DDPNOpt: Differential Dynamic Programming Neural Optimizer

Anonymous authors
Paper under double-blind review

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 reformulating the training procedure from the trajectory optimization perspective. We first show that most widely-used algorithms for training DNNs can be linked to the Differential Dynamic Programming (DDP), a celebrated second-order method rooted in the Approximate Dynamic Programming. In this vein, we propose a new class of optimizer, DDP Neural Optimizer (DDPNOpt), for training feedforward and convolution networks. DDPNOpt features layer-wise feedback policies which improve convergence and reduce sensitivity to hyper-parameter over existing methods. 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. We also observe DDPNOpt has surprising benefit in preventing gradient vanishing. 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:

$$\min_{\bar{u}} J(\bar{u}; x_0) := \left[ \phi(x_T) + \sum_{t=0}^{T-1} \ell_t(x_t, u_t) \right]$$

s.t. $$x_{t+1} = f_t(x_t, u_t)$$, (OCP)

where $$x_t \in \mathbb{R}^n$$ and $$u_t \in \mathbb{R}^m$$ 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, $$\bar{u} \triangleq \{u_t\}_{t=0}^{T-1}$$, such that the accumulated cost $$J$$ over the finite horizon $$t \in \{0, 1, \cdots, T\}$$ is minimized. Problems with the form of OCP appear in multidisciplinary areas since it describes a generic multi-stage decision making problem (Gamkrelidze, 2013), and have gained commensurate interest recently in deep learning (Weinan, 2017; Liu & Theodorou, 2019).

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 (Weinan, 2017). The dynamical system perspective provides a mathematically-sound explanation for existing DNN models (Lu et al., 2019). It also leads to new architectures inspired by numerical differential equations and physics (Lu et al., 2017; Chen et al., 2018; Greydanus et al., 2019). In this vein, one may interpret the training as the parameter identification (PI) of nonlinear dynamics. However, PI typically involves (i) searching time-independent parameters (ii) given trajectory measurements at each time step (Voss et al., 2004; Peifer & Timmer, 2007). Neither setup holds in practical DNNs training, which instead optimizes time- (i.e. layer-) varying parameters given the target measurements only at the final stage.

An alternative perspective, which often leads to a richer analysis, is to recast network weights as control variables. Through this lens, OCP describes w.l.o.g. the training objective composed of layer-wise loss (e.g. weight decay) and terminal loss (e.g. cross-entropy). This perspective (see Table 1) has been explored recently to provide theoretical statements for convergence and generalization (Weinan et al., 2018; Seidman et al., 2020). On the algorithmic side, while OCP has motivated new architectures (Benning et al., 2019) and methods for breaking sequential computation (Gunther et al., 2020; Zhang et al., 2019), OCP-inspired optimizers remain relatively limited, often restricted to either specific network class (e.g. discrete weight) (Li & Hao, 2018) or small-size dataset (Li et al., 2017).
The aforementioned works are primarily inspired by the Pontryagin Maximum Principle (PMP; Boltyanskii et al. (1960)), which characterizes the first-order optimality conditions to OCP. Another parallel methodology which receives little attention is the Approximate Dynamic Programming (ADP; Bertsekas et al. (1995)). Despite both originate from the optimal control theory, ADP differs from PMP in that at each time step a locally optimal feedback policy (as a function of state $x_t$) is computed. These policies, as opposed to the vector update from PMP, are known to enhance the numerical stability of the optimization process when models admit chain structures (e.g. in DNNs) (Liao & Shoemaker 1992; Tassa et al. 2012). Practical ADP algorithms such as the Differential Dynamic Programming (DDP; Jacobson & Mayne (1970)) appear extensively in modern autonomous systems for complex trajectory optimization (Tassa et al. 2014; Gu 2017). However, whether they can be lifted to large-scale stochastic optimization, as in the DNN training, remains unclear.

In this work, we make a significant advance toward optimal-control-theoretic training algorithms inspired by ADP. We first show that most existing first- and second-order optimizers can be derived from DDP as special cases. Built upon this intriguing connection, we present a new class of optimizer which marries the best of both. The proposed method, DDP Neural Optimizer (DDPNOpt), features layer-wise feedback policies, as we will show through experiments, improve convergence and robustness. To enable efficient training, DDPNOpt adapts several key components from different methodologies, including (i) adaptive curvature estimation from existing methods, (ii) stabilization techniques used in trajectory optimization, and (iii) an efficient factorization to OCP. These lift the complexity by orders of magnitude compared with other OCP-inspired baselines, without sacrificing the performance. In summary, we present the following contributions.

- We draw a novel perspective of DNN training from the trajectory optimization viewpoint, based on a theoretical connection between existing training methods and the DDP algorithm.
- We 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 against unstable hyperparameter (e.g. large learning rate) setups.
- We show that DDPNOpt achieves competitive performance against existing training methods on classification datasets and outperforms previous OCP-inspired methods in both training performance and runtime complexity. We also identify DDPNOpt can mitigate vanishing gradient.

2 Preliminaries

We will start with the Bellman principle to OCP and leave discussions on PMP in Appendix A.1.

**Theorem 1** (Dynamic Programming (DP) (Bellman 1954)). 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

$$V_t(x_t) = \min_{u_t(x_t) \in \Gamma_{x_t}} \left\{ \ell_t(x_t, u_t) + V_{t+1}(f_t(x_t, u_t)) \right\}_{Q_t(x_t, u_t) \equiv Q_t},$$

where $\Gamma_{x_t} : \mathbb{R}^n \mapsto \mathbb{R}^m$ denotes a set of mapping from state to control space. Then, we have $V_0(x_0) = J^*(x_0)$ be the optimal objective value to OCP. Further, let $\mu_t^*(x_t) \in \Gamma_{x_t}$ be the minimizer of Eq. 1 for each $t$, then the policy $\pi^* = \{\mu_t^*(x_t)\}_{t=0}^{T-1}$ is globally optimal in the closed-loop system.

Hereafter we refer $Q_t(x_t, u_t)$ to the Bellman objective. The Bellman principle recasts minimization over a control sequence to a sequence of minimization over each control. The value function $V_t$ summarizes the optimal cost-to-go at each stage, provided all afterward stages also being minimized.

**Notation:** We will always use $t$ as the time step of dynamics and denote a subsequence trajectory until time $s$ as $\check{x}_t \triangleq \{x_t\}_{t=0}^s$, with $\check{x}_t \triangleq \{x_t\}_{t=0}^T$ as the whole. For any real-valued time-dependent function $F_t$ and a given state-control pair, $x_t \in \mathbb{R}^n$ and $u_t \in \mathbb{R}^m$, we denote the derivatives of $F_t$ evaluated on $(x_t, u_t)$ as, e.g. $\nabla_x F_t \in \mathbb{R}^n \nabla_{x_t} F_t \in \mathbb{R}^n \times \mathbb{R}^m$ and $\nabla_{x_t u_t} F_t \in \mathbb{R}^m$. For notational simplicity, we will sometime abbreviate them respectively as $F_t^x$, $F_t^{x_t}$, and $F_t^{x_t u_t}$.
We note that all derivatives appearing in Eq. 3 are evaluated at the time step. Consider keeping track of the state differential between the new simulated trajectory and the nominal trajectory. Eq. 2 gives us the backward update for the nominal trajectory where the second order approximation remains valid. Substituting Eq. 4 back to play an important role later in our analysis. Note that this policy is only optimal locally around the linear form given by

\[ \delta u^*_t(\delta x_t) = \arg \min_{\delta u_t(\delta x_t) \in \Gamma^*} \left\{ \frac{1}{2} \delta x_t^T \begin{bmatrix} 1 & 0 \\ \delta u_t & \delta u_t \end{bmatrix}^T \begin{bmatrix} 0 & Q^T_t \\ Q_t & 0 \end{bmatrix} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \frac{1}{2} \delta u_t^T \begin{bmatrix} 1 \\ \delta u_t \end{bmatrix} \right\} , \]

where

\[ Q^T_{xx} = \ell^T_x f^T_x + \ell^T_x V^{t+1}_{xx} f^T_x + V^{t+1}_{xx} \cdot f^T_{xx} , \quad Q^T_{uu} = \ell^T_u f^T_u + \ell^T_u V^{t+1}_{uu} f^T_u + V^{t+1}_{uu} \cdot f^T_{uu} , \quad Q^T_{ux} = \ell^T_x f^T_u + \ell^T_x V^{t+1}_{ux} f^T_u + V^{t+1}_{ux} \cdot f^T_{ux} , \]

We note that all derivatives appearing in Eq. 3 are evaluated at \((x_t, u_t)\), i.e. the state-control pair at time \(t\) among the nominal trajectory. The derivatives of \(Q_t\) follow standard chain rule and the dot notation represents the product of a vector with a 3D tensor. This \(T_{\delta x_t} = \{b_t + A_t \delta x_t : b_t \in \mathbb{R}^m, A_t \in \mathbb{R}^{m \times m}\}\) denotes the space of all affine mapping from \(\delta x_t\). The analytic solution to Eq. 2 admits a linear form given by

\[ \delta u^*_t(\delta x_t) = k_t + K_t \delta x_t , \]

where \(k_t \triangleq -(Q^T_{uu})^{-1} Q^T_{u} \) and \(K_t \triangleq -(Q^T_{uu})^{-1} Q^{T}_{ux} \)

(4)

denote the open and feedback gains, respectively. \(\delta x_t\) is called the state differential, which will play an important role later in our analysis. Note that this policy is only optimal locally around the nominal trajectory where the second order approximation remains valid. Substituting Eq. 4 back to Eq. 2 gives us the backward update for \(V_{xx}\) and \(V_{ux}\).

\[ V^T_{xx} = Q^T_{xx} - Q^T_{ux} (Q^T_{uu})^{-1} Q^T_{u} , \quad \text{and} \quad V^T_{ux} = Q^T_{ux} - Q^T_{ux} (Q^T_{uu})^{-1} Q^T_{ux} . \]

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 the nominal trajectory.

### 3 Differential Dynamic Programming Neural Optimizer

#### 3.1 Training DNNs as Trajectory Optimization

Recall that DNNs can be interpreted as dynamical systems where each layer is viewed as a distinct time step. Consider e.g. the propagation rule in feedforward layers,

\[ x_{t+1} = \sigma_t(h_t) , \quad h_t = g_t(x_t, u_t) = W_t x_t + b_t . \]

The vector-tensor product is a contraction mapping on the dimension of the vector space, e.g. \(V_x \cdot f^T_{xx} = \sum_{i=1}^n V^{(i)}_x f^{(i)}_{xx}\), where \(V^{(i)}_x\) is the element of the vector \(V_x\) and \(f^{(i)}_{xx}\) is the Hessian of the corresponding element.
We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]

We now draw a novel connection between the training procedure of DNNs and DDP. Let us first summarize the Back-propagation (BP) with gradient descent in Alg. 2 and compare it with DDP (Alg. 1). At each training iteration, we treat the current weight as the control \( \bar{u} \) conditioned by the block-wise inverse Hessian at stage-wise Newton, in which the gradient is pre-computed using Gauss-Newton (GN) approximation:

\[
\nabla^2_u f_t \approx \mathbb{E}[\nabla^2 u_t f_t] = \mathbb{E}[(x_t \otimes J_{h_t})(x_t \otimes J_{h_t})^T] \approx \mathbb{E}[\nabla^2_x t] \otimes \mathbb{E}[\nabla^2_h t].
\]
Table 2: Update rule at each layer \( t, u_t \leftarrow u_t - \eta M_t^{-1} d_t \). (Expectation taken over batch data)

<table>
<thead>
<tr>
<th>Methods</th>
<th>Precondition matrix ( M_t )</th>
<th>Update direction ( d_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>( I_t )</td>
<td>( \mathbb{E}[J_{u_t}] )</td>
</tr>
<tr>
<td>RMSprop</td>
<td>( \text{diag}(\sqrt{\mathbb{E}[J_{u_t} + J_{u_t}^T]} + \epsilon) ) &amp; ( \mathbb{E}[J_{u_t}] )</td>
<td></td>
</tr>
<tr>
<td>KFAC &amp; EKFAC</td>
<td>( \mathbb{E}[x_t x_t^T] \otimes \mathbb{E}[J_{u_t}, J_{u_t}^T] ) &amp; ( \mathbb{E}[Q_{u_t} + Q_{u_t}^T \delta x_t] )</td>
<td></td>
</tr>
<tr>
<td>vanilla DDP</td>
<td>( Q_{u_t} ) &amp; ( \mathbb{E}[Q_{u_t} + Q_{u_t}^T \delta x_t] )</td>
<td></td>
</tr>
<tr>
<td>DDPNOpt</td>
<td>( M_t \in \left{ \text{diag}(\sqrt{\mathbb{E}[Q_{u_t} + Q_{u_t}^T]} + \epsilon), \right. \mathbb{E}[x_t x_t^T] \otimes \mathbb{E}[V^T_{u_t} V^T_{u_t}] } \right} \mathbb{E}[Q_{u_t} + Q_{u_t}^T \delta x_t]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 summarizes the difference in curvature approximation \( (\text{Curvature approximation}). \)

**Proposition 3**

Scales, we make GN approximation to Outer-product factorization. degenerate to the optimizer it adapts for curvature approximation whenever all \( \mathbf{2} \) consider different layer-wise objectives. As a direct implication from Proposition \( \mathbf{3} \), DDPNOpt will adapt the same curvature approximation to \( Q_{u_t} \). For instance, we can approximate \( Q_{u_t} \) simply with an identity matrix \( I_t \), adaptive diagonal matrix \( \text{diag}(\sqrt{\mathbb{E}[Q_{u_t} + Q_{u_t}^T]}), \) or the GN matrix:

\[
Q_{u_t} \approx \mathbb{E}[Q_{u_t} Q_{u_t}^T] = \mathbb{E}[\mathbb{E}[x_t \otimes V_h^T v_h^T] \otimes \mathbb{E}[V^T_{u_t} V^T_{u_t}]].
\] (15)

Curvature approximation. Next, since DNNs are highly over-parametrized models, \( u_t \) (i.e. the layer weight) will be in high-dimensional space. This makes \( Q_{u_t} \) and \( (Q_{u_t})^{-1} \) computationally intractable to solve; thus requires approximation. Recall the interpretation we draw in Eq. \( \mathbf{8} \) where existing optimizers differ in approximating the Hessian \( \nabla^2 f \) to \( \hat{\nabla} \phi \) for GN), then we have the factorization for all \( Q_{u_t} \).

**Outer-product factorization.** 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 (Nar et al., 2019; Lezama et al., 2013). 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 persists throughout the DDP backward pass.

**Proposition 3** (Outer-product factorization in DDPNOpt). Consider the OCP where \( \ell_t \equiv \ell_t(u_t) \) is independent of \( x_t \). If the terminal-stage Hessian can be expressed by the outer product of vector \( z^T \), \( \nabla^2 \phi(x_T) = z^T \otimes z^T \) (for instance, \( z^T = \nabla \phi \) for GN), then we have the factorization for all \( t \):

\[
Q_{u_t} = q_{u_t} \otimes q_{u_t}^T, \quad Q_{x_t} = q_{x_t} \otimes q_{x_t}^T, \quad V_{x_t} = z_{x_t} \otimes z_{x_t}^T.
\] (16)

\( q_{u_t}, q_{x_t}, \) and \( z_{x_t} \) are outer-product vectors which are also computed along the backward pass.

\[
q_{u_t} = q_{u_t}^T z_{x_t} + 1, \quad q_{x_t} = q_{x_t}^T z_{x_t}^T + 1, \quad z_{x_t} = \sqrt{1 - q_{u_t}^T (Q_{u_t})^{-1} q_{u_t}} q_{x_t}^T.
\] (17)

The derivation is left in Appendix A. In other words, the outer-product factorization at the final layer can be backward propagated to all proceeding layers. Thus, large matrices, such as \( Q_{u_t}, Q_{x_t}, \) \( V_{x_t} \), and even feedback policies \( K_t \) can be factorized accordingly, greatly reducing the complexity.
As shown in Table 2, the two frameworks differ in computing the update directions. Algorithm 3 much stabler direction will be instead may cause an.

Next, to show how the state differential \( \delta x_t \) (Eq. 7). The resulting update law has the same intercept but with an additional feedback term linear in \( \delta x \) (shown as red arrows). Thus, DDPNOpt searches for an update from the affine mapping \( \Gamma'_{xx} \) (Eq. 2), rather than the vector space \( \mathbb{R}^m \) (Eq. 1).

Next, to show how the state differential \( \delta x_t \) arises during optimization, notice from Alg. 1 that \( \hat{x}_t \) can be compactly expressed as \( \hat{x}_t = F_t(x_0, \hat{u} + \delta \hat{u}^t(\delta \hat{x})) \) Therefore, \( \delta x_t = \hat{x}_t - x_t \) captures the state difference when new updates \( \delta \hat{u}^t(\delta \hat{x}) \) are applied until layer \( t - 1 \). Now, consider the 2D example in Fig. 2a. Back-propagation proposes the update directions (shown as blue arrows) from the first-order derivatives expanded along the nominal trajectory \( (\hat{x}, \hat{u}) \). However, as the weight at each layer is correlated, parameter updates from previous layers \( \delta \hat{u}_{s}^t \) affect proceeding states \( \{x_t : t > s\} \), 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 perspective, a much stabler direction will be instead \( \nabla_{\hat{u}_t} J_t(\hat{x}_t, \hat{u}_t) \) (shown as orange), where the derivative is

\[ F_t = f_1 \circ \cdots \circ f_0 \] denotes the compositional dynamics propagating \( x_0 \) with the control sequence \( \{u_t\}_{t=0}^T \).
We note that computing layer-wise value Hessians with only first-order expansion on the dynamics training results. Table 3 presents the results over (Eq. 12) resembles the computation in Gauss-Newton method (Botev et al., 2017). For other control-

Thus, the feedback direction compensates the over-shoot by steering the GD update toward the state $x_t$. This is exactly what DDPNOpt proposes, as we can derive the relation (see Appendix A.3),

$$K_t \delta x_t \approx \arg \min_{\delta u_t(\delta x_t) \in \Gamma'_{x_t}} \| \nabla_{u_t} J(\hat{x}_t, u_t + \delta u_t(\delta x_t)) - \nabla_{u_t} J(x_t, u_t) \|. \quad (18)$$

Thus, the feedback direction compensates the over-shoot by steering the GD update toward $\nabla_{u_t} J(\hat{x}_t, u_t)$ after observing $\delta x_t$. The difference between $\nabla_{u_t} J(\hat{x}_t, u_t)$ and $\nabla_{u_t} J(x_t, u_t)$ cannot be neglected especially during early training when the loss landscape contains nontrivial curvature everywhere (Alain et al., 2019). In short, the use of feedback $K_t$ and state differential $\delta x_t$ arises from the fact that deep nets exhibit chain structures. DDPNOpt feedback policies thus have a stabilization effect on robustifying the training dynamics against e.g. improper hyper-parameters which may cause unstable training. 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 (Liao & Shoemaker, 1992).

Remarks on other optimizers. Our discussions so far rigorously explore the connection between DDP and stage/layer-wise Newton, thus include many popular second-order training methods. General Newton method coincides with DDP only for linear dynamics (Murray & Yakowitz, 1984), despite both share the same convergence rate when the dynamics is fully expanded to second order. We note that computing layer-wise value Hessians with only first-order expansion on the dynamics (Eq. 12) resembles the computation in Gauss-Newton method (Botev et al., 2017). For other control-theoretic methods, e.g. PID optimizers (An et al., 2018), they mostly consider the dynamics over training iterations. DDPNOpt instead focuses on the dynamics inherited in the DNN architecture.

5 Experiments

5.1 Performance on Classification Dataset

Networks & Baselines Setup. We first validate the performance of training fully-connected (FCN) and convolution networks (CNN) using DDPNOpt on classification datasets. FCN consists of 5 fully-connected layers with the hidden dimension ranging from 10 to 32, depending on the size of the dataset. CNN consists of 4 convolution layers (with $3 \times 3$ kernel, 32 channels), followed by 2 fully-connected layers. We use ReLU activation on all datasets except Tanh for WINE and DIGITS to better distinguish the differences between optimizer. The batch size is set to 8-32 for FCNs-trained datasets and 128 for datasets trained with CNN. As DDPNOpt combines strengths from both standard training methods and OCP framework, we select baselines from both sides. This includes first-order methods, i.e. SGD (with tuned momentum), RMSprop, Adam, and second-order method EKFA (George et al., 2018), which is a recent extension of the popular KFAC (Martens & Grosse, 2015). For OCP-inspired methods, we compare DDPNOpt with vanilla DDP and E-MSA (Li et al., 2017), which is also a second-order method yet built upon the PMP framework. Regarding the curvature approximation used in DDPNOpt ($M_t$ in Table 2), we found that using adaptive diagonal and GN matrices respectively for FCNs and CNNs give the best performance in practice. We leave the complete experiment setup and additional results in Appendix A.6.

Training Results. Table 3 presents the results over 10 random trials. It is clear 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 with any reasonable amount of computational resources when the problem size goes beyond FC networks. This is in contrast to DDPNOpt which adapts amortized
To understand the effect of feedback policies more perceptually, in Fig. 3 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 [Liao & Shoemaker 1992], which suggests the feedback can stabilize the optimization when e.g. larger control updates are taken. Since larger control corresponds to a further step size in the application of DNN training, one should expect DDPNOpt to show its robustness as the learning rate increases. 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 seemingly disconnected methodologies between DNN training and trajectory optimization.

5.2 DISCUSSION ON FEEDBACK POLICIES

Visualization of Feedback Policies. To understand the effect of feedback policies more perceptually, in Fig. 5 we visualize the feedback policy when training CNNs. This is done by first conducting curvature estimation from widely-used methods; thus exhibits much stabler training dynamics with superior convergence. In Table 4 we provide the analytic runtime and memory complexity among different methods. While vanilla DDP grows cubic wrt BX, DDPNOpt reduces the computation by orders of magnitude with efficient approximation presented in Sec. 3. As a result, when measuring the actual computational performance with GPU parallelism, DDPNOpt 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 by exploiting the factorization in Proposition 3.

Ablation Analysis. On the other hand, the performance gain between DDPNOpt and standard methods appear comparatively small. We conjecture this is due to the inevitable use of similar curvature adaptation, as the local geometry of the landscape directly affects the convergence behavior. 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 \( Q_{t,x}^i \) vanishes, DDPNOpt degenerates to the method associated with each precondition matrix. For instance, DDPNOpt with identity (resp. adaptive diagonal and GN) precondition \( M_i \) will generate the same updates as SGD (resp. RMSprop and EKFAC) when all \( Q_{t,x}^i \) are zeroed out. In other words, these DDPNOpt variants can be viewed as the DDP-extension to existing baselines.

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

<table>
<thead>
<tr>
<th>DataSet</th>
<th>SGD-m</th>
<th>RMSProp</th>
<th>Adam</th>
<th>EKFAC</th>
<th>OCP-inspired baselines</th>
<th>E-MSA</th>
<th>vanilla DDP</th>
<th>DDPNOpt (ours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINE</td>
<td>94.35</td>
<td>98.10</td>
<td>98.13</td>
<td>94.60</td>
<td></td>
<td></td>
<td></td>
<td>98.18</td>
</tr>
<tr>
<td>DIGITS</td>
<td>95.36</td>
<td>94.33</td>
<td>94.98</td>
<td>95.24</td>
<td></td>
<td></td>
<td></td>
<td>95.13</td>
</tr>
<tr>
<td>MNIST</td>
<td>92.65</td>
<td>91.89</td>
<td>92.54</td>
<td>92.73</td>
<td></td>
<td></td>
<td></td>
<td>93.30</td>
</tr>
<tr>
<td>F-MNIST</td>
<td>82.84</td>
<td>83.87</td>
<td>84.36</td>
<td>84.12</td>
<td></td>
<td></td>
<td></td>
<td>84.98</td>
</tr>
</tbody>
</table>

Table 4: Computational complexity in backward pass. (\( B \): batch size, \( X \): hidden state dim., \( L \): # of layers)

<table>
<thead>
<tr>
<th>Method</th>
<th>Adam</th>
<th>Vanilla DDP</th>
<th>DDPNOpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>( O(X^2 L) )</td>
<td>( O(BX^2 L) )</td>
<td>( O(X^2 L + BX) )</td>
</tr>
<tr>
<td>Speed</td>
<td>( O(BX^2 L) )</td>
<td>( O(B^3 X^2 L) )</td>
<td>( O(BX^2 L) )</td>
</tr>
</tbody>
</table>

Figure 3: Runtime comparison on MNIST.
As shown in Fig. 6b, the update norm from DDPNOpt is typically 5-10 times larger. We note that in this experiment, we replace the cross-entropy (CE) with Max-Mahalanobis center (MMC) loss, a new basis, by having either unbounded activation function or residual blocks, DDPNOpt provides an alternative from the algorithmic perspective.

**Vanishing Gradient.** Lastly, we present an interesting finding on how the feedback policies help mitigate vanishing gradient (VG), a notorious effect when DNNs become impossible to train as gradients vanish along Back-propagation. Fig. [3] reports results on training a sigmoid-activated DNN on DIGITS using MMC loss. DDPNOpt2nd denotes when the layer dynamics is fully expanded to the second order.

**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.

**Figure 5:** Visualization of the feedback policies on MNIST. These differential maps differ from adversarial perturbation (Goodfellow et al., 2014) as the former directly links the parameter update to the change in activation, thus being more interpretable.

**Figure 6:** Training a 9-layer sigmoid-activated FCN on DIGITS using MMC loss. DDPNOpt2nd denotes when the layer dynamics is fully expanded to the second order.

**6 Conclusion**

In this work, we introduce DDPNOpt, a new class of optimizer arising from a novel perspective by bridging DNN training to optimal control and trajectory optimization. DDPNOpt features layer-wise feedback policies which improve convergence and robustness to hyper-parameters over existing optimizers. It outperforms other OCP-inspired methods in both training performance and scalability. This work provides a new algorithmic insight and bridges between deep learning and optimal control.
REFERENCES


A Appendix

A.1 Connection between Pontryagin Maximum Principle and DNNs Training

Development of the optimality conditions to OCP can be dated back to 1960s, characterized by both the Pontryagin’s Maximum Principle (PMP) and the Dynamic Programming (DP). Here we review Theorem of PMP and its connection to training DNNs.

**Theorem 4** (Discrete-time PMP [Pontryagin et al., (1962)].) Let \( \bar{u}^* \) be the optimal control trajectory for OCP and \( \bar{x}^* \) be the corresponding state trajectory. Then, there exists a co-state trajectory \( \bar{p}^* \equiv \{ p_t^* \}_{t=1}^T \), such that

\[
\begin{align*}
\bar{x}_{t+1}^* &= \nabla_x \bar{H}_t (\bar{x}_t^*, \bar{p}_{t+1}^*, \bar{u}_t^*) \cdot \bar{x}_0^* = \bar{x}_0, \\
\bar{p}_t^* &= \nabla_x \bar{H}_t (\bar{x}_t^*, \bar{p}_{t+1}^*, \bar{u}_t^*) \cdot \bar{p}^*_T = \nabla_x \bar{\phi}(\bar{x}_T^*) \ . \\
\bar{u}_t^* &= \arg\min_{v \in \mathbb{R}^m} \bar{H}_t (\bar{x}_t^*, \bar{p}_{t+1}^*, v) .
\end{align*}
\]

where \( \bar{H}_t : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R} \) is the discrete-time Hamiltonian given by

\[
\bar{H}_t (\bar{x}_t, \bar{p}_{t+1}, \bar{u}_t) \equiv \ell_t(\bar{x}_t, \bar{u}_t) + \bar{p}_{t+1}^T f_t(\bar{x}_t, \bar{u}_t) ,
\]

and Eq. (19b) is called the adjoint equation.

The discrete-time PMP theorem can be derived using KKT conditions, in which the co-state \( \bar{p}_t \) is equivalent to the Lagrange multiplier. Note that the solution to Eq. (19b) admits an open-loop process in the sense that it does not depend on state variables. This is in contrast to the Dynamic Programming principle, in which a feedback policy is considered.

It is natural to ask whether the necessary condition in the PMP theorem relates to first-order optimization methods in DNN training. This is indeed the case as pointed out in Li et al. (2017):

**Lemma 5** (Li et al. (2017)). Back-propagation satisfies Eq. (19b) and gradient descent iteratively solves Eq. (19c).

Lemma 5 follows by first expanding the derivative of Hamiltonian wrt \( x_t \),

\[
\nabla_{x_t} \bar{H}_t (x_t, p_{t+1}, u_t) = \nabla_{x_t} \ell_t (x_t, u_t) + \nabla_{x_t} f_t (x_t, u_t)^T p_{t+1} = \nabla_{x_t} J (\bar{u}; x_0) .
\]

Thus, Eq. (19b) is simply the chain rule used in the Back-propagation. When \( \bar{H}_t \) is differentiable wrt \( u_t \), one can attempt to solve Eq. (19c) by iteratively taking the gradient descent. This will lead to

\[
\bar{u}_{t+1}^{(k+1)} = \bar{u}_t^{(k)} - \eta \nabla_{u_t} \bar{H}_t (x_t, p_{t+1}, u_t) = \bar{u}_t^{(k)} - \eta \nabla_{u_t} J (\bar{u}; x_0) ,
\]

where \( k \) and \( \eta \) denote the update iteration and step size. Thus, existing optimization methods can be interpreted as iterative processes to match the PMP optimality conditions.

Inspired from Lemma 5, Li et al. (2017) proposed a PMP-inspired method, named Extended Method of Successive Approximations (E-MSA), which solves the following augmented Hamiltonian

\[
\begin{align*}
\tilde{H}_t (x_t, p_{t+1}, u_t, x_{t+1}, p_t) &\equiv \bar{H}_t (x_t, p_{t+1}, u_t) \\
&+ \frac{1}{2} \rho \| x_{t+1} - f_t (x_t, u_t) \| + \frac{1}{2} \rho \| p_t - \nabla_{x_t} H_t \| .
\end{align*}
\]

\( \tilde{H}_t \) is the original Hamiltonian augmented with the feasibility constraints on both forward states and backward co-states. E-MSA solves the minimization

\[
\bar{u}_t^* = \arg\min_{u_t \in \mathbb{R}^m} \tilde{H}_t (x_t, p_{t+1}, u_t, x_{t+1}, p_t)
\]

with L-BFGS per layer and per training iteration. As a result, we consider E-MSA also as second-order method.

A.2 PROOF OF PROPOSITION 2

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

Lemma 6. Assume \( Q_{ux}^t = 0 \) at all stages, then we have
\[
V_{x}^t = \nabla_{x_{1:t}} J, \quad \text{and} \quad V_{xx}^t = \nabla_{x_{1:t}}^2 J, \quad \forall t. \tag{25}
\]

Proof. It is obvious to see that Eq. (25) 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
\[
V_{x}^t = Q_{x}^t - Q_{ux}^t (Q_{uu}^t)^{-1} Q_{u}^t = \ell_{x}^t + f_{x}^T \nabla_{x_{t+1}} J = \nabla_{x_{1:t}} J, \\
V_{xx}^t = Q_{xx}^t - Q_{ux}^t (Q_{uu}^t)^{-1} Q_{u}^t = \nabla_{x_{1:t}} \{ \ell_{x}^t + f_{x}^T \nabla_{x_{t+1}} J \} = \nabla_{x_{1:t}}^2 J,
\]
where we recall \( J_t = \ell_t + J_{t+1} (f_t) \) in Eq. (8).

Now, Eq. (11) follows by substituting Eq. (25) to the definition of \( Q_{u}^t \) and \( Q_{uu}^t \)
\[
Q_{u}^t = \ell_{u}^t + f_{u}^T V_{x}^{t+1} = \ell_{u}^t + f_{u}^T \nabla_{x_{t+1}} J = \nabla_{u_{1:t}} J, \\
Q_{uu}^t = \ell_{u}^t + f_{u}^T V_{x}^{t+1} f_{u}^T + V_{x}^{t+1} f_{u}^T. f_{u}^T \\
= \ell_{u}^t + f_{u}^T (\nabla_{x_{t+1}}^2 J) f_{u}^T + \nabla_{x_{t+1}} J \cdot f_{u}^T \\
= \nabla_{u_{1:t}} \{ \ell_{u}^t + f_{u}^T \nabla_{x_{t+1}} J \} = \nabla_{u_{1:t}}^2 J.
\]

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

A.3 DERIVATION OF EQ. 18

Eq. (18) follows by an observation that the feedback policy \( K_t \delta x_t = - (Q_{uu}^t)^{-1} Q_{ux}^t \delta x_t \) stands as the minimizer of the following objective
\[
K_t \delta x_t = \arg \min_{\delta u_t(\delta x_t) \in \Gamma'(\delta x_t)} \| \nabla_{u_{1:t}} Q(x_t + \delta x_{1:t}, u_t + \delta u_t(\delta x_t)) - \nabla_{u_{1:t}} Q(x_t, u_t) \|, \tag{26}
\]
where \( \Gamma'(\delta x_t) \) denotes all affine mappings from \( \delta x_t \) to \( \delta u_t \) and \( \| \cdot \| \) can be any proper norm in the Euclidean space. Eq. (26) follows by the Taylor expansion of \( Q(x_t + \delta x_{1:t}, u_t + \delta u_t) \) to its first order
\[
\nabla_{u_{1:t}} Q(x_t + \delta x_{1:t}, u_t + \delta u_t) = \nabla_{u_{1:t}} Q(x_t, u_t) + Q_{ux}^t \delta x_t + Q_{uu}^t \delta u_t.
\]

When \( Q = J \), we will arrive at Eq. (18). From Proposition 2, we know the equality holds when all \( Q_{ux}^s \) vanish for \( s > t \). In other words, the approximation in Eq. (18) becomes equality when all afterward layer-wise objectives \( s > t \) are expanded only wrt \( u_s \).

A.4 PROOF OF PROPOSITION 3

Proof. We will prove Proposition 3 by backward induction. Suppose at layer \( t + 1 \), we have
\[
V_{xx}^{t+1} = z_{x}^{t+1} \otimes z_{x}^{t+1} \quad \text{and} \quad \ell_t = \ell_t(u_t), \quad \text{then Eq. (3) becomes}
\]
\[
Q_{xx}^t = f_{x}^T V_{xx}^{t+1} f_{x}^T = f_{x}^T (z_{x}^{t+1} \otimes z_{x}^{t+1}) f_{x}^T = (f_{x}^T z_{x}^{t+1}) \otimes (f_{x}^T z_{x}^{t+1}) \\
Q_{ux}^t = f_{u}^T V_{xx}^{t+1} f_{x}^T = f_{u}^T (z_{x}^{t+1} \otimes z_{x}^{t+1}) f_{x}^T = (f_{u}^T z_{x}^{t+1}) \otimes (f_{u}^T z_{x}^{t+1}).
\]

Setting \( q_{x}^t := f_{x}^T z_{x}^{t+1} \) and \( q_{u}^t := f_{u}^T z_{x}^{t+1} \) will give the first part of Proposition 3.

Next, to show the same factorization structure preserves through the preceding layer, it is sufficient to show \( V_{xx}^t = z_{x}^t \otimes z_{x}^t \) for some vector \( z_{x}^t \). This is indeed the case.
\[
V_{xx}^t = Q_{xx}^t - Q_{ux}^t (Q_{uu}^t)^{-1} Q_{u}^t \\
= q_{x}^t \otimes q_{x}^t - (q_{u}^t \otimes q_{x}^t)^T (Q_{uu}^t)^{-1} (q_{u}^t \otimes q_{x}^t) \\
= q_{x}^t \otimes q_{x}^t - (q_{u}^t (Q_{uu}^t)^{-1} q_{u}^t) (q_{x}^t \otimes q_{x}^t),
\]
where the last equality follows by observing \( q_u^T (Q_{uu})^{-1} q_u \) is a scalar. Set \( z_u^t = \sqrt{1 - q_u^T (Q_{uu})^{-1} q_u} \). \( q_u^t \) will give the desired factorization.

### A.5 Derivation of Eq. [12]

For notational simplicity, we drop the superscript \( t \) and denote \( V_x' = \nabla_x V_{t+1}(x_{t+1}) \) as the derivative of the value function at the next state.

\[
\begin{align*}
Q_u &= \ell_u + f_u^T V_x' = \ell_u + g_u^T \sigma_h V_x', \\
Q_{uu} &= \frac{\partial}{\partial u} (g_u^T \sigma_h V_x') \\
&= \ell_{uu} + g_u^T V_{x' \sigma_h} + g_u V_{x'} \sigma_h g_u + g_u V_{x'} \sigma_h g_u + g_{uu} \sigma_h V_x' \\
&= \ell_{uu} + g_u^T (V_{hh} + V_{x' \sigma_h}) g_u + V_h \cdot g_{uu}
\end{align*}
\]

The last equation follows by recalling \( V_h = \sigma_h V_{x'} \), and \( V_{hh} = \sigma_h V_{x' \sigma_h} \). Follow similar derivation, we have

\[
\begin{align*}
Q_x &= \ell_x + g_x^T V_h \\
Q_{xx} &= \ell_{xx} + g_x^T (V_{hh} + V_{x' \sigma_h}) g_x + V_h \cdot g_{xx} \\
Q_{ux} &= \ell_{ux} + g_x^T (V_{hh} + V_{x' \sigma_h}) g_u + V_h \cdot g_{ux}
\end{align*}
\]

**Remarks:** For feedforward networks, the computational overhead in Eq. [12] and [27] can be mitigated by leveraging its affine structure. Since \( g \) is bilinear in \( x_t \) and \( u_t \), the terms \( g_{ux}^t \) and \( g_{uu}^t \) vanish. The tensor \( g_{ux}^t \) admits a sparse structure, whose computation can be simplified to

\[
\begin{align*}
[g_{ux}^t]_{(i,j,k)} &= 1 \quad \text{iff} \quad j = (k-1)n_t+1+i \\
[V_h^t \cdot g_{ux}^t]_{(k-1)n_t+1:k,n_t+1,k} &= V_{h}^t
\end{align*}
\]

For the coordinate-wise nonlinear transform, \( \sigma_h^t \) and \( \sigma_{hh}^t \) are diagonal matrix and tensor. In most learning instances, stage-wise losses typically involved with weight decay alone; thus the terms \( f_{x'}^t, f_{x''}^t, f_{ux}^t \) also vanish.

### A.6 Experiment Detail

#### A.6.1 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. 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 5. We use the implementation [https://github.com/Thrandis/EKFAC-pytorch](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. (2017)] does not support GPU implementation. We impose the GN factorization presented in Proposition 3 for all CNN training. Regarding the machine information, we conduct our experiments on GTX 1080 Ti, RTX TITAN, and four Tesla V100 SXM2 16GB.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>(7e-2, 5e-1)</td>
</tr>
<tr>
<td>Adam &amp; RMSprop</td>
<td>(7e-4, 1e-2)</td>
</tr>
<tr>
<td>EKFAC</td>
<td>(1e-2, 3e-1)</td>
</tr>
</tbody>
</table>

Table 5: Hyper-parameter search in Table 3.
A.6.2 ADDITIONAL EXPERIMENT AND DISCUSSION

**Batch trajectory optimization on synthetic dataset.** One of the difference between DNN training and trajectory optimization is that for the former, we aim to find an ultimate control law that can drive every data point in the training set, or sampled batch, to its designed target. Despite seemly trivial from the ML perspective, this is a distinct formulation to OCP since the optimal policy typically varies at different initial state. As such, we validate performance of DDPNOpt in batch trajectories optimization on a synthetic dataset, where we sample data from $k \in \{5, 8, 12, 15\}$ Gaussian clusters in $\mathbb{R}^{30}$. Since conceptually a DNN classifier can be thought of as a dynamical system guiding trajectories of samples toward the target regions belong to their classes, we hypothesize that for the DDPNOpt to show its effectiveness on batch training, the feedback policy must act as an ensemble policy that combines the locally optimal policy of each class. Fig. 7 shows the spectrum distribution, sorted in a descending order, of the feedback policy in the prediction layer. The result shows that the number of nontrivial eigenvalues matches exactly the number of classes in each setup (indicated by the vertical dashed line). As the distribution in the prediction layer concentrates to $k$ bulks through training, the eigenvalues also increase, providing stronger feedback to the weight update.

**Effect on Variation Reduction during Training.** Fig. 8 reports additional results on the variance difference during optimization. We use the same setup as in Fig. 4a, i.e. we keep all hyper-parameters the same for each experiment so that the performance difference only comes from the existence of
feedback policies. For most cases, having additional updates from DDP feedback policies stabilizes the training dynamics by reducing its variation over random initialization. Lastly, Fig. 9 reports the same setup for Adam.
A.7 ADDITIONAL CLARIFICATION

A.7.1 PROCEDURE TO GENERATE FIG. 5

Algorithm 4 Visualizing the Feedback Policies

1: Input: Image $x$ (we drop the time subscript for notational simplicity, i.e., $x = x_0$)
2: Perform backward pass of DDPNOpt. Compute $(k_t, K_t)$ backward
3: Perform SVD on $K_t$
4: Extract the right-singular vector corresponding to the largest singular value, denoted $v_{\text{max}} \in \mathbb{R}^n$
5: Project $v_{\text{max}}$ back to the image space using deconvolution procedures introduced in Zeiler & Fergus (2014)

Figure 10: Pictorial illustration for Alg. 4

A.7.2 ADDITIONAL EXPERIMENTS ON VANISHING GRADIENT

Here we provide additional experiments on vanishing gradient. Recall that Fig. 6 reports the training performance using MMC loss on Sigmoid-activated networks. In Fig. 11a, we report the result when training the same networks using CE loss (notice the numerical differences in the $y$ axis for different objectives). None of the presented optimizers were able to escape from vanishing gradient, as evidenced by the vanishing update magnitude. On the other hands, changing the networks to ReLU-activated networks eliminates the vanishing gradient, as shown in Fig. 11b.

Figure 11: Vanishing gradient experiment for different losses and nonlinear activation functions.

Fig. 12 illustrates the selecting process on the learning-rate tuning when we report Fig. 6. For each baseline we draw multiple learning rates from an appropriate search space, which we detail in Table 5. As shown in Fig. 12, the training performance for both SGD-VGR and EKFAC remains unchanged when tuning the learning rate. In practice, we observe unstable training with SGD-VGR when the learning rate goes too large. On the other hands, DDPNOpt and DDPNOpt2nd are able to escape from VG with all tested learning rates. Hence, Fig. 6 combines Fig. 12a (SGD-VGR-lr$^{0.1}$) and Fig. 12c (EKFAC-lr$^{0.03}$, DDPNOpt-lr$^{0.03}$, DDPNOpt2nd-lr$^{0.03}$) for best visualization.
Figure 12: Vanishing gradient experiment for different learning rate setups.

Fig. 13 reports the performance with other first-order adaptive optimizers including Adam and RMSprop. In general, adaptive first-order optimizers are more likely to escape from vanishing gradient since the diagonal precondition matrix (recall $M_t = \mathbb{E}[J_{tu} \odot J_{tu}]$ in Table 2) rescales the vanishing update to a fixed norm. However, as shown in Fig. 13, DDPNOpt* (the variant of DDPNOpt that utilize similar adaptive first-order precondition matrix) converges faster compared with these adaptive baselines.

Figure 13: Vanishing gradient experiment for other optimizers. The legend “DDPNOpt*” denotes DDPNOpt with adaptive diagonal matrix. 

A.7.3 ABLATION ANALYSIS WITH BEST-TUNED BASELINES

Table 6: Learning rate = 0.1

<table>
<thead>
<tr>
<th></th>
<th>SGD</th>
<th>DDPNOpt with $M_t = I_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train Loss</td>
<td>0.035</td>
<td><strong>0.032</strong></td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>95.36</td>
<td><strong>95.52</strong></td>
</tr>
</tbody>
</table>

Table 7: Learning rate = 0.001

<table>
<thead>
<tr>
<th></th>
<th>RMSprop</th>
<th>DDPNOpt with $M_t = \text{diag}(\sqrt{\mathbb{E}[Q_{tu}^T \odot Q_{tu}^T]} + \epsilon)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train Loss</td>
<td>0.058</td>
<td><strong>0.052</strong></td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>94.33</td>
<td><strong>94.63</strong></td>
</tr>
</tbody>
</table>

Table 8: Learning rate = 0.03

<table>
<thead>
<tr>
<th></th>
<th>EKFAC</th>
<th>DDPNOpt with $M_t = \mathbb{E}[x_t x_t^T] \odot \mathbb{E}[V_h^T V_h]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train Loss</td>
<td>0.074</td>
<td><strong>0.067</strong></td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td><strong>95.24</strong></td>
<td>95.19</td>
</tr>
</tbody>
</table>
A.7.4 Numerical absolute values in hyper-parameter grids

Fig. 4b reports the relative performance between each baseline and its DDPNOpt counterpart under different learning rate and regularization setups. In Table 9 and 10 we report the absolute numerical values of this experiment. For instance, the most left-upper grid in Fig. 4a, i.e. the training loss difference between DDPNOpt and SGD with learning rate 0.4 and $V_{\infty}$ regularization $5 \times 10^{-5}$, corresponds to 0.1974 – 0.1662 in Table 9. All values in these tables are averaged over 10 seeds.

Table 9: Training Loss. ($\epsilon_{V_{\infty}}$ denotes the Tikhonov regularization on $V_{\infty}$.)

<table>
<thead>
<tr>
<th>Learn Rate</th>
<th>SGD</th>
<th>DDPNOpt with $M_t = I_t$</th>
<th>$\epsilon_{V_{\infty}} = 5 \times 10^{-5}$</th>
<th>$1 \times 10^{-4}$</th>
<th>$5 \times 10^{-4}$</th>
<th>$1 \times 10^{-3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4</td>
<td>0.1974</td>
<td>0.1662</td>
<td>0.1444</td>
<td>0.1322</td>
<td><strong>0.1067</strong></td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>0.5809</td>
<td>0.4989</td>
<td>0.4867</td>
<td>0.3263</td>
<td><strong>0.2764</strong></td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>1.0493</td>
<td>0.9034</td>
<td>0.8240</td>
<td>0.6592</td>
<td><strong>0.5381</strong></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.7801</td>
<td>1.6898</td>
<td>1.4597</td>
<td>1.1784</td>
<td>1.3166</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Learn Rate</th>
<th>RMSprop</th>
<th>DDPNOpt with $M_t = \text{diag}(\sqrt{E[Q_u \odot Q_u]} + \epsilon)$</th>
<th>$\epsilon_{V_{\infty}} = 1 \times 10^{-9}$</th>
<th>$1 \times 10^{-8}$</th>
<th>$5 \times 10^{-6}$</th>
<th>$1 \times 10^{-5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.1949</td>
<td>0.1638</td>
<td>0.1714</td>
<td>0.1746</td>
<td><strong>0.1588</strong></td>
<td></td>
</tr>
<tr>
<td>0.02</td>
<td>0.4691</td>
<td>0.4559</td>
<td>0.4489</td>
<td><strong>0.4390</strong></td>
<td>0.4773</td>
<td></td>
</tr>
<tr>
<td>0.03</td>
<td>0.8156</td>
<td><strong>0.7675</strong></td>
<td>0.7736</td>
<td>0.7790</td>
<td>0.7983</td>
<td></td>
</tr>
<tr>
<td>0.045</td>
<td>1.3103</td>
<td>1.2740</td>
<td>1.2956</td>
<td><strong>1.2568</strong></td>
<td>1.2578</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Learn Rate</th>
<th>EKFAC</th>
<th>DDPNOpt with $M_t = E[x;x^T] \otimes E[V_{h}^t V_{h}^T]$</th>
<th>$\epsilon_{V_{\infty}} = 1 \times 10^{-7}$</th>
<th>$5 \times 10^{-6}$</th>
<th>$5 \times 10^{-5}$</th>
<th>$1 \times 10^{-5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>0.0757</td>
<td><strong>0.0636</strong></td>
<td>0.0659</td>
<td>0.0691</td>
<td>0.0717</td>
<td></td>
</tr>
<tr>
<td>0.09</td>
<td>0.2274</td>
<td><strong>0.2087</strong></td>
<td>0.2164</td>
<td>0.2091</td>
<td>0.2223</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>0.3260</td>
<td>0.2771</td>
<td>0.3003</td>
<td>0.2543</td>
<td><strong>0.2510</strong></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>0.5959</td>
<td>0.5462</td>
<td><strong>0.5282</strong></td>
<td>0.5299</td>
<td>0.5858</td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Accuracy (%). ($\epsilon_{V_{\infty}}$ denotes the $V_{\infty}$ regularization.)

<table>
<thead>
<tr>
<th>Learn Rate</th>
<th>SGD</th>
<th>DDPNOpt with $M_t = I_t$</th>
<th>$\epsilon_{V_{\infty}} = 5 \times 10^{-5}$</th>
<th>$1 \times 10^{-4}$</th>
<th>$5 \times 10^{-4}$</th>
<th>$1 \times 10^{-3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4</td>
<td>91.46</td>
<td>91.98</td>
<td>92.71</td>
<td>92.90</td>
<td><strong>93.12</strong></td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>81.73</td>
<td>83.64</td>
<td>84.09</td>
<td>88.39</td>
<td><strong>89.39</strong></td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>70.48</td>
<td>73.42</td>
<td>75.44</td>
<td>80.62</td>
<td><strong>82.87</strong></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>55.76</td>
<td>57.70</td>
<td>62.82</td>
<td>65.01</td>
<td><strong>60.74</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Learn Rate</th>
<th>RMSprop</th>
<th>DDPNOpt with $M_t = \text{diag}(\sqrt{E[Q_u \odot Q_u]} + \epsilon)$</th>
<th>$\epsilon_{V_{\infty}} = 1 \times 10^{-9}$</th>
<th>$1 \times 10^{-8}$</th>
<th>$5 \times 10^{-6}$</th>
<th>$1 \times 10^{-5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>91.48</td>
<td>92.14</td>
<td>91.80</td>
<td>91.73</td>
<td><strong>92.52</strong></td>
<td></td>
</tr>
<tr>
<td>0.02</td>
<td>84.15</td>
<td>84.82</td>
<td>85.02</td>
<td><strong>85.23</strong></td>
<td>83.00</td>
<td></td>
</tr>
<tr>
<td>0.03</td>
<td>73.07</td>
<td>75.24</td>
<td><strong>75.73</strong></td>
<td>74.29</td>
<td>74.16</td>
<td></td>
</tr>
<tr>
<td>0.045</td>
<td>59.80</td>
<td>59.16</td>
<td>60.98</td>
<td><strong>61.75</strong></td>
<td>59.87</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Learn Rate</th>
<th>EKFAC</th>
<th>DDPNOpt with $M_t = E[x;x^T] \otimes E[V_{h}^t V_{h}^T]$</th>
<th>$\epsilon_{V_{\infty}} = 1 \times 10^{-7}$</th>
<th>$5 \times 10^{-6}$</th>
<th>$5 \times 10^{-5}$</th>
<th>$1 \times 10^{-5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>93.70</td>
<td>93.84</td>
<td>93.88</td>
<td><strong>94.31</strong></td>
<td>94.06</td>
<td></td>
</tr>
<tr>
<td>0.09</td>
<td>90.84</td>
<td>91.13</td>
<td><strong>91.45</strong></td>
<td>91.23</td>
<td>91.24</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>88.88</td>
<td>89.69</td>
<td>89.89</td>
<td>90.18</td>
<td><strong>90.94</strong></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>81.82</td>
<td>83.79</td>
<td>84.09</td>
<td><strong>84.15</strong></td>
<td>82.55</td>
<td></td>
</tr>
</tbody>
</table>
Table 11: Performance comparison on accuracy (%). All values averaged over 10 seeds.

<table>
<thead>
<tr>
<th>DataSet</th>
<th>SGD-m</th>
<th>Standard baselines</th>
<th>OCP-inspired baselines</th>
<th>DDPNOpt (ours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RMSProp</td>
<td>Adam</td>
<td>EKFAC</td>
</tr>
<tr>
<td>Feed-forward</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WINE</td>
<td>94.35 ± 1.13</td>
<td>98.10 ± 1.14</td>
<td>98.13 ± 0.30</td>
<td>94.60 ± 1.14</td>
</tr>
<tr>
<td>DIGITS</td>
<td>95.36 ± 0.31</td>
<td>94.33 ± 0.47</td>
<td>94.98 ± 0.25</td>
<td>95.24 ± 0.26</td>
</tr>
<tr>
<td>MNIST</td>
<td>92.65 ± 0.07</td>
<td>91.89 ± 0.15</td>
<td>92.54 ± 0.09</td>
<td>92.73 ± 0.12</td>
</tr>
<tr>
<td>F-MNIST</td>
<td>82.49 ± 0.33</td>
<td>83.87 ± 0.16</td>
<td>84.36 ± 0.07</td>
<td>84.12 ± 0.11</td>
</tr>
<tr>
<td>CNN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MNIST</td>
<td>97.94 ± 0.05</td>
<td>98.05 ± 0.03</td>
<td>98.04 ± 0.03</td>
<td>98.02 ± 0.04</td>
</tr>
<tr>
<td>SVHN</td>
<td>89.00 ± 0.13</td>
<td>88.41 ± 0.19</td>
<td>87.76 ± 0.31</td>
<td>90.63 ± 0.03</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>71.26 ± 0.17</td>
<td>70.52 ± 0.37</td>
<td>70.04 ± 0.35</td>
<td>71.85 ± 0.11</td>
</tr>
</tbody>
</table>