Chapter 1

Introduction and background

One approach for generating hypotheses about neural computation, is to train recurrent neural networks (RNNs) to mimic the behavior of humans and other animals performing experimental tasks: the inputs to the RNN are time-varying signals representing experimental stimuli and we adjust the parameters of the RNN so its time-varying outputs are the desired behavioral responses. How should we adjust the parameters of the RNN to achieve the desired target outputs? The supervised learning approach is to construct an error function, $E(\boldsymbol{\theta})$, that quantifies the error between the RNN output and the desired output as a function of the RNN parameters $\boldsymbol{\theta}$. We then iteratively update the parameters to minimize the error. We can motivate the form of the parameter updates as follows. Imagine we want to update the parameters by a small amount $\boldsymbol{\epsilon}$. Let $\|\boldsymbol{\epsilon}\| = c$ where c is some small constant. Let's find $\boldsymbol{\epsilon}$ that does the most to immediately minimize the error function, $E(\boldsymbol{\theta}+\boldsymbol{\epsilon})$. Because we are only taking a small step from our current parameter value we can approximate the error function by a Taylor series in powers of $\boldsymbol{\epsilon}$

$$E(\boldsymbol{\theta}+\boldsymbol{\epsilon}) = E(\boldsymbol{\theta}) + \nabla E(\boldsymbol{\theta})^T \boldsymbol{\epsilon} \tag{1.1}$$

and our problem becomes

$$\begin{aligned} \underset{\boldsymbol{\epsilon}}{\text{minimize}} \quad & E(\boldsymbol{\theta}) + \nabla E(\boldsymbol{\theta})^T \boldsymbol{\epsilon} \\ \text{subject to} \quad & \|\boldsymbol{\epsilon}\| = c. \end{aligned} \tag{1.2}$$

We can solve equation (1.2) using Lagrange multipliers and find the $\boldsymbol{\epsilon}$ that minimizes $E(\boldsymbol{\theta}+\boldsymbol{\epsilon})$ is

$$\begin{aligned}\boldsymbol{\epsilon} &= -\frac{c}{\|\nabla E(\boldsymbol{\theta})\|}\nabla E(\boldsymbol{\theta}) \\ &= -\alpha \nabla E(\boldsymbol{\theta}) \end{aligned} \tag{1.3}$$

where $\alpha \equiv c/\|\nabla E(\boldsymbol{\theta})\| > 0$. So heading in the direction of the negative gradient is the parameter update that does the most to immediately minimize the error function. Substituting the expression for $\boldsymbol{\epsilon}$ into equation (1.1) yields

$$E(\boldsymbol{\theta} - \alpha \nabla E(\boldsymbol{\theta})) = E(\boldsymbol{\theta}) - \alpha \|\nabla E(\boldsymbol{\theta})\|^2 < E(\boldsymbol{\theta}) \tag{1.4}$$

and we see the error decreases as desired. To continue to minimize the error function we repeatedly update the parameters of the neural network according to equation (1.3). The parameters at iteration i, $\boldsymbol{\theta}^{(i)}$, are updated to obtain the parameters at iteration $i+1$:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \alpha \nabla E(\boldsymbol{\theta}^{(i)}) \tag{1.5}$$

where α is a positive constant called the learning rate. This update procedure is called gradient descent and is the standard training algorithm for neural networks (Cauchy, 1847; Robbins and Monro, 1951; Schmidhuber, 2015; Bottou et al., 2018). There are two terms in this algorithm, the learning rate and the gradient of the error function. Both of these terms are crucial for successfully training neural networks so we will consider each of them in turn.

1.1 Learning rate

There is no general prescription for finding the best learning rate and we might expect the value to change as learning progresses and the local curvature of the error function changes. A learning rate that is too small decreases the error function very slowly and a learning rate that is too large may cause the error to fluctuate around, but never reach, a minimum, or even diverge and grow (Figure 1.1).

Automatically adjusting the learning rate during training is an area of active research. In practice it is not a single constant but a set of constants, one for each parameter, that are adaptively updated and can increase or decrease according to the history of the previous gradients, taking special account of the magnitude of the gradients, their

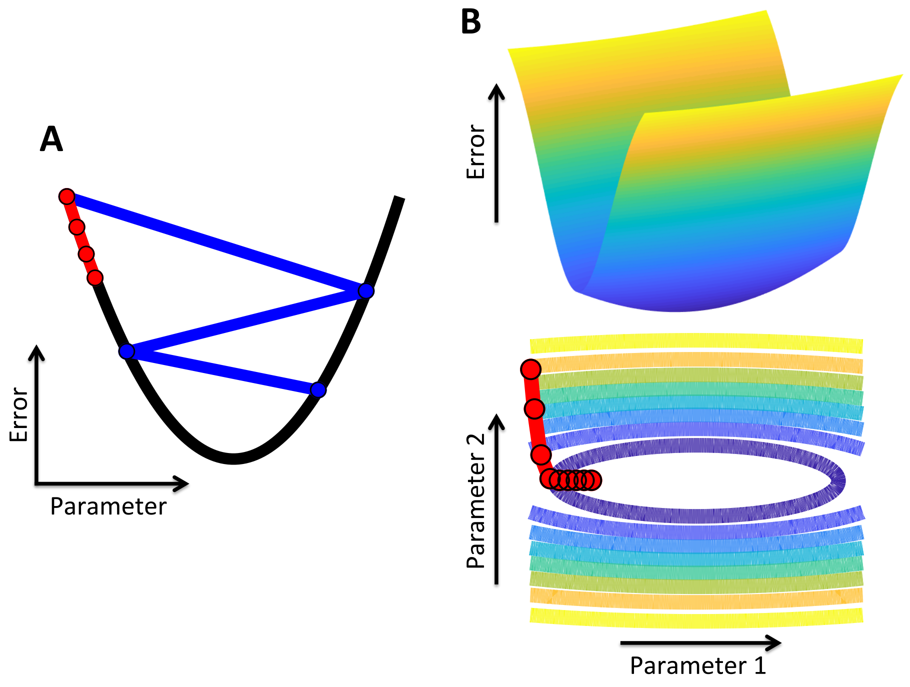

Figure 1.1: Gradient descent on one and two dimensional error surfaces. **(A)** The error (black curve) is a quadratic function of a single parameter. This parameter is varied with gradient descent to find the minimum of the error function, using either a small learning rate (red curve) or large learning rate (blue curve). A learning rate that is too small decreases the error function very slowly and a learning rate that is too large may cause the error to fluctuate around the minimum or even diverge and grow. **(B)** The error (top figure) is a quadratic function of two parameters. The contours of this error function are shown (bottom) superimposed over the parameter values. The red curve shows multiple parameter updates using gradient descent. The gradient is perpendicular to the contour lines so for this highly elongated error function the gradient steps are initially far from the direction of the minimum.

variance, and estimates of the curvature of the error surface (Nesterov, 1983; Qian, 1999; Duchi et al., 2011; Hinton et al., 2012; Zeiler, 2012; Kingma and Ba, 2015; Ruder, 2017). We can gain some intuition into how we should adjust the learning rate based on the shape of the error function by creating a Taylor series around our current parameter value and approximating the error function based on the value, slope, and curvature. The second-order Taylor approximation of the function E near the parameters $\boldsymbol{\theta}$ is

$$E(\boldsymbol{\theta}+\boldsymbol{\epsilon}) = E(\boldsymbol{\theta}) + \nabla E(\boldsymbol{\theta})^T \boldsymbol{\epsilon} + \frac{1}{2}\boldsymbol{\epsilon}^T H \boldsymbol{\epsilon} \tag{1.6}$$

where $H_{ij} = \frac{\partial^2 E}{\partial\theta_i \partial\theta_j}$ is the Hessian matrix of second derivatives. For functions of a single variable there is only one second derivative, and this quantifies how quickly the slope changes or how "curved" the function. For functions of N variables the N eigenvalues of the Hessian summarize the curvature along the direction of their corresponding eigenvector; small/large absolute magnitude of the eigenvalue corresponds to small/large curvature (Figure 1.2) which can either be curved up or down depending on the sign of the eigenvalue.

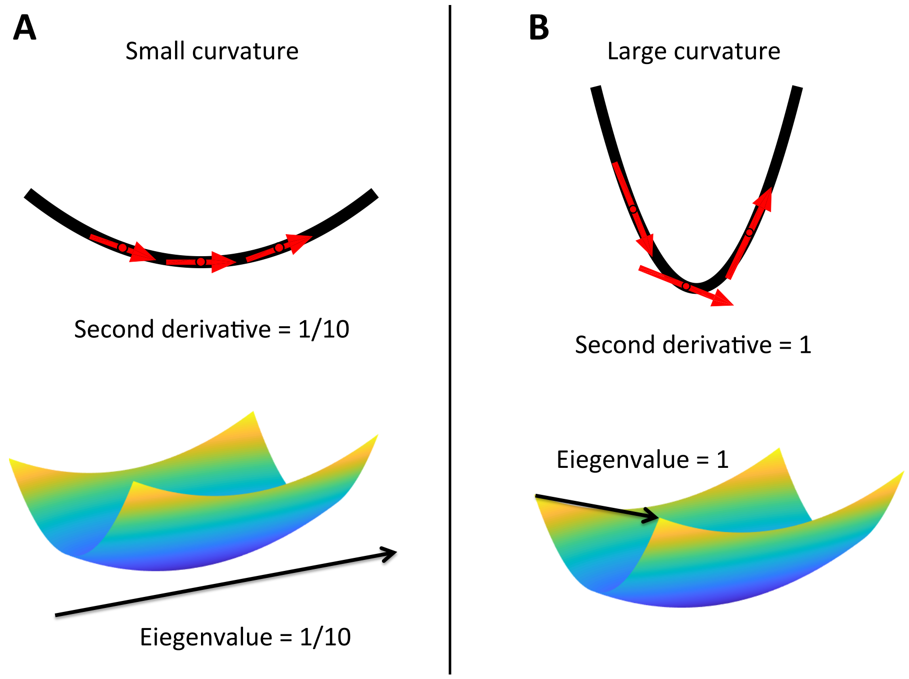

Figure 1.2: The eigenvalues of the Hessian quantify the curvature. In the top row, the slope, or derivative of the error surface, is shown with red arrows. The second derivative is just the rate of change of the slope. In **(A)** this slope changes slowly and the surface has small curvature. In **(B)** the slope changes quickly and the surface has large curvature. For functions of more than one variable the eigenvalues of the Hessian summarize the curvature along the direction of their corresponding eigenvector (black arrows, in the bottom figures). The eigenvalue is the multivariate generalization of the second derivative for one dimensional error surfaces. The bottom row shows an error surface that depends on two parameters, and we see that a small eigenvalue indicates a direction of small curvature and a large eigenvalue indicates a direction of large curvature.

Similar to how we motivated equation (1.5) from the first-order Taylor approximation, we will now find $\boldsymbol{\epsilon}$ that minimizes $E(\boldsymbol{\theta}+\boldsymbol{\epsilon})$ when we use the second-order Taylor approximation, and have access to information about the curvature of the error function. We will see that this allows us to relate the learning rate, α in equation (1.5), to the curvature, or more precisely the eigenvalues of the Hessian. It is intuitive that the curvature should effect the optimal learning rate; for example, when the error function is almost flat and has a very shallow curvature we need to update the parameters a lot to reach a minimum, so the learning rate should be large.To find $\boldsymbol{\epsilon}$ that minimizes $E(\boldsymbol{\theta}+\boldsymbol{\epsilon})$ we will take the gradient of equation (1.6) with respect to $\boldsymbol{\epsilon}$, set the result equal to zero, and then solve. This prescription may not find a sensible $\boldsymbol{\epsilon}$ if $E(\boldsymbol{\theta}+\boldsymbol{\epsilon})$ does not have a minimum, for example, if the error function curves downwards in some directions and there is no minimum. So we will assume the error function curves upwards and thus all eigenvalues of H are greater than zero.

$$\begin{aligned}\nabla_{\boldsymbol{\epsilon}} E(\boldsymbol{\theta}+\boldsymbol{\epsilon}) &= \nabla_{\boldsymbol{\epsilon}}\Big(E(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})^T \boldsymbol{\epsilon} + \frac{1}{2}\boldsymbol{\epsilon}^T H \boldsymbol{\epsilon}\Big) \qquad (1.7)\\ &= \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) + H\boldsymbol{\epsilon} \\ &\overset{\text{set}}{=} 0 \\ \boldsymbol{\epsilon} &= -H^{-1}\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) \qquad (1.8)\end{aligned}$$

Writing equation (1.8) in the form of an iterative algorithm for updating the parameters

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - H^{-1}\nabla E(\boldsymbol{\theta}^{(i)}) \qquad (1.9)$$

we arrive at what is called Newton's method. The Hessian, H, is a symmetric matrix so from the spectral theorem of linear algebra we know that the eigenvectors of H form an orthonormal basis set and we can write any vector as a weighted sum of these eigenvectors. In particular we'll express $\nabla E(\boldsymbol{\theta}^{(i)})$ in terms of the eigenvectors, $\boldsymbol{v_i}$, of H. $H\boldsymbol{v_i} = \lambda_i \boldsymbol{v_i}$ so $H^{-1}\boldsymbol{v_i} = \frac{1}{\lambda_i}\boldsymbol{v_i}$ and

$$H^{-1}\nabla E(\boldsymbol{\theta}) = \sum_i \frac{1}{\lambda_i}\big(\nabla E(\boldsymbol{\theta})^T \boldsymbol{v_i}\big)\boldsymbol{v_i} \qquad (1.10)$$

Equation (1.10) decomposes the parameter update into components along the direction of each eigenvector. The portion of the gradient along the direction of eigenvector $\boldsymbol{v_i}$ is $\big(\nabla E(\boldsymbol{\theta})^T \boldsymbol{v_i}\big)$. To update the parameters along this direction we multiply this gradient by a learning rate of $1/\lambda_i$. In summary, from equation (1.10) we see that if the error function has a minimum and is well described by our second-order Taylor approximation, e.g. the error is a quadratic function whose Hessian has eigenvalues greater than zero, the optimal step size along the direction of an eigenvector is 1/eigenvalue. In fact, if we are minimizing a quadratic function (with a Hessian whose eigenvalues are greater than zero) then the minimum will be reached after a single parameter update from equation (1.9).

If the Hessian has directions of negative curvature then the quadratic approximation does not have a minimum along these directions. However, the real error surface of deep neural networks is not quadratic and so we might expect there will be some maximum step size we can take along these negative curvature directions before the error starts to increase again. Several authors have suggested that $1/|\lambda_i|$ may also be the optimal step size to use in this case as well, under some conditions concerning the extent to which local information generalizes to unseen parts of the error surface (Nocedal and Wright, 2006; Murray, 2010; Dauphin et al., 2014). However, recent empirical work has shown that for deep neural networks the optimal step size along directions of negative curvature is much larger than $1/|\lambda_i|$ (Alain et al., 2018). Interestingly, even though the loss surface of deep neural networks is not quadratic the optimal step size along directions of positive curvature was empirically close to the $1/\lambda_i$ predicted by the quadratic approximation (Alain et al., 2018).

We have seen how information about the curvature can help us select learning rates when the error function is well approximated by a quadratic surface with positive eigenvalues, and these intuitions can be generalized to deep neural networks with potentially negative curvature directions by approximating the Hessian with a positive semi-definite matrix, like the Gauss-Newton or Fisher matrix (Schraudolph, 2002; LeRoux et al., 2007; Martens, 2010; Martens and Sutskever, 2012). However, it is computationally expensive to compute and invert the Hessian or its matrix approximations and so in practice neural networks are most commonly optimized using gradient descent as in equation (1.5). There is some debate about whether the learning rate should be adaptively adjusted for each parameter (using a method which might successively approximate the curvature with a diagonal matrix) or whether better performance is obtained with a single well chosen initial learning rate, decay schedule, and momentum (Hardt et al., 2016; Wilson et al., 2017; Keskar and Socher, 2017). In either case, it is worth revisiting the quadratic function and optimizing this with a single learning rate to see when this is easy or hard and thus understand how we should alter the error surface to improve optimization when complete curvature information is not available.

Let's minimize the quadratic function

$$\mathrm{f}(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + \boldsymbol{c} \qquad \boldsymbol{x} \in \mathbb{R}^N \tag{1.11}$$

where A has positive eigenvalues. This is a simple model but can approximate many functions via its interpretation as the second-order Taylor series from equation (1.6), where the Hessian is approximated with the Gauss-Newton matrix or Fisher information matrix (Amari, 1998; Martens, 2017). We can assume that A is symmetric. If A is not symmetric we could replace it with the symmetric matrix $\overline{A} = (A + A^T)/2$ without changing f($\boldsymbol{x}$) because $\boldsymbol{x}^T A\boldsymbol{x} = \boldsymbol{x}^T \overline{A}\boldsymbol{x}$. To find the minimum of f($\boldsymbol{x}$) we set the derivative equal to zero and obtain

$$\nabla \mathrm{f}(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b} \overset{\text{set}}{=} 0 \tag{1.12}$$

$$\boldsymbol{x}^* = A^{-1}\boldsymbol{b} \tag{1.13}$$

where $\boldsymbol{x}^*$ is the solution that minimizes f($\boldsymbol{x}$). Now let's compare this optimal solution with the approximate solutions found using the gradient descent updates from equation (1.5). For the quadratic function, the parameters at iteration i are updated to obtain the parameters at iteration i+1 as follows.

$$\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \alpha(A\boldsymbol{x}^{(i)} - \boldsymbol{b}) \tag{1.14}$$

The evolution of the parameters is clearer when we project the error between the gradient update and the optimal solution onto the eigenvectors of the Hessian matrix, A. We denote the mth eigenvector by $\boldsymbol{v_m}$ so $A\boldsymbol{v_m} = \lambda_m \boldsymbol{v_m}$ and

$$\begin{aligned}\boldsymbol{v}_{\boldsymbol{m}}^T(\boldsymbol{x}^{(i)} - \boldsymbol{x}^*) &= \boldsymbol{v}_{\boldsymbol{m}}^T\big(\boldsymbol{x}^{(i-1)} - \alpha(A\boldsymbol{x}^{(i-1)} - \boldsymbol{b}) - \boldsymbol{x}^*\big)\\ &= (1-\alpha\lambda_m)\boldsymbol{v}_{\boldsymbol{m}}^T(\boldsymbol{x}^{(i-1)} - \boldsymbol{x}^*)\\ &= (1-\alpha\lambda_m)^i\,\boldsymbol{v}_{\boldsymbol{m}}^T(\boldsymbol{x}^{(0)} - \boldsymbol{x}^*)\end{aligned} \tag{1.15}$$

The error along each eigenvector evolves independently and decreases by a factor of $(1-\alpha\lambda_m)$ after each parameter update. Summing over the components of the error along each eigenvector yields

$$\boldsymbol{x}^{(i)} - \boldsymbol{x}^* = \sum_m (1-\alpha\lambda_m)^i\,\boldsymbol{v}_{\boldsymbol{m}}^T(\boldsymbol{x}^{(0)} - \boldsymbol{x}^*)\boldsymbol{v_m} \tag{1.16}$$

Notice that if we use a separate learning rate along the direction of each eigenvector, $\alpha_m = 1/\lambda_m$, then the parameters, $\boldsymbol{x}^{(i)}$, reach the minimum after a single update, and we recover the previous result. However, if we use the same learning rate for all parameters then not all terms in this sum will decrease at the same rate (assuming all eigenvalues are not the same) and we will only reach the optimal solution after repeated parameter updates. For most values of the learning rate the error along eigenvector directions of high curvature (large eigenvalues) will decrease first, followed by a period of slow convergence as the error along the small eigenvalue directions is minimized. The error between the minimum value of the quadratic function and the value after parameter update $\boldsymbol{x}^{(i)}$ is

$$f(\boldsymbol{x}^{(i)}) - f(\boldsymbol{x}^*) = \frac{1}{2}\sum_m (1-\alpha\lambda_m)^{2i} \, (\boldsymbol{v}_m^T(\boldsymbol{x}^{(0)} - \boldsymbol{x}^*))^2 \, \lambda_m \tag{1.17}$$

In order for the error to decrease for any value of $\boldsymbol{x}^{(0)}$ the learning rate must be within some bounds, namely, $|1-\alpha\lambda_m| < 1$ or

$$0 < \alpha < \frac{2}{\lambda_m} \qquad \text{for all } m \tag{1.18}$$

If $|1-\alpha\lambda_m|$ is close to zero then the error along this direction will decrease quickly, and if this term is near one the error will decrease slowly. The magnitude of $|1-\alpha\lambda_m|$ controls the speed of convergence. In order to reach the optimal solution all components of the error must decrease, so the speed of convergence is limited by the term with the maximum value of $|1-\alpha\lambda_m|$ (assuming the components of the error along each eigenvector are initially nonzero). Therefore, in order to ensure the worst-case convergence is as fast as possible, we'll choose α such that no values of $|1-\alpha\lambda_m|$ are large, i.e. α minimizes

$$\begin{aligned}\max_{1\le m\le N} |1-\alpha\lambda_m| &= \max\left\{|1-\alpha\lambda_{\max}|, |1-\alpha\lambda_{\min}|\right\} \\ &= \max\begin{cases} 1-\alpha\lambda_{\min}, & 0<\alpha<\frac{2}{\lambda_{\min}+\lambda_{\max}} \\ \alpha\lambda_{\max}-1, & \frac{2}{\lambda_{\min}+\lambda_{\max}} \le \alpha < \frac{2}{\lambda_{\max}} \end{cases}\end{aligned} \tag{1.19}$$

where $\lambda_{\max}$ denotes the largest eigenvalue and $\lambda_{\min}$ denotes the smallest. This expression defines two line segments: $1-\alpha\lambda_{\min}$ has a maximum value of 1 when $\alpha = 0$ and then slopes downwards as α increases. $\alpha\lambda_{\max}-1$ has a maximum value of 1 when $\alpha = 2/\lambda_{\max}$ and decreases as α decreases. The optimal constant learning rate in this setting is the value of α that minimizes expression (1.19) and is

$$\alpha^* = \frac{2}{\lambda_{\min}+\lambda_{\max}} \tag{1.20}$$

(Elman and Golub, 1994; Yuan, 2008; Goh, 2017). With this learning rate the upper bound on the speed of convergence is

$$\max_{1\le m\le N} |1-\alpha^*\lambda_m| = \frac{\lambda_{\max}/\lambda_{\min}-1}{\lambda_{\max}/\lambda_{\min}+1} \tag{1.21}$$

From equation (1.21) we see that it is not small or large curvature that makes learning with gradient descent difficult, rather, it is unequal curvature along different directions. When $\lambda_{\max}/\lambda_{\min} \gg 1$ equation (1.21) approaches one and the error along some directions will decrease very slowly.

The error surface of a deep neural network is more complicated than this quadratic example but these results still inform research intuitions and practical recommendations. For example, to encourage uniform curvature along different directions of the error surface the inputs are adjusted, e.g. each component of the input vector is transformed so it has zero mean and unit variance over the whole training set (Hinton et al., 2012). More generally, many of the recent advances in training neural networks are thought to improve optimization by smoothing the error surface or biasing parameter optimization to regions with more uniform curvature (Im et al., 2016; Hardt and Ma, 2017; Li et al., 2017; Karras et al., 2018; Santurkar et al., 2018; Rahaman et al., 2018; Xing et al., 2018).

1.2 Backpropagation through time and real-time recurrent learning

To find the parameters that minimize our error function using gradient descent from equation (1.5), we must calculate the derivative of the error function with respect to the parameters of our model. We can rely on the definition of the derivative and use the method of finite differences to obtain the partial derivative of E with respect to the ith parameter $\boldsymbol{\theta_i}$:

$$\frac{\partial E}{\partial \boldsymbol{\theta_i}} = \lim_{\epsilon \to 0} \frac{E(\boldsymbol{\theta_i} + \epsilon) - E(\boldsymbol{\theta_i})}{\epsilon} \tag{1.22}$$

where ϵ is some infinitesimally small number. However, this naive method is inefficient and slow as we must perturb each parameter and then calculate $E(\boldsymbol{\theta_i} + \epsilon)$. Passing the training data through the model and calculating the error, E, is called the *forward pass*. If we have N parameters and use equation (1.22) to estimate the gradient, $\nabla E(\boldsymbol{\theta})$, we will need to make N forward passes to calculate the perturbations of each parameter and a final forward pass to calculate the baseline error $E(\boldsymbol{\theta})$ at our current parameter values. We can reduce the computational load by perturbing the activities of the *units* in the neural network instead of the *weights*. Once we know how we want a unit to change we can *compute* how to change the weights. A typical neural network has fewer units than weights and so the number of forward passes required to compute $\partial E / \partial \text{unit}_i$ is less than the number required for $\partial E / \partial \boldsymbol{\theta}_i$. However, we can increase the efficiency of the gradient calculation even more. In a standard neural network each parameter, or unit activity, does not have an independent effect on the error function and so we can use a method called backpropagation that requires only a single forward pass and another pass of comparable speed, called the backward pass.

To demonstrate how to efficiently calculate derivatives of the error function with respect to the parameters, and highlight some of the associated problems, we will consider a simple recurrent neural network defined by the following equations.

$$\boldsymbol{h}(t) = W^{\text{rec}}\sigma\big(\boldsymbol{h}(t-1)\big) + W^{\text{in}}\boldsymbol{x}(t) \tag{1.23}$$

$$\boldsymbol{y}(t) = W^{\text{out}}\boldsymbol{h}(t) \tag{1.24}$$

where $\boldsymbol{h}(t)$ is a vector denoting the activity of the units in the network at time t. The hidden units in the network at time t receive input from other units at time $t-1$ through the recurrent weight matrix W^{rec} and also receive external input, $\boldsymbol{x}(t)$, that enters the network through the matrix W^{in}. σ is a nonlinearity, e.g. tanh, applied to each element of the vector $\boldsymbol{h}(t)$. The output of the network, $\boldsymbol{y}(t)$, is a linear sum of the unit activities via the matrix W^{out}. To optimize the parameters, W^{rec}, W^{in}, and W^{out} we need to quantify the error between the network outputs, $\boldsymbol{y}(t)$, and the target outputs, $\boldsymbol{y}^{\text{target}}(t)$. For simplicity we will assume we only care about the network output at the last timestep T. So if we use the squared difference to quantify the error we have

$$E = \frac{1}{2}\sum_{j=1}^{N_{\text{out}}}\big(\boldsymbol{y}_j(T) - \boldsymbol{y}_j^{\text{target}}(T)\big)^2 \tag{1.25}$$

where the sum is over all N_{out} outputs.

To calculate the derivatives it is helpful to see how variations in different variables propagate through the network and how they contribute to the error function by unrolling the network activity over time as shown in Figure 1.3. Figure 1.3A shows the variables in the recurrent neural network defined by equations (1.23) and (1.24). Figure 1.3B shows the same network at each timestep. The arrows indicate dependencies between variables, and are drawn when one variable has a direct effect on another variable. For example, the hidden unit activity at the first timestep, $\boldsymbol{h}(1)$, only effects the hidden unit activity at later timesteps by altering the values of $\boldsymbol{h}(2)$.

To update the parameters, W^{in}, W^{rec}, and W^{out} with gradient descent we must calculate the gradient of the error function with respect to each of these parameters. The calculation is most straightforward for W^{out} so we will start with this. Variations in W^{out}

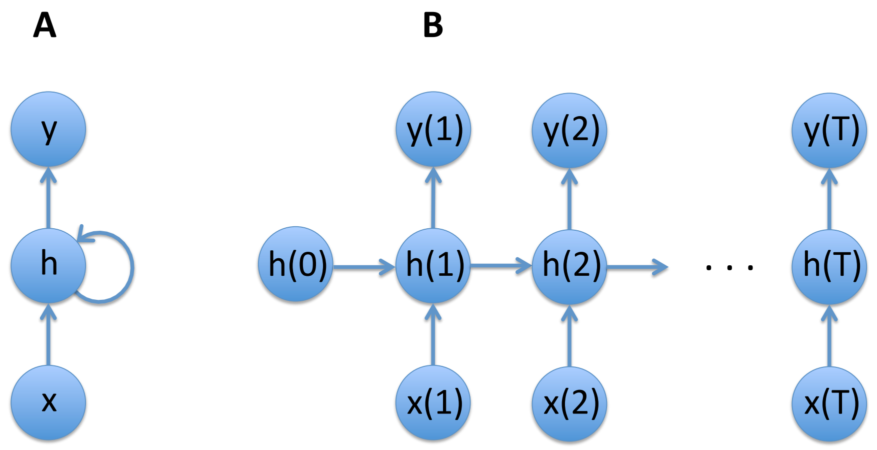

Figure 1.3: **(A)** Recurrent neural network **(B)** Recurrent neural network unrolled in time: time is represented spatially as a new layer in a deep feedforward network with a new layer at each timestep. Arrows are drawn when one variable has a direct effect on another variable. For example, the hidden unit activity at the first timestep, $\boldsymbol{h}(1)$, only effects the hidden unit activity at later timesteps through $\boldsymbol{h}(2)$.

only effect E through $\boldsymbol{y}(T)$ so

$$\begin{aligned}\frac{\partial E}{\partial W^{\text{out}}_{ij}} &= \sum_{k=1}^{N^{\text{out}}} \frac{\partial E}{\partial \boldsymbol{y}_k(T)} \frac{\partial \boldsymbol{y}_k(T)}{\partial W^{\text{out}}_{ij}} \\ &= \sum_{k=1}^{N^{\text{out}}} \big(\boldsymbol{y}_k(T) - \boldsymbol{y}^{\text{target}}_k(T)\big)\delta_{ki}\boldsymbol{h}_j(T) \\ &= \big(\boldsymbol{y}_i(T) - \boldsymbol{y}^{\text{target}}_i(T)\big)\boldsymbol{h}_j(T) \end{aligned} \tag{1.26}$$

where we used equations (1.24) and (1.25) to compute the partial derivatives, and δ_{ij} is the Kronecker delta which takes a value of 1 when $i = j$ and is 0 otherwise.

To motivate the gradient calculation for W^{in} and W^{rec} consider that the only way variations in these parameters effect the error is through $\boldsymbol{h}(T)$. So

$$\frac{\partial E}{\partial W^{\text{in}}_{ij}} = \sum_{k=1}^{N^{\text{rec}}} \frac{\partial E}{\partial \boldsymbol{h}_k(T)} \frac{d\boldsymbol{h}_k(T)}{dW^{\text{in}}_{ij}} \tag{1.27}$$

$$\frac{\partial E}{\partial W^{\text{rec}}_{ij}} = \sum_{k=1}^{N^{\text{rec}}} \frac{\partial E}{\partial \boldsymbol{h}_k(T)} \frac{d\boldsymbol{h}_k(T)}{dW^{\text{rec}}_{ij}} \tag{1.28}$$

Computing $\partial E/\partial \boldsymbol{h}_k(T)$ is straightforward as we will see. However, we must also compute $d\boldsymbol{h}_k(T)/dW_{ij}$, the change in the hidden unit activity at time T as a result of changing

W_{ij} taking into account the effect that changing this weight has on the entire network trajectory from $t = 1$ through the final timestep. Calculating the derivative of the hidden unit activity with respect to the parameters is an intermediate calculation that can be used to find gradients for any neural network having the form

$$h(t) = \mathrm{f}(h(t-1), x(t), \theta) \tag{1.29}$$

where θ is a parameter and f is an arbitrary function of the unit activity at the previous timestep, $h(t-1)$, and the input $x(t)$. To find $dh(t)/d\theta$ we can differentiate this expression with respect to θ and obtain

$$\begin{aligned} \frac{dh(t)}{d\theta} &= \frac{\partial h(t)}{\partial h(t-1)}\frac{dh(t-1)}{d\theta} + \frac{\partial h(t)}{\partial x(t)}\frac{dx(t)}{d\theta} + \frac{\partial h(t)}{\partial \theta} \\ &= \frac{\partial h(t)}{\partial h(t-1)}\frac{dh(t-1)}{d\theta} + \frac{\partial h(t)}{\partial \theta} \end{aligned} \tag{1.30}$$

where we assume the input $x(t)$ does not depend on θ so $dx(t)/d\theta$ is zero. This is a recursion relation for $dh(t)/d\theta$ in terms of $dh(t-1)/d\theta$. If we assume the initial activity of the network does not depend on θ then the recursion is initialized with

$$\frac{dh(0)}{d\theta} = 0 \tag{1.31}$$

This approach for computing gradients in terms of a recursion for $dh(t)/d\theta$ is called real-time recurrent learning (Robinson and Fallside, 1987; Williams and Zipser, 1989). Using this approach and the definition of $\boldsymbol{h}(t)$ from equation (1.23) to solve our original problem yields

$$\begin{aligned} \frac{d\boldsymbol{h}_k(t)}{dW^{\text{in}}_{ij}} &= \frac{d}{dW^{\text{in}}_{ij}}\left(\sum_m W^{\text{rec}}_{km}\sigma(\boldsymbol{h}_m(t-1)) + \sum_m W^{\text{in}}_{km}\boldsymbol{x}_m(t)\right) \\ &= \sum_m W^{\text{rec}}_{km}\sigma'(\boldsymbol{h}_m(t-1))\frac{d\boldsymbol{h}_m(t-1)}{dW^{\text{in}}_{ij}} \ + \ \delta_{ki}\boldsymbol{x}_j(t) \end{aligned} \tag{1.32}$$

$$\begin{aligned} \frac{d\boldsymbol{h}_k(t)}{dW^{\text{rec}}_{ij}} &= \frac{d}{dW^{\text{rec}}_{ij}}\left(\sum_m W^{\text{rec}}_{km}\sigma(\boldsymbol{h}_m(t-1))\right) \\ &= \delta_{ik}\sigma(\boldsymbol{h}_j(t-1)) \ + \ \sum_m W^{\text{rec}}_{km}\sigma'(\boldsymbol{h}_m(t-1))\frac{d\boldsymbol{h}_m(t-1)}{dW^{\text{rec}}_{ij}} \end{aligned} \tag{1.33}$$

The only other term we need in order to calculate the gradients using equations (1.27) and (1.28) is $\partial E/\partial \boldsymbol{h}(T)$. Variations in $\boldsymbol{h}$ only contribute to the error through $\boldsymbol{y}(T)$ so

$$\frac{\partial E}{\partial \boldsymbol{h}_k(T)} = \sum_m \frac{\partial E}{\partial \boldsymbol{y}_m(T)}\frac{\boldsymbol{y}_m(T)}{\partial \boldsymbol{h}_k(T)} \tag{1.34}$$

$$= \sum_m \left(\boldsymbol{y}_m(T) - \boldsymbol{y}^{\text{target}}_m(T)\right)W^{\text{out}}_{mk} \tag{1.35}$$

Real-time recurrent learning is not commonly used in practice as it is slower than backpropagation through time, which we will cover next. In the recursion relations for $d\boldsymbol{h}/dW$ from equations (1.32) and (1.33) notice that we are calculating the derivative of a vector with respect to a matrix. In the recursion for $d\boldsymbol{h}/dW^{\text{rec}}$ we must store and compute N^3 numbers, where N is the number of hidden units in our network. If we could write a set of recursion relations for a quantity that depended on derivatives of a scalar with respect to a vector we would only have N numbers to store and fewer operations to compute. This is the advantage of backpropagation through time, which uses a recursion relation for $\partial E/\partial \boldsymbol{h}$ (Werbos, 1974, 1982; Rumelhart et al., 1986; Robinson and Fallside, 1987).

It is instructive to compute the gradients using backpropgation through time as this will allow us to see the much-studied vanishing and exploding gradient problem associated with this algorithm (Hochreiter, 1991; Bengio et al., 1994; Pascanu et al., 2012). Similar to the calculation we performed for real-time recurrent learning, we'll first express the derivatives of the error function with respect to W^{in} and W^{rec} in terms of the quantity we will use in the recursion, i.e. $\partial E/\partial \boldsymbol{h}(t)$. Then we will write down a recursion relation, along with the initial condition, to solve for $\partial E/\partial \boldsymbol{h}(t)$. Variations in W^{in}_{ij} and W^{rec}_{ij} only effect E through $\boldsymbol{h}_i(t)$ so

$$\begin{aligned}\frac{\partial E}{\partial W^{\text{in}}_{ij}} &= \sum_t \frac{\partial E}{\partial \boldsymbol{h}_i(t)} \frac{\partial \boldsymbol{h}_i(t)}{\partial W^{\text{in}}_{ij}} \\ &= \sum_t \frac{\partial E}{\partial \boldsymbol{h}_i(t)} \boldsymbol{x}_j(t) \end{aligned} \tag{1.36}$$

$$\begin{aligned}\frac{\partial E}{\partial W^{\text{rec}}_{ij}} &= \sum_t \frac{\partial E}{\partial \boldsymbol{h}_i(t)} \frac{\partial \boldsymbol{h}_i(t)}{\partial W^{\text{rec}}_{ij}} \\ &= \sum_t \frac{\partial E}{\partial \boldsymbol{h}_i(t)} \sigma(\boldsymbol{h}_j(t-1)) \end{aligned} \tag{1.37}$$

To obtain the recursion relation for $\partial E/\partial \boldsymbol{h}(t)$ notice that when $t = T$ variations in $\boldsymbol{h}(T)$ give rise to variations in E only through $\boldsymbol{y}(T)$, so

$$\begin{aligned}\frac{\partial E}{\partial \boldsymbol{h}_i(T)} &= \sum_k \frac{\partial E}{\partial \boldsymbol{y}_k(T)} \frac{\partial \boldsymbol{y}_k(T)}{\partial \boldsymbol{h}_i(T)} \\ &= \sum_k \left(\boldsymbol{y}_k(T) - \boldsymbol{y}^{\text{target}}_k(T)\right) W^{\text{out}}_{ki} \end{aligned} \tag{1.38}$$

This expression contains terms we know and is how we start the recursion. To compute $\partial E/\partial \boldsymbol{h}(t)$ for other values of t notice that when $t < T$ variations in $\boldsymbol{h}(t)$ give rise to variations in E only through $\boldsymbol{h}(t+1)$ so

$$\begin{aligned}\frac{\partial E}{\partial \boldsymbol{h}_i(t)} &= \sum_k \frac{\partial E}{\partial \boldsymbol{h}_k(t+1)} \frac{\partial \boldsymbol{h}_k(t+1)}{\partial \boldsymbol{h}_i(t)} \\ &= \sum_k \frac{\partial E}{\partial \boldsymbol{h}_k(t+1)} W^{\text{rec}}_{ki} \sigma'(\boldsymbol{h}_i(t))\end{aligned} \tag{1.39}$$

Writing this as a single equation for the entire vector $\partial E/\partial \boldsymbol{h}(t)$ yields

$$\frac{\partial E}{\partial \boldsymbol{h}(t-1)} = diag(\sigma'(\boldsymbol{h}(t)))\,(W^{\text{rec}})^{\intercal} \frac{\partial E}{\partial \boldsymbol{h}(t)} \tag{1.40}$$

Equation (1.40) gives the recursion relation to go from time t to time $t-1$. This recursion is backwards in time, starting from $t = T$ and ending at $t = 1$ motivating the name of the algorithm: backpropagation through time. We will consider two limiting cases of equation (1.40) to better understand the behavior when t becomes large. Imagine the nonlinearity σ is actually the identity transformation, i.e. there is no nonlinearity in equation (1.23) and $\sigma(\boldsymbol{h}(t)) = \boldsymbol{h}(t)$. So $\sigma' = 1$ and equation (1.40) becomes

$$\frac{\partial E}{\partial \boldsymbol{h}(t-1)} = (W^{\text{rec}})^{\intercal} \frac{\partial E}{\partial \boldsymbol{h}(t)} \tag{1.41}$$

Let $W^{\text{rec}} = U\Sigma V^{\intercal}$ be the singular value decomposition of W^{rec} and assume all of the singular values, s, are the same for simplicity. Then

$$\left\| \frac{\partial E}{\partial \boldsymbol{h}(t-1)} \right\| = s \left\| \frac{\partial E}{\partial \boldsymbol{h}(t)} \right\| \tag{1.42}$$

where we have used the fact that U and V are orthogonal matrices by the definition of the singular value decomposition and so $UU^{\intercal} = 1$ and $V^{\intercal}V = 1$. For each timestep we go further into the past the norm is multiplied by another factor of s. For example, if there are 100 timesteps in our recurrent neural network then

$$\left\| \frac{\partial E}{\partial \boldsymbol{h}(1)} \right\| = s^{99} \left\| \frac{\partial E}{\partial \boldsymbol{h}(T)} \right\| \tag{1.43}$$

We see that in the linear network, the norm of $\partial E/\partial \boldsymbol{h}(t)$ is stable if the singular values of W^{rec} are 1. This has inspired many initialization and training schemes for nonlinear networks with the goal of keeping the singular values of W^{rec} near 1 (Socher et al., 2013; Saxe et al., 2014; Le et al., 2015; Arjovsky et al., 2016). The vanishing gradient problem occurs when $s < 1$, causing the norm to decrease exponentially towards zero. The gradients in equations (1.36) and (1.37) are a sum of terms containing $\partial E/\partial \boldsymbol{h}(t)$ so, if $s < 1$, the only terms that will meaningfully contribute to the gradient are terms near the final timestep T. Long-term dependencies get a weight that is exponentially smaller in t compared to short-term dependencies, making it impossible to learn relationships between temporally distant events. The exploding gradient problem occurs when $s > 1$, causing the norm to increase exponentially as we move further back in time away from the final timestep T. Pascanu et al. (2012) hypothesized that this exploding gradient corresponds to encounters with a steep mountain-like error surface. They suggested thinking of the error landscape as a series of shallow valleys sloping towards steady error reduction, adjacent to steep mountains of high error. If gradient descent takes a small step onto the mountain, i.e. a region of high curvature, the gradient will explode, kicking the parameters away from the mountain and also far away from the valley, likely increasing the error. The error surface is generally not known a priori and so parameter updates will unavoidably step onto regions of high curvature and the gradient will explode. To prevent the negative consequences of this exploding gradient, individual elements of the gradient vector are often truncated when they exceed some maximum absolute value, or alternatively, the norm of the gradient is rescaled if it exceeds a maximum, before using the gradient vector to update the parameters via gradient descent (Mikolov, 2012; Pascanu et al., 2012).

We have considered the vanishing and exploding gradient problems in the context of a linear recurrent network as it is easy to see how the singular values of W^{rec} influence the gradient. However, the nonlinearity also effects the gradient. For simplicity, consider equation (1.40) when the network only has a single unit. The recurrence relation becomes

$$\frac{\partial E}{\partial h(t-1)} = \sigma'(h(t))\, W^{\text{rec}} \frac{\partial E}{\partial h(t)} \tag{1.44}$$

If there is no nonlinearity then $\sigma'(h(t)) = 1$ and $W^{\text{rec}} = 1$ leads to a stable value for $\partial E/\partial h(t)$. However, in the presence of the nonlinearity, even if $W^{\text{rec}} = 1$, $\partial E/\partial h(t)$ can vanish or explode depending on the slope of the nonlinearity. For saturating nonlinearities like the logistic function or hyperbolic tangent the slope can be very small in some regions causing $\partial E/\partial h(t)$ to vanish. To avoid this problem it is important to initialize the parameters so these nonlinearities are not in their saturating regimes where the derivative is near zero. To mitigate the vanishing gradient problem caused by nonlinearities with slopes less than 1, the rectified linear nonlinearity $\sigma(h) = \max(0, h)$ is often used as this has a derivative of 1 for positive inputs (Jordan, 1986; Hahnloser et al., 2000; Jarrett et al., 2009; Nair and Hinton, 2010; Glorot et al., 2011).

We have discussed the vanishing and exploding gradient problems for the specific neural network model defined by equations (1.23) and (1.24) but another approach for reducing these problems is to alter the architecture of the network by changing these equations. Many architectures have been explored. For example, we can add delays in the network in order to effectively skip from early times to later times (Lin et al., 1996; ElHihi and Bengio, 1996). In the unrolled graph of Figure 1.3 this would correspond to arrows connecting $\boldsymbol{h}(1)$, for example, to $\boldsymbol{h}(T)$ providing a shorter path for propagating gradient information. However, a standard RNN inevitably morphs the unit activity from one timestep to the next, $\boldsymbol{h}(t) = f(\boldsymbol{h}(t-1))$, leading to a degradation of information transfer across time. The most common architectural solution is to simply copy the activity at one timepoint directly to another timepoint, $\boldsymbol{h}(t) = \boldsymbol{h}(t-1)$, preserving the flow of information. This is one of the essential ideas motivating long short-term memory (LSTM) units and other more recent architectural advances (Hochreiter and Schmidhuber, 1997; Cho et al., 2014; He et al., 2015; Srivastava et al., 2015; Zilly et al., 2016). These have proven to be good architectures for performing computations but if we are modeling the brain it seems likely that using components with some of the same weaknesses will force the solutions found by RNNs to more closely match the brain. For example, in the projects described in the rest of this thesis we use RNNs having units with individual time constants far too short to solve the working memory tasks we train the RNNs to solve. This forces the RNNs to use the dynamics of multiple units together to store information and perform the tasks. This is in contrast to solutions we would obtain with a LSTM network, or one of its variants, where information can be stored by a single unit and so no network dynamics would be required.

The specific network model we use in the remainder of this thesis is defined by the following equations. The dynamics of each unit in the network $h_i(t)$ is governed by the standard continuous-time RNN equation:

$$\tau \frac{d\mathrm{v}_i(t)}{dt} = -\mathrm{v}_i(t) + \sum_{j=1}^{N^{\mathrm{rec}}} W_{ij}^{\mathrm{rec}} h_j(t) + \sum_{k=1}^{N^{\mathrm{in}}} W_{ik}^{\mathrm{in}} I_k(t) + b_i + \xi_i(t) \tag{1.45}$$

for $i = 1, \ldots, N^{\mathrm{rec}}$. The activity of each unit, $h_i(t)$, is related to the activation of that unit, $\mathrm{v}_i(t)$, through a nonlinearity which we generally take to be $h_i(t) = \tanh(\mathrm{v}_i(t))$. Each unit receives input from other units through the recurrent weight matrix W^{rec} and also receives external input, $I(t)$, that enters the network through the weight matrix W^{in}. Each unit has two sources of bias, b_i which is learned and $\xi_i(t)$ which represents noise intrinsic to the network and is taken to be Gaussian with zero mean and constant variance. To perform tasks with the RNN we linearly combine the ‘firing rates’ of units in the network and use this as the output. The linear readout neurons, $y_j(t)$, are given by the following equation:

$$y_j(t) = \sum_{i=1}^{N^{\text{rec}}} W_{ji}^{\text{out}} h_i(t) \tag{1.46}$$

This model can be motivated from the dynamics of spiking neurons (Dayan and Abbott, 2001; Shriki et al., 2003; Harish and Hansel, 2015). However, in this case the unit activity must be interpreted as the *positive* firing rate of a neuron and the recurrent weight matrix W^{rec} satisfies Dale's law, i.e. each neuron has either an excitatory or inhibitory effect on all of its postsynaptic targets and so all the elements within a column of W^{rec} have the same sign. This interpretation may be overly restrictive as theoretical and empirical work suggests that unit activities in equation (1.45) can be interpreted as linear combinations of neural firing rates (Mante et al., 2013; Yamins et al., 2014; DePasquale et al., 2016). This interpretation implies there are no positivity constraints on the unit activities and the recurrent weight matrix need not satisfy Dale's law.

The RNN defined by equation (1.45) shares some of the same limitations as the brain, namely, individual units with limited memory and computational capabilities, forcing the RNN to use interactions between multiple units and the collective dynamics of the network to solve problems. This could increase the similarity between the computational mechanisms used by the RNN and those of the brain. However, we might wonder if the RNN defined by equation (1.45) is too limited in its computational capabilities and thus not able to solve certain classes of problems, a priori restricting its applicability for modeling certain aspects of cognition. Fortunately, it is Turing complete for infinite precision states and infinite computation time (Siegelmann and Sontag, 1992, 1994; Siegelmann, 1999; Chen et al., 2017), although see Weiss et al. (2018) for some caveats in more realistic scenarios with finite precision and computing time.