OPT2020: 12th Annual Workshop on Optimization for Machine Learning

DDPNOpt: Differential Dynamic Programming Neural Optimizer

Guan-Horng Liu GHLIU@GATECH.EDU
Tianrong Chen TIANRONG.CHEN@GHLIU.EDU
Evangelos A. Theodorou EVANGELOS.THEODOROU@GHLIU.EDU
Georgia Institute of Technology, Atlanta, GA 30332, USA

Abstract

Interpretation of Deep Neural Networks (DNNs) training as an optimal control problem with nonlinear dynamical systems has received considerable attention recently, yet the algorithmic development remains relatively limited. In this work, we make an attempt along this line by first showing that most widely-used algorithms for training DNNs can be linked to the Differential Dynamic Programming (DDP), a celebrated second-order method rooted in trajectory optimization. In this vein, we propose a new class of optimizer, DDP Neural Optimizer (DDPNOpt), for training DNNs. DDPNOpt features layer-wise feedback policies which improve convergence and robustness. It outperforms other optimal-control inspired training methods in both convergence and complexity, and is competitive against state-of-the-art first and second order methods. Our work opens up new avenues for principled algorithmic design built upon the optimal control theory.

1. Introduction

In this work, we consider the following optimal control problem (OCP) in the discrete-time setting:

$$\min_{\bar{\boldsymbol{u}}} J(\bar{\boldsymbol{u}};\boldsymbol{x}_0) := \left[\phi(\boldsymbol{x}_T) + \sum_{t=0}^{T-1} \ell_t(\boldsymbol{x}_t,\boldsymbol{u}_t)\right] \quad \text{s.t. } \boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t,\boldsymbol{u}_t)\,, \tag{OCP}$$

where $\boldsymbol{x}_t \in \mathbb{R}^n$ and $\boldsymbol{u}_t \in \mathbb{R}^m$ represent the state and control at each time step t. $f_t(\cdot,\cdot)$, $\ell_t(\cdot,\cdot)$ and $\phi(\cdot)$ respectively denote the nonlinear dynamics, intermediate cost and terminal cost functions. OCP aims to find a control trajectory, $\bar{\boldsymbol{u}} \triangleq \{\boldsymbol{u}_t\}_{t=0}^{T-1}$, such that the accumulated cost J over the finite horizon T is minimized. Problems with the form of OCP describes a generic multi-stage decision making problem [4], and have gained commensurate interest recently in deep learning [14, 23].

Central to the research along this line is the interpretation of DNNs as *discrete-time nonlinear dynamical systems*, where each layer is viewed as a distinct time step [7, 15, 16, 23]. When we further regard network weights as *control variables*, OCP describes w.l.o.g. the training objective composed of layer-wise loss and terminal loss. This perspective (see Table 1) has been explored recently for theoretical analysis [19, 24]. Algorithmically, however, OCP-inspired optimizers remain limited, often restricted to specific network class (*e.g.* discrete weight) or small dataset [11, 12].

The aforementioned works are primarily inspired by the Pontryagin Maximum Principle (PMP, [3]). Another parallel methodology which receives little attention is the Approximate Dynamic Programming (ADP, [2]). ADP differs from PMP in that at each time step a locally optimal *feedback policy* is computed. These policies are known to enhance the numerical stability of the optimization process when models admit chain structures (*e.g.* in DNNs) [13, 20]. Practical ADP algorithms such

© G.-H. Liu, T. Chen & E.A. Theodorou.

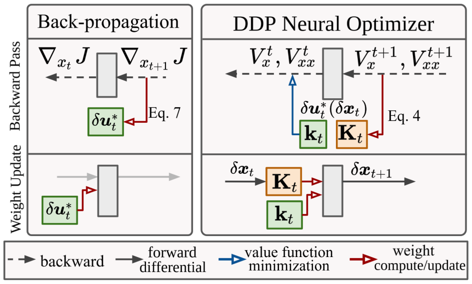

Figure 1: Computational graph.

Table 1: Terminology mapping

	Deep Learning	Optimal Control
J	Total Loss	Trajectory Cost
$\boldsymbol{x}_t$	Activation Vector	State Vector
$\boldsymbol{u}_t$	Weight Parameter	Control Vector
f	Layer Propagation	Dynamical System
ϕ	End-goal Loss	Terminal Cost
ℓ	Weight Decay	Intermediate Cost

as the Differential Dynamic Programming (DDP, [9]) appear extensively in modern autonomous systems [8, 21]. However, whether they can be lifted to large-scale optimization remains unclear.

In this work, we make a significant advance toward optimal-control-theoretic training algorithms inspired by ADP. We first draw a novel perspective of DNN training from trajectory optimization, based on a theoretical connection between existing training methods and the DDP algorithm. We then present a new class of optimizer, **DDPNOpt**, that performs a distinct backward pass inherited with Bellman optimality and generates layer-wise feedback policies to robustify the training. We show that DDPNOpt achieves competitive performance on classification datasets and outperforms previous OCP-inspired methods in both training performance and runtime complexity.

2. Preliminaries

Theorem 1 (Bellman Optimality [1]) *Define a value function $V_t : \mathbb{R}^n \mapsto \mathbb{R}$ at each time step that is computed backward in time using the Bellman equation*

$$V_t(\boldsymbol{x}_t) = \min_{\boldsymbol{u}_t(\boldsymbol{x}_t) \in \Gamma_{\boldsymbol{x}_t}} \underbrace{\ell_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + V_{t+1}(f_t(\boldsymbol{x}_t, \boldsymbol{u}_t))}_{Q_t(\boldsymbol{x}_t, \boldsymbol{u}_t) \equiv Q_t}, \quad V_T(\boldsymbol{x}_T) = \phi(\boldsymbol{x}_T), \tag{1}$$

where $\Gamma_{\boldsymbol{x}_t} : \mathbb{R}^n \mapsto \mathbb{R}^m$ denotes a set of mapping from state to control space. Then, we have $V_0(\boldsymbol{x}_0) = J^*(\boldsymbol{x}_0)$ be the optimal objective value to OCP. Q_t is often refer to the Bellman objective.

Unfortunately, solving Eq. 1 in high dimension suffers from the Bellman curse of dimensionality. To mitigate the computational, DDP (see Alg. 1) proposes to approximate Q_t with its second order. Given a nominal trajectory $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$, it iteratively optimizes the objective value, where each iteration consists a backward and forward pass. During the backward pass, DDP performs second-order expansion on the Bellman objective Q_t and computes the updates from the following minimization:

$$\delta \boldsymbol{u}_t^*(\delta \boldsymbol{x}_t) = \operatorname*{arg\,min}_{\delta \boldsymbol{u}_t(\delta \boldsymbol{x}_t) \in \Gamma'_{\delta \boldsymbol{x}_t}} \left\{ \frac{1}{2} \begin{bmatrix} \mathbf{1} \\ \delta \boldsymbol{x}_t \\ \delta \boldsymbol{u}_t \end{bmatrix}^{\mathsf{T}} \begin{bmatrix} \mathbf{0} & {Q_{\boldsymbol{x}}^t}^{\mathsf{T}} & {Q_{\boldsymbol{u}}^t}^{\mathsf{T}} \\ Q_{\boldsymbol{x}}^t & Q_{\boldsymbol{x}\boldsymbol{x}}^t & Q_{\boldsymbol{x}\boldsymbol{u}}^t \\ Q_{\boldsymbol{u}}^t & Q_{\boldsymbol{u}\boldsymbol{x}}^t & Q_{\boldsymbol{u}\boldsymbol{u}}^t \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \delta \boldsymbol{x}_t \\ \delta \boldsymbol{u}_t \end{bmatrix} \right\}. \tag{2}$$

The derivatives of Q_t follow standard chain rule. $\Gamma'_{\delta \boldsymbol{x}_t} = \{\mathbf{b}_t + \mathbf{A}_t \delta \boldsymbol{x}_t : \mathbf{b}_t \in \mathbb{R}^m, \mathbf{A}_t \in \mathbb{R}^{m \times n}\}$ denotes the set of all *affine* mapping from $\delta \boldsymbol{x}_t$. The minimizer to Eq. 2 admits a linear form given by

$$\delta \boldsymbol{u}_t^*(\delta \boldsymbol{x}_t) = \boldsymbol{k}_t + \boldsymbol{K}_t \delta \boldsymbol{x}_t \text{ , where } \boldsymbol{k}_t \triangleq -(Q_{\boldsymbol{u}\boldsymbol{u}}^t)^{-1} Q_{\boldsymbol{u}}^t \text{ , } \boldsymbol{K}_t \triangleq -(Q_{\boldsymbol{u}\boldsymbol{u}}^t)^{-1} Q_{\boldsymbol{u}\boldsymbol{x}}^t \text{ ,} \tag{3}$$

Algorithm 1: Differential Dynamic Programming

```
Input: $\bar{\boldsymbol{u}} \triangleq \{\boldsymbol{u}_t\}_{t=0}^{T-1}$, $\bar{\boldsymbol{x}} \triangleq \{\boldsymbol{x}_t\}_{t=0}^{T}$
Set $V_{\boldsymbol{x}}^T = \nabla_{\boldsymbol{x}}\phi$ and $V_{\boldsymbol{xx}}^T = \nabla_{\boldsymbol{x}}^2\phi$
for $t = T-1$ to 0 do
    Compute $\delta\boldsymbol{u}_t^*(\delta\boldsymbol{x}_t)$ using $V_{\boldsymbol{x}}^{t+1}$, $V_{\boldsymbol{xx}}^{t+1}$ (Eq. 2, 3)
    Compute $V_{\boldsymbol{x}}^t$ and $V_{\boldsymbol{xx}}^t$ using Eq. 4
end for
Set $\hat{\boldsymbol{x}}_0 = \boldsymbol{x}_0$
for $t = 0$ to $T-1$ do
    $\boldsymbol{u}_t^* = \boldsymbol{u}_t + \delta\boldsymbol{u}_t^*(\delta\boldsymbol{x}_t)$, where $\delta\boldsymbol{x}_t = \hat{\boldsymbol{x}}_t - \boldsymbol{x}_t$
    $\hat{\boldsymbol{x}}_{t+1} = f_t(\hat{\boldsymbol{x}}_t, \boldsymbol{u}_t^*)$
end for
$\bar{\boldsymbol{u}} \leftarrow \{\boldsymbol{u}_t^*\}_{t=0}^{T-1}$
```

Algorithm 2: Back-propagation with GD

```
Input: $\bar{\boldsymbol{u}} \triangleq \{\boldsymbol{u}_t\}_{t=0}^{T-1}$, $\bar{\boldsymbol{x}} \triangleq \{\boldsymbol{x}_t\}_{t=0}^{T}$,
learning rate $\eta$
Set $\boldsymbol{p}_T \equiv \nabla_{\boldsymbol{x}_T} J_T = \nabla_{\boldsymbol{x}}\phi$
for $t = T-1$ to 0 do
    $\delta\boldsymbol{u}_t^* = -\eta\nabla_{\boldsymbol{u}_t} J_t = -\eta(\ell_{\boldsymbol{u}}^t + {f_{\boldsymbol{u}}^t}^{\mathsf{T}}\boldsymbol{p}_{t+1})$
    $\boldsymbol{p}_t \equiv \nabla_{\boldsymbol{x}_t} J_t = {f_{\boldsymbol{x}}^t}^{\mathsf{T}}\boldsymbol{p}_{t+1}$
end for
for $t = 0$ to $T-1$ do
    $\boldsymbol{u}_t^* = \boldsymbol{u}_t + \delta\boldsymbol{u}_t^*$
end for
$\bar{\boldsymbol{u}} \leftarrow \{\boldsymbol{u}_t^*\}_{t=0}^{T-1}$
```

respectively denote the open and feedback gains. $\delta\boldsymbol{x}_t$ is called the *state differential*, which we will discuss later. Substituting Eq. 3 back to Eq. 2 gives us the backward update for $V_{\boldsymbol{x}}$ and $V_{\boldsymbol{xx}}$,

$$V_{\boldsymbol{x}}^t = Q_{\boldsymbol{x}}^t - {Q_{\boldsymbol{ux}}^t}^{\mathsf{T}}(Q_{\boldsymbol{uu}}^t)^{-1}Q_{\boldsymbol{u}}^t \ , \quad \text{and} \quad V_{\boldsymbol{xx}}^t = Q_{\boldsymbol{xx}}^t - {Q_{\boldsymbol{ux}}^t}^{\mathsf{T}}(Q_{\boldsymbol{uu}}^t)^{-1}Q_{\boldsymbol{ux}}^t \ . \tag{4}$$

In the forward pass, DDP applies the feedback policy sequentially from the initial time step while keeping track of the state differential between the new simulated trajectory and nominal trajectory.

3. Training DNNs as Trajectory Optimization

First, recall that DNNs can be interpreted as dynamical systems where each layer is viewed as a distinct time step. Consider the layer-wise propagation rule, $\boldsymbol{x}_{t+1} = \sigma_t(g_t(\boldsymbol{x}_t, \boldsymbol{u}_t))$, where σ_t and g_t denote the nonlinear activation function and the affine transform parametrized by the vectorized weight $\boldsymbol{u}_t$. $\boldsymbol{x}_t$ represents the activation vector at layer t. Hence, the equation can be seen as a dynamical system (by setting $f_t \equiv \sigma_t \circ g_t$ in OCP) propagating the activation $\boldsymbol{x}_t$ using $\boldsymbol{u}_t$. Next, notice that the gradient descent (GD) update, denoted $\delta\bar{\boldsymbol{u}}^* \equiv -\eta\nabla_{\bar{\boldsymbol{u}}}J$, can be break down into each layer, *i.e.* $\delta\bar{\boldsymbol{u}}^* \triangleq \{\delta\boldsymbol{u}_t^*\}_{t=0}^{T-1}$, and computed backward through a per-layer objective J_t defined as

$$\delta\boldsymbol{u}_t^* = \underset{\delta\boldsymbol{u}_t \in \mathbb{R}^{m_t}}{\arg\min}\{J_t + \nabla_{\boldsymbol{u}_t}J_t^{\mathsf{T}}\delta\boldsymbol{u}_t + \tfrac{1}{2}\delta\boldsymbol{u}_t^{\mathsf{T}}(\tfrac{1}{\eta}\boldsymbol{I}_t)\delta\boldsymbol{u}_t\} \ , \tag{5}$$

$$\text{where } J_t(\boldsymbol{x}_t, \boldsymbol{u}_t) \triangleq \ell_t(\boldsymbol{u}_t) + J_{t+1}(f_t(\boldsymbol{x}_t, \boldsymbol{u}_t), \boldsymbol{u}_{t+1}) \ , \quad J_T(\boldsymbol{x}_T) \triangleq \phi(\boldsymbol{x}_T) \ . \tag{6}$$

We now draw a novel connection between the training procedure of DNNs and trajectory optimization. Let us summarize the Back-propagation with GD in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control $\bar{\boldsymbol{u}}$ that simulates the activation sequence $\bar{\boldsymbol{x}}$. Starting from this nominal trajectory $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$, both algorithms recursively define some layer-wise objectives (J_t in Eq. 6 vs V_t in Eq. 1), compute the weight/control update from the quadratic expansions (Eq. 5 vs Eq. 2), and then carry certain information ($\nabla_{\boldsymbol{x}_t}J_t$ vs $(V_{\boldsymbol{x}}^t, V_{\boldsymbol{xx}}^t)$) backward to the preceding layer. The two computation graphs are summarized in Fig. 1. Below we make this connection formally and provide conditions when the two algorithms become equivalent.

Proposition 2 *Assume $Q_{\boldsymbol{ux}}^t = \boldsymbol{0}$ at all stages, then the backward dynamics of the value derivative can be described by the Back-propagation,* i.e. $\forall t, V_{\boldsymbol{x}}^t = \nabla_{\boldsymbol{x}_t}J$. *Further, we have*

$$Q_{\boldsymbol{u}}^t = \nabla_{\boldsymbol{u}_t}J \ , \quad Q_{\boldsymbol{uu}}^t = \nabla_{\boldsymbol{u}_t}^2 J \ , \quad \textit{and} \quad \delta\boldsymbol{u}_t^*(\delta\boldsymbol{x}_t) = -(\nabla_{\boldsymbol{u}_t}^2 J)^{-1}\nabla_{\boldsymbol{u}_t}J \ . \tag{7}$$

Table 2: Update rule at each layer t, $\boldsymbol{u}_t \leftarrow \boldsymbol{u}_t - \eta \boldsymbol{M}_t^{-1} \boldsymbol{d}_t$. (Expectation taken over batch data)

Methods	Precondition matrix $\boldsymbol{M}_t$	Update direction $\boldsymbol{d}_t$
SGD	$\boldsymbol{I}_t$	$\mathbb{E}[J_{\boldsymbol{u}_t}]$
RMSprop	$\operatorname{diag}(\sqrt{\mathbb{E}[J_{\boldsymbol{u}_t} \odot J_{\boldsymbol{u}_t}]} + \epsilon)$	$\mathbb{E}[J_{\boldsymbol{u}_t}]$
KFAC & EKFAC	$\mathbb{E}[\boldsymbol{x}_t \boldsymbol{x}_t^{\mathsf{T}}] \otimes \mathbb{E}[J_{\boldsymbol{h}_t} J_{\boldsymbol{h}_t}^{\mathsf{T}}]$	$\mathbb{E}[J_{\boldsymbol{u}_t}]$
vanilla DDP	$\mathbb{E}[Q^t_{\boldsymbol{uu}}]$	$\mathbb{E}[Q^t_{\boldsymbol{u}} + Q^t_{\boldsymbol{ux}} \delta \boldsymbol{x}_t]$
DDPNOpt	$\boldsymbol{M}_t \in \left\{ \begin{array}{c} \boldsymbol{I}_t \,, \\ \operatorname{diag}(\sqrt{\mathbb{E}[Q^t_{\boldsymbol{u}} \odot Q^t_{\boldsymbol{u}}]} + \epsilon) \,, \\ \mathbb{E}[\boldsymbol{x}_t \boldsymbol{x}_t^{\mathsf{T}}] \otimes \mathbb{E}[V^t_{\boldsymbol{h}} {V^t_{\boldsymbol{h}}}^{\mathsf{T}}] \end{array} \right\}$	$\mathbb{E}[Q^t_{\boldsymbol{u}} + Q^t_{\boldsymbol{ux}} \delta \boldsymbol{x}_t]$

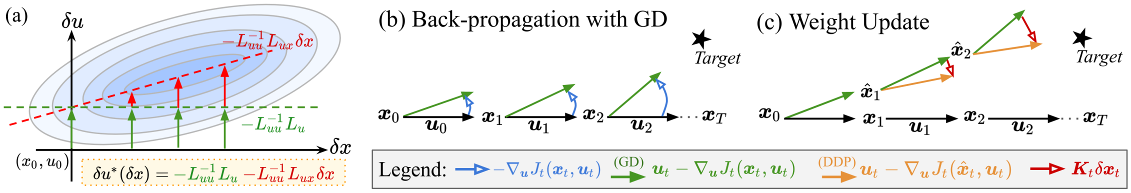

Figure 2: (a) A toy illustration of the standard update (green) and the DDP feedback (red). (bc) Trajectory optimization viewpoint of DNN training.

In other words, the DDP policy is equivalent to the stage-wise Newton, in which the gradient is preconditioned by the block-wise inverse Hessian at each layer. If further we have $Q^t_{\boldsymbol{uu}} \approx \frac{1}{\eta}\mathbf{I}$*, then DDP degenerates to Back-propagation with gradient descent.*

We leave the proof in Appendix A.1. Proposition 2 states that the backward pass in DDP collapses to Back-propagation when $Q_{\boldsymbol{ux}}$ vanishes at all stages. In other words, DDP differs from existing methods in that it expands the layer-wise objective wrt not only $\boldsymbol{u}_t$ but $\boldsymbol{x}_t$. To make some intuitions, consider the example in Fig. 2a. Given an objective L expanded at (x_0, u_0), standard second-order methods apply the update $\delta u = -L_{uu}^{-1} L_u$ (shown as green arrows). DDP differs in that it also computes the *mixed* partial derivatives, *i.e.* L_{ux}. The resulting update law has the same intercept but with an additional feedback term linear in δx (shown as red arrows). Thus, DDP searches for an update from the affine mapping $\Gamma'_{\delta \boldsymbol{x}_t}$ (Eq. 2), rather than the vector space $\mathbb{R}^{m_t}$ (Eq. 5).

Now, to show how the state differential $\delta \boldsymbol{x}_t$ arises during optimization, notice from Alg. 1 that $\hat{\boldsymbol{x}}_t$ can be compactly expressed as $\hat{\boldsymbol{x}}_t = F_t(\boldsymbol{x}_0, \bar{\boldsymbol{u}} + \delta \bar{\boldsymbol{u}}^*(\delta \bar{\boldsymbol{x}}))$[1]. Hence, $\delta \boldsymbol{x}_t = \hat{\boldsymbol{x}}_t - \boldsymbol{x}_t$ captures the state difference when new updates $\delta \bar{\boldsymbol{u}}^*(\delta \bar{\boldsymbol{x}})$ are applied until layer $t-1$. Now, consider the 2D example in Fig 2b. Back-propagation proposes the update directions (shown as blue arrows) from the first-order derivatives expanded along the nominal trajectory $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$. However, as the weight at each layer is correlated, parameter updates from previous layers affect proceeding states, thus the trustworthiness of their descending directions. As shown in Fig 2c, cascading these (green) updates may cause an *over-shoot* wrt the designed target. From the trajectory optimization viewpoint, a much stabler direction will be instead $\nabla_{\boldsymbol{u}_t} J_t(\hat{\boldsymbol{x}}_t, \boldsymbol{u}_t)$ (shown as orange), where the derivative is evaluated at the new cascading state $\hat{\boldsymbol{x}}_t$ rather than $\boldsymbol{x}_t$. This is what DDP proposes, where the feedback $\boldsymbol{K}_t \delta \boldsymbol{x}_t$ compensates the over-shoot and steers the GD update toward $\nabla_{\boldsymbol{u}_t} J_t(\hat{\boldsymbol{x}}_t, \boldsymbol{u}_t)$ after observing $\delta \boldsymbol{x}_t$.

In short, the use of feedback $\boldsymbol{K}_t$ and state differential $\delta \boldsymbol{x}_t$ in DDP to stabilize and robustify the training dynamics arises from the fact that *deep nets exhibit chain structures*. This perspective (*i.e.* optimizing chained parameters) is explored rigorously in trajectory optimization, where DDP is shown to be numerically stabler than direct optimization such as Newton method [13].

---

[1] $F_t \triangleq f_t \circ \cdots \circ f_0$ denotes the compositional dynamics propagating $\boldsymbol{x}_0$ with the control sequence $\{\boldsymbol{u}_s\}_{s=0}^{t}$.

Table 3: Performance comparison on accuracy (%). All values averaged over 10 seeds.

	DataSet	Standard baselines				OCP-inspired baselines		**DDPNOpt (ours)**
		SGD	RMSProp	Adam	EKFAC	E-MSA	vanilla DDP	
Feedforward	DIGITS	**95.36**	94.33	94.98	95.24	94.87	91.68	95.13
	MNIST	92.65	91.89	92.54	92.73	90.24	N/A	**93.30**
	F-MNIST	82.49	83.87	84.36	84.12	82.04		**84.98**
CNN	MNIST	97.66	98.05	98.04	98.02	96.48	N/A	**98.09**
	SVHN	88.05	88.41	87.76	90.63	79.45		**90.70**
	CIFAR-10	68.95	70.52	70.04	71.85	61.42		**71.92**

4. Differential Dynamic Programming Neural Optimizer

Here we present DDP Neural Optimizer and validate its performance on training DNNs. We leave implementation details, pseudo-code, and experiment setup in Appendix A.2 and A.3. DDPNOpt follows the same procedure in vanilla DDP (Alg. 1). However, since f_t is highly over-parametrized, $Q^t_{\boldsymbol{uu}}$ will be computationally intractable to solve. Recall the interpretation we draw in Eq. 6 where GD minimizes the quadratic expansion of J_t with the Hessian $\nabla^2_{\boldsymbol{u}_t} J_t$ replaced by $\frac{1}{\eta}\boldsymbol{I}_t$. Similarly, adaptive first-order (*resp.* second-order methods) can be recovered by approximating $\nabla^2_{\boldsymbol{u}_t} J_t$ with the diagonal of the covariance (*resp.* Gauss-Newton (GN)) matrix. DDPNOpt adapts the same curvature approximation to $Q^t_{\boldsymbol{uu}}$, except these approximations are constructed using (V, Q) rather than J. Table 2 summarizes the update rule for different methods. Bellman framework differs from Back-propagation in computing the update directions $\boldsymbol{d}_t$, where the former applies the feedback through additional forward pass with $\delta\boldsymbol{x}_t$. The connection between these two $\boldsymbol{d}_t$ is built in Proposition 2. Compared with vanilla DDP, DDPNOpt leverages efficient approximation of $\boldsymbol{M}_t$ inspiring by existing methods, which greatly increase the scability.

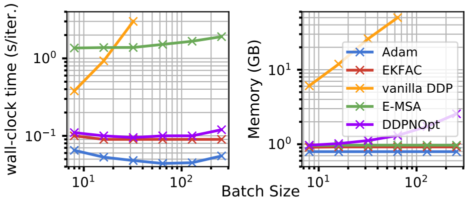

Figure 3: Runtime comparison on MNIST.

Table 3 reports the results on classification datasets. All networks consist of 5-6 layers. For standard training baselines, we select first-order methods, *i.e.* SGD, RMSprop, Adam, and second-order method EKFAC [5], which is a recent extension of KFAC [17]. For OCP-inspired methods, we compare DDPNOpt with vanilla DDP and E-MSA [12], which is also a second-order method yet built upon the PMP framework. It is clear from Table 3 that DDPNOpt outperforms two OCP baselines on *all datasets and network types*. In practice, both baselines suffer from unstable training and require careful tuning on the hyper-parameters. In fact, we are not able to obtain results for vanilla DDP when the problem size goes beyond DIGITS. This is in contrast to DDPNOpt which adapts amortized curvature estimation from widely-used methods; thus exhibits much stabler training dynamics with superior convergence. In Fig 3, we compare the runtime and memory complexity among different methods. While vanilla DDP scales poorly with batch size, DDPNOpt reduces the computation by orders. It runs nearly as fast as standard methods and outperforms E-MSA by a large margin. The additional memory complexity, when comparing DDP-inspired methods with Back-propagation methods, comes from the layer-wise feedback policies. However, DDPNOpt is much memory-efficient compared with vanilla DDP. On the other hand, the performance gain between DDPNOpt and standard methods appear comparatively small. This is due to the inevitable use of similar curvature adaptation, as the local geometry of the landscape directly affects the convergence behavior. In practice, we find DDPNOpt best shows its effect when using larger learning rates (*i.e.* when training becomes unstable). We leave discussions on this ablation analysis in Appendix A.3.2.

5. Conclusion

We introduce DDPNOpt, a new class of optimizer arising from bridging DNN training to trajectory optimization. DDPNOpt features layer-wise feedback policies which improve convergence over existing optimizers and outperforms other OCP-inspired methods in training and scalability.

References

[1] Richard Bellman. The theory of dynamic programming. Technical report, Rand corp santa monica ca, 1954.

[2] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*. Athena scientific Belmont, MA, 1995.

[3] Vladimir Grigor'evich Boltyanskii, Revaz Valer'yanovich Gamkrelidze, and Lev Semenovich Pontryagin. The theory of optimal processes. i. the maximum principle. Technical report, TRW SPACE TECHNOLOGY LABS LOS ANGELES CALIF, 1960.

[4] R Gamkrelidze. *Principles of optimal control theory*, volume 7. Springer Science & Business Media, 2013.

[5] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. In *Advances in Neural Information Processing Systems*, pages 9550–9560, 2018.

[6] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[7] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, pages 15353–15363, 2019.

[8] Tianyu Gu. *Improved trajectory planning for on-road self-driving vehicles via combined graph search, optimization & topology analysis*. PhD thesis, Carnegie Mellon University, 2017.

[9] D.H. Jacobson and D.Q. Mayne. *Differential Dynamic Programming*. Modern analytic and computational methods in science and mathematics. American Elsevier Publishing Company, 1970. URL `https://books.google.com/books?id=tA-oAAAAIAAJ`.

[10] José Lezama, Qiang Qiu, Pablo Musé, and Guillermo Sapiro. Ole: Orthogonal low-rank embedding-a plug and play geometric loss for deep learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8109–8118, 2018.

[11] Qianxiao Li and Shuji Hao. An optimal control approach to deep learning and applications to discrete-weight neural networks. *arXiv preprint arXiv:1803.01299*, 2018.

[12] Qianxiao Li, Long Chen, Cheng Tai, and E Weinan. Maximum principle based algorithms for deep learning. *The Journal of Machine Learning Research*, 18(1):5998–6026, 2017.

[13] Li-zhi Liao and Christine A Shoemaker. Advantages of differential dynamic programming over newton's method for discrete-time optimal control problems. Technical report, Cornell University, 1992.

[14] Guan-Horng Liu and Evangelos A Theodorou. Deep learning theory review: An optimal control and dynamical systems perspective. *arXiv preprint arXiv:1908.10920*, 2019.

[15] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. *arXiv preprint arXiv:1710.10121*, 2017.

[16] Yiping Lu, Zhuohan Li, Di He, Zhiqing Sun, Bin Dong, Tao Qin, Liwei Wang, and Tie-Yan Liu. Understanding and improving transformer from a multi-particle dynamic system point of view. *arXiv preprint arXiv:1906.02762*, 2019.

[17] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417, 2015.

[18] Kamil Nar, Orhan Ocal, S Shankar Sastry, and Kannan Ramchandran. Cross-entropy loss and low-rank features have responsibility for adversarial examples. *arXiv preprint arXiv:1901.08360*, 2019.

[19] Jacob H Seidman, Mahyar Fazlyab, Victor M Preciado, and George J Pappas. Robust deep learning as optimal control: Insights and convergence guarantees. *Proceedings of Machine Learning Research vol xxx*, 1:14, 2020.

[20] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913. IEEE, 2012.

[21] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.

[22] Emanuel Todorov and Weiwei Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 300–306. IEEE, 2005.

[23] E Weinan. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, 2017.

[24] E Weinan, Jiequn Han, and Qianxiao Li. A mean-field optimal control formulation of deep learning. *arXiv preprint arXiv:1807.01083*, 2018.

[25] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

Appendix A.

A.1. Proof of Proposition 2

Proof We first prove the following lemma which connects the backward pass between two frameworks in the degenerate case.

Lemma 3 *Assume* $Q^t_{\boldsymbol{ux}} = \mathbf{0}$ *at all stages, then we have*

$$V^t_{\boldsymbol{x}} = \nabla_{\boldsymbol{x}_t} J \text{ , and } \quad V^t_{\boldsymbol{xx}} = \nabla^2_{\boldsymbol{x}_t} J \text{ , } \quad \forall t \text{ .} \tag{8}$$

Proof It is obvious to see that Eq. 8 holds at $t = T$. Now, assume the relation holds at $t + 1$ and observe that at the time t, the backward passes take the form of

$$\begin{aligned} V^t_{\boldsymbol{x}} &= Q^t_{\boldsymbol{x}} - Q^{t\,\mathsf{T}}_{\boldsymbol{ux}} (Q^t_{\boldsymbol{uu}})^{-1} Q^t_{\boldsymbol{u}} = \ell^t_{\boldsymbol{x}} + {f^t_{\boldsymbol{x}}}^{\mathsf{T}} \nabla_{\boldsymbol{x}_{t+1}} J = \nabla_{\boldsymbol{x}_t} J \text{ ,} \\ V^t_{\boldsymbol{xx}} &= Q^t_{\boldsymbol{xx}} - Q^{t\,\mathsf{T}}_{\boldsymbol{ux}} (Q^t_{\boldsymbol{uu}})^{-1} Q^t_{\boldsymbol{ux}} = \nabla_{\boldsymbol{x}_t} \{ \ell^t_{\boldsymbol{x}} + {f^t_{\boldsymbol{x}}}^{\mathsf{T}} \nabla_{\boldsymbol{x}_{t+1}} J \} = \nabla^2_{\boldsymbol{x}_t} J \text{ ,} \end{aligned}$$

where we recall $J_t = \ell_t + J_{t+1}(f_t)$ in Eq. 6. ■

Now, Eq. 7 follows by substituting Eq. 8 to the definition of $Q^t_{\boldsymbol{u}}$ and $Q^t_{\boldsymbol{uu}}$

$$\begin{aligned} Q^t_{\boldsymbol{u}} &= \ell^t_{\boldsymbol{u}} + {f^t_{\boldsymbol{u}}}^{\mathsf{T}} V^{t+1}_{\boldsymbol{x}} = \ell^t_{\boldsymbol{u}} + {f^t_{\boldsymbol{u}}}^{\mathsf{T}} \nabla_{\boldsymbol{x}_{t+1}} J = \nabla_{\boldsymbol{u}_t} J \text{ ,} \\ Q^t_{\boldsymbol{uu}} &= \ell^t_{\boldsymbol{uu}} + {f^t_{\boldsymbol{u}}}^{\mathsf{T}} V^{t+1}_{\boldsymbol{xx}} f^t_{\boldsymbol{u}} + V^{t+1}_{\boldsymbol{x}} \cdot f^t_{\boldsymbol{uu}} \\ &= \ell^t_{\boldsymbol{uu}} + {f^t_{\boldsymbol{u}}}^{\mathsf{T}} (\nabla^2_{\boldsymbol{x}_{t+1}} J) f^t_{\boldsymbol{u}} + \nabla_{\boldsymbol{x}_{t+1}} J \cdot f^t_{\boldsymbol{uu}} \\ &= \nabla_{\boldsymbol{u}_t} \{ \ell^t_{\boldsymbol{u}} + {f^t_{\boldsymbol{u}}}^{\mathsf{T}} \nabla_{\boldsymbol{x}_{t+1}} J \} = \nabla^2_{\boldsymbol{u}_t} J \text{ .} \end{aligned}$$

Consequently, the DDP feedback policy degenerates to layer-wise Newton update. ■

A.2. DDPNOpt Implementation Details

When the dynamics is represented by the layer propagation (*i.e,* when $f_t \equiv \sigma_t \circ g_t$), we can expand Eq. 2 as:

$$\begin{aligned} Q^t_{\boldsymbol{x}} &= {g^t_{\boldsymbol{x}}}^{\mathsf{T}} V^t_{\boldsymbol{h}} \text{ ,} & Q^t_{\boldsymbol{xx}} &= {g^t_{\boldsymbol{x}}}^{\mathsf{T}} V^t_{\boldsymbol{hh}} g^t_{\boldsymbol{x}} + V^t_{\boldsymbol{h}} \cdot g^t_{\boldsymbol{xx}} \text{ ,} \\ Q^t_{\boldsymbol{u}} &= \ell^t_{\boldsymbol{u}} + {g^t_{\boldsymbol{u}}}^{\mathsf{T}} V^t_{\boldsymbol{h}} \text{ ,} & Q^t_{\boldsymbol{ux}} &= {g^t_{\boldsymbol{u}}}^{\mathsf{T}} V^t_{\boldsymbol{hh}} g^t_{\boldsymbol{x}} + V^t_{\boldsymbol{h}} \cdot g^t_{\boldsymbol{ux}} \text{ ,} \end{aligned} \tag{9}$$

where $V^t_{\boldsymbol{h}} \triangleq {\sigma^t_{\boldsymbol{h}}}^{\mathsf{T}} V^{t+1}_{\boldsymbol{x}}$ and $V^t_{\boldsymbol{hh}} \triangleq {\sigma^t_{\boldsymbol{h}}}^{\mathsf{T}} V^{t+1}_{\boldsymbol{xx}} \sigma^t_{\boldsymbol{h}} + V^{t+1}_{\boldsymbol{x}} \cdot \sigma^t_{\boldsymbol{hh}}$. The matrix-vector product with the Jacobian of the affine transform (*i.e.* $g^t_{\boldsymbol{u}}, g^t_{\boldsymbol{x}}$) can be evaluated efficiently for both feedforward (FF) and convolution (Conv) layers:

$$\boldsymbol{h}_t \overset{\text{FF}}{=} \boldsymbol{W}_t \boldsymbol{x}_t + \boldsymbol{b}_t \Rightarrow {g^t_{\boldsymbol{x}}}^{\mathsf{T}} V^t_{\boldsymbol{h}} = \boldsymbol{W}_t^{\mathsf{T}} V^t_{\boldsymbol{h}} \text{ ,} \quad {g^t_{\boldsymbol{u}}}^{\mathsf{T}} V^t_{\boldsymbol{h}} = \boldsymbol{x}_t \otimes V^t_{\boldsymbol{h}} \text{ ,} \tag{10}$$

$$\boldsymbol{h}_t \overset{\text{Conv}}{=} \omega_t * \boldsymbol{x}_t \qquad \Rightarrow {g^t_{\boldsymbol{x}}}^{\mathsf{T}} V^t_{\boldsymbol{h}} = \omega_t^{\mathsf{T}} \hat{*} V^t_{\boldsymbol{h}} \text{ ,} \quad {g^t_{\boldsymbol{u}}}^{\mathsf{T}} V^t_{\boldsymbol{h}} = \boldsymbol{x}_t \hat{*} V^t_{\boldsymbol{h}} \text{ ,} \tag{11}$$

where $\otimes$, $\hat{*}$, and $*$ respectively denote the Kronecker product and (de-)convolution operator.

When the memory efficiency becomes nonnegligible as the problem scales, we make GN approximation to $\nabla^2 \phi$ as the low-rank structure at the prediction layer has been observed for problems concerned in this work [10, 18]. In the following proposition, we show that for a specific type of OCP, which happens to be the case of DNN training, such a low-rank structure preserves throughout the DDP backward pass.

Proposition 4 (Outer-product factorization in DDPNOpt) *Consider the OCP where $\ell_t \equiv \ell_t(\boldsymbol{u}_t)$ is independent of $\boldsymbol{x}_t$, If the terminal-stage Hessian can be expressed by the outer product of vector $\boldsymbol{z}_{\boldsymbol{x}}^T$, $\nabla^2 \phi(\boldsymbol{x}_T) = \boldsymbol{z}_{\boldsymbol{x}}^T \otimes \boldsymbol{z}_{\boldsymbol{x}}^T$ (for instance, $\boldsymbol{z}_{\boldsymbol{x}}^T = \nabla \phi$ for GN), then we have the factorization for all t:*

$$Q^t_{\boldsymbol{ux}} = \boldsymbol{q}^t_{\boldsymbol{u}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}} \,, \quad Q^t_{\boldsymbol{xx}} = \boldsymbol{q}^t_{\boldsymbol{x}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}} \,, \quad V^t_{\boldsymbol{xx}} = \boldsymbol{z}^t_{\boldsymbol{x}} \otimes \boldsymbol{z}^t_{\boldsymbol{x}} \,. \tag{12}$$

$\boldsymbol{q}^t_{\boldsymbol{u}}$, $\boldsymbol{q}^t_{\boldsymbol{x}}$, and $\boldsymbol{z}^t_{\boldsymbol{x}}$ are outer-product vectors which are also computed along the backward pass.

$$\boldsymbol{q}^t_{\boldsymbol{u}} = {f^t_{\boldsymbol{u}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}} \,, \quad \boldsymbol{q}^t_{\boldsymbol{x}} = {f^t_{\boldsymbol{x}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}} \,, \quad \boldsymbol{z}^t_{\boldsymbol{x}} = \sqrt{1 - {\boldsymbol{q}^t_{\boldsymbol{u}}}^{\mathsf{T}} (Q^t_{\boldsymbol{uu}})^{-1} \boldsymbol{q}^t_{\boldsymbol{u}}} \; \boldsymbol{q}^t_{\boldsymbol{x}} \,. \tag{13}$$

Proof We will prove Proposition 4 by backward induction. Suppose at layer $t+1$, we have $V^{t+1}_{\boldsymbol{xx}} = \boldsymbol{z}^{t+1}_{\boldsymbol{x}} \otimes \boldsymbol{z}^{t+1}_{\boldsymbol{x}}$ and $\ell_t \equiv \ell_t(\boldsymbol{u}_t)$, then Eq. 2 becomes

$$\begin{aligned}
Q^t_{\boldsymbol{xx}} &= {f^t_{\boldsymbol{x}}}^{\mathsf{T}} V^{t+1}_{\boldsymbol{xx}} f^t_{\boldsymbol{x}} = {f^t_{\boldsymbol{x}}}^{\mathsf{T}} (\boldsymbol{z}^{t+1}_{\boldsymbol{x}} \otimes \boldsymbol{z}^{t+1}_{\boldsymbol{x}}) f^t_{\boldsymbol{x}} = ({f^t_{\boldsymbol{x}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}}) \otimes ({f^t_{\boldsymbol{x}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}}) \\
Q^t_{\boldsymbol{ux}} &= {f^t_{\boldsymbol{u}}}^{\mathsf{T}} V^{t+1}_{\boldsymbol{xx}} f^t_{\boldsymbol{x}} = {f^t_{\boldsymbol{u}}}^{\mathsf{T}} (\boldsymbol{z}^{t+1}_{\boldsymbol{x}} \otimes \boldsymbol{z}^{t+1}_{\boldsymbol{x}}) f^t_{\boldsymbol{x}} = ({f^t_{\boldsymbol{u}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}}) \otimes ({f^t_{\boldsymbol{x}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}}) \,.
\end{aligned}$$

Setting $\boldsymbol{q}^t_{\boldsymbol{x}} := {f^t_{\boldsymbol{x}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}}$ and $\boldsymbol{q}^t_{\boldsymbol{u}} := {f^t_{\boldsymbol{u}}}^{\mathsf{T}} \boldsymbol{z}^{t+1}_{\boldsymbol{x}}$ will give the first part of Proposition 4.

Next, to show the same factorization structure preserves through the preceding layer, it is sufficient to show $V^t_{\boldsymbol{xx}} = \boldsymbol{z}^t_{\boldsymbol{x}} \otimes \boldsymbol{z}^t_{\boldsymbol{x}}$ for some vector $\boldsymbol{z}^t_{\boldsymbol{x}}$. This is indeed the case.

$$\begin{aligned}
V^t_{\boldsymbol{xx}} &= Q^t_{\boldsymbol{xx}} - {Q^t_{\boldsymbol{ux}}}^{\mathsf{T}} (Q^t_{\boldsymbol{uu}})^{-1} Q^t_{\boldsymbol{ux}} \\
&= \boldsymbol{q}^t_{\boldsymbol{x}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}} - (\boldsymbol{q}^t_{\boldsymbol{u}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}})^{\mathsf{T}} (Q^t_{\boldsymbol{uu}})^{-1} (\boldsymbol{q}^t_{\boldsymbol{u}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}}) \\
&= \boldsymbol{q}^t_{\boldsymbol{x}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}} - ({\boldsymbol{q}^t_{\boldsymbol{u}}}^{\mathsf{T}} (Q^t_{\boldsymbol{uu}})^{-1} \boldsymbol{q}^t_{\boldsymbol{u}}) (\boldsymbol{q}^t_{\boldsymbol{x}} \otimes \boldsymbol{q}^t_{\boldsymbol{x}}) \,,
\end{aligned}$$

where the last equality follows by observing ${\boldsymbol{q}^t_{\boldsymbol{u}}}^{\mathsf{T}} (Q^t_{\boldsymbol{uu}})^{-1} \boldsymbol{q}^t_{\boldsymbol{u}}$ is a scalar.

Set $\boldsymbol{z}^t_{\boldsymbol{x}} = \sqrt{1 - {\boldsymbol{q}^t_{\boldsymbol{u}}}^{\mathsf{T}} (Q^t_{\boldsymbol{uu}})^{-1} \boldsymbol{q}^t_{\boldsymbol{u}}} \; \boldsymbol{q}^t_{\boldsymbol{x}}$ will give the desired factorization.

■

In other words, the outer-product factorization at the final layer can be backward propagated to all proceeding layers. Thus, large matrices, such as $Q^t_{\boldsymbol{ux}}$, $Q^t_{\boldsymbol{xx}}$, $V^t_{\boldsymbol{xx}}$, and even feedback policies $\boldsymbol{K}_t$, can be factorized accordingly, reducing complexity by orders.

Lastly, we apply Tikhonov regularization on the value Hessian $V^t_{\boldsymbol{xx}}$ [20], which improves the stability when the dimension of the activation varies along the DDP backward pass. We also expand the dynamics only up to first order as the stability obtained by keeping only the linearized dynamics is thoroughly discussed and widely adapted in practical DDP usages [20, 22].

Below we provide the pseudo-code for DDPNOPt.

Algorithm 3: Differential Dynamic Programming Neural Optimizer (DDPNOPt)

```
Input: dataset $\mathcal{D}$, learning rate $\eta$, training iteration $K$, (optional) Tikhonov regularization $\epsilon_V$
Initialize the network weights (i.e. nominal control trajectory) $\bar{\boldsymbol{u}}^{(0)}$
for $k = 0$ to $K$ do
    Sample batch initial state from dataset , $\boldsymbol{X}_0 \equiv \{\boldsymbol{x}_0^{(i)}\}_{i=1}^{B} \sim \mathcal{D}$
    for $t = 0$ to $T - 1$ do
        $\boldsymbol{x}_{t+1}^{(i)} = f_t(\boldsymbol{x}_t^{(i)}, \boldsymbol{u}_t^{(k)}), \forall i$    ▷ Forward simulation
    end for
    Set $V^T_{\boldsymbol{x}^{(i)}} = \nabla_{\boldsymbol{x}} \Phi(\boldsymbol{x}_T^{(i)})$ and $V^T_{\boldsymbol{xx}^{(i)}} = \nabla^2_{\boldsymbol{x}^{(i)}} \Phi(\boldsymbol{x}_T^{(i)}), \quad \forall i$
    for $t = T - 1$ to $0$ do
        Compute $Q^t_{\boldsymbol{u}}$, $Q^t_{\boldsymbol{x}}$, $Q^t_{\boldsymbol{xx}}$, $Q^t_{\boldsymbol{ux}}$ with Eq. 9 (or Eq. 12 if factorization is used), $\forall i$
        Compute $\mathbb{E}[Q^t_{\boldsymbol{uu}}]$ with one of the precondition matrices in Table 2    ▷ Backward pass
        Store the layer-wise feedback policy $\delta \boldsymbol{u}_t^*(\delta \boldsymbol{X}_t) = \frac{1}{B} \sum_{i=1}^{B} \boldsymbol{k}_t^{(i)} + \boldsymbol{K}_t^{(i)} \delta \boldsymbol{x}_t^{(i)}$
        Compute $V^t_{\boldsymbol{x}^{(i)}}$ and $V^t_{\boldsymbol{xx}^{(i)}}$ with Eq. 4 (or Eq. 13 if factorization is used), $\forall i$
        $V^t_{\boldsymbol{xx}^{(i)}} \leftarrow V^t_{\boldsymbol{xx}^{(i)}} + \epsilon_V \boldsymbol{I}$ if Tikhonov regularization is preferable, $\forall i$
    end for
    Set $\hat{\boldsymbol{x}}_0^{(i)} = \boldsymbol{x}_0^{(i)}, \quad \forall i$
    for $t = 0$ to $T - 1$ do
        $\boldsymbol{u}_t^* = \boldsymbol{u}_t + \delta \boldsymbol{u}_t^*(\delta \boldsymbol{X}_t)$, where $\delta \boldsymbol{X}_t = \{\hat{\boldsymbol{x}}_t^{(i)} - \boldsymbol{x}_t^{(i)}\}_{i=1}^{B}$    ▷ Additional forward pass
        $\hat{\boldsymbol{x}}_{t+1}^{(i)} = f_t(\hat{\boldsymbol{x}}_t^{(i)}, \boldsymbol{u}_t^*), \quad \forall i$
    end for
    $\bar{\boldsymbol{u}}^{(k+1)} \leftarrow \{\boldsymbol{u}_t^*\}_{t=0}^{T-1}$
end for
```

A.3. Experiment Detail

A.3.1. SETUP

All networks in the classification experiments are composed of 5-6 layers. For the intermediate layers, we use ReLU activation on all dataset, except Tanh on WINE and DIGITS to better distinguish the differences between optimizers. We use identity mapping at the last prediction layer on all dataset except WINE, where we use sigmoid instead to help distinguish the performance among optimizers. For feedforward networks, the dimension of the hidden state is set to 10-32. On the other hand, we use standard 3 × 3 convolution kernels for all CNNs. The batch size is set 8-32 for dataset trained with feedforward networks, and 128 for dataset trained with convolution networks. For each baseline we select its own hyper-parameter from an appropriate search space, which we detail in Table 4. We use the implementation in https://github.com/Thrandis/EKFAC-pytorch for EKFAC and implement our own E-MSA in PyTorch since the official code released from Li et al. [12] does not support GPU implementation. Regarding the curvature approximation used in DDPNOpt ($\boldsymbol{M}_t$ in Table 2), we found

Table 4: Hyper-parameter search in Table 3

Methods	Learning Rate
SGD	(7e-2, 5e-1)
Adam & RMSprop	(7e-4, 1e-2)
EKFAC	(1e-2, 3e-1)

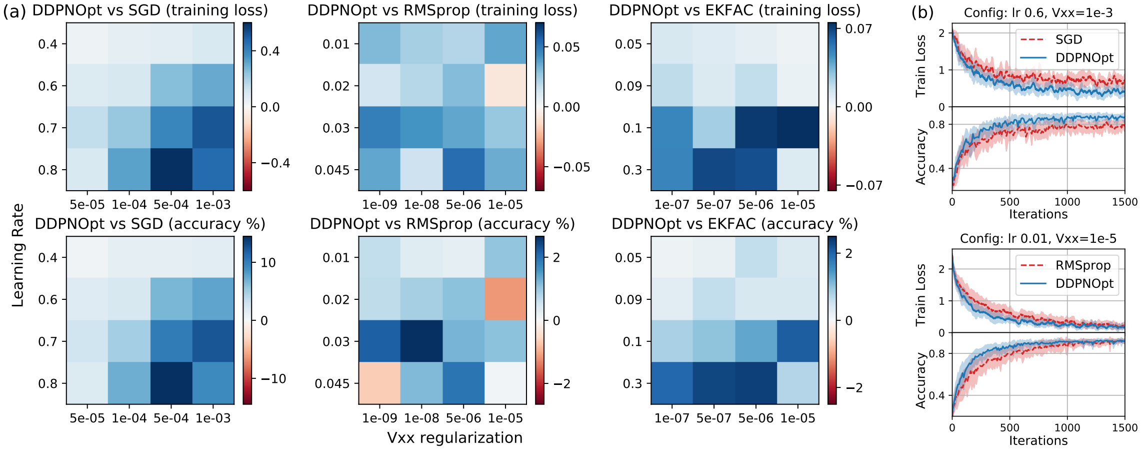

Figure 4: (a) Performance difference between DDPNOpt and baselines on DIGITS across hyper-parameter grid. Blue (*resp.* red) indicates an improvement (*resp.* degradation) over baselines. We observe similar behaviors on other datasets. (b) Examples of the actual training dynamics.

that using adaptive diagonal and GN matrices respectively for feedforward and convolution networks give the best performance in practice. We impose the GN factorization presented in Proposition 4 for all CNN training. Regarding the machine information, we conduct our experiments on GTX 1080 TI, RTX TITAN, and four Tesla V100 SXM2 16GB.

A.3.2. ABLATION ANALYSIS ON FEEDBACK POLICIES

To identify scenarios where DDPNOpt best shows its effectiveness, we conduct an ablation analysis on the feedback mechanism. This is done by recalling Proposition 2: when $Q^t_{\boldsymbol{ux}}$ vanishes, DDPNOpt degenerates to the method associated with each precondition matrix. For instance, DDPNOpt with identity (*resp.* adaptive diagonal and GN) precondition $\boldsymbol{M}_t$ will generate the same updates as SGD (*resp.* RMSprop and EKFAC) when all $Q^t_{\boldsymbol{ux}}$ are zeroed out. In other words, these DDPNOpt variants can be viewed as the *DDP-extension* to existing baselines.

In Fig. 4a we report the performance difference between each baseline and its associated DDPNOpt variant. Each grid corresponds to a distinct training configuration that is averaged over 10 random trails, and we keep all hyper-parameters (*e.g.* learning rate and weight decay) the same between baselines and their DDPNOpt variants. Thus, the performance gap only comes from the feedback policies, or equivalently the update directions in Table 2. Blue (*resp.* red) indicates an improvement (*resp.* degradation) when the feedback policies are presented. Clearly, the improvement over baselines remains consistent across most hyper-parameters setups, and the performance gap tends to become obvious as the learning rate increases. This aligns with the previous study on numerical stability [13], which suggests the feedback can robustify unstable dynamics when a further step size, *i.e.* a larger control, is taken. As shown in Fig. 4b, such a stabilization can also lead to smaller variance and faster convergence. This sheds light on the benefit gained by bridging two seemly disconnected methodologies between DNN training and trajectory optimization.

A.3.3. DISCUSSION: VISUALIZATION OF FEEDBACK POLICIES

To understand the effect of feedback policies more perceptually, we conduct eigen-decomposition on the feedback matrices of convolution layers and project the leading eigenvectors back to image

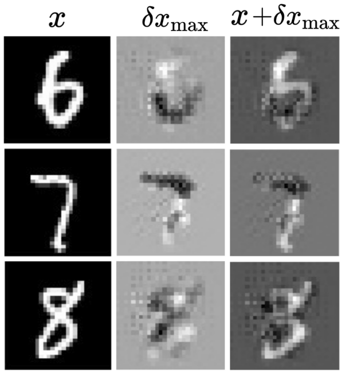

Figure 5: Visualization of the feedback policies on MNIST.

space, following Zeiler and Fergus [25]. These feature maps, denoted $\delta x_{\max}$ in Fig. 5, correspond to the dominating differential image that the policy shall respond with during weight update. Fig. 5 shows that the feedback policies indeed capture non-trivial visual features related to the pixel-wise difference between spatially similar classes, *e.g.* $(8,3)$ or $(7,1)$. These differential maps differ from adversarial perturbation [6] as the former directly links the parameter update to the change in activation; thus being more interpretable.