# openSAP
# Enterprise Deep Learning with TensorFlow

**Week 03 Unit 01**

00:00:08 Hello and welcome back to Week 3 of "Enterprise Deep Learning with TensorFlow". This unit is called "The Need for Deeper Networks".

00:00:19 Last week, we learned the nuts and bolts of how to set up experiments for deep learning and we saw an end-to-end example of building a classifier for structured data using TensorFlow Estimators.

00:00:30 That went all the way from data visualizations, to training the model and feature engineering, and then finally, serving this model to production using TensorFlow.

00:00:41 We also talked about the architecture principles for deep learning applications, and how deep learning works together with the overall software stack when building end-to-end applications.

00:00:54 In this unit, we will dive deeper and understand the reasons why we need deep networks, and we will also get to a hands-on experiment with our own deep neural networks.

00:01:07 But first, let's look at shallow networks and understand the limitations of these types of architecture. Shallow networks, or wide networks as they are commonly referred to, are very good at memorizing the data.

00:01:20 This is due to the large number of neurons that can effectively remember the input data that is fed to the network during training. While this could lead to very low training errors, it results in poor generalization to new data,

00:01:36 which is known as "overfitting", and we have already talked about this. On the other hand, deep networks, where we have lots of layers of hidden neurons and that might have fewer neurons overall,

00:01:51 often generalize better because each layer can learn better and higher-level abstractions of the data that capture complex interactions in the data.

00:02:02 Deep networks are often more complex in their design due to the larger number of hyperparameters, such as the number of neurons per layer, the number of layers, and so on.

00:02:13 but these different layers really help the model abstract and learn better feature representation. So, for example, a deep neural network could learn to classify a face by learning first

00:02:26 local representations of edges and curves in lower layers of the network, and then higher levels of the network could learn features like eyes, nose, and a mouth to then overall classify an image as a face.

00:02:43 And as a result of learning these abstractions, deep neural networks are generally less prone to memorizing the data in the training set. And it's important to understand that this does not mean that neural networks never memorize data.

00:02:57 But they're good at learning non-linear variations of the inputs and they have a lesser probability of just memorizing the data that is given to them during training.

00:03:10 So let's look at the need for non-linearity in neural networks. Deep neural networks, no matter how deep, without non-linear activations effectively just behave like a single-layer neural network,

00:03:23 and this is because of the linear combinations in the different neural activations. Overall, it would just form one big linear function if there were no non-linearities within each neural activation.

00:03:38 Another point to note is that complex data cannot often be separated with simple linear transformations owing to the distribution of the data across various subspaces.

00:03:49 Non-linear activations in neural networks allow mapping the input data into spaces where they can be possibly linearly separable. In the example on this slide, a linear function would not be able to separate the circles from the crosses,

00:04:02 and we would clearly need a non-linear model to separate these data points. Let us look at the learning process in a neural network in a bit more detail.

00:04:15 There are two main processes, two main phases during the training process, that the network uses to learn. One is called the forward pass, and the other one is called the backward pass.

00:04:26 In the forward pass, information moves forward through the network, from the input to the final output layer, through all the intermediate hidden nodes and hidden layers of the network.

00:04:38 So we start with the input data, and we compute the activation scores of each layer and pass it on to the next layer. Using the scores evaluated during the forward pass, a neural network then evaluates the error

00:04:51 contributed by each neuron on the current training batch, and then this error is propagated backwards through the model, through the backpropagation phase.

00:05:03 During the backpropagation, the weights of the network are modified in a way that the error is decreased. And this is really the learning phase, the learning update for how the model trains on the data.

00:05:21 Let us look at some of the learning objectives that are widely used when designing a neural network. First, there is the loss, which is the error made by the model on a single data point,

00:05:35 and it measures the difference between what the model is currently predicting and the true value that we want the model to predict. Another term that is often used is the cost function,

00:05:48 which is basically the average of the loss function over the entire training data. So the loss is on a single example, and the cost is the overall error made by the model on the training data.

00:06:01 Then there is a commonly used final layer of the model which is called a softmax classifier or a softmax layer. Our softmax layer is used to convert the final scores of the activations in the model

00:06:16 into probability values for each class in the output of the model. For building a neural network classifier, we often convert the final output into these

00:06:26 probability vectors over the classes to basically say how likely the model is to predict each class. And then we can use the final softmax activation, the probability prediction for each class,

00:06:42 to compute the loss of the predictions made by the model compared to the true labels on the training data. And this loss is then used as the initial step to optimize the model in the backward propagation.

00:06:56 That leaves us with a choice of an optimization algorithm to compute the exact updates and perform the updates in the backward propagation step.

00:07:05 One of the most common optimizers used is what is called mini-batch stochastic gradient descent. This is the optimization technique used for the model parameter updates when training the network.

00:07:20 A stochastic gradient descent is an approximation of the overall gradient descent optimization that is used to minimize the objective function on the training data.

00:07:33 So mini-batch stochastic gradient only optimizes on a small batch of a small part of the training data at a time to minimize the overall objective, and iteratively approximating the cost and the gradient of the entire training data,

00:07:49 which makes it much faster in practice than computing the loss and the gradient over the entire dataset, which is called batch gradient descent.

00:07:59 And batch gradient descent needs to look at the entire training dataset before making a single weight update to the network, so stochastic gradient descent is much preferred in practice,

00:08:10 unless you have very small training datasets. Stochastic gradient descent is commonly used for optimizing and training neural networks

00:08:19 because it's simple and is an effective algorithm that works across a lot of different datasets. But there are more advanced optimizers available today – algorithms like RMSprop or ADAM –

00:08:32 which you can easily use in TensorFlow but which we will not cover in this unit. We will now move on to a practical example using a Jupyter notebook

00:08:45 to show how deep neural networks perform compared to shallow neural networks. So here is a Jupyter notebook example that shows how to train a deep network for the ZALORA fashion MNIST dataset.

00:09:03 This is a great dataset made available by ZALORA with small black-and-white images of fashion articles from 10 different classes.

00:09:14 So here in the notebook, we start by first downloading the dataset and unpacking it, and then separating the data into a training and a test set for both the input data and the labels.

00:09:34 We then continue to import some necessary libraries, and we can look at the data by just plotting some images from the data.

00:09:43 You see here the different labels of the dataset. So there are 10 classes, from t-shirts to bags to ankle boots.

00:09:51 And here is an example of the dataset showing an image of a sneaker with the correct label that comes with this image saying it's a sneaker.

00:10:04 What we'll do in this example is train three different types of networks with different numbers of hidden layers, and we will see how they perform in classifying these images into their respective classes.

00:10:20 Here are a few helper classes that help us initialize the input data in our Tensor graph, as well as the output labels, and to initialize necessary parameters for TensorFlow.

00:10:34 We then define this helper function that helps us create a single hidden layer in the TensorFlow graph, using TensorFlow scopes to neatly separate different layers

00:10:44 when we want to view the results in TensorPort later. I'll just point out these two lines where we really create the weight matrix W, as well as the bias b.

00:10:59 And then the activation of the layer is the multiplication of the weights and the inputs plus the bias, which gives us z. And then the activation of the neurons is put through a provided activation function, which gives us the non-linearity in the model.

00:11:22 From this, we move on to first define a one-layer neural network with no non-linearities. So we have a single hidden layer defined here,

00:11:34 which takes the input data X and outputs the hidden layer. And the hidden layer is then fed into the output, which are the logits.

00:11:47 And from there on, we can compute the entropy, which gives us the loss, which we can then use for optimizing the neural network using the TensorFlow GradientDescentOptimizer.

00:12:04 And finally, here is a function that helps us to evaluate the model performance by comparing the predictions with the correct labels.

00:12:18 So from here, we move on to train a simple network with one layer, and we iterate over the training dataset in mini- batches to then train the model,

00:12:36 and then we also evaluate the model in every run on the test data to see how both the training error and the test data develop through the course of the training.

00:12:48 When we look at the log of the training run, we see that the training error comes down. So the accuracy, which is 1 minus the error, increases from initially 76% to a total of 94%.

00:13:02 So the model learns to fit the training data, and the test accuracy also increases from about 80% to 84% after 20 epochs over the training data.

00:13:18 We can also evaluate the neural network by looking at the predications and inspecting it manually. So here I've plotted a few images from the test

00:13:30 together with the predicted label and the actual label in the dataset. So this image here was predicted as a coat but it's actually a shirt,

00:13:41 This was correctly predicted as a pullover, this was correctly predicted as a sandal. This was predicted as a pullover but it was a coat, and this was predicted as a dress but it was also a coat.

00:13:55 So you see, the model learned something quite reasonable, but it still makes some errors on the data. The next example then shows a one-layer neural network, but we use non-linearity in the data.

00:14:07 In this case, we use a linear rectifier. The overall code doesn't change much.

00:14:14 The only thing that really changes is that we use the TensorFlow ReLU function here as the activation function. And, basically performing that same training and evaluation method,

00:14:27 we see that the accuracy of the model has improved to 85%. And again, we can look at some of the examples in the test set to see where the model still makes errors.

00:14:42 And we can see that it still makes a few errors: Again, here, this is predicted to be a dress although it's a coat.

00:14:50 Although arguably this is really quite close and even some humans might say that this could very well be a dress. Finally, we have a model with three layers, and so this would be an example of a deep network

00:15:07 where we hope that the model learns better intermediate representations and features of the data during the training process. So here the difference is that we create three hidden layers

00:15:23 where each hidden layer feeds into the following hidden layer as the input. Again, we train the model using mini-batch gradient descent,

00:15:37 and after 20 epochs using the same training procedure as before, this model has increased to 87% test accuracy.

00:15:48 This improvement here is not dramatic as even this linear model does reasonably well. But it shows you how a deeper network is able to get to both a higher training accuracy as well as the higher test accuracy

00:16:02 compared to the single-layer network that we have compared it to. And again, finally, we can look at the examples – here again, there are some errors.

00:16:15 Again, this was predicted to be a coat but it's still a shirt; the sandals are still correct; but at least we finally managed to realize that the last image is indeed a coat, as the model predicts here.

00:16:28 Finally, here is a way to visualize the graph that we have just built. We can do that here within the notebook,

00:16:38 and we can visualize and interactively explore the TensorFlow compute graph of our model, which is a very helpful feature when debugging your model

00:16:51 and trying to understand where the architecture might be wrong. So I greatly encourage you to use these features in TensorFlow when you're building your own models.

00:17:03 Okay, that was "The Need for Deeper Networks". In the next unit, Steve Jaeger will be introducing sequence models to us,

00:17:12 and we will learn how to build models and process non-fixed input lengths in variable inputs in time series data. Steve, over to you.

**Week 03 Unit 02**

00:00:07 Hello everybody. I am Steve. Welcome to the second unit of Week 3 in our Enterprise Deep Learning course.

00:00:13 In today's session, we will introduce a very important data type for many applications in business and beyond: sequential data and models that can actually leverage their potential.

00:00:26 But first, a quick recap. In the last unit, we learned about feed-forward networks, how to train them, and how to make them deep.

00:00:34 And we will build on that knowledge. And you want to make sure that you understood the basic concepts from that unit

00:00:40 before we move on to the second part. So today we will have a gentle introduction to sequence models,

00:00:48 and have a look at the characteristics of sequential data and applications. Let's start right away.

00:00:55 The textbook example of sequential data is time series. In prior neural networks, we assumed all inputs without an inherent order.

00:01:03 For some tasks, that might be fine, but for others, that's a really bad idea.

00:01:09 We will, for example, assume that the products we sell over the year follow a certain temporal pattern. Without keeping them in sequence, all this information would be lost.

00:01:20 Another prime example of sequential data is natural language if we just considered text as a sequence of words or characters.

00:01:29 Yeah, let's consider this toy example of predicting the next word in a sequence. In this scenario, the context of the previous words surely matters.

00:01:38 And yeah, it's not as easy as it might seem in the first place, as we usually have multiple and ambiguous possible outcomes,

00:01:45 and a variable length input sequence, which complicates things further. So the takeaway here is that when we handle sequential data,

00:01:56 we usually have two things to keep in mind. One is the inherent order of the data,

00:02:01 and the other one is the variable length of the sequence, and that we need to deal with that somehow.

00:02:11 In most cases, we want to learn a model from our sequential data that is able to predict the next element in that sequence.

00:02:18 So among the many factors that may impact the next element, we can differentiate between external factors and the sequence itself.

00:02:26 External factors can play a very important role. For example, when we predict sales numbers for certain products,

00:02:32 the weather can explain a lot of the patterns we observe. The same applies for text,

00:02:38 where we take into account the conversation as a context, or the speaker. Our focus will be primarily on the sequence data itself.

00:02:48 And although this simplifies things for us, this does prevent a prediction of a sequence usually coming along

00:02:55 with some error that is not attributed to a model, but it's just random noise.

00:03:00 And if we assume the next word to be a function of the previous words, as we do here, there's still some kind of unexplained noise left.

00:03:11 So how would we set up a typical sequence prediction task? Let's take the task of predicting the next word in a sequence.

00:03:20 We assume that we have a fixed vocabulary of V different words that can appear. Now the task of our model is to compute the probability of the next word,

00:03:30 given a prior sequence of words. Once we come up with a model that is capable of doing so,

00:03:35 we just need to output the word that maximizes this probability. Now, how can we come up with such a model?

00:03:44 Our standard probabilistic language modeling approach is our "n-grams". They answer the question we posed before:

00:03:52 Given a sequence of words, what is the likelihood of the next word being Wt+1? We infer this probability distribution by just raw counting all n-grams occurring in our training data.

00:04:07 So how many words we take into account is defined by n. So if we take the example here, "the cat is",

00:04:15 and before this we processed a book about how great cats are, the chances are good that our n-gram model will assign a high probability to "cute",

00:04:23 which is right in this case. And given the word "the cat is",

00:04:27 which is, by the way, a different n-gram than "is the cat". So order matters here.

00:04:33 It is important to note that we therefore pose an independency assumption over our corpus, by saying: The next word only depends on the last n words.

00:04:42 You might have guessed it – long values of n become very memory- consuming and expensive to compute due to combinatorics.

00:04:49 As most word combinations won't occur at all, we are left with a very large and very sparse matrix.

00:04:57 As a rule of thumb, assume n = 3, which is a trigram for large corpora, and n = 2, which is a bigram, for small corpora in most practical implementations.

00:05:13 So n-grams are only able to take into account dependencies if they happen to be in the range of n. And this is a harsh drawback, especially for natural language processing,

00:05:22 and can be a real game breaker for some applications. So in this example, "the cat on the piano is",

00:05:29 this will lead to a wrong prediction as we are only taking into account the last three words, and our model does not help us in predicting the right attribute of the cat,

00:05:39 but rather initiates a complaint about how expensive our piano was. So yeah, another model that kind of copes with long-term dependencies in its own way is the "bag of words".

00:05:52 A bag of words is exactly what its name suggests. We throw all words in a sequence in a bag so we know that they are there.

00:06:00 But we do not know which one was put there first, last, or in the middle. So every sentence can be represented with a fixed-size vector of dimensionality V

00:06:11 that is built upon the count of each word in the sentence. We can feed this representation of the sentence or the document to any kind of model

00:06:20 that we train over our corpus. The good thing about bag of words is that we now have a fixed-length vector that represents the word count.

00:06:28 On the other hand, compared to our n-gram model, we have lost the key characteristic, and that is the ordering.

00:06:38 As we will also see later, the word vector is inherently connected to the meaning we want to convey in a sentence. So yeah, let's look at this example:

00:06:46 "the cat caught" + "mice" and "the caught cat" + "cried". So for the cat, in this example sentence,

00:06:52 it definitely matters whether you are the hunter or the hunted. So we trade a fixed size and easy-to-compute representation for the semantics of our words,

00:07:02 which is not exactly a good deal. And this is exactly where sequence models come into play,

00:07:09 and we want to show that we can do better. The key idea in the following sentence models

00:07:16 will be to learn a representation of the past part of a sequence that carries information from past sequences with it.

00:07:24 So instead of conditioning our model on all previous words themselves, we encode prior parts of the sequence in a single vector, "h".

00:07:32 That represents the information that really matters for our task at hand. So this is also helpful for the signature of our resulting function.

00:07:43 And instead of a variable length input, we compress all relevant information for prediction of the next word,

00:07:50 or the next element of the sequence, to a fixed vector.

00:07:53 Our aim is to incorporate the best qualities of the previously presented models. And what are applications of these sequence models?

00:08:05 Such models can be very powerful and perform difficult tasks, like text classification, in order to categorize documents,

00:08:14 which also includes sentiment analysis, to extract the sentiment of a text, a customer e-mail, no matter how long or short the input text is.

00:08:26 Another very important use case that most of you have probably already used yourselves is machine translation, where you translate one sequence of words to another sequence of words

00:08:35 with potentially different sequence lengths for input and output. As in this example – when you translate from English to German.

00:08:48 So yeah, in order to truly leverage such models, we need to talk about how we want to represent the input text that we give to it.

00:08:56 And we will talk about this in the next unit. And until now, the sequence model is still a kind of black box,

00:09:04 but we will dedicate the rest of Week 3 to how to create a sequence model with deep neural networks. And yeah, that's the content for our next unit.

00:09:15 So we will talk about more powerful and easier- to-handle representations of words, and how we can create them with a simple neural network called Word2Vec.

00:09:25 Thanks and see you in the next unit.

**Week 03 Unit 03**

00:00:08 Hello everybody. Welcome back. This is the third unit of Week 3 in our Enterprise Deep Learning course.

00:00:14 Today we will talk about vector representations of words and how to learn them. In the last unit, we got an introduction to sequential data, possible modeling approaches, and applications.

00:00:27 We also touched on the difficulties of representing textual data briefly. We will have a deep dive on this topic today as it is a very important prerequisite for sequence models.

00:00:39 So, in today's unit, we will learn how to represent text as an input and how we can make our models learn those representations on their own with a model called Word2Vec.

00:00:52 As we discussed in the last unit, we have a lot of powerful use cases in natural language processing like text classification, machine translation, sentiment analysis, and so on, which all have high business value.

00:01:03 For all those tasks, we need to find a numerical representation of text to train our model with, be it a neural network or also a traditional machine learning algorithm.

00:01:15 Although you can also work with text at the document or character level, we will stick to words for later in the unit and also for the week.

00:01:25 The simplest way to represent words is what is known as "one-hot encoding". Given a vocabulary V, every word is converted to a vector of length V

00:01:35 that is 1 at the index of the word we want to represent and 0 otherwise. This is pretty straightforward, but it comes at a cost.

00:01:48 When we compare one-hot vectors by calculating distance or similarity measures for vectors, those are meaningless. The one-hot vectors do not share any kind of similarity as all information for each word is just stored in one dimension.

00:02:03 It is not only counterintuitive for humans that the vector of "monkey" is as close to the vector of "gorilla" as it is to the vector of "table". It's also bad for our modeling efforts because

00:02:16 our models would also profit from having semantics encoded into our vectors. Another disadvantage of one-hot vectors is that each additional word in the vocabulary spans another dimension.

00:02:29 With several thousand or even millions of tokens in a language, we have to handle very large and very sparse vectors.

00:02:41 So with this in mind, wouldn't it be great to have a more dense representation of words that also captures semantics? Let's think of a hypothetical example.

00:02:51 Imagine we want to arrange all royal titles and houses in a vector space. The naïve approach would be to create a different dimension for each title by storing all information in just one dimension.

00:03:03 But what if we tried to distribute the meaning across several dimensions? This is what we call a "distributed" representation, in contrast to a "localist" representation, such as one-hot.

00:03:14 We could have a handful of latent features for royalty, age and so on to allocate our royals in a much smaller space with continuous values.

00:03:25 For example, you would have the vectors for "King" and "Prince" scoring similarly at the masculinity feature, but very different at the "age" feature.

00:03:34 In addition, each additional title like "Princess", "Duke", or "Duchess" can be represented in this small space. Just to be clear, this is just a toy example,

00:03:44 and our real distributed representations won't be as easily interpreted as this. So take this with a pinch of salt.

00:03:51 Luckily, we don't even need anybody to encode such features by hand or come up with engineered features – we have a neural network to do the job for us.

00:04:00 And to get it to work, we need to teach it what "similar" and "different" actually mean in that sense. Therefore, we will borrow a very powerful concept from statistical natural language processing:

00:04:16 the idea of distributional similarity. It basically states that words in a similar context likely also have a similar meaning.

00:04:25 So, for example, if I say: "My neighbor has trees in his yard" and "My neighbor has bushes in his yard", we can assume that "bushes" and "trees" are somewhat similar as they appear in a similar context.

00:04:39 This is the basic idea of our vector representation for words which we will call "word embeddings" for later in this unit. According to this line of thought, words are represented through their context.

00:04:52 In reverse, this also means that words not appearing in a similar context should be farther away from each other – at least in some features.

00:05:01 So sticking with this tree example, we can tell that trees and bushes are both in a yard, but we do not observe bushes with trunks.

00:05:10 We can tell that bushes and trees are somewhat close, but not as close as, for example, the vector of trees should be to the vector of oak.

00:05:21 With those ideas, understanding the neural network that creates such embeddings is straightforward again. On this slide, you see the model that generates those embeddings if we feed it with enough text.

00:05:33 It's called Word2Vec, and we will run through its skip-gram variant step by step now. First things first: What is our input and what is our output?

00:05:43 It's exactly those tuples that we just saw previously. They consist of a center word and several context words in a window around the center word.

00:05:53 Those are generated by sliding through our text. And we then train a neural network with only one hidden layer

00:06:00 on the task of predicting the context words, based on the center word. In the network, this looks like this:

00:06:09 We have the one-hot encoding of our center word at the input, and multiply it by a weight matrix that has an entry of size N for each word in the vocabulary.

00:06:21 This is basically just a lookup of the word embedding for the center word. This is already our hidden layer for this neural network.

00:06:29 After another step of matrix multiplication, we feed it through a nonlinearity, calculate our errors, and update our weights as we have seen in the previous units.

00:06:40 So our final vectors are then just the trained weights. We count on the network to have extracted the features that are useful in predicting the context,

00:06:50 which are stored in the neural network weights, as we see here. And just for completeness' sake, the reverse model is called "continuous bag of words" and it works fine as well.

00:07:05 It solves the counter-task of predicting the center word based on the context. So, what does the result look like?

00:07:17 Well, instead of a ten-thousand dimensional vector of zeros, we have a few-hundred dimensional vector of real valued numbers.

00:07:24 And to give you an idea about the concepts our word embeddings extracted, look at this table: The examples shown here are from word vectors trained on the Google news corpus.

00:07:37 In this example, we compare columnwise which vectors are closest to the first entry in each column in terms of cosine similarity. You see that color vectors cluster together, you see that tropical fruits do the same thing,

00:07:51 and number vectors are even sorted in ascending order. For animals, you will find subspecies that at least I didn't even know before.

00:08:05 This intuitive translation of closeness in terms of vector distance to semantic closeness is not the only amazing property of our word vectors.

00:08:13 They are also capable of transmitting certain concepts they have learned. We have reduced the dimensionality of our word embeddings from several hundred to two, and plotted them here.

00:08:24 You see, the distances between certain vectors encode concepts that we use in our every communication. This includes, for example, the relation of a capital city to a country.

00:08:39 The rationale here is that Rome is to Italy what Berlin is to Germany. So in vector terms, the vector displacement between Rome and Italy is about the same

00:08:50 as the vector displacement between Berlin and Germany. So we have extracted the concepts from just raw unstructured text.

00:09:00 And, as you might have expected, this also applies to other concepts, like gender in our next example, or certain relationships in a working environment.

00:09:14 So, as we want to visualize this, let's look at a quick demo. In this small notebook, we will load the Google word vectors that we just talked about into our environment.

00:09:33 Therefore, we just need to instantiate this model object here from the binary... and just explore our word embeddings in a similar manner to how we did it on the slide.

00:09:54 So, for example, the vector for "dog" is just a 300- dimensional vector, over which the last 10 components look like this.

00:10:03 So, we didn't lie – several hundred dimensions and continuous numbers. What about word similarity?

00:10:15 With this convenience wrapper, you just look at the most similar word – in this case "banana" – and retrieve the five most similar vectors in terms of cosine similarity.

00:10:25 So for "banana", it's clearly just the plural, which we cut for this slide, but for the other ones, you see a lot of tropical fruits.

00:10:35 The same thing applies to chocolate, with "dark chocolate", "chocolates", "caramel", and other sweets. For our next step, we can look at our word analogies, which we've briefly touched upon.

00:10:52 Our word vectors also come up to this challenge. We have the vector of "woman" minus the vector of "man" plus the vector of "king" equals the vector of "queen",

00:11:02 or at least it's the most similar vector to the resulting vector. If you do "v(woman)" minus "v(girl)" plus "v(boy)", you will receive "v(man)" as an output.

00:11:11 And the relationship of a "puppy" to a "dog" is the same as from a "cat" to a "kitten". This is also for more abstract examples, as we see here.

00:11:22 So a "pond" is to a "lake" what "small" is to "large". And if we just add up our word vectors and retrieve the most similar word vector,

00:11:33 like we did here for Chinese rivers, we see that the results kind of make sense as well.

00:11:41 And as we also want to visualize this in TensorBoard, in a similar fashion to what Josh did,

00:11:49 let's look at this. For this visualization, we have taken 10,000 word vectors and reduced their dimensionality,

00:12:02 which in this case is 128, to just 3 with a very smart dimensionality reduction technique called "t-SNE", which is already implemented in TensorBoard.

00:12:14 A very convenient thing. So you see that, for example, the number vectors cluster again here,

00:12:19 you see that month vectors cluster here again, you see that pronouns cluster here, and all these semantic patterns that we also have in natural language are kind of mirrored in this visualization as well, and in our vectors.

00:12:39 So, for example, you see different languages clustering together here as well. So as a final remark and closing the loop to our sequence models, this works not only for words.

00:12:53 The Word2Vec idea has been applied to a lot of different sequential datasets, from domains like biology, business interactions, app behaviors, and even emojis.

00:13:05 So it's a very flexible concept that can be applied to different domains and therefore increase the representational power of your input data.

00:13:17 And as we are now fully equipped with meaningful representations of words and an understanding of what sequence models are, we want to look at advanced neural network architectures in the next lecture for sequence models.

00:13:30 In particular, we will look at "recurrent neural networks" and how we can train them. Thanks, and see you in the next unit.

**Week 03 Unit 04**

00:00:08 Hello everybody. Welcome to the fourth unit of Week 3 in our "Enterprise Deep Learning with TensorFlow" course.

00:00:14 In this session, we want to talk about the basics of recurrent neural networks. In the last unit, we talked about vector representations of words

00:00:24 and how we can use them to encode semantics in our input. In today's unit, we go one step further:

00:00:33 after having our sequential data in a useful format, we will introduce a very powerful class of sequence models:

00:00:39 recurrent neural networks, or in short: RNNs. We will briefly recap the sequence types we know so far

00:00:46 and then we go right into the RNN architecture and how to train the network. So yeah, first things first:

00:00:55 To build our model, we first need to define the end-to-end signature of our data. So in the simplest case, where we don't need sequence models at all,

00:01:05 we just map a vector to a vector, "One-to-One". In other tasks, we might want to create a variable length output from a vector.

00:01:17 For example, in image captioning, where we generate text that describes an image. This is called: vector-to-sequence, or "One-to-Many".

00:01:27 In the reverse case, we map a sequence to a vector. So an application for this example is our well- known sentiment analysis

00:01:37 of, for example, movie reviews or customer e-mails in an automatic fashion. And, last but not least, we also want to be able to map a sequence to another sequence

00:01:50 which can both have variable length. A very important use case for such models is, for example, machine translation.

00:01:58 After having now defined our input-output signature, let's look at how a recurrent neural network deals with those.

00:02:09 The basic idea behind RNNs is to make use of sequential information. So in a traditional neural network,

00:02:16 we assume that all inputs have no inherent order that we can leverage. But for many tasks this a very bad idea, as we found out in the previous units.

00:02:26 You remember what we said about predicting the next word, and how important the context is. An RNN is different in that sense:

00:02:35 RNNs are called "recurrent" because they perform the same task for every element of a sequence, with the output being dependent on the previous computations.

00:02:45 Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.

00:02:53 In theory, RNNs can make use of arbitrarily long sequences in the past, but in practice, they are limited to looking back only a few steps – more on this later.

00:03:07 Equivalent to the previous illustration but maybe a bit more intuitive is the unrolled version of our RNN.

00:03:14 You see that unlike a traditional deep neural network, which uses different parameters at each layer,

00:03:20 an RNN shares the same parameters at each time step and therefore it computes the same task at each time step as well.

00:03:28 This has several implications, but also greatly reduces the total number of parameters we need to learn during training.

00:03:38 The main feature of this vanilla RNN is its hidden state h_t. You can think of the hidden state as the memory of the network

00:03:46 that captures information from the past. This hidden state is updated according to the current input x_t

00:03:54 and it represents the only input to the output Let's go through the computations behind this step by step.

00:04:02 One note: we will omit bias terms to make the math a bit easier. In the beginning, we feed our cell an input x_t,

00:04:13 which is multiplied with a weight matrix Wx. The previous hidden state h_t-1 is transformed by its own parameter matrix W_h.

00:04:24 In order to calculate this hidden state… we just add those together...

00:04:30 and apply some kind of nonlinearity, which we call F in this example. And yeah, that's basically it.

00:04:37 We have updated our hidden state h_t that we can propagate further through our network now. We want to have a closer look at the calculation of h_t to understand what is happening here:

00:04:53 The same principle as before applies, so we use the same function and the same weight parameters over and over again.

00:04:59 Keep this in mind also for the later section of the unit. And if you look at the computation of h_t,

00:05:05 you see that the exact structure that we formulated in Unit 2 just appears here. We encode all information from the previous steps in the hidden state

00:05:15 and our output depends on this representation and the current input. If we want to create an output,

00:05:24 for example in a sequence-to-sequence model, we only use this hidden state and its corresponding weight matrix... Wy...

00:05:36 and feed them through a softmax layer. This creates a probability distribution over our classes in classification tasks.

00:05:42 This is just for illustration, so for a regression task you would, of course, use another output function. Long story short – the takeaways for the basic recurrent cell:

00:05:56 We share the same weight parameters for all t, our output depends, at least theoretically, on all previous inputs and the current input.

00:06:05 And finally, calculation of the output means matrix multiplication of the same matrices a lot of times. Such a single layer of recurrent cells is good for understanding its inner workings,

00:06:19 but if we want to cover more structure, we need to go deeper. As in the feedforward network,

00:06:25 we can also stack RNN cells to increase the expressiveness of our model. And then again, we have the sharing of parameters across each layer of the unfolded graph.

00:06:38 So, now, have we solved sequential data for good, now that we can take into account previous information?

00:06:46 Hmm, let's see. For example, consider the following two sentences

00:06:49 where we want to predict the last word based on an input that came very early: "Tom was cooking pasta. Jenny walked in."

00:06:59 "Both ate [...]" Well yeah, the model should output "pasta". In the other case, we have:

00:07:05 "Tom was cooking pasta in the kitchen. Jenny walked in. They talked about what happened during the day." And so on and so forth.

00:07:13 And then: "Tom replied that he was cooking pasta." Can our vanilla RNN remember this context, filter out all the unnecessary information,

00:07:22 and predict "pasta" in the second sentence as well? Well, despite all our modeling efforts, this is not the case in practice,

00:07:31 and will look at the reason for this now. Recall the goal of our RNN implementation with the propagation of context through faraway time steps.

00:07:43 Despite having just a single set of weight matrices that we share through time, training those models is not easy.

00:07:50 And let us try to use the tools we have learned so far to analyze the reasons for that. During forward propagation, we multiply the same matrices over and over again.

00:08:00 And you can think of this step as an amplification or dampening of certain kinds of signals. In a perfect sequence model,

00:08:07 we want to have the signals from many time steps before to have an impact on the prediction at time step t,

00:08:13 or in general, a much later step. The same applies when we backpropagate the error through our RNN at training time.

00:08:22 And yeah, recall the error signals and deltas we used in the first unit of this week? Now we need them again,

00:08:29 as at each time step, the error signal will flow backward through our network. So we compute our loss, take the derivatives,

00:08:36 and accordingly calculate the deltas, and let it flow. At time step t, we will probably have a good update for our weight matrices,

00:08:45 so in the very late step. But the way we constructed this RNN here, will make the signal become too weak or too strong

00:08:54 as we propagate it backward in time. Well, what's the reason for that?

00:09:01 Our main problem here is the way we designed our RNN. Let's assume, for simplicity's sake, a simpler RNN example without any non-linearity

00:09:12 and even without an input. In this case, the hidden state h_t just depends on our weight matrix Wh.

00:09:20 If we now assume that we propagate information through many time steps, we just multiply the same matrix again and again and again.

00:09:29 Well, what does this mean? Without diving too much into the math here,

00:09:36 let's say we can decompose our weight matrix into a matrix Q of eigenvectors, its eigenvalues lambda and Q inverse.

00:09:45 You can think of this decomposition as characterizing the properties of our weight matrix. After multiplying this t times, this will by definition just raise our eigenvalues in lambda to the power of t.

00:09:59 We have left out a lot of math behind all this, especially the part where we derive this for actual backpropagation in training.

00:10:07 But this doesn't really matter because the result is the same for our toy example here and for more complex structures.

00:10:14 All in all, they mean bad news for our vanilla RNN training efforts. If the eigenvalues are above one, our gradient will literally explode.

00:10:25 It will be out of bounds before you can even say "recurrent neural network". This explosion of the gradient will severely impact our training results.

00:10:35 The good thing is that there is an easy way to circumvent such a behavior: If the norm of a gradient is too large, we can just clip it to a certain maximum value.

00:10:43 Thereby, we avoid jumping too far on the error surface and potentially harming our optimization process. In the other case, where our eigenvalue is below 1, we have to be a bit more creative.

00:10:57 This case is called "the vanishing gradient problem", and it basically says that layers that are far away from the error signal

00:11:04 will learn very little to nothing as the magnitude of their update through the gradient is too small. Imagine you have an upper bound of 0.9 to your eigenvalue

00:11:14 and then you want to propagate the error 100 time steps back, you will have 0.9 to the power of 100,

00:11:21 and almost none of your signal will reach the earlier layers. That means that past information will either have minimal or even no impact.

00:11:31 You can think of this vanishing gradient problem as if the flow of the gradient through the network is interrupted at each time step going backward in time.

00:11:40 And again. And again. So this makes learning very difficult, especially for deep recurrent networks.

00:11:52 These problems, such as the vanishing gradient, are in no way exclusive to RNNs – they also appear in feed forward networks if you don't take measures against them.

00:12:02 But due to their inherent depth and the architecture, and the RNN architecture we proposed here,

00:12:08 they are very susceptible to this behavior. So one way to tackle this problem is to use different activation functions

00:12:16 that do not saturate as easily as sigmoid or hyperbolic tangent, for example the rectified linear unit.

00:12:23 Another measure is to put some thought into how to initialize our weight parameters in the first place. But for deep models, we will have to do some changes in the basic recurrent cell architecture.

00:12:40 And this will be the subject of our next unit. And now that we have proposed our vanilla RNN and also discussed its disadvantages,

00:12:51 let's look at a quick demo that shows what our vanilla RNN still can do. In this notebook, we will use the TensorFlow Estimator API,

00:13:04 and have a quick overview here, if you want to read the details. As a first step, as usual, we load our necessary modules,

00:13:15 and set up our environment. By this, we mean the hyperparameters for training, and some log directories.

00:13:25 The basic task we want to solve, as this is just a vanilla RNN example, is to make our model learn the godfather of periodic functions, design function.

00:13:37 So the nice thing here is that we don't need to worry about data as we can just create it by ourselves.

00:13:43 And this is what this cell just does. We create our data.

00:13:49 And give it over to an input routine... here "def train_input"...

00:13:55 which just cuts our data down... which just delivers our cut-down data to the dataset API of TensorFlow.

00:14:05 The same thing applies for the validation input, with some minor additions.

00:14:13 You can happily ignore this, it's just some helper functions. And then here comes the fun part:

00:14:18 The recurrent neural network architecture is defined by this function here, "rnn_model". So we give it our features, our labels,

00:14:27 if we want to train or predict the corresponding parameters and our configuration. We first define our BasicRNNCell,

00:14:37 and TensorFlow provides us with it so we can skip all the math here, and...

00:14:46 then construct the network accordingly. We have our output and layers defined as a static RNN in this installment.

00:14:59 And then add some additional, fully connected layers just right after the RNN output, which is then just covered by the slim.

00:15:12 And yeah... This is basically our Estimator – we give it the possibility to predict and to train.

00:15:22 I have some hooks here again. And then we can start training and evaluation.

00:15:27 So we instantiate our Estimator by just handing it over our rnn_model function with decoding parameters and configurations.

00:15:39 Set it into training and then evaluate. There's a long training going on here, as you

00:15:48 and we output a lot of details. But what we really want to know is what our output looks like.

00:15:53 And yeah, if you see this... This is really sigmoid sine-like for me.

00:16:00 And it shows that with this simple architecture, the neural network has just learned a completely different function that it was not trained on before.

00:16:10 Just with the input of five sequential data points from the sine function. Yeah.

00:16:19 And that's it for this little toy example. In the next unit, we will introduce you to more advanced recurrent architectures

00:16:35 that are the building blocks for a lot of state-of- the-art solutions in natural language processing and for sequential data in general.

00:16:43 Thanks a lot for watching, and see you in the next unit.

**Week 03 Unit 05**

00:00:08 Hello everybody. Welcome to the last unit in Week 3 on sequence models.

00:00:15 In the previous unit, we learned the basics about recurrent neural networks. We talked about how they are in theory capable of taking into account a large context from past information.

00:00:26 But we also talked about practical issues when we try to implement a basic RNN, especially when you remember the vanishing gradient problem.

00:00:34 In today's unit, we will cover an advanced RNN architecture to cope with those problems. They are called LSTMs (long-short term memories) and GRUs (gated recurrent units).

00:00:46 Those cells are pretty much used in most deep recurrent models today and are also used in most state-of-the-art sequence models.

00:00:54 After today's unit you will have a powerful tool for a lot of deep learning applications. A quick recap before we start – let's look at activation functions

00:01:05 because we will need them for the step-by-step walkthrough. When we talk about sigmoid activation,

00:01:12 we talk about a function that squashes every input you give it towards a range of 0 to 1. And yeah, don't tell this to your statistician next door,

00:01:21 but you can think of it as a probability, but not in a rigorous sense. The hyperbolic tangent, or tanh, is very similar,

00:01:31 but squashes the input into the range between 1 and −1, and also has some beneficial statistical properties.

00:01:39 We will now dive into this. So yeah, let's look at our CRM.

00:01:44 And the good news is our basic structure of our RNN network remains the same. So you still have a recurrent cell and you still have the unfolding through time.

00:01:55 The difference now is that our RNN has a different kind of recurrent cell, so the building blocks differ here.

00:02:04 In particular, we allow our cells to keep and to forget past information – based on a mechanism the model will learn on its own.

00:02:14 We have three main features of our LSTM cell that define its capability in memorizing our long-term dependencies:

00:02:22 First, we introduce a memory state that is intended to capture long-term information. We still have our hidden state, which we called h_t in the previous unit.

00:02:32 And yeah, you can imagine it as a kind of "working memory" for the cell. And, last but not least, we also have "gates" that regulate the flow of information in the cell and between cells.

00:02:46 Without further ado, this is it – this is the LSTM architecture. Yeah, I should probably say it is one architecture,

00:02:53 because there are many different implementations, but if you go through this one unharmed you will get the others as well.

00:02:59 So in general, the LSTM cell is similar to our vanilla RNN, except these two structural differences you see here:

00:03:07 First, we have an additional memory state, c_t, for the long-term memory. Second, we have explicit mechanisms that help the cell to decide what to forget and remember,

00:03:18 our "gates". This means that at every time step, we forget information from the past sequences,

00:03:24 we add new information based on the current input, and we update the existing knowledge accordingly.

00:03:31 So yeah, let's look at the memory state, at the gates, in more detail to unravel this cell. The memory state c_t is the central point in this architecture.

00:03:42 It is our best hope for actually remembering long-ago parts of our sequence. It carries over information from cell to cell

00:03:50 and then decides at certain points which parts of the memory state we want to forget and which parts we want to update.

00:03:57 These decisions are made by gates, and we will go through all of them now. So yeah, how much does the past matter?

00:04:06 This is decided by the "forget gate". It has its own set of weight parameters

00:04:12 and takes into account the past hidden state, h_t-1, as well as the current input, x_t,

00:04:19 to decide which part of the previous cell state should be kept. You might ask: "Why is it a good thing to be able to forget information

00:04:27 when we are always talking about how important long-term dependencies are?" Let's think of an example from our favorite use case:

00:04:34 predicting the next word in a sentence. Think of the cell state as having the information of the gender of a certain subject in our text.

00:04:44 This is important, for example, to predict the corresponding pronoun. Naturally, as we move through our text,

00:04:50 we want to be able to forget the gender of the past subjects. Recall that this is a sigmoid?

00:04:56 So when we apply it element-wise, for each element of the memory we can decide which fraction to forget. Now, we are not quite done yet – our model can still do more, and has to.

00:05:12 Certainly we also want to take into account the current input at time step t. And how much it affects our memory state is decided by the "input gate".

00:05:21 The input gate decides how much the current input will matter based on the previous hidden state and the current input,

00:05:29 so the same structure as before. We create our new memory state, not just in the same fashion,

00:05:35 but with a different nonlinearity. And finally then, we update the memory of the cell with all the information we just computated.

00:05:47 As you see in this formula, it is just a weighted sum of the past memory state and the new memory state we calculated.

00:05:54 So we have c_t (the memory state) equals the forget gate times the past memory state, plus the output of the input gate times the updated (or short-term) current cell update.

00:06:08 This allows us to either forget the past, forget the signal of the current input, or combine them, which is a really powerful feature.

00:06:17 And you might ask again: "Isn't this forgetting and updating kind of the same thing? Why do we need those two mechanisms?"

00:06:24 It's a good thought, keep it. We will come back to this later. What is actually pretty amazing about those cells

00:06:30 is that the decision on which input to keep and which to discard is made by the network alone. So it is solely learned from our data.

00:06:42 Last but not least: the output gate. What do we need it for?

00:06:46 It allows you to keep different information for short-term use than is stored in the memory state. Well, we want to allow our network to decide whether the current memory cell

00:06:59 matters for a certain prediction or not. The idea here is that you have to decide which part of the current state

00:07:07 should be exposed as the output at time step t to make a prediction at time step t. This allows the network to carry over information in c_t to the next time step,

00:07:17 but still output something meaningful at the current prediction. Think of it as deciding which elements of long- term memory to pull into working memory.

00:07:30 Did we now solve the vanishing gradient problem with all our fancy gates? Kind of.

00:07:36 With the memory state, we have a channel for information to flow across time relatively unchanged. In contrast to our vanilla RNN cell from last unit,

00:07:45 we are able to copy or forget previous information – and we don't have to add and average over and over again,

00:07:51 so we are given that our model decides that this is necessary. But everything has its cost:

00:07:57 In our case, we have to train a lot of parameters. And this in turn means that we need more data for our model to be able to generalize.

00:08:07 Nevertheless, LSTMs are a very powerful architecture and are all over the place in academia and in industry.

00:08:15 So, you remember when we talked about how to simplify the update of the memory cell? The gated recurrent unit is the answer to that question.

00:08:24 Here we only have an update gate and a reset gate. And we also only have one hidden state and no memory state.

00:08:31 The reset gate allows us to ignore the past states completely when calculating a new hidden state,

00:08:37 while our update gate tells us to what extent we should update our hidden state at all. You can think of the update gate as the equivalent of both the input and forget gate in our LSTM cell.

00:08:51 So yeah, you can also assume that the GRU cell is kind of a simplified version of the LSTM cell. The whole idea of creating additional connections to improve the flow of information

00:09:06 is not only used in these LSTM cells or GRU cells, but also in our next example, called "residual networks".

00:09:16 Here we add skip connections to be able to train really deep networks and make our model more expressive.

00:09:23 Like in the long-term memory of the LSTM, the skip connections allow our gradient to flow uninterrupted during training.

00:09:32 We will look at this next week in more detail. With all this theoretical stuff, let's switch briefly to a small demo.

00:09:45 So in this demo, we want to predict the sentiment of a given text. It is set up in the following way:

00:09:52 with two input data files – one with sentences from movie reviews that are positive and one with movie reviews that are negative, or have a negative connotation.

00:10:05 To build our LSTM architecture, we do the following: So first, we load the necessary modules here.

00:10:13 Then we set up our environment as we've also seen in previous demos. Once you have access to this, you are free to change all the parameters you see here

00:10:23 and try it out on your own, because you will see that this will have a severe impact on the performance of the model.

00:10:29 In the next step, we create our data set. And for this, as in most text tasks, we first have to clean our corpus.

00:10:38 So this function basically returns a cleaned version of our sentences without any trailing white spaces or non- alphanumeric characters, and so on.

00:10:53 We then apply this function just to our input, and we truncate after certain thresholds so we have the same size of the input

00:11:02 for all our modeling endeavors. Then we create a dictionary for each word, so we tokenize it,

00:11:12 and convert our sentences to a sequence of just those integers which represent each token. So...

00:11:21 then we do a quick sanity check to see whether this all worked. And then we have a function to pad our zeros to get smaller-sized sentences

00:11:31 up to our previously defined threshold. Then we convert this all into a NumPy array,

00:11:40 split according to training and test set, and create an input function as we will again use our TensorFlow Estimator API.

00:11:49 And this is all relatively straightforward here with our NumPy input function, which is kindly provided by TensorFlow Estimator.

00:11:57 The model input parameters are given by this dictionary again. And this is a simpler version of our hook.

00:12:06 Now comes the fun part – we define our LSTM modeling function. And we have an input layer, as you will probably have assumed.

00:12:14 Then we have a word-embedding layer. Now there's a difference to what we did in Unit 3:

00:12:20 We talked about training those word embeddings on a different corpus and then importing them into our model.

00:12:27 We can also just add an embedding layer and trail our whole network to end to end, but you're also free to initialize our word embeddings here with the vectors we discussed in Unit 3.

00:12:39 Then it's relatively straightforward: We define the basic LSTM cells here.

00:12:51 Add a DropoutWrapper for regularization, which is not that important for this stage,

00:12:56 and just have our MultiRNNCell and a dynamic_rnn for all our modeling efforts. So yeah, this is basically the whole modeling architecture in just some lines of code.

00:13:10 We then add our logits and allow our model to predict and to train with this "if" catch, and have to define our loss function, which we define as a sigmoid cross_entropy loss with our logits.

00:13:27 So yeah... After defining our optimizer and our training operation, we have our estimator defined.

00:13:33 And we can now go on and add and instantiate our estimator, which is done in this cell.

00:13:41 And after we call our training routine, TensorFlow does its magic and outputs everything here. So for all the things that happen during training...

00:13:54 We might want to have an evaluation dictionary as well, as we do by calling our evaluate function from the estimator. And now let's have a check.

00:14:04 Did all this effort work at all? And can we predict the sentiment for our movie reviews?

00:14:11 This is done here. So what is the sentiment of some example sentences? So you might see that in general it retrieves some pretty interesting sentences.

00:14:23 So: "you come away wishing though that the movie spent a lot less time trying to", which is maybe not that easy to catch for another example.

00:14:33 But we also see that some things didn't work, like: "the jokes are flat and the action looks fake".

00:14:41 So you are invited to change hyperparameters, change the training epochs, and try to make this model work even better than it does now.

00:14:52 And I hope you enjoy it. So yeah, this was it from Week 3.

00:15:00 I hope you enjoyed it. You now have the basic tools to rebuild some of the most advanced architectures for sequential data.

00:15:06 Next week, we will continue with convolutional neural networks and dive real deep into image data. Thanks for watching.

00:15:14 And I wish you good luck for the weekly assignment of Week 3. See you.

www.sap.com