Methodology

In conventional Vector Autoregression (VAR) dependencies of any system variable on past realizations of itself and its covariates are modelled through linear equations. This corresponds to a particular case of the broader class of Deep Vector Autoregressions investigated here and will serve as the baseline for our analysis.

Vector Autoregression

Let $\mathbf{y}_t$ denote the $(K \times 1)$ vector of variables at time t. Then the VAR(p) with p lags and a constant deterministic term is simply a linear system of stochastic equations of the following form:

$$\mathbf{y}_t = \mathbf{c} + \mathbf{A}_1\mathbf{y}_{t-1} + \mathbf{A}_2\mathbf{y}_{t-2} + ... + \mathbf{A}_p\mathbf{y}_{t-p} + \mathbf{u}_t (\#eq:redform) \tag{1}$$

The matrices $\mathbf{A}_m \in \mathbb{R}^{K\times K}$, where $m \in \{1, ..., p\}$, contain the reduced form coefficients and $\mathbf{u}_t \in \mathbb{R}^{K\times 1}$ is a vector of errors for which $\mathbb{E}\mathbf{u}_t$, $\mathbb{E}\mathbf{u}_t\mathbf{u}_t^T = \Sigma$ and $\mathbb{E}\mathbf{u}_t\mathbf{u}_s^T = \mathbf{0}$ for all $t \neq s$. We refer to @ref(eq:redform) as the **reduced form** representation of the VAR(p) because all right-hand side variables are predetermined [@kilian2017structural].

We can restate @ref(eq:redform) more compactly as

$$\mathbf{y}_t = \mathbf{A}\mathbf{Z}_{t-1} + \mathbf{u}_t (\#eq:sur) \tag{2}$$

where $\mathbf{A} = (\mathbf{c}, \mathbf{A}_1, \mathbf{A}_2, ..., \mathbf{A}_p) \in \mathbb{R}^{K\times(Kp+1)}$ and $\mathbf{Z}_{t-1} = (1, \mathbf{y}_{t-1}^T, ..., \mathbf{y}_{t-p}^T)^T \in \mathbb{R}^{(Kp+1)\times 1}$. The expression in @ref(eq:sur) demonstrates that the VAR(p) can be considered as a **seemingly unrelated regression** (SUR) model composed of individual regressions with common regressors [@greene2012econometric]. In fact, it is useful to note for our purposes that the VAR(p) can be estimated efficiently through equation-by-equation OLS regression. In particular, it follows from @ref(eq:sur) that

$$y_{it} = c_i + \sum_{m=1}^{p}\sum_{j=1}^{K} a_{jm}y_{jt-m} + u_{it} \quad , \quad \forall i = 1, ..., K (\#eq:single-var) \tag{3}$$

which corresponds to the key modelling assumption that at any point in time t any time series $i \in 1, ..., K$ is just a weighted sum of past realizations of itself and all other variables in the system. This assumption makes the estimation of VAR(p) processes remarkably simple. Perhaps more importantly, the assumption of linearity also greatly facilitates inference about VARs.

For implementation purposes it is generally more useful to estimate the VAR(p) through one single OLS regression. To this end let $\tilde{\mathbf{A}} = \mathbf{A}^T$ and note that @ref(eq:sur) can be restated even more compactly as

$$\mathbf{y} = \mathbf{Z}\tilde{\mathbf{A}} + \mathbf{u}_t (\#eq:var-ols) \tag{4}$$

with $\mathbf{y} = (\mathbf{y}_1, ..., \mathbf{y}_T)^T \in \mathbb{R}^{T\times K}$ and $\mathbf{Z} \in \mathbb{R}^{T\times(Kp+1)}$. Then the closed form solution for OLS is simply $\tilde{\mathbf{A}} = (\mathbf{Z}^T\mathbf{Z})^{-1}\mathbf{Z}^T\mathbf{y}$ and hence

$$\mathbf{A} = \mathbf{y}^T\mathbf{Z}(\mathbf{Z}\mathbf{Z}^T)^{-1} (\#eq:ols-sol) \tag{5}$$

Deep Vector Autoregression

We propose the term Deep Vector Autoregression to refer to the broad class of Vector Autoregressive models that use deep learning to model the dependences between system variables through time. In particular, as before, we let $\mathbf{y}_t$ denote the $(K \times 1)$ vector that describes the state of system at time t. Consistent with the conventional VAR structure we assume that each individual time series y_{it} can be modelled as a function of lagged realizations of all variables y_{jt-p}, $j = 1, ..., K$, $m = 1, ..., p$. More specifically, we have

$$y_{it} = f_i \left(\mathbf{y}_{t-1:t-p}; \theta \right) + v_{it} \quad , \quad \forall i = 1, ..., K (\#eq:single-dvar) \tag{6}$$

where $\mathbf{y}_{t-1:t-p} = \{y_{jt-m}\}_{j=1,...,K}^{m=1,...,p}$ is the vector of lagged realizations, f_i is a variable specific mapping from past lags to the present and θ is a vector of parameters. While in the conventional VAR above we assumed that the multivariate process can be modelled as a system of linear stochastic equations, our proposed Deep VAR(p) can similarly be understood as a system of potentially highly non-linear equations. As we argued earlier, Deep Learning has been shown to be remarkably successful at learning mappings of arbitrary functional forms [@goodfellow2016deep].

Note that the input and output dimensions in @ref(eq:single-dvar) are exactly the same as in the conventional VAR(p) model (equation @ref(eq:single-var)): f_i maps from $\mathbf{y}_{t-1:t-p} \in \mathbb{R}^{Kp \times 1}$ to a scalar. Our proposed plain-vanilla approach to Deep VARs diverges as little as possible from the conventional approach: it boils down to simply modelling each of the univariate outcomes in @ref(eq:single-dvar) as a deep neural network. We can restate this approach more compactly as

$$\mathbf{y}_t = \mathbf{f}(\mathbf{y}_{t-1:t-p}; \theta) + \mathbf{v}_t (\#eq:dvar) \tag{7}$$

where $\mathbf{f}(\cdot) = (f_1(\cdot), f_2(\cdot), ..., f_K(\cdot))^T \in \mathbb{R}^{K \times 1}$ is just the stacked vector of mappings to univariate outcomes described in @ref(eq:single-dvar).

The notation in @ref(eq:dvar) gives rise to a more unified and general approach to Deep VARs that would treat the whole process as one single dynamical system to be modelled through one deep neural network $\mathbf{g}$:

$$\mathbf{y}_t = \mathbf{g}(\mathbf{y}_{t-1:t-p}; \theta) + \mathbf{v}_t (\#eq:dvar-uni) \tag{8}$$

This approach is in fact proposed and investigated by @verstyuk2020modeling in his upcoming publication. We decided to go with the approach in @ref(eq:dvar) for two reasons: firstly, the link to conventional VARs is made abundantly clear through this implementation and, secondly, we found that the equation-by-equation approach produces good modelling outcomes and is relatively easy to implement using state-of-the art software.

Finally, note that if f_i in @ref(eq:single-var) is assumed to be linear and additive for all $i = 1, ..., K$ then we are back to the conventional VAR(p). This illustrates the point we made earlier that the linear VAR(p) is just a particular case of a Deep VAR(p). Since the model described in equations @ref(eq:single-dvar) and @ref(eq:dvar) is less restrictive but otherwise consistent with the conventional VAR framework, we expect that it outperforms the traditional approach towards modelling multivariate time series processes.

Deep Neural Networks - a whistle-stop tour

So far we have been speaking about deep learning in rather general terms. For example, above we have referred to our model of choice for learning the mapping $f_i : \mathbf{y}_{t-1:t-p} \mapsto y_{it}$ as a **deep neural network**. The class of deep neural networks can further be roughly divided into **feedforward neural networks** and **recurrent neural networks**. As the term suggests, the latter is generally used for sequential data and therefore our preferred model of choice. Nonetheless, below we will begin by briefly exploring feedforward neural networks first. This should serve as a good introduction to neural networks more generally and (even though we have not tested this empirically) there is good reason to believe that even Deep VARs using feedforward neural networks perform well.

Deep Feedforward Neural Networks

The term **deep feedforward neural network** or **multilayer perceptron** (MLP) is used to describe a broad class of models that are composed of possibly many functions that together make up the directed acyclical graph. The functions $f_i(\cdot)$ - sometimes referred as layers $\mathbf{h}_i$ - are chained together hierarchically with the first layer feeding forward its outputs to the second layer and so on [@goodfellow2016deep]. Applied to our case, an MLP with H hidden layers can be loosely defined as follows:

$$f_i(\mathbf{y}_{t-1:t-p};\theta) = f_i^{(H)}\left(f_i^{(H-1)}\left(...f_i^{(1)}\left(\mathbf{y}_{t-1:t-p}\right)\right)\right)(\#eq:mlp) \tag{9}$$

The depth of the MLP is defined by the number of hidden layers H, where, generally speaking, deeper networks are more complex.

The desired outputs of any $f_i^{(h)}$ that will serve as inputs for $f_i^{(h+1)}$ cannot be inferred from the training data $\mathbf{y}_{t-1:t-p}$ ex-ante, which is where the term **hidden** layer stems from. Each $f_i^{(h)}$ is typically valued on a vector of hidden units, each of them receiving a vector of inputs from $f_i^{(h-1)}$ and returning a scalar that is referred to as activation value. This approach is inspired by neuroscience, hence the term **neural** network [@goodfellow2016deep].

Deep Recurrent Neural Networks

Recurrent neural networks (RNN) are based on the idea of persistent learning: a continuous process that evolves gradually and at each step uses information about its prior states instead of continuously reinventing itself and starting from scratch. To this end, RNNs develop the basic concepts underlying feedforward neural networks by incorporating feedback loops. Formally the loop is typically made explicit as follows

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t;\theta)(\#eq:hidden-state) \tag{10}$$

where $\mathbf{h}_t \in \mathbb{R}^{N\times 1}$ corresponds to the hidden state of the dynamical system at time t that the RNN learns [@goodfellow2016deep], and N corresponds to the number of hidden units in each hidden layer, known as the width of the layer. In the given context we have that $\mathbf{x}_t = \mathbf{y}_{t-1:t-p}$ as specified in @ref(eq:dvar). Given some random initial hidden state vector $\mathbf{h}_0$ the RNN updates parameters sequentially at each time step t as follows

$$\begin{aligned} \mathbf{a}_t &= \mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{h}_{-1} \\ \mathbf{h}_t &= \tanh(\mathbf{a}_t) \qquad\qquad (\#eq:rnn) \\ \hat{\mathbf{y}}_t &= \mathbf{c} + \mathbf{V}\mathbf{h}_t \end{aligned} \tag{11}$$

where $\mathbf{b} \in \mathbb{R}^{N\times 1}$ and $\mathbf{c} \in \mathbb{R}^{K\times 1}$ are vectors of constants (biases), tanh is the hyperbolic tangent activation function and $\mathbf{W}$, $\mathbf{U}$, $\mathbf{V}$ are coefficient matrices, where $\mathbf{W}, \mathbf{U} \in \mathbb{R}^{N\times N}$ and $\mathbf{V} \in \mathbb{R}^{K\times N}$. Note that to simplify the notation we have omitted the layer index in @ref(eq:rnn): to be specific, $\mathbf{h}_t$ really represents $\mathbf{h}_t^{(H)}$ (the ultimate hidden layer), $\mathbf{h}_{-1}$ stands for $\mathbf{h}_t^{(H-1)}$ (the penultimate layer). Finally, at each step t the first layer $\mathbf{h}_t^{(0)}$ of the forward propagation corresponds to $\mathbf{y}_{t-1:t-p}$.

A shortfall of generic recurrent neural networks is that they fail to capture long-term dependencies. More specifically, if parameters are propagated over too many stages in a simple RNN, it typically suffers from the problem of **vanishing gradients** [@goodfellow2016deep]. Fortunately, there exist effective extensions of the RNN, most notably the long short-term memory (LSTM), which was introduced by @hochreiter1997long and is our model of choice for Deep VARs. The key idea underlying LSTMs is to regulate exactly how much information is propagated from one cell state vector $\mathbf{C}_{t-1}$ to the next $\mathbf{C}_t$ through the introduction of so called sigmoid gates:

"The LSTM [has] the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through." — @olah2015understanding

These regulating gate layers include a **forget gate** $\mathbf{f}_t$, an **input gate** $\mathbf{i}_t$ and a **output gate** $\mathbf{o}_t$. Each of them are vector-values sigmoid functions whose elements $\mathbf{f}_{it}, \mathbf{i}_{it}, \mathbf{o}_{it}$ are bound between 0 and 1. Their individual purposes are implied by their names: faced with $\mathbf{h}_{t-1}$ and $\mathbf{y}_{t-1:t-p}$, the forget gate regulates how much of each individual unit in $\mathbf{C}_{t-1}$ is retained. Then the input gate regulates which units of $\mathbf{C}_{t-1}$ should be updated and to what candidate values $\tilde{\mathbf{C}}_{t-1}$. Using the previous two steps the actual update is performed according to the following rule

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_{t-1} (\#eq : lstm - update) \tag{12}$$

where $\odot$ indicates the element-wise product. Finally, the output gate acts like a filter on $\mathbf{C}_t$: the new hidden state is computed as $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t)$ where as before we use the hyperbolic tangent as our activation function.[1] Formally, we can summarize the LSTM neural network underlying our Deep VAR framework as follows:

$$\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{b}_f + \mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{h}_{-1}\right) \\
\mathbf{i}_t &= \sigma\left(\mathbf{b}_i + \mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{h}_{-\mathbf{1}}\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{b}_o + \mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{h}_{-1}\right) \\
\mathbf{C}_t &= \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tanh\left(\mathbf{b}_C + \mathbf{W}_C \mathbf{h}_{t-1} + \mathbf{U}_C \mathbf{h}_{-1}\right) \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \\
\hat{\mathbf{y}}_t &= \mathbf{c} + \mathbf{V}\mathbf{h}_t
\end{aligned} (\#eq : lstm) \tag{13}$$

which is best understood when read from top to bottom. Once again we have simplified the notation by ommitting the layer index in @ref(eq:rnn). The same notation as before applies.

Model selection

There are at least two important modelling choices to be made in the context of conventional VARs. The first choice concerns properties of the time series data itself, in particular the order of integration and cointegration. The second choice is about the the lag order p. In order to arrive at appropriate decisions regarding these choices the VAR literature provides a set of guiding principles. We propose to apply these same principles to the Deep VAR, firstly because they are intuitive and simple and secondly because treating both models equally to begin with allows for a better comparison of the two models at the subsequent modelling stages.

Stationarity

When working with time series we are generally concerned about stationarity. Broadly speaking stationarity ensures that the future is like the past and hence any predictions we make based on past data adequately describe future outcomes. In order to state stationarity conditions in the VAR context it is convenient to restate the K-dimensional VAR(p) process in companion form as

$$\mathbf{Y}_t = \begin{pmatrix} \mathbf{c} \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \mathbf{A}\mathbf{Y}_{t-1} + \begin{pmatrix} \mathbf{u}_t \\ 0 \\ \vdots \\ 0 \end{pmatrix} (\#eq : companion) \tag{14}$$

[1]For a clear and detailed exposition see @olah2015understanding.

where $\mathbf{Y}_t = (\mathbf{y}_t^T, ..., \mathbf{y}_{t-p+1}^T)^T \in \mathbb{R}^{Kp \times 1}$ and $\mathbf{A} \in \mathbb{R}^{Kp \times Kp}$ is referred to as the companion matrix [@kilian2017structural]. Stationarity of the VAR(p) follows from stability: a VAR(p) is stable if the effects of shocks to the system eventually die out. Stability can be assessed through the system's autoregressive roots or equivalently by looking at the eigenvalues of the companion matrix $\mathbf{A}$ [@kilian2017structural]. In particular, for the VAR(p) in @ref(eq:companion) to be stable we condition that the Kp eigenvalues λ that satisfy

$$\det(\mathbf{A} - \lambda \mathbf{I}_{Kp}) = 0$$

are all of absolute value less than one. Stability implies that the first and second moments of the VAR(p) process are time-invariant, hence ensuring weak stationarity [@kilian2017structural].

A straight-forward way to deal with stationarity of VARs is to simply ensure that the individual time series entering the system are stationary. This usually involves differencing the time series until they are stationary: for any time series y_i that is integrated of order $I(\delta)$, there exists a δ-order difference that is stationary. An immediate drawback of this approach is the loss of information contained in the levels of the time series. Modelling approaches that take into account conintegration of individual time series can ensure system stationarity and still let individually non-stationary time series enter the system in levels [@hamilton1994time].

Lag order

The VARs lag order p can to some extent be thought of as the persistency of the process: past innovations that still affect outcomes in time t happened at most p periods ago. From a pure model selection perspective we can also think of additional lags in terms of additional regressors that add to the model's complexity. From that perspective, choosing a lower lag order corresponds to a form of regularization as it pertains to a more parsimonious model.

Various strategies have been proposed to estimate the true or optimal lag order p empirically [@kilian2017structural]. Among the most common ones are sequential testing procedures and selection based on information criteria. The former involves sequentially adding or removing lags - **bottom-up** and **top-down** testing, respectively - and then testing model outcomes in each iteration. A common point of criticism of sequential procedures is that the order tests matters (@lutkepohl2005new).

Here we will focus on selection based on information criteria, which to some extent makes the trade-off between bias an variance explicit [@kilian2017structural]. In particular, it generally involves minimizing information criteria of the following form

$$C(m) = \log(\det(\hat{\Sigma}(m))) + \ell(m)(\#eq:ic) \tag{15}$$

where $\hat{\Sigma}$ is just the sample estimate of the covariance matrix or errors and ℓ is a loss function that penalizes high lag orders. In particular, we have that our best estimate of the optimal lag order p is simply

$$\hat{p} = \arg \min_{m \in \mathcal{P}} C(m)(\#eq:ic-min) \tag{16}$$

where $\mathcal{P} = [m_{\min}, m_{\max}]$. We will consider all of the most common functional choices for @ref(eq:ic).

Neural Network Architecture

As mentioned above, a NN is therefore charaterized for having a very large number of parameters and therefore making it relatively easy to overfit the data. This is a concern specially when the NN is trained in a relatively small dataset. In time series the amount of data to train the model is quite limited, therefore having so many parameters may result in the NN learning the noise of the function, potentially leading to overfitting which in turn results in failry poor performance results of the model in the test set. Therefore some regularization must be added into the model in order to avoid overfitting.

As proposed in @srivastava2014dropout, one way to reduce overfitting can be fitting a lot of NN in the data and the averaging its predictions. This is not feasible due to computational problems. What one can do instead is to randomly drop some units in each layer. This is also known as dropout, this technique simulates the training of multiple NN with different architectures in the same data set. By using dropout, we add some noise into the model. This avoids a certain layer trying to adapt to a mistake made by a previous one, hence potentially leading to complex adaptations which results in overfitting. Dropout therefore can be used as a regularization technique that makes the model more robust.

The way we implement dropout in the Deep VAR model is by specifying the probability to which a node is removed from the layer. This means that the information that arrives to that unit is forgotten.

As another effort to avoid overfitting, we use the same architecture for each layer of the NN. That is, we do not allow each layer to have different number of units, nor different activation functions nor different dropout rates, which could potentially result in the model learning the noise. In turn the Deep VAR model is characterized for having equally sized layers of size N.

```r
knitr::include_graphics("www/nn.png")
```

Figure @ref(fig:nn-arch) shows the neural network architecture for the case of two lags ($p = 2$) and four variables ($K = 4$). We can see that the first layer corresponds to the inputs, that is, the input layer $\in \mathbb{R}^{Kp \times 1}$. This architecture consists of two hidden layers each made of fourteen units. Given that we are aiming at prediciting four variables the output layer consists of four nodes. The edges illustrate the flow of information of the network, where each edge has an associated weight.

Finally, with respect to how the LSTMs underlying the Deep VAR are compiled, the Adam optimization algorithm @kingma2014adam is used. This algorithm can be used instead of the more traditional stochastic gradient descent to update network weights. There are several reasons to use this algorithm that are particulary appealing, among them we have its straighforward implementation, that it is computationally efficient and that the hyper-parametrization has an intuitive interpretation and typically requires little tuning. Adam is different to classic stochastic gradient descent given that in the latter learning rates and always the same for all weight updates and hence does not change during the training.

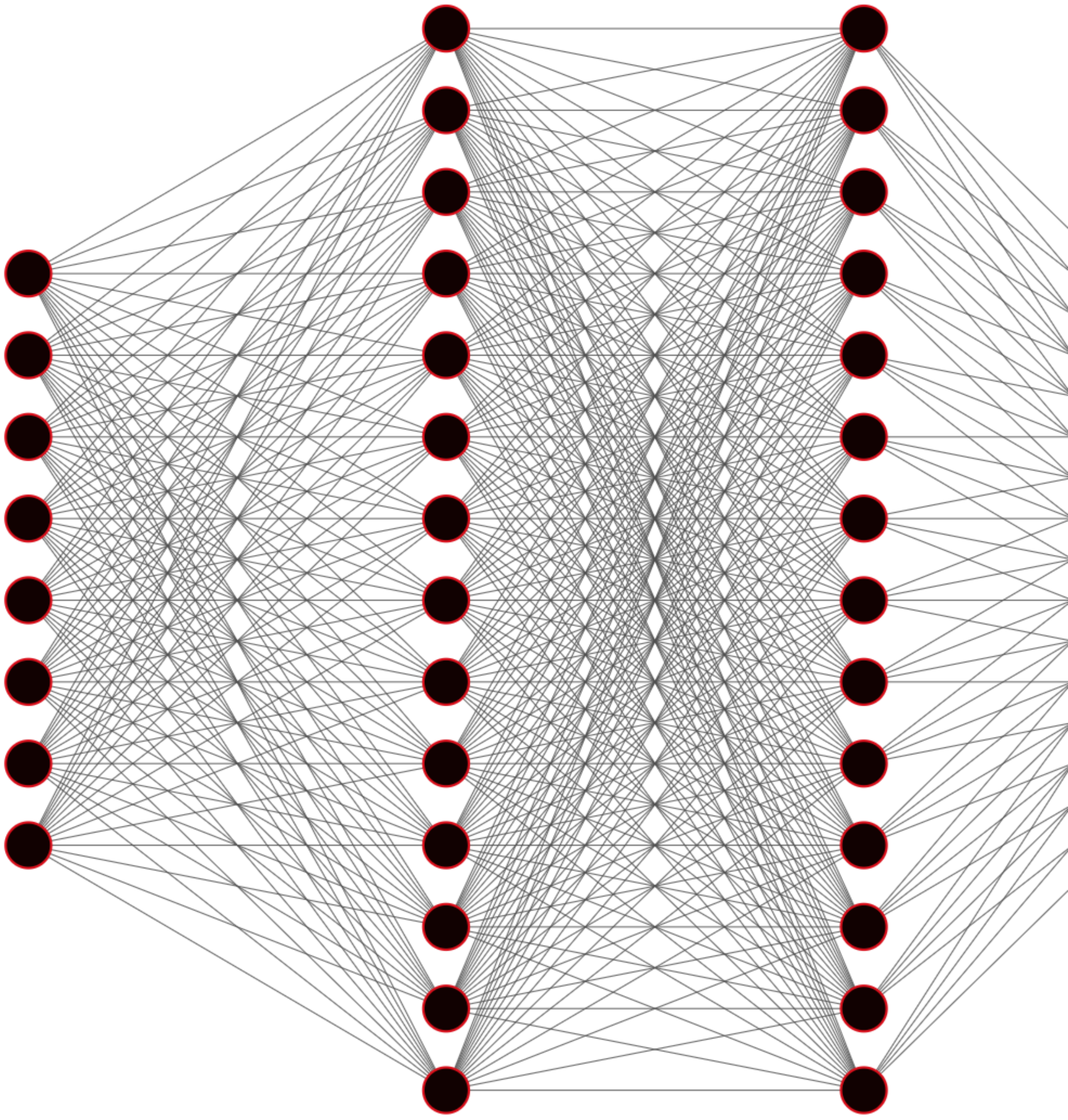

Figure 1: Neural Network Architecture.