Supplementary Material for Decoupled Neural Interfaces using Synthetic Gradients

A. Unified View of Synthetic Gradients

The main idea of this paper is to learn a *synthetic gradient*, *i.e.* a separate prediction of the loss gradient for every layer of the network. The synthetic gradient can be used as a drop-in replacement for the backpropagated gradient. This provides a choice of two gradients at each layer: the gradient of the true loss, backpropagated from subsequent layers; or the synthetic gradient, estimated from the activations of that layer.

In this section we present a unified algorithm, $BP(\lambda)$, that mixes these two gradient estimates as desired using a parameter λ. This allows the backpropagated gradient to be used insofar as it is available and trusted, but provides a meaningful alternative when it is not. This mixture of gradients is then backpropagated to the previous layer.

A.1. BP(0)

We begin by defining our general setup and consider the simplest instance of synthetic gradients, $BP(0)$. We consider a feed-forward network with activations h_k for $k \in \{1, \ldots, K\}$, and parameters θ_k corresponding to layers $k \in \{1, \ldots, K\}$. The goal is to optimize a loss function L that depends on the final activations h_K. The key idea is to approximate the gradient of the loss, $g_k \approx \frac{\partial L}{\partial h_k}$, using a *synthetic gradient*, g_k. The synthetic gradient is estimated from the activations at layer k, using a function $g_k = g(h_k, \phi_k)$ with parameters ϕ_k. The overall loss can then be minimized by stochastic gradient descent on the synthetic gradient,

$$\frac{\partial L}{\partial \theta_k} = \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial \theta_k} \approx g_k \frac{\partial h_k}{\partial \theta_k}.$$

In order for this approach to work, the synthetic gradient must also be trained to approximate the true loss gradient. Of course, it could be trained by regressing g_k towards $\frac{\partial L}{\partial h_k}$, but our underlying assumption is that the backpropagated gradients are not available. Instead, we "unroll" our synthetic gradient just one step,

$$g_k \approx \frac{\partial L}{\partial h_k} = \frac{\partial L}{\partial h_{k+1}}\frac{\partial h_{k+1}}{\partial h_k} \approx g_{k+1}\frac{\partial h_{k+1}}{\partial h_k},$$

and treat the unrolled synthetic gradient $z_k = g_{k+1}\frac{\partial h_{k+1}}{\partial h_k}$ as a constant training target for the synthetic gradient g_k. Specifically we update the synthetic gradient parameters ϕ_k so as to minimise the mean-squared error of these one-step unrolled training targets, by stochastic gradient descent on $\frac{\partial (z_k - g_k)^2}{\partial \phi_k}$. This idea is analogous to bootstrapping in the TD(0) algorithm for reinforcement learning (Sutton, 1988).

A.2. BP(λ)

In the previous section we removed backpropagation altogether. We now consider how to combine synthetic gradients with a controlled amount of backpropagation. The idea of BP(λ) is to mix together many different estimates of the loss gradient, each of which unrolls the chain rule for n steps and then applies the synthetic gradient,

$$\begin{aligned} g_k^n &= g_{k+n}\frac{\partial h_{k+n}}{\partial h_{k+n-1}} \ldots \frac{\partial h_{k+1}}{\partial h_k} \\ &\approx \frac{\partial L}{\partial h_{k+n}}\frac{\partial h_{k+n}}{\partial h_{k+n-1}} \ldots \frac{\partial h_{k+1}}{\partial h_k} \\ &= \frac{\partial L}{\partial h_k}. \end{aligned}$$

We mix these estimators together recursively using a weighting parameter λ_k (see Figure 1),

$$\bar{g}_k = \lambda_k \bar{g}_{k+1}\frac{\partial h_{k+1}}{\partial h_k} + (1-\lambda_k) g_k.$$

The resulting λ-weighted synthetic gradient $\bar{g}_k$ is a geometric mixture of the gradient estimates $g_k^1, \ldots, g_K^2$,

$$\bar{g}_k = \sum_{n=k}^{K} c_k^n g_k^n.$$

where $c_k^n = (1-\lambda_n)\prod_{j=k}^{n-1}\lambda_j$ is the weight of the nth gradient estimator g_k^n, and $c_k^K = 1 - \sum_{n=1}^{K-1} c_k^n$ is the weight for the final layer. This geometric mixture is analogous to the λ-return in $TD(\lambda)$ (Sutton, 1988).

To update the network parameters θ, we use the λ-weighted synthetic gradient estimate in place of the loss gradient,

$$\frac{\partial L}{\partial \theta_k} = \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial \theta_k} \approx \bar{g}_k \frac{\partial h_k}{\partial \theta_k}$$

To update the synthetic gradient parameters ϕ_k, we unroll the λ-weighted synthetic gradient by one step, $\bar{z}_k =$

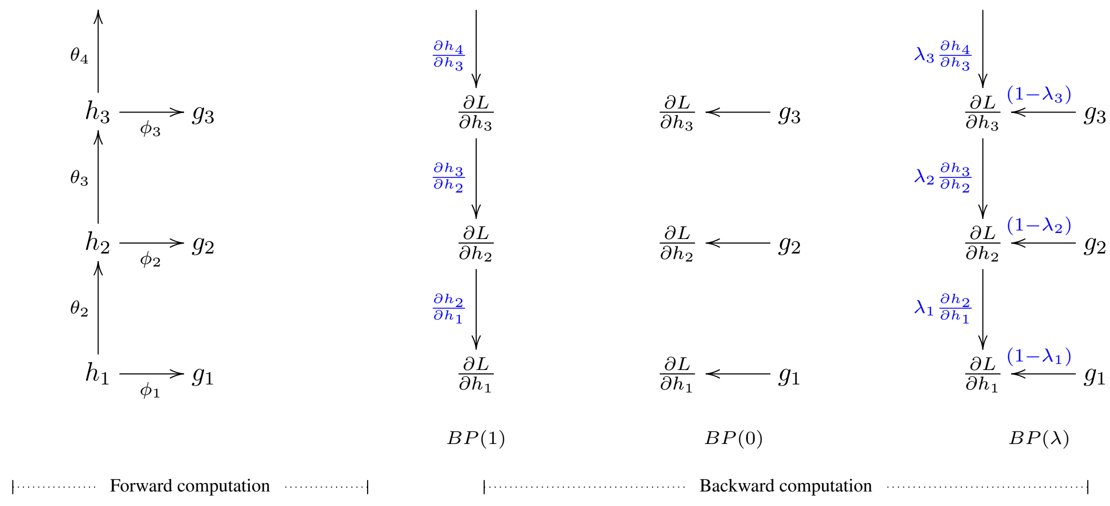


Figure 8. (Left) Forward computation of synthetic gradients. Arrows represent computations using parameters specified in label. (Right) Backward computation in BP(λ). Each arrow may post-multiply its input by the specified value in blue label. BP(1) is equivalent to error backpropagation.

$\bar{g}_{k+1}\frac{\partial h_{k+1}}{\partial h_k}$, and treat this as a constant training target for the synthetic gradient g_k. Parameters are adjusted by stochastic gradient descent to minimise the mean-squared error between the synthetic gradient and its unrolled target, $\frac{\partial(\bar{z}_k - g_k)^2}{\partial \phi_k}$.

The two extreme cases of $BP(\lambda)$ result in simpler algorithms. If $\lambda_k = 0 \; \forall k$ we recover the $BP(0)$ algorithm from the previous section, which performs no backpropagation whatsoever. If $\lambda_k = 1 \; \forall k$ then the synthetic gradients are ignored altogether and we recover error backpropagation. For the experiments in this paper we have used binary values $\lambda_k \in \{0, 1\}$.

A.3. Recurrent $BP(\lambda)$

We now discuss how $BP(\lambda)$ may be applied to RNNs. We apply the same basic idea as before, using a synthetic gradient as a proxy for the gradient of the loss. However, network parameters θ and synthetic gradient parameters ϕ are now shared across all steps. There may also be a separate loss l_k at every step k. The overall loss function is the sum of the step losses, $L = \sum_{k=1}^{\infty} l_k$.

The synthetic gradient g_k now estimates the cumulative loss from step $k+1$ onwards, $g_k \approx \frac{\partial \sum_{j=k+1}^{\infty} l_j}{\partial h_k}$. The λ-weighted synthetic gradient recursively combines these future estimates, and adds the immediate loss to provide an overall estimate of cumulative loss from step k onwards,

$$\bar{g}_k = \frac{\partial l_k}{\partial h_k} + \lambda_k \bar{g}_{k+1} \frac{\partial h_{k+1}}{\partial h_k} + (1 - \lambda_k) g_k.$$

Network parameters are adjusted by gradient descent on the cumulative loss,

$$\frac{\partial L}{\partial \theta} = \sum_{k=1}^{\infty} \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial \theta} = \sum_{k=1}^{\infty} \frac{\partial \sum_{j=k}^{\infty} l_j}{\partial h_k}\frac{\partial h_k}{\partial \theta} \approx \sum_{k=1}^{\infty} \bar{g}_k \frac{\partial h_k}{\partial \theta}.$$

To update the synthetic gradient parameters ϕ, we again unroll the λ-weighted synthetic gradient by one step, $\bar{z}_k = \frac{\partial l_k}{\partial h_k} + \bar{g}_{k+1}\frac{\partial h_{k+1}}{\partial h_k}$, and minimise the MSE with respect to this target, over all time-steps, $\sum_{k=1}^{\infty} \frac{\partial(\bar{z}_k - g_k)^2}{\partial \phi}$.

We note that for the special case $BP(0)$, there is no backpropagation at all and therefore weights may be updated in a fully online manner. This is possible because the synthetic gradient estimates the gradient of cumulative future loss, rather than explicitly backpropagating the loss from the end of the sequence.

Backpropagation-through-time requires computation from all time-steps to be retained in memory. As a result, RNNs are typically optimised in N-step chunks $[mN, (m+1)N]$. For each chunk m, the cumulative loss is initialised to zero at the final step $k = (m+1)N$, and then errors are backpropagated-through-time back to the initial step $k = mN$. However, this prevents the RNN from modelling longer term interactions. Instead, we can initialise the backpropagation at final step $k = (m+1)N$ with a synthetic gradient g_k that estimates long-term future loss, and then backpropagate the synthetic gradient through the chunk. This algorithm is a special case of $BP(\lambda)$ where $\lambda_k = 0$ if $k \mod N = 0$ and $\lambda_k = 1$ otherwise. The experiments in Sect. 3.1 illustrate this case.

A.4. Scalar and Vector Critics

One way to estimate the synthetic gradient is to first estimate the loss using a *critic*, $v(h_k, \phi) \approx \mathbb{E}[L|h_k]$, and then use the gradient of the critic as the synthetic gradient, $g_k = \frac{\partial v(h_k,\phi)}{\partial h_k} \approx \frac{\partial L}{\partial h_k}$. This provides a choice between a scalar approximation of the loss, or a vector approximation of the loss gradient, similar to the scalar and vector critics suggested by Fairbank (Fairbank, 2014).

These approaches have previously been used in control (Werbos, 1992; Fairbank, 2014) and model-based reinforcement learning (Heess et al., 2015). In these cases the dependence of total cost or reward on the policy parameters is computed by backpropagating through the trajectory. This may be viewed as a special case of the $BP(\lambda)$ algorithm; intermediate values of $\lambda < 1$ were most successful in noisy environments (Heess et al., 2015).

It is also possible to use other styles of critics or error approximation techniques such as Feedback Alignment (Lillicrap et al., 2016), Direct Feedback Alignment (Nøkland, 2016), and Kickback (Balduzzi et al., 2014)) – interestingly Czarnecki et al. (2017) shows that they can all be framed in the synthetic gradients framework presented in this paper.

B. Synthetic Gradients are Sufficient

In this section, we show that a function $f(h_t, \theta_{t+1:T})$, which depends only on the hidden activations h_t and downstream parameters $\theta_{t+1:T}$, is sufficient to represent the gradient of a feedforward or recurrent network, without any other dependence on past or future inputs $x_{1:T}$ or targets $y_{1:T}$.

In (stochastic) gradient descent, parameters are updated according to (samples of) the expected loss gradient,

$$\begin{aligned}
\mathbb{E}_{x_{1:T},y_{1:T}}\left[\frac{\partial L}{\partial \theta_t}\right] &= \mathbb{E}_{x_{1:T},y_{1:T}}\left[\frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial \theta_t}\right] \\
&= \mathbb{E}_{x_{1:T},y_{1:T}}\left[\mathbb{E}_{x_{t+1:T},y_{t:T}|x_{1:t},y_{1:t-1}}\left[\frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial \theta_t}\right]\right] \\
&= \mathbb{E}_{x_{1:T},y_{1:T}}\left[\mathbb{E}_{x_{t+1:T},y_{t:T}|h_t}\left[\frac{\partial L}{\partial h_t}\right]\frac{\partial h_t}{\partial \theta_t}\right] \\
&= \mathbb{E}_{x_{1:T},y_{1:T}}\left[g(h_t, \theta_{t+1:T})\frac{\partial h_t}{\partial \theta_t}\right]
\end{aligned}$$

where $g(h_t, \theta_{t+1:T}) = \mathbb{E}_{x_{t+1:T},y_{t:T}|h_t}\left[\frac{\partial L}{\partial h_t}\right]$ is the expected loss gradient given hidden activations h_t. Parameters may be updated using samples of this gradient, $g(h_t, \theta_{t+1:T})\frac{\partial h_t}{\partial \theta_t}$.

The synthetic gradient $g(h_t, v_t) \approx g(h_t, \theta_{t+1:T})$ approximates this expected loss gradient at the current parameters $\theta_{t+1:T}$. If these parameters are frozen, then a sufficiently powerful synthetic gradient approximator can learn to perfectly represent the expected loss gradient. This is similar to an actor-critic architecture, where the neural network is

	Layers	MNIST (% Error)				CIFAR-10 (% Error)			
		No Bprop	Bprop	DNI	cDNI	No Bprop	Bprop	DNI	cDNI
FCN	3	9.3	2.0	1.9	2.2	54.9	43.5	42.5	48.5
	4	12.6	1.8	2.2	1.9	57.2	43.0	45.0	45.1
	5	16.2	1.8	3.4	1.7	59.6	41.7	46.9	43.5
	6	21.4	1.8	4.3	1.6	61.9	42.0	49.7	46.8
CNN	3	0.9	0.8	0.9	1.0	28.7	17.9	19.5	19.0
	4	2.8	0.6	0.7	0.8	38.1	15.7	19.5	16.4

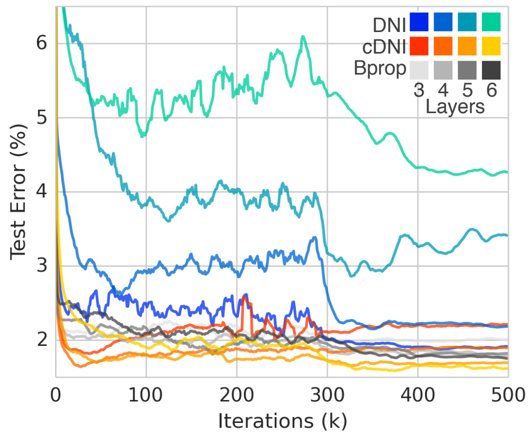


Table 2. Using DNI between every layer for FCNs and CNNs on MNIST and CIFAR-10. *Left:* Summary of results, where values are final test error (%) after 500k iterations. *Right:* Test error during training of MNIST FCN models for regular backpropagation, DNI, and cDNI (DNI where the synthetic gradient model is also conditioned on the labels of the data).

the actor and the synthetic gradient is the critic.

In practice, we allow the parameters to change over the course of training, and therefore the synthetic gradient must learn online to track the gradient $g(h_t, \theta_{t+1:T})$

C. Additional Experiments

Every layer DNI We first look at training an FCN for MNIST digit classification (LeCun et al., 1998b). For an FCN, "layer" refers to a linear transformation followed by batch-normalisation (Ioffe & Szegedy, 2015) and a rectified linear non-linearity (ReLU) (Glorot et al., 2011). All hidden layers have the same number of units, 256. We use DNI as in the scenario illustrated in Fig. 3 (d), where DNIs are used between *every* layer in the network. *E.g.* for a four layer network (three hidden, one final classification) there will be three DNIs. In this scenario, every layer can be updated as soon as its activations have been computed and passed through the synthetic gradient model of the layer above, without waiting for any other layer to compute or loss to be generated. We perform experiments where we

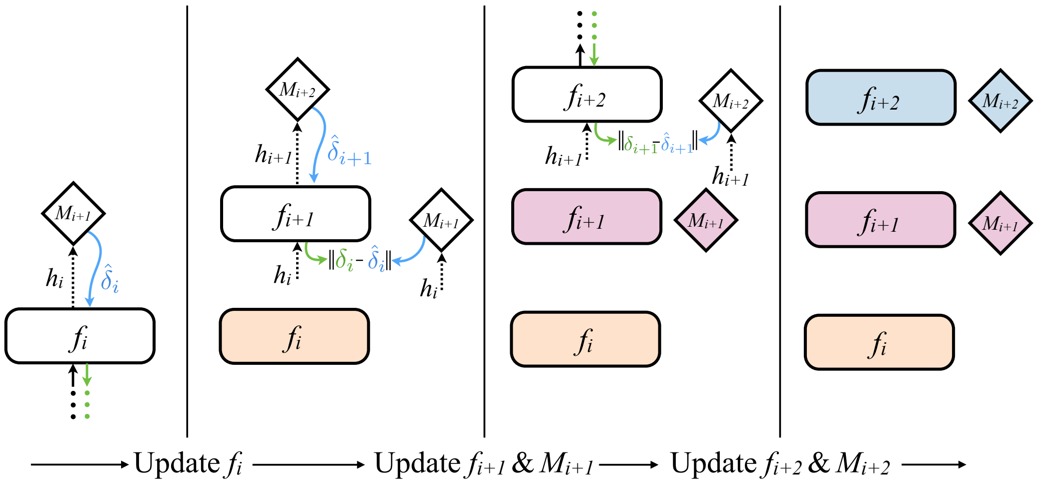

f_i
M_{i+1}
h_i
$\hat{\delta}_i$
f_{i+1}
M_{i+2}
h_{i+1}
$\hat{\delta}_{i+1}$
$\|\delta_i - \hat{\delta}_i\|$
f_{i+2}
$\|\delta_{i+1} - \hat{\delta}_{i+1}\|$
Update f_i
Update f_{i+1} & M_{i+1}
Update f_{i+2} & M_{i+2}

Figure 9. The execution during training of a feed-forward network. Coloured modules are those that have been updated for this batch of inputs. First, layer i executes it's forward phase, producing h_i, which can be used by M_{i+1} to produce the synthetic gradient $\hat{\delta}_i$. The synthetic gradient is pushed backwards into layer i so the parameters θ_i can be updated immediately. The same applies to layer $i+1$ where $h_{i+1} = f_{i+1}(h_i)$, and then $\hat{\delta}_{i+1} = M_{i+2}(h_{i+1})$ so layer $i+1$ can be updated. Next, $\hat{\delta}_{i+1}$ is backpropagated through layer $i+1$ to generate a target error gradient $\delta_i = f'_{i+1}(h_i)\hat{\delta}_{i+1}$ which is used as a target to regress $\hat{\delta}_i$ to, thus updating M_{i+1}. This process is repeated for every subsequent layer.

vary the depth of the model (between 3 and 6 layers), on MNIST digit classification and CIFAR-10 object recognition (Krizhevsky & Hinton, 2009). Full implementation details can be found in Sect. D.1.

Looking at the results in Table 2 we can see that DNI does indeed work, successfully update-decoupling all layers at a small cost in accuracy, demonstrating that it is possible to produce effective gradients *without either label or true gradient information*. Further, once we condition the synthetic gradients on the labels, we can successfully train deep models with very little degradation in accuracy. For example, on CIFAR-10 we can train a 5 layer model, with backpropagation achieving 42% error, with DNI achieving 47% error, and when conditioning the synthetic gradient on the label (cDNI) get 44%. In fact, on MNIST we successfully trained up to 21 layer FCNs with cDNI to 2% error (the same as with using backpropagation). Interestingly, the best results obtained with cDNI were with *linear* synthetic gradient models.

As another baseline, we tried using historical, stale gradients with respect to activations, rather than synthetic gradients. We took an exponential average historical gradient, searching over the entire spectrum of decay rates and the best results attained on MNIST classification were 9.1%, 11.8%, 15.4%, 19.0% for 3 to 6 layer FCNs respectively – marginally better than using zero gradients (no backpropagation) and far worse than the associated cDNI results of 2.2%, 1.9%, 1.7%, 1.6%. Note that the experiment described above used stale gradients with respect to the activations which do not correspond to the same input example used to compute the activation. In the case of a fixed training dataset, one could use the stale gradient from the same input, but it would be stale by an entire epoch and contains no new information so would fail to improve the model. Thus, we believe that DNI, which uses a parametric approximation to the gradient with respect to activations, is the most desirable approach.

This framework can be easily applied to CNNs (LeCun et al., 1998a). The spatial resolution of activations from layers in a CNN results in high dimensional activations, so we use synthetic gradient models which themselves are CNNs without pooling and with resolution-preserving zero-padding. For the full details of the CNN models please refer to Sect. D.1. The results of CNN models for MNIST and CIFAR-10 are also found in Table 2, where DNI and cDNI CNNs perform exceptionally well compared to true backpropagated gradient trained models – a three layer CNN on CIFAR-10 results in 17.9% error with backpropagation, 19.5% (DNI), and 19.0% (cDNI).

Single DNI We look at training an FCN for MNIST digit classification using a network with 6 layers (5 hidden layers, one classification layer), but splitting the network into two unlocked sub-networks by inserting a single DNI at a variable position, as illustrated in Fig. 3 (c).

Fig. 11 (a) shows the results of varying the depth at which the DNI is inserted. When training this 6 layer FCN with vanilla backpropagation we attain 1.6% test error. Incorporating a single DNI between two layers results in between 1.8% and 3.4% error depending on whether the DNI is af-

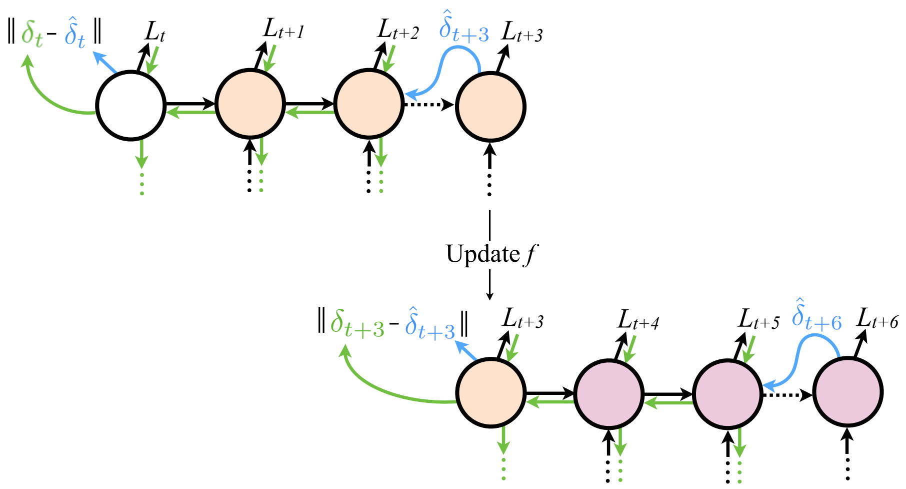


Figure 10. The execution during training of an RNN, with a core function f, shown for $T = 3$. Changes in colour indicate a weight update has occurred. The final core of the last unroll is kept in memory. Fresh cores are unrolled for T steps, and the synthetic gradient from step T (here $\hat{\delta}_{t+3}$ for example) is used to approximate the error gradient from the future. The error gradient is backpropagated through the earliest T cores in memory, which gives a target error gradient for the last time a synthetic gradient was used. This is used to generate a loss for the synthetic gradient output of the RNN, and all the T cores' gradients with respect to parameters can be accumulated and updated. The first T cores in memory are deleted, and this process is repeated. This training requires an extra core to be stored in memory ($T + 1$ rather than T as in normal BPTT). Note that the target gradient of the hidden state that is regressed to by the synthetic gradient model is slightly stale, a similar consequence of online training as seen in RTRL (Williams & Zipser, 1989).

ter the first layer or the penultimate layer respectively. If we decouple the layers without DNI, by just not backpropagating any gradient between them, this results in bad performance – between 2.9% and 23.7% error for after layer 1 and layer 5 respectively.

One can also see from Fig. 11 (a) that as the DNI module is positioned closer to the classification layer (going up in layer hierarchy), the effectiveness of it degrades. This is expected since now a larger portion of the whole system never observes true gradient. However, as we show in Sect. 3.3, using extra label information in the DNI module almost completely alleviates this problem.

We also plot the synthetic gradient regression error (L_2 distance), cosine distance, and the sign error (the number of times the sign of a gradient dimension is predicted incorrectly) compared to the true error gradient in Fig. 12. Looking at the L_2 error, one can see that the error jumps initially as the layers start to train, and then the synthetic gradient model starts to fit the target gradients. The cosine similarity is on average very slightly positive, indicating that the direction of synthetic gradient is somewhat aligned with that of the target gradient, allowing the model to train. However, it is clear that the synthetic gradient is not tracking the true gradient very accurately, but this does not seem to impact the ability to train the classifiers.

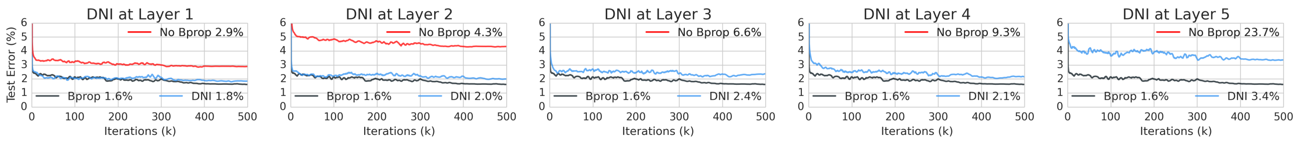
C.1. Underfitting of Synthetic Gradient Models

If one takes a closer look at learning curves for DNI model (see Fig. 15 for training error plot on CIFAR-10 with CNN model) it is easy to notice that the large test error (and its degradation with depth) is actually an effect of underfitting and not lack of ability to generalise or lack of convergence of learning process. One of the possible explanations is the fact that due to lack of label signal in the DNI module, the network is over-regularised as in each iteration DNI tries to model an expected gradient over the label distribution. This is obviously a harder problem than modelling actual gradient, and due to underfitting to this subproblem, the whole network also underfits to the problem at hand. Once label information is introduced in the cDNI model, the network fits the training data much better, however using synthetic gradients still acts like a regulariser, which also translates to a reduced test error. This might also suggest, that the proposed method of conditioning on labels can be further modified to reduce the underfitting effect.

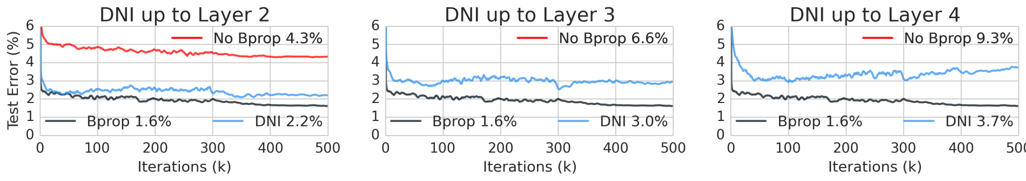
D. Implementation Details

D.1. Feed-Forward Implementation Details

In this section we give the implementation details of the experimental setup used in the experiments from Sect. 3.3.



(a)



(b)

Figure 11. Test error during training of a 6 layer fully-connected network on MNIST digit classification. Bprop (grey) indicates traditional, synchronous training with backpropagation, while DNI (blue) shows the use of a (a) single DNI used after a particular layer indicated above, and (b) every layer using DNI up to a particular depth. Without backpropagating any gradients through the connection approximated by DNI results in poor performance (red).

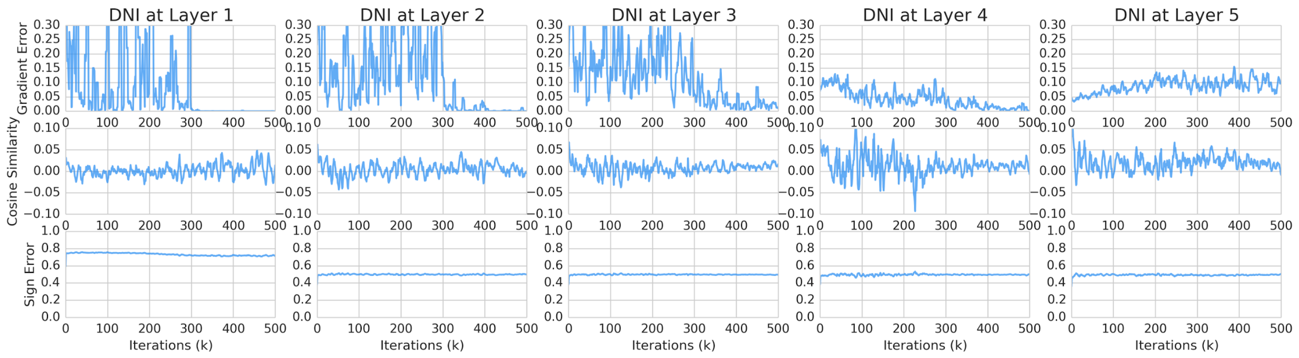


Figure 12. Error between the synthetic gradient and the true backpropagated gradient for MNIST FCN where DNI is inserted at a single position. Sign error refers to the average number of dimensions of the synthetic gradient vector that do not have the same sign as the true gradient.

Conditional DNI (cDNI) In order to provide DNI module with the label information in FCN, we simply concatenate the one-hot representation of a sample's label to the input of the synthetic gradient model. Consequently for both MNIST and CIFAR-10 experiments, each cDNI module takes ten additional, binary inputs. For convolutional networks we add label information in the form of one-hot encoded channel masks, thus we simply concatenate ten additional channels to the activations, nine out of which are filled with zeros, and one (corresponding to sample's label) is filled with ones.

Common Details All experiments are run for 500k iterations and optimised with Adam (Kingma & Ba, 2014) with batch size of 256. The learning rate was initialised at 3×10^{-5} and decreased by a factor of 10 at 300k and 400k steps. Note the number of iterations, learning rate, and learning rate schedule was not optimised. We perform a hyperparameter search over the number of hidden layers in the synthetic gradient model (from 0 to 2, where 0 means we use a linear model such that $\hat{\delta} = M(h) = \phi_w h + \phi_b$) and select the best number of layers for each experiment type (given below) based on the final test performance. We used cross entropy loss for classification and L_2 loss for synthetic gradient regression which was weighted by a factor of 1 with respect to the classification loss. All input data was scaled to $[0, 1]$ interval. The final regression layer of all synthetic gradient models are initialised with zero weights and biases, so initially, zero synthetic gradient is produced.

MNIST FCN Every hidden layer consists of fully-connected layers with 256 units, followed by batch-normalisation and ReLU non-linearity. The synthetic gradient models consists of two (DNI) or zero (cDNI) hidden layers and with 1024 units (linear, batch-normalisation, ReLU) followed by a final linear layer with 256 units.

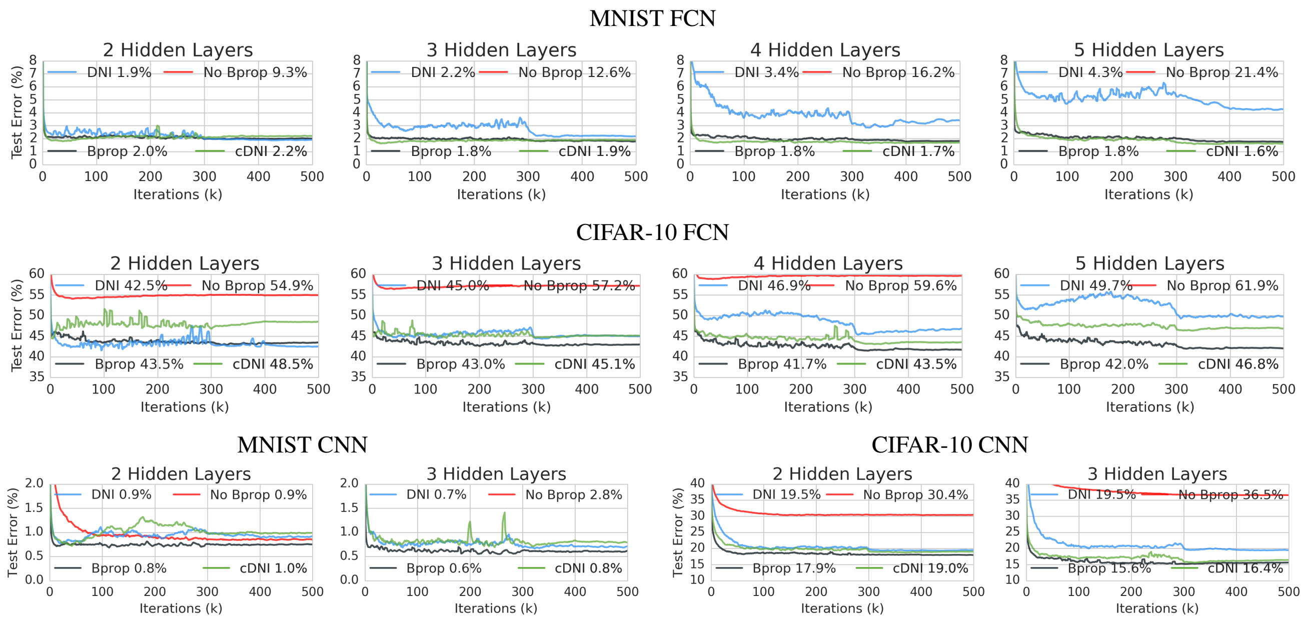


Figure 13. Corresponding test error curves during training for the results in Table 2. (a) MNIST digit classification with FCNs, (b) CIFAR-10 image classification with FCNs. DNI can be easily used with CNNs as shown in (c) for CNNs on MNIST and (d) for CNNs on CIFAR-10.

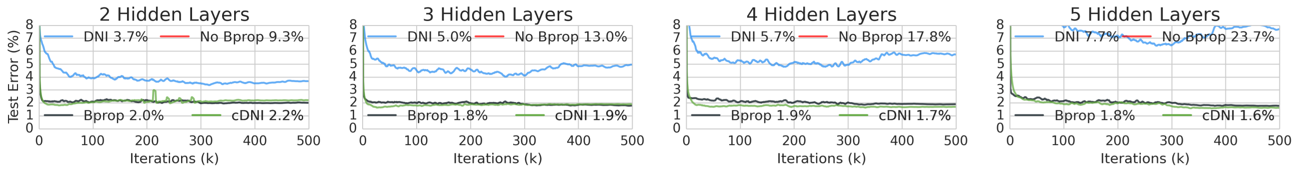


Figure 14. Linear DNI models for FCNs on MNIST.

MNIST CNN The hidden layers are all convolutional layers with 64 5×5 filters with resolution preserving padding, followed by batch-normalisation, ReLU and 3×3 spatial max-pooling in the first layer and average-pooling in the remaining ones. The synthetic gradient model has two hidden layers with 64 5×5 filters with resolution preserving padding, batch-normalisation and ReLU, followed by a final 64 5×5 filter convolutional layer with resolution preserving padding.

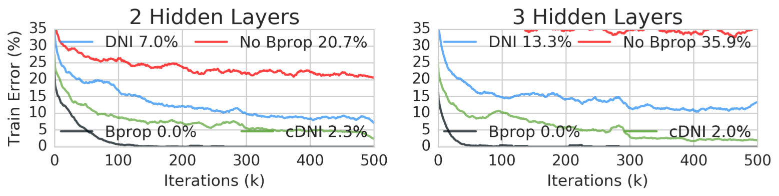
CIFAR-10 FCN Every hidden layer consists of fully-connected layers with 1000 units, followed by batch-normalisation and ReLU non-linearity. The synthetic gradient models consisted of one hidden layer with 4000 units (linear, batch-normalisation, ReLU) followed by a final linear layer with 1000 units.

CIFAR-10 CNN The hidden layers are all convolutional layers with 128 5×5 filters with resolution preserving padding, followed by batch-normalisation, ReLU and 3×3 spatial max-pooling in the first layer and avg-pooling in the remaining ones. The synthetic gradient model has two hidden layers with 128 5×5 filters with resolution preserving padding, batch-normalisation and ReLU, followed by a final 128 5×5 filter convolutional layer with resolution preserving padding.

Complete Unlock. In the completely unlocked model, we use the identical architecture used for the synthetic gradient model, but for simplicity both synthetic gradient and synthetic input models use a single hidden layer (for both DNI and cDNI), and train it to produce synthetic inputs $\hat{h}_i$ such that $\hat{h}_i \simeq h_i$. The overall training setup is depicted in Fig. 6. During testing all layers are connected to each other for a forward pass, *i.e.* the synthetic inputs are not used.

D.2. RNN Implementation Details

Common Details All RNN experiments are performed with an LSTM recurrent core, where the output is used for a final linear layer to model the task. In the case of DNI and DNI+Aux, the output of the LSTM is also used



(a)

Figure 15. (a) Training error for CIFAR-10 CNNs.

as input to a single hidden layer synthetic gradient model with the same number of units as the LSTM, with a final linear projection to two times the number of units of the LSTM (to produce the synthetic gradient of the output and the cell state). The synthetic gradient is scaled by a factor of 0.1 when consumed by the model (we found that this reliably leads to stable training). We perform a hyperparameter search of whether or not to backpropagate synthetic gradient model error into the LSTM core (the model was not particularly sensitive to this, but occasionally backpropagating synthetic gradient model error resulted in more unstable training). The cost on the synthetic gradient regression loss and future synthetic gradient regression loss is simply weighted by a factor of 1.

Copy and Repeat Copy Task In these tasks we use 256 LSTM units and the model was optimised with Adam with a learning rate of 7×10^{-5} and a batch size of 256. The tasks were progressed to a longer episode length after a model gets below 0.15 bits error. The Copy task was progressed by incrementing N, the length of the sequence to copy, by one. The Repeat Copy task was progressed by alternating incrementing N by one and R, the number of times to repeat, by one.

Penn Treebank The architecture used for Penn Treebank experiments consists of an LSTM with 1024 units trained on a character-level language modelling task. Learning is performed with the use of Adam with learning rate of 7×10^-5 (which we select to maximise the score of the baseline model through testing also 1×10^{-4} and 1×10^{-6}) without any learning rate decay or additional regularisation. Each 5k iterations we record validation error (in terms of average bytes per character) and store the network which achieved the smallest one. Once validation error starts to increase we stop training and report test error using previously saved network. In other words, test error is reported for the model yielding minimum validation error measured with 5k iterations resolution. A single iteration consists of performing full BPTT over T steps with a batch of 256 samples.

D.3. Multi-Network Implementation Details

The two RNNs in this experiment, Network A and Network B, are both LSTMs with 256 units which use batch-normalisation as described in (Cooijmans et al., 2016). Network A takes a 28×28 MNIST digit as input and has a two layer FCN (each layer having 256 units and consisting of linear, batch-normalisation, and ReLU), the output of which is passed as input to its LSTM. The output of Network A's LSTM is used by a linear classification layer to classify the number of odd numbers, as well as input to another linear layer with batch-normalisation which produces the message to send to Network B. Network B takes the message from Network A as input to its LSTM, and uses the output of its LSTM for a linear classifier to classify the number of 3's seen in Network A's datastream. The synthetic gradient model has a single hidden layer of size 256 followed by a linear layer which produces the 256-dimensional synthetic gradient as feedback to Network A's message.

All networks are trained with Adam with a learning rate of 1×10^{-5}. We performed a hyperparameter search over the factor by which the synthetic gradient should by multiplied by before being backpropagated through Network A, which we selected as 10 by choosing the system with the lowest training error.

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.

Cooijmans, T., Ballas, N., Laurent, C., and Courville, A. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.

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

Fairbank, M. *Value-gradient learning*. PhD thesis, City University London, UK, 2014.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

Heess, N, Wayne, G, Silver, D, Lillicrap, T P, Erez, T, and Tassa, Y. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 2944–2952, 2015.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*, 2015.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images, 2009.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998a.

LeCun, Y., Cortes, C., and Burges, C. The mnist database of handwritten digits, 1998b.

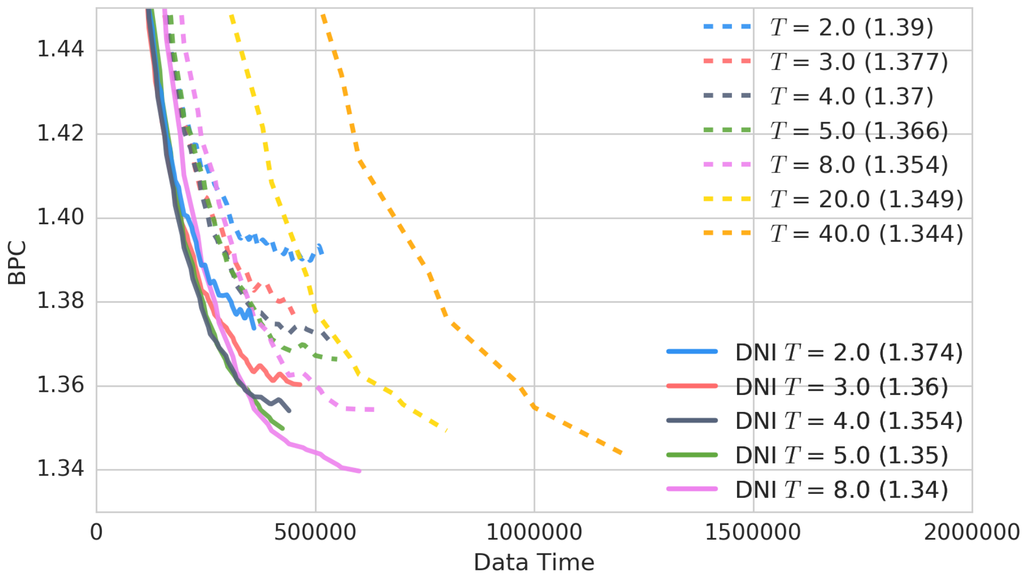
Lillicrap, T P, Cownden, D, Tweed, D B, and Akerman, C J. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7, 2016.

Nøkland, A. Direct feedback alignment provides learning in deep neural networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 1037–1045. Curran Associates, Inc., 2016.

Sutton, R S. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

Werbos, P J. Approximating dynamic programming for real-time control and neural modeling. In White, David A. and Sofge, Donald A. (eds.), *Handbook of Intelligent Control*, chapter 13, pp. 493–525. Van Nostrand Reinhold, New York, 1992.

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



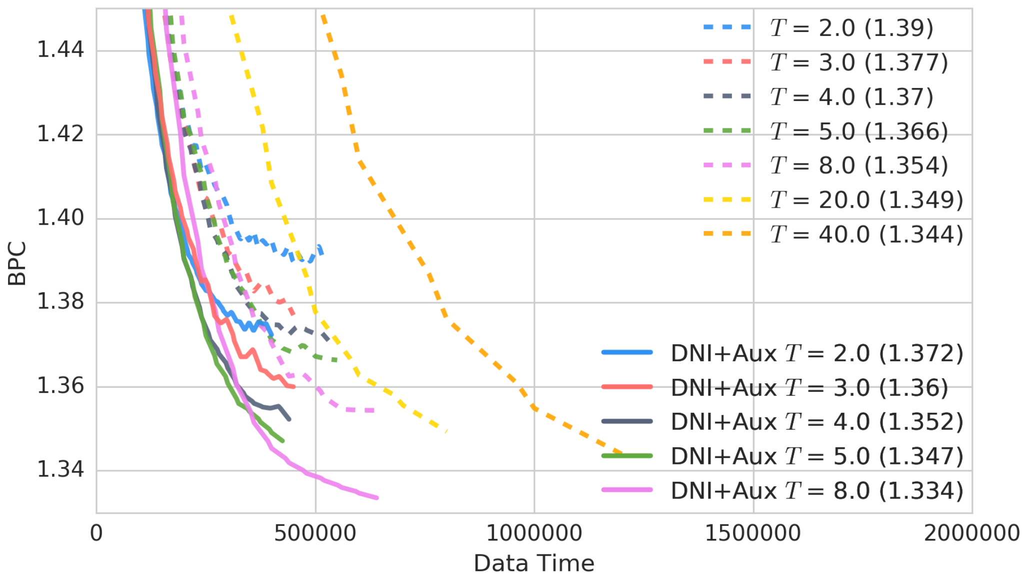


Figure 16. Test error in bits per character (BPC) for Penn Treebank character modelling. We train the RNNs with different BPTT unroll lengths with DNI (solid lines) and without DNI (dashed lines). Early stopping is performed based on the validation set. Top shows results with DNI, and bottom shows results with DNI and future synthetic gradient prediction (DNI+Aux). Bracketed numbers give final test set BPC.

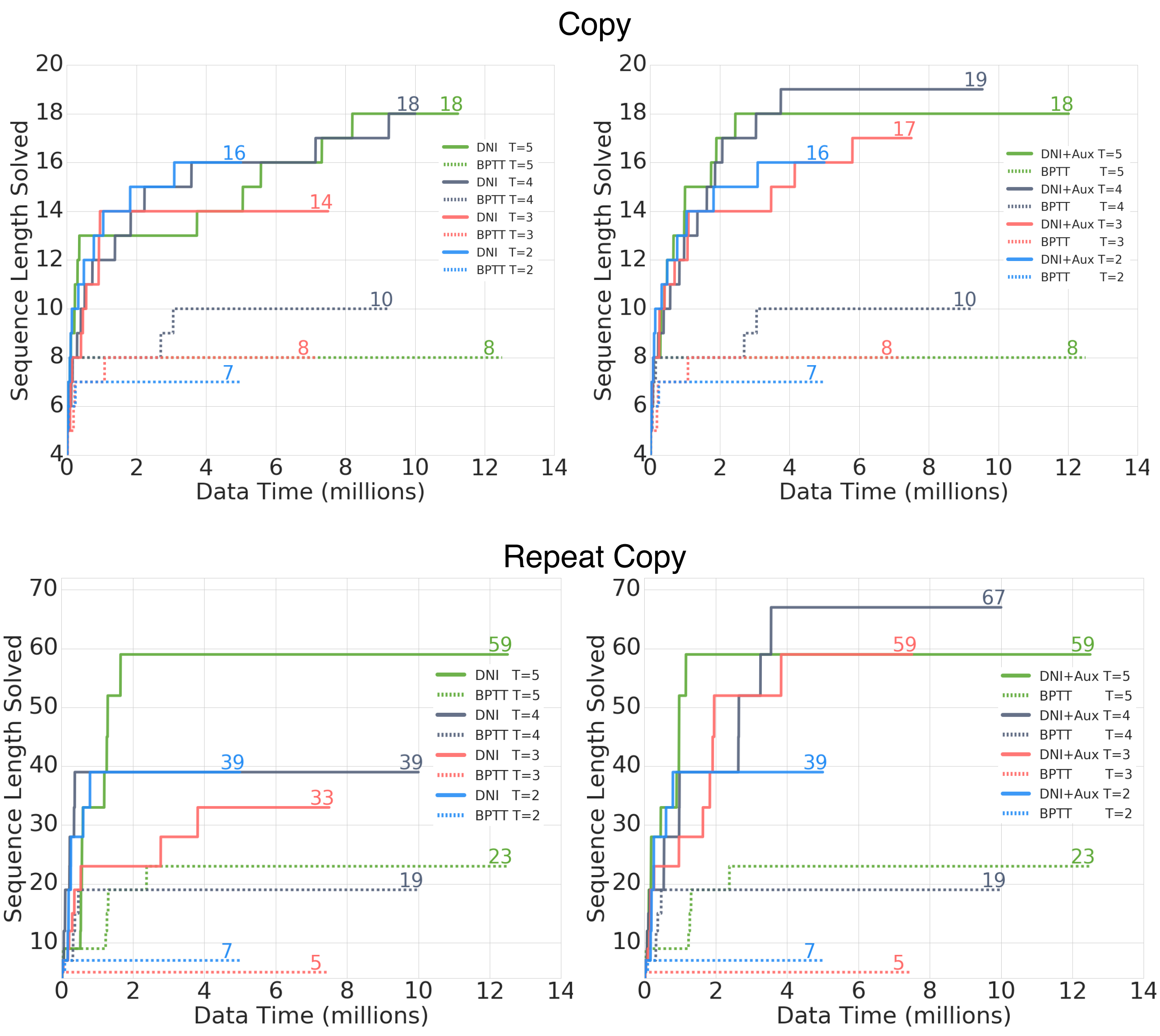


Figure 17. The task progression for Copy (top row) and Repeat Copy (bottom row) without future synthetic gradient prediction (left) and with future synthetic gradient prediction (right). For all experiments the tasks' time dependency is advanced after the RNN reaches 0.15 bits error. We run all models for 2.5M optimisation steps. 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.