Deep Learning in Target Space

Michael Fairbank M.FAIRBANK@ESSEX.AC.UK
Spyridon Samothrakis SSAMOT@ESSEX.AC.UK
Luca Citi LCITI@ESSEX.AC.UK
Department of Computer Science and Electronic Engineering
University of Essex
Colchester, CO4 3SQ, UK

Editor:

arXiv:2006.01578v2 [cs.NE] 7 Nov 2020

Abstract

Deep learning uses neural networks which are parameterised by their weights. The neural networks are usually trained by tuning the weights to directly minimise a given loss function. In this paper we propose to reparameterise the weights into targets for the firing strengths of the individual nodes in the network. Given a set of targets, it is possible to calculate the weights which make the firing strengths best meet those targets. It is argued that using targets for training addresses the problem of exploding gradients, by a process which we call cascade untangling, and makes the loss-function surface smoother to traverse, and so leads to easier, faster training, and also potentially better generalisation, of the neural network. It also allows for easier learning of deeper and recurrent network structures. The necessary conversion of targets to weights comes at an extra computational expense, which is in many cases manageable. Learning in target space can be combined with existing neural-network optimisers, for extra gain. Experimental results show the speed of using target space, and examples of improved generalisation, for fully-connected networks and convolutional networks, and the ability to recall and process long time sequences and perform natural-language processing with recurrent networks.

Keywords: Deep Learning, Neural Networks, Targets, Exploding Gradients, Cascade Untangling

1. Introduction

A feed-forward artificial neural network (NN) is a function $f(\vec{x}, \vec{w})$, parameterised by a weights vector $\vec{w}$, that maps an input vector $\vec{x}$ to an output vector $\vec{y} = f(\vec{x}, \vec{w})$. This paper initially considers feed-forward fully-connected layered NNs with n_{L} layers, as illustrated in Figure 1.

NNs can be used in many problem domains, including pattern recognition, classification and function approximation (Bishop, 1995; Goodfellow et al., 2016). There are also numerous industrial and scientific applications for NNs, including vision, neurocontrol, language translation, image captioning, reinforcement learning and game playing (Silver et al., 2017; Mnih et al., 2015; Karpathy and Fei-Fei, 2015; Fairbank et al., 2014a,b; Sutskever et al., 2014; Samothrakis et al., 2016).

Training a NN means deciding upon an appropriate value for the weights vector $\vec{w}$ so that the NN performs the desired task successfully. This training process is usually an

©2020 Michael Fairbank, Spyros Samothrakis, Luca Citi.

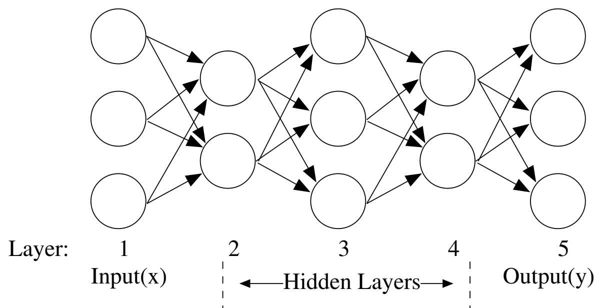

Figure 1: Diagram showing an example feed-forward NN with structure "3-2-3-2-3". Here there are five layers (i.e. $n_\mathrm{L} = 5$). An input vector $\vec{x} \in \mathbb{R}^3$ (in this example) is fed in as input to the left-most layer of nodes. The data propagates along the forward arrows (the *weights*) causing nodes in the next layer to fire. Each layer fires one-by-one towards the right, and finally produces an output vector, $\vec{y} \in \mathbb{R}^3$. The precise equations governing a NN are given in Section 2.1. Bias weights are not shown here, and this NN does not include shortcut connections.

iterative numerical method that works by trying continually to adjust $\vec{w}$ so as to minimise some real-valued loss function $L(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_{n_\mathrm{p}}, \vec{w})$ for a given set of n_p example input vectors $(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_{n_\mathrm{p}})$. In a supervised-learning task, the loss function is designed so that when minimised, each output vector $\vec{y}_i = f(\vec{x}_i, \vec{w})$ matches as possible closely some given data label or desired value $\vec{y}_i^*$, for each input vector $\vec{x}_i$ for $i \in \{1, \ldots, n_\mathrm{p}\}$. In unsupervised tasks, the loss function would represent some other objective, for example a penalty in a reinforcement-leaning problem, or an ability to reconstruct or group the input data.

The loss function L, measures how well the NN is achieving its desired task and its value at each point in weight space creates a surface, which the training process attempts to traverse to find a suitably low point. Most training algorithms use the gradient of the error function with respect to the weights, $\frac{\partial L}{\partial \vec{w}}$, which is calculated by the celebrated backpropagation algorithm (Werbos, 1974; Rumelhart et al., 1986). Two major difficulties for training are that the loss surface can be very crinkly in places, making the algorithms very slow, and also that the surface may be riddled with sub-optimal local minima and saddle points. It is these problems that the various training algorithms in existence, including novel activation functions and weight-initialisation schemes, are designed to overcome, to varying extents.

When a NN processes an input vector $\vec{x}$, as illustrated in Figure 1, the internal (hidden) neurons and output neurons in it will fire at different strengths, or *activations*. Hence there is a real number, the activation strength, associated with each node. These activation values can be gathered together for all hidden layers and the output layer to form a single vector, $\vec{a}$.

Hence for each input vector $\vec{x}_i$, and given set of weights $\vec{w}$, there will be an associated activation vector $\vec{a}_i$. Given the NN weights $\vec{w}$ and several input vectors $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_{n_\mathrm{p}}\}$, the set of vectors $\{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_{n_\mathrm{p}}\}$ is uniquely determined by the equations that govern the NN's

operation. Conversely, given an arbitrary set of *target* activation vectors, $\{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_{n_p}\}$, and corresponding input vectors, $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_{n_p}\}$, a relatively cheap calculation using linear algebra could take place to uniquely determine the weight vector $\vec{w}$ that most closely achieves the set of target-activation vectors. Therefore the training process could work by iteratively improving the targets, instead of the weights. That is the central idea of this paper: to do NN training in target space (the space of all possible sets $\{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_{n_p}\}$) instead of the usual weight space (the space of all possible $\vec{w}$).

The motivation for switching from weight-space learning to target space is now discussed. With weight-space learning, any small adjustment to a weight in an early layer shown in Fig. 1 will make the activations coming out of that layer change by a correspondingly small amount. However these changed activations will have a knock-on effect in changing the activations in the next layer, and so on with each subsequent layer, often forming a *cascade* of changes which reverberate through the later layers.

If the subsequent layers' neurons are all close to their firing thresholds, or are on a particularly steep part of the activation function, then the small change in the early layer could have a *catastrophic scrambling* effect on the NN output. This is why the error surface in weight space is so crinkly, or even chaotic (Skorokhodov and Burtsev, 2019; Phan and Hagan, 2013). This is not a desirable property for any learning strategy to have to cope with. Another way of stating that a small change to a weight causes a catastrophic scrambling of behaviour, is to say that the sensitivity of the loss function with respect to that weight is very large. This is referred to as the exploding-gradients problem (Hochreiter and Schmidhuber, 1997a), and we hypothesise that this is the main reason why NNs with many layers are difficult to train using standard backpropagation.

With target space, any small change to the targets for one layer will still cause a correspondingly small change to the activations of that layer. But then the algorithm that tries to match the node activations to their targets in the subsequent layers will try to choose the weights intelligently so the disturbance to later layers is minimised, an effect which we call *cascade untangling*. If successful, this should minimise the disturbance caused by the initial small change, and hence make the error surface in target space much smoother than that of weight space, directly addressing the exploding-gradients problem. Increased smoothness of the surface will also reduce the number of local minima in it, and make the crevices in it wider and easier to follow by gradient descent.

This should be increasingly beneficial for NNs with many layers, and even more so for recurrent neural networks (RNNs) where the output of a neural network is looped back to be combined with subsequent inputs, causing data to cycle around the network many times. Training RNNs has been notoriously difficult. We discuss target-space techniques for RNNs in Section 4.1, but initally focus on feed-forward networks.

If the cascade untangling of target-space learning works as intended, then the resulting loss-function surface should be smoother in general, and in particular it should be flatter at the final resting place of the optimisation process. This should encourage better generalisation power of the neural network, since flat minima are hypothesised by Hochreiter and Schmidhuber (1997b) to produce better generalisation than a sharper minimum.

The experimental results given in this paper show that using target space does indeed allow for gaining better performance in the training of deeper networks than occurs with weight space, and includes examples of improved generalisation and improved number of

training iterations required for feed-forward networks, recurrent networks and convolutional layered networks; but with a higher computational cost per training iteration (due to the linear algebra process which converts from target space to weight space). We argue that this extra cost motivates choosing deeper but narrow network architectures, when training a network in target space.

The rest of the paper is structured as follows. In the rest of this section, we discuss related published work. In Section 2 we define the main target-space algorithm for feed-forward layered neural networks, and then discuss background technical information about the method in Section 3. In Section 4 we show how the method can be extended to convolutional and recurrent neural networks. In Section 5, we give experimental results for feed-forward, convolutional and recurrent neural networks. Finally, in Section 6, we give conclusions.

1.1 Related work

Target-space techniques were originally proposed by Rohwer (1990) under the name of "moving targets", and then re-proposed under different names by Atiya and Parlos (2000); Enrique Castillo and Alonso-Betanzos (2006). There are some technical difficulties with these early works, which were later identified and improved upon by Carreira-Perpinan and Wang (2014), and follow-up work. These prior target-space methods, and their modern variants, are described in more detail in Section 3.5.

Other modern deep-learning methods enable the training of deep networks in a different way from target space. Some of these are described here.

Exploding gradients in deep neural networks were first analysed by Hochreiter and Schmidhuber (1997a). They also identified and defined the opposite problem, vanishing gradients, which also occurs in deep and recurrent networks. The solution they proposed, Long Short-Term Memory (LSTM) networks, focuses on solving vanishing gradients in recurrent networks, and is very effective, especially at spotting and exploiting patterns in long time sequences. The target-space solution we propose focuses only on addressing exploding gradients, but when combined with a powerful optimiser like Adam, can also learn and exploit long time sequences (even compared to LSTM networks); as shown in Section 5.3.

Glorot et al. (2011) identified that vanishing and exploding gradients could largely be controlled by changing the non-linear functions used which affect the node's firing activation. They proposed to replace the conventional logistic-sigmoid and hyperbolic-tangent function by a rectified linear function, $\mathrm{ReLU}(x)$. Since their proposed activation function has a maximum gradient of 1, it limits the scale of a cascade of changes arising from any perturbed weight, and hence eases training of deep networks. It does not entirely prevent the gradients from decaying/exploding though, since the magnitude of the gradients are also amplified proportional to the magnitude of the weights in each layer (Hochreiter and Schmidhuber, 1997a). Furthermore, the rectified linear function produces some problems of its own, with its unbound magnitude of its output; which can lead to infinities appearing, particularly in recurrent networks. These infinities make the proposed ReLU activation function inappropriate for recurrent networks. We compare and include our method with a variant of this activation function in Section 5.

Another significant recent breakthrough in training deep networks has been through the careful choice of the magnitude by which weights are randomised before training commences. The magnitudes derived by Glorot and Bengio (2010) and He et al. (2015) are carefully chosen so that the mean and variance in activations of each node remain 0 and 1 respectively, regardless of the depth of the network. This prevents the activations at each layer growing without bound, or saturating on the flat parts of the tanh activation function, and thus prevent gradients from decaying or exploding.

Batch Normalisation (BN) (Ioffe and Szegedy, 2015) is a powerful method for helping with the training of deep networks. This method can be viewed as a simplification and close relative of target space, and also similar in aim as the above weight-initilisation methods, in that BN prevents the activations of nodes at subsequent layers from growing or saturating without bound. BN works by setting an individual "target" for the mean μ and standard-deviation σ for every node in a layer. These are applied to normalise the entire training batch passing through the given node. This normalisation can help by performing some limited form of cascade untangling, but to a lesser extent than target space does, since with BN the targets are just summary statistics for a whole node. BN is proven to work well in practice, and there has been some discussion on how it works so well (Santurkar et al., 2018). BN also has a relatively low computational cost compared to target space. However target space can do a better job of cascade untangling and training deep networks. We describe empirical comparisons of BN to target space in Section 5.

2. Target-Space Algorithm for Layered Feed-Forward Networks

In the first two subsections we describe the notation for ordinary weight-space learning for neural networks. The target-space algorithm is then defined in the subsequent subsections.

2.1 Terminology and feed-forward mechanism for a Neural Network

We extend the basic NN architecture described in Figure 1 to act on a batch of size n_b patterns simultaneously. Concatenate the batch of input column vectors $\{\vec{x}_{b_1}, \vec{x}_{b_2}, \ldots, \vec{x}_{b_{n_\mathrm{b}}}\}$ side by side into a single matrix X with n_b columns. Then we can define a feed-forward neural network (FFNN) as a function that maps this matrix, X, to an output matrix, Y. The network is split into n_L layers of *nodes* or *neurons*, each node having an activation function, $g : \mathbb{R} \to \mathbb{R}$, and there being a matrix of weights between each pair of layers.

The layers, respectively, consist of d_1, d_2, …, d_{n_L} nodes, as shown in Figure 1. Thus $X \in \mathbb{R}^{d_1 \times n_\mathrm{b}}$ and $Y \in \mathbb{R}^{d_{n_\mathrm{L}} \times n_\mathrm{b}}$.

In the most general case, each layer j is connected to each later layer $k > j$, via a matrix of weights $W_{j,k} \in \mathbb{R}^{d_k \times d_j}$. The network is then said to have "all shortcut connections". However in the more common case, shortcut connections are not included and the only non-zero weight matrices are between consecutive layers.

The activation function g is usually smooth and monotonic, and must be non-linear. It is often taken to be $g(x) = \tanh(x)$ or the ReLU function (Glorot et al., 2011). We allow the function g to be applied to a vector or matrix in an elementwise manner, i.e. $(g(A))^{ij} := g(A^{ij})$, for all i and j; and allow similar elementwise application for its first derivative, g'.

Each node has a "bias" which can be implemented by having an extra "layer 0" which contains just one node that always has activation of unity. Thus for each layer j, $W_{0,j} \in \mathbb{R}^{d_j \times 1}$ is a column vector of weights coming from layer 0, which represent bias values for layer j.

The activations are calculated layer-by-layer, according to Algorithm 1. In line 4 of the algorithm, $\mathbb{I}(j)$ denotes the set of integer layer-numbers of all layers that *feed forwards into* layer j. So for example, for a fully-connected layered network with all shortcut connections, $\mathbb{I}(3) = \{0, 1, 2\}$.

Algorithm 1 Feed-Forward Dynamics

1: $A_0 \leftarrow [1\ 1\ \dots\ 1]$ {Bias nodes $\in \mathbb{R}^{1 \times n_\mathrm{b}}$; a row vector of 1s}
2: $A_1 \leftarrow X$ {Input matrix. $X \in \mathbb{R}^{d_1 \times n_\mathrm{b}}$.}
3: **for** $j = 2$ to n_L **do**
4: $\quad S_j \leftarrow \sum_{k \in \mathbb{I}(j)} W_{k,j} A_k$ {Sums received by each node. $S_j \in \mathbb{R}^{d_j \times n_\mathrm{b}}$.}
5: $\quad A_j \leftarrow g(S_j)$ {Apply activation function. $A_j \in \mathbb{R}^{d_j \times n_\mathrm{b}}$.}
6: **end for**
7: $Y \leftarrow A_{n_\mathrm{L}}$ {Output Matrix. $Y \in \mathbb{R}^{d_{n_\mathrm{L}} \times n_\mathrm{b}}$.}

Running the feed-forward algorithm with an input matrix X generates a sequence of intermediate matrices, A_j and S_j for all layers j, whose elements hold the activations and sums, respectively, of each layer's nodes. These matrices and the output matrix Y are to be retained for later use. The pth column of each matrix X, A_j, S_j and Y all correspond to the same pattern p.

Even though the proposed architecture is layered, it is still a fully generic FFNN architecture. To act as if there were no layers and each neuron was acting individually, we could just set $d_j = 1$ for all j, and include all shortcut connections.

2.2 Gradient Descent in Weight Space

The objective of *training* a neural-network is to set the values of the weights so that they minimise a given *loss function*, or *error function*, $L : (X, \vec{w}) \to \mathbb{R}$, where $\vec{w}$ is a vector of all of the weights in the network. For supervised learning, the most common loss functions are the mean-squared error and cross-entropy loss.

The gradient-based learning algorithms that act in weight space require the gradients of the loss function with respect to the weight matrices, i.e. the set of quantities $\frac{\partial L}{\partial W_{k,j}}$, which are calculated by the standard backpropagation algorithm (Rumelhart et al., 1986), which follows.

Define δS_j and δA_j as working matrices with the same dimension in each case as S_j and A_j, respectively, for all layers j. These will be used to hold forms of $\frac{\partial L}{\partial S_j}$ and $\frac{\partial L}{\partial A_j}$, respectively, specifically the *ordered* partial derivatives as defined by Werbos (1974) and as used in automatic differentiation (Rall, 1981).

We use $\mathbb{I}^{-1}(k)$ to denote the set of integer layer-numbers of all layers that *feed forwards out of* layer k. $g'(x)$ means the first derivative of the activation function. $A \odot B$ means the Hadamard or elementwise product.

Using this notation Alg. 2 calculates the error gradients in weight space.

Algorithm 2 Error Backpropagation

Require: S_j and A_j matrices calculated according to Alg. 1

1: **for** $j = n_{\mathrm{L}}$ to 2 step -1 **do**

2: $\quad \delta A_j \leftarrow \begin{cases} \frac{\partial L}{\partial Y} & \text{if } j = n_{\mathrm{L}} \\ \sum_{k \in \mathbb{I}^{-1}(j)} (W_{j,k})^T (\delta S_k) & \text{otherwise} \end{cases}$

3: $\quad \delta S_j \leftarrow (\delta A_j) \odot g'(S_j)$

4: $\quad \frac{\partial L}{\partial W_{k,j}} \leftarrow (\delta S_j)(A_k)^T \quad \forall\, 0 \leq k < j$

5: **end for**

Let $\vec{w}$ vector be a shorthand for the flattened vector of all weights in the network; and similarly let $\frac{\partial L}{\partial \vec{w}}$ be a flattened vector of all of the $\frac{\partial L}{\partial W_{k,j}}$ matrices.

For weight-space learning, this gradient is used in a gradient-based optimisation algorithm. For example the gradient could be used for ordinary gradient descent:

$$\Delta \vec{w} = -\eta \frac{\partial L}{\partial \vec{w}}, \tag{1}$$

which is applied iteratively, for a small positive learning rate η. The learning rate η can be changed over training time, or a more advanced optimiser could be used to try to accelerate learning (e.g. RPROP (Riedmiller and Braun, 1993), conjugate gradients (Møller, 1993), Levenberg-Marquardt (Bishop, 1995), RMSProp (Tieleman and Hinton, 2012), or Adam (Kingma and Ba, 2014)).

2.3 Stacked Layer Input-Matrix and Weight-Matrix Notation

For layer j, define $A_{[0:j)}$ to be shorthand form for a vertically stacked block matrix of all the A_k matrices that provide an input to layer j, i.e. for all the $k \in \mathbb{I}(j)$. For example, for a simple layered feed-forward network we would have,

$$A_{[0:j)} := \begin{pmatrix} A_0 \\ A_{j-1} \end{pmatrix}, \tag{2a}$$

(where A_0 is the layer of bias nodes), and if all shortcut connections were present, then this would become,

$$A_{[0:j)} := \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{j-1} \end{pmatrix}. \tag{2b}$$

Also define $W_{[0:j]}$ as a side-by-side block concatenation of all the weight matrices that input to layer j. For example, with for a simple layered feed-forward network, we would get:

$$W_{[0:j]} := \begin{pmatrix} W_{0,j} & W_{(j-1),j} \end{pmatrix}, \tag{3a}$$

and, if all all shortcut connections were present, we would get,

$$W_{[0:j]} := \begin{pmatrix} W_{0,j} & W_{1,j} & \dots & W_{(j-1),j} \end{pmatrix}. \tag{3b}$$

This simplifies the formula for the NN feed-forward equations; line 4 of Algorithm 1 becomes,

$$S_j \leftarrow W_{[0:j]} A_{[0:j)}. \tag{4}$$

2.4 Using Targets to Parameterise a Neural Network Instead of Weights

So far the neural-network parameters have been the weights $\vec{w}$. We now describe how we can switch the representation to "targets".

Define the matrices T_2, T_3, $\dots$, T_{n_L}, to be the "target matrices" for each layer. These have the same dimensions as the corresponding S_j matrices. In the target-space approach, the set of T_j matrices will be the learnable parameters, replacing the role of the weight matrices. The weight matrices are relegated into calculated quantities that are dependent on the T_j matrices.

The target matrix for each layer T_j holds the "targets" for the S_j matrix at that layer; hence we want to choose the weights which make the S_j matrices get as close as possible to the T_j matrices, or to minimise $||S_j - T_j||$, where $||\cdot||$ denotes the Frobenius norm. To simplify computational complexity, we do this in a greedy layer-by-layer manner.

Substituting (4) shows that we therefore need to find

$$W_{[0:j]} = \arg\min_{W} \left[\left|\left| W A_{[0:j)} - T_j \right|\right|^2 + \lambda \left|\left| W \right|\right|^2 \right], \tag{5}$$

where the $\lambda\,||W||^2$ term is included to provide Tikhonov regularisation, which ensures that the solution in W is unique and kept reasonably small. The minimisation in (5) is a standard least-squares problem from linear algebra, with solution

$$W_{[0:j]} = T_j \left(A_{[0:j)}\right)^{\dagger}, \tag{6}$$

where the $\dagger$ indicates a regularised pseudoinverse matrix, defined by

$$A^{\dagger} := A^T (A A^T + \lambda I)^{-1}. \tag{7}$$

Here AA^T is referred to as the Gramian matrix, $\lambda \geq 0$ specifies the amount of Tikhonov regularisation, and I is the identity matrix. The presence of λI in (7) prevents the occurrence of non-invertible matrices.[1]

Hence the layer weights and activations can be calculated layer by layer. The full method by which the weights are calculated from the target matrices is given in Algorithm 3.

The main inputs to this algorithm are an input matrix $\overline{X}$ with batch size $\overline{n}_{\text{b}}$, and a list of target matrices T_j. The main outputs of this algorithm are the realised weight matrices, W_j.

1. An alternative to Tikhonov regularisation would be to use the Truncated Singlular Value Decomposition pseudoinverse, but this was avoided because the truncation means the derivatives are not as smooth. However the SVD (or similar decompositions) may be used to implement (7) in practice, to obtain improved numerical stability.

Algorithm 3 Converting Targets to Weights, in a FFNN, with Sequential Cascade untangling

1: $A_0 \leftarrow [1\ 1\ \ldots\ 1]$ {Bias nodes. $A_0 \in \mathbb{R}^{1\times\overline{n}_\text{b}}$.}
2: $A_1 \leftarrow \overline{X}$ {Input matrix. $\overline{X} \in \mathbb{R}^{d_1\times\overline{n}_\text{b}}$.}
3: **for** $j = 2$ to n_L **do**
4: $\quad W_{[0:j]} \leftarrow T_j\left(A_{[0:j)}\right)^\dagger$ {Calculates weights to layer j. $T_j \in \mathbb{R}^{d_j\times\overline{n}_\text{b}}$.}
5: $\quad S_j \leftarrow W_{[0:j]}A_{[0:j)}$ {$S_j \in \mathbb{R}^{d_j\times\overline{n}_\text{b}}$.}
6: $\quad A_j \leftarrow g(S_j)$ {$A_j \in \mathbb{R}^{d_j\times\overline{n}_\text{b}}$.}
7: **end for**

Because $A_{[0:j)}$ is a shorthand for a stack of activation matrices A_j, as defined in (2), it is intended that the changes to A_j in line 6 will immediately affect the $A_{[0:j)}$ matrices referenced in line 4 for higher values of j. This is what carries forwards the changes of an earlier layer, so that they can be corrected for by a later layer.

Once these weight matrices are obtained, they are then used in Alg. 1 to calculate the actual NN output. Note that Alg. 3 followed by Alg. 1 are run back-to-back, in that order, and can therefore be viewed as one continuous computational graph. (This is in contrast with some prior published work on target space, e.g. Carreira-Perpinan and Wang, 2014, where there are alternating phases of updating the W_j matrices followed by updating T_j matrices. In our method, the W_j matrices are defined as functions of the T_j matrices, and there are no alternating phases.)

Note that the aim of matching the S_j matrices to their targets will not be achieved exactly. In general where the number of patterns $\overline{n}_\text{b}$ is larger than the rank of the weight matrix, matching the targets exactly will be impossible. Hence we carry forward the disturbances actually achieved to the S_j matrices, as opposed to the disturbances intended by T_j matrices, in Line 6 of Alg. 3. Then the subsequent layers' targets will act to continue to try to dampen down this disturbance, taking into account the fact that the previous layer's targets will not have been met exactly, so that the subsequent cascade of changes is always minimised as much as possible. Hence we refer to the algorithm as having *Sequential cascade untangling* (SCU).

We found SCU to be much more effective when training neural networks than an alternative of assuming targets are met exactly, which would be implemented by replacing line 6 of Alg. 3 by

$$A_j \leftarrow g(T_j). \tag{8}$$

Since this approach does not carry forwards the actual cascade of changes beyond just one layer, we call this alternative approach "optimistic cascade untangling" (OCU), and this is what prior published research (for example, Rohwer, 1990; Atiya and Parlos, 2000; Enrique Castillo and Alonso-Betanzos, 2006) has always done. Experiments in Sec. 5.1 (Fig. 5) show a significant improvement in performance from using SCU over OCU on the Two-Spirals classification problem, and experiments in Sections 5.3 and 5.4 show the advantage it gives in recurrent neural networks.

2.5 Calculating the Learning Gradient in Target Space

The previous subsection described an algorithm which converts targets to weights. The next objective is to be able to do gradient descent in target space, i.e. with respect to the targets themselves.

Algorithm 3 can be viewed as a mapping function m from targets to weights, such that $\vec{w} = m(\vec{\tau})$, where $\vec{\tau}$ is a shorthand for the vector of all target matrices flattened and concatenated together. Given such a differentiable mapping function, m, we can convert gradient descent in weight space to gradient descent in target space by the chain rule:

$$\frac{\partial L}{\partial \vec{\tau}} = \left(\frac{\partial m}{\partial \vec{\tau}}\right)^T \frac{\partial L}{\partial \vec{w}}. \tag{9}$$

where $\frac{\partial m}{\partial \vec{\tau}}$ uses Jacobian matrix notation, and $\frac{\partial L}{\partial \vec{w}}$ and $\frac{\partial L}{\partial \vec{\tau}}$ are treated as column vectors.

This gradient $\frac{\partial L}{\partial \vec{\tau}}$ allows us to perform gradient descent in target space, directly on the main neural-network objective function, L.

Algorithm 4 applies (9) to calculate the $\frac{\partial L}{\partial T_j}$ matrices, for Algorithm 3's mapping method. The algorithm uses workspace matrices δA_j and δS_j which are identically dimensioned to their non-prefixed counterparts, for each layer j. The matrix $\delta A_{[0:j)}$ is built up of δA_k matrices, in the same way as Equation (2), and similarly $\frac{\partial L}{\partial W_{[0:j]}}$ is composed of $\frac{\partial L}{\partial W_{k,j}}$ matrices like (3). It is assumed that these matrices point to the same underlying data, so for example, changing $\delta A_{[0:3)}$ will immediately affect δA_2, and vice versa.

Algorithm 4 Calculation of Learning Gradient in Target Space

Require: S_j, A_j and W_j matrices calculated by Alg. 3, and $\frac{\partial L}{\partial W_{k,j}}$ matrices calculated by Alg. 2.

1: $\forall j, \delta A_j \leftarrow 0$
2: **for** $j = n_{\mathrm{L}}$ to 2 step -1 **do**
3: $\quad \delta S_j \leftarrow (\delta A_j) \odot g'(S_j)$
4: $\quad \frac{\partial L}{\partial T_j} \leftarrow \left(\frac{\partial L}{\partial W_{[0:j]}} + (\delta S_j) {A_{[0:j)}}^T\right) (A^{\dagger}_{[0:j)})^T$
5: $\quad \delta A_{[0:j)} \leftarrow \delta A_{[0:j)} + {W_{[0:j]}}^T (\delta S_j - \frac{\partial L}{\partial T_j}) + (A_{[0:j)} A^T_{[0:j)} + \lambda I)^{-1} \left(\left(\frac{\partial L}{\partial W_{[0:j]}}\right)^T + A_{[0:j)} (\delta S_j)^T\right) (T_j - S_j)$
6: **end for**

The useful outputs of this algorithm are the quantities $\frac{\partial L}{\partial T_j}$, for all layers j, which can be written collectively as $\frac{\partial L}{\partial \vec{\tau}}$. Hence the algorithm gives $\frac{\partial L}{\partial \vec{\tau}}$, which can be used to perform gradient descent:

$$\Delta \vec{\tau} = -\eta \frac{\partial L}{\partial \vec{\tau}} \tag{10}$$

As with weight-space gradient descent, a more advanced optimiser might be applied to achieve a speed up.

The target-gradient computation algorithm (Alg. 4) is derived in Appendix A. The most interesting part of the derivation is the differentiation under the matrix inverse operation. This was omitted by prior research (Rohwer, 1990; Enrique Castillo and Alonso-Betanzos, 2006), which indicates that their learning gradients were incorrect. Our informal experiments (not recorded here) showed that this severely reduced performance of those prior algorithms. Modern automatic-differentiation (Werbos, 2005; Rall, 1981; Abadi et al., 2016) libraries correctly handle differentiation under a matrix inverse, but as this step is non-obvious to derive manually, we have included the explicit algorithm here. Alternatively, if Alg. 3 followed by Alg. 1 followed by the calculation of L is passed through an automatic-differentiation library, then $\frac{\partial L}{\partial \vec{\tau}}$ should be calculated correctly, automatically.

We note that it is possible to combine Algs. 2 and 4 into one algorithm, which calculates $\frac{\partial L}{\partial T}$ matrices directly, without first calculating $\frac{\partial L}{\partial \vec{w}}$, which is a little more computationally efficient. However we have chosen to keep these two stages explicitly separate. This separation aids using randomly selected mini-batches to calculate $\frac{\partial L}{\partial \vec{w}}$ in Algs. 1-2, but to then retain a fixed $\overline{X}$ for the conversion of $\frac{\partial L}{\partial \vec{w}}$ to $\frac{\partial L}{\partial \vec{\tau}}$ by the target-space algorithms. In other words, we may wish to use a different input matrix X in Alg. 1 from $\overline{X}$ used in Alg. 3, and this choice might be useful if mini-batching (discussed in the next section) was required. Another advantage of keeping the two algorithms separated is that $\frac{\partial L}{\partial \vec{w}}$ in Alg. 2 can be calculated by standard software packages, for any arbitrary loss function. For example it would be easy to modify Alg. 2 to include some L2 regularisation, and then use that regularised gradient $\frac{\partial L}{\partial \vec{w}}$ as input to Alg. 4, without requiring any further modifications to Alg. 4.

3. Technical Aspects for Target-Space Implementations

The previous section has defined the main target-space method. We now consider some technical aspects, including how to use mini-batching, the effects of choice of λ, the computational complexity, how to initialise the target variables at the start of training, detail of differences between this method and previous published target-space work, and convergence properties of our method.

3.1 Mini-batching and the Choice of $\overline{X}$

For very large datasets, it becomes prohibitively expensive to compute $\frac{\partial L}{\partial \vec{w}}$ in Alg. 2 for the whole dataset. Hence with very large datasets, it is standard practice in deep-learning to use mini-batches; that is to operate on a smaller, randomly chosen, subset of the training data in any one training iteration, with $n_\mathrm{b} \ll n_\mathrm{p}$. The mini-batch chosen would be used to build the input matrix X inputted to Alg. 1. Using mini-batching also introduces a stochastic element to the optimisation process, which is also beneficial in finding flatter final minima in the loss-function surface, and thus improving generalisation (Bottou, 2010; Masters and Luschi, 2018).

As noted in Section 2.5, it is possible to use a different X for the computation of $\frac{\partial L}{\partial \vec{w}}$ in Algs. 1-2 from the $\overline{X}$ used in the target-space calculations of Alg. 3-4. But unlike the random mini-batches which may be used for calculating $\frac{\partial L}{\partial \vec{w}}$, the $\overline{X}$ used for target space

must be fixed; because every time we shuffle the mini-batches in $\overline{X}$, the corresponding learnable quantities T_j would have their meaning scrambled, which would disrupt learning.

For computational efficiency, it is possible for the patterns in $\overline{X}$ to be a mini-batch, i.e. a subset of the entire training set, or even a fixed random matrix[2]. But it must be a *fixed* matrix.

The larger $\overline{n}_\text{b}$ is (where $\overline{n}_\text{b}$ is the number of columns in $\overline{X}$), the more computationally expensive things will become. So how large should $\overline{n}_\text{b}$ be? Ideally, $\overline{n}_\text{b}$ should be sufficiently large so that the Gramian matrix in (7) would be as close to full-rank as possible. The more non-zero eigenvalues this product has, i.e. the more linearly independent columns in each A_j, the more useful the pseudoinverses calculated will be in performing cascade untangling (defined in Section 1).

Since the side-dimension of $A_j A_j^T$ is equal to the number of inputs to layer j, as a rule of thumb, we recommend to set $\overline{n}_\text{b}$ to be preferably as large as the widest layer in the network, and more so if the computational expense can be spared; as this will usually ensure the Gramian matrix is full rank.

Due to the regularisation term in (7), it is not disastrous if there are too few patterns in $\overline{X}$; however it will mean that target-space learning will not be able to generate full-rank weight matrices in any layer where the number of layer inputs exceeds $\overline{n}_\text{b}$. To avoid this problem, while maintaining computational efficiency, this motivates the use of network architectures which are deep and narrow, as opposed to architectures with a large number of nodes to each hidden layer.

3.2 Choice of λ

For choosing λ in equation (7): if it is too large then the effect of the pseudoinverses in (7) will be dulled in their ability to perform cascade untangling. Hence for large λ, the benefits of target-space learning start to disappear.

If λ is too small, then the inverse might become close-to-singular. This would mean small changes in A_j make large changes to the generated weight matrices, and hence the learning gradients in target space would become too steep.

If instability in learning is observed, then λ could be increased, to try to remove any particularly steep gradients in target space caused by the matrix inversion process. We used either $\lambda = 0.001$ or $\lambda = 0.1$ in all experiments in this paper.

Note that the λ in equation (5) is performing L2 regularisation only on the mapping between targets and weights. It does not limit the final magnitude of the weights in the neural network, since there is no restriction of the magnitude of T in equation (6), and there is no cost on the magnitude of T appearing in the main training objective function L. Hence, this L2 regularisation should not be confused with a desire to apply L2 regularisation on the weights of the neural network (weight decay), which would have the intention of regularising the neural network into having smaller magnitude weights. If that was required, then explicit weight decay terms (on the magnitudes of W) should be added into L.

2. See Sec. 5.4 for an example of this.

3.3 Computational Complexity

In this section we estimate the ratio of computational complexity between each iteration of learning in target space compared to weight space. In doing so, we ignore the computation of activation functions, and matrix additions, assuming these are dwarfed by matrix-multiplication operations.

For a given layer j, the input matrix to that layer is $A_{[0:j)}$, the weight matrix is $W_{[0:j]}$ and the target matrix is T_j. For brevity, we will denote these three matrices without subscripts, as A, W and T. Let n_{i} be an initialism for the number of inputs to the layer (i.e. the number of rows in A) and let n_{o} be the number of outputs from the layer (i.e. the number of rows in T).

Since $A \in \mathbb{R}^{n_{\mathrm{i}} \times \overline{n}_{\mathrm{b}}}$, and if $\overline{n}_{\mathrm{b}} > n_{\mathrm{i}}$, then direct multiplication to form the Gramian AA^T will take ${n_{\mathrm{i}}}^2 \overline{n}_{\mathrm{b}}$ floating-point operations (flops). Assuming matrix inversion takes roughly n^3 flops, and since the Gramian is of shape $n_{\mathrm{i}} \times n_{\mathrm{i}}$, the formation of $(AA^T + \lambda I)^{-1}$ will take a further $(n_{\mathrm{i}})^3$ flops. The formation of the product with A^T in equation (7) will take a further $(n_{\mathrm{i}})^2 \overline{n}_{\mathrm{b}}$ flops. Since $T \in \mathbb{R}^{n_{\mathrm{o}} \times \overline{n}_{\mathrm{b}}}$, the multiplication by T in equation (6) will take a further $n_{\mathrm{i}} n_{\mathrm{o}} \overline{n}_{\mathrm{b}}$ flops. Hence summing these four terms gives the total time to form the pseudoinverse and calculate the weight matrix in (6), as $(n_{\mathrm{i}})^3 + 2(n_{\mathrm{i}})^2 \overline{n}_{\mathrm{b}} + n_{\mathrm{i}} n_{\mathrm{o}} \overline{n}_{\mathrm{b}}$.

If however $\overline{n}_{\mathrm{b}} < n_{\mathrm{i}}$, then the matrix A is taller than it is wide, and (7) can be rearranged using the Woodbury matrix identity into an equivalent but more efficient form:

$$A^{\dagger} := (A^T A + \lambda I)^{-1} A^T. \tag{11}$$

If this version is used, then the computational complexity is identically derived, resulting in the same flop-count expression but with all occurrences of n_{i} and $\overline{n}_{\mathrm{b}}$ swapped.

Hence the resulting overall flop count for calculating W by a pseudoinverse, assuming the faster of the two equations (7) and (11) is used, is

$$\text{Flop count for } W \text{ calculation} = \begin{cases} (n_{\mathrm{i}})^3 + 2(n_{\mathrm{i}})^2 \overline{n}_{\mathrm{b}} + n_{\mathrm{i}} n_{\mathrm{o}} \overline{n}_{\mathrm{b}} & \text{if } n_{\mathrm{i}} < \overline{n}_{\mathrm{b}} \\ (\overline{n}_{\mathrm{b}})^3 + 2(\overline{n}_{\mathrm{b}})^2 n_{\mathrm{i}} + n_{\mathrm{i}} n_{\mathrm{o}} \overline{n}_{\mathrm{b}} & \text{otherwise} \end{cases} \tag{12}$$

Once the W matrix for the layer is formed, the feed-forward calculation of the product $S_j = WA$ takes place, which is the same computational complexity as is required in ordinary weight space, i.e. requiring

$$\text{Flop count for } S_j \text{ calculation} = n_{\mathrm{i}} n_{\mathrm{o}} \overline{n}_{\mathrm{b}} \tag{13}$$

If it can be assumed that the number of nodes in each layer of the neural network is approximately the same, so that $d_j = \overline{d}$ for all j, and no shortcut connections are present, then we can assume that $n_{\mathrm{i}} \approx n_{\mathrm{o}} \approx \overline{d}$ (ignoring the single input from the bias node). If we further assume that the size of the batch $\overline{n}_{\mathrm{b}}$ is larger than $\overline{d}$ (so that also $\overline{n}_{\mathrm{b}} > n_{\mathrm{i}}$), then summing the expressions in (12) and (13) and simplifying shows that the flop count for each layer of the target space Alg. 3 is bounded above by $4\overline{d}^2 \overline{n}_{\mathrm{b}}$. In comparison, the weight-space forward-pass algorithm for a single layer is just given by (13), i.e. $\overline{d}^2 n_{\mathrm{b}}$ flops. Hence the ratio of computation between target space and weight space is approximately upper-bounded by $(4\overline{n}_{\mathrm{b}}/n_{\mathrm{b}})$. Since automatic differentiation produces backward computations of similar algorithmic complexity as to the forward pass, the overall computation ratios for

forward-and-backward passes between target space and weight space, when summed over all layers, is still approximately $(4\overline{n}_\mathrm{b}/n_\mathrm{b})$.

Note that in the above ratio, $\overline{n}_\mathrm{b}$ is the batch-size used for the target space matrix $\overline{X}$, and n_b is the batch size for the weight-space input matrix X. Hence if smaller mini-batches are used to acquire the weight-space gradient than are used in the target-space algorithms, then the time per iteration of the target-space algorithm (which cannot use tiny mini-batches) would become increasingly large in comparison to the weight-space calculations. Hence in the extreme case of pattern-by-pattern learning ($n_\mathrm{b}=1$), the target-space algorithm would be slower by a very significant factor of approximately $4\overline{n}_\mathrm{b}$.

In the experiments of this paper, we use $\overline{n}_\mathrm{b}=n_\mathrm{b}$, and the resulting theoretical ratio of 4 holds out in the experiments for fully connected neural networks. However further consideration is required for convolutional networks, as discussed in section 4.2.

3.4 Target initialisation

At the start of training, the layer target matrices T_j need to be randomised. We used a truncated normal distribution, with mean 0 and a fixed variance to randomise each element of each T_j matrix.

Since these initial layer targets have the same fixed variance at every layer, the variance of the magnitudes of the layer activations should be the same at every layer of the initially-randomised network. This is in contrast to weight-space initialisation, where unless the initial randomised weight magnitudes are chosen very carefully (such as by using the methods proposed by He et al. (2015); Glorot and Bengio (2010)), then the activations at subsequent layers can grow exponentially, eventually either saturating or becoming zero.

We have empirically found that it may be beneficial to run Alg. 3 once immediately after the initial targets are randomised, to compute the weight matrices and S_j, and then to apply

$$T_j \leftarrow S_j,\ \forall j, \tag{14}$$

exactly once before training commences. This simply projects the newly-randomised targets on to the hypersurface through target space which represents the subset of targets which are exactly achievable. This step is done in all of the target space experiments presented in this paper. It remains to be seen how much value this step adds, or whether it is equivalent to just choosing a smaller magnitude for the targets' initial randomisation.

3.5 Relationship to Prior Target-Space Research

The work by Rohwer (1990) is a stand-out early work on target space which we discuss here, along with more recent notable work, particularly those following on from Carreira-Perpiñán and Wang (2012). Some of the prior work is dedicated to recurrent networks (e.g. Atiya and Parlos, 2000), some is dedicated to feed-forward networks with one hidden layer (Enrique Castillo and Alonso-Betanzos, 2006), and some (especially more recent publications) is dedicated to general deep architectures (e.g. Rohwer, 1990; Carreira-Perpiñán and Wang, 2012; Lee et al., 2015a,b; Taylor et al., 2016; Zhang et al., 2016; Frerix et al., 2017).

In some of the prior works, the process which converts targets into weights seeks to minimise $||g(S_j)-T_j||$ or $\left|\left|S_j-g^{-1}(T_j)\right|\right|$ instead of $||S_j-T_j||$. Unfortunately there is no closed-form solution to minimise $||g(S_j)-T_j||$ with respect to the weights, and the second

option $||S_j - g^{-1}(T_j)||$ requires the function g to be invertible and the domain of T_j to be restricted to the range of g.

Early prior published work (Rohwer, 1990; Atiya and Parlos, 2000; Enrique Castillo and Alonso-Betanzos, 2006) is only applicable to the sum-of-squared loss function, and hence only to supervised regression problems. A significant defect of these early target-space methods, which probably held back their greater adoption, is that instead of optimising the main objective function L, they instead optimise an intermediate loss function, similar in concept (ignoring bias and shortcut connections) to

$$E(X, \vec{\tau}) = \sum_j ||W_j g(T_{j-1}) - T_j||^2, \tag{15}$$

instead of the true sum-of-squares cost function,

$$E(X, \vec{\tau}) = \sum_j ||W_j g(S_{j-1}) - T_j||^2. \tag{16}$$

They aim to minimise (15) with respect to the variables T_j, subject to each W_j satisfying (6), and subject to the final layer's targets satisfying $T_{n_\mathrm{L}} = Y^*$, where Y^* is the target data in the supervised regression problem. If (15) is successfully minimised down to zero then it will follow that $T_j = S_j$ for all j, and (15) will match (16), and so the supervised learning problem will be solved. However seeing as it is in general impossible to achieve a zero error in (15), it means that the first network layer will fail to achieve $S_1 = T_1$ exactly, and hence the "input" to the second layer in (15), namely $g(T_1)$, will be wrong. This misalignment between S_j and T_j will grow more and more as the layer number j increases. The end result is that local minima in (15) do not align with local minima in (16), and so gradient descent on (15) does not actually minimise the intended loss function. This was a crucial error limiting the applicability of the methods by Rohwer (1990) and Enrique Castillo and Alonso-Betanzos (2006). Additionally the work by Rohwer (1990) and Enrique Castillo and Alonso-Betanzos (2006) make an incorrect derivative calculation in computing the learning gradient, by omitting to differentiate through the matrix inverse operation of equation (7). A related error of following the wrong gradient descent direction appears in the work of Atiya and Parlos (2000). They approximate $\frac{\partial L}{\partial T_j} = 0$ for all $j < n_\mathrm{L}$, which is incorrect since cascade untangling can never occur perfectly.

Later work rectifies these problems. The work by Carreira-Perpiñán and Wang (2012) refers to the target variables as auxiliary coordinates. They solve the problems associated with (15) by instead using a bespoke objective function that is something like a weighted sum between (15) and (16), and where the weighting towards (16) is gradually increased during learning. This ensures that it is (16) that is finally optimised, while benefitting from the easier learning of (15) in earlier training. However their method requires alternating phases of minimisation with respect to W_j followed by minimising with respect to T_j; and then both of these phases need interlacing with increasing the weighting of (16) versus (15). Our method streamlines this process by having a single optimisation to do, which avoids zig-zagging through the search space, and allows for acceleration methods to be applied. But in comparison, their method increases the decoupling of the layers by successfully using an equation based on (15) for the majority of the learning process.

Frerix et al. (2017) extend upon the work of Carreira-Perpiñán and Wang (2012) but they modify the cost function so that the targets within it are anchored to the forward-propagated activations (by an equation similar to (14); so that the targets are no-longer free variables to be learned). This modification creates an implicit quadratic cost function attached to each layer (similar to (15)) which enables the use of a semi-implicit optimisation algorithm based on proximal updates. The proximal updates can converge under much higher learning rates than would be possible with ordinary gradient descent.

In "Difference Target Propagation", Lee et al. (2015b) define a method which uses learnable targets for each hidden layer. In this method, the target at one layer T_j is iteratively set to $L_P^{-1}(T_{j+1})$, where L_P^{-1} is an inverse function of the layer's forward-propagation function, and where this inverse (being generally an unknown function) is learned by an auto-associative network which learns to model L_P for each given network layer. This method potentially allows training of networks with discrete activation functions. In "Deeply-Supervised Nets", Lee et al. (2015a) add an extra support-vector machine classifier for the output of each layer. This provides extra training information; a kind of target for each hidden layer, which proves very effective in training deep classification networks.

Taylor et al. (2016) use learnable targets for both the A_j and S_j matrices, and update these learnable variables with iterative application of a closed-form Bregman method, which trains the network to solve the objective function, without needing to use any form of gradient descent. Zhang et al. (2016) use a similar iterative scheme to train neural networks to generate supervised hash codes.

In summary, much of the prior work shows the potential and power of target space, and the recent prior work addresses the problems appearing earlier in novel ways.

Our work provides several notable further enhancements and alternatives to the prior work, particularly regarding the introduction of the SCU method, which we show in our experiments is beneficial to performance. Furthermore, none of the prior work shows how to separate the input matrix $\overline{X}$ (which is used for calculating the weights from targets) from the input matrix X (which is used to run the neural network in Alg. 1). Our work also introduces the correction of gradient calculations through the pseudoinverse operation (which is necessary to apply (9) correctly); the separation of the main objective's loss function from the intermediate closed-form least-squares minimisation; and the introduction of mini-batches. The simplicity of the method, and the view of searching in "target space", gives a single, simple, gradient-descent objective, i.e. (10), which can easily be combined with existing acceleration schemes such as Adam.

3.6 Convergence Properties and Representation Capabilities of Target Space

The target-space gradient descent update (10) is derived to be true gradient descent on the loss function $L(\vec{x}, \vec{w})$, where the weights are functions of the targets as prescribed by Alg. 3. The loss function L is the main learning objective function, as chosen by the practitioner. For example, for a regression problem, L could be the mean-squared error, or for classification problems, it could be cross-entropy loss.

A potential source of confusion is that there is a second loss function appearing in the least-squares sub-problem given by (5), and also that the targets in each layer will not usually be matched exactly; but this least-squares sub-problem is completely separate from

the neural-network's main objective function, L. To see this more clearly, the mapping from targets to weights, $\vec{w} = m(\vec{\tau})$, given by Alg. 3, could be replaced by any other well-defined differentiable mapping function. Regardless of what the differentiable function m is, and regardless of how well any targets are matched or not matched, gradient descent is still performed on the main neural-network objective function L by (9).

Any sufficiently small step size in target space by (10) will yield a decrease in L, since, to first order:

$$
\begin{aligned}
\Delta L &\approx (\Delta\vec{\tau})^T \frac{\partial L}{\partial \vec{\tau}} && \text{(ignoring higher order terms)} \\
&= -\eta \left(\frac{\partial L}{\partial \vec{\tau}}\right)^T \frac{\partial L}{\partial \vec{\tau}} && \text{(by (10))} \\
&= -\eta \left(\frac{\partial L}{\partial \vec{w}}\right)^T \frac{\partial m}{\partial \vec{\tau}} \left(\frac{\partial m}{\partial \vec{\tau}}\right)^T \frac{\partial L}{\partial \vec{w}} && \text{(by (9))} \qquad (17) \\
&\leq 0
\end{aligned}
$$

Since the function L has a lower bound, L will decrease monotonically but not beyond that bound. Hence convergence of $L(\vec{x}, m(\vec{\tau}))$ to some limit is guaranteed. Similarly, the standard convergence proofs for gradient descent with appropriately chosen step sizes apply here (Bertsekas, 1999, Section 1.2.2).

Since the differentiable mapping function m is arbitrary, the convergence guarantees work just as well as for the OCU and SCU variants described in Section 2.4. The difference is that we hope that the target-space loss-surface is smoother in one variant than the other (and that both variants are smoother than in weight space), and therefore they will produce faster convergence and better generalisation (which can only be justified empirically; see discussion in Section 1 and empirical results in Section 5).

An important question is the relationships between "solutions" (i.e. stationary points) of the target-space problem, $L(\vec{x}, m(\vec{\tau}))$, and those of the original weight-space one, $L(\vec{x}, \vec{w})$. Equation (9) shows that whenever $\frac{\partial L}{\partial \vec{w}} = 0$, we must also have $\frac{\partial L}{\partial \vec{\tau}} = 0$. Furthermore, when Alg. 3 is used to define the mapping function m, Appendix B shows that whenever $\frac{\partial L}{\partial \vec{\tau}} = 0$, we must also have $\frac{\partial L}{\partial \vec{w}} = 0$. Hence any stationary point in target space is also a stationary point in weight space, and also the reverse is true.

Equation (17) shows that gradient descent in target space is equivalent to descent in weight space, but where each weight-space direction is multiplied by a positive semi-definite pre-conditioner matrix $\frac{\partial m}{\partial \vec{\tau}} \left(\frac{\partial m}{\partial \vec{\tau}}\right)^T$. However by explicitly working in target space, we get the benefit of being able to apply an acceleration procedure to the descent steps in target space, such as Adam, and still retain the convergence guarantees proven for that acceleration method. We would lose these guarantees if we applied the semi-definite preconditioner matrix in weight space, and then applied Adam afterwards.

For any given set of weights we can run the neural network forwards, and can capture the sums S_j at each layer j, and assign these to the targets at each layer, by (14). Ignoring the Tikhonov regularisation in (5), this will mean the targets will generate weights approximately equal to the given set of weights that we started with. This shows that any point in weight space has at least one equivalent representation in target space, and hence any

local minimum in weight space could be reached by gradient descent from an appropriate random start point in target space.

4. Specific Deep Architectures

The target-space method can be extended to different neural architectures and layer types. Here we show specifically how the method can be extended to convolutional neural networks and recurrent neural networks.

4.1 Application to RNNs

Recurrent neural networks (RNNs) are a powerful architecture of neural networks, which extend the feed-forward network by having one or more recurrent (backward pointing) weights. These feedback connections allow information from previous inputs be retained and to contribute extra information to subsequent inputs to the network. This creates short-term memory, which allows the network to remember and act on past inputs, enabling a RNN to potentially have much greater functionality than a FFNN, potentially allowing it to act like an agent interacting with an environment. Successful RNN applications are in areas such as neurocontrol, time-series analysis, image captioning, language translation, and question answering (Karpathy and Fei-Fei, 2015; Fairbank et al., 2014a,b; Sutskever et al., 2014; Samothrakis et al., 2016). However RNNs are generally more difficult to train than feed-forward networks, with major challenges being vanishing or exploding learning gradients, making it difficult for a RNN to remember information over long time sequences.

This section describes how a RNN can be trained in target-space. Target-space methods potentially allow RNNs to tackle more complex time sequences and data-processing tasks which previously have been very challenging for RNNs to solve.

A simplified recurrent architecture is shown in Fig. 2. This architecture consumes n_t input matrices $X^{(t)}$, one at each time step $t \in \{1, ..., n_\mathrm{t}\}$, and produces n_t output matrices $Y^{(t)}$. At each time step, data from an input matrix $X^{(t)}$ enters the RNN from the left and propagates forwards in the usual manner. When data reaches the "context layer", layer c_L, it loops back to the start of the RNN, and is combined with the next input matrix to go through the RNN again. Data loops around the recurrent layers many times, each time also passing through the exit layers which perform some final post-processing on the data to deliver the output matrices $Y^{(t)}$.

Pseudocode is given in Alg. 5. In this notation, layer 0 is reserved for the bias nodes; layer 1 is for the input matrices $X^{(t)}$, and layer 2 is for feedback received from the later context layer c_L. Superscript numbers in brackets indicate the time step, t.

Each input matrix $X^{(t)}$ may itself contain a batch of several patterns (one in each column). Hence the matrices $A_j^{(t)}$ and $S_j^{(t)}$ have dimension $d_j \times n_\mathrm{b}$.

An appropriate loss function L would be chosen that is a function of some or all of the $Y^{(t)}$ matrices, and then the gradient of this loss function with respect to the weights of the network, $\frac{\partial L}{\partial \vec{w}}$, can be found by automatic differentiation, using for example, backpropagation through time (Werbos, 1990), in execution time $O(n_\mathrm{b} n_\mathrm{t} n_\mathrm{w})$, where n_w is the number of weights in the network. Then, assuming weight-space is being used, an iterative optimizer would use this gradient information to tune $\vec{w}$, and train the network.

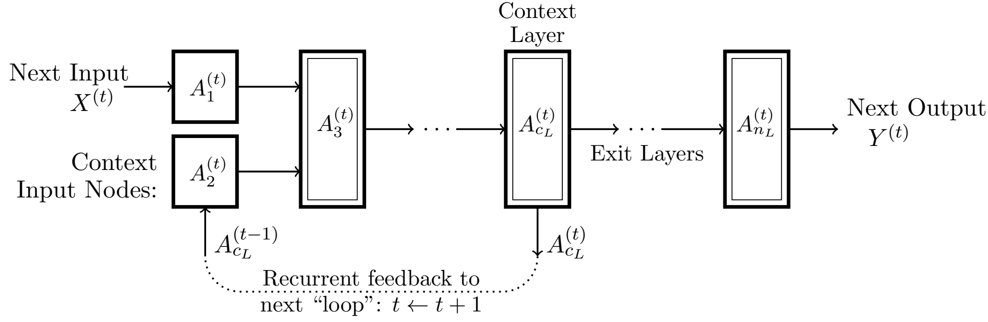

Figure 2: Diagram showing dataflow in a Recurrent Neural Network (RNN). Arrows show dataflow. Each rectangle shows a layer of nodes in the neural network; the layers with only a single rectangle are those that make no transformation to the incoming data. The data cycles around the network multiple times in "loops", each loop indexed by t. Algorithm 5 describes the process in greater detail. The "exit layers" do any necessary post-processing on the data. Extra shortcut connections, or repeated recurrent structure, may be present to obtain different RNN architectures.

Algorithm 5 Recurrent NN Dynamics

Require: On entry, require n_t input matrices $X^{(t)} \in \mathbb{R}^{d_1 \times n_\text{p}}$.
1: $A_{c_\text{L}}^{(0)} \leftarrow 0$ {Initial context units are zero}
2: **for** $t = 1$ to n_t **do**
3: $\quad A_0^{(t)} \leftarrow [1\ 1\ \dots\ 1]$ {Bias nodes}
4: $\quad A_1^{(t)} \leftarrow X^{(t)}$ {t^th input matrix.}
5: $\quad A_2^{(t)} \leftarrow A_{c_\text{L}}^{(t-1)}$ {Feedback from context layer}
6: $\quad$ **for** $j = 3$ to n_L **do**
7: $\quad\quad S_j^{(t)} \leftarrow W_{[0:j]} A_{[0:j)}^{(t)}$
8: $\quad\quad A_j^{(t)} \leftarrow g(S_j^{(t)})$
9: $\quad$ **end for**
10: $\quad Y^{(t)} \leftarrow A_{n_\text{L}}^{(t)}$ {t^th output matrix. $Y^{(t)} \in \mathbb{R}^{d_{n_\text{L}} \times n_\text{p}}$.}
11: **end for**

To incorporate target-space learning for a RNN, the intermediate objective is to make all the $S_j^{(t)}$ coming from Alg. 5 match as closely as possible some given target matrices $T_j^{(t)}$, for all time steps t. Hence, considering line 7 of Alg. 5, the objective is to choose a weight matrix $W_{[0:j]}$ so as to achieve,

$$W_{[0:j]} A_{[0:j)}^{(t)} \approx T_j^{(t)} \quad \text{for all } 1 \le t \le n_\text{t},$$

or equivalently to achieve, as closely as possible,

$$W_{[0:j]} A_{[0:j)}^{(:)} \approx T_j^{(:)},$$

where we have defined

$$A^{(:)}_{[0:j)} := \left(A^{(1)}_{[0:j)} \quad A^{(2)}_{[0:j)} \quad \dots \quad A^{(n_\mathrm{t})}_{[0:j)}\right),\ 3 \leq j \leq n_\mathrm{L} \tag{18a}$$

and

$$T^{(:)}_{j} := \left(T^{(1)}_{j} \quad T^{(2)}_{j} \quad \dots \quad T^{(n_\mathrm{t})}_{j}\right),\ 3 \leq j \leq n_\mathrm{L} \tag{18b}$$

The least squares solution to this is the same as in (6) and (7):

$$W_{[0:j]} = T^{(:)}_{j}\left(A^{(:)}_{[0:j)}\right)^{\dagger}, \tag{19}$$

however since this is a RNN, we now have the problem in that it is not possible to know the values of $A^{(:)}_{[0:j)}$ until the network can by run by Alg. 5; but that algorithm cannot be run until equation (19) is solved.

To break out of this cyclic dependency, we can approximate using the "optimistic" cascade untangling (OCU), given by (8), and therefore just set:

$$A^{(t)}_{j} \leftarrow g\left(T^{(t)}_{j}\right)\ \forall t. \tag{20}$$

This OCU step only needs doing on the context layer which feeds backward connections to the input layers. For the rest of the layers, it is preferable to use the SCU method. Alg. 6 shows how to do this in detail. This algorithm calculates the weights of a RNN from a given list of target matrices $T^{(t)}_{j}$, using the SCU method wherever possible, and the OCU method for the recurrent layer. The algorithm includes in line 10 an attempt to correct the error introduced by the OCU step once the exit layers (shown in Fig. 2) are reached.

To modify the algorithm to a fully OCU method, then we would replace line 8 by equation (20), and delete lines 7 and 10.

For the reasons discussed in Sec. 3.1, the content and length of the target-space input matrices, $\overline{X}^{(t)}$ for $t = 1, \dots, \overline{n}_\mathrm{t}$, may differ from the content and length of the weight-space input matrices ($X^{(t)}$ for $t = 1, \dots, n_\mathrm{t}$).

This algorithm merely outputs a set of weights of the RNN. The RNN would then have to be run separately, using Alg. 5, to obtain the set of output matrices $Y^{(t)}$.

Since Alg. 6 defines the mapping from targets to weights, it is possible to calculate the learning gradient with respect to the targets (first going via $\frac{\partial L}{\partial \vec{w}}$) using automatic differentiation, and hence train the RNN in target space. For example, if Alg. 6 followed by Alg. 5 is passed to an auto-differentiation toolbox, then the toolbox will be able to correctly calculate $\frac{\partial L}{\partial \vec{\tau}}$ by differentiation through both algorithms Sequentially. Section 5 shows experiments which do this, with successful results.

The bottleneck in algorithmic complexity for Alg. 6 is in forming the Gramian matrix AA^T, which will take ${n_\mathrm{i}}^2\overline{n}_\mathrm{b}\overline{n}_\mathrm{t}$ flops by direct multiplication. This is similar to a full forward-unroll of the RNN with the input matrix $\overline{X}$. Hence the relative complexity of running Alg. 6 using $\overline{X}$, compared to Alg. 5 using input matrix X, is approximately $\overline{n}_\mathrm{b}\overline{n}_\mathrm{t}/n_\mathrm{b}n_\mathrm{t}$. This motivates a choice of using a small value of $\overline{n}_\mathrm{t}$ where possible. See Section 5.4 for an example.

Algorithm 6 Conversion of Targets to Weights for a RNN (using SCU)

Require: On entry, require $\overline{n}_\text{t}$ input matrices $\overline{X}^{(t)} \in \mathbb{R}^{d_1 \times \overline{n}_\text{b}}$.

1: $A_0^{(t)} \leftarrow [1\ 1\ \dots\ 1]\ \forall t$ {Bias nodes}

2: $A_1^{(t)} \leftarrow \overline{X}^{(t)}\ \forall t$

3: $A_{c_\text{L}}^{(t)} \leftarrow g\left(T_{c_\text{L}}^{(t)}\right)\ \forall t$ {Estimates $A_{c_\text{L}}^{(t)}$ matrices by OCU method.}

4: $A_2^{(:)} \leftarrow \begin{pmatrix} 0 & A_{c_\text{L}}^{(1)} & A_{c_\text{L}}^{(2)} & \dots & A_{c_\text{L}}^{(n_t - 1)} \end{pmatrix}$ {Applies recurrent feedback from layer c_L to layer 2. Hence $A_2^{(:)}$ is a block shifted-right version of $A_{c_\text{L}}^{(:)}$.}

5: **for** $j = 3$ to n_L **do**

6: $\quad W_{[0:j]} \leftarrow T_j^{(:)} \left(A_{[0:j)}^{(:)}\right)^\dagger$ {Calculates weights to layer j}

7: $\quad S_j^{(:)} \leftarrow W_{[0:j]} A_{[0:j)}^{(:)}$.

8: $\quad A_j^{(:)} \leftarrow g\left(S_j^{(:)}\right)$ {SCU method}

9: $\quad$ **if** $j = c_\text{L}$ **then**

10: $\quad\quad$ Use the newly calculated $W_{[0:j]}$ matrices (for $3 \le j \le c_\text{L}$) to run Alg. 5 (using $\overline{X}^{(t)}$ as the input matrices), up to layer c_L, to obtain the true $A_{[0:c_\text{L}+1)}^{(:)}$ matrices. {This is an attempt to correct for the OCU estimation made in line 3.}

11: $\quad$ **end if**

12: **end for**

4.2 Application to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) represent one of the most powerful modern deep-learning architectures and are particularly applicable to vision problems. The key innovation of the convolutional neural network is the 2D-convolution operation: a smaller weight matrix is "convolved" (i.e. a sliding dot product is performed) with the source image to calculate the activations in the next layer. The convolutional operation means the weight matrix connecting one layer to the next can be much smaller than that of a fully connected network; and also that this smaller group of weights, the convolutional "kernel", will be applied to multiple patches of the image. This reuse helps in generalisation, and helps preserve spatial relationships in the image from one layer to the next.

A CNN network structure is usually comprised of a mixture of layer types - including one or more convolutional layers, one or more down-sampling (max-pooling) layers, flattening operations that reduce a tensor from rank 4 down to rank 2, and one or more regular fully-connected layers (as described in Section 2). Further details of how these layers all work and are arranged with each other are given by LeCun et al. (1998).

In generating a target-space method for training a CNN, it is only the convolutional layers and fully-connected layers that have any weights, and so only those two layer types that need modifying.

Each convolutional layer takes as input a 2D image, of size width×height, with a third depth dimension representing a number of input channels. Together with the batch size, n_b, this input image is a rank-4 tensor, of shape [n_b, input_height, input_width, input_channels]. The convolutional kernel that acts on it is a rank-4 tensor of shape [kernel_height, ker-

nel_width, input_channels, output_channels], and the layer's final output is a rank-4 tensor of shape [n_{b}, output_height, output_width, output_channels].

The entire convolutional layer's operation can be split into 6 steps:

1. Flatten the kernel to a 2-D matrix with shape [kernel_height×kernel_width×input_channels, output_channels]. Call this matrix W.
2. Extract image patches from the input tensor, and reshape them, to form a patches matrix A of shape [num_patches, kernel_height×kernel_width×input_channels], where num_patches= n_{b}×output_height×output_width.
3. Multiply the kernel matrix W by the patches matrix A, obtaining $S = WA$.
4. Add in the bias to S.
5. Reshape the result back into rank-4 tensor of shape [n_{b}, output_height, output_width, output_channels]
6. Apply the activation function g.

To optimise this process, so as to be able to easily modify it for target-space training, we first combine the bias addition of step 4 with the matrix multiplication of step 3. This can be achieved by adding an extra row of 1s into A, as was done in equation (2a), and an extra column of weights to W, as was done in equation (3a).

Then we need a target matrix T of the same dimension as S in line 3. Given this target matrix and the matrix A, we can derive the weights which best achieve the targets using the same least-squares process as with equation (6), i.e. $W = TA^{\dagger}$.

This derived weight matrix W is then used to calculate the actual product $S = WA$, and steps 5 and 6 (the reshape and activation function) are applied, completing the convolutional layer's behaviour.

The fully-connected layers are handled with their own target matrices and least-squares solution, as in Alg. 3. The rest of the layer-types in the CNN are unchanged - down-sampling does not use any targets (or weights), and nor does the reshape operation.

Automatic differentiation can be used to compute the necessary learning gradients.

This completes the description of how to use target space with a conventional CNN architecture.

4.2.1 Computational Complexity for a CNN layer in Target Space

We now consider the computational complexity of the CNN-target space forward pass. Notate the kernel width and heights by k_{w} and k_{h} respectively, and the number of input and output channels by n_{ic} and n_{oc} respectively. Let n_{patch} be the number of image patches to be taken from each image. Since the number of inputs operated on by the flattened W matrix is $n_{\mathrm{i}} = k_{\mathrm{h}} k_{\mathrm{w}} n_{\mathrm{ic}}$, and the number of outputs is $n_{\mathrm{o}} = n_{\mathrm{oc}}$, and the number of columns in the patches matrix A is $n_{\mathrm{b}'} = \overline{n}_{\mathrm{b}} n_{\mathrm{patch}}$, then substituting these factors into (12) gives a total flop count for the formation of W as:

$$\text{CNN Flop count for } W \text{ formation} = \begin{cases} (k_{\mathrm{h}}k_{\mathrm{w}}n_{\mathrm{ic}})^3 + 2(k_{\mathrm{h}}k_{\mathrm{w}}n_{\mathrm{ic}})^2 n_{\mathrm{b}'} + k_{\mathrm{h}}k_{\mathrm{w}}n_{\mathrm{ic}}n_{\mathrm{oc}}n_{\mathrm{b}'} & \text{if } k_{\mathrm{h}}k_{\mathrm{w}}n_{\mathrm{ic}} < n_{\mathrm{b}'} \\ (n_{\mathrm{b}'})^3 + 2(n_{\mathrm{b}'})^2 k_{\mathrm{h}}k_{\mathrm{w}}n_{\mathrm{ic}} + k_{\mathrm{h}}k_{\mathrm{w}}n_{\mathrm{ic}}n_{\mathrm{oc}}n_{\mathrm{b}'} & \text{otherwise} \end{cases} \tag{21}$$

In contrast, the weight-space CNN forward pass only requires the formation of S, where the flop count is given by (13), which equates to only $k_\mathrm{h} k_\mathrm{w} n_\mathrm{ic} n_\mathrm{oc} n_\mathrm{b} n_\mathrm{patch}$ flops.

If we argue like in Section 3.3 that $n_{\mathrm{b}'} > n_\mathrm{i}$ (which is quite probable with the large number of image patches being processed by a CNN), and $n_\mathrm{oc} \approx n_\mathrm{ic}$, then the flop count in target space is bounded above by

$$\begin{aligned}\text{CNN Flop count for } W \text{ formation} &\lessapprox 3(k_\mathrm{h} k_\mathrm{w} n_\mathrm{ic})^2 n_{\mathrm{b}'} + k_\mathrm{h} k_\mathrm{w} (n_\mathrm{ic})^2 n_{\mathrm{b}'} \\ &= (3(k_\mathrm{h} k_\mathrm{w}) + 1) k_\mathrm{h} k_\mathrm{w} (n_\mathrm{ic})^2 \overline{n}_\mathrm{b} n_\mathrm{patch} \end{aligned} \tag{22}$$

and hence the ratio of the flop count in target space to that in weight space is bounded above by approximately $(3(k_\mathrm{h} k_\mathrm{w}) + 1)\overline{n}_\mathrm{b}/n_\mathrm{b}$. This is not a constant bound, even when $\overline{n}_\mathrm{b} = n_\mathrm{b}$, unlike that found in Section 3.3.[3] Hence there is an incentive in target space to choose CNN architectures with smaller kernel matrices, or to only use a subset of patches when forming the pseudoinverse matrix. In the CNN architectures used in the experiments of Section 5.2, the ratio is empirically found to be around 7, which is considerably better than the upper-bound approximation of (22). Part of this improvement might be down to the fact that the backward pass of automatic differentiation can reuse the expensive matrix products and inverses computed in the forward pass.

5. Experiments

In this section we show the performance of the target-space method on the Two-Spirals benchmark problem, and on four classic small-image vision benchmark problems for convolutional neural networks, and then we demonstrate the target-space method on some bit-stream manipulation tasks and a sentiment analysis task for recurrent neural networks.

The experiments show the effectiveness of the target-space method, in ability to train deep networks and produce improved generalisation. There are improved generalisation results on the CNN vision benchmarks compared to the equivalent weight-space method applied to the same CNN architecture. In the recurrent network tasks, it shows the target-space method being able to solve problems with long time-sequences, which appear to be intractable in weight space.

All experiments were implemented using Python and Tensorflow v1.14 on a Tesla K80 GPU.[4] Shading in graphs indicates 95% confidence intervals as calculated by the Python Seaborn package.

5.1 Two-Spirals Experiments

The Two-Spirals classification problem consists of 194 two-dimensional training points, arranged in two interleaving spiral shapes, corresponding to the two output classes, each spiral revolving through three complete revolutions. The training and test sets are shown in Fig. 3. The test set was created as the angular midpoints between consecutive training points.

3. In future work, it is possible to remove this numerator factor of $k_\mathrm{h} k_\mathrm{w}$, since with a stride-length of 1 there is significant overlap between patches in the matrix A, and therefore optimisations can be made when forming AA^T.

4. Source code for experiments is available at `https://github.com/mikefairbank/dlts_paper_code`

A layered network architecture was used, with dimensions 2-5-5-5-2, and with all short-cut connections, following Riedmiller and Braun (1993). The cross-entropy loss function was used for training, and the tanh activation function used on all hidden layers, with softmax on the output layer.

Fig. 3 shows the output function of two trained networks, mapped to a single scalar output, and visually indicates that the solutions attained in target space are smoother and capture the essence of the problem better than in weight space.[5]

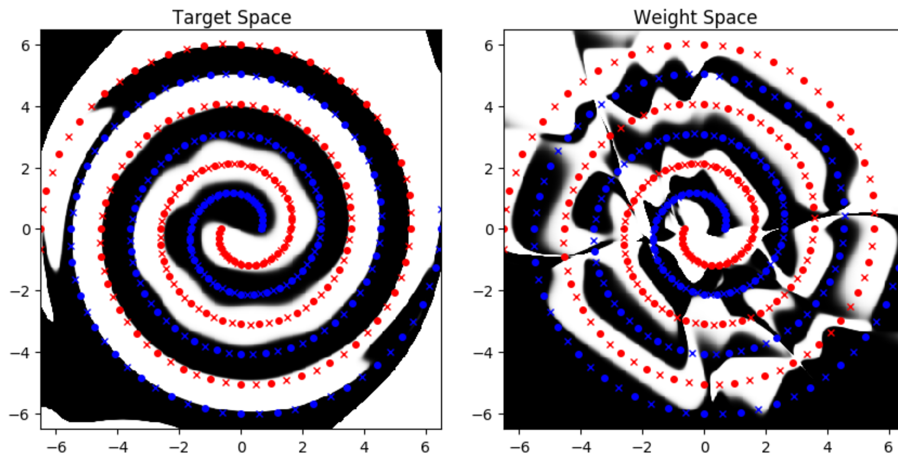

Figure 3: Typical results for the two-spirals trained network, after 4,000 Adam iterations; target space versus weight space. Red/blue crosses denote test set; circles denote the training set. Grey-scale background indicates network output for the given (x, y)-coordinate input. Smoothness of the target-space result shows how successful generalisation is more likely.

Fig. 4-left shows the problem being solved using gradient-descent with optimal learning rates empirically determined as $\eta = 10$ for target space and $\eta = 0.1$ for weight space. The results show that with optimal learning rates, the target-space algorithm can fully learn the two-spirals problem's training set, and generalise well to the test set, in around 1,000 epochs; compared to around 40,000 epochs for weight space to mostly learn the training set only. It does not seem possible to generalise as well to the test set in weight space, likely due to the unevenness appearing in Fig. 3-right.

Fig. 4-right shows results when the Adam optimiser was used, and shows a similar outcome. The learning rate used was 0.01, which was found to be beneficial to both target space and weight space on this problem. In this problem, the target-space gradient descent converges to a solution in fewer epochs than Adam in weight space.

These results all seem consistent with the target-space motivation for making the loss-function surface smoother, and the minima commonly found lead to better generalisation.

5. Although it should be noted that Levenberg Marquardt and conjugate gradient training can produce similarly nice solutions as the left figure.

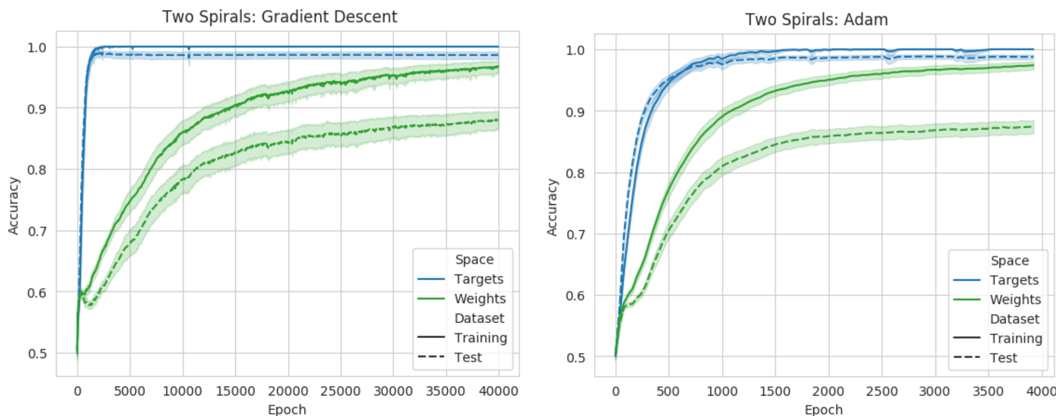

Figure 4: Results for Two-Spirals learning, using Batch Gradient Descent (on left) and Adam optimiser (on right).

In our implementation the amount of CPU time used was on average 3.5 times longer for each target-space training iteration compared to each weight-space iteration. In all experiments, the full data-set was used in all training batches ($n_{\mathrm{b}} = \overline{n}_{\mathrm{b}} = 194$). With target space, $\lambda = 0.001$ was used for equation (7), and initial targets were randomised using a truncated normal distribution with $\sigma = 1$, followed by the projection given by (14). For weight-space learning, the weights were randomised using the method of Glorot and Bengio (2010).

Fig. 5 shows the effectiveness of the Sequential Cascade untangling (SCU) variant against the Optimistic Cascade untangling (OCU) target-space algorithm (described in Section 2.4), and indicates that the SCU method is more stable and effective than the OCU method.

The same graph also shows that Batch Normalisation does not seem to help on this problem and network size, and in fact performs worse in weight space than without batch normalisation. Batch normalisation does significantly help though in the CNN experiments described in the next subsection.

5.2 CNN Experiments

In this set of experiments we train convolutional neural networks on the following four classic small-image classification problems:

- The MNIST digit dataset: 60,000 training samples of 28-by-28 grey-scale pixellated hand-written numeric digits, each labelled from 0-9, and a test set of 10,000 samples (LeCun et al., 2010).
- MNIST-Fashion dataset: 60,000 28x28 grayscale images of 10 labelled fashion categories, along with a test set of 10,000 images (Xiao et al., 2017).
- CIFAR10 dataset: 50,000 32x32 colour training images, labelled over 10 categories, and 10,000 test images (Krizhevsky et al., 2009).

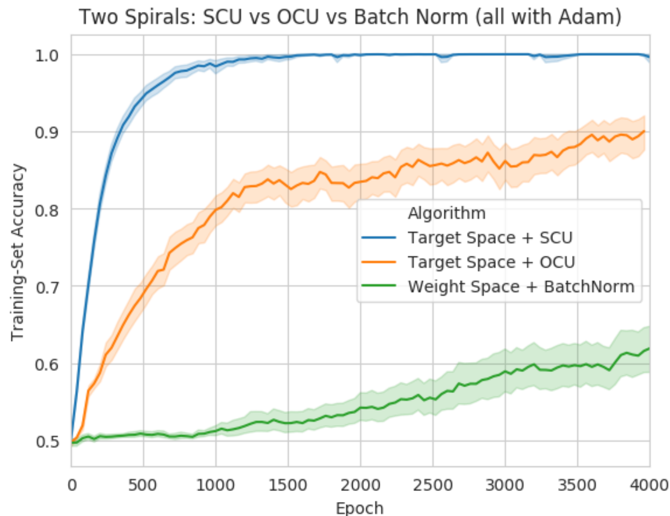

Figure 5: Results for Two-Spirals learning, using Adam Optimiser, comparing two forms of target space: Optimistic Cascade untangling (OCU) versus Sequential Cascade untangling (SCU), and against Batch Normalisation in weight space

- CIFAR100 dataset: 50,000 32x32 colour training images, labelled over 100 categories, and 10,000 test images (Krizhevsky et al., 2009).

All of these datasets were used as training data without any modification to the training images. For example, we did not use any data-augmentation techniques, such as image rescaling and distortion, which are known to help improve neural-network performance (and to be necessary to achieve state-of-the art classification performance).

The networks used here all had six compound convolutional/pooling layers, each of which consisted of a convolutional operation (with a square kernel of size $m \times m$, applied with stride-length 1 with "same" padding, and c output channels) followed by an application of the activation function, followed by (possibly) an application of max-pooling (with a square kernel of size $k \times k$, and applied with stride-length k). Each max pooling operation of side length k reduces the side-length of the image by factor k. Hence each compound convolutional layer can be summarised by a 3-tuple (m, c, k), with $k = 1$ if no max-pooling is used. Using this 3-tuple notation, the network architectures considered are listed in Table 1.

After the convolutional layers, the layer output is flattened, and then passed through a number of fully-connected (dense) layers, as described in Table 1.

All non-final layers used the "leaky-relu" activation function (Maas et al., 2013) defined by,

$$\text{LReL}(x) = \max(x, 0.2x), \tag{23}$$

and the final layer used softmax activation. Leaky-relu was found to slightly be better than the ReLU function, since it leaves fewer zeros in the activations which can potentially

Benchmark Problem	Convolutional Layers (Convolution size - Number of channels - Max Pool size)	Dense Layers
MNIST	(3-16-1)-(3-16-2)-(3-32-1)-(3-32-2)-(3-64-1)-(3-64-2)	128-10
MNIST-Fashion	(3-16-1)-(3-16-2)-(3-32-1)-(3-32-2)-(3-64-1)-(3-64-2)	128-10
CIFAR-10	(3-32-1)-(3-32-2)-(3-64-1)-(3-64-2)-(3-128-1)-(3-128-2)	128-10
CIFAR-100	(3-32-1)-(3-32-2)-(3-64-1)-(3-64-2)-(3-128-1)-(3-128-2)	512-128-100

Table 1: Convolutional Network Architectures considered for MNIST Problem

stall learning after the weights are initially randomised; and also can potentially make the Gramian matrix in (7) low rank.

The networks were trained with the cross-entropy loss function and the Adam optimizer, with learning rate 0.001 for weight-space learning, and 0.01 for target-space learning. Mini-batches of size $n_\text{b} = 100$ were randomly generated at each iteration, for computing the $\frac{\partial L}{\partial \vec{w}}$ gradient. A fixed mini-batch of size $\overline{n}_\text{b} = 100$ was used for the targets' input matrix $\overline{X}$.

In weight space, the weight initialisation used magnitudes defined by He et al. (2015), which are derived to work well with LReL. In target space, the targets values were all initially randomised with a truncated normal distribution with standard deviation 0.1, followed by the projection operation given by (14). $\lambda = 0.1$ was used in equation (7).

Results, after 400 epochs of training in each case, are shown in Table 2. The results show the target-space method helping generalisation performance, both with and without dropout (Srivastava et al., 2014), and when comparing against weight space both with and without batch-normalisation; and with ensemble architectures. The benefit of target space is noticeable in the latter 3 benchmark problems; mostly so in the most challenging benchmark problem, i.e. CIFAR100.

Algorithm (no dropout)	MNIST	MNIST-Fashion	CIFAR-10	CIFAR-100
Weight Space	99.32(±0.03)%	91.34(±0.01)%	75.85(±0.06)%	40.1(±0.4)%
Weight Space + Batch Normalisation	**99.53**(±0.02)%	**92.0**(±0.1)%	79.9(±0.1)%	45.82(±0.04)%
Target Space	99.4(±0.1)%	90.5(±1.8)%	**82.2**(±0.2)%	**50.7**(±0.1)%
Algorithm (with dropout)	MNIST	MNIST-Fashion	CIFAR-10	CIFAR-100
Weight Space	99.48(±0.02)%	93.3(±0.01)%	83.37(±0.05)%	52.8(±0.1)%
Weight Space + Batch Normalisation	99.59(±0.02)%	93.8(±0.2)%	85.94(±0.08)%	58.75(±0.02)%
Target Space	99.59(±0.01)%	**93.85**(±0.0)%	**87.55**(±0.09)%	**59.98**(±0.02)%
Algorithm (with dropout + ensemble)	MNIST	MNIST-Fashion	CIFAR-10	CIFAR-100
Weight Space	99.59 %	93.97 %	85.64 %	56.84 %
Weight Space + Batch Normalisation	99.64 %	94.3 %	87.7 %	61.82 %
Target Space	99.64 %	**94.47** %	**89.37** %	**63.1** %

Table 2: Test-Set Accuracies for CNN Experiments, on Standard Datasets

When dropout was used, it was applied with a dropout probability of 0.2 to all non-final dense layers, and all even-numbered convolutional layers. The results show that dropout provides useful benefit to both weight-space learning and target-space learning.

When dropout was used in target space, dropout was independently applied during both the feed-forward algorithm used to calculate $\frac{\partial L}{\partial \vec{w}}$ using the mini-batch input matrix X, and the feed-forward algorithm to map from target space to weight space using the fixed input matrix $\overline{X}$.[6]

When batch normalisation was used, it was applied to every convolutional layer and to every non-final dense layer. Batch normalisation is only applicable to weight-space learning. In target space learning, the targets for each layer already define the batch mean and standard-deviation which batch normalisation hopes to specify; making the combination of batch normalisation with target space redundant.

The error margins in Table 2 are calculated as the standard-deviations of just two trials; but are sufficiently small to convey the trend adequately.

When the ensemble of networks were used, the outputs of the two networks created in the two trials were averaged after softmax. Ensemble networks can usually generalise better than any of their constituent networks individually, assuming the outputs of the constituent networks are somewhat independent of each other. In this scenario the independence comes from different initial randomisation, different shuffling of mini-batches, and different choices of the $\overline{X}$ matrix used by the target-space algorithm. The results show that target space and weight space are assisted by using such an ensemble; even one comprised of only two networks.

The ratio of CPU time spent on the target-space algorithms compared to weight space algorithms were as shown in Table 3, all being in the ratio of approximately 7.

Algorithm-type	MNIST	MNIST-Fashion	CIFAR-10	CIFAR-100
Weight Space	1.3 hours	1.3 hours	2.3 hours	2.4 hours
Target Space	7 hours	7 hours	16 hours	16 hours

Table 3: CNN Training Time for 1 Network (400 epochs; using a GPU)

5.3 Bit-Stream Recurrent Neural-Network Experiments

In this section we describe two recurrent neural-network experiments regarding remembering and manipulating streams of bits.

The first experiment is to memorise and recall a random stream of bits. The RNN receives a new random bit at every time step t, and must output the bit it saw at the time step $t - N$, where N is the delay length. As the delay length is increased, the problem gets harder, since more bits must be memorised.

For example if the delay length is $N = 2$, and the RNN receives a bit stream such as “1,1,1,1,0,1” (with most recent bits appearing at the right) then the RNN is expected to produce an output stream “-,-,1,1,1,1”. (The first two outputs in the sequence, each indicated by here “-”, are ignored, since the delay length in this example is 2.)

The neural network has architecture $1 - (N + 3) - 2$, with the hidden layer being fully connected to itself with recurrent connections (corresponding to setting $c_{\mathrm{L}} = 3$ in Fig. 2). The hidden layer used tanh activation functions, and the final layer used softmax with

6. Generalisation results were noticeably worse if dropout in target space was applied to either one of these two stages without the other.

cross-entropy loss function. The loss function was made to ignore the first N outputs in the stream (since these are undefined).

The $N+3$ recurrent hidden nodes are enough to allow the network to remember the most recent N bits (with 3 spare nodes to add a little flexibility in solution), as required; for example the RNN could learn manipulate the remembered bits with a rotate-right bit-wise operation, so as to successfully queue and recall the bits, and forget about bits older than N.

A batch size of 8,000 random bit streams of length $n_\mathrm{t} = N + 50$ was used to train the network. Random mini-batches of size $n_\mathrm{b} = 100$ were used during each training iteration. A fixed mini-batch of size $\overline{n}_\mathrm{b} = 100$ with $\overline{n}_\mathrm{t} = n_\mathrm{t}$ was used for the target-space matrices $\overline{X}^{(t)}$.

In weight space, the weight initialisation used magnitudes defined by Glorot and Bengio (2010). In target space, the targets values were randomised with a truncated normal distribution with standard deviation 1, followed by a projection by equation (14). This projection step seemed to improve results for the target-space experiments.

The networks were trained with 50,000 iterations of Adam optimiser, with learning rate 0.001 for both weight-space and target space, and with $\lambda = 0.1$ for target space.

A result was considered a success if a classification accuracy $\geq 99\%$ was achieved on the test set at any training iteration; otherwise it was a failure.

Results are shown in Fig. 6 for various delay lengths. They show that the target-space method is able to learn sequences with a delay length of around two to three times as long as the weight-space methods are capable of, with a significantly less steep rise in the number of training iterations required for success; and that the target-space SCU method is significantly stronger than the target-space OCU method.

For comparison, an extra experiment was made using an LSTM network. Here the $N+3$ hidden nodes were replaced by $N+3$ LSTM memory cells. The LSTM network was trained in weight space, again using Adam for 50,000 iterations. Results are shown in the same Fig. 6. This trial shows that the LSTM network does not seem to help in solving this problem in weight-space.

In a second RNN experiment, we modify the task from pure mermorization into one of binary addition. In this experiment, the target output is the binary sum of the stream of bits with the N-step delayed stream. To ease binary addition, the stream is assumed to arrive in bit-wise little-endian form.

For example, if $N = 2$, and the bit stream received is "1,0,1,1,0,1", then the target output stream that the RNN must learn is "-,-,0,0,0,1", which is calculated by binary addition: 1101+1011=00011. Here the target output stream terminated before the final carry bit could be delivered, so only the 0001 remained.

As this problem was slightly harder than the previous one, since the relationship between the target-bit sequence and the past sequence is quite well disguised (the relationship has similarities to a delayed XOR problem but there is also a hidden carry-bit process to discover), we gave the recurrent network $N+5$ hidden recurrent nodes, i.e. two more than previously.

Results are shown in Fig. 7. The experimental conditions are otherwise unchanged from the previous RNN experiment.

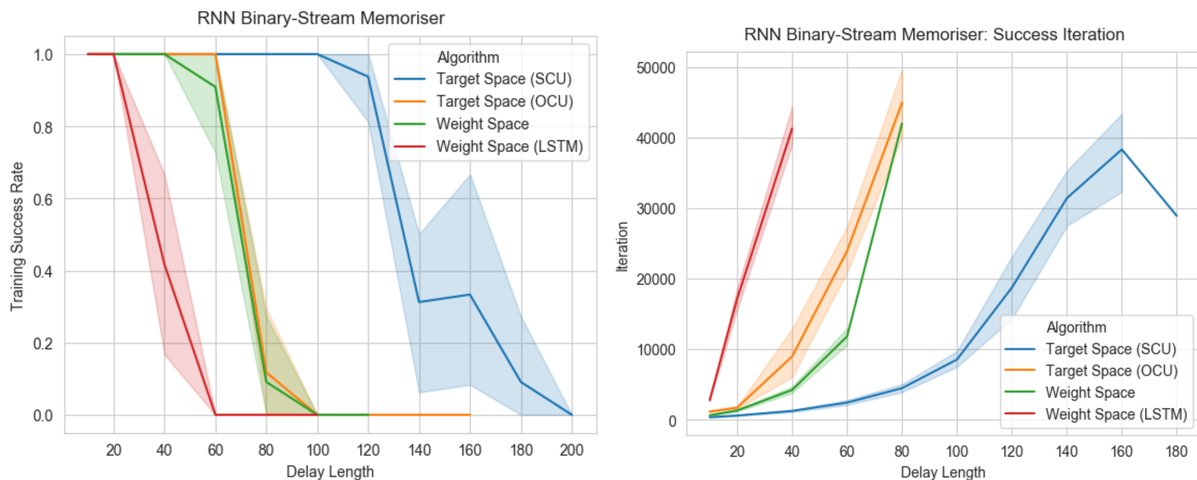

Figure 6: Memorisation of a delayed binary stream of bits using a RNN. The left graph shows the ratio of trials which were successful in correctly learning > 99% of the output bits bits correctly (in a test set). The right-hand graph shows, for those successful trials, the average iteration number at which success was first achieved.

In this experiment the strength of the target-space methods are again shown, with the SCU method again being capable of coping with delay lengths two to three times as long as the weight-space methods, and with better scaling of the number of iterations required. The strength of the SCU method's results confirms the value of lines 8 and 10 in Alg. 6, when compared to the OCU method.

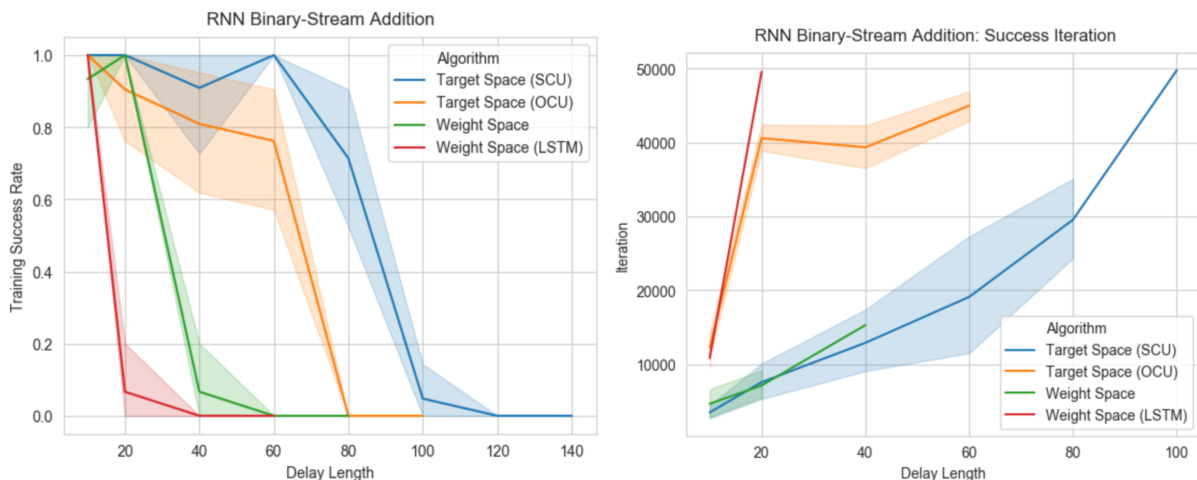

Figure 7: Addition of a delayed binary stream of bits using a RNN.

In both of these RNN experiments, the SCU method significantly beats the LSTM network. It therefore seems that the exploding-gradients problem (which target-space networks are designed to address) is more significant in this problem than the vanishing-gradients problem (which LSTM networks are designed to address). A complication in making this comparison is that Adam was used. Adam might have been picking up and aggressively accelerating tiny components of the gradients in target space, thus counteracting the vanishing gradients and helping the target-space methods compete with LSTM. Possibly in a more noisy problem environment, it will not be possible to accelerate such tiny gradients due to the low signal-to-noise ratio. In that case a combination of LSTM plus target space could be attempted.

5.4 RNN Movie-Review Sentiment Analysis

In this final experiment we trained a RNN to solve the natural-language processing task of sentiment analysis for 50,000 movies reviews from the Internet Movie Database (IMDB) website. In this binary classification task, each review is labelled as either positive or negative. The dataset was obtained from the Tensorflow/Keras packages, with a 50-50 training/test-set split, using options of only including the top 5000 most frequent words, and padding/truncating all reviews to a length of 500 words each.

A word-embedding vector of length 32 was used to encode each word from the vocabulary of size 5000 (Bengio et al., 2003; Mikolov et al., 2013). Once each word is converted into an embedded vector, the neural-network architecture is the same as in the previous experiment, but with 32 inputs, 100 nodes in the recurrent layer, and two output nodes. Each embedded word of a review is fed to the RNN one-by-one, making the sequence length $n_\mathrm{t} = 500$. Only the final output matrix of the neural network, $Y^{(500)}$, is observed.

Results are shown in Fig. 8 and are summarised in Table 4, and show that the target-space method's performance slightly exceeds that of the LSTM network, and significantly exceeds ordinary neural networks trained in weight space.

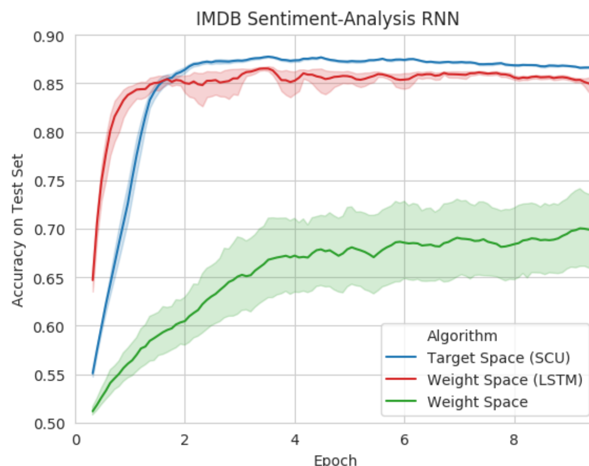

Figure 8: Results for Movie Sentiment Analysis RNN Problem.

Algorithm / Network Type	Best Test Accuracy	Total GPU time for 10 Epochs
Weight Space	71.1(±8.3)%	9.1 mins
Weight Space + LSTM	86.7(±0.2)%	19.1 mins
Target Space	87.7(±0.1)%	13.9 mins

Table 4: Results for Movie Sentiment Analysis RNN Problem

All neural networks were trained using Adam with learning rate 0.001, and mini-batch sizes of $n_{\mathrm{b}} = 40$. The target-space algorithm used $\lambda = 0.001$. Weights and targets were initially randomised as in the previous subsection. Word embeddings were also initially randomised (using a normal distribution with $\mu = 0$ and $\sigma = 0.1$). Hence all weight and target matrices, and the embedding vectors, were learned in an end-to-end training process.

To customise the target-space method to handle word embeddings efficiently, a fixed sequence of target-space input matrices $\overline{X}^{(t)}$ was chosen, for a sequence length of just $\overline{n}_{\mathrm{t}} = 60$, and mini-batch size $\overline{n}_{\mathrm{b}} = 40$. For efficiency, it was chosen that these matrices would represent some already-embedded word sequences. Hence each matrix $\overline{X}^{(t)} \in \mathbb{R}^{32\times 40}$, for $t = 1, \ldots, 60$. Each of the $\overline{X}^{(t)}$ matrices was generated using a uniform random distribution in the range [-1,1], and then held constant throughout training. The lower sequence length $\overline{n}_{\mathrm{t}} = 60$ improves the algorithmic complexity factor (given at the end of Section 4.1), and results in a more competitive target-space training time in Table 4. Even though this sequence length ($\overline{n}_{\mathrm{t}} = 60$) was less than the true sequence length ($n_{\mathrm{t}} = 500$), the combination of fixed matrices $\overline{X}^{(t)}$ and target matrices $T_j^{(t)}$ provide enough information to define the weight matrices $W_{[0:j]}$ unambiguously using Alg. 6; even though $\overline{X}^{(t)}$ are fixed random matrices, and therefore do not conform to any valid movie-review style of writing. Hence the learning gradient $\frac{\partial L}{\partial \vec{w}}$ (which now includes the gradient of the learnable embedding matrix, $W_{embed} \in \mathbb{R}^{5000\times 32}$), can be calculated in weight space, using the full sequence lengths (500), and then converted to a target-space gradient $\frac{\partial L}{\partial \vec{\tau}}$, using Alg. 6, followed by automatic differentiation. To compute the gradient $\frac{\partial L}{\partial W_{embed}}$ in target space, in order to optimise those learnable variables too, we just used its value in weight space, without any modification.

6. Conclusions

The target-space method provides an alternative search space in which to train deep and recurrent neural networks. The theory and experiments indicate that the loss-function surfaces being optimised are indeed smoother and easier to optimise in target-space than in weight space. This increased smoothness potentially leads to easier solution of problems and potentially leads to better generalisation capabilities in the final neural networks produced.

Using target space comes at an added computational expense. In fully connected networks, where the batch sizes for X and $\overline{X}$ roughly match, this is usually a modest constant cost of approximately 3 or 4 times as much computation per training iteration. With CNNs it can be more, being around 7 times in the CNN architectures considered in this paper, and more so if wider convolutional kernels are used. With the RNN experiments, which can

be considered as extremely deep and narrow networks, the timings were of similar order of magnitude between weight space and target space. It is hoped that by careful choice of architecture, focusing on deeper networks with narrower hidden layers (possibly with several narrower layers running in parallel, which has already been proven as a powerful design by Xie et al., 2017, in their “ResNeXt” CNN design), and avoiding pattern-by-pattern learning, these costs can be minimised.

It has been shown how to combine mini-batching with target space. The lack of mini-batching has historically been a major Achilles heel in the adoption of some previous sophisticated optimisers (for example conjugate gradients or Levenberg-Marquardt), with very large datasets.

Target-space methods are particularly promising in recurrent neural-network environments. In the examples given, problems with sequence lengths that were previously intractable have been solved, and the LSTM results were surpassed in a natural-language problem. This is despite the fact that LSTM networks have extra features, such as memory gates, which make the learning task easier, yet the target-space learning has still managed to make ordinary RNNs outperform them. In the feed-forward problems given, target space has consistently produced better generalisation in deeper neural networks.

The theoretical motivation for target space, in that using targets should be able to untangle the cascades of changes caused during training, with a beneficial outcome, appears to be feasible. Hence target space aims to directly address the recognised “exploding-gradients” problem which exists in deep learning.

Regarding a hypothetical future of neural networks being able to produce simple programs similar to those formed by human programmers, we hypothesise that whenever a neural network gets to a really interesting point of training, then the neural activations will often all be very close to their firing thresholds, and the exploding-gradients problem becomes really significant in progressing training any further. For example if the neural network training process had somehow successfully managed to build a series of interlocking XOR gates, which were almost all working well together so as to implement a conventional computer program out of those logic gates, then the scrambling of behaviour from any potential infinitesimal weight change will always make learning destabilise in weight space. The target-space approach is designed to be helpful in these circumstances, and would seem to have more chance of making further progress than a simple weight-space search would.

Our experimental results with recurrent neural networks over long time sequences combined with data-processing outperform the equivalent LSTM networks. Hence it seems that in those problems at least, the exploding-gradients problem is more significant than vanishing gradients; at least when Adam is allowed to accelerate the small gradients in target space. This is particularly paradoxical when it is noted that the objective of the target-space cascade untangling is to dampen down learning gradients even more, thus amplifying the vanishing-gradients problem.

Many significant deep-learning innovations exist in prior published work. These include the closely-related method of batch normalisation, plus modern activation functions, optimisers, and weight-initialisation techniques. Many of these are more computationally efficient than target space, but are maybe slightly less effective; and some can be combined with target space. Sophisticated neural architectures, such as LSTM, CNNs, and more recently, attention models, Differentiable Neural Computers and Neural Turing Machines

(Graves et al., 2014, 2016), exist, which all add to neural-network functionality, and which could all in-principle be trained in target space. So in final conclusion, the target-space method seems to be a powerful additional tool which has tremendous potential for the enhancement of deep learning.

Appendix A. Derivation of Algorithm 4

A.1 Preliminary Definitions

Single-Entry Matrix:
Define $[J^{ij}]$ to be the *single-entry matrix* with element at row i and column j equal to $\begin{cases} 1 & \text{if } m=i \text{ and } n=j \\ 0 & \text{otherwise} \end{cases}$ (Petersen and Pedersen, 2012). This is useful when differentiating a matrix with respect to one of its elements, since $\frac{\partial A}{\partial A^{ij}} = [J^{ij}]$, with $[J^{ij}]$ having the same dimensions as A.
Raised Indices Notation:
Define upper indices (without parentheses) after a matrix variable to indicate the matrix element, so that for example A^{ij} is the element of A with row index i and column index j.

Define raised indices $[ij]$ in square brackets after a scalar function $f(i,j)$ to mean the whole matrix whose element at row i and column j is $f(i,j)$. For example, $\left(A^{ij}\right)^{[ij]} \equiv A$, and $\left(A^{ji}\right)^{[ij]} \equiv A^T$.
Frobenius Inner Product, $\langle A, B\rangle_F$
For two $m \times n$ matrices A and B, define $\langle A, B\rangle_F := \sum_{\forall i,j} A^{ij}B^{ij}$. This inner product is useful when using the chain rule; for example, if X, Y and Z are matrices with $X = X(Y)$ and $Y = Y(Z)$ then $\frac{\partial X^{mn}}{\partial Z^{ij}} = \left\langle \frac{\partial X^{mn}}{\partial Y}, \frac{\partial Y}{\partial Z^{ij}} \right\rangle_F$. Furthermore, if $L(X)$ is a scalar function, then $\frac{\partial L}{\partial Y} = \left(\left\langle \frac{\partial X}{\partial Y^{ij}}, \frac{\partial L}{\partial X}\right\rangle_F\right)^{[ij]}$.

A.2 Basic Lemma for Combining Frobenius Inner Product with Single-entry Matrix

A useful result for combining the inner product with $[J^{ij}]$ is

$$\left(\left\langle A[J^{ij}]B, C\right\rangle_F\right)^{[ij]} = A^T C B^T \tag{24}$$

since $\left\langle A[J^{ij}]B, C\right\rangle_F = \sum_{mn}(A[J^{ij}]B)^{mn}C^{mn} = \sum_{mn}\left(\sum_{pq} A^{mp}[J^{ij}]^{pq}B^{qn}\right)C^{mn}$
$= \sum_{mn}(A^{mi}B^{jn}C^{mn}) = (A^T C B^T)^{ij}$.

Similarly,

$$\left(\left\langle A[J^{ij}]^T B, C\right\rangle_F\right)^{[ij]} = B C^T A \tag{25}$$

A.3 Matrix Differentiation

Differentiating a scalar by a matrix gives an identically dimensioned matrix, e.g. $\left(\frac{\partial L}{\partial X}\right)^{ij} := \frac{\partial L}{\partial X^{ij}}$. Similarly for differentiating a matrix by a scalar: $\left(\frac{\partial X(a)}{\partial a}\right)^{ij} := \frac{\partial X^{ij}(a)}{\partial a}$.

Matrix differentiation follows the usual product rule:

$$\frac{\partial AB}{\partial X^{mn}} = \frac{\partial A}{\partial X^{mn}}B + A\frac{\partial B}{\partial X^{mn}}. \tag{26}$$

For example, if A,B and C are constant matrices, then

$$\frac{\partial AXBXC}{\partial X^{mn}} = A[J^{mn}]BXC + AXB[J^{mn}]C.$$

The derivative of an inverse matrix A^{-1} is $\frac{\partial A^{-1}}{\partial A^{ij}} = -A^{-1}[J^{ij}]A^{-1}$ (Brookes, 2011). Combining this with the product rule gives

$$\begin{aligned}\frac{\partial (BB^T+\lambda I)^{-1}}{\partial B^{ij}} &= -(BB^T+\lambda I)^{-1}\left([J^{ij}]B^T + B[J^{ij}]^T\right)(BB^T+\lambda I)^{-1} \\ &= -(BB^T+\lambda I)^{-1}[J^{ij}]B^\dagger - (B^\dagger)^T[J^{ij}]^T(BB^T+\lambda I)^{-1} \end{aligned} \tag{27}$$

And so,

$$\begin{aligned}\frac{\partial B^\dagger}{\partial B^{ij}} =& \frac{\partial B^T(BB^T+\lambda I)^{-1}}{\partial B^{ij}} && \text{(by (7))} \\ =& [J^{ij}]^T(BB^T+\lambda I)^{-1} - B^T\frac{\partial (BB^T+\lambda I)^{-1}}{\partial B^{ij}} && \text{(by product rule)} \\ =& [J^{ij}]^T(BB^T+\lambda I)^{-1} - \left(B^\dagger[J^{ij}]B^\dagger + B^\dagger B[J^{ij}]^T(BB^T+\lambda I)^{-1}\right) && \text{(by (27))} \\ =& (I-B^\dagger B)[J^{ij}]^T(BB^T+\lambda I)^{-1} - B^\dagger[J^{ij}]B^\dagger && (28)\end{aligned}$$

A.4 Ordered Partial Derivatives

Define the notation $\frac{\partial}{\partial^*}$ to be the *ordered* partial derivatives (Werbos, 1974), which take into account cascading changes to all later layers' weights and activations by Algorithm 3. For example $\frac{\partial \vec{w}}{\partial^* A_j^{mn}}$ describes how all the layers' weights would change according to Algorithm 3 if a small perturbation was forced to occur to A_j^{mn}.

For a layer j, define $\delta A_j := \left(\frac{\partial \vec{w}}{\partial^* A_j^{mn}}\frac{\partial L}{\partial \vec{w}}\right)^{[mn]}$. This matrix accounts for what effect a small change to A_j will have on L, solely through the effect of cascading changes to later layers' weights via alg. 3. Note that δA is subtly different from $\frac{\partial L}{\partial^* A}$ since at the final layer $\delta A_{n_\mathrm{L}} = 0$ (since there are no later layers whose weights can change), but $\frac{\partial L}{\partial^* A_{n_\mathrm{L}}} = \frac{\partial L}{\partial Y} \neq 0$. Similarly, define $\delta S_j := \left(\frac{\partial \vec{w}}{\partial^* S_j^{mn}}\frac{\partial L}{\partial \vec{w}}\right)^{[mn]}$ and $\delta W_{[0:j]} := \left(\frac{\partial \vec{w}}{\partial^* W_{[0:j]}^{mn}}\frac{\partial L}{\partial \vec{w}}\right)^{[mn]}$.

A.5 Derivation of Algorithm 4

Define $\delta A_{[0:j)}$ to be the composite of δA_j matrices in the same way that the $A_{[0:j)}$ matrices are composed of A_j matrices, analogous to Eq. (2).

The matrices $W_{[0:j]}$, $A_{[0:j)}$, T_j, Y_j, A_j and S_j are for an arbitrary layer j. Throughout the following, all these matrices refer to the same subscripted value of j, therefore we omit this subscript to ease presentation. To avoid the clash of variable names between A_j and $A_{[0:j)}$, we define $B \equiv A_{[0:j)}$ and $A \equiv A_j$ as shorthand.

First we give useful results for $\frac{\partial W}{\partial B^{mn}}$ and δW:

$$\begin{aligned}\frac{\partial W}{\partial B^{mn}} =& T\left[(I-B^\dagger B)[J^{mn}]^T(BB^T+\lambda I)^{-1} - B^\dagger[J^{mn}]B^\dagger\right] && \text{(by (6) and (28))} \\ =& (T-S)[J^{mn}]^T(BB^T+\lambda I)^{-1} - W[J^{mn}]B^\dagger && \text{(by (6) and (4))} \quad (29)\end{aligned}$$

The derivation for $\delta W = \left(\frac{\partial \vec{w}}{\partial^* W_{[0:j]}^{mn}}\frac{\partial L}{\partial \vec{w}}\right)^{[mn]}$ in Equation (30) starts by adding two terms. The first term, $\frac{\partial L}{\partial W}$, accounts for the contribution from the changing weights in that par-

ticular layer. The second term, $\left(\langle \delta S, \frac{\partial S}{\partial W^{pq}} \rangle_F\right)^{[pq]}$, accounts for the cascading changes to all later layers' weights (by the definition of δS).

$$
\begin{aligned}
\delta W =& \frac{\partial L}{\partial W} + \left(\left\langle \delta S, \frac{\partial S}{\partial W^{mn}} \right\rangle_F \right)^{[mn]} \\
=& \frac{\partial L}{\partial W} + (\langle \delta S, [J^{mn}]B \rangle_F)^{[mn]} && \text{(by (4) and (26))} \\
=& \frac{\partial L}{\partial W} + (\delta S)B^T && \text{(by (24))} \qquad (30)
\end{aligned}
$$

To derive a formula that calculates $\frac{\partial L}{\partial T}$ for a particular layer given $\frac{\partial L}{\partial \vec{w}}$, we first note that changing T^{mn} for one layer will initially just change the weights of that layer, according to $\frac{\partial W}{\partial T^{mn}}$. Then cascading changes to the later layers' weights will occur via Algorithm 3, as a consequence of this initial single layer's change of weights, and therefore all these cascading effects are represented by δW. Combining these two factors with the Frobenius inner-product gives:

$$
\begin{aligned}
\frac{\partial L}{\partial T} &= \left(\left\langle \delta W, \frac{\partial W}{\partial T^{mn}} \right\rangle_F \right)^{[mn]} \\
&= \left(\left\langle \delta W, [J^{mn}]B^\dagger \right\rangle_F \right)^{[mn]} && \text{(by (6) and (26))} \\
&= \left(\frac{\partial L}{\partial W} + (\delta S)B^T \right) (B^\dagger)^T && \text{(by (24) and (30))} \qquad (31)
\end{aligned}
$$

This requires calculation of the δS matrices for each layer. Since $A^{mn} = g(S^{mn})$, the chain rule gives

$$
\delta S = \delta A \odot g'(S) \tag{32}
$$

The derivation for δB is given in Equation (33). The first line of this derivation consists of two terms which are present, respectively, because changing B will change the weights for that layer directly (via the equation $W = TB^\dagger$), and will also change the sums for that layer directly (via the equation $S = WB$). The effects of these two changes are what the terms δW and δS, respectively, are defined to represent.

$$
\begin{aligned}
\delta B =& \left(\left\langle \delta W, \frac{\partial W}{\partial B^{mn}} \right\rangle_F + \left\langle \delta S, \frac{\partial S}{\partial B^{mn}} \right\rangle_F \right)^{[mn]} \\
=& \left(\left\langle \delta W, \left((T-S)[J^{mn}]^T (BB^T + \lambda I)^{-1} - W[J^{mn}]B^\dagger \right) \right\rangle_F + \langle \delta S, W[J^{mn}] \rangle_F \right)^{[mn]} \\
& && \text{(by (29), (4) and (26))} \\
=& (BB^T + \lambda I)^{-1} (\delta W)^T (T-S) - W^T (\delta W)(B^\dagger)^T + W^T \delta S && \text{(by (24) and (25))} \\
=& W^T \left(\delta S - \frac{\partial L}{\partial T} \right) + \left[(BB^T + \lambda I)^{-1} \left(\frac{\partial L}{\partial W} + (\delta S)B^T \right)^T (T-S) \right] && \text{(by (31) and (30))}
\end{aligned} \tag{33}
$$

This enables us to find δB from δS for a particular layer. Since $\delta A_{[0:j)} \equiv \delta B$ is composed of δA_{j-1}, and $\delta A_{n_\mathrm{L}} = 0$ we can calculate the δA matrices backwards, layer by layer. Thus equations (31), (32) and (33) give lines 4, 3, and 5 of Alg. 4 respectively.

Appendix B. Proof that a stationary point in target space corresponds to a stationary point in weight space

In this appendix we show that if $\vec{\tau}^*$ is a stationary point for the target-space problem, i.e. $\frac{\partial L}{\partial \vec{\tau}}\big|_{\vec{\tau}=\vec{\tau}^*} = 0$, then the corresponding vector of weights $\vec{w}^*$ obtained from $\vec{\tau}^*$ through Algorithm 3 is a stationary point for the resulting weight-space problem, i.e. $\frac{\partial L}{\partial \vec{w}}\big|_{\vec{w}=\vec{w}^*} = 0$.

After a preliminary definition and two lemmas, the main theorem and proof follows.

Definition: Let

$$A^{\ddagger} := A + \lambda\,(A^{+})^T. \tag{34}$$

where A^+ denotes the (non-regularised) Moore-Penrose pseudoinverse (Golub and Van Loan, 2013, Section 5.5.2).

Lemma 1 *For any real-valued matrix A, the following identity holds: $A\,A^{\dagger}A^{\ddagger} = A$.*

Proof *We use the singular value decomposition (SVD) to prove this. Let the shape of A be $m \times n$, and the rank of A be r. Let the full SVD of A be given by*

$$A = USV^T, \tag{35}$$

where the matrices $U \in \mathbb{R}^{m\times m}$, $S \in \mathbb{R}^{m\times n}$ and $V \in \mathbb{R}^{n\times n}$, the only non-zero elements of S are on its leading diagonal, and where U and V are orthogonal. Since A is rank r, the matrix S will have its first r diagonal elements as non-zero and the remaining elements all zero. Hence we can partition S into block-matrix form as follows:

$$S = \begin{pmatrix} \Sigma & 0 \\ 0 & 0 \end{pmatrix}, \tag{36}$$

where Σ is a diagonal matrix of shape $r \times r$, and the zeros are rectangular matrices of appropriate shape so as to make $S \in \mathbb{R}^{m\times n}$. Since Σ is square and full rank, its Moore-Penrose pseudoinverse simplifies into an ordinary inverse:

$$\Sigma^+ = \Sigma^{-1}. \tag{37}$$

Using the SVD, and repeatedly cancelling orthogonal self-products such as U^TU and V^TV, we can write:

$$A^{\dagger} = VS^T(SS^T + \lambda I)^{-1}U^T \qquad \textit{(by (7))} \tag{38}$$

$$A^{+} = VS^{+}U^T \qquad \textit{(Moore-Penrose SVD)} \tag{39}$$

$$A^{\ddagger} = U(S + \lambda S^{+})V^T \qquad \textit{(by (34) and (39))} \tag{40}$$

Substituting these into the left-hand side of the lemma's identity, and cancelling orthogonal self-products, gives,

$$
\begin{aligned}
A\,A^{\dagger}A^{\ddagger} &= USS^T(SS^T+\lambda I)^{-1}(S+\lambda S^{+})V^T \\
&= U\begin{pmatrix}\Sigma^2 & 0\\ 0 & 0\end{pmatrix}\begin{pmatrix}(\Sigma^2+\lambda I)^{-1} & 0\\ 0 & \lambda^{-1}I\end{pmatrix}\begin{pmatrix}\Sigma+\lambda\Sigma^{-1} & 0\\ 0 & 0\end{pmatrix}V^T && \text{(by (36) and (37))}\\
&= U\begin{pmatrix}\Sigma^2(\Sigma^2+\lambda I)^{-1}(\Sigma+\lambda\Sigma^{-1}) & 0\\ 0 & 0\end{pmatrix}V^T && \text{(block multiplication)}\\
&= U\begin{pmatrix}\Sigma(\Sigma^2+\lambda I)^{-1}(\Sigma^2+\lambda\Sigma\Sigma^{-1}) & 0\\ 0 & 0\end{pmatrix}V^T && \text{(commute diagonal matrices)}\\
&= A && \text{(by (35) and (36))}
\end{aligned}
$$

This proves the lemma. ■

Remark: Lemma 1 is analogous to the identity for the non-regularised Moore-Penrose pseudoinverse given by $A\,A^{+}A = A$ (Golub and Van Loan, 2013, Section 5.5.2), which also holds for any real-valued matrix A.

Lemma 2 *When the weights are calculated from the targets by Algorithm 3, given any layer j, such that $j = n_{\mathrm{L}}$ or $\frac{\partial L}{\partial W_{[0:k]}} = 0$ for all $k > j$, we have $\frac{\partial L}{\partial T_j} = 0 \Rightarrow \frac{\partial L}{\partial W_{[0:j]}} = 0$.*

Proof *First, let us write explicitly the derivative of the loss function L with respect to the weights of layer j:*

$$
\begin{aligned}
\frac{\partial L}{\partial W_{[0:j]}} &= \left(\left\langle \frac{\partial L}{\partial S_j}, \frac{\partial S_j}{\partial W^{mn}_{[0:j]}}\right\rangle_F\right)^{[mn]} \\
&= \left(\left\langle \frac{\partial L}{\partial S_j}, [J^{mn}]A_{[0:j)}\right\rangle_F\right)^{[mn]} && \text{(by (4) and (26))}\\
&= \frac{\partial L}{\partial S_j}\,A^T_{[0:j)}, && \text{(by (24))} \qquad (41)
\end{aligned}
$$

And similarly, explicitly state its derivative with respect to the targets of the same layer:

$$
\frac{\partial L}{\partial T_j} = \left(\left\langle \frac{\partial L}{\partial S_j} + \sum_{k>j}\left(\left\langle \frac{\partial L}{\partial W_{[0:k]}}, \frac{\partial W_{[0:k]}}{\partial S^{pq}_j}\right\rangle_F\right)^{[pq]}, \frac{\partial S_j}{\partial T^{mn}_j}\right\rangle_F\right)^{[mn]}. \qquad (42)
$$

In this equation, instead of following (31)*, we have used the chain rule to produce an expression that explicitly connects $\frac{\partial L}{\partial T}$ to $\frac{\partial L}{\partial S}$. The summation in* (42) *evaluates to δS_j defined in Section A.4, which accounts for the effects of the weights in all later layers $k > j$ which will change by Alg. 3 as a result of a change to T_j. Following on from the initial assumptions of this lemma, which were that either $j = n_{\mathrm{L}}$ or the condition $\frac{\partial L}{\partial W_{[0:k]}} = 0$ holds for all $k > j$,*

therefore the summation vanishes and (42) reduces to:

$$\begin{aligned}
\frac{\partial L}{\partial T_j} &= \left(\left\langle \frac{\partial L}{\partial S_j}, \frac{\partial S_j}{\partial T_j^{mn}} \right\rangle_F\right)^{[mn]} \\
&= \left(\left\langle \frac{\partial L}{\partial S_j}, \frac{\partial\left(T_j\left(A_{[0:j)}\right)^{\dagger} A_{[0:j)}\right)}{\partial T_j^{mn}} \right\rangle_F\right)^{[mn]} && \text{(by (4) and (6))} \\
&= \left(\left\langle \frac{\partial L}{\partial S_j}, [J^{mn}]\left(A_{[0:j)}\right)^{\dagger} A_{[0:j)} \right\rangle_F\right)^{[mn]} && \text{(by (26))} \\
&= \frac{\partial L}{\partial S_j} A_{[0:j)}^T \left(A_{[0:j)}^T\right)^{\dagger}. && \text{(by (24))} \qquad (43)
\end{aligned}$$

Aiming from a contradiction, let us assume that $\frac{\partial L}{\partial W_{[0:j]}} \neq 0$. *Then one can choose an appropriately sized column vector* u *such that* $\frac{\partial L}{\partial W_{[0:j]}} u \neq 0$. *We now consider a second vector* $v = \left(A_{[0:j)}^T\right)^{\ddagger} u$, *using (34), and write:*

$$\begin{aligned}
\frac{\partial L}{\partial T_j} v &= \frac{\partial L}{\partial S_j} A_{[0:j)}^T \left(A_{[0:j)}^T\right)^{\dagger} v && \text{(by (43))} \qquad (44) \\
&= \frac{\partial L}{\partial S_j} A_{[0:j)}^T \left(A_{[0:j)}^T\right)^{\dagger} \left(A_{[0:j)}^T\right)^{\ddagger} u && \text{(by the definition of } v\text{)} \\
&= \frac{\partial L}{\partial S_j} A_{[0:j)}^T u && \text{(by Lemma 1)} \\
&= \frac{\partial L}{\partial W_{[0:j]}} u. && \text{(by (41))} \qquad (45)
\end{aligned}$$

While we initially assumed that the last line (45) is non-zero, the first line (44) must be zero, due to $\frac{\partial L}{\partial \vec{\tau}} = 0$. *This contradiction proves the lemma.* ■

Theorem 3 *When the weights are calculated from the targets by Algorithm 3, we have* $\frac{\partial L}{\partial \vec{\tau}} = 0 \implies \frac{\partial L}{\partial \vec{w}} = 0$.

Proof *We shall prove this result by induction, by first showing it holds for the last layer (used as the base case), and then showing that if it holds for all subsequent layers then it must also hold for the current layer (the inductive step).*

Lemma 2 explicitly handles the case where $j = n_{\mathrm{L}}$, *thus the base-case claim, that* $\frac{\partial L}{\partial \vec{\tau}} = 0$ *implies* $\frac{\partial L}{\partial W_{[0:n_{\mathrm{L}}]}} = 0$, *is true. Next we consider the inductive step, i.e. that if* $\frac{\partial L}{\partial \vec{\tau}} = 0$ *and* $\frac{\partial L}{\partial W_{[0:k]}} = 0$ *for all* $k > j$, *then* $\frac{\partial L}{\partial W_{[0:j]}}$ *must be zero. Again, Lemma 2 applies here, since it explicitly applies to* $\frac{\partial L}{\partial W_{[0:k]}} = 0$ *for all* $k > j$, *and therefore the inductive step is also true. This completes the proof by induction.* ■

This final theorem concludes the proof that $\frac{\partial L}{\partial \vec{\tau}} = 0$ implies $\frac{\partial L}{\partial \vec{w}} = 0$, i.e. that a stationary point for the target-space problem is also a stationary point for the corresponding weight-space problem obtained through Algorithm 3.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

Amir F Atiya and Alexander G Parlos. New results on recurrent network training: unifying the algorithms and accelerating convergence. *IEEE transactions on neural networks*, 11 (3):697–709, 2000.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Dimitri P Bertsekas. *Nonlinear programming.* Athena Scientific Belmont, MA, 2 edition, 1999.

Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

M. Brookes. The matrix reference manual, 2011. URL `http://www.ee.ic.ac.uk/hp/staff/dmb/matrix/intro.html`.

Miguel Carreira-Perpinan and Weiran Wang. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics*, pages 10–19, 2014.

Miguel Á Carreira-Perpiñán and Weiran Wang. Distributed optimization of deeply nested systems. *arXiv*, pages arXiv–1212, 2012.

Oscar Fontenla-Romero Enrique Castillo, Bertha Guijarro-Berdias and Amparo Alonso-Betanzos. A very fast learning method for neural networks based on sensitivity analysis. *Journal of Machine Learning Research (JMLR)*, 7:1159–1182, July 2006. URL `jmlr.csail.mit.edu/papers/volume7/castillo06a/castillo06a.pdf`.

M. Fairbank, D. Prokhorov, and E. Alonso. Clipping in neurocontrol by adaptive dynamic programming. *IEEE Transactions on Neural Networks and Learning Systems*, 25(10): 1909–1920, Oct 2014a. doi: 10.1109/TNNLS.2014.2297991.

Michael Fairbank, Shuhui Li, Xingang Fu, Eduardo Alonso, and Donald Wunsch. An adaptive recurrent neural-network controller using a stabilization matrix and predictive inputs to solve a tracking problem under disturbances. *Neural Networks*, 49(0):74 – 86, 2014b. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/j.neunet.2013.09.010. URL http://www.sciencedirect.com/science/article/pii/S0893608013002438.

Thomas Frerix, Thomas Möllenhoff, Michael Moeller, and Daniel Cremers. Proximal backpropagation. *arXiv preprint arXiv:1706.04638*, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, fourth edition, 2013.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997a.

Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997b.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2:18, 2010.

Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. In *Artificial intelligence and statistics*, pages 562–570, 2015a.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Joint european conference on machine learning and knowledge discovery in databases*, pages 498–515. Springer, 2015b.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Martin Fodslette Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533, 1993.

Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook, nov 2012. *URL http://www2. imm. dtu. dk/pubdb/p. php*, 3274, 2012.

Manh Cong Phan and Martin T Hagan. Error surface of recurrent neural networks. *IEEE transactions on neural networks and learning systems*, 24(11):1709–1721, 2013.

L. B. Rall. Automatic differentiation: Techniques and applications. *Lecture Notes in Computer Science*, 120, 1981.

Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, 1993.

Richard Rohwer. The "moving targets" training algorithm. In *Advances in neural information processing systems*, pages 558–565, 1990.

David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Spyridon Samothrakis, Tom Vodopivec, Maria Fasli, and Michael Fairbank. Match memory recurrent networks. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 1339–1346. IEEE, 2016.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Ivan Skorokhodov and Mikhail Burtsev. Loss landscape sightseeing with multi-point optimization, 2019.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

I Sutskever, O Vinyals, and QV Le. Sequence to sequence learning with neural networks. *Advances in NIPS*, 2014.

Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. Training neural networks without gradients: A scalable admm approach. In *International conference on machine learning*, pages 2722–2731, 2016.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Technical Report*, 2012.

Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

Paul J. Werbos. Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, No. 10, pages 1550–1560, 1990.

Paul J. Werbos. Backwards differentiation in AD and neural nets: Past links and new opportunities. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 15–34. Springer, 2005.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

Ziming Zhang, Yuting Chen, and Venkatesh Saligrama. Efficient training of very deep neural networks for supervised hashing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1487–1495, 2016.