#h1 Chapter 26 - Deep Belief Neural Networks

#h2 Layers, Donkey...! Onions have layers! Ogres have layers!

#h3 26.1 - Introduction

#pg According to Google, the usage of the bigram "deep belief" started taking off in earnest in 2005, suggesting the first deep belief neural networks (DBNNs) were invented in or slightly before that point in time. This is somewhat misleading, as the term DBNN is more of a rebranding than it is a novel invention. At the core, a DBNN is neural network with one or more hidden layers. Hidden layers take no outside inputs and produce no final outputs, but instead sit in between the layers that do. Defined like this, these types of networks do not qualitatively differ from the multilayer neural networks we discussed in the previous chapter. Instead, the dividing line is usually drawn quantitatively, where DBNN have many hidden layers, and require massive data sets and specialized hardware (GPUs) to train efficiently. That is not to say that a lot of major innovations have been introduced to how we design and build neural networks during the DBNN era, allowing them to be this much bigger, deeper and take on ever more complex data sets. With the increase in widespread availability of faster computers, GPUs, cloud computing services like AWS, building this new generation of neural networks became available to a much large audience. Perhaps equally important, open-source frameworks to build and train DBNNs were released, removing the need for individual researchers and engineers to implement the complex underlying algorithms that do the many computations needed to train and run DBNNs, such as backpropagation, weight optimization, hyperparameter tuning, etc. These frameworks include Caffe, Torch, Keras and Google's TensorFlow. The latter is perhaps the most widely used framework at the time of writing. TensorFlow is built to be a general-purpose framework, capable of much more than just building DBNNs with. It is specifically designed to scale easily, both in terms of the complexity of the networks we design, and the size of the data needed to train them. We shall use it from this point on, as it makes the demo code we wish to present and for you to possible adopt and experiment with much more concise and easier to understand. There are many resources online you can use to master TensorFlow, including Google's own excellent website, which comes with many tutorials, demos, datasets, and dashboard visualization tools. We will acknowledge that some of the notebooks we present can be traced back to their original TensorFlow versions in some detail, as that is how we got started using TensorFlow ourselves. There is no point in reinventing the wheel, especially when the actual inventors of said wheel have done a great job documenting them. When possible, we tried to use datasets more closely aligned with our work here at MSK, but data availability and privacy concerns meant this was not always possible.

#im ../../../assets/figures/026/026-01.png 50 256 Figure 28.01 - There are many open-source deep learning toolboxes to choose from

#h3 26.2 - (Leaky) ReLu activation functions

#pg Before we start building our very first TensorFlow model, we want to highlight some innovations that were all part of the shift from traditional Multilayer Neural Networks into the new era of Deep Belief Neural Networks. DBNNs were made possible by the ever-increasing speed and power of our laptops and desktops, the adoption of Graphics Processing Units or GPUs to carry out the massively parallel computations efficiently, and the internet and its users creating the necessary volume of content needed to ever train the highly complex models people imagined. However, in addition to these external opportunities creating themselves, some key technical innovations were made to how these DBNN architectures were designed, implemented, and trained. The first of these innovations was the introduction of new activation function, the Rectified Linear Unit (or ReLU) and the leaky Rectified Linear Unit (or leakyReLU). Fig. 26.2 shows the input-output relationship for these two activation functions. The rectification applied to the ReLU effectively silences the output of a node if that output is negative. When the output is positive, the relationship between input and output is 1:1 or y = x. For the leaky ReLU unit, the rectification for negative outputs isn't complete, allowing at least some of the negative output to be transmitted. One benefit of these units is that they prevent gradients from saturating in deep networks, and thus mitigates the risk of vanishing gradients we discussed previously (Chapter 24). Second, since they are completely (ReLU) or almost completely (leaky ReLU) silent for a large range of possible inputs, it forces the nodes to adopt sparse representations. A sparse representation is preferred, with each node encoding something that is unique about the relationship between input and outputs. When we have many nodes and/or not enough volume or variation in the training data, the nodes will all gravitate towards the most dominant features in the inputs, when we use sigmoidal or tanh activation functions. As a result, the nodes do not specialize sufficiently during training. The ReLU unit with its rectified response overcomes this issue partially. Addition methods can be used to further mitigate this problem, see the section of Dropout below. The leaky ReLu and other variants of the ReLU were later introduced to make the complete rectification less definitive, allowing activation a <= 0 to still yield a range of outputs, rather than just 0 in the original ReLU. This gives up on the sparsity that the ReLU offered, but also fixed the problem of the 'dying' ReLU. If a ReLU in it’s a <= 0 regime, its output will be zero. If during training no input or set of inputs can evoke a change in weights big enough to move it out of that regime, it will forever produce an output of 0, no matter what the input. It is considered to have died. Allowing at least for some activity other than 0, it can still discriminate between the different classes, or at least a subset of them.

#h3 26.3 - Optimization and Learning

#pg Parallel to the introduction to the ReLU activation function, researchers were improving on new optimization schemes that allowed for some flexibility to adapt the training regime during training, something that conventional gradient descent does not implement. These newer optimization methods allow us to tweak the original idea of stochastic gradient descent and make it more stable, converge faster and not get stuck in local minima as often has yielded a few upgrades to the original idea.

#h3 26.4 - Optimization and Learning - AdaGrad and RMSProp

#pg In certain high dimensional problem like those found in computer vision, often there is simply not enough gradient in all dimensions for a node to adjust its weight. Most nodes will see only small changes from input to input as these changes a distributed over the high dimensional space. In classic Stochastic gradient descent, we have a single learning parameter alpha that determines how fast/far we travel along the gradients to find a minimum. It is fixed during training and is the same for each weight we need to estimate. Adaptative Gradient optimization uses a per-parameter learning rate. This improves the performance on problems with sparse gradients. Root Mean Square Propagation (RMSProp) also has uses a per-parameter learning rate that are a function of how quickly the gradient was changing previously. Higher magnitude gradients in the past speed it up (bigger learning rate) and lower magnitudes slow it down.

#h3 26.5 - Optimization and Learning - Adam

#pg Adam optimization combines the benefits of the two above. A deep dive into Adam is outside the scope of this class, but a quick introduction might make it less of a mystery. But AdaGrad and RMSProp keep some running average of the mean gradient. Adam in addition uses not only the first moment (mean) but also the second moment (variance). Additional parameters control what of the gradient should be considered a real change versus noise by the optimizer. The intuition in the end is that Adam moves faster if gradient becomes steeper, but with caution, not trying to get fooled that the high gradient is noise, rather than signal.

#h3 26.6 - Optimization and Learning - Other optimizers

#pg Adam is still the optimizer of choice for most neural nets being developed even today, but other optimizer have been proposed, such as Momentum, AdaDelta and ‘Follow the regularized leader’ or FtRL. In the end, all are still descendants from the same original idea of gradient descent.

#h3 26.7 - Dropout

#pg Another potential problem of DBNNs with many nodes is that during training these nodes become too highly specialized in or too sensitive to a particular irrelevant feature in the input data. If we aim for sparsity, we might see this arise as a side effect. Although we should do our best to make sure are data properly randomized and balanced, there is always going to be some regularity in our input data that in the wild is irrelevant to the thing we are predicting. Imagine I am trying to predict from a large set of images, whether the picture contains a cat or a dog. It is quite possible that lots of the cat picture were taken indoors, whereas the pictures of the dogs were taken outside. So, the dog pictures feature a blue sky and green grass more often than our cat pictures. This has obviously nothing to do with what constitutes an image of a cat or dog but is nevertheless correlated with it in our data set. And a neural network doesn’t know what any of those things are, only that they are correlated. And this correlation drives the neural network to label pictures with blue skies as dog pictures. Regardless of their being a dog in them or not. A surprising and surprisingly effective way to fight this problem is by means of dropout. During each learning iteration, a proportion of the nodes is simply ignored as if it was never there. Each iteration, we pick a new set of nodes, based on some set probability. This forces the nodes to share the responsibility of encoding/detecting dogs and cats, mitigating the overfitting issue. Dropout and the ReLU optimize for opposing goals: the ReLU unit aims for sparsity, with nodes being specialized to features that matter. Dropout, at the same time, aims to prevent a too great amount of sparsity, especially when nodes become to specialized in a feature, and to a feature that does not generalize to all possible inputs.

#h3 26.8 - Introduction into TensorFlow

#pg Now we have discussed a few additional innovations that to us gave birth to the era of DBNNs, we can take some time walking through implementing these DBNNs in what is perhaps the most popular deep belief platform: Google's TensorFlow. Interestingly the core idea behind TensorFlow isn't necessary just about Deep Belief Neural Networks. For one, it is more flexible than that, allowing for a range of numerical methods to be implemented, like curve fitting complex timeseries or ray tracing in 3D rendering. Rather, TensorFlow, like Google's other big innovation MapReduce, is a data model and/or programming paradigm. By design, it uses an abstract data representation, that can be run on many different systems, such as a serial CPU, a parallel GPU, or a distributed set of interconnected computing devices in the cloud. In TensorFlow, this data abstraction takes the form of tensors. We have seen many examples of N-dimensional data sets, which we organized in 2D arrays or matrices, usually denoted with the letter X. The rows are the individual data points, each column a dimension of the data. But this is a bit misleading. The array itself is two-dimensional, yet we typically refer to its dimensionality as the number of columns. It is therefore better to say that data matrix X (which rows and columns) is a flattened representation of the actual N-dimensional data. When instead we use a different axis for each dimension, we call the resulting data structure a tensor. More formally: a vector is a 1D representation of a set of points, an array is a 2D representation of a set of vectors, and a tensor is a N-D representation of a set of arrays. The concept of a tensor might be new to you, but we have already seen many examples of tensors throughout this book, with many more to come. Specifically, a color image is a tensor. It has 3 axes: one for width, one for height, and one for the color channels. Each element thus represents the pixel value at a particular coordinate and color channel: image[100,100,0] is the intensity of the red color channel at the coordinate x=100, y=100 in the image. It is possible to flatten this representation to a more traditional array, as you can see in figure 26.2. However, this flattening is not always a desired transform. We can easily extract a small patch within a larger image if it is in its original tensor format. For example, we can take out the center 64 pixels of a 256x256x3 image with ease. However, if we had flattened the representation of the image, we would have to do a lot more bookkeeping and careful calculations of the indices that respond to a spatial continuous subregion of an image. In addition, if we were to parallelize processing of these images between multiple computing devices, like a stack of GPUs, the tensor representation allows us to quickly divide up the image, sending each part to different GPU, with the understanding that the algorithm we are running can operate on a subset of the input images. In short, representing data as Tensors allows TensorFlow to deal with high dimensional data very effectively, especially when such data is operated on in a distributed way by multiple parallel computing processes.

#im ../../../assets/figures/026/026-03.png 50 256 Figure 28.02 - Ever more powerful GPUs allowed not only to play games with better graphics, higher resolution and frame rates, but also allowed us to do the algebriac operations needed in training and running a DBNN much faster a traditional serial CPU woul ever allow for. The top GPU was the very first I owned personally. It had 2MB of memory. Below is an already outdated but still modern variant, the NVidia GTX 1080.

#h3 26.9 - Building our first TensorFlow model

#pg In earlier versions of TensorFlow, building DBNNs repurposed code from an older framework called Keras. Keras had/has a very intuitive way to design neural network architectures, and over many releases, Keras and TensorFlow have been pretty much merged into a single framework. Although TensorFlow allows a user to specify models in a variety of different ways (many roads lead to Rome when it comes to TensorFlow), we will use the Keras approach to get you familiar with building models. Let us start by computing some extremely straightforward: a linear regression.

#im ../../../assets/figures/026/026-02.png 50 256 Figure 28.03 - Ultimately, an image is an array or better yet, a tensor. And it is operated on it as such.

#h3 26.10 - TensorFlow linear regression - Synthetic Data

#pg For our very first TensorFlow model, we generate some synthetic data. Specifically, we create individual data points by picking a random age between 10 and 90 years old (X). We made the risk (y) of being diagnosed positive for cancer a linear function of age, where the probability of developing cancer increases every year lived with 5/1000 (0.005), give or take as some small and normally distributed noise was also added to the outcome y. With our X and y data now created, we split them into training and testing data. TensorFlow has some functions that allow you to do that at the level of tensors, but to keep things simple in our first TensorFlow example, we are working with NumPy arrays directly loaded into memory, rather than TensorFlow’s tensor representation in which data is loaded into memory only when needed (i.e., lazy execution). Therefore, with this small synthetic dataset, we use the scikit learn function train\_test\_split to divide our data.

#br

#cc no\_points = 1000

#cc age = np.random.randint(10,90,(no\_points,1))

#cc noise = np.random.randn(no\_points,1) \* 0.05

#cc cancer\_risk = (age \* 0.005) + noise

#br

#ca Code 26.1 - Synthetic data was created by creating a population of patients, uniformly distributed in their fictious age between 10 and 90 years old. Cancer risk was set to 0.005 x age, and a small amount of normally distributed noise was added to create some variability between age and cancer\_risk.

#h3 26.11 - TensorFlow linear regression - NumPy to TensorFlow

#pg With the data divided into training and testing, we can create TensorFlow data objects by combining the input and label data. These objects can extract data on the fly, without having to preload all of it into memory. It allows for caching and prefetching of data to reduce the I/O during training. This allows us to train networks efficiently on very large amounts data, like the 1 million+ images in the 1000 visual object categories in ImageNet.

#br

#cc X\_train, X\_test, y\_train, y\_test = train\_test\_split(

#cc age,

#cc cancer\_risk,

#cc test\_size=0.33,

#cc random\_state=42)

#cc train\_dataset = tf.data.Dataset.from\_tensor\_slices((

#cc train\_examples,

#cc train\_labels))

#cc test\_dataset = tf.data.Dataset.from\_tensor\_slices((

#cc test\_examples,

#cc test\_labels))

#br

#ca Code 26.2 - Dividing our data into training and test data and create TensorFlow tensor objects from the NumPy array we created.

#h3 26.12 - TensorFlow linear regression - Model Specification

#pg We will use the Keras style definition of our neural network architecture, where layers are added in sequence. In the traditional sense, a. layer is a set node performing some computation on the inputs it receives (combined with the weights on these inputs). The resulting output of the nodes in the layers which serves as the inputs to a subsequent layer or serves as the final output of the network back to the user. In TensorFlow, layers do not necessarily always follow this strict neural network definition. Instead, layers are defined more broadly, capable of performing any necessary mathematical operation on the input to a layer. For example, z-scoring can be implemented by two layers in sequence.

#br

#cc risk\_model = tf.keras.Sequential([

#cc layers.Dense(input\_shape=[1], units=1)

#cc ])

#br

#ca Code 26.3 - Specifying our linear regression model as a single FC (dense) layer.

#br

#pg The first layer subtracts the mean vector across all inputs from the vector of inputs, a second layer will subsequently divide the vector of standard deviation across all inputs from each vector of inputs. Jointly, they implement z-scoring normalization. The layers implement operations like Batch Normalization and Max Pooling, two useful operations we will encounter in greater detail in the next chapter on convolutional neural networks. When creating layers, we can specify the activation function applied to it, or simply add an activation function as a new layer. Other layers implement convolution, others (e.g., LTSM layers) allow for memory-like processes, retaining earlier inputs and combining them with newer inputs. The latter allows us to model the flux of our system through time and is preferred method for forecasting time series (like the stock market). In addition, several layers are specialized in manipulating inputs like images, resizing, reshaping, and rescaling them as seen fit. As you can probably imagine, our first TensorFlow model is quite straightforward, implemented just one fully connected (FC) dense layer that will need to learn how to transform the input (age) into a predicted output value representing cancer risk. If we look at the output from the model.summary() command, we see that the model has two trainable parameters. The first is the single weight that transforms the input x (age) into y (risk), the second is the bias which allows for the risk to be offset relative the origin. In other words, at age 0, the risk for cancer doesn't have to be exactly zero, the bias value can shift this up or down, whereas the weight parameter provides the slope of the resulting model.

#br

#cc optimizer = tf.keras.optimizers.Adam(

#cc learning\_rate=0.002,

#cc beta\_1=0.9, beta\_2=0.999,

#cc epsilon=1e-07, amsgrad=False, name='Adam');

#br

#ca Code 26.4 - Defining our optimizer to be of type 'Adam' with an initial learning rate of 0.002.

#h3 26.13 - TensorFlow linear regression - Optimizer

#pg After specifying the model, we need to pick an optimizer to use during training. The most widely used optimization method at the time of writing is probably still Adam, so we use it in our first TensorFlow model as well

#br

#cc risk\_model.compile(

#cc optimizer=optimizer,

#cc loss='mean\_absolute\_error')

#cc risk\_model.summary()

#cc

#cc Layer (type) Output Shape Param #

#cc =====================================================

#cc dense\_4 (Dense) (None, 1) 2

#cc Total params: 2

#cc Trainable params: 2

#cc Non-trainable params: 0

#cc =====================================================

#br

#ca Code 26.5 - Compile the model, providing it both a predefined optimizer and a suitable loss function

#h3 26.14 - TensorFlow linear regression - Compile and summarize

#pg Before we can train the model, we will need to compile it. When we do, we specify the optimizer as well as the appropriate loss function. Remember that in regression, there isn't a measure of accuracy, so during training of a regression model, we only keep track of the loss across epoch. In the next example where we will build a simple classification model, we will use both loss and accuracy, during both training and validation phases.

#br

#cc history = risk\_model.fit(train\_ds, verbose=1,

#cc batch\_size=128,epochs=20,

#cc validation\_data=test\_ds)

#br

#ca Code 26.6 - Train the compiles model, using 20 epochs and a batch size of 128

#h3 26.15 - TensorFlow linear regression - Fit and predict

#pg After we have compiled the model, we can start training it using the code snippet in Code 26.6. The fit function allows you to set several parameters, including in the above command are the most relevant: the batch size (the size of the batch compared to the total size of the training data defines the number of batches run each epoch) , the number of epochs, the verbosity level and what data set should be used for validation in between training epochs. The return data into history is the epoch-by-epoch amount of loss, validation loss, and other metrics is specified (such as accuracy for a classification algorithm). Finally, we can use the trained model to predict a new set of data, and save the model, weights, and all to a TensorFlow data directory.

#br

#cc history = risk\_model.fit(train\_ds, verbose=1,

#cc batch\_size=128,epochs=20,

#cc validation\_data=test\_ds)

#cc risk\_model.save('my-tf-linear-regression-risk-model');

#cc y\_pred = risk\_model.predict(test\_ds)

#br

#ca Code 26.7 - Train the compiles model, using 20 epochs and a batch size of 128. After that, save the model, weights included, and create prediction for the test data using 'predict'.

#h3 26.16 - Multiclass Classification Loss

#pg This first example uses linear regression with a single scalar output (cancer risk), for which we can use a straightforward loss function: the mean absolute error. But when we move away from regression and want to classify inputs instead, we have to revisit what loss function we should use, especially in multiclass classification problems. Remember that in logistic regression, the model yield a single output, bounded between 0 and 1. From this, we computed the error between the actual and predicted output, from which in turn we computed the loss. When we faced a multiclass problem (more than 2 classes) we took a one-versus-all approach, building one classifier for each class that was trained to discriminate between its class and all the other classes. And in a winner take all approach, we picked the model that was most certain about being correct, yielding the highest probability for its class to be the actual class given the input. In DBNNs performing classification we don’t have to build one network for each input class. Instead, we have the output layer have as many nodes as there are classes. And since we will be using gradient descent, we will need to compute some relevant measure of loss. The most common loss function used in DBNN multi-class classifiers is cross-entropy. It is not a bad idea to see a few examples of this to get an intuitive sense of it. Given that have one node per digit, the output of our network is a 10-element vector. To compare it to our ground truth of what digit was in the input image, we also encode the ground truth y as a 10-element vector, with a one at the index corresponding to the digit, zeros elsewhere. What we want to compute is some metric that tells us how similar these two vectors h and y are. By encoding the ground truth, the way, we did, we have created a vector of probabilities. It is our ground truth, so there is probability of 1 at the digit in input image, and a probability of 0 anywhere else. Perhaps a good start is to have the predicted output of our network also represent probabilities, or at the very least values between 0 and 1. In that way we can use an information-based metric, like entropy, to measure their similarity. We already know one way of guaranteeing that, which the sigmoid activation rule. A different way is not to pass the activation in our final layer through an activation rule, but instead apply the so-called soft-max function. It is a fancy sounding name, but it means we normalize the activation of node by dividing it by the sum of activation across the 10 output nodes. This bounds the outputs to [0,1]. The distinction between the two measures is primarily that in SoftMax, the output of each node is not independent from the output in the other 9 nodes, whereas if we use a sigmoid activation rule, they are. If we let our final layer have a sigmoid activation function, we refer to the computed loss as binary cross-entropy loss, whereas we apply the SoftMax instead, it is referred to as categorical cross-entropy loss.

#h3 26.17 - TensorFlow classification - Iris dataset

#h3 26.18 - Conclusion

#h3 26.19 - Demos

#h3 26.20 - References