Chapter 6
Neural Networks

Thomas R. Cook

6.1 Introduction

Neural networks have emerged in the last 10 years as a powerful and versatile set of machine learning tools. They have been deployed to produce art, write novels, read handwriting, translate languages, caption images, interpret MRIs, and many other tasks. In this chapter, we will introduce neural networks and their application to forecasting.

Though neural networks have recently become very popular, they are an old technology. Early work on neurons as computing units dates as far back as 1943 (McCulloch and Pitts) with early commercial applications arising in the late 1950s and early 1960s. The development of neural networks since then, however, has been rocky. In the late 1960s, work by Minsky and Papert showed that perceptrons (an elemental form of neural network) were incapable of emulating the exclusive-or (XOR) function. This led to a sharp decline in neural network research that lasted through the mid-1980s. From the mid-1980s through the end of the century, neural networks were a productive but niche area of computer science research. Starting in the mid-2000s however, neural networks have seen widespread adoption as a powerful machine learning method. This surge in popularity has been attributable

The views expressed are those of the author and do not necessarily reflect the views of the Federal Reserve Bank of Kansas City or the Federal Reserve System.

T. R. Cook
Federal Reserve Bank of Kansas City, Kansas City, MO, USA
e-mail: thomas.cook@kc.frb.org

P. Fuleky (ed.), *Macroeconomic Forecasting in the Era of Big Data*,
Advanced Studies in Theoretical and Applied Econometrics 52,
https://doi.org/10.1007/978-3-030-31150-6_6

largely to a confluence of factors: algorithmic developments that made neural networks useful for practical applications; advances in processing power that made model training feasible; the rise of "big data"; and a few high-profile successes in areas such as computer vision. At this point in time, neural networks have gained mainstream appeal in areas far beyond computer science such as bioinformatics, geology, medicine, chemistry, and others.

In economics and finance, neural networks have been used since the early 1990s, mostly in the context of microeconomics and finance. Much of the early work focused on bankruptcy prediction (see Altman, Marco, & Varetto, 1994; Odom & Sharda, 1990; Tam, 1991). Additional research using neural networks to predict creditworthiness was performed around this time and there is a growing appetite among banks to use artificial intelligence for credit underwriting. More recently, in the area of finance, neural networks have been successfully used for market forecasting (see Dixon, Klabjan, & Bang, 2017; Heaton, Polson, & Witte, 2016; Kristjanpoller & Minutolo, 2015; McNelis, 2005 for examples). There has been some limited use of neural networks in macroeconomic research (see Dijk, Teräsvirta, & Franses, 2002; Terasvirta & Anderson, 1992, as examples), but much of this research seems to have occurred prior to the major resurgence of neural networks in the 2010s.

Although there are already many capable tools in the econometric toolkit, neural networks are a worthy addition because of their versatility of use and because they are universal function approximators. This is established by the theorem of universal approximation, first put forth by Cybenko (1989) with similar findings offered by Hornik, Stinchcombe, and White (1989) and further generalized by Hornik (1991). In summary, the theorem states, that for any continuous function $f : \mathbb{R}^m \mapsto \mathbb{R}^n$, there exists a neural network with one hidden layer, G, that can approximate f to an arbitrary level of accuracy (i.e., $|f(\boldsymbol{x}) - G(\boldsymbol{x})| < \epsilon$ for any $\epsilon > 0$). While there are other algorithms that can be used as universal function approximators, neural networks require few assumptions (inductive biases), have a tendency to generalize well, and scale well to the size of the input space in ways that other methods do not.

Neural networks are often associated with "big data." The reason for this is that people often associate neural networks with complex modeling tasks that are difficult/impractical with other types of models. For example we often hear of neural networks in reference to computer vision, speech translation and drug discovery. Each of these types of task produces high-dimensional, complex outputs (and likely takes equivalently complex inputs). And, like any type of model, the amount of data needed to train a neural network typically scales to the dimensionality of its inputs/outputs. Take, for example, the Inception neural network model (Szegedy et al., 2015). This is an image classification model that learns a distribution over about 1000 categories that are then used to classify an image. The capabilities of this model are impressive, but to get the network to learn such a large distribution

of possible image categories, researchers made use of a dataset containing over one million labeled images.[1]

The remainder of this chapter will proceed as follows. In the remainder of this section the technical aspects of neural networks will be presented, focusing on the fully connected network as a point of reference. Section 6.2 will discuss neural network model design considerations. Sections 6.3 and 6.4 will introduce recurrent networks and encoder-decoder networks. Section 6.5 will provide an applied example in the form of unemployment forecasting.

6.1.1 Fully Connected Networks

A fully connected neural network, sometimes called a multi-layer perceptron, is among the most straightforward types of neural network models. It consists of several interconnected layers of neurons that translate inputs into a target output.

The fully connected neural network, and neural networks generally, are fundamentally comprised of neurons. A neuron is simply a linear combination of inputs, plus a constant term (called a *bias*), and transformed through a function (called an *activation function*),

$$f(\boldsymbol{x}\boldsymbol{\beta} + \alpha),$$

where $\boldsymbol{x}$ is an n-length vector of inputs, $\boldsymbol{\beta}$ is a corresponding vector of weights, and α is a scalar bias term.

Neurons are typically stacked into layers. Layers can have various forms, but the most simple is called a dense, or fully connected layer. For a layer with p neurons, let $\boldsymbol{B} = (\boldsymbol{\beta}_1 \dots \boldsymbol{\beta}_p)$ so that $\boldsymbol{B}$ has the dimensions $(n \times p)$, and let $\boldsymbol{\alpha} = (\alpha_1 \dots \alpha_p)$. The matrix $\boldsymbol{B}$ supplies weights for each term in the input vector to each of the p neurons while $\boldsymbol{\alpha}$ supplies the bias for each neuron. Given an n-length input vector $\boldsymbol{x}$, we can write a dense layer with p neurons as,

$$\begin{aligned} g(\boldsymbol{x}) &= f(\boldsymbol{x}\boldsymbol{B} + \boldsymbol{\alpha}) \\ &= \begin{bmatrix} f(\boldsymbol{x}\boldsymbol{\beta}_1 + \boldsymbol{\alpha}_1) \\ f(\boldsymbol{x}\boldsymbol{\beta}_2 + \boldsymbol{\alpha}_2) \\ \vdots \\ f(\boldsymbol{x}\boldsymbol{\beta}_p + \boldsymbol{\alpha}_p) \end{bmatrix}^T . \end{aligned} \tag{6.1}$$

[1] Specifically, a subset of the imagenet dataset. See Russakovsky et al. (2015).

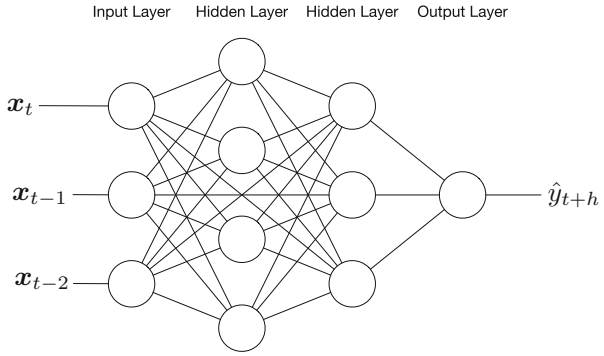

Fig. 6.1 Diagram of a fully connected network as might be constructed for a forecasting task. The network takes in several lags of the input vector $\boldsymbol{x}$ and returns an estimate of the target at the desired forecast horizon $t + h$. This network has two hidden layers with three and four neurons respectively. The output layer is a single neuron, returning a one-dimensional output

As should be clear from this expression, each element in the input vector bears some influence on (or connection to) each of the p neurons, which is why we call this a fully connected layer.

The layer described in Eq. (6.1) can also accept higher-order input such as an $m \times n$ matrix of several observations, $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_m)'$, in which case,

$$
\begin{aligned}
g(\boldsymbol{X}) &= f(\boldsymbol{X}\boldsymbol{B} + \boldsymbol{\alpha}) \\
&= \begin{bmatrix} f(\boldsymbol{x}_1\boldsymbol{\beta}_1 + \boldsymbol{\alpha}_1) & \dots & f(\boldsymbol{x}_1\boldsymbol{\beta}_p + \boldsymbol{\alpha}_p) \\ \vdots & \ddots & \vdots \\ f(\boldsymbol{x}_m\boldsymbol{\beta}_1 + \boldsymbol{\alpha}_1) & \dots & f(\boldsymbol{x}_m\boldsymbol{\beta}_p + \boldsymbol{\alpha}_p) \end{bmatrix}.
\end{aligned}
$$

A fully connected, feed forward network (Fig. 6.1), with K layers is formed by connecting dense layers together so that the output of the preceding layer serves as the input for the current layer. Let k index a given layer, then the output of the k-th layer is

$$
\begin{aligned}
g_k(\boldsymbol{X}) &= f_k(g_{k-1}(\boldsymbol{X})\boldsymbol{B}_k + \boldsymbol{\alpha}_k) \\
g_0(\boldsymbol{X}) &= \boldsymbol{X},
\end{aligned}
$$

The parameters of the network are all elements $\boldsymbol{B}_k, \boldsymbol{\alpha}_k$ for $k \in (1 \dots K)$. For simplicity, denote these parameters by $\boldsymbol{\theta}$, where $\boldsymbol{\theta}_k = (\boldsymbol{B}_k, \boldsymbol{\alpha}_k)$. Further, for simplicity, denote the final output of the network $G(\boldsymbol{X}; \boldsymbol{\theta}) = g_K(\boldsymbol{X})$.

In the context of a supervised learning problem (such as a forecasting problem), we have a known target, $\boldsymbol{y}$, estimated as $\hat{\boldsymbol{y}} = G(\boldsymbol{X}; \boldsymbol{\theta})$, and we can define a loss function, $L(\boldsymbol{y}; \boldsymbol{\theta})$ to summarize the discrepancy between our estimate and target. To estimate the model, we simply find $\boldsymbol{\theta}$ that minimizes $L(\boldsymbol{y}; \boldsymbol{\theta})$. The estimation procedure will be discussed in greater detail below.

6.1.2 Estimation

To fit a neural network, we follow a modified variation of gradient descent. Gradient descent is an iterative procedure. For each of $\boldsymbol{\theta}_k \in \boldsymbol{\theta}$, we calculate the gradient of the loss, $\nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{y}; \boldsymbol{\theta})$. The negative gradient tells us the direction of steepest descent and the direction in which to adjust $\boldsymbol{\theta}_k$ to reduce $L(\boldsymbol{y}; \boldsymbol{\theta})$. After calculating $\nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{y}; \boldsymbol{\theta})$, we perform an update,

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_k - \gamma \nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{y}; \boldsymbol{\theta}).$$

where γ controls the size of the update and is sometimes referred to as the learning rate. After updates are computed for all $\boldsymbol{\theta}_k \in \boldsymbol{\theta}$, $L(\boldsymbol{y}; \boldsymbol{\theta})$ is recomputed. These steps are repeated until a stopping rule has been reached (e.g., the $L(\boldsymbol{y}; \boldsymbol{\theta})$ falls below a preset threshold).

The computation of $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{y}; \boldsymbol{\theta})$ is costly and increases with the size of $\boldsymbol{X}$ and $\boldsymbol{y}$. To reduce this cost, and the overall computation time needed, we turn to stochastic gradient descent (SGD). This is a modification of gradient descent in which updates to $\boldsymbol{\theta}$ are calculated using only one observation at a time. For each iteration of the procedure, one observation, $\{\boldsymbol{x}, y\}$, is chosen, then updates to $\boldsymbol{\theta}$ are calculated and applied as described for gradient descent, and a new observation is chosen for use in the next iteration of the procedure. By using SGD, we reduce the time needed for each iteration of the optimization procedure, but increase the expected number of iterations needed to reach performance equivalent to gradient descent.

In many cases the speed of optimization can be further boosted through Mini-batch SGD. This is a modification of SGD in which updates are calculated using several observations at a time. Mini-batch SGD should generally require fewer iterations than SGD, but computing the updates for each iteration will be more computationally costly. The per-iteration cost of calculating updates to $\boldsymbol{\theta}$, however, should be lower than for gradient descent. Mini-batch SGD is by far the most popular procedure for fitting a neural network.

Fitting a neural network is a non-convex optimization problem. It is possible and quite easy for a mini-batch SGD procedure to get stuck at local minima or saddle points (Dauphin et al., 2014). To overcome this, a number of modified optimization algorithms have been proposed. These include RMSprop and adaGrad (Duchi, Hazan, & Singer, 2011). Generally, these modifications employ adaptive learning rates and/or notions of momentum to encourage the optimization algorithm to choose appropriate learning rates and avoid suboptimal local minima (see Ruder, 2016 for a review).

More recently, Adam (Kingma & Ba, 2014) has emerged as a popular variation of gradient descent and as argued in Ruder (2016), "Adam may be the best overall choice [of optimizer]." Adam modifies vanilla gradient descent by scaling the learning rates of individual parameters using the estimated first and second moments of the gradient. Let $\boldsymbol{\theta}_k$ be the estimated value of $\boldsymbol{\theta}_k$ at the current step in the Adam optimization procedure, then we can estimate the first and second moments of the

gradient of $\boldsymbol{\theta}_k$ via exponential moving average,

$$\begin{aligned}
\boldsymbol{\mu}_t &= \boldsymbol{\mu}_{t-1}\gamma_{\boldsymbol{\mu}} + (1-\gamma_{\boldsymbol{\mu}})(\nabla_{\boldsymbol{\theta}_k} L) \\
\boldsymbol{v}_t &= \boldsymbol{v}_{t-1}\gamma_{\boldsymbol{v}} + (1-\gamma_{\boldsymbol{v}})(\nabla_{\boldsymbol{\theta}_k} L)^2 \\
\boldsymbol{\mu}_0 &= \mathbf{0} \\
\boldsymbol{v}_0 &= \mathbf{0},
\end{aligned}$$

where arguments to the loss function are suppressed for readability. Both $\gamma_{\boldsymbol{\mu}}$ and $\gamma_{\boldsymbol{v}}$ are hyper parameters that control the pace at which $\boldsymbol{\mu}$ and $\boldsymbol{v}$ change. With $\boldsymbol{\mu}$ and $\boldsymbol{v}$, we can assemble an approximate signal to noise ratio of the gradient and use that ratio as the basis for the update step:

$$\begin{aligned}
\hat{\boldsymbol{\mu}}_t &= \tfrac{\boldsymbol{\mu}_t}{1-(\gamma_{\boldsymbol{\mu}})^t} \\
\hat{\boldsymbol{v}}_t &= \tfrac{\boldsymbol{v}_t}{1-(\gamma_{\boldsymbol{v}})^t} \\
\boldsymbol{\theta}_k &\leftarrow \boldsymbol{\theta}_k + \gamma \tfrac{\hat{\boldsymbol{\mu}}_t}{\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon},
\end{aligned}$$

where $\hat{\boldsymbol{\mu}}_t$ and $\hat{\boldsymbol{v}}_t$ correct for the bias induced by the initialization of $\boldsymbol{\mu}$ and $\boldsymbol{v}$ to zero, γ is the maximum step-size for any iteration of the procedure (the learning rate), and where division should be understood in this context as element-wise division. By constructing the update from a signal to noise ratio the path of gradient descent becomes smoother. That is, the algorithm is encouraged to take large step sizes along dimensions of the gradient that are steep and relatively stable; it is cautioned to take small step sizes along dimensions of the gradient that are shallow or relatively volatile. As a result, parameter updates are less volatile. The authors of the algorithm suggest values of $\gamma_{\boldsymbol{\mu}} = 0.9999$, $\gamma_{\boldsymbol{v}} = 0.9$ and $\epsilon = 1e-8$.

Gradient Estimation

Each of the optimization routines described in the previous section rely upon the computation of $\nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{y}; \boldsymbol{\theta})$ for all $\boldsymbol{\theta}_k \in \boldsymbol{\theta}$. This is achieved through the backpropagation algorithm (Rumelhart, Hinton, & Williams, 1986), which is a generalization of the chain rule from calculus. Consider, for example, a network $G(\boldsymbol{X}; \boldsymbol{\theta})$ with an accompanying loss $L(\boldsymbol{y}; \boldsymbol{\theta}) = \frac{1}{2m}\|G(\boldsymbol{X}; \boldsymbol{\theta}) - \boldsymbol{y}\|_2^2 = \frac{1}{2m}\sum_i^m (G(\boldsymbol{X}; \boldsymbol{\theta})_i - y_i)^2$. Then

$$\nabla_{G(\boldsymbol{X};\boldsymbol{\theta})} L(\boldsymbol{y}; \boldsymbol{\theta}) = \frac{1}{m}(G(\boldsymbol{X}; \boldsymbol{\theta}) - \boldsymbol{y}).$$

To derive $\nabla_{\boldsymbol{\theta}_K} L(\boldsymbol{y}; \boldsymbol{\theta})$, we simply apply chain rule to the above equation, suppressing arguments to G and g for notational simplicity:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}_K} L(\boldsymbol{y}; \boldsymbol{\theta}) &= \frac{\partial G}{\partial \boldsymbol{\theta}_K}^T \nabla_G L(\boldsymbol{y}) \\
&= \begin{bmatrix} (f'(g_{k-1}\boldsymbol{B}_k + \boldsymbol{\alpha}_k) g_{k-1})^T \frac{1}{m}(G - \boldsymbol{y}) \\ (f'(g_{k-1}\boldsymbol{B}_k + \boldsymbol{\alpha}_k))^T \frac{1}{m}(G - \boldsymbol{y}), \end{bmatrix}^T
\end{aligned}$$

where $\frac{\partial G}{\boldsymbol{\theta}_K}$ is a generalized form of a Jacobian matrix, capable of representing higher-order tensors, and f' indicates the first derivative of f with respect to its argument.

Collecting right-hand-side gradients into Jacobian matrices, we can extend the application of backpropagation to calculate the gradient of the loss with respect to any of the set of parameters $\boldsymbol{\theta}_k$:

$$\nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{y}; \boldsymbol{\theta}) = \frac{\partial L(\boldsymbol{y}; \boldsymbol{\theta})}{\partial G_k} \frac{\partial g_K}{\partial g_{K-1}} \frac{\partial g_{K-1}}{\partial g_{K-2}} \cdots \frac{\partial g_{k+1}}{g_k} \frac{\partial g_k}{\partial \boldsymbol{\theta}_k}.$$

6.1.3 Example: XOR Network

To illustrate the concepts discussed thus far, we will review a simple, well known network that illustrates the construction of a neural network from end to end. This network is known as the XOR network (Minsky & Papert, 1969; Rumelhart, Hinton, & Williams, 1985). It was an important hurdle in the development of neural networks.

Consider a dataset with labels $\boldsymbol{y}$ whose values depend on features, $\boldsymbol{X}$:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \boldsymbol{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The label of any given observation follows the logic of the exclusive-or operation – $y_i = 1$ only if $\boldsymbol{x}_i$ contains *exactly* one non-zero element.

We can build a fully connected network, $G(\boldsymbol{X}; \boldsymbol{\theta})$ that perfectly represents this relationship using only two layers and three neurons (two in the first layer and one in the last layer):

$$g_1(\boldsymbol{X}) = f(\boldsymbol{X}\boldsymbol{B}_1 + \boldsymbol{\alpha}_1) \tag{6.2}$$

$$g_2(\boldsymbol{X}) = f(g_1(\boldsymbol{X})\boldsymbol{B}_2 + \boldsymbol{\alpha}_2) \tag{6.3}$$

$$\boldsymbol{B}_1 = \begin{bmatrix} \beta_{11} & \beta_{12} \\ \beta_{13} & \beta_{14} \end{bmatrix} \quad \boldsymbol{B}_2 = \begin{bmatrix} \beta_{21} \\ \beta_{22} \end{bmatrix}$$

$$f(a) = \frac{1}{1 + e^{-a}}.$$

The structure of this network is illustrated in Fig. 6.2.

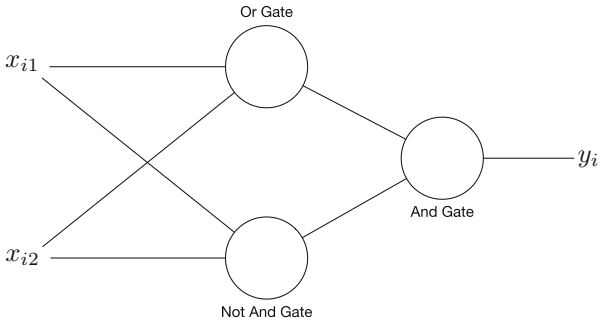

Fig. 6.2 The XOR network. This network is sufficiently simple that each neuron can be labeled according to the logical function it performs

Because this is a classification problem, we will measure loss by log-loss (i.e., negative log-likelihood)[2]:

$$\begin{aligned} L(\boldsymbol{y};\boldsymbol{\theta}) &= -\sum_{i}^{m} y_i \, log(G(\boldsymbol{X};\boldsymbol{\theta})_i) + (1-y_i) \, log(1-G(\boldsymbol{X};\boldsymbol{\theta})_i) \\ &= -\left(\boldsymbol{y} \, log(G(\boldsymbol{X};\boldsymbol{\theta})) + (1-\boldsymbol{y}) log(1-G(\boldsymbol{X};\boldsymbol{\theta}))\right). \end{aligned} \tag{6.4}$$

Calculation of gradients for the final layer yields

$$\frac{\partial L(\boldsymbol{y};\boldsymbol{\theta})}{\partial G(\boldsymbol{X};\boldsymbol{\theta})} = \frac{\boldsymbol{y} - G(\boldsymbol{X};\boldsymbol{\theta})}{(G(\boldsymbol{X};\boldsymbol{\theta})-1)G(\boldsymbol{X};\boldsymbol{\theta})}$$

and application of chain rule provides gradients for $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$,

$$\begin{aligned} \nabla_{\boldsymbol{\theta}_2} L(\boldsymbol{y};\boldsymbol{\theta}) &= \begin{bmatrix} \frac{\partial L(\boldsymbol{y};\boldsymbol{\theta})}{\partial G(\boldsymbol{X};\boldsymbol{\theta})} \frac{\partial G(\boldsymbol{X};\boldsymbol{\theta})}{\partial \boldsymbol{B}_2} \\ \frac{\partial L(\boldsymbol{y};\boldsymbol{\theta})}{\partial G(\boldsymbol{X};\boldsymbol{\theta})} \frac{\partial G(\boldsymbol{X};\boldsymbol{\theta})}{\partial \boldsymbol{\alpha}_2} \end{bmatrix} \\ &= \begin{bmatrix} g_1(X)^T(\boldsymbol{y} - G(\boldsymbol{X};\boldsymbol{\theta})) \\ (\boldsymbol{y} - G(\boldsymbol{X};\boldsymbol{\theta})) \end{bmatrix} \end{aligned} \tag{6.5}$$

$$\begin{aligned} \nabla_{\boldsymbol{\theta}_1} L(\boldsymbol{y};\boldsymbol{\theta}) &= \begin{bmatrix} \frac{\partial L(\boldsymbol{y};\boldsymbol{\theta})}{\partial G(\boldsymbol{X};\boldsymbol{\theta})} \frac{\partial G(\boldsymbol{X};\boldsymbol{\theta})}{\partial g_1(\boldsymbol{X})} \frac{\partial g_1(\boldsymbol{X})}{\partial \boldsymbol{B}_1} \\ \frac{\partial L(\boldsymbol{y};\boldsymbol{\theta})}{\partial G(\boldsymbol{X};\boldsymbol{\theta})} \frac{\partial G(\boldsymbol{X};\boldsymbol{\theta})}{\partial g_1(\boldsymbol{X})} \frac{\partial g_1(\boldsymbol{X})}{\partial \boldsymbol{\alpha}_1} \end{bmatrix} \\ &= \begin{bmatrix} X^T(\boldsymbol{y} - G(\boldsymbol{X};\boldsymbol{\theta}))\boldsymbol{B}_2^T \odot f'(\boldsymbol{X}\boldsymbol{B}_1 + \boldsymbol{\alpha}_1) \\ (\boldsymbol{y} - G(\boldsymbol{X};\boldsymbol{\theta}))\boldsymbol{B}_2^T \odot f'(\boldsymbol{X}\boldsymbol{B}_1 + \boldsymbol{\alpha}_1), \end{bmatrix} \end{aligned} \tag{6.6}$$

[2]We use log-loss because it is the convention (in both the machine learning and statistical literature) for this type of categorization problem. Other loss functions, including mean squared error would likely work as well.

Algorithm 1 Gradient descent to fit the XOR network

Data: $\boldsymbol{X}$, $\boldsymbol{y}$
Input: γ, stop_rule
Initialize $\boldsymbol{\theta} = B_1, B_2, \alpha_1, \alpha_2$ to random values
while stop_rule not met **do**
  Forward pass:
  calculate $\hat{\boldsymbol{y}} = G(\boldsymbol{X}; \boldsymbol{\theta})$ as in (6.2)–(6.3)
  calculate $L(\boldsymbol{y}; \boldsymbol{\theta})$ by log-loss$(\hat{\boldsymbol{y}}, \boldsymbol{y})$ as in (6.4)
  Backward pass:
  calculate $\nabla_{\boldsymbol{\theta}_2} = \nabla_{\boldsymbol{\theta}_2} L(\boldsymbol{y}; \boldsymbol{\theta})$ as in (6.5)
  calculate $\nabla_{\boldsymbol{\theta}_1} = \nabla_{\boldsymbol{\theta}_1} L(\boldsymbol{y}; \boldsymbol{\theta})$ as in (6.6)
  Update:
  $\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_1 - \gamma \nabla_{\boldsymbol{\theta}_1}$
  $\boldsymbol{\theta}_2 \leftarrow \boldsymbol{\theta}_2 - \gamma \nabla_{\boldsymbol{\theta}_2}$
end while

where f' indicates the first derivative of the activation function (i.e., $f'(a) = f(a) \odot (1 - f(a))$). To fit (or *train*) this model, we minimize $L(\boldsymbol{y}; \boldsymbol{\theta})$ via vanilla gradient descent as described in Algorithm 1.

Figure 6.3 illustrates the results of this training process. It shows that, as the number of training iterations increases, the model output predicts the correct classification of each element in $\boldsymbol{y}$.

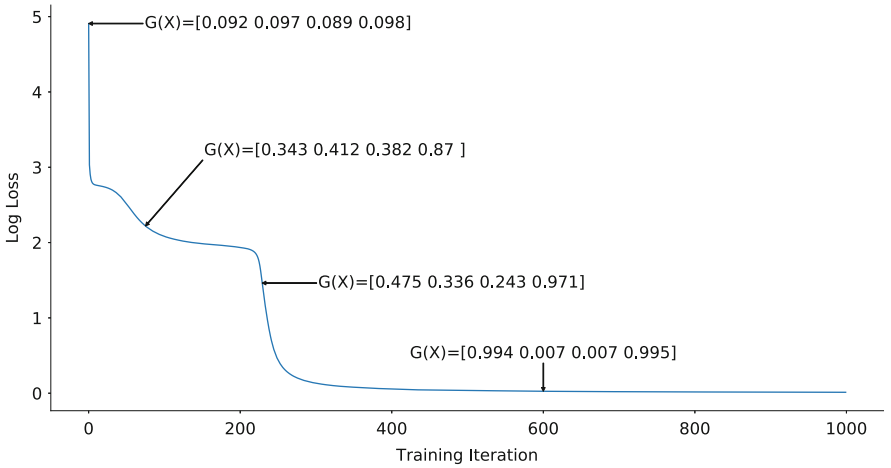

Fig. 6.3 The path of the loss function over the training process. Annotations indicate the model prediction at various points during the training process. The model target is $\boldsymbol{y} = [1, 0, 0, 1]$

6.2 Design Considerations

The XOR neural network is an example of a network that is very deliberately designed in the way that one might design a circuit or economic model. Each neuron in the network carries out a specific and identifiable task. The neurons in g_1 learn to emulate OR and NOT AND gates on the input, while the g_2 neuron learns to emulate an AND gate on the output of g_1.[3] It is somewhat unusual to design neural network models this explicitly. Moreover explicitly designing a neural network this way obviates one of the central advantages of neural network models: a network model with sufficient number of neurons and an appropriate amount of training data can learn to approximate any function without being *ex ante* and explicitly designed to approximate that function.

The typical process for designing a neural network occurs without guidance from an explicit, substantive theory. Instead, the process of designing a neural network is usually functional in nature. As such, when designing a neural network, we are usually left with many design decisions or, alternatively stated, a large space of hyper parameters to explore. Finding the optimal set of hyper parameters needed to make a neural network work effectively for a given problem is one of the biggest challenges to building a successful model. Efficient, automatic processes to optimize model hyper parameters is an active area of research. In this section, we will discuss some of the common design decisions that we must make when designing a neural network model.

6.2.1 Activation Functions

Activation functions are what enable neural networks to approximate non-linear functions. Any differentiable function can be used as an activation function. Moreover, some non-differentiable functions can also be used, as long as there are relatively few points of non-differentiability. Activation functions also tend to be monotonic, though this is not required. The influence of an activation function on model performance is inherently related to the structure of the network model, the method of weight initialization, and idiosyncrasies in the data.

For model training to be successful, the codomain of the final layer activation function must admit the range of possible target values, $\boldsymbol{y}$. For many forecasting tasks, then, the most appropriate final layer activation function is the identify function, $f(a) = a$.

Generally for hidden layers, we want to choose activation functions that return the value of the input (i.e., approximate identity) when the value of the input is near

[3] See Bland (1998), Rumelhart et al. (1985) for further discussion.

Table 6.1 Common activation functions

Sigmoid	$f(a) = \frac{1}{1+e^a}$
ReLU (Nair & Hinton, 2010)	$f(a) = \begin{cases} a & a > 0 \\ 0 & a \leq 0 \end{cases}$
Leaky ReLU (Maas, Hannun, & Ng, 2013)	$f(a) = \begin{cases} a & a > 0 \\ a\,\alpha & a \leq 0 \end{cases}$ & $0 < \alpha \ll 1$
Hyperbolic tangent (Karlik & Olgac, 2011)	$f(a) = \frac{e^a - 1}{e^a + 1}$
Swish	$f(x\beta) = x\,\frac{1}{1+e^{x\beta}}$

zero. This is a desirable property because it removes complications with weight initialization (Sussillo & Abbott, 2014).

Additionally, we want to choose activations that are unbounded (in at least one direction). This helps to prevent neuron saturation (which occurs when gradients approach zero). In turn, this helps prevent the problem of vanishing gradients (Bengio, Simard, & Frasconi, 1994; Glorot & Bengio, 2010) in which early network layers update very slowly. The severity of this problem scales to the depth of the network (assuming the same, bounded, activation function is used for every layer in the network). In the extreme, this can cause adjustments to model weights to effectively stop very early in the training process. It is largely because of the vanishing gradient problem that sigmoid and hyperbolic tangent (see Table 6.1 below) have fallen out of favor for general use.

Table 6.1 provides a list of some common activation functions. Sigmoid and Hyperbolic Tangent activation functions were commonly used in the early development of neural networks, but in recent years the Rectified Linear Unit (ReLU) has become the most popular choice for activation function. Other activation functions such as Swish have emerged more recently and, while they have not found the same widespread adoption, recent research suggests that they may perform better than ReLU in general settings (Ramachandran, Zoph, & Le, 2017).

6.2.2 Model Shape

Cybenko (1989) provides the universal approximation theorem, which establishes that a feed forward network with a single hidden layer can approximate any continuous function. Hornik et al. (1989) provides a related and contemporaneous result. As a matter of theory then, no network should need to be larger than two layers (an output layer and a hidden layer) to predict a target from a given input. This, however, requires that each layer (especially the hidden layer) contain sufficient neurons to approximate the desired function. Indeed, the hidden layer in a two layer network may require as many neurons as the number of training samples,

N, to effectively approximate a desired function (Huang, 2003; Huang & Babri, 1997).

Additional hidden layers can drastically reduce the parameter space without impinging the expressiveness of the model (Hastad, 1986; Telgarsky, 2016). For example, results from Huang (2003) show that a three layer network with m output neurons can exactly fit the target data when the first layer contains $\sqrt{(m+2)N} + 2\sqrt{N/(m+2)}$ neurons and the second layer contains $m\sqrt{N/(m+2)}$ neurons. Combined, this three layer network has $2\sqrt{(m+2)N} \ll N$ neurons. This result establishes the size of a three layer network that is needed to considerably *overfit* the training data. As such, it establishes an upper bound to the parameterization of a three layer network. Note that increasing the number of layers does not serve to improve the performance of the model *per se*, but rather lowers the number of neurons required to fit the model. Further, difficulties with weight initialization and vanishing/exploding gradients increase with depth.

There are few well-established rules for determining *ex ante* how many layers a network should have or how many neurons should go in each layer. Broadly speaking, over-parameterization of a network will not impact the model's accuracy as long as an appropriate training methodology is used (Zou, Cao, Zhou, & Gu, 2018). But over-parameterization will increase computational costs and it may increase the likelihood that training becomes prematurely stuck in a suboptimal minima. Under-parameterization, on the other hand will limit the expressiveness of a network and yield an under-performing model. The most obvious, heuristic strategy to determining the appropriate size and shape of a network is to begin with a small network and successively adjust its depth (the number of layers) or width (the number of neurons in each layer) in small increments to improve training accuracy.

6.2.3 Weight Initialization

While gradient descent and backpropagation provide a method to optimize parameters in a neural network, we must set the initial values for the parameters. Caution must be taken when initializing weights as bad initializations can cause gradients to saturate (i.e., reduce to small values near zero) prematurely. When this happens, the associated neuron will produce the same output regardless of variation in its input. These neurons are called "dead neurons." In practice, a few dead neurons will not influence the accuracy of a model if the network layer is large. If however, most or all of the neurons in a layer die, then gradient descent will lose the ability to update earlier layers and the network will become effectively unresponsive to its input. Poor weight initialization can also cause volatility in the training process, and may prevent gradient descent from finding an ideal set of parameters.

One might suspect that i.i.d. random draws from a distribution would be sufficient to initialize all weights in a network. For example, we might initialize all weights in a network with a random draw from a standard normal distribution. Indeed this was a common approach with early neural networks. For small networks, this will work.

However for deep neural networks, this is inadequate and will tend to encourage the problems described in the preceding paragraph. Indeed, it is the inadequacy of random initialization that led researchers to conclude that deep neural networks performed worse than simple ones (Bengio, Lamblin, Popovici, & Larochelle, 2007).

Early breakthroughs in weight initialization came in 2006 and 2007 (Bengio et al., 2007; Hinton, Osindero, & Teh, 2006) in the form of network pre-training. This is a method where the network is built iteratively, one layer at a time. We begin with a single-layer network with weights initialized to random values. Then train that single-layer network. When training is complete, recover the weights for the layer as the initialization weights for that layer. Then add an additional layer and repeat the process until the network is complete. This process is still occasionally employed, but it is time-consuming for large networks.

Instead, consider the method put forth in Glorot and Bengio (2010). This paper observes that the tendency for gradients to vanish (or explode) is somewhat controlled by keeping variances consistent across layers. To avoid vanishing gradients, we want to initialize weights so that the variance of the output of each layer is roughly consistent with the variance of the output of the preceding layer (and ultimately the variance of the input). To achieve this, the authors suggest initializing all weights $\beta_i \in \boldsymbol{B}_k$ as,

$$\beta_i \sim N\left(0, \frac{2}{p_k + p_{k-1}}\right),$$

where p_k is the number of output neurons for layer k, and p_{k-1} is the number of output neurons from the preceding layer (i.e., the number of input neurons to the current layer). This approach has been widely adopted in the neural network community as it tends to produce good results.

6.2.4 Regularization

To build models that generalize well, it is necessary prevent overfitting. This can partially be accomplished by adopting a training regime that uses out of sample data to determine when gradient descent should stop. We can further prevent overfitting by limiting the complexity of a neural network. To do this, we engage in the process of regularization. There are a number of approaches to regularization; we will discuss two of the more commonly used forms: weight decay and dropout.

Weight decay, or alternatively L2 regularization, applies a loss penalty to each weight in a layer according to its L2 norm: $\frac{\lambda_k}{2}\|\boldsymbol{B}_k\|_2^2$. The hyper parameter λ_k controls the magnitude of the penalty. When weight decay is employed, it is typically applied identically to each layer. Consider a network G containing no bias terms, so that all of the network weights can be represented in a single vector $\boldsymbol{\theta} = (\text{vec}(\boldsymbol{B}_1) \ldots \text{vec}(\boldsymbol{B}_K))$, and where, for each layer $\lambda_k = \lambda$. Then we can rewrite

the model's objective function[4] J to incorporate the loss function, L along with the penalty as

$$J(\boldsymbol{\theta}) = L(\boldsymbol{y}; \boldsymbol{\theta}) + \frac{\lambda_k}{2} \sum_{k}^{K} \|\boldsymbol{B}_k\|_2^2$$
$$= L(\boldsymbol{y}; \boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2$$

with a gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{y}; \boldsymbol{\theta}) + \lambda \boldsymbol{\theta}.$$

Through some rearrangement of terms in the (vanilla gradient descent) update step, it becomes clear why this type of regularization is called weight decay:

$$\boldsymbol{\theta} \leftarrow (1 - \gamma\lambda)\boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

That is, by applying an L2 regularization penalty, we are imposing a reduction in $\boldsymbol{\theta}$ by a factor of $(1 - \gamma\lambda)$ at each iteration of the training process. For a given non-zero $\beta_i \in \boldsymbol{\theta}$, if during the training process $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{y}; \boldsymbol{\theta})$ does not encourage movement in the direction of β_i, then it will decay towards zero. In the aggregate, then, the application of weight decay will produce parameter estimates that emphasize the parameters that represent significant contribution to the reduction of the objective function (Goodfellow, Bengio, & Courville, 2016). At the same time, the application of weight decay discourages the model fitting procedure from overreacting to non-systematic variation in the model target (Krogh & Hertz, 1992). Note that for weight decay to work properly, λ must be set so that $\gamma\lambda < 1$

Outside of weight decay, a common approach to regularization is a process called dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). Consider a network $G(\boldsymbol{X})$ with a layer k and its preceding layer $k-1$ with p neurons. The application of dropout to layer k draws a p-length vector $\boldsymbol{r} \sim Bernoulli(\pi)$ at each step in the training process. It then modifies the input to layer k,

$$g_k(\boldsymbol{X}) = f(r \odot g_{k-1}(\boldsymbol{x})\boldsymbol{B}_k + \alpha_k).$$

This modification is only applied during the training process. After the model has been trained,

$$g_k(\boldsymbol{X}) = f(g_{k-1}(\boldsymbol{x})\boldsymbol{B}_k + \alpha_k).$$

[4]In this setting, the goal of the model fitting process would be to minimize this objective function.

The application of r to the output of layer $k-1$ effectively turns some of the neurons in the network off. The dropout procedure accomplishes two things.

First, dropout limits overfitting by breaking heavily correlated updates of connected neurons (co-adaption). Updates become heavily correlated when one neuron updates to compensate for the output of a connected neuron. This is undesirable as it tends to correspond to fitting idiosyncrasies in the data and thus overfitting (see discussion in Srivastava et al., 2014). Dropout introduces instability in the inputs to a layer, thus breaking the ability of a neuron in that layer to become overly dependent on the output from any given neuron in the preceding layer. This breaks co-adaptation and thus reduces the propensity for overfitting.

Second, dropout allows us to approximate many models at once. Since dropout will set the output of a random number of neurons to zero, it achieves the effect of removing those neurons (briefly) from the network. With the neurons removed, we can consider the network to be an example of a sparse network sampled from G. Srivastava et al. (2014) argue that this interpretation suggests that training a network with dropout provides estimates that approximate a model averaging over many sparse networks. Gal and Ghahramani (2016) extend this view to argue that models with dropout can be interpreted as Bayesian models. Specifically, they argue that dropout in a deep neural network is equivalent to variational inference with a Gaussian process. By applying dropout during inference as well as estimation, we can generate uncertainty estimates via bootstrap simulation.

6.2.5 Data Preprocessing

Neural network models do not require strong assumptions about the data generating process. As a matter of practice however, neural network models are quite sensitive to several properties of the data.

When feeding a model with more than one feature, it is important that the features are at roughly similar scales (to within about an order of magnitude). In theory, a neural network should be able to adjust to inputs of differing scales. But in the initial iterations of training, larger-scaled inputs will dominate gradients and thus parameter adjustments. This can lead to premature saturation of the neurons or very slow model convergence. Pre-scaling the model inputs to have similar scales will alleviate this problem. Typical approaches include scaling inputs to standard normal distribution (normalization), and scaling inputs to the interval $(0, 1]$ through the following affine transformation:

$$x^* = \frac{x - min(\boldsymbol{x})}{max(\boldsymbol{x}) - min(\boldsymbol{x})}.$$

Beyond scaling the data, it is important to consider its bounds. Neural networks excel at generating predictions that generally lie within the boundaries of the training data. Out-of-bounds predictions are subject to more error. In some cases,

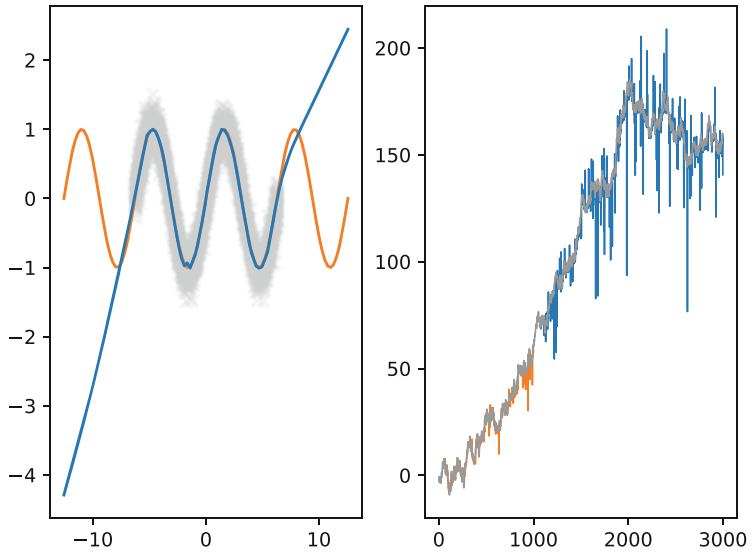

Fig. 6.4 Left: neural network predictions of a sine wave. The training data (in gray) is randomly distributed about the sin curve on the interval $[-2\pi, 2\pi]$. Trained model estimates (blue) are shown for the interval $[-4\pi, 4\pi]$. The sin function (orange) is provided for reference. Right: neural network predictions for a random walk with drift. Training data consists of the first 1000 observations of the walk. In-bounds model predictions (orange) are shown for the first 1000 observations. Out-of-bounds model predictions (blue) are shown for observations beyond observation 1040

this error will be severe. See, for example the left panel in Fig. 6.4. A neural network was given scaler training values $x \in [-2\pi, 2\pi]$, and trained to predict corresponding values of y distributed about $sin(x)$. After training, the model faithfully reproduces $sin(x)$ within the interval represented by the training data. Predictions outside of this interval (i.e., out-of-bounds) do not conform to a sine wave and resemble a linear extrapolation from the model predictions of the nearest training data.

In other cases, out-of-bounds predictions may present errors that are less severe. The right panel of Fig. 6.4 shows neural network predictions of a random walk with drift. A fully-connected network was trained on the first 1000 observations. For each observation y_t, the network was provided with prior observations, $(y_{t-1}, y_{t-2} \ldots y_{t-30})$, as input. The figure shows predictions on test-data (i.e., data not used for model training, but generated from the same random walk process). The network can fit in-bounds observations (the first 1000 observations) quite closely. Predictions for out-bounds predictions follow the general trend of the random walk, but are subject to considerably more error. The size of the error tends to grow with distance from the training data.

For economists, the issues posed by out-of-bounds predictions will most likely create complications in dealing with non-stationary data. To reduce the potential

for errors in model prediction, researchers can transform data into a mean-reverting (or as nearly mean-reverting as possible) form using standard econometric tools. An alternative technique that has seen success in recent years is to employ wavelet networks for forecasting with non-stationary data. Wavelet networks refer to networks that operate on data that has been preprocessed through a wavelet decomposition (see Jothimani, Yadav, & Shankar, 2015; Lineesh, Minu, & John, 2010; Minu, Lineesh, & John, 2010, as examples).[5]

6.3 RNNs and LSTM

For purposes of forecasting we are almost always making use of time-series data or data that is in some other way sequential. We can incorporate the temporal dependencies of our data into fully connected networks by structuring model inputs as in a distributed lag model. This approach, however, increases the model input space and requires corresponding increases to the size of model's parameter space. It also requires that all inputs to the model be of the same size and will require us to drop one observation per lag in our data.

Recurrent neural networks (RNNs) are a type of neural network that is designed for sequence data; in the context of forecasting, these type of networks can be a good alternative to a fully connected model. Unlike a fully connected network, a recurrent neural network layer imposes an ordering on its inputs and considers them as a sequence. Consider a sequence[6] $\boldsymbol{x} = (x_1, x_2 \ldots x_T)$. We can write a basic RNN (Fig. 6.5) model as $G(\boldsymbol{x}; \boldsymbol{\theta})$, with the output of any given layer written as:

$$
\begin{aligned}
g_t(\boldsymbol{x}) &= f(x_t \boldsymbol{B}_{\boldsymbol{x}} + g_{t-1}(\boldsymbol{x}) \boldsymbol{B}_g) \\
g_0(\boldsymbol{x}) &= \mathbf{0},
\end{aligned}
$$

where $\boldsymbol{B}_{\boldsymbol{x}}$ is a $1 \times p$ matrix of weights, $\boldsymbol{B}_g$ is a $p \times p$ matrix of weights, and the resulting $g_t(\boldsymbol{x})$ is a p-length vector.

This model diverges substantially from the fully connected architecture discussed in Sect. 6.1.1. All layers share a single set of weights, $(\boldsymbol{B}_x, \boldsymbol{B}_g)$. Further, while each layer $g_t(\boldsymbol{x})$ receives input from the preceding layer $g_{t-1}(\boldsymbol{x})$, each layer also receives external input from the t-th element in $\boldsymbol{x}$. Because each layer includes a new input and because each layer's output is taken as input to the subsequent layer, we can think of $g_t(\boldsymbol{x})$ as representing the *state* of the model at a specific point in the sequence. The state of the model at t is an accumulation of the model response

[5] An alternative form of the wavelet neural network uses wavelet functions as activation functions for hidden nodes in the network. This form of wavelet network, however, is designed to improve optimization speeds, create self-assembling networks, or achieve ends other than accommodating non-stationary data.

[6] We focus here on a sequence of scalar values. All discussion in this section extends to sequences of multi-dimensional input (e.g., a sequence of vectors).

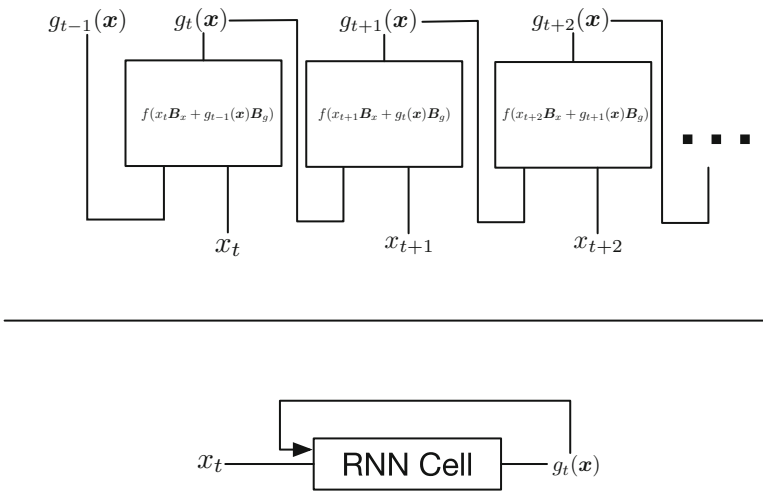

Fig. 6.5 Comparison of cell-based and unrolled implementations of an RNN. The top network represents the unrolled conceptualization of the RNN. The bottom network illustrates a network containing an RNN cell

to all items in $\boldsymbol{x}$ prior to t and as such, can be thought of as a representation of the network's *memory*.

In a forecasting framework, we might only be primarily interested in the final layer output $g_T(\boldsymbol{x})$, which we could treat as an estimate of a target variable at a specified forecast horizon, y_{T+h}. However, because of the structure of this network and the fact that its parameters are shared, we can collect the output of each layer as $\boldsymbol{g} = (g_1(\boldsymbol{x}), g_2(\boldsymbol{x}) \dots g_T(\boldsymbol{x}))$ in which case the network becomes a mapping $G : \boldsymbol{x} \rightarrow \boldsymbol{g}$.

To this point, we have been discussing the model $G(\boldsymbol{x}; \boldsymbol{\theta})$ as a set of T network layers. This conception of an RNN is called the "unrolled" form of an RNN. It is useful and more intuitive to illustrate the concept of an RNN in the context of its unrolled form. In practice, however, it is often more efficient to program the RNN as a special network object called a cell. A cell implements a for-loop in the computational graph of the model. Implemented as a cell, the RNN produces the entire sequence $\boldsymbol{g}$ from the inputs $\boldsymbol{x}$ and occupies the same space in a network as a single layer. This makes it easier to embed the RNN into larger networks and brings computational benefits in terms of memory efficiency.

A simple RNN, such as the one described above, illustrates the concept of an RNN, but it will not perform well will lengthy input sequences. As discussed by Bengio et al. (1994), they will have difficulty learning long-term time-dependencies. For example, a simple RNN may have difficulty learning that the impact of a shock in an input time series leads the response in the output series by several periods. When simple RNN models do learn long-term dependencies, they usually suffer from vanishing gradients. When this occurs, the model parameters become

established based upon only early portions of the input sequence and later portions of the input sequence have no effect on parameter updates.

The Long Short Term Memory (LSTM) network (Fig. 6.6) (Hochreiter & Schmidhuber, 1997) has emerged as a variant of the RNN that does not suffer from vanishing gradients and is capable of learning long-term dependencies. This type of network incorporates both long-run state information (long-run memory) as well as short-term state information (short-term memory). This type of network also includes mechanisms for resetting the long-run memory and thereby helping to avoid the vanishing gradients problem (Gers, Schmidhuber, & Cummins, 2000).

An LSTM network is a collection of several equations that take the current input, x_t, the network output generated at the previous timestep, $\boldsymbol{h}_{t-1}$, and the network state $\boldsymbol{s}_{t-1}$, which is responsible for its long-run memory, and produce a new output, $\boldsymbol{h}_t$, and an updated version of the network state, $\boldsymbol{s}_t$. Collecting $\boldsymbol{Z}_t = [x_t, \boldsymbol{h}_{t-1}]$, and writing the inverse logit function as σ, we can represent an LSTM cell with two equations,

$$\boldsymbol{s}_t = \overbrace{\underbrace{\boldsymbol{s}_{t-1}}_{\text{old state}} \odot \underbrace{\sigma(\boldsymbol{Z}_t \boldsymbol{B}_d)}_{\text{delete selection}}}^{\text{forget}} + \overbrace{\underbrace{\sigma(\boldsymbol{Z}_t \boldsymbol{B}_i)}_{\text{modification selection}} \odot \underbrace{\tanh(\boldsymbol{Z}_t \boldsymbol{B}_c)}_{\text{modification magnitude}}}^{\text{modify}} \tag{6.7}$$

$$\boldsymbol{h}_t = \sigma(\boldsymbol{Z}_t \boldsymbol{B}_o) \odot \tanh(s_t). \tag{6.8}$$

Equation (6.7) updates the LSTM cell's memory. It is comprised of two components. The first component is a forget step, which selects which components of the cell's memory to delete. The second component is a modification step which identifies which portions of the state should be modified and the extent of modification. The raw cell output, $\boldsymbol{h}_t$, is a representation of the cell state (memory) filtered through an output gate based on the current input and previous cell output. The use of hyperbolic tangent (tanh) activation functions serves to maintain the scale of values in the state and cell outputs. This helps to prevent gradients from vanishing or exploding.

With the raw output, $\boldsymbol{h}_t$ from an LSTM cell, we typically add an additional layer to transform it into a direct prediction that is compatible with our target variable, $\hat{y}_t = f(\boldsymbol{h}_t \boldsymbol{\beta}_y)$.

Note, the parameters $\boldsymbol{\theta} = [\boldsymbol{B}_d, \boldsymbol{B}_i, \boldsymbol{B}_c, \boldsymbol{B}_o, \boldsymbol{\beta}_y]$ are shared across timesteps. At the same time, note that the LSTM cell passes the raw output $\boldsymbol{h}_t$ and long-run state, $\boldsymbol{s}_t$, from one timestep to the next. Thus, even though the LSTM cell parameters are shared across timesteps, the computation of the gradients for the parameters requires iterating backward through the timesteps (see Werbos, 1990) to compute

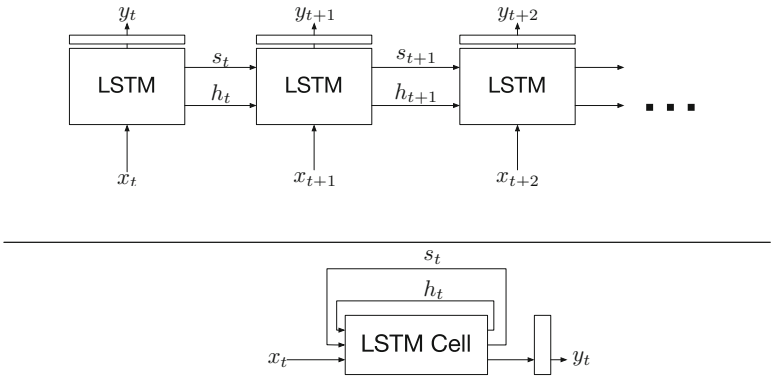

Fig. 6.6 Illustration of rolled and unrolled versions of an LSTM cell. This figure is similar to Fig. 6.4, the top network represents the unrolled conceptualization of the LSTM. The bottom network illustrates a network containing an LSTM cell. The operations within the layers labeled "LSTM" and "LSTM Cell" are provided in Eqs. (6.7)–(6.8). The networks are shown with a final layer that transforms output h into its final form, y

the intermediate gradients for the state and raw output:

$$\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{h}_t} = \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial y_t}\boldsymbol{\beta}_y + \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{h}_{t+1}}\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}$$
$$\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{s}_t} = \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{h}_t} \odot \tanh'(\boldsymbol{s}_t) + \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{s}_{t+1}}\frac{\partial \boldsymbol{s}_{t+1}}{\partial \boldsymbol{s}_t}.$$

We can recover these gradients by observing that $\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{s}_{T+1}} = \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{h}_{T+1}} = \mathbf{0}$ and that

$$\begin{aligned}\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{h}_{t-1}} =& \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \sigma(\boldsymbol{Z}_t\boldsymbol{B}_d)}\sigma'(\boldsymbol{Z}_t\boldsymbol{B}_d)\dot{\boldsymbol{B}}_d + \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \sigma(\boldsymbol{Z}_t\boldsymbol{B}_i)}\sigma'(\boldsymbol{Z}_t\boldsymbol{B}_i)\dot{\boldsymbol{B}}_i + \\ & \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \tanh(\boldsymbol{Z}_t\boldsymbol{B}_c)}\tanh'(\boldsymbol{Z}_t\boldsymbol{B}_c)\dot{\boldsymbol{B}}_c + \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \sigma(\boldsymbol{Z}_t\boldsymbol{B}_o)}\sigma'(\boldsymbol{Z}_t\boldsymbol{B}_o)\dot{\boldsymbol{B}}_o \\ \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{s}_{t-1}} =& \frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{s}_t} \odot \sigma(\boldsymbol{Z}_t\boldsymbol{B}_d),\end{aligned}$$

where $\dot{\boldsymbol{B}}$ indicates the portion of the parameter that is multiplied by $\boldsymbol{h}_{t-1}$ in $Z_t\boldsymbol{B} = (x_t, h_{t-1})\boldsymbol{B}$.

With these intermediate gradients, we can calculate gradients for each item in $\boldsymbol{\theta}$,

$$\begin{aligned}
\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{\beta}_y} &= \boldsymbol{h}\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{y}} \\
\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{B}_d} &= Z_t\left(\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial s_t} \odot \boldsymbol{s}_{t-1} \odot \sigma'(Z_t\boldsymbol{B}_d)\right) \\
\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{B}_i} &= Z_t\left(\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial s_t} \odot \tanh(Z_t,\boldsymbol{B}_c) \odot \sigma'(Z_t\boldsymbol{B}_i)\right) \\
\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{B}_c} &= Z_t\left(\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial s_t} \odot \sigma(Z_t,\boldsymbol{B}_i) \odot \tanh'(Z_t\boldsymbol{B}_c)\right) \\
\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{B}_o} &= Z_t\left(\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial h_t} \odot \tanh(\boldsymbol{s}_t) \odot \sigma'(Z_t\boldsymbol{B}_o)\right).
\end{aligned}$$

With the gradients calculated, we can fit the model via gradient descent.

6.4 Encoder-Decoder

The LSTM model can be used in the context of forecasting as follows. Consider a time series $\boldsymbol{X} = (x_1 \dots x_T)$, a target series corresponding to an h-step ahead forecast horizon $\boldsymbol{y} = (y_{1+h}, y_{2+h} \dots y_{T+h})$, and an LSTM model $G(\boldsymbol{x};\boldsymbol{\theta}) = \hat{y}_{T+h}$. The estimate produced by the LSTM model would be analogous to a direct forecast (see Marcellino, Stock, & Watson, 2006). An iterative forecast could be generated, but the fundamental LSTM model would remain unchanged.

Instead, we can make use of an *encoder-decoder* network (Cho et al., 2014; Sutskever, Vinyals, & Le, 2014). This type of network is a member of a broader class of networks called sequence-to-sequence networks. The encoder-decoder architecture was initially developed to facilitate language modeling tasks (e.g., translation). Specifically, it was developed to allow a model to predict words in the output while considering the context of individual words in the input along with the context of the words that have already been predicted in the output.

The model is comprised of two components, aptly named the encoder and the decoder. The encoder consists of the RNN model from the previous section, $G(\boldsymbol{x};\boldsymbol{\theta})$. For the purposes of our discussion here, consider the encoder to be an LSTM cell with an accompanying fully connected final layer. The encoder takes the sequence $\boldsymbol{x}$ and returns a fixed-length representation. Conventionally, we specify this fixed-length representation as the final output from the models, $g_T(\boldsymbol{x})$. We also recover from $G(\boldsymbol{x};\boldsymbol{\theta})$ the RNN cell's final state, $\boldsymbol{s}_T$.

The second component of the model is called the decoder. It consists of an RNN network and a final, fully connected layer. Whereas the encoder began with a variable length sequence and produced a fixed-length output $g_T(\boldsymbol{x})$, the decoder

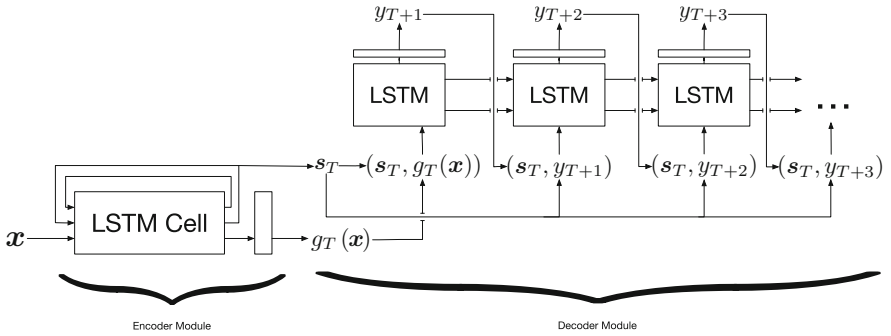

Fig. 6.7 Illustration of an encoder-decoder network. The figure shows an LSTM cell encoding inputs $\boldsymbol{x}$ into a fixed-length representation via an LSTM cell. The fixed-length representation is then processed through an unrolled LSTM network (the decoder module) to produce a variable length sequence $\boldsymbol{y}$. Network weights for the encoder module are shared across timesteps; weights for the decoder module are also shared across timesteps

begins with a fixed-length input $g_T(\boldsymbol{x})$ and produces a variable length output. It does this by taking output of the previous timestep as input to produce output for the current timestep. In practice, for use in forecasting, we would fix the length of the decoder output to correspond to the desired forecast horizon.

As a specific implementation, consider the decoder, $D(\boldsymbol{g}, \boldsymbol{s}_T; \boldsymbol{\theta})$, as an LSTM network with a fully connected final layer. Following Cho et al. (2014), the decoder takes in the final encoder state, s_T, as part of its input at every timestep. Denote by $\boldsymbol{h}_t$ the raw output[7] of the LSTM at timestep t and as produced by d_t. We can write the decoder output of each timestep along the forecast horizon, $h \in (1, 2, \ldots, H)$, as

$$
\begin{aligned}
y_{T+h} &= f(d_{T+h}([y_{T+h-1}, \boldsymbol{s}_T], \boldsymbol{h}_{T+h-1})\boldsymbol{\beta}_y) \\
y_{T+0} &= f(d_{T+0}([g_T(\boldsymbol{x}), \boldsymbol{s}_T], 0)\boldsymbol{\beta}_y),
\end{aligned}
$$

where f is the decoder's final layer activation function and where $\boldsymbol{\beta}_y$ is the vector of weights for corresponding to the decoder's final layer. Note that just as with the RNN cell, the parameters $\boldsymbol{\beta}_y$ are shared across timesteps. The reason for this is to ensure that the raw output from the RNN cell is converted into a target output in a consistent fashion for each timestep. Figure 6.7 provides an illustration of this entire encoder-decoder network.

Gradients for this model are derived in the same fashion as they are for RNNs, via backpropagation through time. As with the other neural network models discussed in this chapter, we train this model using gradient descent.

[7] In other words, the output of the decoder LSTM prior to the final, fully connected layer.

6.5 Empirical Application: Unemployment Forecasting

In this section we will examine the performance of the three neural network architectures (Fully Connected, LSTM, Encoder-Decoder) as applied to the task of unemployment forecasting. This analysis will closely follow Cook and Hall (2017).

6.5.1 Data

To test the performance of the neural network approach, we trained each of the models presented above to predict the civilian unemployment rate. This measure is collected monthly by the US Bureau of Labor and Statistics. It measures the percentage of the labor force that is currently unemployed. The unemployment rate only measures unemployment in the US. At the time of this writing, data for the unemployment rate is available as far back as 1948, and as recently as last month.

Unemployment is a useful indicator to target for this exercise for a few reasons. First, unemployment is a substantively meaningful indicator to forecast; the Federal Reserve works to manage the unemployment rate as part of its dual mandate, and it is closely monitored by economic actors and scholars across a variety of sectors.

Second, in contrast to GDP, unemployment usually undergoes limited revision after its initial release. This is an important consideration since it allows us to generally sidestep the problems of collecting and assembling appropriate "vintages" of the data. We use the last release of the unemployment rate for all training and testing. To be clear, the largest discrepancy between the original vintage of the data and final release of the data is about 23 basis points, with the average discrepancy being nine basis points. We will assume the impact of these discrepancies on the predictive accuracy of our forecasts to be negligible.

For this exercise, we will target 1, 3, 6, 9, and 12 month forecast horizons for the target. For each forecast horizon, we train each of the three models presented above, yielding 15 total model variants for training.

The target will be the sole series used as input for each of the models. For each observation, the model inputs are the previous 36 monthly values of the target, along with first and second order differences in the target. In theory, the model could identify and extract the first and second order differences of the input data, but we supply them directly because (1) we can be reasonably certain that they will supply the model with useful information and (2) because it allows us to reduce the training time and simplify the model structure.

It is possible and relatively easy to add additional series to these models and there should be performance gains from doing so. We will refrain from adding additional series here, however, as this will simplify our discussion of the model.

6.5.2 Model Specification

The fully connected model is comprised of one hidden layer, with a 32 neurons, and a final output layer consisting of a single neuron. The ReLU activation function is applied to each neuron in the hidden layer. The output layer neuron uses a linear (i.e., the identity) activation function. Dropout is applied to all layers with a probability of dropout set to 10%. Weight decay is also applied to all layer with a value of 0.0009. Each of the hyperparameters was chosen via hand tuning.

The LSTM model is comprised of a single LSTM cell with state and output sizes set to twelve. Due to complexities with the LSTM cell, it does not employ dropout or weight decay. The output layer of the LSTM model, is a single neuron with a linear activation function. We could apply the output layer to all outputs from the LSTM cell yielding $G(\boldsymbol{x}|\boldsymbol{\theta}) = (f(\boldsymbol{h}_1\boldsymbol{\beta}_y), f(\boldsymbol{h}_2\boldsymbol{\beta}_y), \ldots f(\boldsymbol{h}_T\boldsymbol{\beta}_y))$. However, since we only care about the final output from the sequence, we discard the output of all earlier timesteps and apply the output layer to only the output from the final timestep, yielding our model output $G(\boldsymbol{x}|\boldsymbol{\theta}) = f(\boldsymbol{h}_T\boldsymbol{\beta}_y)$. This reduces the computational cost of model training by reducing the complexity of calculating $\frac{\partial L(y_t;\boldsymbol{\theta})}{\partial \boldsymbol{\beta}_y}$.

The Encoder-Decoder model uses two LSTM cells and a final, fully connected output layer. The encoder module is identical to the LSTM model just described. The decoder module consists of an LSTM module with a state size of twelve. A final output layer consisting of a single neuron with a linear activation function is applied to the output of each timestep. The parameters of this output layer are shared across all timesteps. As described Eq. (6.4), the initial input to the decoder is the output from the encoder module. At every subsequent timestep, the input to the decoder is the decoder output from the previous timestep.

6.5.3 Model Training

We construct a training data set from the unemployment rate data from 1963 to 1996. Every tenth observation in this period is sequestered into a validation dataset. We use the validation dataset to evaluate the performance of the model and implement early stopping in the training process. The remainder of the data, from 1997 to 2015, is sequestered into a testing dataset. We use this dataset to assess the performance of the trained model.

The training process is subject to stochasticity. The initial weights for each model network are randomly distributed using Xavier initialization. Random weight initialization drives stochasticity in the training process. Beyond this, there are a few other sources of stochasticity in the training process, including dropout and the optimization routine itself (mini-batch Adam).

As a consequence of the stochasticity inherent to the model training process, repeated runs of the same model will yield trained networks that vary in their weights and, consequently, in forecasts. To accommodate this variance, we train 30

Table 6.2 Performance metrics for DARM and neural network models at 0–4 quarter prediction horizons

Horizon		Fully connected	LSTM	Encoder decoder	DARM
1 Months	Mean MAE	20.61	4.10	4.02	11.7
	St. Dev.	4.22	0.12	0.07	
3 Months	Mean MAE	25.38	15.43	15.53	32.8
	St. Dev.	5.50	0.08	0.21	
6 Months	Mean MAE	34.64	28.76	29.00	49.3
	St. Dev.	4.45	0.61	0.25	
9 Months	Mean MAE	47.69	44.99	44.79	65.8
	St. Dev.	3.26	1.93	0.88	
12 Months	Mean MAE	63.45	63.06	61.01	90.7
	St. Dev.	3.04	3.28	1.70	

All metrics presented as hundredths of one percent

instances of each model. This allows us to assess expected model performance as well as assess the variance in performance across repeated runs of the same model.

All model variants trained in less than 5 min.

6.5.4 Results

Model performance is provided in Table 6.2. Each of the first three columns describe the performance of a model in terms of test mean absolute error (MAE), aggregated across repeated iterations. The mean MAE indicates the average model performance. The standard deviation of the MAE gives some sense of the distribution in model performance across repeated trainings of a model. The final column provides performance metrics against a benchmark model.

As a benchmark, we consider a direct[8] autoregressive model (DARM) that uses monthly data. The model is specified as follows:

$$\hat{y}_{t+h} = \sum_{i=1}^{k} \beta_i y_{t-i}, \tag{6.9}$$

where t indexes the time of forecast, k is the number of lags, and n indicates the forecast horizon. In this paper, we use the DARM model estimates published by the SPF (Stark, 2017).

[8]This is to be contrasted with an iterative model, in which the next-step-ahead is forecast and then iterative extrapolation is used to generate a prediction for the desired forecast horizon.

Broadly speaking, each of the neural network models outperform the benchmark model, with the exception of the fully connected model at the 1 month horizon. The encoder-decoder and LSTM models outperform the fully connected models quite strongly at the early horizons. At the 9 and 12 month horizons, the models converge in performance. It is notable, however, that the standard deviation of the mean absolute forecasting error is considerably lower for the LSTM and encoder-decoder models, with the encoder-decoder model having the lowest variance in performance at most horizons.

6.6 Conclusion

This chapter has discussed the fundamentals of neural network models with a primary focus on their application to supervised, predictive tasks. Through this discussion, it showed the flexibility of neural networks and their potential for application to econometric tasks such as forecasting. Yet this chapter is by no means a complete description of the potential of neural networks in econometric settings. Macroeconomists might find additional uses for neural networks in unsupervised econometric applications (e.g., interpreting textual data or generating low dimensional representations of large datasets), or agent-based applications (where neural networks might be used in the context of reinforcement learning). Moreover, as new sources of "Big Data" emerge, economists will be able to train networks to produce increasingly sophisticated outputs or to operate on increasingly complex inputs. Lastly, it is important to note that neural networks represent an area of rapid methodological research and innovation. For example, strong efforts are afoot to adapt neural networks for use within the framework of causal inference. As these efforts develop, so will the utility of neural networks in macroeconomic analysis.

References

Altman, E. I., Marco, G., & Varetto, F. (1994). Corporate distress diagnosis: Comparisons using linear discriminant analysis and neural networks (the Italian experience). *Journal of Banking & Finance, 18*(3), 505–529.

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems* (pp. 153–160).

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks, 5*(2), 157–166. Retrieved from http://www.comp.hkbu.edu.hk/~markus/teaching/comp7650/tnn-%2094-gradient.pdf

Bland, R. (1998). *Learning xor: Exploring the space of a classic problem*. Stirling: Department of Computing Science and Mathematics, University of Stirling.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint, 1406.1078.

Cook, T. R., & Hall, A. S. (2017). Macroeconomic indicator forecasting with deep neural networks. *Federal Reserve Bank of Kansas City Research Working Paper* (pp. 17-11).

Cybenko, G. (1989). Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems, 2*, 183–192.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems* (pp. 2933–2941).

Dijk, D. v., Teräsvirta, T., & Franses, P. H. (2002). Smooth transition autoregressive models—A survey of recent developments. *Econometric Reviews, 21*(1), 1–47.

Dixon, M., Klabjan, D., & Bang, J. H. (2017). Classification-based financial markets prediction using deep neural networks. *Algorithmic Finance, 6*(3–4), 67–77.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*(7), 2121–2159.

Gal, Y., & Ghahramani, Z. (2016). Dropout as a Bayesian approximation. In *Proceedings of the 33rd International Conference on Machine Learning* (Vol. 3, pp. 1661–1680).

Gers, F. A., Schmidhuber, J., & Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation, 12*(10), 2451–2471.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feed-forward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 249–256). Retrieved from http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf?%20hc_location=ufi

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge: MIT Press. http://www.deeplearningbook.org

Hastad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (pp. 6–20).

Heaton, J., Polson, N. G., & Witte, J. H. (2016). Deep learning in finance. arXiv preprint, 1602.06561.

Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation, 18*(7), 1527–1554.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks, 4*(2), 251–257.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks, 2*(5), 359–366.

Huang, G.-B. (2003). Learning capability and storage capacity of two-hidden-layer feedforward networks. *IEEE Transactions on Neural Networks, 14*(2), 274–281.

Huang, G.-B., & Babri, H. A. (1997). General approximation theorem on feedforward networks. In *Proceedings of the 1997 International Conference on Information, Communications and Signal Processing* (Vol. 2, pp. 698–702). Piscataway: IEEE.

Jothimani, D., Yadav, S. S., & Shankar, R. (2015). Discrete wavelet transform-based prediction of stock index: A study on national stock exchange fifty index. *Journal of Financial Management and Analysis, 28*(2), 35–42.

Karlik, B., & Olgac, A. V. (2011). Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems, 1*(4), 111–122. Retrieved from https://www.researchgate.net/publication/%20228813985_Performance_Analysis_of_Various_Activation_Functions_in_Generalized_MLP_Architectures_of_Neural_Networks

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint, 1412.6980.

Kristjanpoller, W., & Minutolo, M. C. (2015). Gold price volatility: A forecasting approach using the artificial neural network–GARCH model. *Expert Systems with Applications, 42*(20), 7245–7251.

Krogh, A., & Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems* (pp. 950–957).

Lineesh, M., Minu, K., & John, C. J. (2010). Analysis of nonstationary nonlinear economic time series of gold price: A comparative study. In *International Mathematical Forum* (Vol. 5, 34, pp. 1673–1683). Citeseer.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning* (Vol. 30, *1*, p. 3). Retrieved from http://robotics.stanford.edu/~amaas/papers/%20relu_hybrid_icml2013_final.pdf

Marcellino, M., Stock, J. H., & Watson, M. W. (2006). A comparison of direct and iterated multistep AR methods for forecasting macroeconomic time series. *Journal of Econometrics, 135*, 499–526.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics, 5*(4), 115–133.

McNelis, P. (2005). *Neural networks in finance: Gaining predictive edge in the market*. Amsterdam: Elsevier.

Minsky, M., & Papert, S. (1969). *Perceptrons: An introduction to computation geometry* (Vol. 200, pp. 355–368). Cambridge: MIT Press.

Minu, K., Lineesh, M., & John, C. J. (2010). Wavelet neural networks for nonlinear time series analysis. *Applied Mathematical Sciences, 4*(50), 2485–2495.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning* (pp. 807–814). Retrieved from http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf

Odom, M. D., & Sharda, R. (1990). A neural network model for bankruptcy prediction. In *Proceedings of the 1990 International Joint Conference on Neural Networks* (pp. 163–168). Piscataway: IEEE.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. arXiv preprint, 1710.05941. Retrieved from https://arxiv.org/pdf/1710.05941

Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint, 1609.04747.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation*. San Diego: California University, La Jolla Institute for Cognitive Science.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature, 323*(6088), 533–536. Retrieved from http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S.,… Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision, 115*(3), 211–252.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929–1958.

Stark, T. (2017). *Error statistics for the survey of professional forecasters for unemployment rate*. Philadelphia: Federal Reserve Bank of Philadelphia. Retrieved from https://www.philadelphiafed.org/-/media/research-and-data/%20real-time-center/survey-of-professional-forecasters/data-%20files/unemp/spf_error_statistics_unemp_1_aic.pdf?la=en

Sussillo, D., & Abbott, L. (2014). Random walk initialization for training very deep feedforward networks. arXiv preprint, 1412.6558.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems* (pp. 3104–3112).

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D.,… Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9).

Tam, K. Y. (1991). Neural network models and the prediction of bank bankruptcy. *Omega, 19*(5), 429–445.

Telgarsky, M. (2016). Benefits of depth in neural networks. arXiv preprint, 1602.04485.
Terasvirta, T., & Anderson, H. M. (1992). Characterizing nonlinearities in business cycles using smooth transition autoregressive models. *Journal of Applied Econometrics, 7*(S1), S119–S136.
Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE, 78*(10), 1550–1560.
Zou, D., Cao, Y., Zhou, D., & Gu, Q. (2018). Stochastic gradient descent optimizes over-parameterized deep ReLU networks. arXiv preprint, 1811.08888.