#h1 Chapter 25 - Multilayer Perceptrons

#h2 XOR, Backpropagation, and nonlinearities

#h3 25.1 - Introduction

#pg Wait? Another chapter on neural networks to work through before we discuss what you really like to read about? Don't worry. We will get to deep belief neural networks (DBNNs) soon enough (i.e., in the next chapter). To be honest, the phrase 'deep belief neural network' to us is more of a rebranding than a logical progression from the multilayer neural networks that preceded them. Google the definition of 'deep belief neural network,' and you will likely find that what best defines a DBNN is that it has one or (many) more hidden layers. Hidden layers are layers that neither received direct inputs and neither are their outputs the final output. Instead, they receive intermediate levels of activation from previous layers, which can be both other hidden layers themselves and the input layer. Similarly, hidden layers might send their activity to the final output layer or pass it along to the next hidden layer. And if you look up the definition of a multilayer neural network, say a multilayer perceptron, you will find a very similar definition. Both terms refer to neural networks with one or more intermediate (or hidden) layers between the input and output layers. Multilayer neural networks were the first step toward such networks. They were made possible by the discovery (or invention, it is hard to find a proper verb for how new algorithms come into being) of backpropagation. In addition, new activation functions introduced nonlinearities that allowed for more complicated mapping of inputs onto outputs. DBNNs are a continuation of this research, finding even better activation functions, creating more adaptive gradient descent variants, finding new ways to increase neural network stability, optimizing learning, and preventing the increasingly larger neural nets from overfitting the data with even bigger data sets.

#im ../assets/figures/025/025-01.png 50 192 Figure 25.1 - In their 1969 book 'Perceptrons', Marvin Minsky (left) and Seymour Papert provided evidence that more complicated and interesting computational problems required the introduction of both nonlinearities and hidden layers if neural networks would ever be used to solve them.

#h3 25.2 - Minsky and Papert's monkey wrench

#pg In the previous chapter, we built our first ANNs and turned them into simple logic gates. In addition, we used them to classify a relatively simple simulated but noisy dataset. And, in discussing the history of ANNs, we paused at Paul Rosenblatt's Perceptron and his learning rule to train it automatically. For good reason. A few years after the Perceptron was introduced, Marvin Minsky and Seymour Papert (1969) published the book 'Perceptrons - an introduction to computational geometry', see Figure 25.1. In it, the authors unequivocally demonstrated that certain computational problems could never be computed by the input-output neural networks of the day, with the XOR logic gate being a classic example, see

#im ../assets/figures/025/025-02.png 50 256 Figure 25.2 - The infamous XOR problem. Classifying apples as large OR red or classifying apples as both large AND red can be done with a single input-output neural network. In both bases, an infinite number of linear functions exist that divide the two classes according to their desired label (large and red, large, or red). However, when we try to distinguish apples as large and green or small and red from those that are large and red or small and green, no such linear function exists to divide the examples into their two classes neatly.

#br

#pg While this is certainly true, it was also understood that ANNs could easily solve many types of problems by introducing hidden (i.e., intermediate ) layers between the input/output layers on Rosenblatt's Perceptron. However, no algorithms could apply Rosenblatt's learning rule to update the weights in such multilayer networks ( which at the time were typically referred to as multilayer perceptrons). This led to a decline in the interest in neural networks, and the focus of AI shifted to logic and rule-based knowledge systems instead.

#h3 25.3 - Enter Backpropagation

#pg Fortunately, a solution was found a few years later. And that solution is still at the heart of our ANNs. Although the timeline and order in which backpropagation was introduced to the neural network community are somewhat convoluted, many agree that Werbos, extending his original 1975 thesis, introduced the neural network-specific implementation in 1982 (Werbos, 1975, 1982). But it would be unfair not to give credit to Linnainmaa, a Finish student describing a precursor of the algorithm in his 1970 thesis (Linnainmaa, 1970). A combination of backpropagation and gradient descent proved to be a powerful tool for training multilayer perceptions. It allowed us, theoretically, to train neural networks of any number of hidden layers. We use the word 'theoretically' here because even though backpropagation did solve the problem of training multilayer neural networks, it is slow and computationally very taxing, especially on the hardware of that era. Nevertheless, ANNs were again an active research topic, even though the neural networks being built were too small to solve actual practical, real-world problems for quite some time. Before we dive into the idea, intuition, and a little bit of math, it is worth noting a slight problem with backpropagation regarding biological neural networks. Simply put, there is no evidence brains use a similar algorithm or mechanism to achieve learning. Whereas most other components of a modern ANN are abstractions that can be mapped on known biological neural networks, backpropagation cannot. That is fine if you just want to build a functional and accurate ANN. However, the big neural networks have recently started showing disturbing errors that suggest that ANNs have diverged significantly from their biological counterparts, as these errors are not seen in biological systems. We will come back to this later.

#im ../assets/figures/025/025-03.png 50 256 Figure 25.3 - Backpropagation. After a forward pass, computing the predicted out from inputs by propagating the activation through the layers, we can compute the error and loss for a particular input. To update each weight in the network, we back-propagate the error from the output nodes to the previous hidden layer, from the hidden layer to the input layer or another hidden layer, all the way back to the first set of linking inputs to the activation of the input layer.

#h3 25.4 - Backpropagation in a nutshell

#pg Backpropagation is an algorithm that aims to distribute the (prediction) error seen at the output layer backward through the network, computing at each step and each layer how much each node contributed to that error. In this way, the nodes that contributed more to the error will see more significant changes in their weights than those with little to no influence on the error. In order words, it backpropagates the error seen at the output backwards through the network, computing gradients for each node, which are subsequently updated using gradient descent as an optimization algorithm. The actual implementation of backpropagation falls outside of the scope of this course, and even conveying a non-technical intuition beyond what we have already detailed above is a challenge for another day. Safe to say, most if not all neural network toolboxes have very efficient backpropagation algorithms implemented. We felt gradient descent was worth discussing in detail in chapter 14 since it is at the heart of not only neural networks but at the heart of so many machine algorithms. It also has a very clear geometric interpretation that non-technical readers can pick up on without knowing its exact implementation. We feel that it is not a given that an in-depth look at the exact mathematical proof and implementation of backpropagation will help develop a similar intuition. However, we always urge readers to follow the references to learn more. One more thing about the difference between gradient descent and backpropagation. There appears to be some variation in how different researchers and engineers would describe backpropagation, especially in relation to gradient descent. Some consider backpropagation to have replaced gradient descent, while others see them as complementary. We feel the latter is a more accurate take. Remember from Chapter 14 that in gradient descent, we need to compute the gradient given the model's output prediction based on its current weights and input relative to the ground truth of the actual outcome given the same input- the loss. This gradient tells us how to update the weights of our model. Without being able to 'see' into the distance of a loss landscape, we decide that the best way to reach an optimal valley where the loss is (hopefully) at a minimum is to step in the direction of the steepest downward gradient. If you have a neural network that only has an input and output layer (like a traditional perceptron), it is very straightforward to compute these gradients. However, this is not the case for neural networks with one or more hidden layers. The backpropagation algorithm makes it possible to estimate these gradients on all the weights, even those that connect to the hidden layers. The algorithm works backward from the output and computes each node's contribution in any preceding layer to the error computed between predicted and actual outputs. Backpropagation is not updating the weights. It merely helps calculate the gradients needed for updating the weights. In traditional gradient descent, this additional step of updating weights is trivial, and perhaps why some see backpropagation as basically doing all the work. However, updating weights doesn't need to depend solely on the gradients. More modern extensions of the gradient algorithm, such as Adam, are more adaptive and can tweak learning rates and momentum while the algorithm is being trained. To summarize: backpropagation is the algorithm that gives you the gradients, gradient descent, and extensions of it combine the gradients with other performance metrics to update the weights most optimally at every iteration and epoch.

#im ../assets/figures/025/025-04.png 50 256 Figure 25.4 - The Scikit learn toolbox for Python is an excellent way to apply machine learning to your data, including tools to build (relatively simple) multilayer neural networks. For neural networks of considerable size and requiring large amounts of training data, toolboxes that integrate GPU and distributed computing are likely to train, despite the increased overhead of building models in these toolboxes (e.g., TensorFlow).

#h3 25.5 - Scikit-learn

#pg Before committing to TensorFlow as our go-to toolbox to design, implement and train neural networks, we will use Scikit-Learn, a widely used python machine learning toolbox. Scikit-learn has an API that is easier to master and quicker to develop than TensorFlow, Keras, or PyTorch. It isn't optimized to handle the large amounts of training data modern neural networks use to train, and it provides more cookie-cutter-like algorithms. Nevertheless, it is a great starting point to quickly test some ideas, such as different network architectures, the number of hidden layers, activation rules, etc. We, therefore, use it in this chapter to build a few multilayer perceptrons and delay our introduction to TensorFlow for when we dive into the truly deep belief neural networks in the next chapter.

#im ../assets/figures/025/025-05.png 50 256 Figure 25.5 - XOR Problem revisited. (A) A linearly separable dataset. An infinite number of decision boundaries exist that will perfectly predict the class of each point. (B) A non-separable dataset. No matter how hard we try, there isn't a single linear decision boundary we can draw in 25.5B.

#h3 25.6 - The XOR Problem revisited

#pg Let us briefly revisit why a two-layer (input and output) neural network cannot model an XOR logic gate. Instead of the strictly binary logic gate, we can easily create a data set with the same properties and, therefore, the same problem: it is not linearly separable. Consider Figures 25.5A and 25.5B. Both show four clusters of data located at each corner. In Figure 25.5A, the top left and top right clusters are all labeled as 0, while the bottom left and bottom right clusters are labeled as 1. In contrast, in Fig 25.5B, the bottom left, and top right clusters are labeled 0, and the bottom right and top left are labeled 1. Correctly classifying the data in Figure 25.5A with a 2-layer neural network is easy and almost trivial. There are an infinite number of functions y = ax + b that we can think of separating the two clusters perfectly, and you can see one of them in Figure 25.5A, dashed line. Correctly classifying the data in Figure 25.5B, on the other hand, is impossible. For a 2-layer network. Try your best, but you won't be able to draw a straight line that has all data points labeled 0 on one side of it while all data points labeled 1 are on the other side of it. To see why we once more will use geometry to create the necessary intuition of the XOR problem.

#im ../assets/figures/025/025-06.png 50 256 Figure 25.6 - Using OLS, we can estimate the best transformation that collapses the original data in (A) to a single feature (C) which divides the data points of each of the two classes perfectly. In contrast, for our data that is not linearly separable (B), we will find no transformation that cleanly separates data points based on class (C). Additional layers and nonlinearities are needed.

#h3 25.7 - 2-layer neural network and linear regression

#pg In Chapters 23 and 24, we discussed, outlined, and implemented Rosenblatt's Perceptron. Recall that formally we can express the Perceptron as generating output (the prediction) ypredicted from inputs X, learned weights W and the perceptron activation function. Specifically: ypredicted = f(Xw + b), where Xw is the dot product between the inputs and the weights, b is the bias unit, and function f is a thresholding function that assigns 1 to any positive output of Xw + b and 0 to any negative output of Xw + b. If you didn't skip all the chapters up to the start of our discussion on neural networks, you might see a familiar equation in all of this. Indeed, if we ignore the thresholding activation function applied on the output of ypredicted = Xw + b (and thus the output in a 2-layer perceptron), the equation is that of linear regression. In linear regression, we referred to the columns of X as the different regressors in our data and the rows as the individual data points.

#im ../assets/figures/025/025-07.png 50 256 Figure 25.7 - (A) Geometrically, during training, we aim to find a transformation of the inputs X using weight matrix W to transform all points in X onto a new decision space. In this example, points in 3D are mapped into 2D, where a simple thresholding along the output of even one node y will yield accurate predictions. (B) A neural network layer computing the incoming activity on a particular layer starts with computing the dot product between the vector of inputs X (length 4) and the weights of each node (4 weights per 2 output nodes, plus a bias for each node) to produce a response of each node in the receiving layer y. This is mathematically equivalent to applying a linear model in the form y=Xw + b seen in (C). Suppose we apply a non-linear activation function to the output of layer 2. In that case, the neural network can achieve a more complex transformation from layer to layer, allowing it to fit datasets that are not linearly separable.

#br

#pg In neural networks, X represents the input to the network. If we have a dataset X of size n=100 rows by m=4 columns, it means each data point has m=4 dimensions which map onto the four inputs nodes of the Perceptron. If the output layer consists of 2 nodes, the weight matrix that links the input and output layers is thus 4x2 elements in size. Each output node j (2 in total) receives activity from all four input nodes i, where wij indicates the relative weight of that input node i to output node j. We previously learned how to use ordinary least squares (OLS) or gradient descent to estimate the weights W. OLS is a closed-form solution and guarantees optimal computed weights W. This doesn't mean it will always give you good models, quite the opposite. If the data we are trying to model using linear regression isn't linear, OLS will fail to model the relationship between X and y.

#im ../assets/equations/025/png/025-01.png 50 32 Equation 25.1 - Ordinary Least Squares solution to find the best linear transform M that projects X into Y. Suppose this model, which is completely linear, already transforms the data points in input X into a space of predicted output, in which points belonging to one class are separated by a linear hyperplane. In that case, you do not need a more complex model. Remember, when you hear hoofs, think horses, not zebras. Or, when you live in Tanzania or Kenya, you probably want to think of zebras, not horses. Or consider what is probably the best innovation catchphrase, introduced by the Skunkworks team (responsible for designing many hi-tech planes for the US armed forces, including the U-2, SR-71 Blackbird, F-117 Nighthawk, F-22 Raptor, and F-35 lighting): 'keep it simple, stupid'.

#br

#pg Again, the perceptron model is equivalent to the OLS if we leave out the thresholding. However, it might not estimate the same optimal weights, as it uses an iterative method, i.e., gradient descent. However, both approaches are mathematically equivalent. And since the XOR data isn't linearly separable, no linear model will perform well on such data. Regardless of whether we frame it as linear regression or as a neural network model.

#h3 25.8 - Testing for linear separability

#pg How do we know our data isn't linearly separable? Of course, we could spend our days drawing hyperplane after hyperplane through our data to find one that has all inputs of class 1 fall on one side and all inputs of class 0 fall on the other side of the hyperplane. But this because quite tedious, if not outright impossible, for any data set with more than three dimensions. Luckily, we can use linear algebra, specifically OLS itself, to quickly derive an answer to the question of linear separability. It is based on the following. Imagine your data points being distributed on a 1-dimensional line, representing the activity of a single output node predicting one of two. When the class associated with the input is 0, the activity of the output node is less than 0. When the class is 1, the output is greater than 0. In this scenario, it is obvious the data is linearly separable, and we only must test for the sign of the output (which is precisely what the thresholding in the Perceptron model is designed to do). If they do not overlap completely, the inputs belonging to each class are always separable to some degree. And even if the output of each class overlaps partially, there is still an ideal hyperplane that separates them optimally (at output = 0), even though because of the overlap, 100% prediction accuracy is not obtainable. So how do we relate that observation to the true situation in our separable data set (in Figure 25.4A) and non-separable data (Figure 25.4B)?

#im ../assets/figures/025/025-08.png 50 256 Figure 25.8 - Again, in the case of linearly separable data (A), finding a decision boundary that perfectly separates the data points of both classes is straightforward. In contrast, although we might converge on a set of parameters that specify our decision boundary, in the case of non-separable data (B), a linear decision boundary cannot cleanly separate the data points between the two classes.

#br

#pg Enter OLS. Leaving the derivation to textbooks that discuss this topic exclusively and, in more depth, we will state the following as fact. Suppose a global transformation (meaning the same transformation is applied to each point) can geometrically transform our data into a linearly separable dataset (like in Figure 25.5A). In that case, that data is linearly separable. This is a great characteristic of a linear system. Transform them as often as you want; some data properties will not change, like linear separability, provided the transformation is linear, like the rigid or affine transformation that combines translation, rotation, scaling, and shearing we saw in Chapter 12. In practice, you can test for this as follows. Take your original data in n dimensions, X of size (mxn), and associate each data point with one of two outcomes (say -1 for all y = 0 and 1 for all y = 1), yielding a vector Y of mx1. We aim to find the best fitting transformation matrix M that maps our matrix X as close as possible to the vector Y. Since this might require a non-zero translation, we do need to add a column of ones to both X and Y for reasons we explained in previous chapters. But to recap: the column of ones acts as an intercept or offset, a fixed shift of all data points along each axis independently. In neural networks, such an offset is implemented as a bias node. In linear regression, a scaler (the b in y = Xw + b). The remaining elements of matrix M jointly represent other transformations, like rotation, scaling, and shearing. We can find the most optimal transformation matrix M onto Y, using OLS, see Equation 25.1. Subsequently, applying this matrix M to X yields a vector Ypredicted that we can plot as points on a single line. If Ypredicted shows a clear separation between points belonging to each class, we can conclude that the original data X is linearly separable. This is because the linear separability will be retained no matter what linear transformation we apply. Figure 25.6A and 25.6B represent the data in 25.4A and 25.4C after 1) finding the optimal transformation matrix M between the data X (2-dimensional) and outcome Y (1-dimensional) and 2) applying this matrix to the data X. As you can see in these two figures, the transformation matrix estimated for Figure 24A yields a clear separation of the data points in Y. In contrast, for the data in Figure 24B, even the most optimal transformation cannot get all the examples of one class to be to the left of 0 (negative) and the examples of the other class to be all to the right of 0 (positive). You might at this point ask about the thresholding. Can't thresholding introduce nonlinearities that allow for more complex, non-linear models to be captured by 2-layer perceptrons? No, unfortunately not. Not in the case of the 2-layer Perceptron.

#h3 25.9 - 2-layer Perceptrons with a linear activation function

#pg The thresholding at the output layer does not change the function governing the hyperplane given by the weights. This does not change when a non-linear activation function is introduced. Indeed, when we train a standard input-output layer perceptron model on the same two data sets, we can use the first data set to train a model with 100% accuracy, whereas training on the second dataset results in a model predicting inputs with an accuracy not statistically higher than chance. You can work through the notebooks to confirm this for yourself, see Figure 25.9.

#im ../assets/figures/025/025-09.png 50 192 Figure 25.9 - (A) A two-layer Perceptron was trained on the separable (B) and non-separable data (C). The gradient signifies the probability a new input would be classified as green or red. The separable data (B) is easily captured by the Perceptron model, and its decision boundary can easily discriminate between the two classes. In contrast, the same two-layer Perception cannot fit a decision boundary that separates the different classes in non-separable data (C) cleanly and with an accuracy above chance.

#h3 25.10 - Non-linear activation functions

#pg In their book on perceptrons, Minsky and Papert rightfully pointed out that 2-layer perceptrons would always fail to solve non-linearly separable problems. To them, this didn't mean the whole endeavor of neural networks had been a waste of time. On the contrary, they quickly pointed out that by going beyond two layers and introducing intermediate or hidden layers, non-linearly separable problems could be solved. This is true, but it isn't the whole truth. There is another ingredient that our new generation of multilayer networks needs. And that ingredient: non-linear activation functions implemented through the neural network governing the output of all nodes, not just thresholding of the final output layer. Simply put, non-linear activation functions allow neural networks to learn non-linear relationships. A single hidden layer that implements a nonlinearity is already sufficient to solve the XOR problem but introducing a single hidden layer without such a nonlinearity is not.

#im ../assets/figures/025/025-10.png 50 192 Figure 25.10 - (A) A linear activation function is sometimes called an identity activation function. Mathematically, it is identical to applying no activation function but instead just propagates the activation computed from the inputs to the layer and the weights on these inputs. (B) The sigmoid function is S-shaped, asymptoting on both sides of 0. This maps the range of inputs to it to a value between 0 and 1, which is the posterior probability our network assigns to n different output classes. (C) The hyperbolic tangent or tanh function looks very similar to the sigmoid. However, it is bound by -1 and 1 rather than 0 and 1. This means we can't read the output of the tanh activation function as a strictly posterior probability or confidence our networks assigns to our output classes. However, the tanh function and its derivative have other mathematical properties that make them more efficient in training neural networks.

#h3 25.11 - Linear, Tanh, and Sigmoid activation functions

#pg We can introduce nonlinearities in our network by taking the strictly linear component of the output (computed from the inputs and weights using linear algebra) and adding a non-linear component to it by applying a non-linear activation function. Starting with our discussion on linear regression in Chapter 14, we have looked at how we could optimally apply a linear transform to our input data to minimize the error between our model's prediction and the ground truth data (Figure 25.10A). Shortly afterward, we introduced the sigmoid function (Figure 25.10B) for a different algorithmic objective: classification (e.g., logistic regression) rather than regression. Here, we add a common activation function, the hyperbolic tangent, or tanh for short (Figure 25.10C). Although both the tanh and sigmoid are very similar in their shape former has some beneficial properties over the sigmoid in terms of its mathematical properties. The only other notable difference is that even though both sigmoid and tanh are S-Shaped curves, the sigmoid lies between 0 and 1. whereas tanh lies between 1 and -1. Although they introduce nonlinearities, we desperately need to build more complex and interesting neural networks. Neither activation function is without drawbacks. Specifically, as both activation functions asymptote away from 0, we can see a phenomenon in which the gradient is very flat away from the origin of the state space (due to the asymptoting of the activation functions themselves). This is referred to as the vanishing gradient problem, and several methods exist to counteract it. The most powerful of these is introducing yet another non-linear activation function: the Rectified Linear Unit (ReLU). The introduction of the ReLU function is one of the insights that catapulted multilayer neural networks into a new age, where they were rebranded as Deep Belief Neural Networks (perhaps because the term deep belief neural net does have an intriguing and high-tech ring to it). Therefore, we hold off on discussing it in greater detail until the next chapter.

#im ../assets/figures/025/025-11.png 50 256 Figure 25.11 - If a series of layers do not introduce any nonlinearities but instead passes along only the computed output y from inputs X, weights W, and bias b: y = Xw + b, the sequence of transforms can be combined into a single matrix, applied once to the inputs to obtain the output of the network. This reduces a neural network of n number of hidden layers back to the original Perceptron model, still unable to be trained on linearly non-separable input data.

#h3 25.12 - multilayer neural networks with linear activation functions

#pg Having hidden layers is not a guarantee that the mapping between linearly non-separable data can nevertheless be learned by a multilayer algorithm. As outlined before, a sequence of layers with strictly linear activations is mathematically equivalent to combining all weights matrices into one and directly computing the network output from its inputs by a single forward pass using a linear activation function, see Figure 25.11. Verifying this observation, we trained a 3-layer neural network (3 nodes in the inputs layer, three in the hidden layer, and one node as the output). Indeed, even with an extra layer, but without any nonlinearities introduced by a non-linear activation function, the multilayer neural network also fails to predict the non-separably data accurately, see Figure 25.12

#im ../assets/figures/025/025-12.png 50 192 Figure 25.12 - (A) Network architecture with a linear activation function. (B) A linear decision boundary (color gradient) is easily found for linearly separable data. (C) Without any nonlinearities introduced, however, our multilayer neural network does not allow us to now separate the data points of each class any better than our 2-layer traditional Perceptron.

#h3 25.13 - A Multilayer perceptron with strictly linear activation functions.

#pg Without any nonlinearities, a multilayer perceptron computes the activity A of layer n as the dot product of the inputs to this layer (from a previous layer or the actual inputs to the network itself) and the associated weights W of each node of layer n on each of the inputs reaching it. This activity is the output of the nodes in layer n and might serve as the input to a subsequent layer or represent the network's final output. If the latter, we can threshold this activity, giving us a prediction of the output class the input was associated with. However, before that final layer, where we might (or might not) threshold the data or perhaps apply a sigmoid function to it, no matter how many layers we create between inputs and outputs, the computation from layer to layer remains strictly linear. In algebraic terms, we are taking an input X and applying a sequence of transformation matrices M to it until the final output. And as we have already seen, these types of transformations are commutative. Commutative means that the order in which we apply these matrices does not change the outcome. From this, it also follows that all the transformations at each layer can be combined into a single transformation by computing the sequence of dot products between each layer's weight (read: transformation) matrix. In other words, a strictly linearly multilayer neural network is equivalent to a two-layer network when all the weight matrices to each layer have been combined into one. And that means that no matter how many layers we keep adding, it will never do better than a two-layer perceptron.

#im ../assets/figures/025/025-13.png 50 192 Figure 25.13 - When we introduce a non-linear activation function (tanh in our example), our network can learn more complex decision boundaries, making a previously non-separable problem separable. As you can see, the decision boundary now seems to fold to data belonging to each class is highly complex but very effective in accurately predicting outputs from inputs. The colored gradient background continues to signal the neural network's output at each combination of feature 1 and feature 2.

#h3 25.14 - Multilayer neural network with non-linear activation functions

#pg So how does a multilayer perceptron, or any type of multilayer neural network, solve the XOR problem (and others) pointed out by Minsky and Papert? As mentioned above, we need more than just creating intermediate layers to solve issues like the XOR logic gate. Specially, we need to introduce nonlinearities. Nonlinearities allow the neural network to find much more complex, non-smooth, and non-monotonic hyperplanes that we need to build a neural network that can handle problems that aren't linearly separable. And the place to introduce such nonlinearities is the activation function of each node at each layer. Again, geometry allows us to convey this intuition without needing pages of mathematical proof.

#h3 25.15 - Predicting Breast Cancer Risk

#pg Moving away from contrived examples like an XOR logic gate, we will use a real-world data set to train a multilayer neural network. The dataset links several lifestyle choices, bio and demographic data, and environmental factors to a possible positive or negative breast cancer diagnosis. Our task is to see what risk factors have the most significant impact and if two or more risk factors interact to have a non-linear effect on cancer risk. It is helpful to look beyond mere measures of accuracy, recall, precision, and F1 scores. Instead, here we will focus on not just accuracy but also visualizing and interpreting the model to build further on our intuition of what these networks end up learning. We introduce the concept of interpretability here and will look closely at how the weights in our new model capture the relationship between demographics and breast cancer risk.

#h3 25.16 - Predicting Genre from Spotify Audio Features

#pg In a second demo notebook, we use data from the music streaming service Spotify. Users can use their API to retrieve artist and track information, including any musical genres an artist identifies with and 12 audio features extracted from each track. These audio features were developed in-house at Spotify, and little is known about how they were derived as the features remain proprietary now. Luckily, the features are somewhat self-descriptive, and all 12 are listed in Table 25.1. The genres used by Spotify are quite detailed and numerous, so we sought to consolidate them into a few top-level genres: 'house', 'rap', 'pop', 'rock', 'metal',' jazz', 'electronic', 'Latin', 'K-pop', 'country', and 'hip hop'.

#h3 25.17 - Conclusion

#pg In the chapter, we looked closely at how hidden layers and non-linear activation functions can fit more complex datasets.

#h3 25.18 - References

#bs

#be

#bp Linnainmaa, S (1976). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

#bp P. J. Werbos (1982). Applications of advances in non-linear sensitivity analysis. System Modeling and Optimization. Proc. IFIP, Springer.

#bp Minsky, M., Papert, S. A. (2017). Perceptrons: An Introduction to Computational Geometry. United States: MIT Press.

#bp Goodfellow, I., Courville, A., Bengio, Y. (2016). Deep Learning. United Kingdom: MIT Press.

#be