Decoupled Neural Interfaces using Synthetic Gradients

Max Jaderberg [1] **Wojciech Marian Czarnecki** [1] **Simon Osindero** [1] **Oriol Vinyals** [1] **Alex Graves** [1] **David Silver** [1]
Koray Kavukcuoglu [1]

Abstract

Training directed neural networks typically requires forward-propagating data through a computation graph, followed by backpropagating error signal, to produce weight updates. All layers, or more generally, modules, of the network are therefore locked, in the sense that they must wait for the remainder of the network to execute forwards and propagate error backwards before they can be updated. In this work we break this constraint by decoupling modules by introducing a model of the future computation of the network graph. These models predict what the result of the modelled subgraph will produce using only local information. In particular we focus on modelling error gradients: by using the modelled *synthetic gradient* in place of true backpropagated error gradients we decouple subgraphs, and can update them independently and asynchronously *i.e.* we realise *decoupled neural interfaces*. We show results for feed-forward models, where every layer is trained asynchronously, recurrent neural networks (RNNs) where predicting one's future gradient extends the time over which the RNN can effectively model, and also a hierarchical RNN system with ticking at different timescales. Finally, we demonstrate that in addition to predicting gradients, the same framework can be used to predict inputs, resulting in models which are decoupled in both the forward and backwards pass – amounting to independent networks which co-learn such that they can be composed into a single functioning corporation.

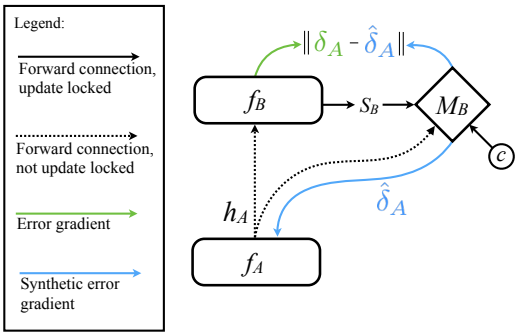


Figure 1. General communication protocol between A and B. After receiving the message h_A from A, B can use its model of A, M_B, to send back *synthetic gradients* $\hat{\delta}_A$ which are trained to approximate real error gradients δ_A. Note that A does not need to wait for any extra computation after itself to get the correct error gradients, hence decoupling the backward computation. The feedback model M_B can also be conditioned on any privileged information or context, c, available during training such as a label.

1. Introduction

Each layer (or module) in a directed neural network can be considered a computation step, that transforms its incoming data. These modules are connected via directed edges, creating a forward processing graph which defines the flow of data from the network inputs, through each module, producing network outputs. Defining a loss on outputs allows errors to be generated, and propagated back through the network graph to provide a signal to update each module.

This process results in several forms of *locking*, namely: (i) *Forward Locking* – no module can process its incoming data before the previous nodes in the directed forward graph have executed; (ii) *Update Locking* – no module can be updated before all dependent modules have executed in forwards mode; also, in many credit-assignment algorithms (including backpropagation (Rumelhart et al., 1986)) we have (iii) *Backwards Locking* – no module can be updated before all dependent modules have executed in both forwards mode and backwards mode.

Forwards, update, and backwards locking constrain us to running and updating neural networks in a sequential, synchronous manner. Though seemingly benign when training

[1]DeepMind, London, UK. Correspondence to: Max Jaderberg <jaderberg@google.com>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

simple feed-forward nets, this poses problems when thinking about creating systems of networks acting in multiple environments at different and possibly irregular or asynchronous timescales. For example, in complex systems comprised of multiple asynchronous cooperative modules (or agents), it is undesirable and potentially unfeasible that all networks are update locked. Another example is a distributed model, where part of the model is shared and used by many downstream clients – all clients must be fully executed and pass error gradients back to the shared model before the model can update, meaning the system trains as fast as the slowest client. The possibility to parallelise training of currently sequential systems could hugely speed up computation time.

The goal of this work is to remove update locking for neural networks. This is achieved by removing backpropagation. To update weights θ_i of module i we drastically approximate the function implied by backpropagation:

$$\begin{aligned}\frac{\partial L}{\partial \theta_i} &= f_{\text{Bprop}}((h_i, x_i, y_i, \theta_i), \ldots)\frac{\partial h_i}{\partial \theta_i} \\ &\simeq \hat{f}_{\text{Bprop}}(h_i)\frac{\partial h_i}{\partial \theta_i}\end{aligned}$$

where h are activations, x are inputs, y is supervision, and L is the overall loss to minimise. This leaves dependency only on h_i – the information local to module i.

The premise of this method is based on a simple protocol for learnt communication, allowing neural network modules to interact and be trained without update locking. While the communication protocol is general with respect to the means of generating a training signal, here we focus on a specific implementation for networks trained with gradient descent – we replace a standard *neural interface* (a connection between two modules in a neural network) with a Decoupled Neural Interface (DNI). Most simply, when a module (*e.g.* a layer) sends a message (activations) to another module, there is an associated model which produces a predicted error gradient with respect to the message immediately. The predicted gradient is a function of the message alone; there is no dependence on downstream events, states or losses. The sender can then immediately use these *synthetic gradients* to get an update, without incurring any delay. And by removing update- and backwards locking in this way, we can train networks without a synchronous backward pass. We also show preliminary results that extend this idea to also remove forward locking – resulting in networks whose modules can also be trained without a synchronous forward pass. When applied to RNNs we show that using synthetic gradients allows RNNs to model much greater time horizons than the limit imposed by truncating backpropagation through time (BPTT). We also show that using synthetic gradients to decouple a system of two RNNs running at different timescales can greatly increase training speed of the faster RNN.

Our synthetic gradient model is most analogous to a value function which is used for gradient ascent (Baxter & Bartlett, 2000) or critics for training neural networks (Schmidhuber, 1990). Most other works that aim to remove backpropagation do so with the goal of performing biologically plausible credit assignment, but this doesn't eliminate update locking between layers. *E.g.* target propagation (Lee et al., 2015; Bengio, 2014) removes the reliance on passing gradients between layers, by instead generating target activations which should be fitted to. However these targets must still be generated sequentially, propagating backwards through the network and layers are therefore still update- and backwards-locked. Other algorithms remove the backwards locking by allowing loss or rewards to be broadcast directly to each layer – *e.g.* REINFORCE (Williams, 1992) (considering all activations are actions), Kickback (Balduzzi et al., 2014), and Policy Gradient Coagent Networks (Thomas, 2011) – but still remain update locked since they require rewards to be generated by an output (or a global critic). While Real-Time Recurrent Learning (Williams & Zipser, 1989) or approximations such as (Ollivier & Charpiat, 2015; Tallec & Ollivier, 2017) may seem a promising way to remove update locking, these methods require maintaining the full (or approximate) gradient of the current state with respect to the parameters. This is inherently not scalable and also requires the optimiser to have global knowledge of the network state. In contrast, by framing the interaction between layers as a local communication problem with DNI, we remove the need for global knowledge of the learning system. Other works such as (Taylor et al., 2016; Carreira-Perpinán & Wang, 2014) allow training of layers in parallel without backpropagation, but in practice are not scalable to more complex and generic network architectures.

2. Decoupled Neural Interfaces

We begin by describing the high-level communication protocol that is used to allow asynchronously learning agents to communicate.

As shown in Fig. 1, Sender A sends a message h_A to Receiver B. B has a model M_B of the utility of the message h_A. B's model of utility M_B is used to predict the feedback: an error signal $\hat{\delta}_A = M_B(h_A, s_B, c)$ based on the message h_A, the current state of B, s_B, and potentially any other information, c, that this module is privy to during training such as the label or context. The feedback $\hat{\delta}_A$ is sent back to A which allows A to be updated immediately. In time, B can fully evaluate the true utility δ_A of the message received from A, and so B's utility model can be updated to fit the true utility, reducing the disparity between $\hat{\delta}_A$ and δ_A.

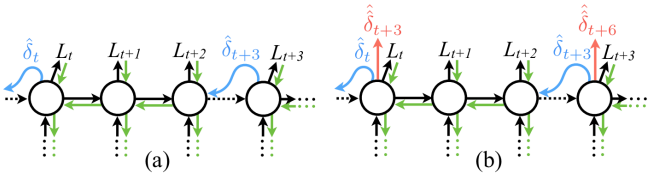


Figure 2. (a) An RNN trained with truncated BPTT using DNI to communicate over time: Every timestep a recurrent core takes input and produces a hidden state h_t and output y_t which affects a loss L_t. The core is unrolled for T steps (in this figure $T = 3$). Gradients cannot propagate across the boundaries of BPTT, which limits the time dependency the RNN can learn to model. However, the recurrent core includes a synthetic gradient model which produces synthetic gradients $\hat{\delta}_t$ which can be used at the boundaries of BPTT to enable the last set of unrolled cores to communicate with the future ones. (b) In addition, as an auxiliary task, the network can also be asked to do future synthetic gradient prediction: an extra output $\hat{\hat{\delta}}_{t+T}$ is computed every timestep, and is trained to minimise $\|\hat{\hat{\delta}}_{t+T} - \hat{\delta}_{t+T}\|$.

This protocol allows A to send messages to B in a way that A and B are *update decoupled* – A does not have to wait for B to evaluate the true utility before it can be updated – and A can still learn to send messages of high utility to B.

We can apply this protocol to neural networks communicating, resulting in what we call Decoupled Neural Interfaces (DNI). For neural networks, the feedback error signal $\hat{\delta}_A$ can take different forms, *e.g.* gradients can be used as the error signal to work with backpropagation, target messages as the error signal to work with target propagation, or even a value (cumulative discounted future reward) to incorporate into a reinforcement learning framework. However, as a clear and easily analysable set of first steps into this important and mostly unexplored domain, we concentrate our empirical study on differentiable networks trained with backpropagation and gradient-based updates. Therefore, we focus on producing error gradients as the feedback $\hat{\delta}_A$ which we dub *synthetic gradients*.

Notation To facilitate our exposition, it's useful to introduce some notation. Without loss of generality, consider neural networks as a graph of function operations (a finite chain graph in the case of a feed-forward models, an infinite chain in the case of recurrent ones, and more generally a directed acyclic graph). The forward execution of the network graph has a natural ordering due to the input dependencies of each functional node. We denote the function corresponding to step i in a graph execution as f_i and the composition of functions (*i.e.* the forward graph) from step i to step j inclusive as $\mathcal{F}_i^j$. We denote the loss associated with layer, i, of the chain as L_i.

2.1. Synthetic Gradient for Recurrent Networks

We begin by describing how our method of using synthetic gradients applies in the case of recurrent networks; in some ways this is simpler to reason about than feed-forward networks or more general graphs.

An RNN applied to infinite stream prediction can be viewed as an infinitely unrolled recurrent core module f with parameters θ, such that the forward graph is $\mathcal{F}_1^\infty = (f_i)_{i=1}^\infty$ where $f_i = f \ \forall i$ and the core module propagates an output y_i and state h_i based on some input x_i: $y_i, h_i = f_i(x_i, h_{i-1})$.

At a particular point in time t we wish to minimise $\sum_{\tau=t}^\infty L_\tau$. Of course, one cannot compute an update of the form $\theta \leftarrow \theta - \alpha \sum_{\tau=t}^\infty \frac{\partial L_\tau}{\partial \theta}$ due to the infinite future time dependency. Instead, generally one considers a tractable time horizon T

$$\theta - \alpha \sum_{\tau=t}^{\infty} \frac{\partial L_\tau}{\partial \theta} = \theta - \alpha \Big(\sum_{\tau=t}^{t+T} \frac{\partial L_\tau}{\partial \theta} + \Big(\sum_{\tau=T+1}^{\infty} \frac{\partial L_\tau}{\partial h_T}\Big) \frac{\partial h_T}{\partial \theta}\Big)$$
$$= \theta - \alpha \Big(\sum_{\tau=t}^{t+T} \frac{\partial L_\tau}{\partial \theta} + \delta_T \frac{\partial h_T}{\partial \theta}\Big)$$

and as in truncated BPTT, calculates $\sum_{\tau=t}^{t+T} \frac{\partial L_\tau}{\partial \theta}$ with backpropagation and approximates the remaining terms, beyond $t+T$, by using $\delta_T = 0$. This limits the time horizon over which updates to θ can be learnt, effectively limiting the amount of temporal dependency an RNN can learn. The approximation that $\delta_T = 0$ is clearly naive, and by using an appropriately *learned* approximation we can hope to do better. Treating the connection between recurrent cores at time $t+T$ as a Decoupled Neural Interface we can approximate δ_T, with $\hat{\delta}_T = M_T(h_T)$ – a learned approximation of the future loss gradients – as shown and described in Fig. 2 (a).

This amounts to taking the infinitely unrolled RNN as the full neural network $\mathcal{F}_1^\infty$, and chunking it into an infinite number of sub-networks where the recurrent core is unrolled for T steps, giving $\mathcal{F}_t^{t+T-1}$. Inserting DNI between two adjacent sub-networks $\mathcal{F}_t^{t+T-1}$ and $\mathcal{F}_{t+T}^{t+2T-1}$ allows the recurrent network to learn to communicate to its future self, without being update locked to its future self. From the view of the synthetic gradient model, the RNN is predicting its own error gradients.

The synthetic gradient model $\hat{\delta}_T = M_T(h_T)$ is trained to predict the true gradients by minimising a distance $d(\hat{\delta}_T, \delta_T)$ to the target gradient δ_T – in practice we find L_2 distance to work well. The target gradient is ideally the true gradient of future loss, $\sum_{\tau=T+1}^\infty \frac{\partial L_\tau}{\partial h_T}$, but as this is not a tractable target to obtain, we can use a target gradient that is itself bootstrapped from a synthetic gradient and then backpropagated and mixed with a number of steps of true

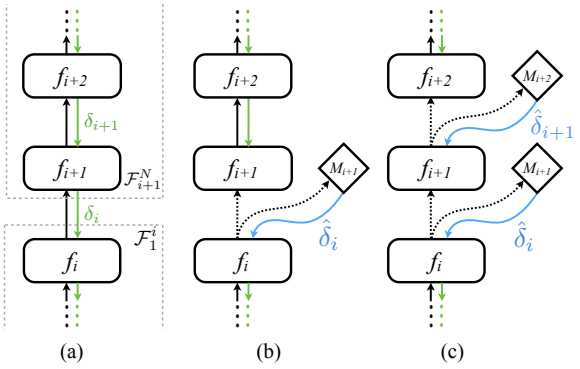


Figure 3. (a) A section of a vanilla feed-forward neural network $\mathcal{F}_1^N$. (b) Incorporating one synthetic gradient model for the output of layer i. This results in two sub-networks $\mathcal{F}_1^i$ and $\mathcal{F}_{i+1}^N$ which can be updated independently. (c) Incorporating multiple synthetic gradient models after every layer results in N independently updated layers.

gradient, *e.g.* $\delta_T = \sum_{\tau=T+1}^{2T} \frac{\partial L_\tau}{\partial h_T} + \hat{\delta}_{2T+1} \frac{\partial h_{2T}}{\partial h_T}$. This bootstrapping is exactly analogous to bootstrapping value functions in reinforcement learning and allows temporal credit assignment to propagate beyond the boundary of truncated BPTT.

This training scheme can be implemented very efficiently by exploiting the recurrent nature of the network, as shown in Fig. 10 in the Supplementary Material. In Sect. 3.1 we show results on sequence-to-sequence tasks and language modelling, where using synthetic gradients extends the time dependency the RNN can learn.

Auxiliary Tasks We also propose an extension to aid learning of synthetic gradient models for RNNs, which is to introduce another auxiliary task from the RNN, described in Fig. 2 (b). This extra prediction problem is designed to promote coupling over the maximum time span possible, requiring the recurrent core to explicitly model short term and long term synthetic gradients, helping propagate gradient information backwards in time. This is also shown to further increase performance in Sect. 3.1.

2.2. Synthetic Gradient for Feed-Forward Networks

As another illustration of DNIs, we now consider feed-forward networks consisting of N layers $f_i, i \in \{1, \ldots, N\}$, each taking an input h_{i-1} and producing an output $h_i = f_i(h_{i-1})$, where $h_0 = x$ is the input data. The forward execution graph of the full network can be denoted as as $\mathcal{F}_1^N$, a section of which is illustrated in Fig. 3 (a).

Define the loss imposed on the output of the network as $L = L_N$. Each layer f_i has parameters θ_i that can be trained jointly to minimise $L(h_N)$ with a gradient-based update rule

$$\theta_i \leftarrow \theta_i - \alpha\, \delta_i \frac{\partial h_i}{\partial \theta_i}\ ;\ \ \delta_i = \frac{\partial L}{\partial h_i}$$

where α is the learning rate and $\frac{\partial L}{\partial h_i}$ is computed with backpropagation. The reliance on δ_i means that the update to layer i can only occur after the remainder of the network, *i.e.* $\mathcal{F}_{i+1}^N$ (the sub-network of layers between layer $i+1$ and layer N inclusive) has executed a full forward pass, generated the loss $L(h_N)$, then backpropagated the gradient through every successor layer in reverse order. Layer i is therefore update locked to $\mathcal{F}_{i+1}^N$.

To remove the update locking of layer i to $\mathcal{F}_{i+1}^N$ we can use the communication protocol described previously. Layer i sends h_i to layer $i+1$, which has a communication model M_{i+1} that produces a synthetic error gradient $\hat{\delta}_i = M_{i+1}(h_i)$, as shown in Fig. 3 (b), which can be used immediately to update layer i and all the other layers in $\mathcal{F}_1^i$

$$\theta_n \leftarrow \theta_n - \alpha\, \hat{\delta}_i \frac{\partial h_i}{\partial \theta_n},\ \ n \in \{1, \ldots, i\}.$$

To train the parameters of the synthetic gradient model M_{i+1}, we simply wait for the true error gradient δ_i to be computed (after a full forwards and backwards execution of $\mathcal{F}_{i+1}^N$), and fit the synthetic gradient to the true gradients by minimising $\|\hat{\delta}_i - \delta_i\|_2^2$.

Furthermore, for a feed-forward network, we can use synthetic gradients as communication feedback to decouple every layer in the network, as shown in Fig. 3 (c). The execution of this process is illustrated in Fig. 9 in the Supplementary Material. In this case, since the target error gradient δ_i is produced by backpropagating $\hat{\delta}_{i+1}$ through layer $i+1$, δ_i is not the true error gradient, but an estimate bootstrapped from synthetic gradient models later in the network. Surprisingly, this does not cause errors to compound and learning remains stable even with many layers, as shown in Sect. 3.3.

Additionally, if any supervision or context c is available at the time of synthetic gradient computation, the synthetic gradient model can take this as an extra input, $\hat{\delta}_i = M_{i+1}(h_i, c)$.

This process allows a layer to be updated as soon as a forward pass of that layer has been executed. This paves the way for sub-parts or layers of networks to be trained in an asynchronous manner, something we show in Sect. 3.3.

2.3. Arbitrary Network Graphs

Although we have explicitly described the application of DNIs for communication between layers in feed-forward networks, and between recurrent cores in recurrent networks, there is nothing to restrict the use of DNIs for arbitrary network graphs. The same procedure can be applied

to any network or collection of networks, any number of times. An example is in Sect. 3.2 where we show communication between two RNNs, which tick at different rates, where the communication can be learnt by using synthetic gradients.

2.4. Mixing Real & Synthetic Gradients

In this paper we focus on the use of synthetic gradients to replace real backpropagated gradients in order to achieve update unlocking. However, synthetic gradients could also be used to augment real gradients. Mixing real and synthetic gradients results in $BP(\lambda)$, an algorithm anolgous to $TD(\lambda)$ for reinforcement learning (Sutton & Barto, 1998). This can be seen as a generalized view of synthetic gradients, with the algorithms given in this section for update unlocked RNNs and feed-forward networks being specific instantiations of $BP(\lambda)$. This generalised view is discussed further in Sect. A in the Supplementary Material.

3. Experiments

In this section we perform empirical expositions of the use of DNIs and synthetic gradients, first by applying them to RNNs in Sect. 3.1 showing that synthetic gradients extend the temporal correlations an RNN can learn. Secondly, in Sect. 3.2 we show how a hierarchical, two-timescale system of networks can be jointly trained using synthetic gradients to propagate error signals between networks. Finally, we demonstrate the ability of DNIs to allow asynchronous updating of layers a feed-forward network in Sect. 3.3. More experiments can be found in Sect. C in the Supplementary Material.

3.1. Recurrent Neural Networks

Here we show the application of DNIs to recurrent neural networks as discussed in Sect. 2.1. We test our models on the Copy task, Repeat Copy task, as well as character-level language modelling.

For all experiments we use an LSTM (Hochreiter & Schmidhuber, 1997) of the form in (Graves, 2013), whose output is used for the task at hand, and additionally as input to the synthetic gradient model (which is shared over all timesteps). The LSTM is unrolled for T timesteps after which backpropagation through time (BPTT) is performed. We also look at incorporating an auxiliary task which predicts the output of the synthetic gradient model T steps in the future as explained in Sect. 2.1. The implementation details of the RNN models are given in Sect. D.2 in the Supplementary Material.

Copy and Repeat Copy We first look at two synthetic tasks – Copy and Repeat Copy tasks from (Graves et al., 2014). Copy involves reading in a sequence of N characters and after a stop character is encountered, must repeat the sequence of N characters in order and produce a final stop character. Repeat Copy must also read a sequence of N characters, but after the stop character, reads the number, R, which indicates the number of times it is required to copy the sequence, before outputting a final stop character. Each sequence of reading and copying is an episode, of length $T_{\text{task}} = N + 3$ for Copy and $T_{\text{task}} = NR + 3$ for Repeat Copy.

While normally the RNN would be unrolled for the length of the episode before BPTT is performed, $T = T_{\text{task}}$, we wish to test the length of time the RNN is able to model with and without DNI bridging the BPTT limit. We therefore train the RNN with truncated BPTT: $T \in \{2, 3, 4, 5\}$ with and without DNI, where the RNN is applied continuously and across episode boundaries. For each problem, once the RNN has solved a task with a particular episode length (averaging below 0.15 bits error), the task is made harder by extending N for Copy and Repeat Copy, and also R for Repeat Copy.

Table 1 gives the results by reporting the largest T_{task} that is successfully solved by the model. The RNNs without DNI generally perform as expected, with longer BPTT resulting in being able to model longer time dependencies. However, by introducing DNI we can extend the time dependency that is able to be modelled by an RNN. The additional computational complexity is negligible but we require an additional recurrent core to be stored in memory (this is illustrated in Fig. 10 in the Supplementary Material). Because we can model larger time dependencies with a smaller T, our models become more data-efficient, learning faster and having to see less data samples to solve a task. Furthermore, when we include the extra task of predicting the synthetic gradient that will be produced T steps in the future (DNI + Aux), the RNNs with DNI are able to model even larger time dependencies. For example with $T = 3$ (*i.e.* performing BPTT across only three timesteps) on the Repeat Copy task, the DNI enabled RNN goes from being able to model 33 timesteps to 59 timesteps when using future synthetic gradient prediction as well. This is in contrast to without using DNI at all, where the RNN can only model 5 timesteps.

Language Modelling We also applied our DNI-enabled RNNs to the task of character-level language modelling, using the Penn Treebank dataset (Marcus et al., 1993). We use an LSTM with 1024 units, which at every timestep reads a character and must predict the next character in the sequence. We train with BPTT with and without DNI, as well as when using future synthetic gradient prediction (DNI + Aux), with $T \in \{2, 3, 4, 5, 8\}$ as well as strong baselines with $T = 20, 40$. We measure error in bits per

	BPTT							DNI					DNI + Aux				
$T =$	2	3	4	5	8	20	40	2	3	4	5	8	2	3	4	5	8
Copy	7	8	10	8	-	-	-	16	14	18	18	-	16	17	19	18	-
Repeat Copy	7	5	19	23	-	-	-	39	33	39	59	-	39	59	67	59	-
Penn Treebank	1.39	1.38	1.37	1.37	1.35	1.35	1.34	1.37	1.36	1.35	1.35	1.34	1.37	1.36	1.35	1.35	1.33

Table 1. Results for applying DNI to RNNs. Copy and Repeat Copy task performance is reported as the maximum sequence length that was successfully modelled (higher is better), and Penn Treebank results are reported in terms of test set bits per character (lower is better) at the point of lowest validation error. No learning rate decreases were performed during training.

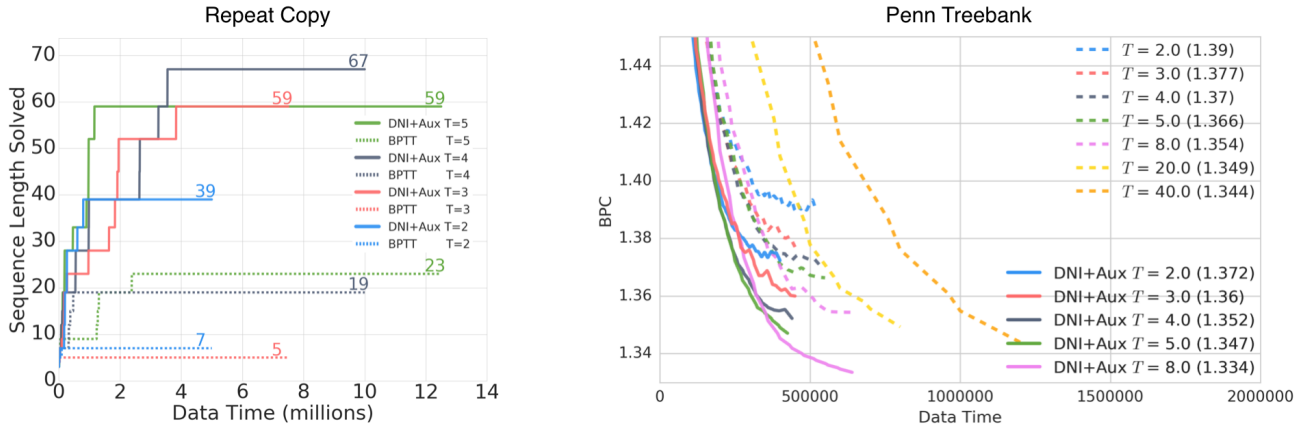


Figure 4. Left: The task progression during training for the Repeat Copy task. All models were trained for 2.5M iterations, but the varying unroll length T results in different quantities of data consumed. The x-axis shows the number of samples consumed by the model, and the y-axis the time dependency level solved by the model – step changes in the time dependency indicate that a particular time dependency is deemed solved. DNI+Aux refers to DNI with the additional future synthetic gradient prediction auxiliary task. *Right:* Test error in bits per character (BPC) for Penn Treebank character modelling. We train the RNNs with different BPTT unroll lengths with DNI (solid lines) and without DNI (dashed lines). Early stopping is performed based on the validation set. Bracketed numbers give final test set BPC.

character (BPC) as in (Graves, 2013), perform early stopping based on validation set error, and for simplicity do not perform any learning rate decay. For full experimental details please refer to Sect. D.2 in the Supplementary Material.

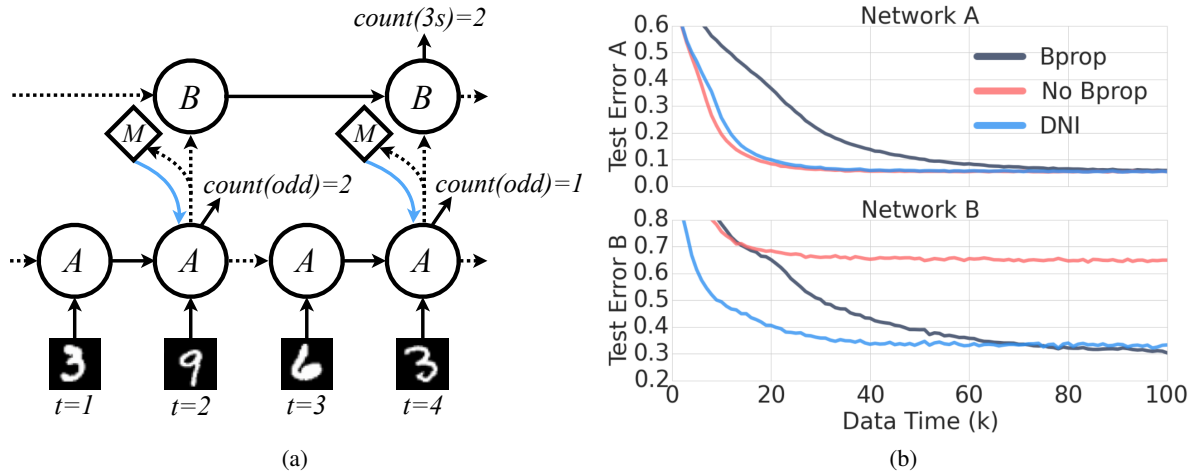
The results are given in Table 1. Interestingly, with BPTT over only two timesteps ($T = 2$) an LSTM can get surprisingly good accuracy at next character prediction. As expected, increasing T results in increased accuracy of prediction. When adding DNI, we see an increase in speed of learning (learning curves can be found in Fig. 4 (Right) and Fig. 16 in the Supplementary Material), and models reaching greater accuracy (lower BPC) than their counterparts without DNI. As seen with the Copy and Repeat Copy task, future synthetic gradient prediction further increases the ability of the LSTM to model long range temporal dependencies – an LSTM unrolled 5 timesteps with DNI and future synthetic gradient prediction gives the same BPC as a vanilla LSTM unrolled 20 steps, only needs 58% of the data and is $2\times$ faster in wall clock time to reach 1.35BPC.

Although we report results only with LSTMs, we have found DNI to work similarly for vanilla RNNs and Leaky RNNs (Ollivier & Charpiat, 2015).

3.2. Multi-Network System

In this section, we explore the use of DNI for communication between arbitrary graphs of networks. As a simple proof-of-concept, we look at a system of two RNNs, Network A and Network B, where Network B is executed at a slower rate than Network A, and must use communication from Network A to complete its task. The experimental setup is illustrated and described in Fig. 5 (a). Full experimental details can be found in Sect. D.3 in the Supplementary Material.

First, we test this system trained end-to-end, with full backpropagation through all connections, which requires the joint Network A-Network B system to be unrolled for T^2 timesteps before a single weight update to both Network A and Network B, as the communication between Network A to Network B causes Network A to be update locked to Network B. We the train the same system but using synthetic gradients to create a learnable bridge between Net-



(a) (b)

Figure 5. (a) System of two RNNs communicating with DNI. Network A sees a datastream of MNIST digits and every T steps must output the number of odd digits seen. Network B runs every T steps, takes a message from Network A as input and must output the number of 3s seen over the last T^2 timesteps. Here is a depiction where $T = 2$. (b) The test error over the course of training Network A and Network B with $T = 4$. Grey shows when the two-network system is treated as a single graph and trained with backpropagation end-to-end, with an update every T^2 timesteps. The blue curves are trained where Network A and Network B are decoupled, with DNI (blue) and without DNI (red). When not decoupled (grey), Network A can only be updated every T^2 steps as it is update locked to Network B, so trains slower than if the networks are decoupled (blue and red). Without using DNI (red), Network A receives no feedback from Network B as to how to process the data stream and send a message, so Network B performs poorly. Using synthetic gradient feedback allows Network A to learn to communicate with Network B, resulting in similar final performance to the end-to-end learnt system (results remain stable after 100k steps).

work A and Network B, thus decoupling Network A from Network B. This allows Network A to be updated T times more frequently, by using synthetic gradients in place of true gradients from Network B.

Fig. 5 (b) shows the results for $T = 4$. Looking at the test error during learning of Network A (Fig. 5 (b) Top), it is clear that being decoupled and therefore updated more frequently allows Network A to learn much quicker than when being locked to Network B, reaching final performance in under half the number of steps. Network B also trains faster with DNI (most likely due to the increased speed in learning of Network A), and reaches a similar final accuracy as with full backpropagation (Fig. 5 (b) Bottom). When the networks are decoupled but DNI is not used (*i.e.* no gradient is received by Network A from Network B), Network A receives no feedback from Network B, so cannot shape its representations and send a suitable message, meaning Network B cannot solve the problem.

3.3. Feed-Forward Networks

In this section we apply DNIs to feed-forward networks in order to allow asynchronous or sporadic training of layers, as might be required in a distributed training setup. As explained in Sect. 2.2, making layers decoupled by introducing synthetic gradients allows the layers to communicate with each other without being update locked.

Asynchronous Updates To demonstrate the gains by decoupling layers given by DNI, we perform an experiment on a four layer FCN model on MNIST, where the backwards pass and update for every layer occurs in random order and only with some probability p_{update} (*i.e.* a layer is only updated after its forward pass p_{update} of the time). This completely breaks backpropagation, as for example the first layer would only receive error gradients with probability p^3_{update} and even then, the system would be constrained to be synchronous. However, with DNI bridging the communication gap between each layer, the stochasticity of a layer's update does not mean the layer below cannot update, as it uses synthetic gradients rather than backpropagated gradients. We ran 100 experiments with different values of p_{update} uniformly sampled between 0 and 1. The results are shown in Fig. 7 (Left) for DNI with and without conditioning on the labels. With $p_{\text{update}} = 0.2$ the network can still train to 2% accuracy. Incredibly, when the DNI is conditioned on the labels of the data (a reasonable assumption if training in a distributed fashion), the network trains perfectly with only 5% chance of an update, albeit just slower.

Complete Unlock As a drastic extension, we look at making feed-forward networks completely asynchronous, by removing forward locking as well. In this scenario, every layer has a synthetic gradient model, but also a synthetic *input* model – given the data, the synthetic input

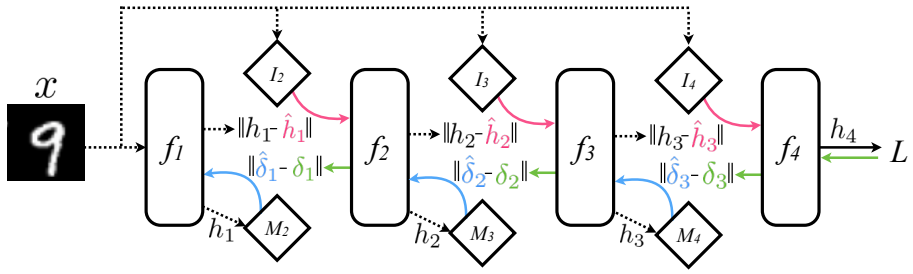


Figure 6. Completely unlocked feed-forward network training allowing forward and update decoupling of layers.

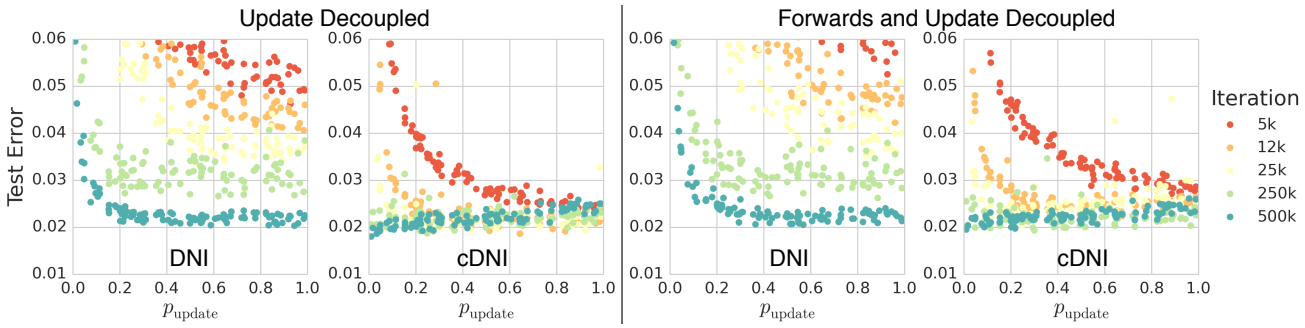


Figure 7. Left: Four layer FCNs trained on MNIST using DNI between every layer, however each layer is trained stochastically – after every forward pass, a layer only does a backwards pass with probability p_{update}. Population test errors are shown after different numbers of iterations (turquoise is at the end of training after 500k iterations). The purple diamond shows the result when performing regular backpropagation, requiring a synchronous backwards pass and therefore $p_{\text{update}} = 1$. When using cDNIs however, with only 5% probability of a layer being updated the network can train effectively. *Right:* The same setup as previously described however we also use a synthetic *input* model before every layer, which allows the network to also be *forwards decoupled*. Now every layer is trained completely asynchronously, where with probability $1 - p_{\text{update}}$ a layer does not do a forward pass or backwards pass – effectively the layer is "busy" and cannot be touched at all.

model produces an approximation of what the input to the layer will be. This is illustrated in Fig. 6. Every layer can now be trained independently, with the synthetic gradient and input models trained to regress targets produced by neighbouring layers. The results on MNIST are shown in Fig. 7 (Right), and at least in this simple scenario, the completely asynchronous collection of layers train independently, but co-learn to reach 2% accuracy, only slightly slower. More details are given in the Supplementary Material.

4. Discussion & Conclusion

In this work we introduced a method, *DNI using synthetic gradients*, which allows decoupled communication between components, such that they can be independently updated. We demonstrated significant gains from the increased time horizon that DNI-enabled RNNs are able to model, as well as faster convergence. We also demonstrated the application to a multi-network system: a communicating pair of fast- and slow-ticking RNNs can be decoupled, greatly accelarating learning. Finally, we showed that the method can be used facilitate distributed training by enabling us to completely decouple all the layers of a feed-forward net – thus allowing them to be trained asynchronously, non-sequentially, and sporadically.

It should be noted that while this paper introduces and shows empirical justification for the efficacy of DNIs and synthetic gradients, the work of Czarnecki et al. (2017) delves deeper into the analysis and theoretical understanding of DNIs and synthetic gradients, confirming the convergence properties of these methods and modelling impacts of using synthetic gradients.

To our knowledge this is the first time that neural net modules have been decoupled, and the update locking has been broken. This important result opens up exciting avenues of exploration – including improving the foundations laid out here, and application to modular, decoupled, and asynchronous model architectures.

References

Balduzzi, D., Vanchinathan, H., and Buhmann, J. Kickback cuts backprop's red-tape: Biologically plausible

credit assignment in neural networks. *arXiv preprint arXiv:1411.6191*, 2014.

Baxter, J. and Bartlett, P. L. Direct gradient-based reinforcement learning. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 3, pp. 271–274. IEEE, 2000.

Bengio, Y. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.

Carreira-Perpinán, M A and Wang, W. Distributed optimization of deeply nested systems. In *AISTATS*, pp. 10–19, 2014.

Czarnecki, W M, Swirszcz, G, Jaderberg, M, Osindero, S, Vinyals, O, and Kavukcuoglu, K. Understanding synthetic gradients and decoupled neural interfaces. *arXiv preprint*, 2017.

Graves, A. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

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

Lee, D., Zhang, S., Fischer, A., and Bengio, Y. Difference target propagation. In *Machine Learning and Knowledge Discovery in Databases*, pp. 498–515. Springer, 2015.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

Ollivier, Y. and Charpiat, G. Training recurrent networks online without backtracking. *arXiv preprint arXiv:1507.07680*, 2015.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Schmidhuber, Jürgen. Networks adjusting networks. In *Proceedings ofDistributed Adaptive Neural Information Processing', St. Augustin*. Citeseer, 1990.

Sutton, R S and Barto, A G. Reinforcement learning: An introduction, 1998.

Tallec, C. and Ollivier, Y. Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*, 2017.

Taylor, G, Burmeister, R, Xu, Z, Singh, B, Patel, A, and Goldstein, T. Training neural networks without gradients: A scalable admm approach. *ICML*, 2016.

Thomas, P. S. Policy gradient coagent networks. In *Advances in Neural Information Processing Systems*, pp. 1944–1952, 2011.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Williams, R. J. and Zipser, D. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.