HyperAdam: A Learnable Task-Adaptive Adam for Network Training

Shipeng Wang and **Jian Sun*** and **Zongben Xu**

School of Mathematics and Statistics, Xi'an Jiaotong University, Xi'an, 710049, China

wangshipeng8128@stu.xjtu.edu.cn, {jiansun, zbxu}@xjtu.edu.cn

Abstract

Deep neural networks are traditionally trained using human-designed stochastic optimization algorithms, such as SGD and Adam. Recently, the approach of learning to optimize network parameters has emerged as a promising research topic. However, these learned black-box optimizers sometimes do not fully utilize the experience in human-designed optimizers, therefore have limitation in generalization ability. In this paper, a new optimizer, dubbed as *HyperAdam*, is proposed that combines the idea of "learning to optimize" and traditional Adam optimizer. Given a network for training, its parameter update in each iteration generated by HyperAdam is an adaptive combination of multiple updates generated by Adam with varying decay rates . The combination weights and decay rates in HyperAdam are adaptively learned depending on the task. HyperAdam is modeled as a recurrent neural network with AdamCell, WeightCell and StateCell. It is justified to be state-of-the-art for various network training, such as multilayer perceptron, CNN and LSTM.

1 Introduction

Deep learning approach has exhibited strong capabilities in data representation (LeCun, Bengio, and Hinton 2015), nonlinear mapping (Sutskever, Vinyals, and Le 2014), distribution learning (Goodfellow et al. 2014), etc. Deep learning not only has wide applications in a broad field of academical studies, such as image analysis (He et al. 2016), speech recognition (McMahan and Rao 2018), robotics (Lillicrap et al. 2016), inverse problem (Yang et al. 2016), but also draws attention of industry for realization in products.

One challenge in deep learning is the effective optimization of deep network parameters, required to be generalizable to varying network architectures, e.g., network type, depth, width, non-linear activation functions. For a neural network $f(x; w)$, the aim of network training is to find the optimal network parameters $w^* \in \mathbb{R}^p$ to minimize empirical loss between the network output given input $x_i \in \mathbb{R}^d$ and the corresponding target label $y_i \in \mathbb{R}^b$:

$$w^* = \arg\min_w \Sigma_{i=1}^N l(f(x_i; w), y_i), \tag{1}$$

---

*corresponding author

where $\{(x_i, y_i)\}_{i=1}^N$ is the training set. For a deep neural network, the dimension p and number of training data N are commonly large in real applications.

The network, as a learning machine, is referred to as a *learner*, the loss for network training is defined as an *optimizee*, and the optimization algorithm to minimize optimizee is referred to as an *optimizer*. For example, a gradient-based optimizer can be written as a function O that maps the gradient g_t to network parameter update d_t in t-th iteration:

$$d_t(\Theta) = O(g_t, \mathcal{H}_t; \Theta), \tag{2}$$

where $\mathcal{H}_t$ represents the historical gradient information and Θ represents the hyperparameters of the optimizer.

The human-designed optimizers, such as stochastic gradient descent (SGD) (Robbins and Monro 1951), RMSProp (Tieleman and Hinton 2012), AdaGrad (Duchi, Hazan, and Singer 2011), AdaDelta (Zeiler 2012) and Adam (Kingma and Ba 2015), are popular in network training. They have well generalization ability to various network architectures and tasks. Adam takes the statistics of gradients as the historical information recursively accumulated with constant decay rates (i.e., β, γ in Alg. 1). Though universal, Adam suffers from unsatisfactory convergence in some cases because of the constant decay rates (Reddi, Kale, and Kumar 2018).

Recently, "learning to optimize", i.e., learning the optimizer by data-driven approach, triggered the interest of community. The optimizer (Andrychowicz et al. 2016) outputs the update vector by RNN, whose generalization ability is improved by two training tricks (Lv, Jiang, and Li 2017). This idea is also applied to optimizing derivative-free black-box functions (Chen et al. 2017). From the perspective of reinforcement learning, the optimizer is taken as policy (Li and Malik 2016). Though faster in decreasing training loss than the traditional optimizers in some cases, the learned optimizers do not always generalize well to diverse variants of learners. Moreover, they can not be guaranteed to output a descent direction in each iteration for network training.

In this paper, we propose an effective optimizer *Hyper-Adam*, which is a generalized and learnable optimizer inspired by Adam. For network optimization, the parameter update generated by HyperAdam is an ensemble of updates generated by Adam with different decay rates. Both decay rates and combination weights for ensemble are adaptively learned depending on the task. To implement this

Algorithm 1 Adam Optimizer

Require:
1: Initialized parameter w_0, step size α, batch size N_B.
2: Exponential decay rates β, γ; dataset $\{(x_i, y_i)\}_{i=1}^N$.
Initialize: $m_0 = 0, v_0 = 0$.
3: **for all** $t = 1, \ldots, T$ **do**
4: Draw random batch $\{(x_{i_k}, y_{i_k})\}_{k=1}^{N_B}$ from dataset
5: $g_t = \Sigma_{k=1}^{N_B} \nabla l(x_{i_k}, y_{i_k}, w_{t-1})$
6: $m_t = \beta m_{t-1} + (1 - \beta) g_t$ ▷ *moving average*
7: $v_t = \gamma v_{t-1} + (1 - \gamma) g_t^2$
8: $\tilde{m}_t = \frac{m_t}{1-\beta^t}, \tilde{v}_t = \frac{v_t}{1-\gamma^t}$ ▷ *correcting bias*
9: $\hat{m}_t = \frac{\tilde{m}_t}{\sqrt{\tilde{v}_t} + \varepsilon}$
10: $w_t = w_{t-1} - \alpha \hat{m}_t$
11: **end for**
12: **return** final parameter w_T.

Algorithm 2 Task-Adaptive HyperAdam

Require:
1: Initialized parameter w_0, step size α, batch size N_B.
2: Dataset $\{(x_i, y_i)\}_{i=1}^N$.
Initialize:
3: $\mathbf{m}_0, \mathbf{v}_0, \hat{\boldsymbol{\beta}}_0, \hat{\boldsymbol{\gamma}}_0, \mathbf{s}_0 = \mathbf{0} \in \mathbb{R}^{p \times J}$, $\mathbf{1} \in \mathbb{R}^{p \times J}$, ε=1e-24 .
4: **for all** $t = 1, \ldots, T$ **do**
5: Draw random batch $\{(x_{i_k}, y_{i_k})\}_{k=1}^{N_B}$ from dataset
6: $g_t = \Sigma_{k=1}^{N_B} \nabla l(x_{i_k}, y_{i_k}, w_{t-1})$
7: $\mathbf{G_t} = [g_t, \ldots, g_t]$ ▷ $\mathbf{G_t} \in \mathbb{R}^{p \times J}$
8: $\mathbf{s}_t = F_h(\mathbf{s}_{t-1}, g_t; \Theta_h)$ ▷ *current state*
9: $\boldsymbol{\beta}_t \triangleq [\beta_t^1, \ldots, \beta_t^J] = F_u(\mathbf{s}_t, \mathbf{m}_{t-1}; \Theta_u)$
10: $\boldsymbol{\gamma}_t \triangleq [\gamma_t^1, \ldots, \gamma_t^J] = F_r(\mathbf{s}_t, \mathbf{m}_{t-1}; \Theta_r)$
11: $\mathbf{m}_t = \boldsymbol{\beta}_t \odot \mathbf{m}_{t-1} + (1 - \boldsymbol{\beta}_t) \odot \mathbf{G_t}$
12: $\mathbf{v}_t = \boldsymbol{\gamma}_t \odot \mathbf{v}_{t-1} + (1 - \boldsymbol{\gamma}_t) \odot \mathbf{G}_t^2$
13: $\hat{\boldsymbol{\beta}}_t = \boldsymbol{\beta}_t \odot \hat{\boldsymbol{\beta}}_{t-1} + (1 - \boldsymbol{\beta}_t) \odot \mathbf{1}$
14: $\hat{\boldsymbol{\gamma}}_t = \boldsymbol{\gamma}_t \odot \hat{\boldsymbol{\gamma}}_{t-1} + (1 - \boldsymbol{\gamma}_t) \odot \mathbf{1}$
15: $\tilde{\mathbf{m}}_t = \mathbf{m}_t / \hat{\boldsymbol{\beta}}_t, \tilde{\mathbf{v}}_t = \mathbf{v}_t / \hat{\boldsymbol{\gamma}}_t,$ ▷ *correcting bias*
16: $\hat{\mathbf{m}}_t \triangleq [\hat{m}_t^1, \ldots, \hat{m}_t^J] = \frac{\tilde{\mathbf{m}}_t}{\sqrt{\tilde{\mathbf{v}}_t} + \varepsilon}$ ▷ *moment field*
17: $\boldsymbol{\rho}_t \triangleq [\rho_t^1, \ldots, \rho_t^J] = F_q(\mathbf{s}_t; \Theta_q)$ ▷ *weight field*
18: $d_t = \Sigma_{j=1}^J \rho_t^j \odot \hat{m}_t^j$
19: $w_t = w_{t-1} - \alpha d_t$
20: **end for**
21: **return** final parameter w_T.

---

idea, AdamCell and WeightCell are respectively designed to generate candidate updates and weights to combine them, conditioned on the output of StateCell for modeling task-dependent state. As a recurrent neural network, parameters of HyperAdam are learned by training on a meta-train set.

The main contribution of this paper is two-fold. First, to the best of our knowledge, this is a first task-adaptive optimizer taking merits of adaptive moment estimation approach (i.e., Adam) and learning-based approach in a single framework. It opens a new door to design learning-based optimizer inspired by traditional human-designed optimizers. Second, extensive experiments justify that the learned HyperAdam outperforms traditional optimizers, such as Adam and learning-based optimizers for training a wide range of neural networks, e.g., deep MLP, CNN, LSTM.

2 Related Works

2.1 Learning to Optimize

With the goal of facilitating learning of novel tasks, *meta-learning* is developed to extract knowledge from observed tasks (Amit and Meir 2018; Ren et al. 2018; Finn et al. 2017; Snell, Swersky, and Zemel 2017; Wichrowska et al. 2017; Santoro et al. 2016; Daniel, Taylor, and Nowozin 2016).

This paper focuses on the meta-learning task of optimizing network parameters, commonly termed as "learning to optimize". It originates from several decades ago (Schmidhuber 1992; Naik and Mammone 1992) and is developed afterwards (Hochreiter, Younger, and Conwell 2001; Younger, Hochreiter, and Conwell 2001). Recently, a more general optimizer that conducts parameter update by LSTM with gradient as input is proposed in (Andrychowicz et al. 2016). Two effective training techniques, "Random Scaling" and "Combination with Convex Functions", are proposed to improve the generalization ability (Lv, Jiang, and Li 2017). Subsequently, several works use RNN to replace certain process in some optimization algorithms, e.g., variational EM (Marino, Yue, and Mandt 2018), ADMM (Liu et al. 2018). In (Chen et al. 2017), RNN is also used to optimize derivate-free black-box functions.

These pioneering learning-based optimizers have shown promising performance, but did not fully utilize the experience in human-designed optimizers, and sometimes have limitation in generalizing to variants of networks. The proposed optimizer, HyperAdam, is a learnable optimizer but with architecture designed by generalizing traditional Adam optimizer. In the evaluation section, the HyperAdam is justified to have better generalization ability than previous learning-based optimizers for training various networks.

2.2 Adam Method

Vanilla SGD has been improved by adaptive learning rates for each parameter (e.g., AdaGrad, AdaDelta, RMSProp) or (and) Momentum (Tseng 1998). Adam (Kingma and Ba 2015) is an adaptive moment estimation method combining these two techniques, as illustrated in Alg. 1. Adam takes unbiased estimation of second moment of gradients as the ingredient of the coordinate-wise learning rates, and the unbiased estimation of first moment of gradients as the basis for parameter updating. The bias is caused by the initialization of mean and uncentered variance during online moving average with decay rates (i.e., β, γ in Alg. 1). It is easy to verify that the parameters update generated by Adam is invariant to the scale of gradients when ignoring ε.

As observed in (Reddi, Kale, and Kumar 2018), Adam suffers from unsatisfactory convergence due to the constant decay rates when the variance of gradients with respect to optimization steps are large. While, in HyperAdam, the generalized Adam are with learned decay rates adaptive to task state and gradients. Moreover, the ensemble technique
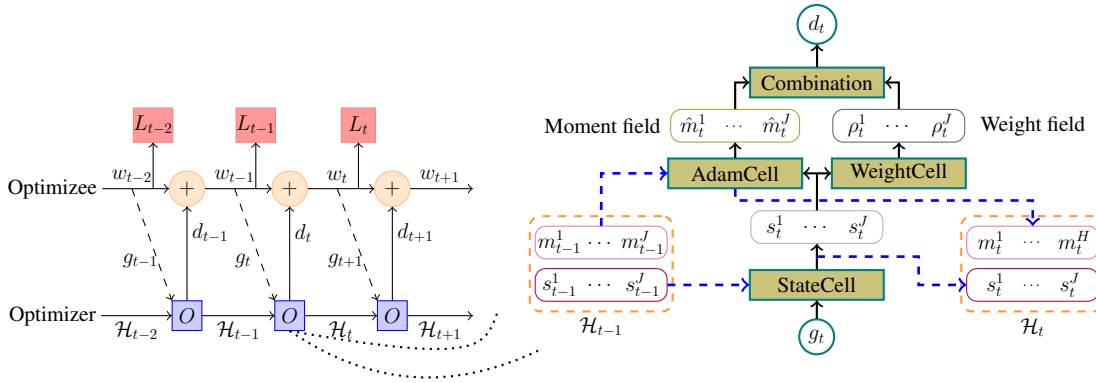
Figure 1: Computational graph of HyperAdam. O represents the optimizer. g_t is the gradient of the optimizee L and d_t is the update vectors. The historical information $\mathcal{H}_t$ consists of the first moment and previous task state.

(Wolpert 1992) of HyperAdam that combines multiple candidate updates can potentially find more reliable descent directions. These techniques are justified to be effective for improving the baseline Adam in evaluation section.

3   HyperAdam

In this section, we introduce the general idea, algorithm and network architecture of the proposed HyperAdam.

3.1   General Idea

Adam is non-adaptive because its hyperparameters (decay rates β and γ in Alg. 1) are constant and set by hand when optimizing a network. According to Alg.1, different hyperparameters make the parameter updates different both in direction and magnitude. Our proposed HyperAdam improves Adam as follows. First, Adam in HyperAdam is designed with multiple learned task-adaptive decay rates and to generate multiple candidate parameter updates with corresponding decay rates in parallel. Second, HyperAdam combines these parameter updates to get the final parameter update using adaptively learned combination weights.

As illustrated in Fig. 2, at a certain point, e.g., w_t in parametric space, multiple update vectors are generated by Adam with different decay rates. The final update d_t is an adaptive combination of these candidate vectors. Considering that, for a deep neural network (Dauphin et al. 2014), there exist abundant saddle points surrounded by high loss
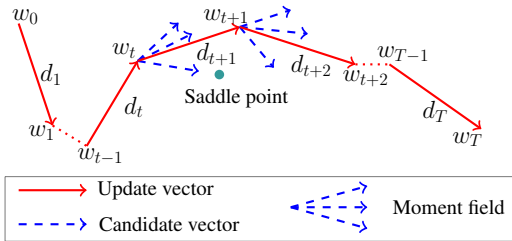
plateaus, a certain candidate update vector may point to a saddle point, but an adaptive combination of several candidate vectors may potentially relieve the possibility of getting stuck in saddle point.

3.2   Task-Adaptive HyperAdam

Based on the above idea, we design a task-adaptive Hyper-Adam in Alg. 2. In iteration t, first, the current state $\mathbf{s}_t$ is determined by *state function* F_h with current gradient g_t and previous state $\mathbf{s}_{t-1}$ as inputs in line 8. Then in lines 9-16, J candidate update vectors $\hat{m}_t^j$ are generated by Adam with J pairs of decay rates (β_t^j, γ_t^j) which are adaptive to the current state $\mathbf{s}_t$ via *decay-rate functions* F_u and F_r. Meanwhile, J task-adaptive weight vectors ρ_t^j are generated by *weight function* F_q with the current state $\mathbf{s}_t$ as input in line 17. Finally, the final update vector d_t is a combination of the candidate updates weighted by weight vectors in line 18. $\hat{\mathbf{m}}_t$ containing candidate updates and $\boldsymbol{\rho}_t$ containing weight vectors are called *moment field* and *weight field* respectively.

As illustrated in the left of Fig. 1, HyperAdam, as an optimizer, is a recurrent mapping O iteratively generating parameter updates. The right of Fig. 1 shows the graphical diagram of HyperAdam having four components. The StateCell corresponds to the *state function* F_h outputting the current state $\mathbf{s}_t = [s_t^1, \ldots, s_t^J]$. With the current state as basis, the *moment field* and *weight field* are produced by AdamCell and WeightCell respectively. The final update d_t is generated in the "Combination" block. We next introduce these components.
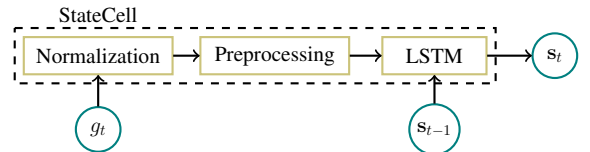

Figure 2: An illustration of parameter optimization of a learner using proposed HyperAdam algorithm.

Figure 3: Diagram of StateCell. Normalization refers to normalizing the gradient g_t with its Euclidean norm.

StateCell The current state $\mathbf{s}_t = [s_t^1, \ldots, s_t^J]$ is determined by the gradient g_t and previous task state $\mathbf{s}_{t-1}$ in the StateCell implementing the *state function* F_h in line 8 of Alg. 2. The diagram of StateCell is illustrated in Fig. 3. After normalized with its Euclidean norm, the gradient is preprocessed by a fully connected layer with Exponential Linear Unit (ELU) (Clevert and and 2016) as activation function. Following preprocessing, the gradient together with the previous task state $\mathbf{s}_{t-1}$ are fed into LSTM (Reiter and Schmidhuber 1997) to generate the current state $\mathbf{s}_t$.

AdamCell AdamCell is designed to implement lines 9-16 in Alg. 2 for generating moment field, i.e. a group of update vectors. We first analyze these lines. Task-adaptive decay rates $\boldsymbol{\beta}_t, \boldsymbol{\gamma}_t$ are generated by *decay-rate functions* in lines 9-10, with which the biased estimations of first and second moment of gradients $\mathbf{m}_t, \mathbf{v}_t$ are recursively accumulated in lines 11-12. The bias factors $\hat{\boldsymbol{\beta}}_t, \hat{\boldsymbol{\gamma}}_t$ are computed in lines 13-14. Finally, moment field is produced with unbiased estimations of first and second moment of gradients in line 16.

Note that the accumulated $\hat{\boldsymbol{\beta}}_t$ (line 13 of Alg. 2) is equivalent to $\mathbf{1} - \boldsymbol{\beta}^t$ when $\boldsymbol{\beta}$ is constant based on lemma 1. It also holds for $\hat{\boldsymbol{\gamma}}_t$. Therefore, we can derive that each component in moment field (line 16 of Alg. 2) is equivalent to a parameter update produced by Adam in line 9 of Alg. 1.

Lemma 1. *$\hat{\beta}_t = \beta \hat{\beta}_{t-1} + (1 - \beta)$ with $\hat{\beta}_0 = 0$ is the online formula of $\hat{\beta}_t = 1 - \beta^t$.*

Proof 1. *See proof in supplementary material.*

If we denote $C_t = [\mathbf{m}_t, \mathbf{v}_t, \hat{\boldsymbol{\beta}}_t, \hat{\boldsymbol{\gamma}}_t]$, $F_t = [\boldsymbol{\beta}_t, \boldsymbol{\gamma}_t, \boldsymbol{\beta}_t, \boldsymbol{\gamma}_t]$ and $\tilde{C}_t = [\mathbf{G}_t, \mathbf{G}_t^2, \mathbf{1}, \mathbf{1}]$ ($\mathbf{1} \in \mathbb{R}^{p \times J}$), based on lemma 1, lines 11-14 in Alg. 2 can be expressed in the following compact formula resembling cell state updating in LSTM:

$$C_t = F_t \odot C_{t-1} + (\mathbf{1} - F_t) \odot \tilde{C}_t. \tag{3}$$

Thus we construct AdamCell, a structure like LSTM, to conduct lines 9-16 in Alg. 2 as illustrated in Fig. 4. F_t determines how much historical information would be forgot like the forget gate in LSTM. We define the *decay-rate functions* F_r, F_u in Alg. 2 to be in parametric forms:

$$\boldsymbol{\beta}_t = \sigma([\mathbf{m}'_{t-1}, \mathbf{s}_t]\boldsymbol{\theta}_u + \mathbf{b}_u), \tag{4}$$

$$\boldsymbol{\gamma}_t = \sigma([\mathbf{m}'_{t-1}, \mathbf{s}_t]\boldsymbol{\theta}_r + \mathbf{b}_r), \tag{5}$$
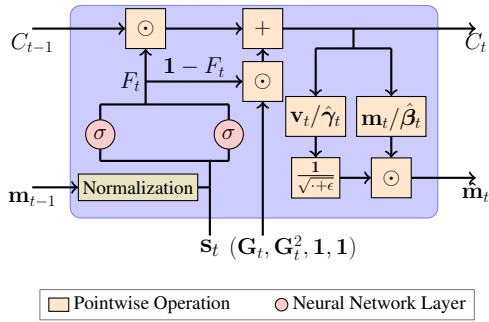

Figure 4: Diagram of AdamCell. The neural network layer corresponds to the decay-rate functions F_u, F_r in Alg. 2.

with $\boldsymbol{\theta}_u, \boldsymbol{\theta}_r \in \mathbb{R}^{2J \times J}$, $\mathbf{b}_u = [b_u, \ldots, b_u]^T \in \mathbb{R}^{p \times J}$, $\mathbf{b}_r = [b_r, \ldots, b_r]^T \in \mathbb{R}^{p \times J}$ and $\mathbf{m}'_{t-1} = [\frac{m_t^1}{\|m_t^1\|_2}, \ldots, \frac{m_t^J}{\|m_t^J\|_2}] \in \mathbb{R}^{p \times J}$, where $\Theta_u = \{\boldsymbol{\theta}_u, b_u\}, \Theta_r = \{\boldsymbol{\theta}_r, b_r\}$ are learnable parameters. The decay-rate functions F_r, F_u output decay rates $\boldsymbol{\beta}_t = [\beta_t^1, \ldots \beta_t^J]$ and $\boldsymbol{\gamma}_t = [\gamma_t^1, \ldots \gamma_t^J]$ respectively, and each pair of decay rates (β_t^j, γ_t^j) determines a candidate update vector $\hat{m}_t^j$ generated by Adam.

WeightCell WeightCell is designed to implement the *weight function* F_q (line 17 in Alg. 2) which outputs the *weight field* with the current state $\mathbf{s}_t$ as input. The weight function is a one-hidden-layer fully connected network with ELU as activation function:

$$\boldsymbol{\rho}_t = \text{ELU}(\mathbf{s}_t \boldsymbol{\theta}_q + \mathbf{b}_q), \tag{6}$$

with $\boldsymbol{\theta}_q \in \mathbb{R}^{J \times J}$ and $\mathbf{b}_q = [b_q, \ldots, b_q]^T \in \mathbb{R}^{p \times J}$ where $\Theta_q = \{\boldsymbol{\theta}_q, b_q\}$ are learnable parameters.

We choose ELU instead of ReLU as activation function to ensure that the weights are not always positive, since some candidate vectors in the *moment field* may not be favorable because of pointing to a bad direction.

Combination The final update d_t is the combination of the candidate update vectors in moment field with weight vectors in weight field (line 18 in Alg. 2):

$$d_t = \Sigma_{j=1}^J \rho_t^j \odot \hat{m}_t^j. \tag{7}$$

Parameter sharing It can be verified that the different coordinates of parameter w and intermediate terms such as $\mathbf{s}_t, \boldsymbol{\beta}_t, \boldsymbol{\gamma}_t$ share the hyperparameter $\Theta = \{\Theta_h, \Theta_q, \Theta_r, \Theta_u\}$ of HyperAdam. For example, different rows of $\boldsymbol{\beta}_t$ in Eqn. (4), corresponding to different coordinates of w, share hyperparameters Θ_u. Moreover, J candidate update vectors are generated in parallel by matrix operations. Consequently, HyperAdam can be applied to training networks with varying dimensional parameters in parallel.

Scale invariance To achieve the scale invariance property same as traditional Adam, the gradient g_t and m_t^j ($j = 1, \ldots, J$) are normalized by their Euclidean norms in StateCell and AdamCell (see proof in supplementary material).

4 Learning HyperAdam

We train HyperAdam on a *meta-train* set consisting of learner (i.e., network in this paper) coupled with corresponding optimizee (loss for training learner) and dataset, which is implemented by TensorFlow. We aim to optimize the parameters of HyperAdam to maximize its capability in training learners over the meta-train set. We expect that the learned HyperAdam can be generalized to optimize more complex networks beyond the learners in meta-train set. We next introduce the training process in details.

Meta-train set consists of triplets of learner f, optimizee L, and dataset $\mathcal{D} = \{X, Y\}$, where $X = \{x_i\}_{i=1}^N$ and $Y = \{y_i\}_{i=1}^N$ represent the data set and corresponding label set. The HyperAdam parameter set Θ is optimized by minimizing the expected cumulative regret (Andrychowicz et al. 2016) on the meta-train set:

$$\mathcal{L}(\Theta) = \mathbb{E}_L[\frac{1}{T}\Sigma_{t=1}^T L(f(X; w_t(\Theta)), Y)], \tag{8}$$

where $w_t(\Theta) = w_{t-1}(\Theta) - \alpha d_t(g_t, \Theta)$ is network parameter of learner f at iteration t when optimized by HyperAdam with parameters $\Theta = \{\Theta_h, \Theta_q, \Theta_r, \Theta_u\}$. $f(X; w_t(\Theta))$ denotes the network output of learner f on dataset $\mathcal{D}$ when network parameter is $w_t(\Theta)$, and $L(\cdot, \cdot)$ is an optimizee, i.e., the loss for training learner f. Therefore, $\mathcal{L}(\Theta)$ defines the expectation of the cumulative loss over meta-train set. Minimizing $\mathcal{L}(\Theta)$ is to find optimal parameter for HyperAdam to reduce training loss L (i.e., optimizee) as lower as possible.

As in (Lv, Jiang, and Li 2017), the learner f is simply taken as a forward neural network with one hidden layer of 20 units and sigmoid as activation function. The optimizee L is defined as $L(f(X; w), Y) = \Sigma_{i=1}^N l(f(x_i; w), y_i)$ where l is the cross entropy loss for the learner f with a minibatch of 128 random images sampled from the MNIST dataset (LeCun et al. 1998). We set the learning rate $\alpha = 0.005$ and maximal iteration $T = 100$ indicating the number of optimization steps using HyperAdam as an optimizer. The number of candidate updates J is set to be 20.

HyperAdam can be seen as a recurrent neural network iteratively updating network parameters. Therefore we can optimize parameter Θ of HyperAdam using BackPropagation Through Time (Werbos 1990) by minimizing $\mathcal{L}(\Theta)$ with Adam, and the expectation with respect to L is approximated by the average training loss for learner f with different initializations. The $T = 100$ steps are split into 5 periods of 20 steps to avoid gradient vanishing. In each period, the initial parameter w_0 and initial hidden state $\mathcal{H}$ are initialized from the last period or generated if it is the first period.

Two training tricks proposed in (Lv, Jiang, and Li 2017) are used here. First, in order to make the training easier, a k-dimensional convex function $h(z) = \frac{1}{k}\|z - \eta\|^2$ is combined with the original optimizee (i.e., training loss), and this trick is called "**Combination with Convex Function**" (CC). η and initial value of z are generated randomly. Second, "**Random Scaling**" (RS), helping to avoid over-fitting, randomly samples vectors c_1 and c_2 of the same dimension as parameter w and z respectively, and then multiply the parameters with c_1 and c_2 coordinate-wisely, thus the optimizee in the meta-train set becomes:

$$L_{ext}(w, z) = L(f(X; c_1 \odot w), Y) + h(c_2 \odot z), \quad (9)$$

with initial parameters $diag(c_1)^{-1}w, diag(c_2)^{-1}z$.

5 Evaluation

We have trained HyperAdam based on 1-layer MLP (basic MLP), we now evaluate the learned HyperAdam for more complex networks such as basic MLP with different activation functions, deeper MLP, CNN and LSTM.

- **Activation functions:** The activation function of basic MLP is extended from sigmoid to ReLU, ELU and tanh.

- **Deep MLP:** The number of hidden layers of MLP is extended to range of $[2, 10]$, and each layer has 20 hidden units and uses sigmoid as activation function.

- **CNN:** Convolution neural networks are with structures of c-c-p-f (CNN-1) and c-c-p-c-c-p-f-f (CNN-2), where c, p and f represent convolution, max-pooling and fully-connected layer respectively. Convolution kernel is with

size of 3×3 and the max-pooling layer is with size of 2×2 and stride 2. CNN-1 and CNN-2 are also trained with batch normalization and dropout respectively.

- **LSTM:** LSTM with hidden state in size of 20 is applied to sequence prediction task using mean squared error loss as in (Lv, Jiang, and Li 2017). Given a sequence $f(0), \ldots, f(9)$ with additive noise, the LSTM is supposed to predict the value of $f(10)$. Here $f(x) = A\sin(wx+\phi)$. The dataset is generated with uniformly random sampling $A \sim U(0, 10), w \sim U(0, \pi/2), \phi \sim U(0, 2\pi)$, and the noise is drawn from Gaussian distribution $N(0, 0.1)$.

We also evaluate whether our learned HyperAdam can well generalize to different datasets, e.g. CIFAR-10 (Krizhevsky 2009). Moreover, the HyperAdam is trained assuming it iteratively optimizes network parameters in fixed iterations $T = 100$, we also evaluate the learned HyperAdam for longer iterative optimization steps as in (Lv, Jiang, and Li 2017). The generalization ability of the networks trained by HyperAdam will be also evaluated preliminarily.

In evaluations, we will compare our HyperAdam with traditional network optimizers such as SGD, AdaDelta, Adam, AdaGrad, Momentum, RMSProp, and state-of-the-art learning-based optimizers including RNNprop (Lv, Jiang, and Li 2017), DMoptimizer (Andrychowicz et al. 2016). For the traditional optimizers, we hand-tuned the learning rates and set other hyperparameters as defaults in TensorFlow. All the initial parameters of learners used in the experiments are sampled independently from the Gaussian distribution. We report the quantitative value as the average measure for training the learner 100 times with random parameter initialization.

5.1 Generalization with Fixed Optimization Steps

We first assume that the learned HyperAdam optimizes the parameters of learners for fixed optimization steps $T = 100$, same as the learning procedure for HyperAdam.

Activation functions As shown in Table 1, HyperAdam is tested for training basic MLP with different activation functions on MNIST dataset, the loss values in Table 1 show that HyperAdam can best generalize to optimize basic MLP with ReLU, ELU and tanh as activation functions, compared with DMoptimizer and RNNprop. Our HyperAdam also outperforms the basic Adam algorithm. The DMoptimizer can not well generalize to basic MLP with ELU activation function, which can be also visually observed in Fig. 5(a).

Deep MLP We further evaluate performance of HyperAdam on learning parameters of MLPs with varying layer

Activation	Adam	DMoptimizer	RNNprop	HyperAdam
sigmoid	0.35	0.38	0.34	**0.33**
ReLU	0.32	1.42	0.31	**0.29**
ELU	0.31	2.02	0.31	**0.28**
tanh	0.34	0.83	**0.33**	0.36

Table 1: Performance for training basic MLP in 100 steps with different activation functions. Each value is the average final loss for optimizing networks in 100 times.
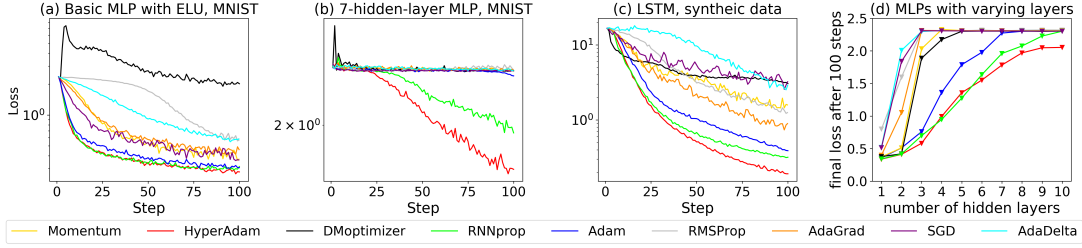
Figure 5: HyperAdam performs best compared with other optimizers on neural networks with different structures.
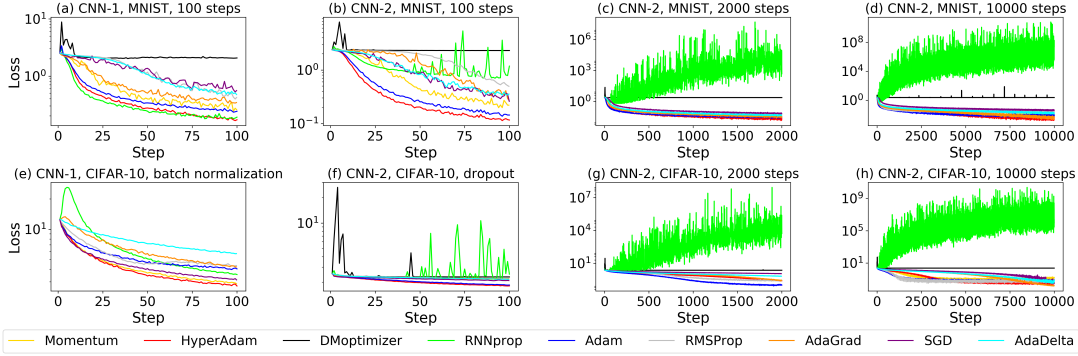

Figure 6: Comparison of different optimizers for optimizing different CNNs in different optimization steps.

numbers. According to Fig. 5(d), for different number of hidden layers ranging from 1 to 10, HyperAdam always performs significantly better than Adam and DMoptimizer. Compared with RNNprop, HyperAdam is better in general, especially for deeper MLP with more than 6 layers. The loss curves in Fig. 5(b) of different optimizers for MLP with 7 hidden layers illustrate HyperAdam is significantly better.

LSTM As shown in Table 2, the "Baseline" task is to utilize one-layer LSTM to predict $f(10)$ on dataset with noise drawn from $N(0, 0.1)$, which is further varied by training on dataset with small noise drawn from $N(0, 0.01)$ ("Small noise") or using two-layer LSTM ("2-layer") for prediction. By comparing the loss values in Table 2, our HyperAdam can better decrease the training losses than the compared optimizers, i.e., Adam, DMoptimizer, RNNprop, HyperAdam. Specifically, Fig. 5(c) shows an example for the comparison in task of "Small noise".
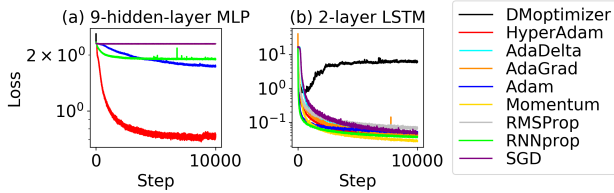

Figure 7: HyperAdam with 10000 optimization steps. Training curves by DMoptimizer, AdaGrad, RMSProp, AdaDelta, Momentum and SGD coincide in the left figure.

Task	Adam	DMoptimizer	RNNprop	HyperAdam
Baseline	0.65	3.10	0.49	**0.42**
Small noise	0.39	3.06	0.32	**0.19**
2-layer	0.51	2.05	0.27	**0.26**

Table 2: Performance on different sequence prediction tasks.

CNN Figure 6(a)-(b) compare training curves of CNN-1 and CNN-2 on MNIST using different optimizers. Figure 6(e)-(f) compare training curves of CNN-1 with batch normalization and CNN-2 with dropout on CIFAR-10 respectively. In these figures, DMoptimizer and RNNprop do not always perform well or even fail while HyperAdam can effectively decrease the training losses in different tasks.

5.2 Generalization to Longer Horizons

We have evaluated HyperAdam for optimizing different learners in fixed optimization steps ($T = 100$), same as the meta-training phase. We now evaluate HyperAdam for its effectiveness in running optimization for longer steps.

Deep MLP Figure 7(a) illustrates the training curves of MLP with 9 hidden layers on MNIST using different optimizers for 10000 steps. DMoptimizer and almost all traditional optimizers, including SGD, Momentum, AdaGrad, AdaDelta and RMSProp, fail to decrease the loss. Our HyperAdam can effectively decrease the training loss.

LSTM The comparison of training two-layer LSTM to predict $f(10)$ with different optimizers for 10000 steps is shown in Fig. 7(b). DMoptimizer decreases the loss first
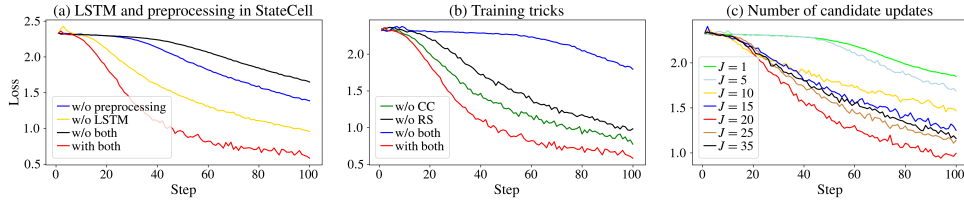
Figure 8: Ablation study for the number of candidate updates, training tricks and structure in StateCell. CC denotes "Combination with Convex function", and RS denotes "Random Scaling".

and then increases the loss dramatically. With similar performance to the traditional optimizers, such as AdaGrad and AdaDelta, our HyperAdam and RNNprop perform better than RMSProp and SGD.

CNN   Figure 6(c)-(d) show the training curves of CNN-2 for 2000 steps and 10000 steps on MNIST dataset. Both RNNprop and DMoptimizer fail to decrease the loss. However, HyperAdam manages to decrease the loss of CNNs and performs slightly better than the traditional network optimizers, such as SGD, Adam and AdaDelta. When training CNN-2 on CIFAR-10 dataset, HyperAdam does not perform as fast as Adam and RMSProp for the first 2000 steps according to Fig. 6(g), but achieves lower loss at 10000th step as shown in Fig. 6(h), while RNNprop and DMoptimizer fail to sufficiently decrease the training loss.

5.3   Generalization of the Learners

The generalization ability of the learners trained by DMoptimizer, RNNprop, Adam and HyperAdam for 10000 steps is evaluated. Table 3 shows the loss, top-1 error and top-2 error of the two learners, CNN-1 and CNN-2 on dataset MNIST, which shows the generalization of learners trained by HyperAdam and Adam are significantly better than those trained by DMoptimizer and RNNprop.

5.4   Ablation Study

We next perform ablation study to justify the effectiveness of key components in HyperAdam.

Task	Measure	Adam	DMoptimizer	RNNprop	HyperAdam
CNN-1 (MNIST)	loss	0.10	2.30	0.36	**0.05**
	top-1	**98.50**%	10.10%	96.46%	98.48%
	top-2	99.59%	20.38%	99.03%	**99.63%**
CNN-2 (MNIST)	loss	0.09	2.30	2.30	**0.07**
	top-1	98.98%	11.35%	11.37%	**99.02%**
	top-2	**99.80**%	21.45%	21.69%	99.78%

Table 3: Generalization of the learner trained by Adam, DMoptimizer, RNNprop and HyperAdam for 10000 steps.

LSTM and preprocessing in StateCell   Figure 8(a) illustrates that HyperAdam achieves lower loss than HyperAdam without LSTM block or (and) preprocessing for training 3-hidden-layer MLP on MNIST, which reflects that the LSTM block and preprocessing help strengthen HyperAdam.

Training tricks   We justify the effectiveness of "Random Scaling" and "Combination with Convex Functions" in our proposed HyperAdam. As shown in Fig. 8(b), HyperAdam trained with both two tricks performs better than HyperAdam trained with either one of them and neither of them for training loss of 3-hidden-layer MLP on MNIST as optimizee, which indicates that the two tricks can enhance the generalization ability of learned HyperAdam.

Number of candidate updates   Figure 8(c) shows the comparison for optimizing cross entropy loss of 4-hidden-layer MLP on MNIST dataset with HyperAdam having different number of candidate updates ($J = 1, 5, 10, 15, 20, 25, 35$). It is observed that the performance of HyperAdam is improved first with the increase of J until 20 achieving best performance, then becomes saturated and decreased with larger number of candidate updates. But all the HyperAdams with $J = 5, 10, 15, 25, 35$ are better than the baseline with single candidate update.

5.5   Computation Time

The time for computing each update by HyperAdam is roughly the same with that of DMoptimizer and RNNprop. For example, the time consuming for computing each update given gradient of a 9-hidden-layer MLP by DMoptimizer, HyperAdam and RNNprop is 0.0023s, 0.0033s and 0.0039s respectively in average. Though faster for computing each update than HyperAdam, Adam is not as efficient as HyperAdam to sufficiently decrease the training loss. When training 8-hidden-layer MLP, HyperAdam takes 26.33s to decrease the loss to 0.6 (the lowest loss that Adam can achieve) while Adam takes 28.97s.

6   Conclusion and Future Work

In this paper, we proposed a novel optimizer HyperAdam implementing "learning to optimize" inspired by the traditional Adam optimizer and ensemble learning. It adaptively combines the multiple candidate parameter updates generated by Adam with multiple adaptively learned decay rates. Based on this motivation, a carefully designed RNN was proposed for implementing HyperAdam optimizer. It was justified to outperform or match traditional optimizers such as Adam, SGD and state-of-art learning-based optimizers in diverse networks training tasks.

In the future, we are interested in applying HyperAdam to train larger scale and more complex networks in vision, NLP, etc., and modeling the correlations among parameter coordinates to further enhance its performance.

7 Acknowledgment

References

Amit, R., and Meir, R. 2018. Meta-learning by adjusting priors based on extended PAC-Bayes theory. In *ICML*, 205–214.

Andrychowicz, M.; Denil, M.; Gomez, S.; Hoffman, M. W.; Pfau, D.; Schaul, T.; Shillingford, B.; and De Freitas, N. 2016. Learning to learn by gradient descent by gradient descent. In *NIPS*, 3981–3989.

Chen, Y.; Hoffman, M. W.; Colmenarejo, S. G.; Denil, M.; Lillicrap, T. P.; Botvinick, M.; and Freitas, N. 2017. Learning to learn without gradient descent by gradient descent. In *ICML*, 748–756.

Clevert, D., and and, T. U. 2016. Fast and accurate deep network learning by exponential linear units (elus). In *ICLR*.

Daniel, C.; Taylor, J.; and Nowozin, S. 2016. Learning step size controllers for robust neural network training. In *AAAI*, 1519–1525.

Dauphin, Y. N.; Pascanu, R.; Gulcehre, C.; Cho, K.; Ganguli, S.; and Bengio, Y. 2014. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*, 2933–2941.

Duchi, J.; Hazan, E.; and Singer, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12:2121–2159.

Finn, C.; Yu, T.; Zhang, T.; Abbeel, P.; and Levine, S. 2017. One-shot visual imitation learning via meta-learning. In *CoRL*, 357–368.

Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; and Bengio, Y. 2014. Generative adversarial nets. In *NIPS*, 2672–2680.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *CVPR*, 770–778.

Hochreiter, S.; Younger, A. S.; and Conwell, P. R. 2001. Learning to learn using gradient descent. In *ICANN*, 87–94.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Krizhevsky, A. 2009. Learning multiple layers of features from tiny images. Technical report.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *Nature* 521(7553):436.

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

Li, K., and Malik, J. 2016. Learning to optimize. In *ICLR*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous control with deep reinforcement learning. In *ICLR*.

Liu, R.; Fan, X.; Cheng, S.; Wang, X.; and Luo, Z. 2018. Proximal alternating direction network: A globally converged deep unrolling framework. In *AAAI*, 1371–1378.

Lv, K.; Jiang, S.; and Li, J. 2017. Learning gradient descent: Better generalization and longer horizons. In *ICML*, 2247–2255.

Marino, J.; Yue, Y.; and Mandt, S. 2018. Learning to infer. In *ICLR*.

McMahan, B., and Rao, D. 2018. Listening to the world improves speech command recognition. In *AAAI*, 378–385.

Naik, D. K., and Mammone, R. 1992. Meta-neural networks that learn by learning. In *IJCNN*, 437–442.

Reddi, S. J.; Kale, S.; and Kumar, S. 2018. On the convergence of adam and beyond. In *ICLR*.

Reiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* 9:1735–80.

Ren, M.; Ravi, S.; Triantafillou, E.; Snell, J.; Swersky, K.; Tenenbaum, J. B.; Larochelle, H.; and Zemel, R. S. 2018. Meta-learning for semi-supervised few-shot classification. In *ICLR*.

Robbins, H., and Monro, S. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* 22(3):400–407.

Santoro, A.; Bartunov, S.; Botvinick, M.; Wierstra, D.; and Lillicrap, T. 2016. Meta-learning with memory-augmented neural networks. In *ICML*, 1842–1850.

Schmidhuber, J. 1992. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation* 4(1):131–139.

Snell, J.; Swersky, K.; and Zemel, R. 2017. Prototypical networks for few-shot learning. In *NIPS*, 4077–4087.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*, 3104–3112.

Tieleman, T., and Hinton, G. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning* 4(2):26–31.

Tseng, P. 1998. An incremental gradient (-projection) method with momentum term and adaptive stepsize rule. *SIAM Journal on Optimization* 8(2):506–531.

Werbos, P. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* 78:1550 – 1560.

Wichrowska, O.; Maheswaranathan, N.; Hoffman, M. W.; Colmenarejo, S. G.; Denil, M.; de Freitas, N.; and Sohl-Dickstein, J. 2017. Learned optimizers that scale and generalize. In *ICML*, 3751–3760.

Wolpert, D. H. 1992. Stacked generalization. *Neural Networks* 5(2):241–259.

Yang, Y.; Sun, J.; Li, H.; and Xu, Z. 2016. Deep admm-net for compressive sensing mri. In *NIPS*, 10–18.

Younger, A. S.; Hochreiter, S.; and Conwell, P. R. 2001. Meta-learning with backpropagation. In *IJCNN*, 2001–2006.

Zeiler, M. D. 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

8   Supplementation

In this supplementary material, we will prove lemma 1 and the claim that HyperAdam is invariant to the scale of gradient (scale invariance) in the submitted paper.

We first present the proof of lemma 1 in Section 3.2.

Lemma. $\hat{\beta}_t = \beta\hat{\beta}_{t-1} + (1 - \beta)$ with $\hat{\beta}_0 = 0$ is the online formula of $\hat{\beta}_t = 1 - \beta^t$.

Proof. Since $\hat{\beta}_0 = 0$, we derive that:

$$
\begin{aligned}
\hat{\beta}_t &= \beta\hat{\beta}_{t-1} + (1 - \beta) \\
&= \beta(\beta\hat{\beta}_{t-2} + (1 - \beta)) + (1 - \beta) \\
&= \cdots \\
&= \beta^t\hat{\beta}_0 + (1 - \beta)\Sigma_{i=1}^t\beta^{t-i} \\
&= 1 - \beta^t,
\end{aligned}
$$

so the lemma is proved. ∎

We then prove the claim on **scale invariance** described in Section 3.2.

Claim. Same to Adam, HyperAdam is also invariant to the scale of gradient when ignoring ε.

Proof. In order to prove the claim, we only need to verify that $\mathbf{s}_t, \boldsymbol{\beta}_t, \boldsymbol{\gamma}_t$ and $\boldsymbol{\rho}_t$ are invariant to the scale of gradient.

Since the gradient will be normalized when it acts as input of the state function F_h corresponding to StateCell, we can rewrite F_h as

$$
\mathbf{s}_t = F_h(\mathbf{s}_{t-1}, g_t; \Theta_h) \triangleq F_h'\left(\mathbf{s}_{t-1}, \frac{g_t}{\|g_t\|}; \Theta_h\right). \qquad (10)
$$

We next consider the scaled gradient sg_t:

$$
\begin{aligned}
F_h(\mathbf{s}_{t-1}, sg_t; \Theta_h) &= F_h'\left(\mathbf{s}_{t-1}, \frac{sg_t}{\|sg_t\|}; \Theta_h\right) \\
&= F_h'\left(\mathbf{s}_{t-1}, \frac{g_t}{\|g_t\|}; \Theta_h\right) \\
&= F_h(\mathbf{s}_{t-1}, g_t; \Theta_h).
\end{aligned}
$$

so $\mathbf{s}_t$ outputted by StateCell is invariant to the scale of the gradient. We can similarly derive that $\boldsymbol{\rho}_t = F_q(\mathbf{s}_t; \Theta_q)$, corresponding to the WeightCell, is invariant to the scale of gradient. Furthermore, the fact that $\boldsymbol{\beta}_t, \boldsymbol{\gamma}_t$ are invariant to the scale of gradient can also be proved in the same way.

Since decay rates $\boldsymbol{\beta}_t, \boldsymbol{\gamma}_t$ are scale invariant, candidate updates $\hat{m}_t^j$ ($j = 1, \ldots, J$) outputted by AdamCell are invariant to the scale of gradient. Therefore, the final update $d_t = \Sigma_{j=1}^J\rho_t^j \odot \hat{m}_t^j$ is invariant to the scale of gradient. ∎