Recurrent Neural Networks

In a **recurrent neural network**, or **RNN**, the input is a sequence. Thus the order in which inputs appear matters.

Examples include any sort of economic or financial time series data, book and movie reviews, newspaper articles, tweets, recorded or transcribed speech, videos, and handwritten documents.

The output could also be a sequence, such as a translated document. But it could just be a scalar, like the sentiment of a movie review.

Figure 19.1 from ISLR/ISLP illustrates a very simple RNN with just one hidden layer.

In the example, there are three sets of weights: $\boldsymbol{B}$ contains the weights for the output layer, $\boldsymbol{W}$ contains the shared weights for the input layer, and $\boldsymbol{U}$ contains the weights for the hidden-to-hidden layers.

Figure 19.1 A recurrent neural network with one hidden layer

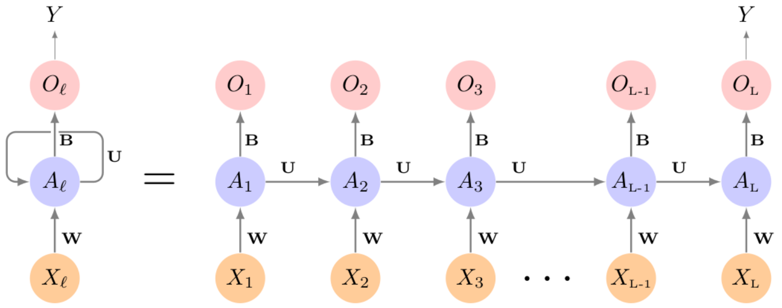


FIGURE 10.12. *Schematic of a simple recurrent neural network. The input is a sequence of vectors* $\{X_\ell\}_1^L$*, and here the target is a single response. The network processes the input sequence* X *sequentially; each* X_ℓ *feeds into the hidden layer, which also has as input the activation vector* $A_{\ell-1}$ *from the previous element in the sequence, and produces the current activation vector* A_ℓ*. The same collections of weights* $\mathbf{W}$, $\mathbf{U}$ *and* $\mathbf{B}$ *are used as each element of the sequence is processed. The*

For the RNN in the figure, we can write the activations as

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{\ell k} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right), \tag{1}$$

where $g(\cdot)$ is ReLU or some other activation function.

The output is given by

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}. \tag{2}$$

For classification, there would then be a logistic function to transform O_ℓ into a probability.

This model involves **weight sharing**. The same weights are used at each step. It is the $A_{\ell k}$ that take history into account.

The intermediate outputs O_ℓ for $\ell < L$ may or may not be of interest.

Often, there will be many input sequence/response pairs indexed by $i = 1, \ldots, n$.

This is similar to a panel, where n is the number of cross-sectional units and L is the number of time periods.

For simplicity, ISLR/ISLP treats L as the same for each sequence. However, movie reviews are not all the same length. But see below.

For the regression case, we minimize

$$\mathrm{SSR} = \sum_{i=1}^{n} (y_i - O_{iL})^2. \tag{3}$$

For the classification case, we would instead maximize a loglikelihood, or minimize a deviance.

RNNs are more expensive to estimate than feed-forward NNs.

Sequential Models for Document Classification

Recall that, for bag-of-words, every document used a vector with 10,000 elements (the number of words in our dictionary).

But now that order matters, it would be extremely inefficient to store a vector of 10,000 elements for every word in each document.

Instead, we use a matrix $\boldsymbol{E}$ of dimension $m \times 10,000$, where m is the **embedding dimension**.

Each column represents a word, and each row gives the m coordinates of that word in the **embedding space**.

We can either learn $\boldsymbol{E}$ as part of training the neural network, or we can use a precomputed $\boldsymbol{E}$, which is known as **weight freezing**.

The pretrained embeddings, `word2vec` and `GloVe`, are widely used.

How this works is illustrated in Figure 19.2. The weights are like factor loadings for principal components.

Figure 19.2 A sequence of 20 words

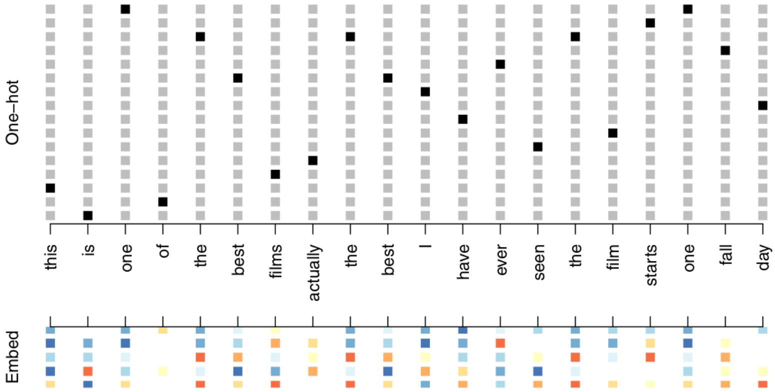


FIGURE 10.13. *Depiction of a sequence of 20 words representing a single document: one-hot encoded using a dictionary of 16 words (top panel) and embedded in an m-dimensional space with $m = 5$ (bottom panel).*

A more realistic version of Figure 19.2 would have 10,000 rows in the top panel, instead of 16.

The embedding dimension, m, would be a lot larger than 5. In the IMDb case, ISLR/ISLP uses 32.

- Since over 91% of movie reviews are less than 500 words long, they set $L = 500$, taking only the last 500 words, and padding the beginning with blanks if necessary.
- Instead of each review requiring a 500×10000 matrix of 0s and 1s, it requires a 500×32 matrix of real numbers.

Unfortunately, the simple RNN performs poorly when applied to the IMDb data. Its accuracy is only 76%, versus 88% for lasso using bag-of-words.

This involved estimating the embedding matrix. Using the pretrained $\boldsymbol{E}$ from `GloVe` worked slightly worse.

So this form of RNN did not perform well.

A more sophisticated type of RNN combines **long-term** and **short-term** memory.

There are two tracks of hidden-layer activations.

One track works as in Figure 19.1, and the other connects activations that are further apart.

This is called an **LSTM RNN**.

Using an LSTM RNN, accuracy increases to 87%. But that is still not quite as good as lasso.

Training LSTM models takes a lot of computer time and human effort.

As of today, the best models achieve accuracy of just over 96% on the IMDb dataset. They do not seem to be LSTMs. See

`https://paperswithcode.com/sota/sentiment-analysis-on-imdb`

Forecasting Time Series

ISLR/ISLP uses a rather odd model to illustrate the use of RNNs for forecasting time series. They use stock-market data.

They have 6051 daily observations on log trading volume, v_t, log DJIA return, r_t, and log volatility (based on absolute price movements), z_t.

The time period is Dec. 3, 1962 to Dec. 31, 1986. Thus they avoid having to deal with Oct. 19, 1987!

Predicting returns is very hard, so they predict trading volume.

An econometrician would probably just regress v_t or changes in v_t on lagged values of all three series for just a few lags.

If desired, regularization could be used, perhaps by using explicitly Bayesian methods.

Trading volume (like volatility) is not independent across time. Figure 19.3 shows the autocorrelation function. Long memory?

Figure 19.3 The autocorrelation function for log trading volume

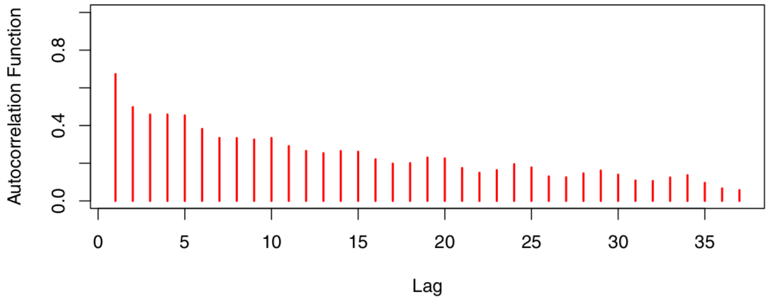


FIGURE 10.15. *The autocorrelation function for* `log_volume`. *We see that nearby values are fairly strongly correlated, with correlations above* 0.2 *as far as 20 days apart.*

For their RNN, the target is $y = v_t$, and the inputs are

$$\boldsymbol{x}_\ell = \begin{bmatrix} v_{t-\ell} \\ r_{t-\ell} \\ z_{t-\ell} \end{bmatrix}, \quad \ell = 1, \ldots, L. \tag{4}$$

For no apparent reason, ISLR/ISLP uses $L + 1 - \ell$ instead of ℓ as the subscript of $\boldsymbol{x}_\ell$.

They use $L = 5$, allowing them to create 6046 $(\boldsymbol{x}, y)$ pairs.

They split the sample after 1979, so they have 4281 training observations and 1770 test ones.

Because of the temporal dependence, it would not make sense to split the sample randomly. But we have to assume that the DGP did not change between the two time periods.

Big changes in the way markets operated happened after 1986. Among other things, bid-ask spreads became vastly smaller.

A conventional autoregression, again with $L = 5$, has 16 coefficients, and achieves a test R^2 of 0.41.

The RNN model achieves a test R^2 of 0.42. It has 205 parameters.

It would not surprise me if some combination of searching over L, imposing restrictions, and ridge-type regularization would make the AR model work better.

They also tried a nonlinear AR model, using a standard feed-forward neural network. It also achieved $R^2 = 0.42$.

Adding day-of-week dummies gets the linear AR and RNN models to 0.46 and the nonlinear AR model to 0.47.

Details of how they did all this are in Section 10.9.6. They used `lm` for the linear AR model and functions from the `keras` package for the neural networks.

It is possible to build a convolutional NN into an RNN, treating the sequence of inputs (say, words) as an image.

The convolution filter slides along the sequence in one dimension.

There can be additional hidden layers, with the sequence A_ℓ feeding into the next layer.

An RNN can be bidirectional, scanning in both directions.

Language translation involves both an input sequence and a target sequence. In **Seq2Seq** learning, the hidden units may capture semantic meaning.

Fitting RNNs can be complicated and computationally costly.

But machine translation has improved massively as a result of using RNNs. Google Translate is the best known, but there are others, notably DeepL Translator, which claims to be more accurate.

When Should We Use Deep Learning?

Deep learning has worked remarkably well for many types of image classification. Does it make sense to become a radiologist?

It also works well for document classification and language translation. But other tools are cheaper and easier to use, and at least some of them are much easier to interpret.

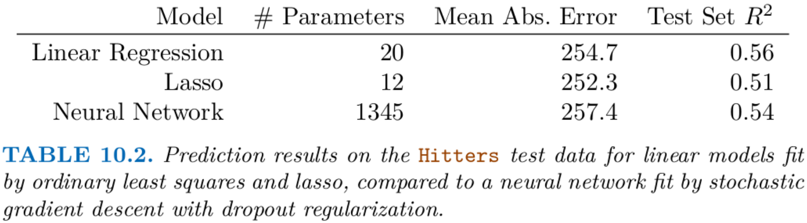
ISLR/ISLP applies several methods to the `Hitters` data. The training set has 176 observations, and the test set 87.

See table on next slide.

The NN had mean absolute test error of 257.4, versus 254.7 for linear regression, 252.3 for lasso, and 224.8 for relaxed lasso.

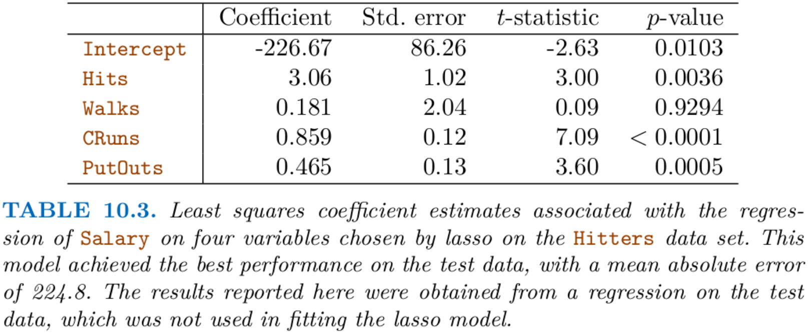
However, the NN in the Python version achieves a mean absolute test error of 229.5, so it works almost as well as relaxed lasso.

Figure 19.4



Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1345	257.4	0.54

TABLE 10.2. *Prediction results on the* `Hitters` *test data for linear models fit by ordinary least squares and lasso, compared to a neural network fit by stochastic gradient descent with dropout regularization.*



	Coefficient	Std. error	t-statistic	p-value
`Intercept`	-226.67	86.26	-2.63	0.0103
`Hits`	3.06	1.02	3.00	0.0036
`Walks`	0.181	2.04	0.09	0.9294
`CRuns`	0.859	0.12	7.09	< 0.0001
`PutOuts`	0.465	0.13	3.60	0.0005

TABLE 10.3. *Least squares coefficient estimates associated with the regression of* `Salary` *on four variables chosen by lasso on the* `Hitters` *data set. This model achieved the best performance on the test data, with a mean absolute error of 224.8. The results reported here were obtained from a regression on the test data, which was not used in fitting the lasso model.*

Fitting Neural Networks

For regression models, we typically minimize the SSR.

This may not be easy, because there are often many parameters, and the SSR function is generally not convex.

These problems occur even for the simplest feed-forward networks with just one hidden layer.

They tend to become worse as the number of hidden layers increases, and for convolutional and recurrent neural networks.

- Because there are so many parameters, overfitting is bound to occur if we do not take steps to prevent it.
- The two main approaches to preventing overfitting are **slow learning** and regularization.
- With slow learning, we stop the minimization process when overfitting is detected.

The basic idea of **gradient descent** algorithms is simple. The details are not at all simple.

We can combine all the parameters (the weights) into a long vector $\boldsymbol{\theta}$ and write the objective function as

$$R(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{\theta}, \boldsymbol{x}_i)\right)^2. \tag{5}$$

Here $f(\boldsymbol{\theta}, \boldsymbol{x}_i)$ is a, possibly very complicated, function that maps from the input vector $\boldsymbol{x}_i$ and the parameters $\boldsymbol{\theta}$ to the fit.

- With gradient descent, we want to go downhill at every step. So we need to know how $R(\boldsymbol{\theta})$ changes as $\boldsymbol{\theta}$ changes.
- The gradient of a function at some point, say $\boldsymbol{\theta}_{(0)}$, is the vector of partial derivatives evaluated at $\boldsymbol{\theta}_{(0)}$. The subscript "$(0)$" may indicate the initial value of $\boldsymbol{\theta}$.
- The gradient has as many elements as $\boldsymbol{\theta}$, often many thousands, perhaps millions, sometimes even billions.

Formally, the gradient of R at $\boldsymbol{\theta}_{(0)}$ is defined as

$$\nabla R(\boldsymbol{\theta}_{(0)}) = \frac{\partial R(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{(0)}}. \tag{6}$$

Suppose we start at $\boldsymbol{\theta}_{(0)}$. Then we want to go downhill.

For a small enough **learning rate** ρ, a step in the opposite direction to the gradient must reduce $R(\boldsymbol{\theta})$. This is called **steepest descent**.

If we use steepest descent with learning rate ρ, we go from $\boldsymbol{\theta}_{(0)}$ to

$$\boldsymbol{\theta}_{(1)} = \boldsymbol{\theta}_{(0)} - \rho \nabla R\big(\boldsymbol{\theta}_{(0)}\big). \tag{7}$$

For nonlinear models with modest numbers of parameters, optimization methods based on **Newton's Method** use information about the Hessian $\boldsymbol{H}(\boldsymbol{\theta})$ as well as the gradient.

For a quadratic function, Newton's Method gets to the minimum in just one step. Steepest descent does not.

A Newton step is $\boldsymbol{\theta}_{(1)} = \boldsymbol{\theta}_{(0)} - \rho \boldsymbol{H}^{-1}\big(\boldsymbol{\theta}_{(0)}\big)\nabla R\big(\boldsymbol{\theta}_{(0)}\big)$.

Steepest descent is likely to be slower than Newton's method when $R(\boldsymbol{\theta})$ is approximately quadratic.

But Newton's method is never used with neural nets, because:

- The Hessian $\boldsymbol{H}$ could easily be an enormous matrix.
- Inverting it could be costly and prone to failure.
- If $\boldsymbol{H}$ is not positive definite, a Newton step may not be downhill.

Thanks to the chain rule, finding the gradient is (conceptually) easy.

Also, since $R(\boldsymbol{\theta})$ is a sum, we just need to figure out the gradient for one observation.

Consider the feed-forward NN with one hidden layer, for which

$$R_i(\boldsymbol{\theta}) = \frac{1}{2}\left(y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big)\right)^2. \tag{8}$$

We need the derivatives with respect to all of the β_k and all of the w_{jk}, for $k = 0, \dots, K$. These are

$$\frac{\partial R_i(\boldsymbol{\theta})}{\partial \beta_k} = -\big(y_i - f(\boldsymbol{\theta}, \boldsymbol{x}_i)\big) g(z_{ik}), \tag{9}$$

where $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$, and

$$\frac{\partial R_i(\boldsymbol{\theta})}{\partial w_{kj}} = -\big(y_i - f(\boldsymbol{\theta}, \boldsymbol{x}_i)\big) g'(z_{ik}) x_{ij}. \tag{10}$$

Every element of the gradient depends on the residual $y_i - f(\boldsymbol{\theta}, \boldsymbol{x}_i)$.

Also, every element of the gradient for $\boldsymbol{\beta}$ depends on $g(z_{ik})$, and every element of the gradient for $\boldsymbol{W}$ depends on $g'(z_{ik})$.

We can build up the entire gradient vector from a few relatively simple components.

Since every element of the gradient depends on the residuals via the chain rule, we can think of assigning fractions of each residual to each of the parameters.

This is called **back-propagation**.

Software for estimating neural nets computes the gradient vector analytically using the residuals, a few relatively simple components, and the chain rule.

Remember that $g(\cdot)$ is extremely simple for ReLU and quite simple, with easily computed first derivative, for logistic and tanh functions.

Of course, this gets more complicated as the model acquires more hidden layers or other features, such as recursion.

A more general, and conceptually easier, way to obtain the gradient is to use **numerical derivatives**.

This is not needed, and would be a bad idea, for neural nets. But since it can be handy in other cases, I will say a little bit about it.

Numerical Derivatives

Numerical derivatives can be very useful.

- We can use them for functions that are difficult to differentiate analytically. Maybe they are implicit or involve simulation.
- We can use them to check the accuracy of analytic derivatives, or of a computer program that implements analytic derivatives.
- We can compute them for smooth functions that we cannot write down analytically.

Unfortunately, the characteristics of floating-point arithmetic make it tricky to take derivatives numerically.

Recall that floating point numbers are stored as

$$m \times b^c = \pm\, 0.d_1\, d_2\, d_3 \ldots d_p \times b^c, \tag{11}$$

where m is the mantissa, b is the base, and c is the exponent.

IEEE 754 double precision uses base 2. It has a 53-bit mantissa and an 11-bit exponent (64 bits total).

Its range is, approximately, $10^{\pm 308}$, and it is accurate to almost 16 decimal digits.

IEEE 754 single precision has a 24-bit mantissa and an 8-bit exponent.

There is also a quadruple-precision format with 128 bits, but it can be expensive, and not all languages implement it.

When successive arithmetic operations are carried out, errors build up, especially when numbers of different sizes and signs are added.

For example, consider the expression

$$69,393,121 - 1.0235 - 69,393,120 = -0.0235. \tag{12}$$

If we evaluate this as written, using single precision, we obtain the answer 0, because 69,393,121 $-$ 1.0235 $\cong$ 69,393,120, where "$\cong$" denotes equality in floating-point arithmetic.

Alternatively, we could first subtract the last number from the first, and then subtract the second. This yields -1.0235, because $69,393,121 - 69,393,120 \cong 0$.

Both answers are wrong! The second one is worse than the first.

This illustrates a major problem of floating-point arithmetic. The order of operations matters. It is emphatically not the case that

$$x + (y + z) = (x + y) + z. \tag{13}$$

Different languages may yield different answers.

The same program may yield different answers if compiled with different compilers, or with different compiler options.

Now back to numerical derivatives. Suppose that x is a scalar and $f(x)$ is a smooth function of it.

We wish to find $f'(x_0)$, the first derivative of $f(x)$ at x_0.

By definition,

$$f'(x) = \lim_{h \to 0} \left(\frac{f(x+h) - f(x)}{h} \right). \tag{14}$$

This suggests that we might use the formula

$$f'(x_0) \cong \frac{f(x_0+h) - f(x_0)}{h} \tag{15}$$

for some h that is chosen to be small.

This **one-sided formula** is not a good choice, however.

There are two sources of error in numerical differentiation:

1. **Truncation error** arises whenever $h > 0$, so that (15) is not quite correct unless f' is constant between x_0 and $x_0 + h$.
2. **Roundoff error** arises from the limitations of floating-point arithmetic. Observe that (15) involves subtracting two numbers of very similar magnitude.

As h gets smaller, truncation error shrinks but roundoff error increases. The latter is $O(1/h)$.

We could eliminate truncation error by making h arbitrarily small, but that would make roundoff error very severe.

It is much better to use the **symmetric formula**

$$f'(x_0) \cong \frac{f(x_0+h) - f(x_0-h)}{2h}. \tag{16}$$

We now evaluate f on both sides of x_0, instead of on just one side.

- It is not hard to show that the truncation error for (15) is $O(h)$, but the truncation error for (16) is $O(h^2)$.
- Since roundoff error is $O(1/h)$ in both cases, we want to use a *larger* value of h for (16) than for (15).
- Optimal choice of h depends on the shape of $f(x)$ in the neighbourhood of x_0, and on the proportional error in the floating-point arithmetic.

For IEEE 754 double-precision arithmetic, we probably want to choose h to be somewhere between 10^{-8} and 10^{-6} when using (15).

But we may want to choose h to be somewhere between 10^{-5} and 0.5×10^{-3} when using (16).

More complicated formulae that involve four or more terms are available. These lead to even smaller errors when used with even larger values of h.

It is possible to take second derivatives numerically, but the accuracy problems are even more severe. For example,

$$f''(x_0) \cong \frac{1}{4h^2}\big(f(x_0+2h)+f(x_0-2h)-2f(x_0)\big). \tag{17}$$

This was obtained by taking the symmetric numerical derivative of a symmetric numerical first derivative.

Notice that h^2 appears in the denominator. So h needs to be large, and the potential for roundoff error is extreme.

It would be insane to use numerical derivatives when estimating neural networks.

Just obtaining them would require $2m + 1$ evaluations of the objective function, where m is the number of parameters.

They would be much less accurate than derivatives obtained using back-propagation.

But, if you were writing your own code for the gradient, numerical derivatives might provide an easy way to check that it is correct.

They can be very useful when solving models for which analytic derivatives are unavailable or need to be checked.

- Always use a symmetric formula like (16), and choose h carefully.
- It may be a good idea to try two or more values of h to verify that they yield similar results.
- Make sure that the parameters are scaled so that all derivatives are of similar magnitude.