Statistical NLP: course notes

Çağrı Çöltekin — SfS / University of Tübingen

2020-05-21

These notes are prepared for the class *Statistical Natural Language Processing* taught in Seminar für Sprachwissenschaft, University of Tübingen.

7 *Artificial neural networks*

Artificial neural networks (ANNs) are powerful machine learning methods. ANNs have been influential for the development of fields like computer science, artificial intelligence and cognitive science. Throughout their history, ANNs enjoyed times of popularity and times that they were 'out of fashion'. Currently, we are in one of their popular times, mainly thorough the methods known as *deep learning*.[1] In this lecture, we will be discussing some of the basic concepts on ANNs. We will build on these in later lectures as we continue studying various methods relevant to NLP.

ANNs are inspired by (networks of) biological neurons. Neurons (depicted in Figure 7.1) are the building blocks of a biological nervous system. In typical operation (with a lot of simplification), a neuron receives signals from other neurons at its dendrites, at connection points that are called synapses. Depending on the inputs, a neuron either 'fires' or stays inactive. When it fires, an electrical signal is sent thorough its axon, which is then passed to the neurons connected to its axon tendrils. The property of nervous systems that are probably most relevant for ANNs is that they are made of simple units which perform a simple computation. However as a whole the system can perform very complex computations.

For most modern ANNs the connection with biological neurons is just a point of inspiration. We do not take ANNs as models of animal neural networks. ANNs are powerful machine learning methods. As we will soon see, they share a lot with the simple machine learning methods (with no reference to the biological systems) we discussed earlier.

[1] Deep learning, as it is commonly understood, is probably more than only use of neural networks. However, ANNs are at the center of the methods that are collectively called deep learning.

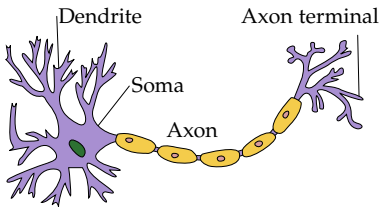

Figure 7.1: A schematic drawing of a biological neuron (image source: Wikipedia).

7.1 *Revisiting perceptron and logistic regression*

Historically, perceptron is the precursor of the neural networks. Remember that perceptron computes a weighted sum of its inputs, passes the sum through a step function, and it is either fires (output $+1$), or does not (output -1). Figure 7.2 shows a schematic description of this process. In principle, one can build a network of perceptrons, resulting in more powerful predictors. However, learning in such a complex networks of perceptrons is not practical. Because of the fact the step function used as the *activation* function is not suitable for learning in more complex networks. The reason for this has to do with the fact that the derivative of the step function is 0 almost

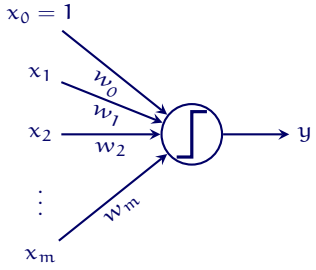

Figure 7.2: A schematic representation of perceptron.

everywhere. As a result, the gradient cannot guide the steps for improving the loss function. As we discussed earlier, the perceptron is trained with a custom algorithm.[2]

We can view logistic regression as a 'soft' version of perceptron. To demonstrate the similarities, Figure 7.3 demonstrates logistic regression similar to the perceptron in Figure 7.2. In words, we get a weighted sum of the inputs, pass the sum through the logistic sigmoid function and output a numeric value in range $(0, 1)$. Since the function (and, hence, the prediction error) we use in logistic regression has non-zero derivative we can use gradient descent to fit the parameters of the model. In fact, a very common unit used in artificial neural networks is identical to logistic regression.

Both the perceptron and the logistic regression are linear classifiers. Although they can solve non-linear classification problems through use of non-linear basis functions, they can only solve problems that are linearly separable in their basic form. Before introducing ANNs, we will first discuss non-linearity.

[2] In fact, another way to think about perceptron algorithm is a (regression) model with a 'hinge loss'. However we will restrict the discussion of ANNs here to more common architectures and use cases.

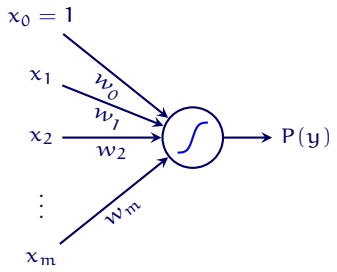

Figure 7.3: A schematic representation of logistic regression.

7.2 *Linear separability and non-linearity*

Two classes are said to be linearly separable if there is a linear boundary between all instances that belong to different classes. Linear separability is an important concept in theory of machine learning, and as we already saw some examples, linearly separable problems tend to be easier to solve. A simple, prototypical example of linearly non-separable problem is the logical XOR function (depicted in Figure 7.4). Remember that XOR of two logical (or binary) variables is true (or 1) if the values differ, and false (or 0) otherwise. The interesting part for us is that the XOR problem is a very simple example of a linearly-non separable problem. It is impossible to draw a line in Figure 7.4 that separates the classes (output of the XOR function).[3]

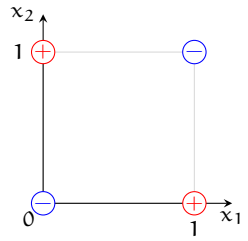

Figure 7.4: XOR function as a counter example of linear separability. The inputs are x_1 and x_2, and the label $-$ used for cases where x_1 xor x_2 is 0, and the label $+$ used for cases where x_1 xor x_2 is 1.

[3] In fact, the fact that perceptron algorithm cannot solve the XOR problem had been one of the reasons that caused a (rather unfounded) disappointment and loss of interest after its first introduction in 1950's.

We already discussed how to turn a linear classifier to a non-linear one. All we need to do is to introduce appropriate non-linear basis functions. For example, if we introduce the basis function $\Phi(\mathbf{x}) = x_1 x_2$ as an additional input, we can easily find coefficients of a linear model that solves the XOR problem. Table 7.1 shows a solution to XOR problem with this basis function. The output of the solution is the XOR value. For perceptron, for example, adding an intercept of -0.5 would return negative sums for the class represented with 0, and positive sums for the other, allowing perceptron to solve this problem. If we map the discriminant line $(x_1 + x_2 - 2x_1x_2 - 0.5 = 0)$ to the original two dimensional input space, we get a non-linear discriminant. Figure 7.5 shows the discriminant line, since the above solution results in a discontinuous function at $x_1 = 0.5$, we have two curves in the plot. The result however, is a solution to the XOR problem.

Table 7.1: A solution to the XOR problem by introducing a non-linear basis function.

x_1	x_2	$x_1 + x_2 - 2x_1x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Another way to look at what we did with adding the basis function $\Phi(\mathbf{x}) = x_1 x_2$ as a predictor to our linear classifier is to map the original input space (non-linearly) into a 3-dimensional space.

Where the each dimension is the terms in the linear equation. Hence, as well as the original input x_1 and x_2, we have another dimension x_1x_2. Figure 7.6 demonstrates this view. Note that in the resulting 3-dimensional space the classes become linearly separable. The red and blue dots in the plot can be separated by a plane.

In general, as we also saw with the polynomial regression example, a non-linear classification problem can be solved with a linear classifier using non-linear basis functions. The multiplicative function we used above is in no way special. There are many non-linear basis function that one can use. In fact, we will see that we can also solve the XOR problem using yet another non-linear function. However, finding useful but minimal (we do not want the too many features, and their associated parameters) non-linear basis functions is not always trivial. Finding/selecting correct non-linear basis functions to be used with linear models is often called *feature engineering*. We will see that one of the advantages of the neural models is reducing this effort by finding the right sort of transformations automatically.

Before finally introducing the neural networks, one last clarification is in order. We often describe non-linearities with abstract functions. For the newcomers to the field, however, it is often unclear what does non-linearity mean in real-world. A common case of non-linearity is the simply a non-linear relation between the predictors and the outcome. A common example for this case is the age and various cognitive abilities, and as a result success in tasks requiring those abilities. When viewed longitudinally, cognitive abilities increase during childhood and youth, however later on, they start to decline with aging. As a result, this so-called U-shaped relation cannot be expressed with linear models. The second common case is interaction. Linear models treat the effects of the predictors additive. The effects add up independently of other predictors. There are many real-world examples where this is not the case. For example, in sentiment analysis the word 'good' is likely be a good predictor of the positive sentiment, while the word 'bad' would likely to indicate the negative sentiment. However, when combined with word 'not', the effects reverse. A linear model adding effects of 'not' and 'good' or 'bad', would not be able to model this interaction. The multiplicative basis functions, like the one in our example, is often a good way to handle these type of non-linear interactions.

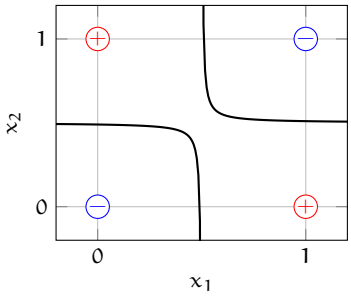

Figure 7.5: A visualization of the solution in table 7.1. Note that the discriminant function is discontinuous at $x_1 = 0.5$.

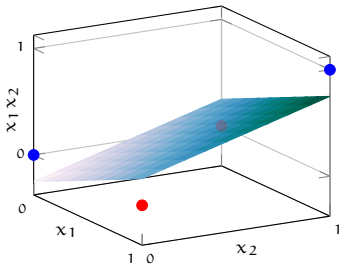

Figure 7.6: Another, three-dimensional, visualization of the solution in table 7.1. Red points mark the positive class (x_1 xor $x_2 = 1$) and blue points mark the negative class (x_1 xor $x_2 = 1$).

7.3 *Multi-layer perceptron*

The simplest neural network architecture is called the *multi-layer perceptron* (MLP). As the name indicates, the network is built by multiple layers of perceptron-like units.[4] Figure 7.7 depicts an MLP with a single hidden layer. The information flow in the network is *feed forward*, the inputs are connected to the hidden layer, the hidden layer outputs are connected to the output layer. There are no backward connections, or connections between the units in the same layer. The

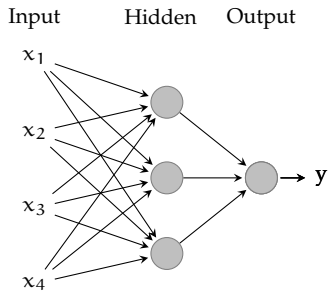

Figure 7.7: A multi-layer perceptron.

[4] Unlike perceptron, however, the units used in modern ANNs have continuous, differentiable activation functions with non-zero gradients which facilitate learning.

layers are also *fully connected*: every unit in a layer is connected to every unit in the next one. The networks we will discuss in this lecture has these two properties. In later lectures we will see networks with sparse connectivity between the units, and ones that are with non-feed forward connections.

A single unit in an ANN functions similar to the linear classifiers. The unit's output is simply a function $f(\cdot)$ of the weighted sum of its inputs:

$$y = f\left(\sum_j^m w_j x_j\right) = f(\boldsymbol{w}\boldsymbol{x})$$

The function $f(\cdot)$, called an *activation function*, is typically a continuous non-linear function. A few examples of common activation functions used in neural networks are shown in Figure 7.9.

The first activation function shown in Figure 7.9 is the now familiar logistic function. The second one is another, s-shaped (sigmoid) function, hyperbolic tangent (tanh). These two functions have been popular since the early days of neural networks. The last one, rectified linear unit, or ReLU, is a piecewise linear function that became popular relatively recently. Common to these functions are that they are differentiable, and have non-zero derivatives (in the range they are intended to operate). In principle, one can use any differentiable function as an activation function. However, some activation functions facilitate learning, and those are used more often in practice.

As hinted above, the choice of activation functions is rather flexible. However, this is true for the hidden layers. On the output layer, the task we want to solve restricts the choices. Although not exclusive, (logistic) sigmoid is most popular choice of activation function for binary classification. As you would remember from the logistic regression, this allows us to interpret the output of the model as the probability of the positive class conditioned on the input. For multi-class classification, the *softmax* function we introduced earlier is a common choice. Remember that softmax is is a generalization of the logistic function to more than two classes, and defined as,

$$P(y = k \mid x) = \frac{e^{\boldsymbol{w}_k \boldsymbol{x}}}{\sum_j e^{\boldsymbol{w}_j \boldsymbol{x}}}.$$

Note that the equation above means that we have one output unit for each class label. The expression in the denominator makes sure that the outputs of all units sum to one.

Although we will mostly discuss neural networks in the context of classification, they can also be used for regression problems. In this case, we typically use the identity function as the activation function. The hidden layers still help finding a non-linear solution (or intermediate representation that maps the non-linear problem to a linearly-solvable one), and the we can view the final output layer, which simply outputs a weighted sum of its input without any non-linear activation function, as scaling the internal representations built by the network to the correct scale/unit of the output variable.

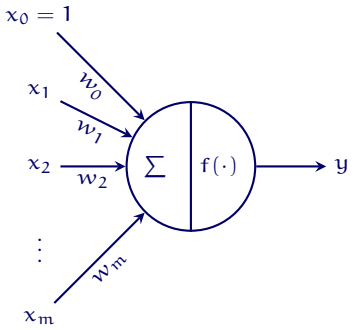

Figure 7.8: A depiction of a single unit in an artificial neural network.

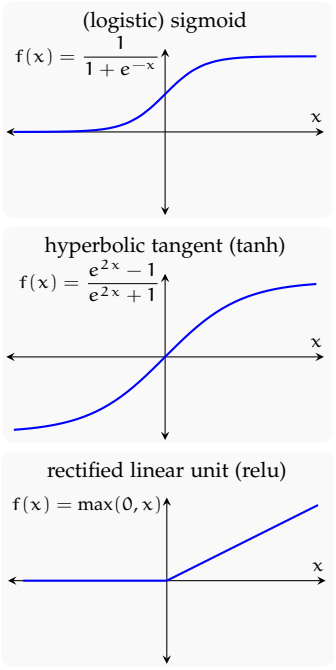

Figure 7.9: Common activation functions for neural networks.

7.4 *Forward propagation in neural networks*

We now are ready to fully define the output of a feed-forward network. We will do this through the simple example presented in Figure 7.10. This network has two predictors and two output variables to predict. The activation function choices of the hidden units is rather flexible, one can choose any activation function that makes sense for a particular problem. However, as noted earlier, the choice is made among a set of well-known, well-tested activation functions. The activation functions for the output units are based on the type of problem solved by the neural network. For this particular example, we have a single hidden layer with two units.

To calculate the output values, it is generally more convenient, and easy to understand, to break down the computations involved into pieces. In this network, we first calculate the weighted sum of the input variables for each hidden unit, and apply the activation function $f(\cdot)$. Then, the units in the output layer takes the output of the hidden layer, compute the weighted sum, and apply the output activation function $g(\cdot)$. More formally,

$$h_j = f\left(\sum_i w_{ij} x_i\right) \quad \text{and} \quad y_k = g\left(\sum_j v_{jk} h_j\right).$$

Note that the input to the activation functions are simply dot products of input vectors and the corresponding weight vectors. If we write the weights for each layer as matrices,

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \quad \text{and} \quad \boldsymbol{V} = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix}$$

we can simplify our notation with,

$$\mathbf{h} = f(\boldsymbol{W}^T \mathbf{x}) = f(\mathbf{x}^T \boldsymbol{W}) \quad \text{and} \quad \mathbf{y} = g(\boldsymbol{V}^T \mathbf{h}) = g(\mathbf{h}^T \boldsymbol{V})$$

where, following the convention, we consider input and hidden vectors as column vectors. What is important to realize here is that, the network computes a series of matrix-vector products, followed by elementwise application of the activation functions. Viewing inputs and outputs of each layer as vectors, function of each layer is performing a (non-linear) transformation of its input.

It is also important to realize that the function the whole network implements can be represented by composition of the functions at each layer. Putting the above together,

$$\mathbf{y} = g\left(f(\mathbf{x}^T \boldsymbol{W}) V\right).$$

To make this discussion more concrete, we will go through a simple example, that intends to solve the XOR problem. The network is schematically described in Figure 7.11. For the sake of example, the weights (marked on the edges) are determined manually. Normally, we want to learn these weights. Since we have a binary classification

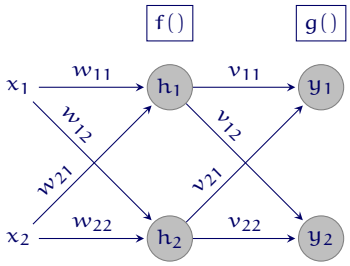

Figure 7.10: A simple multilayer perceptron. $f(\cdot)$ and $g(\cdot)$ are the activation functions used in hidden and the output layers respectively. For the sake of simplicity, we do not have an intercept (bias term) in this model.

[[IMAGE DISCARDED DUE TO '/tikz/external/mode=list and make']]

Figure 7.11: An MLP for solving the XOR problem. The inputs at the bottom that are always 1 are intercept terms.

problem, we use the logistic sigmoid activation at the output units. For the hidden layer, we use square function as activation, which is unusual in real applications, but makes hand-calculations easier.

Now, we go through calculations of input vector $(0,1)$ explicitly.

$$h_1 = f(1 \times x_1 + 1 \times x_2 - 2) = (0 + 1 - 2)^2 = 1$$
$$h_2 = f(1 \times x_1 + 1 \times x_2 + 0) = (0 + 1 + 0)^2 = 1$$
$$y = g(1 \times h_1 + 1 \times h_2 - 3) = \frac{1}{1 + e^1} = 0.73$$

The output of the network for the other possible input combinations can be calculated similarly. Remembering that the operation above can be performed by matrix-vector product, we also can write down our weight matrix, and multiply with the input matrix.

$$\mathbf{h} = f\left(\begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix} \times \begin{bmatrix} -2 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}\right)$$
$$\mathbf{y} = g\left(\begin{bmatrix} 1 & h_1 & h_2 \end{bmatrix} \times \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}\right)$$

We do not explicitly calculate the rest of the input values we are interested, but we give the network's output, including the values at the hidden layer, for all relevant values for the XOR problem in Table 7.2. Note that the output of the network for inputs $(0,1)$ and $(1,0)$ is above 0.5. As a result we classify these values as belonging to the positive class, and since the other two input are below 0.5 they are assigned to the negative class.

Another interesting observation with the solution is presented in Figure 7.12. The upper panel plots the original XOR problem similar to Figure 7.4. The lower panel shows how these points are transformed by the hidden layer. The points that represent different classes on the lower panel are linearly separable. As a result, the output layer, which is simply a binary logistic regression classifier can find a solution. The transformation performed by the hidden layer turns a problem that is not linearly separable into a linearly separable one.

Note that the reason the hidden layer can transform a non-linearly-separable problem into a linearly separable one is the fact that it uses a non-linear activation function. Without the non-linear activation, regardless of the depth of the network, what we do is a series of matrix–vector multiplications, in other words, linear transformations. As demonstrated in Figure 7.13, wihtout non-linear activation functions, the result is yet another linear transformation.

Table 7.2: The solution for the XOR problem using the network in Figure 7.11.

x_1	x_2	h_1	h_2	y
0	0	4	0	0.27
0	1	1	1	0.73
1	0	1	1	0.73
1	1	0	4	0.27

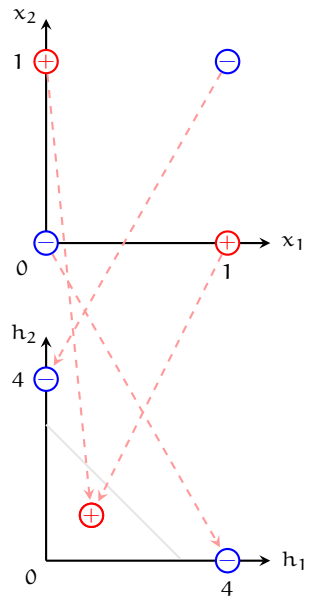

Figure 7.12: A demonstration of the transformation computed by the hidden layer of the network presented in Figure 7.11.

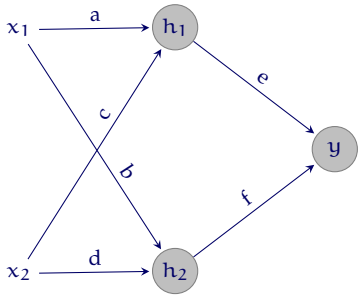

Figure 7.13: A simple network for demonstrating the need for non-linear activation. Without non-linear activation functions the output of the network is $y = (ea + fb)x_1 + (ec + fd)x_2$, a linear transformation of the input variables.

7.5 *Learning in neural networks*

Like the earlier methods we discussed, learning in ANNs is achieved thorough minimizing an error function. The choice of exact error function is related to the task and the network architecture. In general, the minimum of the neural network error functions cannot be found using analytic solutions (as in regression). As a result, we need to employ a search strategy like *gradient descent*, to find the minimum of the error function. We have already seen models whose minimum error can be found using gradient descent. As long as the error function is convex, gradient descent can find the global minimum of the error function. The problem we face with neural networks is that the error functions are not necessarily convex. There may be multiple minima as demonstrated in Figure 7.14. Although we want to find the *global minimum*, gradient descent is not guaranteed to find it. It may stop in one of the *local minima*. Some training procedures may help avoiding local minima. However, there is no general solution to find the global minimum of an ANN error function.

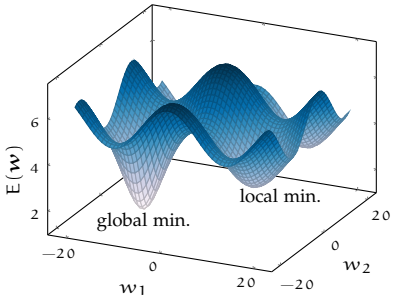

Figure 7.14: A demonstration of multiple minima with two parameters.

7.5.1 *Backpropagation*

Another issue about learning in neural networks arises because of the layered architecture of the system that makes learning in ANNs more challenging. It is computationally non-trivial to assign credit or blame to the weights of the non-final layers. We will discuss the solution and the problem through the simple example we presented earlier, which is repeated with slight modification in Figure 7.15. The figure indicates two possible paths in the network that may have caused the error on output unit y with two different colors. If we had a single layer, gradient descend would update the weights e and f based on their partial derivatives (the steepness of the error function in the corresponding dimension). Since we do not have direct notion of error in the hidden layer, we need a mechanism to determine how to distribute the responsibility for error.

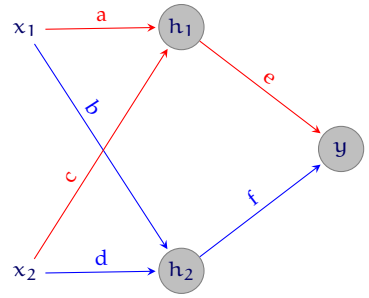

Figure 7.15: A simple neural network for demonstrating the propagation of error. In a single layer network (without hidden layer) the error would be due to weight e or f. For the network above, the error at output node y need to distributed to the weights marked in red and blue.

As noted above, we want to essentially use gradient descent, which means we need the gradient of the error with respect to the weights. We will go through a (very) simplified example based on the network in Figure 7.15. For the sake of demonstration, we will assume that we are minimizing y (normally we minimize the error which is a function of y, but we will soon see that the principles apply to more realistic cases as well). The gradient of the whole network is the vector

$$\nabla y = \left(\frac{\partial y}{\partial a}, \frac{\partial y}{\partial b}, \frac{\partial y}{\partial c}, \frac{\partial y}{\partial d}, \frac{\partial y}{\partial e}, \frac{\partial y}{\partial f} \right)$$

that is, the partial derivatives of the network with respect to each weight. For the sake of demonstration we will calculate the partial derivatives with respect to e, a and c, with some heavy simplification. Partial derivative $\frac{\partial y}{\partial e}$ can be calculated in a rather straightforward way for a differentiable function. All other parts of the network

are constant terms with respect to e.

Calculation of the partial derivatives with respect to a and c is slightly more involved, as h_1 is a function of these variables. However, we can simply apply the chain rule of derivatives[5]

$$\frac{\partial y}{\partial a} = \frac{\partial y}{\partial h_1}\frac{\partial h_1}{\partial a} \quad \text{and} \quad \frac{\partial y}{\partial c} = \frac{\partial y}{\partial h_1}\frac{\partial h_1}{\partial c} \,. \tag{7.1}$$

The main point here is that we can calculate the partial derivative with respect to any of the weights. If our networks gets deeper, the terms for the earlier weights will have more terms due to repeated application of the chain rule, and due to the fact that in our representation above each layer implements two pieces of computation (a linear mapping followed by application of the activation function). Hence, once we factor these in, we will have more terms in the partial derivation calculations in Equation 7.1 even for our simple network above. Furthermore, we can consider the error function as a final node, taking the output of the network, and calculating the error, which means the above notion of calculating gradient works. In fact, this will work for any *computation graph* without cycles.

So far, what we did was just math, telling us a we can calculate the gradient for a feed-forward network. However, you should note that the term $\frac{\partial y}{\partial h_1}$ Equation 7.1 is required for calculating the partial derivatives with respect to both a and c. Repeated calculation of same the same quantities makes a naive attempt to implement the above procedure computationally very inefficient. Making it impossible to use in most modern neural networks which include thousands, if not millions, of parameters. Also note that we typically calculate the error on a large number of inputs with many dimensions, which makes the problem even more complex.

The solution to this problem is called the *backpropagation algorithm*. The idea is similar to many dynamic programming algorithms. The backpropagation algorithm simply stores the quantities like $\frac{\partial y}{\partial h_1}$ above and avoids recalculating them.

[5] In general, derivative of a function $F(x) = f(g(x))$ is calculated using the chain rule of derivatives:

$$F'(x) = f'(g(x))g'(x)$$

7.5.2 *Stochastic and mini-batch gradient descent*

In typical gradient descent learning, the gradient is calculated using the complete training data. However, with large training sets this is computationally inefficient. Together with large number of parameters, the space complexity (required memory) may become an important issue.

There is a well-known, memory-efficient variant, *stochastic gradient descent*, which updates weights for every single training instance. Since the stochastic gradient descent changes the weights for every single training instance, it is noisy, it may sometimes take steps in the opposite direction of the minimum. However, in the long run, it is known converge to the same minimum.

Figure 7.16 demonstrates the possible paths taken by gradient descent and the stochastic gradient descent on an error surface defined on two paramters. Since the error surface is a function of the whole

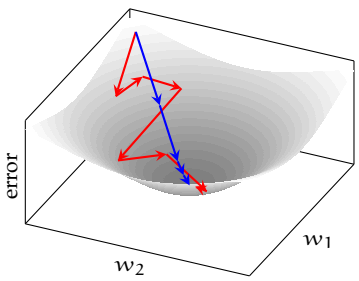

Figure 7.16: A demonstration of the paths taken by gradient descent (blue) and stochastic gradient descent (red) on an (hypothetical) error surface.

input data, the gradient descent will take sure steps toward the minimum with fewer steps. The stochastic version will wander around the error surface more since the gradient is calculated only based on a single input, and likely to take many more steps. However, stochastic gradient descent will also require fewer calculations and less memory at every step.

In practice, it is more common to use a *mini batch* update strategy that is a compromise between full gradient descent and the stochastic version. Mini batches are computationally more attractive, they both fit into memory and can also be computed much faster on vector processing hardware such as graphical processing units (GPUs) in comparison to stochastic gradient descent.

Furthermore, it turns out the batch size is an important parameter in many cases. The choice of batch size affects the outcome of training a neural network, and often large batch sizes may be non-optimal. This effect is likely due to the fact that (full) batch gradient descent converges to the closest minimum with sure steps even if it is a rather 'shallow' local minima. On the other hand stochastic or mini-batch version may skip over the minor 'bumps' in the error surface, eventually finding a better (local) minimum on the error surface.

7.5.3 Countermeasures for overfitting

As in any machine learning model, ANNs can also overfit. They may be even more prone to overfitting due to their complexity in comparison to, e.g., linear classifiers. As in linear models, one way to counteract overfitting is regularization. One can apply L1 or L2 regularization to ANNs, by adding L1 or L2 norm of the weights to the error function.

Another popular method to prevent overfitting is *dropout*, where a randomly chosen input and/or hidden unit in the network is 'turned off', by setting their (output) value to 0. Each layer, has access to only a random view of its input for each input instance. As a result it is forced to learn from partial information. Dropout is known to reduce overfitting. It is also seen as learning an ensemble of multiple classifiers, each operating on a subset of features. It is a technique that is typically used in practice.

Another method of preventing overfitting is *early stopping*. The idea with early stopping is to monitor the loss on a validation set at every epoch (or some other interval like every batch update), and stop when the validation error starts increasing.

The above is not the exclusive listing of the possibilities, and use of one of these methods does not prevent the use of the others. A combination of the three methods discussed above (and others) can also be used within the same network architecture.

7.5.4 *Some tricks of the trade*

Non-convex error functions (multiple minima) mean that gradient descent will not necessarily find the global minimum while training neural networks. Furthermore, the large number of possible variations of architecture and hyperparameters[6] make neural network training often more involved than traditional (linear) models. Although, their renewed popularity made it easier (through common practices, higher-level libraries with better default behavior), training neural networks well, particularly the networks beyond simple ones, require a substantial amount of (hands-on) experience. How to train neural networks properly and efficiently is an active area of research, and often theory and understanding lags behind some established practices. Here, we try to point out some of the common issues.

[6] Just to list a few typical ones: number of layers, number of units at each layer, activation functions at each layer, regularization method and its parameters, number of epochs to train the network, weight initialization, batch size, learning rate, (adaptive) learning method and its parameters ...

One common alternation to gradient descent while training neural networks is to add *momentum*. Even though the mini-batch training is indeed found to be useful, and used in practice almost exclusively, smaller batch sizes may also cause the jumps at every step of gradient descent as demonstrated in Figure 7.16. To make sure that the mini-batch methods do follow a more straight course, one can add the previous gradient (or an average of previous gradients). With the momentum, gradient descent updates become, for example,

$$\Delta w_{ij}(t) = \eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1)$$

where η is the usual learning rate, and α is another hyperparameter determining the strength of the momentum. Intuitively, momentum cause a larger update if the current gradient is in the direction of the previous gradient(s), otherwise it will change its course towards the earlier course of the descent.

Another important factor in neural network training is the *learning rate*. Typically we want to start with higher learning rate, and reduce it as the learning progresses. Simple algorithms that 'decay' the learning rate (e.g. linearly) based on number of iterations are often used in practice. However, there are quite a few *adaptive* algorithms which set the learning (and possibly other parameters such as the momentum parameter we discussed above) in a smart way. We will not go into details of each of these *optimization algorithms*, but note that there are quite a few of them and most machine learning libraries or platforms offer out-of-the-box implementations.[7] For most ANN practitioners, using one of these algorithms is often more practical (for both finding a good minimum and for finding it quickly) than custom adaptions.

[7] Just to name a few: Adagrad, Adadelta, RMSprop, Adam, ...

Summary

This lecture is a first (gentle) introduction to ANNs. The later lectures will cover some (more complex) ANN architectures used in practice.

With the popularity of deep learning, the many tutorials and books on neural networks became available. For general, more comprehensive/technical, introductions to ANNs, the readers are referred to usual textbooks in the field (Hastie, Tibshirani, and Friedman 2009; MacKay 2003; Bishop 2006, e.g.,). For a more NLP-oriented discussion, the third edition of Jurafsky and Martin (2009) includes a chapter on neural networks, and Goldberg (2016) includes a survey of the use of various ANN architectures in NLP along with an introduction (there is also a more recent book, Goldberg 2017, with a similar content).

Bibliography

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer. ISBN: 978-0387-31073-2.

Goldberg, Yoav (2016). "A primer on neural network models for natural language processing". In: *Journal of Artificial Intelligence Research* 57, pp. 345–420.

Goldberg, Yoav (2017). *Neural Network Methods in Natural Language Processing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers. ISBN: 9781627052955.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. Springer series in statistics. Springer-Verlag New York. ISBN: 9780387848587. URL: http://web.stanford.edu/~hastie/ElemStatLearn/.

Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.

MacKay, David J. C. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. ISBN: 978-05-2164-298-9. URL: http://www.inference.phy.cam.ac.uk/itprnn/book.html.