about TensorFlow.
00:06:57	TensorFlow has these end-to-end capabilities, providing you with libraries to go all the way from prototyping, development, to running models in production.
00:07:07	So TensorFlow serves a software library from Google, which provides a high-performance serving system for machine learning, in particular TensorFlow models.
00:07:16	So TensorFlow Serving is really what helps you to serve requests from client applications to your model, and it also helps you manage the whole lifecycle of the model.
00:07:25	So this is really a great thing about TensorFlow, that it's really built for production usage,
00:07:29	including this complete end-to-end process of shipping the models to productions. After you have trained your TensorFlow model,
00:07:38	you typically persist the model in a serialized form, usually on disk or some other storage service.
00:07:44	And then will typically not just have one model but you'll probably have different versions of your model

00:07:51	and you have different versions of different models. You need a system that can help you manage this lifecycle of training different models,
00:08:00	versioning them, and then serving specific versions of these models in production. So what TensorFlow Serving does is that it loads these models from disk
00:08:11	and then serves these models to the client. That means it helps to answer requests from the client doing predictions using the models.
00:08:21	So very simplified – you can think of TensorFlow Serving as a kind of Web server
00:08:26	that helps you to serve inference requests to your TensorFlow models. Before we move on in the topic, let's look at a quick hands-on demo of TensorFlow Serving.
00:08:40	Here we have a Jupyter notebook that shows a small example of how to use TensorFlow Serving. For more comprehensive documentation and advanced examples,
00:08:49	you can refer to the TensorFlow Serving Web site. So we start by training a small and simple neural network model.
00:09:00	We will use the Iris data set. It's a very commonly used UCI benchmark data set for classification.
00:09:10	So in the beginning, we just need to download the necessary files, "iris.training.csv", which holds the training data in "iris_test.csv.",
00:09:20	which holds the test data. So we use w_GET to download these files.
00:09:27	And then we build a simple TensorFlow Estimator using the Estimator DNNClassifier class that we have already seen in Josh's examples.
00:09:37	So these are helper classes for building these: tf.estimator.DNNClassifier class,
00:09:43	here is the number of hidden units that we use, and this is the number of output classes.
00:09:52	And then we build some helper functions for performing the loading of the training data, as well as the test data.
00:10:06	And then we have a helper function that specifies for TensorFlow Serving how the input to the servable – a servable is basically the basic unit of TensorFlow Serving –
00:10:18	it's an object that you can feed with data and it will give you a return value, which is either a prediction by a model, or it could also be computed by a lookup
00:10:28	or some other form of inference. So here in this cell we then finally do the training of the model.
00:10:36	So first we create the Estimator, then we feed it the training data.
00:10:43	And here in the log, we can see how the model trains, how it improves over time. And finally, we get a fairly accurate classifier
00:10:54	that can predict the Iris test data with 96% accuracy. So this would be the stage when we are confident with the model:
00:11:03	We have trained it, we have tested it, and it performs well. And we now want to ship it to some client application.
00:11:12	So first we have to export the model. So this is a step where we persist the model to disk, or to a serialized form.
00:11:19	And we do this with the "classifier.export_savemodel" function. And you see that the model has been successfully saved to this path,
00:11:30	and is now persisted with a proto bar file. Next we just have some helper function that helps us maintain the session that we use to run the model,
00:11:42	and a helper function to load the model. And what we can basically do now is the load the exported model back and restore a session
00:11:53	that we can again use to run inference with the model. So here we can just do some debug operations that help us look at the model that we've just loaded,
00:12:06	understand what operators the computation graph has, and also confirm for us that we have successfully loaded the model.
00:12:15	So here we just list all the operations in the TensorFlow graph that we've just restored. And if you now want to use this TensorFlow graph,

00:12:27	we have to specify what the inputs are that go into the model, and what the predictions are, the outputs that we want to predict.
00:12:37	So in this case, we take the "input_example_tensor". "outputs" at 0 – that is the placeholder for the x.
00:12:46	And we basically just have a variable here that is a placeholder variable where the input should go.
00:12:52	And the same here – this is a variable that shows us where the prediction of the model would come out. And now we can go ahead and use this model to make new predictions.
00:13:03	In this case, we take a feature vector. The Iris data set consists entirely of numerical values.
00:13:11	So this is a feature vector taken from the test data. And we encapsulate it in a serialized data format
00:13:22	that we can then feed into this model that we run in a session by providing it, the test data, as a value for the "input_x_holder" variable.
00:13:35	So this is where the data goes in. And then we make a prediction and we can get the output as a score here.
00:13:44	So here you see the output it correctly predicted as class 2. And we can even then do the argmax over this to get the right prediction.
00:13:55	So this is a very simplified example, a very simple example of how to use TensorFlow Serving. In production, where you really want to run it on a larger server,
00:14:03	and search for things over HTTP and over the Web, It needs a little bit more work where you have to set up a Docker container and set up the server.
00:14:15	We don't have time for that in this unit, but you can refer to the TensorFlow documentation on the Web site.
00:14:23	Let's go back to the slides. So...
00:14:29	In a way, a trained model like we've just seen is a little bit like a black box that you want to use to make a prediction.
00:14:37	So you put in data and you get a prediction back. And a common architecture style to build applications, which is machine learning models,
00:14:46	and to build modern enterprise applications in general, is to have this vertically integrated targeted functionality implemented as microservices.
00:14:57	So having several vertical integrated services, exposed via APIs, and then have your application orchestrate these loosely coupled services to solve the overall business problem.
00:15:09	And these services are often in build and are operated on cloud platforms, which provide the ability to share both expensive GPU resources among different processes and applications,
00:15:20	but also to provide the elastic compute resources that you need for the very different workloads, especially during the training phase and the testing phase of a model.
00:15:31	So we'll take the SAP Leonardo Machine Learning Foundation, which is the cloud-based machine learning platform of SAP,
00:15:38	as an example of how to build enterprise machine learning applications. So at this point, you see a high-level architecture.
00:15:46	So you have the enterprise application sitting on top. And developers can use a platform to either just use existing machine learning services on the platform
00:15:58	to retrain models with their own data by customizing the model without building it from scratch, or to build completely new machine learning models –
00:16:09	either by training a model offline on their own devices, on their own computers and servers, and then deploying it to the Cloud Platform for serving,
00:16:17	or doing the entire training and deployment process on the cloud. Let's have a look at how to consume and deploy a machine learning model to the SAP Leonardo Machine Learning Foundation.

00:16:36	Here's an example of how to bring your own model to the SAP Leonardo Machine Learning Foundation platform. I already ran the example because the whole deployment process takes a little bit too long to show it live,
00:16:48	but we have basically started by uploading the inception computer vision model to the Cloud Platform. So we have uploaded the serialized model file.
00:17:01	And then, once the model was uploaded, I asked the machine learning platform to deploy the model.
00:17:09	And I can look at the model status by saying I want the description of my model. So initially, you see that the model is pending.
00:17:20	But after a little while, the model has successfully been deployed. And then I can go ahead and push my machine learning application.
00:17:34	So in this case, it's a very simple Python application that just self-requests to the inception model. This is a Cloud Foundry application that I push to the platform,
00:17:47	and it starts deploying the app, creating the route, doing the necessary bindings, installing the build packs.
00:17:56	And after that is done, it gives me an "OK". And I'm ready to use an application.
00:18:05	So I need the necessary access tokens. And then I can go ahead and call the machine learning model from client application,
00:18:15	so in this case I just use "curl" to do a HTTP post with an image. And I just get a response back from the model – what kind of classes,
00:18:26	what kind of object the inception model has found in the picture. So in this case, the model's top prediction was that it's the poodle.
00:18:38	And we can just convince ourselves that this is correct. And indeed, this looks like a poodle, so the model made the correct prediction.
00:18:58	This already concludes Week 2. In Week 3, you will learn why we need deeper networks,
00:19:04	and we will also learn about sequence models – these kinds of models that can process inputs of variable lengths.
00:19:10	I wish you good luck for the weekly assignment, and see you next week.

www.sap.com