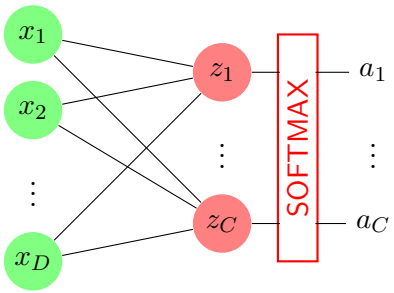
UNIVERSITY OF OXFORD

Machine Learning
Michaelmas Term 2021
Week 8

Problem Sheet 4 Solutions

1 Backpropagation for Multiclass Logistic Regression

We can view multiclass logistic regression as a simple neural network with no hidden layer. Let us suppose that the inputs are D dimensional and that there are C classes. Then the input layer has D units, simply the inputs $x_1, \ldots, x_D$. The output layer has C units; there is full connection between the layers, *i.e.,* every input unit is connected to every output unit. Bias terms for each unit in the output layer are added separately. Finally, there is a non-linear activation function on the output layer, the softmax function. Pictorially, we may represent the neural network as follows:



Dropping superscripts to denote layers, as they are unnecessary in this case, and using the notation in the lectures, we have:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{1.1}$$

$$\mathbf{a} = \text{softmax}(\mathbf{z}) \tag{1.2}$$

Above $\mathbf{W}$ is a $C \times D$ matrix, and $\mathbf{b}$ is a column vector in C dimensions. Recall that the softmax function is defined as:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{l=1}^{C} e^{z_l}} \tag{1.3}$$

For a point $(\mathbf{x}, y)$ in the training data, $y \in \{1, \ldots, C\}$. The model parameters to be learnt are $\mathbf{W}$ and $\mathbf{b}$. We will use the cross entropy loss function, so the contribution of the point $(\mathbf{x}, y)$ to the objective function is given by:

$$\ell(\mathbf{W}, \mathbf{b}; \mathbf{x}, y) = -\log a_y \tag{1.4}$$

Write the expressions for $\frac{\partial \ell}{\partial \mathbf{a}}$, $\frac{\partial \ell}{\partial \mathbf{z}}$, $\frac{\partial \ell}{\partial \mathbf{W}}$ and $\frac{\partial \ell}{\partial \mathbf{b}}$. You should use the backpropagation equations to obtain these derivatives. You should express these derivatives for a single training point $(\mathbf{x}, y)$. When optimising, you'll usually average the gradient over a mini-batch before actually updating

the parameters using a gradient step.

Solution: Let us first compute $\frac{\partial \ell}{\partial \mathbf{a}}$. We have:

$$\frac{\partial \ell}{\partial a_j} = \begin{cases} 0 & \text{if } j \neq y \\ -\frac{1}{a_j} & \text{if } j = y \end{cases} \tag{1.5}$$

And so,

$$\frac{\partial \ell}{\partial \mathbf{a}} = [0, \ldots, -\frac{1}{a_y}, \ldots, 0] \tag{1.6}$$

where the y^{th} coordinate of $\frac{\partial \ell}{\partial \mathbf{a}} = -\frac{1}{a_y}$ and the rest are 0.

Note that,

$$a_j = \frac{e^{z_j}}{\sum_{l=1}^{C} e^{z_l}} \tag{1.7}$$

Thus, we can compute the derivatives

$$\frac{\partial a_j}{\partial z_i} = \begin{cases} -\frac{e^{z_i} e^{z_j}}{\left(\sum_{l=1}^{C} e^{z_l}\right)^2} = -a_i a_j & \text{if } i \neq j \\ \frac{\left(\sum_{l=1}^{C} e^{z_l}\right) e^{z_j} - (e^{z_j})^2}{\left(\sum_{l=1}^{C} e^{z_l}\right)^2} = (1 - a_j) a_j & \text{if } i = j \end{cases} \tag{1.8}$$

Then, by applying chain rule,

$$\begin{aligned} \frac{\partial \ell}{\partial \mathbf{z}} &= \frac{\partial \ell}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \\ &= \left[a_1, a_2, \ldots, a_y - 1, \ldots, a_C\right] \end{aligned} \tag{1.9}$$

Only the y^{th} coordinate of $\frac{\partial \ell}{\partial \mathbf{z}}$ is $a_y - 1$, the i^{th} coordinate for $i \neq y$ is a_i.

Using the backpropagation equations, we have:

$$\frac{\partial \ell}{\partial \mathbf{W}} = \left(\mathbf{x} \frac{\partial \ell}{\partial \mathbf{z}}\right)^{\mathsf{T}} \tag{1.10}$$

$$\frac{\partial \ell}{\partial \mathbf{b}} = \frac{\partial \ell}{\partial \mathbf{z}} \tag{1.11}$$

2 Digit Classification using MLPs

In this problem, we will consider a neural network to classify handwritten digits. You will also use this dataset in your practical this week. In the practical, you will implement a convolutional neural network. Here, we consider networks with fully connected layers and a different way to encode the targets (and the outputs of the network). You are strongly encouraged to implement these networks in tensorflow as well, to help you answer this question.

The inputs are vectors of length 784 (obtained from 28×28 grey pixel images). Rather than output a class, the network will output a vector $\widehat{\mathbf{y}} \in \mathbb{R}^{10}$. The target will be represented as a one-hot encoding, *e.g.,* if the label is 3, we will use the vector $[0,0,1,0,0,0,0,0,0,0]^\mathsf{T}$ (by convention the digit 0 will be the last component in the encoding, not the first).

The data fed into the neural network consists of $\langle(\mathbf{x}_i, \mathbf{y}_i)\rangle$, where $\mathbf{x}_i \in \mathbb{R}^{784}$ and $\mathbf{y}_i \in \{0,1\}^{10}$ is the one-hot encoding of the digit. The output of the neural network is also a vector $\widehat{\mathbf{y}} \in \mathbb{R}^{10}$. The parameters in the network are the weights $\mathbf{W}^i$ and biases $\mathbf{b}^i$ for layers in the network. We'll train the networks by minimizing the squared loss objective (with respect to the parameters of the neural network):

$$\mathcal{L}(\mathbf{W}^i, \mathbf{b}^i) = \sum_{i=1}^{N} (\widehat{\mathbf{y}}_i - \mathbf{y}_i)^2$$

The neural network to be used is shown in Figure 1. There are three layers, one input layer, one *hidden* layer and one output layer. The gradient of the loss function with respect to the parameters can be computed using the backpropagation algorithm as we've seen in the lectures. Answer the following questions:

1. To reduce the size of the network and increase efficiency, your colleague suggests using a binary encoding for the outputs instead of one-hot encoding; so 0 is encoded as 0000, 1 as 0001, and so on. In this case, the output layer only has four neurons rather than 10. Perhaps you could also reduce the number of neurons in the middle layer. What do you think about this suggestion?

   Solution: There is nothing *wrong* with the suggestion as such. However, one might wonder whether there are some unnecessary relationships between the output labels. For example 7 and 5 have embeddings that are closer to each other than 7 and 8. Since our purpose is to have all classes distinguished, we should wonder what effects these embeddings might have. There are several other reasons one could come up with for not preferring binary embeddings; the ultimate test is in trying and seeing if it succeeds or fails. (See (Nielsen, 2015, Chap 1) for some more discussion.)

   The reasons for preferring binary embeddings is simple, they are much more space efficient. So if we could train networks to get high enough accuracy they would be preferred.

2. Show that if you have an already trained neural network that has high accuracy with the *one-hot* encoding, you can design a network that uses binary encoding and achieves roughly the same error. *(Hint: You may need to add an additional layer.)*

   Solution: Denote the network that outputs a one-hot encoding by N. Assuming that exactly one neuron fires in the output layer of N, we can add an additional layer to implement binary embeddings simply by adding "or" gates encoded as sigmoid neurons (Sheet 1). For example the first bit should be 1 if one of neurons corresponding to $1,3,5,7,9$ fired in the previous layer, the second bit should be 1 if one of the neurons corresponding to $2,3,6,7$ fired in the previous layer, and so on.

3. What do you think would happen if you tried to train the neural network you suggested in the previous part directly (rather than adding the last layer by design)?

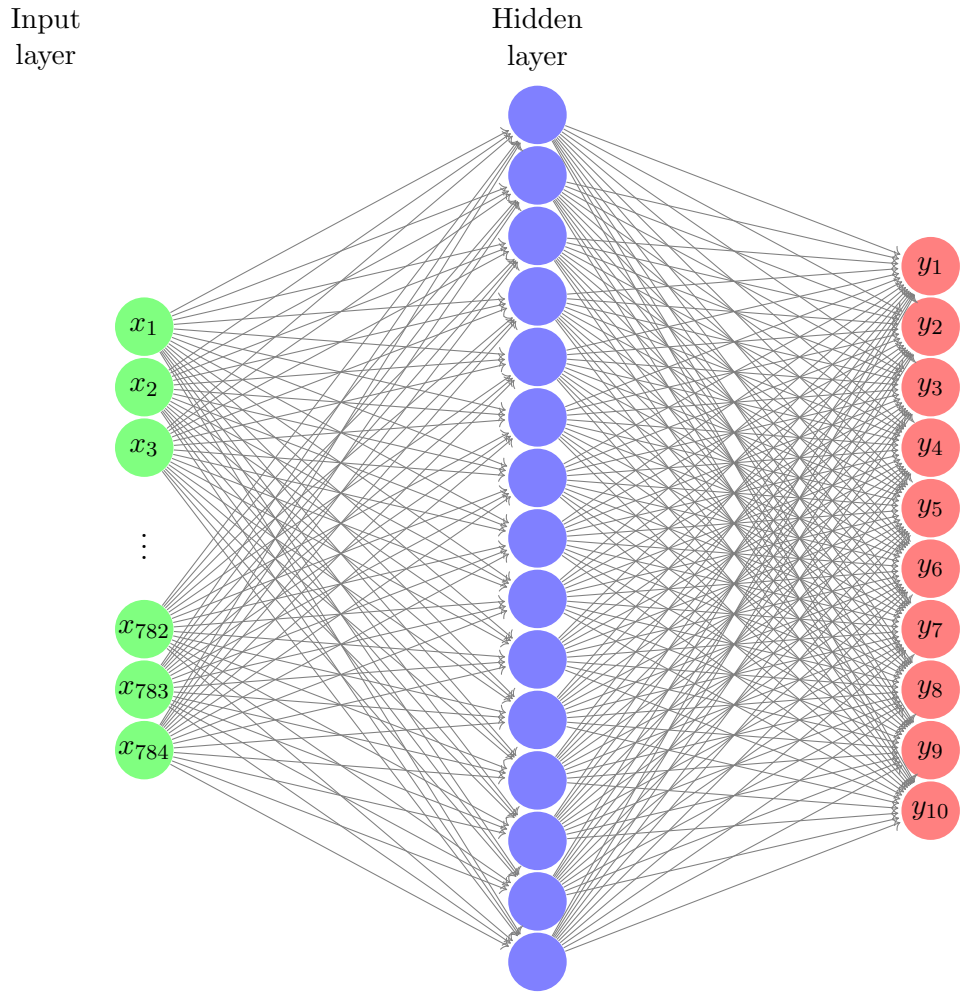


Figure 1: Three-layer neural network for handwritten digit classification.

Solution: The short answer is we have to test and see. With one-hot embedding with one hidden layer of about 100 nodes, I was able to get test accuracy of about 96%. With binary embedding and one hidden layer of 100 nodes, the test accuracy was only 90%. By adding an extra layer of 10 nodes and some regularization I was able to push it up to 93.5%, still somewhat worse than that with one-hot embedding.

As a simple starting point, you can even check what happens with no hidden layer. The one-hot embedding case still gave me something reasonable, close to 90% accuracy. The binary case was much worse, about 75%, though still much better than the 10% accuracy that one would get by random guessing!

3 Neural Nets with Linear Activation Functions

Consider a neural network for multiclass classification. Let us assume that the input is 784 dimensional, we'll us one hidden layer, and there are 10 classes, *e.g.,* digits. For the hidden unit we'll use the "identity" activation function, *i.e.,* $f(x) = x$. Answer the following questions:

1. Suppose you use 25 hidden units and no regularization. For the output layer, we'll use a softmax non-linearity as we did for multiclass logistic regression in Problem 1. An alternative to such a neural network would be to just use the multiclass logistic regression approach directly. Which of the two models is more powerful, *i.e.,* able to represent possibly more complex relationships between inputs and outputs? (For this part, ignore the difficulty of training the models.)

   Solution: Since we are using the identity activation function, we have the following:

   $$\mathbf{z}^2 = \mathbf{W}^2\mathbf{x} \tag{3.1}$$
   $$\mathbf{a}^2 = \mathbf{z}^2 \tag{3.2}$$
   $$\mathbf{z}^3 = \mathbf{W}^3\mathbf{a}^2 = \mathbf{W}^3\mathbf{z}^2 = \mathbf{W}^3\mathbf{W}^2\mathbf{x} \tag{3.3}$$

   Thus, we see that the components of $\mathbf{z}^3$, the pre-activations of the output layer are just a linear function of the input $\mathbf{x}$. Thus, the two models are exactly the same in their representation power.

2. Now suppose you only use 4 hidden units. What can you say about the relative power of the two models? Which model would you prefer?

   Solution: Let us focus on Eq. (3.3). When the hidden layer has only 4 units, $\mathbf{W}^2$ is a 4×784 matrix and $\mathbf{W}^3$ is a 10×4 matrix. Thus, $\mathbf{W}^3\mathbf{W}^2$ is a 10×784 matrix of rank 4.

   On the other hand, if we use multiclass logistic regression, we'd have $\mathbf{z}^3 = \mathbf{W}\mathbf{x}$, where $\mathbf{W}$ is a 10×784 matrix, without any restriction on the rank. Thus, the multiclass logistic regression model is in fact more powerful, in terms of representation capabilities.

   When it comes to choosing a model, representation power is only one aspect to consider. The model with less representation power may be less prone to overfitting. In fact we can view the 784-4-10 structure of the network as imposing an information bottleneck in the middle, while still expressing a linear relationship between the inputs and outputs. Thus, which model one prefers depends on many factors, such as amount of data, the expected relationship between input and output, *etc.*

3. For the network in Part 2, if you use the cross entropy loss function on the output layer, do you get a convex optimisation problem?

   Solution: The answer is no. There are probably many ways to see this, but the simplest one is that the model is invariant to permuting the weights (both incoming and outgoing) on the four hidden units. Thus, whatever the global minimum, there are at least 4! of them and in different parts of the optimisation landscape. So the optimisation problem is not convex.

More formally, if $\mathbf{W}^2$ and $\mathbf{W}^3$ are the weights of the neural network at a non-trivial optima α, then $-\mathbf{W}^2$ and $-\mathbf{W}^3$ gives the same optima α. For the objective function to be convex the objective must be equal to α on the line joining these two solutions. Using $[\mathbf{W}^2; \mathbf{W}^3]$ to denote the concatenation of all of the weights into one long vector of dimension $784 \times 4 + 4 \times 10$:

$$\alpha = \mathcal{L}\left(\lambda\left[\mathbf{W}^2; \mathbf{W}^3\right] + (1-\lambda)\left[-\mathbf{W}^2; -\mathbf{W}^3\right], \mathbf{X}, \mathbf{y}\right), \quad \forall \lambda \text{ s.t. } 0 \leq \lambda \leq 1. \tag{3.4}$$

The midpoint of this line would set all the weights to zero, which cannot be the non-trivial optima α. Thus the objective is not convex.

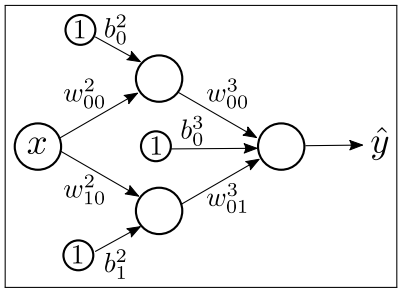
4 Neural Networks as Universal Function Approximators

As remarked in the lectures, neural networks are universal function approximators. In this question, we will discuss some steps for showing this result and what it means to be a universal function approximator a bit further.

Subsequently, in all questions we will exclusively consider neural networks with

- a single input,
- a single hidden layer with an arbitrary number of neurons with ReLU activation functions, and
- an output layer consisting of a single neuron with the identity activation function.

Consequently, the network implements a function $h\colon \mathbb{R} \to \mathbb{R}$. Whenever you are asked below to provide a neural network, your network is required to have those properties. The following picture shows an example of such a network with two neurons in the hidden layer:



1. Write down the model output $\widehat{y}$ as a function of x by application of the forward equations.

   Solution: We have $\mathbf{z}^2 = [z_0^2, z_1^2]^\intercal = [b_0^2 + w_{00}^2 \cdot x, b_1^2 + w_{10}^2 \cdot x]^\intercal$, $\mathbf{a}^2 = [a_0^2, a_1^2]^\intercal = [\max\{0, z_0^2\}, \max\{0, z_1^2\}]^\intercal$, $\widehat{y} = b_0^3 + w_{00}^3 \cdot a_0^2 + w_{01}^3 \cdot a_1^2$, and hence

$$\widehat{y} = b_0^3 + w_{00}^3 \cdot \max\left\{0, b_0^2 + w_{00}^2 \cdot x\right\} + w_{01}^3 \cdot \max\left\{0, b_1^2 + w_{10}^2 \cdot x\right\}.$$

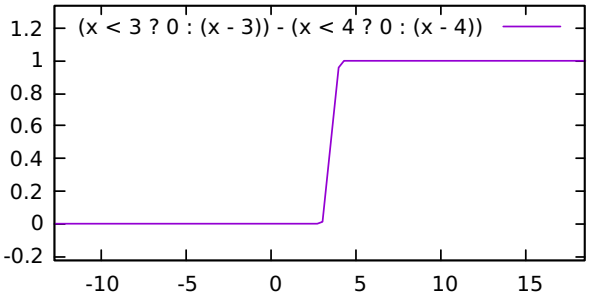


Figure 2: Function $h(x)$.

2. Set $w^2_{00} = w^2_{10} = w^3_{00} = 1$, $w^3_{01} = -1$, $b^2_0 = -3$, $b^2_1 = -4$, and $b^3_0 = 0$. Compute $\widehat{y}$ for the inputs $x_1 = 3$ and $x_2 = 4$. State and sketch the function $h\colon \mathbb{R} \to \mathbb{R}$ implemented by the network.

   Solution: The function is $h(x) = \max\{0, x-3\} - \max\{0, x-4\}$ which equals

$$h(x) = \begin{cases} 0 & \text{if } x < 3 \\ x - 3 & \text{if } 3 \le x \le 4 \\ 1 & \text{if } x > 4. \end{cases}$$

   The function is plotted in Figure 2. Hence $h(x_1) = 0$ and $h(x_2) = 1$.

3. Given constants $c < d$, provide a neural network which implements a function $h_{c,d}\colon \mathbb{R} \to \mathbb{R}$ such that for all $x \in \mathbb{R}$

$$h_{c,d}(x) = \begin{cases} 0 & \text{if } x < c \\ \frac{1}{d-c} \cdot x - \frac{c}{d-c} & \text{if } c \le x \le d \\ 1 & \text{if } x > d. \end{cases}$$

   Solution: We observe that

$$h_{c,d}(x) = \frac{1}{d-c} \cdot \max\{0, x-c\} - \frac{1}{d-c} \cdot \max\{0, x-d\}.$$

   Hence, the function can be implemented using the neural network with two hidden layers given in the question by setting

$$w^2_{00} = w^2_{10} = 1 \qquad w^3_{00} = 1/(d-c) \qquad w^3_{01} = -1/(d-c)$$
$$b^2_0 = -c \qquad b^2_1 = -d \qquad b^3_0 = 0.$$

4. Let $f\colon \mathbb{R} \to \mathbb{R}$ be any function such that on the interval $[0, 1]$, f is continuous and bounded, i.e., there is some constant $c \ge 0$ such that $|f(x)| \le c$ for all $x \in [0, 1]$. Give a high-level

yet mathematically well-founded argument why the class of neural networks considered in this question can approximate f arbitrarily close. More formally, given f and $\varepsilon > 0$, we can provide a neural network implementing some $h\colon \mathbb{R} \to \mathbb{R}$ such that $|f(x) - h(x)| < \varepsilon$ for all $x \in [0, 1]$.

Solution: We first observe that, analogously to the previous question, for any $c < d$ we can derive a neural network $\overline{h}_{c,d}$ such that

$$-h_{c,d}(x) = \overline{h}_{c,d}(x) = \begin{cases} 0 & \text{if } x < c \\ \frac{1}{c-d} \cdot x - \frac{c}{c-d} & \text{if } c \leq x \leq d \\ -1 & \text{if } x > d. \end{cases}$$

For $i > 0$, let $g_{c,d,i} := h_{c,d} + \overline{h}_{c+i,d+i}$. Observe that $g_{c,d,i}$ can be implemented as a neural network, and that g_i creates a spike of height one and length i between c and $d + i$ when $d - c \to 0$. In order to approximate f, we split $[0, 1]$ into a finite number of intervals such that for any interval $[k, \ell]$ and $x \in [k, \ell]$, we have $|f(k + 1/2(\ell - k)) - f(x)| < \varepsilon$ (this is possible since f is continuous and bounded). Denote by $[k_1, \ell_1], \ldots, [k_m, \ell_m]$ the thus obtained intervals, and define the h as

$$h = \sum_{i=1}^{m} f(k_i + 1/2(\ell_i - k_i)) \cdot g_{k_i, k_i + \mu, (\ell_i - k_i)}$$

where μ is infinitesimally close to zero. Again, h can be implemented by a neural network. This function almost achieves our goal, however it will not approximate f well-enough between the consecutive intervals. With a slightly more sophisticated construction, this problem can be circumvented, but this would go beyond the scope of this problem sheet. Moreover, the approach generalises to arbitrary functions $f\colon \mathbb{R}^m \to \mathbb{R}^n$.

5. Argue that for any choice of weights and biases of the network in the figure above, there exists some x_0 after which h behaves like a linear function, i.e., there is some $f(x) = m \cdot x + n$ such that for all $x \geq x_0$, $h(x) = f(x)$. Is the same true if there are arbitrarily many neurons in the hidden layer?

   Solution: For a sufficiently large x_0, $g(x) = \max\{0, b + w \cdot x\}$ will either be 0 or $b + w \cdot x$ for all $x \geq x_0$, which in both cases is a linear function. Since linear functions are closed under addition and multiplication of constants, the statement follows. Obviously, the same argument works in the presence of arbitrarily many neurons in the hidden layer.

6. You have become head of a venture capital firm investing in promising future technologies. A new start-up called BrainDrain is applying for funding at your firm. They claim that using a blockchain-based in-the-cloud approach, they have trained a neural network that, on an input x, outputs a value greater than 0 if x is a prime number and 0 otherwise. Will you invest in this start-up?

   Solution: In the previous question, we have seen that any neural network (as considered in this question) will eventually implement a linear function for sufficiently large inputs. Thus it will either classify all numbers above the threshold as prime or not prime, respectively. The claim made by the start-up is bogus and it would not be wise to invest in it.

5 Recurrent Neural Networks

1. The Truncated Backpropagation Through Time (TBPTT) algorithm for training Recurrent Neural Networks (RRNs) truncates the gradient calculation to a maximum of k steps. Compare a feed forward neural n-gram model trained with SGD and an RNN language model trained with TBPTT truncated to n steps. Does the truncation make the RNN equivalent to the n-gram model or can the RNN learn dependencies longer than n, either in theory or in practice? Discuss.

   Solution: The RNN language model can in theory learn dependencies longer than n. TBPTT only truncates the backward pass, the forward pass is still performed exactly as this can be done efficiently by initialising the first step in each mini-batch with the final hidden state from the previous batch. However, as we saw in the lectures, the vanishing gradient problem can severely limit the ability of an RNN to learn long range dependencies in practice.

2. The BPTT algorithm computes the derivates of the unrolled sequence in reverse order, iteratively computing the following recurrence,

$$\nabla_\theta \mathcal{F} = \sum_{n=1}^{N} \frac{\partial \mathcal{F}}{\partial \mathbf{h}_n} \frac{\partial \mathbf{h}_n}{\partial \theta_n} = \sum_{n=1}^{N} \left[\frac{\partial \mathcal{F}}{\partial \mathbf{h}_{n+1}} \frac{\partial \mathbf{h}_{n+1}}{\partial \mathbf{h}_n} + \frac{\partial \mathcal{F}_n}{\partial \mathbf{h}_n} \right] \frac{\partial \mathbf{h}_n}{\partial \theta_n}.$$

   Where $\mathcal{F} = \sum_n \mathcal{F}_n$ and the θ_n notation refers to the copy of the RNN parameters at timestep n in the unrolled computation graph. The complete gradient for the parameters θ is the sum of gradients for each copy.

   An alternative to backpropagation is to compute the partial derivates using an in order recurrence,

$$\nabla_\theta \mathcal{F} = \sum_{n=1}^{N} \frac{\partial \mathcal{F}_n}{\partial \mathbf{h}_n} \frac{\partial \mathbf{h}_n}{\partial \theta} = \sum_{n=1}^{N} \frac{\partial \mathcal{F}_n}{\partial \mathbf{h}_n} [\dots].$$

   Complete this equation by filling in the missing partial derivatives in the brackets $[\dots]$. Using the completed equation, propose an alternative SGD training algorithm to BPTT. What is the computational and memory complexity of this algorithm, and what are its advantages and disadvantages versus BPTT?

   Solution: The missing term is $\frac{\partial \mathbf{h}_n}{\partial \theta}$ and the completed equation is:

$$\nabla_\theta \mathcal{F} = \sum_{n=1}^{N} \frac{\partial \mathcal{F}_n}{\partial \mathbf{h}_n} \frac{\partial \mathbf{h}_n}{\partial \theta} = \sum_{n=1}^{N} \frac{\partial \mathcal{F}_n}{\partial \mathbf{h}_n} \left[\frac{\partial \mathbf{h}_n}{\partial \theta_n} + \frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \frac{\partial \mathbf{h}_{n-1}}{\partial \theta} \right]. \tag{5.1}$$

   BPTT is an example of reverse model automatic differentiation. The alternative using Equation 5.1 is an example of forward mode automatic differentiation and we can use this to calculate the gradient needed for SGD by recursively computing $\frac{\partial \mathbf{h}_n}{\partial \theta}$ from $\frac{\partial \mathbf{h}_{n-1}}{\partial \theta}$, summing $\frac{\partial \mathcal{F}_n}{\partial \mathbf{h}_n} \frac{\partial \mathbf{h}_n}{\partial \theta}$ as we go. At the end of the sequence we will have accumulated the complete gradient update. The application of this algorithm to RNNs is known as Real

Time Recurrent Learning (RTRL) and there has been a significant amount of research attempting to use it in practice. The key advantage of RTRL is to realise that we do not need to wait until the end of the sequence to make a weight update, as we have a complete gradient at each timestep, we can do an SGD update after every input (this is where the real time in the name comes from!). However, it is also importent to understand that every time we do an SGD update our estimate of $\frac{\partial \mathbf{h}_{n-1}}{\partial \theta}$ becomes 'stale', i.e. it will not be exactly correct with respect to the new weights. If our SGD steps are small enough this is not normally a big problem.

The reason that RTRL is not the standard algorithm for training RNNs is the high computational and space complexity. The term $\frac{\partial \mathbf{h}_n}{\partial \theta}$, which must be stored and passed between timesteps, is a matrix of $|\mathbf{h}_n| \times |\theta|$, where $|\mathbf{h}_n|$ is normally in the 100s or 1000s, while $|\theta|$ could be millions or billions. Compare this to the term $\frac{\partial \mathbf{h}_{n+1}}{\partial \mathbf{h}_n}$ in BPTT which is normally orders of magnitude smaller. Computationally, calculating $\frac{\partial \mathbf{h}_n}{\partial \mathbf{h}_{n-1}} \frac{\partial \mathbf{h}_{n-1}}{\partial \theta}$ requires multiplying a matrix of size $|\mathbf{h}_n| \times |\mathbf{h}_n|$ by one of $|\mathbf{h}_n| \times |\theta|$, which has much higher complexity than the calculations for BPTT.

References

Michael Nielsen. *Neural Networks and Deep Learning*. 2015. Online Book available at: `http://neuralnetworksanddeeplearning.com/`.