Topic 1
An Introduction to Neural Networks an Deep Learning
An Overview
(Draft)

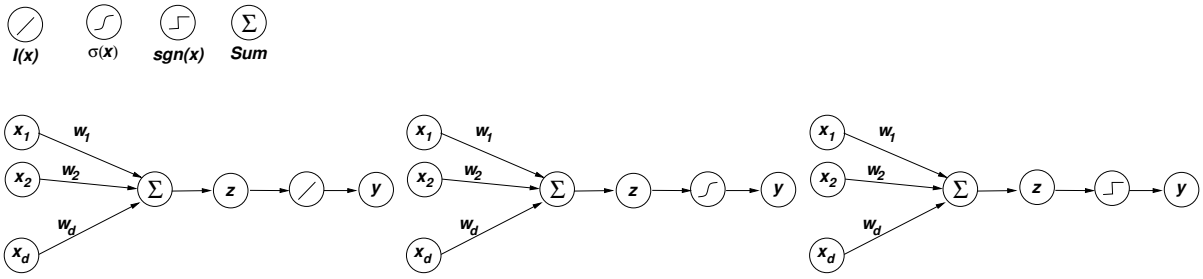
Instructor: Farid Alizadeh

January 27, 2020

1 Introduction

Using the building blocks of linear and logistic regression methods, neural networks construct more complex models for learning complicated cases.

Before we discus neural networks let us show linear and logistic regression in the form of diagrams:



Each diagram first computes a *linear combination* of the input variables $x_1, \ldots, x_d$; the result is $z = w_1x_1 + \cdots + w_dx_d$, which is the output of the node with Σ inside it. Then a function $\phi(\cdot)$ is applied to z and $y = \phi(z)$ is output. The function $\phi(x)$ is called the *transfer function* or the *activation function*. Its inverse is called the *link function*. There are many different functions available, but the most common ones are:

1. **The identity function:** $I(x) = x$ It basically does not change its input. In the left diagram above this function is used. As a result this diagram simply represents linear regression: $y = w_1x_1 + \cdots + w_dx_d$.

2. **sign function** and the **symmetric sign function:**. The $\text{sgn}(x)$ and $\text{ssgn}(x)$ are, $\{0, 1\}$ valued respectively, $\{-1, 1\}$ valued functions. Depending on the context it could be one of the following functions:

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad \text{ssgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{otherwise} \end{cases}$$

   The diagram on the right uses the sign function, and as a results it corresponds to any function that directly classifies according to whether z is positive or negative. So this diagram corresponds to any classification method with linear boundary separating the two classes of y. Also, the sigmoid function $\sigma(x)$ and they hyperbolic tangent functions can be thought of as smoothing or relaxation of the these functions.

3. **The sigmoid and the hyperbolic tangent functions:** $\sigma(x) = \frac{\exp(x)}{1+\exp(x)}$, maps the real line $[-\infty, +\infty]$ to the interval $[0, 1]$. In a sense this function acts a cumulative distribution function (CDF) used in probability theory. Indeed any CDF can replace the sigmoid function.

   The hyperbolic tangent $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$, essentially acts the same way as the sigmoid function, but instead maps the real line to the interval $[-1, 1]$.

   The sigmoid and the tanh functions may be regarded as *softening* and *smoothing* of the sign and the symmetric sign functions. In fact, consider $\sigma(\alpha x)$ and $\tanh(\alpha x)$. As α gets larger, the transition from zero to one for the $\sigma(\alpha x)$ becomes sharper. Similarly, the transition from -1 to 1 becomes sharper as α gets larger. At the limit, the sigmoid and the hyperbolic tangent functions converge to the corresponding sign functions.

4. **Rectified Linear Activation Function (ReLU):** where $f(x) = x$ if $x \geq 0$ and $f(x) = 0$, otherwise. This is a linear function that when composed with linear transformations of data generates flexible piece-wise linear functions. There are many variations of this function, including smoother version, and one that is not equal to zero for $x < 0$, but rather is very slowly decreasing function.

5. **SoftPlus function:** is $f(x) = \log\big(1 + \exp(-x)\big)$. This is a smoothed and softer version of ReLU function.

6. **Hardmax function:** is also multivalued function $\mathbf{f} : \mathbb{R}^n \to \{0, 1\}^n$, where $f(x_i) =$ if $x_i = \max\{x_1, \ldots, x_n\}$, otherwise $f(x_i) = 0$. Note that this terminology is actually not accurate: the output of this function is not really the maximum. Rather, the output is the argmax, not the max. In

other words, the output of the hardmax functions is the *index* at which the maximum is achieved. The definition of the hardmax function is:

$$\max(x_1, x_2, \ldots, x_d) = (o_1, o_2, \ldots, o_d)$$
$$\text{where} \quad o_i = 1 \text{ if } x_i \text{ is the largest among } o_j, \text{ and } = 0 \text{ otherwise.}$$

7. **Softmax function:** is a multivalued function $f : \mathbb{R}^n \to \mathbb{R}^n$, where

$$f_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}.$$

This function is the one we use for multi-class logistic regression. As in the sigmoid function, softmax is the smoothing of the hardmax function. Again, softmax(αx) gets closer to the hardmax function as α gets larger.

2 Basic One Layer Neural Network

We have seen that linear and logistic regression approaches, along with using nonlinear basis functions are powerful tools for learning theory. The *neural networks* start from these models and add several levels of complexity to obtain more sophisticated, and complicated models. We now introduce these additions.

Multiple linear combinations of input variables

In linear and logistic regressions a single linear combination of input data z is computed. We can instead create k different such linear combinations:

$$\begin{aligned} z_1 &= w_{11}x_1 + \cdots + w_{1d}x_d \\ z_2 &= w_{21}x_1 + \cdots + w_{2d}x_d \\ &\cdots\cdots \\ z_k &= w_{k1}x_1 + \cdots + w_{kd}x_d \end{aligned} \tag{1}$$

The number of new variables z_i, that is k could be smaller or larger than the number of original variables x_i that is d.

If $k < d$, it means that we deem many variables in the original data insignificant or redundant. So we are *collapsing* or *projecting* the original d features into a smaller number k of variables.

If $k > d$ we are creating more features that are in the original data. This is done to get more flexible models. We have already done this in polynomial and spline models, where one or two original variables are turned into many more features to allow more flexible polynomials or splines. Here we are using only linear transformations. You may think of each z_i as a different response variable depending on the original variables x_i. But the z_i are not the final variables of interest. Rather, they are intermediate variables which in turn can be helpful to determine the final response variable(s) of interest.

Just to give a simple example, suppose based on just education (x_1) and career (x_2) we wish to see if a person should be approved for certain type of long term credit or not (y). We could first try to estimate from education and career the person's income (z_1), the number of years they will stay in their job (z_2), and the annual raise in their income they may get (z_3). Then from these features we may be able to better estimate whether they qualify for a loan of particular duration. The problem is that the data available to us may not have information about these intermediate variables. Also, It may be that the intermediate variables z_i have no particular interpretation at all. They may be mathematical and artificial constructs that facilitate our modeling of response variables y as a function of input variables x_i. This is essentially what we did in nonparametric linear and logistic regression.

Vector and matrix notation

The linear relations in (1) between x_i and z_j can be expressed more succinctly in vector/matrix notation:

$$\boldsymbol{z} = W\boldsymbol{x} \quad \Leftrightarrow \quad \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1d} \\ w_{2,1} & w_{2,2} & \dots & w_{2d} \\ \vdots & \vdots & \ddots & \dots \\ w_{k,1} & w_{k,2} & \dots & w_{kd} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ x_k \end{pmatrix}$$

Here, $\boldsymbol{x} = (x_1, \dots, x_d)^\top$, $\boldsymbol{z} = (z_1, \dots, z_k)^\top$, both are column vectors[1], and W is a $k \times d$ matrix, a two dimensional array. We use boldface letters to represent column vectors in general. Note that if we also wish to add a constant vector $\boldsymbol{w}_0$ be added to the result, the formula will be $\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{w}_0$. The quantity $\boldsymbol{w}_0$ is called *bias term*. Another way to include the bias term is to assume that $\boldsymbol{x}$ contains, as its first element, the number 1. Then we can stay with the slightly simpler $\boldsymbol{y} = W\boldsymbol{x}$ notation.

Nonlinear transformation through activation functions

After generating the hidden variables z_i by linear combinations of the input variables, we apply an activation function which is usually (but not always) a nonlinear function. There are many choices for activation, and their choice depends on the type of neural network we are building. As we said earlier some of the more common functions are: the identity function $I(x) = x$ which is usually used for regression problems at the output level; the sigmoid $\sigma(x) = \frac{\exp(x)}{1+\exp(x)}$ which is usually used to turn a real number into a probability, and is suitable for classification; and the sign function $\text{sgn}(x) = 1$ if $x > 0$ and $\text{sgn}(x) = 0$ if $x \leq 0$, which is useful when we want to directly classify without computing probabilities first.

[1]The superscript $(\dots)^\top$ means *transpose* which turns a row vector into a column vector and a column vector into a row vector.

Another pair of common activation functions are the hardmax function and the softmax, the ReLU and even linear functions. These functions take r variables and output r outcomes. The softmax and hardmax functions are specially useful for the case where y represent a multi-class classification problems. For instance, a problem where we need to determine if an e-mail is spam, business related, or personal can in the last stage use the softmax function to obtain a probability estimate of each class, and use the hardmax function to definitively select the class. Similarly, the softmax function takes d inputs and outputs d outcomes, and is defined as follows:

$$\text{softmax}(x_1, x_2, \ldots, x_d) = (o_1, o_2, \ldots, o_d)$$
$$\text{where } o_i = \frac{\exp(x_i)}{\exp(x_1) + \exp(x_2) + \cdots + \exp(x_d)}$$

The softmax function can be interpreted as the probability of belonging to the i^{th} class; the sum over all class probabilities adds up to one. The relation between softmax and hard max is similar to the relation between logistic function and the sign function. These functions are used when there are many outputs $y_1, y_2, \ldots$ representing a multi-level classification problem (as opposed to binary classification).

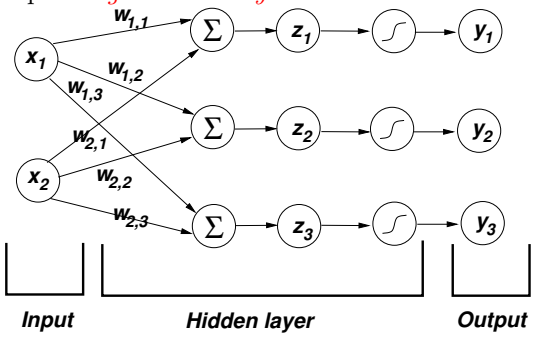
We can also show the nonlinear activation relation in vector format as follows:

$$\mathbf{y} = \boldsymbol{\phi}(\mathbf{z})$$

where $\boldsymbol{\phi}$ takes a vector of length k and returns another vector of length k (more generally, the output and input vectors could be different sizes.) In neural networks, all the functions in $\boldsymbol{\phi}(\dot{})$ are the same (for example all are the logistic function, or all are the sign function).

Graphical representation of single hidden layer neural networks

With this set up a *single hidden layer neural network* can be depicted as :

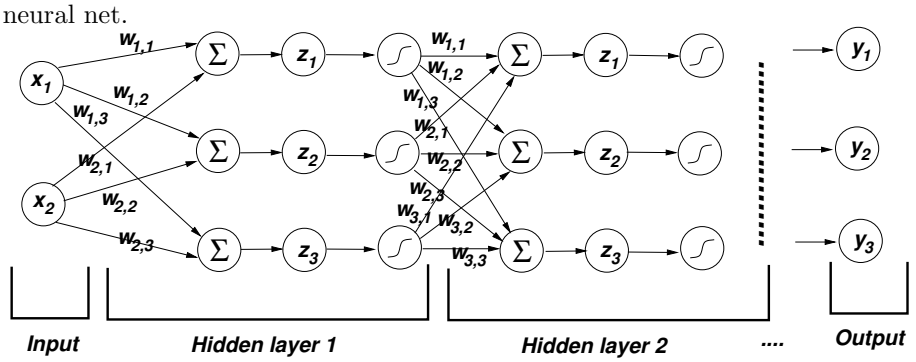


In this case we use two inputs, three outputs and three hidden variables (called *hidden units* in neural net jargon). Also in this case all the activation functions are equal the logistic function.

3 Multi Layer Neural Networks

In the single hidden layer neural networks, we can control the complexity of the model by the number of hidden units z_i. The larger the number of such units, the more complex the neural network models is. It is possible to add complexity in a different way by adding more hidden layers to the neural net.

Instead of sending the outcome of the hidden layer to the output by setting $\mathbf{y} = \boldsymbol{\phi}(W\mathbf{x})$, we could instead add another layer. This means that we could take the outcome of the hidden layer, create a second layer of linear combinations and apply another set of activation functions. This adds a second layer. The process can be continued as many times as we wish. The result is a multi layer neural net.



In this diagram each layer has its own weights, and its own hidden variables. The number of hidden variables could be different in each layer, and the matrix of weights W are also different and with different dimensions.

To sum up, the neural network computes a function $\mathbf{y} = f(x_1, \ldots, x_d | W_1, \ldots, W_k)$. Due to the layered structure of the neural networks the function $f(\cdot)$ can be written as *composition* of functions as follows:

$$f(\mathbf{x}|W_1, \ldots, W_k) = \boldsymbol{\phi}_k \left(W_k \ldots \left(W_3 \boldsymbol{\phi}_2 \left(W_2 \boldsymbol{\phi}_1 (W_1 \mathbf{x}) \right) \right) \ldots \right) \tag{2}$$

where $\boldsymbol{\phi}_i$ are the activation functions. As can be seen this is a highly nonlinear function which is nonlinearly dependent on the parameters $W_1, \ldots, W_k$. We must first determine these parameters, and then use them to predict the output values y_i from a new set of input values x_i.

4 Training Neural Networks

The training process is similar to the linear regression (for continuous y_i) or logistic regression (for classification problems.)

Suppose the data is a set of examples $(\mathbf{y}_1, \mathbf{x}_1), \ldots, (\mathbf{y}_N, \mathbf{x}_N)$, one or both of which could be *vectors* of data. We use the general maximum likelihood method to estimate the most likely parameters $W_1, \ldots, W_k$. The process is in principle,

and on the surface, is similar to linear and logistic regression we have already seen. However, the optimization problems that arise are far more difficult than what we have seen in linear and logistic regression.

Regression with neural networks

In case of regression, we could use the least squares process for computing the estimate for W_i:

$$\min_{W_1,\ldots,W_k} \sum_{i=1}^{N} \big(y_i - f(x_i|W_1,\ldots,W_k)\big)^2$$

To find the minimum, and minimizing W_i, we take derivatives with respect to each parameter and set equal to zero, and attempt to solve the resulting system of nonlinear) equations.

Class probability estimation with neural networks

For classification, first let's suppose that we have only a single binary output variable y. Then usually the last activation function is the logistic function $\sigma(z)$. The result is that we are minimizing the negative entropy function

$$\min_{W_i} \sum_i \big(-y_i \log(p_i) - (1-y_i)\log(1-p_i)\big)$$

where $p_i = \frac{\exp(f(x_i|W_i))}{1+(f(x_i|W_i))}$. Again, when the function $f(\cdot)$ is linear, as in the logistic regression, we can find the optimal W_i by iterative methods. We essentially have to follow the same approach here.

For multiple class case with say K classes, the entropy function to be minimized is

$$\min_{W_i} \sum_i^N \sum_j^K \big(-y_{ij} \log(p_{ij})\big)$$

Here, y_{ij} is 1, if the i^{th} item in the data is part of class j, and zero otherwise.

The p_{ij} are usually calculated based on the softmax function. If the last layer has K outputs, $f_j\big(x|W^{(1)},\ldots,W^{(k)}\big)$, one for each class, then

$$p_{ij} = \frac{\exp\big(f_j(x_i|W^{(1)},\ldots,W^{(k)})\big)}{\sum_{r=1}^{K} \exp\big(f_r(x_i|W^{(1)},\ldots,W^{(k)})\big)}$$

Additional complexity of optimization process in neural networks

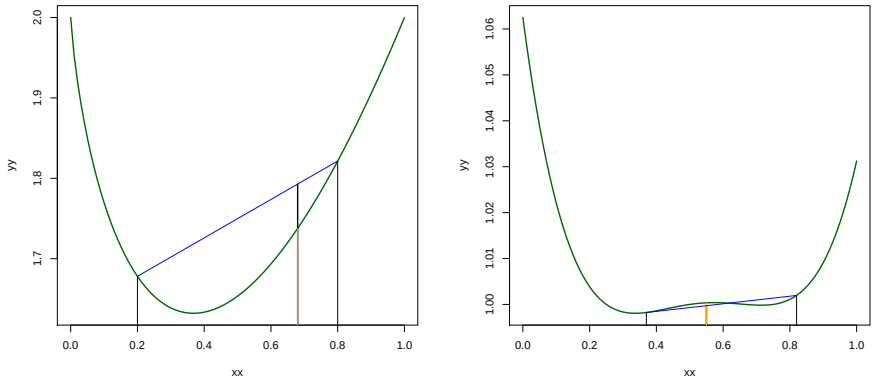
When we did the least squares procedure for the linear regression with normal errors, we were fortunate in that the derivative of the negative log likelihood function was a linear function in unknown parameters. So by setting these

derivatives equal to zero we had to solve a *linear* system of equations. This is fairly easy and every statistical software can accomplish it.

For logistic regression the derivatives of the likelihood function with respect to unknown parameters were not linear, so solving such nonlinear systems of equations were not as straightforward as the linear regression. Nevertheless, the negative log likelihood function in the case of the logistic regression is a *convex function*. In general a real valued function $f(\mathbf{x})$ is convex if for any two points $\mathbf{x}_1, \mathbf{x}_2$ and any $0 \leq \alpha \leq 1$ we have

$$f\big(\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2\big) \leq \alpha f(\mathbf{x}_1) + (1-\alpha)f(\mathbf{x}_2)$$

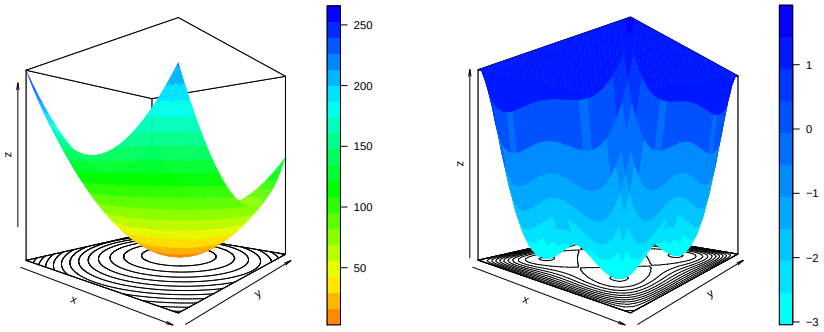
Look at the following examples:



The one on the left is a convex function, and when you take any two points on the graph and connect them by a line segment, then the graph falls below the line segment, always. On the other hand, the graph on the right is not a convex function. We have found two points on the graph where the section of the curve connecting them does not fall below the line segment.

The main significance of convex functions is that first, they have a unique minimum value. Second, from a computational point of view it is relatively easy to find the minimum of these functions.

For neural networks, neither the least squares problem for regression, nor the log likelihood function for the classification problems are convex in the unknown parameters W_i. As a result there may be many, many *local minimums*. Thus, finding the "minimum of minimums" is computationally very challenging, especially as the number of parameters increases.



In the examples above the diagram on the left depicts a convex function with contours showing a single minimum. The one on the right is nonconvex, and as the contours show, there are multiple minimums.

This type of optimization problem is called *global optimization*. In general, there is no satisfactory algorithm that solves global optimization problems in a reasonable time. What we have instead is a long list of heuristic methods that may be able to solve such optimization problems sometimes, while failing to do so in many other cases. A large part of research on neural networks, involves finding effective global optimization algorithms.

In addition to the daunting task of finding the optimal solution, one must consider the following issues for neural network training.

Scaling of input variables

Due to extremely nonlinear nature of neural networks, it is *absolutely essential* to scale the feature variables x_i. The slightest change in the data, for instance changing costs from US$ to Canadian $, will result in very different weights and very different neural networks. Scaling is important for all learning methods, but it is especially crucial for neural networks.

Regularizing the training process

Due to a large number of parameters to be estimated there is a serious possibility of overfitting. Remember that each W_i is an entire matrix, and there are as many matrices as there are layers. So we need to have a mechanism to shrink the number of parameters. Note that in multi-layer neural networks complexity is controlled both by the number of layers, and the number of hidden variables in each layer. We can reduce complexity by removing some of the links, that is by setting some of the parameters equal to zero.

Regularization provides another method of controlling the complexity. The idea is that we change the function we are minimizing (usually the log likelihood)

by adding the *size* of the parameters. So, for regression instead of minimizing the sum of squares we minimize:

$$\min_{W_1,\ldots,W_k}\left[\sum_{i=1}^{N}\Big(y_i - f(\mathbf{x}_i|W_1,\ldots,W_k\Big)^2 + \alpha\Big(\|W_1\|^2+\cdots+\|W_k\|^2\Big)\right]$$

The idea is that while we are driving the sum of squares towards smaller values, we may increase the size of W_i. So the regularized minimization attempts to drive the value of those parameters that are not important towards zero, or make them very small. The regularization term $\alpha\sum_j \|W_j\|^2$ can be added to cross entropy or any other loss function as well.

Regularization is used on ordinary linear regression and in logistic regression. However, for neural networks regularization is crucial and used to make sure only those parameters which are essential have significant values.

Regularization is analogous to the AIC and BIC and similar criteria, with the difference that the penalty for the complexity of the model is *baked in* the optimization process itself. In AIC and BIC, we optimize the likelihood and then add the penalty.

The notion of "size" or *norm* of parameters W_j is not fixed, and we may use many different types of norms. The most common ones are the *Euclidean* norm which is the square root of sum of squares of parameters, and the ℓ_1 norm, which is the sum of absolute values of parameters.

Early stopping of the optimization process

When minimizing the "loss function" (sum of squares for regression or the cross entropy for classification), the algorithm generates a sequence of estimates for the weights W_i. Let's call them $W^{(0)}, W^{(1)}, W^{[2]}, \ldots, W^{[m]}, \ldots$. Any minimizing algorithm will generate a sequence which drives the *training set loss function* smaller. Suppose we set aside part of the data as the test set, then for each new improvement $W^{[m]}$ the training loss function gets smaller all the time. However if, using the $W^{[m]}$ we also estimate the loss function on the test set, we will observe that this loss function initially gets smaller for a while, but at some point the test loss function starts increasing. At that point we can stop and declare $W^{[m]}$ as our estimate of the unknown parameters.

The logic of this operation is that there is really no point in finding the global optimum if we are dealing with noisy data. Instead we stop short of the optimal solution and allow near optimum estimates of W as a way of preventing overfitting.

5 Summary of Properties of Neural Networks

The neural network procedure is a very complicated approach to regression, and classification. As a result, it is not the most suitable approach for simpler problems we have been dealing with in this course. Where neural networks outshine other techniques is in very complex problems. Some examples are: self-driving cars, facial recognition software, natural language processing, playing

trivia games (like Jeopardy!), medical diagnosis, and grand master level chess playing programs. In addition, in such complex tasks, there is a lot of hand training along with trial and error, and help from experts.

Neural networks are suitable for online learning. Once a training data set is used to estimate the weights, the training data itself is no longer needed and can be discarded. If new data becomes available, we can use (in a Bayesian approach) the current weights as priors and retrain and update the weights.

The type of neural networks we have discussed are usually called *feed forward* neural networks. There are a number of variations.

Convolutional Networks: These are special kind of feed forward netwroks, where the matirx parameters W_i represent special linear transformations called *convolution*. THese transformations are sparse, that is many of W_{ij} are either zero, are the same value for many vlaues of ij. These transformations are useful in image processing, and sound processing applications.

Recurrent Networks: These are networks where outputs of some layers can bed fed *back* to the layer, or earlier layer. These kinds of networks are useful for time-series data, where vlaues at a given time t are dependent of the same values at an earlier time, say, $t-1, t-2, \ldots$.