Learning to Boost Training by Periodic Nowcasting Near Future Weights

Jinhyeok Jang [1 2] **Woo-han Yun** [1] **Won Hwa Kim** [3] **Youngwoo Yoon** [1]
Jaehong Kim [1] **Jaeyeon Lee** [1] **ByungOk Han** [1]

Abstract

Recent complicated problems require large-scale datasets and complex model architectures, however, it is difficult to train such large networks due to high computational issues. Significant efforts have been made to make the training more efficient such as momentum, learning rate scheduling, weight regularization, and meta-learning. Based on our observations on 1) high correlation between past weights and future weights, 2) conditions for beneficial weight prediction, and 3) feasibility of weight prediction, we propose a more general framework by intermittently skipping a handful of epochs by periodically forecasting near future weights, i.e., a Weight Nowcaster Network (WNN). As an add-on module, WNN predicts the future weights to make the learning process faster regardless of tasks and architectures. Experimental results show that WNN can significantly save actual time cost for training with an additional marginal time to train WNN. We validate the generalization capability of WNN under various tasks, and demonstrate that it works well even for unseen tasks. The code and pre-trained model are available at https://github.com/jjh6297/WNN.

1. Introduction

Since the resurrection of Deep Neural Network (DNN), a variety of Deep Learning models have shown unprecedented improvements over several traditional prediction tasks such as image classification (LeCun et al., 1998; He et al., 2016), natural language processing (Vaswani et al., 2017; Brown et al., 2020), and reinforcement learning (Mnih et al., 2013; Schulman et al., 2017). In order to tackle complicated tasks in these domains, the architectures of DNNs are becoming deeper and more complex to achieve better predictive performance. It is inevitable for such complex models to utilize extremely large-scale datasets together with high computational costs. Thus, efficiency has become a routine but critical issue, especially at the training stage where its complexity exponentially increases according to the size of the model. Considering the practical use of DNN techniques for a broad range of applications, this issue cannot be solved by adding computing power alone.

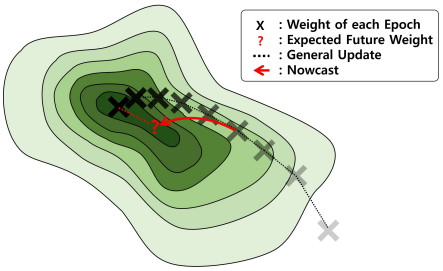


Figure 1. **Illustration of the motivation.** Weight Nowcaster network predicts future values of weight and bias nearby those of real in a DNN architecture. We expected that WNN reduces training time by efficiently skipping future epochs during training process.

Notable efforts have been spent to tackle the computational issue. Architecture minimization (Sandler et al., 2018; Tan & Le, 2019) and efficient physical hardware systems (Jouppi et al., 2017; Khailany et al., 2020) have shown improvements in efficiency. From an algorithmic perspective, diverse training strategies were introduced such as optimizations with advanced momentum (Kingma & Ba, 2014; Loshchilov & Hutter, 2019), scheduling of learning rate (He et al., 2016; Loshchilov & Hutter, 2017), and applying regularizers on trainable parameters in networks (Rodríguez et al., 2017; Jia et al., 2018), which have led to efficient and stable training of DNN models. These algorithmic approaches mainly focus on learning stability; while they accelerate training process to some extent, they do not directly address reducing training time. We also note that these approaches can be used with the proposed WNN.

In this paper, we approach the problem of exhaustive training costs from a different perspective. Specifically, we pro-

[1]ETRI. [2]KAIST. [3]POSTECH. Correspondence to: ByungOk Han <byungok.han@etri.re.kr>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the authors.

pose a Weight Nowcaster Network (WNN), which is a separate DNN that concurrently forecasts updated weights and biases (hereinafter referred to as weights for simplicity) for near future epochs as a sub-module along with a target DNN for the task of interest (*e.g.*, classification). One can easily see that, if WNN can accurately predict future weights based on the previous updates, then those training epochs can be simply skipped. Such an idea was developed based on our observation that changes in loss from various training processes show similar trends in general (Section 3). This means that, even though the training processes of DNNs for different tasks are different, they exhibit a general macroscopic tendency although they may be noisy. Figure 1 briefly illustrates our motivation; if a training process has such a macroscopic pattern, it is possible to jump to a future point in the weight space instead of going through every training epoch. For example, simply predicting one step ahead will easily reduce the total number of epochs required for training in half.

Implementing the ideas described above, we make contributions summarized as follows: **1)** we propose the concept of periodic nowcasting of weights in DNNs and validate the concept by in-depth studies, **2)** we introduce direct regression of weights and a simple plug-in DNN model, WNN, to forecast updated trainable parameters for near future epochs which reduces the total number of epochs to reach model convergence, **3)** we thoroughly validate the superior performance and generalization capability of the proposed WNN with extensive experiments on various tasks, architectures, and datasets. In practice, based on our empirical results to be shown, we saved up to 53% of the training time for image classification tasks with a typical DNN pipeline (*i.e.*, multi-layer perceptron).

2. Related Work

The present paper proposes a meta-learning strategy for accelerating neural network training without compromising performance. In this section, we provide an overview of previous meta-learning techniques for training acceleration in three categories: initialization-level, iteration-level, and entire training-level strategies.

2.1. Initialization-level Acceleration

Initialization-level acceleration learns to choose better initial weights based on target architectures or data. Analytic approaches (Dauphin & Schoenholz, 2019; Zhu et al., 2021) establish task-agnostic criteria for good initialization (*i.e.*, gradient quotient, initial loss), and adjust norm (Dauphin & Schoenholz, 2019) and scaling (Zhu et al., 2021) of the initial weights to meet the criteria through small-scaled optimization before training. Graph HyperNetworks (Knyazev et al., 2021) predict initial weights from model architectures represented in a graph form. A recent approach, using a conditional diffusion transformer (Peebles et al., 2022), adjusts given initial weights by analyzing them with initial loss, but this method was limited to a single architecture and dataset that the transformer was trained on. The initialization-level approach reduces the number of updates required for convergence. However, compared to our work, this approach has a limited acceleration capability (one time acceleration at the initialization step) and poor generalizability (not well applicable to unseen architectures or datasets) and is not compatible with transfer and continual learning that updates initial weights.

2.2. Iteration-level Acceleration

Iteration-level acceleration learns changes in weights along training steps through a separate meta-learning process, and it replaces the conventional analytic optimizers (such as SGD and Adam) with a DNN-based optimizer. This approach is commonly referred to as *Learning to Optimize* (L2O). The early L2O (Andrychowicz et al., 2016) was specific to a target task. To generalize L2O, later works applied random scaling and a convexifying regularizer (Lv et al., 2017), or trained a meta-optimizer on multiple target tasks simultaneously (Wichrowska et al., 2017). Recently, L2O-amalgam (Huang et al., 2022) has generalized a meta-optimizer by distilling multiple teacher optimizers.

2.3. Entire Training-level Acceleration

The most closely related work to our study is Introspection (Sinha et al., 2017), which predicts far future weights based on a long-term training history. It trains a DNN through supervised learning on a set of collected weights from a toy example and applied it to other classification problems. However, Introspection has several limitations: 1) unstable results due to its far-future weight forecasting, 2) the need for a heuristic selection of forecasting period, 3) a limited number of forecasting (only $2 \sim 4$ forecasts in the Introspection experiment), 4) low generalization capability to various tasks, and 5) not performing well with transfer learning from pre-trained weights and learning rate scheduling (*e.g.*, warm-up, decay). We address each of these limitations in the present paper. Between the iteration level and the entire training level, our approach can be positioned, and it is experimentally more stable and better accelerated.

3. Preliminary Study

The concept of forecasting future weight during the learning process was firstly introduced in the Introspection method (Sinha et al., 2017). However, there is a lack of analysis on whether the future weights are related with the past weights, what the beneficial prediction of future weights is, whether it is possible to predict the future weights, and which specific

Table 1. Cosine correlation between future weights of different training sessions. When training DNN models from fixed initial weights, trained future weights have a strong correlation with one another.

Architecture	Cosine Correlation	
	Random Initial Weights	Fixed Initial Weights
Vanilla CNN (Xavier Uniform)	0.0131 ±0.0345	0.9018 ±0.0142
ResNet (He Normal)	0.0594 ±0.1712	0.8959 ±0.0137

strategies are efficient. We first examine the justification of weight prediction through an in-depth pre-study, and introduce our new weight prediction strategy for boosting training speed. Detailed training recipe of each experiment on the preliminary study are described in Appendix. I In this section, we will use the term *weight parameter*, θ, to refer to a single weight parameter that WNN predicts. *Weights*, Θ, are referring a set of parameters.

3.1. The Future is Roughly Predetermined.

Q. Does predictable future exist?

To establish the basis for the weight prediction, we experimentally validate that the target weights (*i.e.*, weights after training) are predictable from the initial weights. We repeatedly trained two architectures from fixed and random initial weights: 1) Vanilla CNN which consists of four convolution layers with [8,16,32,32] channels of 3×3 filters, a flattening layer, a fully-connected layer with 64 hidden neurons, and a classifier and 2) ResNet32 with [16,32,64] channels per each residual block. Then, we examined the correlation among sets of the trained weights at the designated epoch in tens of model training sessions, even with diverse randomness such as batch selection and data augmentation.

As in Table 1, the future weights in different training sessions have strong association with one another when the initial weights are fixed, but not when the initial weights are randomly generated. This result indicates that *the training trajectory of weights is roughly predetermined from the initial weights* even if there is randomness in batch selection or data augmentation. That is, since the direction of training following the initialization of weights is somewhat predetermined, the values of weights during the training process can be utilized as a goal (target) for weight prediction. Details of the analysis are also described in Appendix A.1.

3.2. Beneficial Prediction of Future Weights

To measure quality of predicted weights, we defined three metric using the start weights, Θ_{start}, target weights, Θ_{target}, and predicted weights, $\Theta_{predicted}$. The relative distance represents a distance between Θ_{target} and $\Theta_{predicted}$ normalized by a distance between Θ_{target} and Θ_{start}:

$$Relative\ Distance = \frac{||\theta_{target} - \theta_{predicted}||_2}{||\theta_{target} - \theta_{start}||_2}. \quad (1)$$

The relative direction indicates a cosine correlation between $\theta_{predicted} - \theta_{start}$ and $\theta_{target} - \theta_{start}$ as:

$$Relative\ Direction = \frac{\theta_{predicted} - \theta_{start}}{||\theta_{predicted} - \theta_{start}||_2} \cdot \frac{\theta_{target} - \theta_{start}}{||\theta_{target} - \theta_{start}||_2}, \quad (2)$$

The Relative Required Epochs (RRE) depicts a ratio between the minimal number of epochs required to reach a validation accuracy of Θ_{target} when the network is initialized from $\Theta_{predicted}$ and it when initialized from Θ_{start} as:

$$RRE = \frac{Required\ Epochs\ from\ \Theta_{predicted}}{Required\ Epochs\ from\ \Theta_{start}}, \quad (3)$$

which quantifies training acceleration.

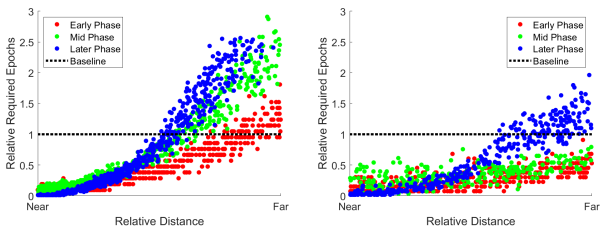
Q. What is the beneficial prediction?

Notice that the relative distance, Eq. (1), and direction, Eq. (2), are defined as notions of error between $\theta_{predicted}$ and θ_{target}. In order to empirically find beneficial prediction of future weights, we conducted weight forecasting simulation experiments with $\Theta_{predicted}$ generated by adding noise to the target Θ_{target} as:

$$\theta_{predicted} \leftarrow \theta_{target} + \epsilon, \quad where \quad \theta \in \Theta \quad (4)$$

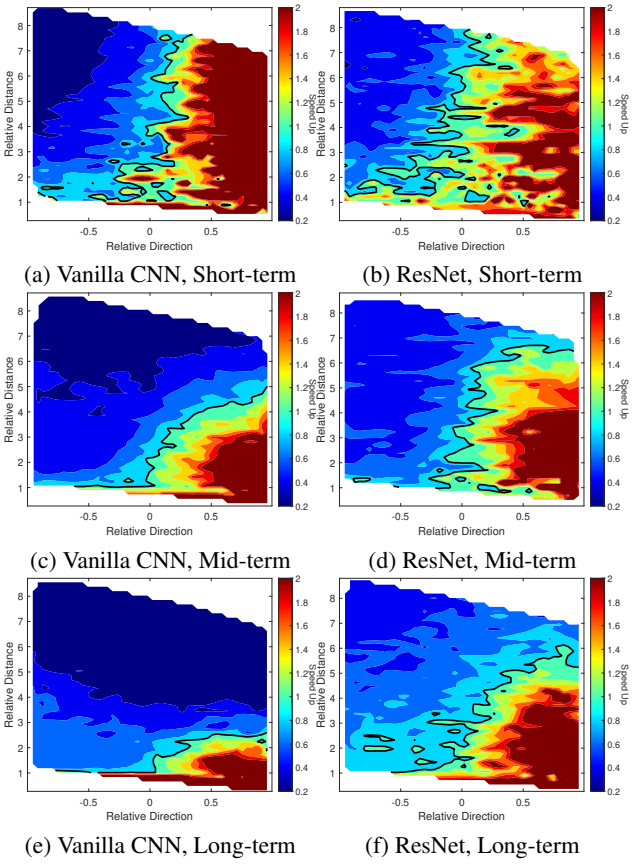
where the Gaussian noise ϵ is generated as explained in Appendix A.2.

Then, we trained the two networks from $\Theta_{predicted}$ as a initial weights, and measured RREs to analyze benefits quantitatively. Figure 2 shows the RREs with a fixed Θ_{start} and varying Θ_{target}; the closer the predicted weights are to the



(a) Vanilla CNN, Xavier Uniform (b) ResNet, He Normal

Figure 2. Experimental results on learning efficiency with virtually generated weights by distance. The learning efficiency is increases as the predicted weights gets nearer the target weights for early (red), mid (green), and later(blue) phases. The distinction between efficiency and inefficiency is represented by the black line.



(a) Vanilla CNN, Short-term (b) ResNet, Short-term

(c) Vanilla CNN, Mid-term (d) ResNet, Mid-term

(e) Vanilla CNN, Long-term (f) ResNet, Long-term

Figure 3. Experimental results on learning efficiency with virtually generated weights by direction and distance. The black line represents the distinction between efficiency and inefficiency, in other words, speed up is 1. The short-term prediction shows the largest area of the efficiency region.

target weights (*i.e.*, low relative distance), the fewer training epochs are required. We also analyzed training efficiency on weight prediction in terms of two factors from the fixed Θ_{target} varying Θ_{start} (Figure 3). Both figures clearly show that *the training becomes faster as the direction and distance of the predicted weights approaches the target weights.*

Q. Can a single prediction model be applied to overall training phases?

We analyzed the results on three different targets Θ_{target} from the early, mid, and later phases of training. As shown from colored results in Figure 2, those trends were consistently found in all early, middle, and later training phases. This indicates that *the amount of training time required can be reduced by being aware of the target weights and choosing new start weights close to it in any training phase.*

Q. What is the best forecasting interval?

On top of that, we performed training experiments with a fixed Θ_{target} and a varying Θ_{start} to identify the best interval to predict future weights. We categorized the forecasting interval into three groups: short-term (from Θ_{t-5} to Θ_t), mid-term (from $\Theta_{t/2}$ to Θ_t), and long-term (from $\Theta_{t/10}$ to Θ_t), where t (20 for ResNet and 50 for Vanilla CNN) means the epoch number of Θ_{target} . The black line in Figure 3 indicates the border of training benefits, and area of beneficial region is the largest in the short-term prediction as described in Table 2. It is clear that *the training benefit is larger the nearer future prediction*. Long-term prediction might fail due to a large weight search space and the nature of non-convexity of weights. We also experimentally observed the trade-off between better prediction and training acceleration from a larger skip (see Table A.5 in Appendix D.3).

Table 2. Quantitative results of Figure 3. The beneficial area represents the size of the region enclosed by the black line.

Setting		Avg. Speed Up	Beneficial Area
Vanilla CNN (Xavier Uniform)	Short-term	$\times$1.7364	0.4488
	Mid-term	$\times$0.9789	0.2470
	Long-term	$\times$0.5761	0.1184
ResNet (He Normal)	Short-term	$\times$1.2368	0.5405
	Mid-term	$\times$1.1083	0.3886
	Long-term	$\times$1.1654	0.3291

3.3. Weight Prediction is Feasible.

In the previous subsection, the prediction benefit for model training was quantitatively validated by adding noise to the target weight without performing the prediction. In this subsection, we use a curve fitting method, a representative method for the regression task, to verify that the future weights are actually predictable and advantageous for model training. Results of other various models of curve fitting to practical training are described in Section 5.2.

Q. Are the future weights predictable?

We conducted experiments on whether the actual predicted weights from the linear curve fitting satisfy the beneficial conditions (speed up > 1; the beneficial region in Figure 3) in terms of the relative distance and direction. In Table 3, the prediction error of curve fitting with short-term and mid-term strategies is described with the degree of training speedup. The training speedup at the prediction error represented as the relative distance and direction is depicted in Figure 3. The curve fitting methods with both strategies meet the beneficial conditions of relative distance and direction. Between the two strategies, the prediction error of mid-term strategy are located at the green-yellow region as described in Figure 3(c) and (d). However, the error of the short-term strategy is located at the red region as described in Figure 3(a) and (b). That is, the short-term prediction accelerates better than the mid-term one. Based on those results, it is clear that *even the most basic curve fitting can forecast future weights, resulting in reducing training time.*

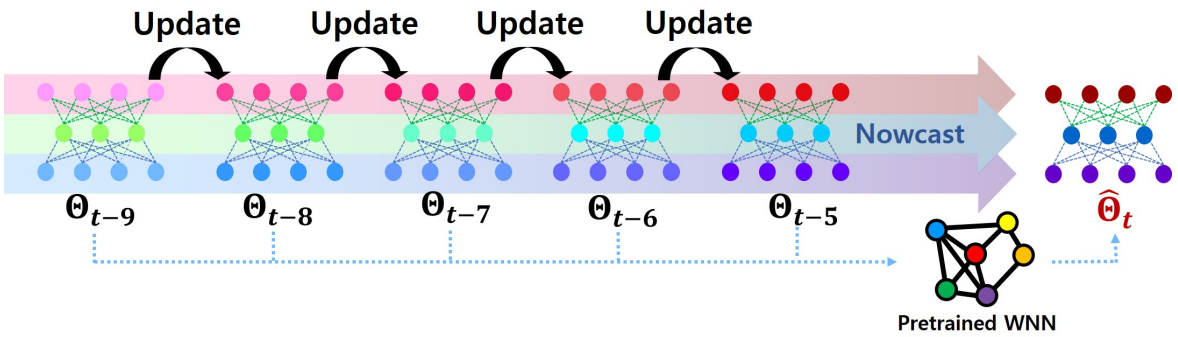


Figure 4. Conceptual view of short-term prediction with the proposed WNN.

Table 3. Experimental results on learning efficiency with several prediction methods. The prediction error is represented by the relative distance and direction. The results indicate that weight prediction is feasible and efficient for model training. Spd.Up. denotes the training speed increase.

Method		Vanilla CNN			ResNet		
		Rel. Dist.	Rel. Dir.	Spd. Up.	Rel. Dist.	Rel. Dir.	Spd. Up.
Curve Fitting (Mid-term)	Conv	2.1282	0.3030	×1.68	1.5422	0.2920	×1.22
	FC	2.1958	0.4317	×1.50	2.1588	0.3962	×1.56
	Bias	3.9395	0.5488	×1.11	N/A	N/A	N/A
	Avg	**2.7545**	**0.4278**	**×1.43**	**1.8505**	**0.3441**	**×1.39**
Curve Fitting (Short-term)	Conv	0.7036	0.7770	×1.42	1.1992	0.4247	×1.18
	FC	1.4660	0.4785	×1.56	0.4280	0.9143	×2.05
	Bias	0.5754	0.8531	×2.05	N/A	N/A	N/A
	Avg	**0.9150**	**0.7028**	**×1.68**	**0.8136**	**0.6695**	**×1.61**
Introspection (Mid-term)	Conv	0.7883	0.6639	×1.70	0.9029	0.4558	×1.16
	FC	0.7049	0.7474	×3.34	0.7016	0.8476	×1.70
	Bias	0.7150	0.6214	×1.70	N/A	N/A	N/A
	Avg	**0.7360**	**0.6775**	**×2.25**	**0.8022**	**0.6517**	**×1.43**
WNN (Short-term)	Conv	1.0311	0.7375	×1.92	0.9849	0.4174	×2.02
	FC	0.7378	0.6663	×2.15	0.7078	0.8331	×2.06
	Bias	0.6990	0.4801	×4.56	N/A	N/A	N/A
	Avg	**0.8226**	**0.6279**	**×2.88**	**0.8256**	**0.6252**	**×2.04**

Q. Can a single prediction model be applied to different types of weights?

As shown in Table 3, the prediction error for weights of convolution layer, fully-connected layer, and bias is distinct. Also, the operations can be differently clustered as shown in Appendix B. That is, each operation has its own tendency. Based on the two observations, *it is better to use separate predictor for each mathematical operation.*

4. Method

4.1. Proposed Strategy: Periodic Short-term Prediction

Based on the observations from the previous section, we established our forecasting strategy: 1) separate forecasting for each mathematical operation, and 2) periodic short-term nowcasting per every 5 epochs as shown in Figure 4. Details are described in Appendix D.2. For more accurate forecasting, we replaced the curve fitting into a DNN model which is a simple neural network-based predictor $\mathcal{F}(\cdot,\cdot)$, called Weight Nowcasting Network (WNN). We designed WNN to predict a short-term update between the future weights and the latest weights by analyzing history of weights in the recent few epochs as illustrated in Figure 5. Inputs of WNN, i.e., weight parameters W_t^o and their temporal differentials dW_t^o, are normalized as W_t and dW_t and divided by two for stable forecasting as

$$W_t^o = [\theta_{t-9}^{l,i,j}, \theta_{t-8}^{l,i,j}, \theta_{t-7}^{l,i,j}, \theta_{t-6}^{l,i,j}, \theta_{t-5}^{l,i,j}]^T, \tag{5}$$

$$dW_t^o = [(\theta_{t-5}^{l,i,j} - \theta_{t-6}^{l,i,j}), ..., (\theta_{t-8}^{l,i,j} - \theta_{t-9}^{l,i,j})]^T, \tag{6}$$

$$W_t = \frac{W_t^o - \theta_{t-5}^{l,i,j}}{2 \cdot (max(W_t^o) - min(W_t^o))}, \tag{7}$$

$$dW_t = \frac{dW_t^o}{2 \cdot (max(W_t^o) - min(W_t^o))}, \tag{8}$$

where l, i, j are the indices of the layer, channel, and spatial coordinate of the target network, and t indicates the number of epoch in training. W_t and dW_t are then fed into our WNN as $\delta = \mathcal{F}(W_t, dW_t)$ which predicts the normalized residual between the latest weight parameter and future parameter.

Then, WNN was trained to minimize ℓ_1 residual error as :

$$\left| \delta - \frac{\theta_t^{l,i,j} - \theta_{t-5}^{l,i,j}}{2 \cdot (max(W_t^o) - min(W_t^o))} \right|_{\ell_1} \tag{9}$$

where the ℓ_1-norm was used to obtain a larger gradient even when the differences are small.

Then, WNN-based prediction can be given as:

$$\hat{\theta}_t^{l,i,j} = \theta_{t-5}^{l,i,j} + 2 \cdot (max(W_t^o) - min(W_t^o)) \cdot \delta \tag{10}$$

skips 5 epochs and directly jumps to the $\theta_t^{l,i,j}$ by updating $\theta_{t-5}^{l,i,j}$ with denormalized δ.

To train WNN, weights in every epoch in various training conditions; architectures, datasets, and augmentation were collected so that the model can be adopted to even unseen cases. More details are to be explained in Section 5.1. We

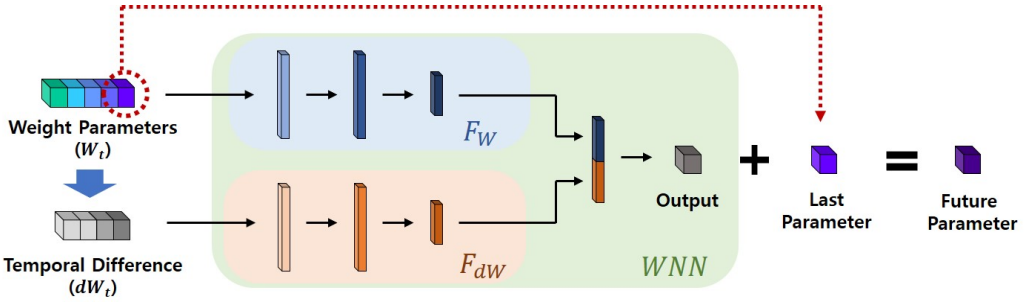


Figure 5. **Weight Nowcaster Network Architecture**. The WNN is composed of simple two-stream networks that use fully-connected layers and an activation network. Feature vectors from those two networks are unified to a feature vector and it is passed through a fully-connected layer. The predicted future weight parameters are obtained by adding outputs and input weight parameters.

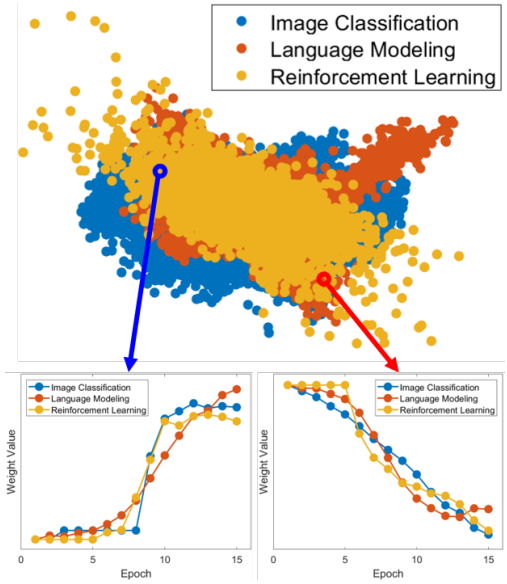


Figure 6. Visualization of histories of weight parameters across different tasks (Image Classification, Language Modeling, Reinforcement Learning). Each dot is a history of changes of a weight parameter for 15 epochs (compressed in 2D).

assumed that each type of weight; convolution weight, fully-connected weight, and bias, have their own characteristics, so individual forecasting networks for each type were separately trained. In Table 3, we report the prediction error and their resulted speed-up for our WNN and the most similar approach to ours, and Introspection. Ablation study about components of our WNN is described in Appendix D.3.

4.2. Weight Change Tendency is Consistent Across Totally Different Tasks.

Most of existing approaches have failed to directly apply their works to a new task without an extra training process (Knyazev et al., 2021; Peebles et al., 2022; Andrychowicz et al., 2016). In this subsection, we analyzed the tendency of element-wise parameter changes on a variety of fundamentally unrelated tasks, including image classification, natural language processing, and reinforcement learning.

Q. Can a pre-trained WNN be applied to unseen task?

We collected weight parameters of the first 15 epochs of training processes from scratch for the three tasks. Then, we decomposed the histories of each parameters of only fully-connected layers as $[\theta_1^{l,i,j}, \theta_2^{l,i,j}, ..., \theta_{15}^{l,i,j}]$ and applied the PCA to reduce dimension of a set of histories for visualizing their tendency. Interestingly, as seen in Figure 6, the individual history of weight parameters formed clusters, *i.e.*, they are not completely heterogeneous in element-wise level despite of different tasks, which suggests that a *training history of parameter changes can be generalized to various tasks*. Then, it sets up the core hypothesis of WNN, *A WNN trained on only image classification can be applicable to a variety of different tasks.* The element-wise trend doesn't consider layer indices, channel indices, or weight spatial coordinates, thus it doesn't indicate all tasks have the same weights. Details of the analysis on weight history are explained in Appendix B.

5. Experiments: Comparison and Analysis

In this section, we performed extensive experiments to testify effectiveness of the proposed method. In Section 5.1, detailed experimental design to construct our WNN is described. Section 5.2 and 5.3 present experimental comparisons with similar approaches on image classification. All the experiments adopted the pre-trained WNN described in this section *without any task-specific fine-tuning*.

5.1. Constructing Weight Nowcaster Network

Collecting Training Data. We trained several target networks multiple times under various training conditions (architectures, datasets, and augmentation methods) and saved weights of every epoch from entire layers for collecting training data. LeNet (LeCun et al., 1998), VGG16 (Simonyan & Zisserman, 2014), ResNet (He et al., 2016), MobileNetV2

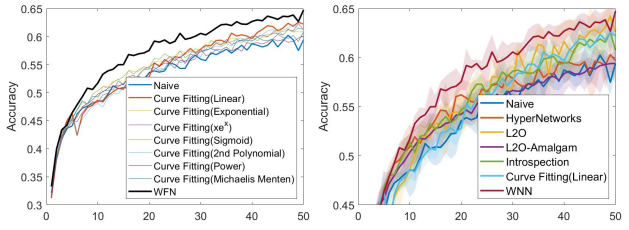
(Sandler et al., 2018), ShuffleNetV2 (Ma et al., 2018), and DenseNet (Huang et al., 2017) were used as the architectures. We assumed that WNN trained on collected weights from image classification task is sufficiently transferable to other tasks. As datasets, we adopted CIFAR10 (Krizhevsky et al., 2009) and MNIST, which are representative datasets in image classification, with various augmentation methods (resize, rotation, noise, data size, and so on). For training WNN, more than 30,000 epochs (∼1.8e+10 parameters) in total under various conditions were collected. For validating WNN, 2,200 epochs of ShuffleNetV2 and ResNet32 on CIFAR10 and MNIST were collected. We empirically chose that WNN skips five epochs by prediction using weights of five previous epochs as an input. The ablation study about components of our WNN is described in Appendix D.3.

5.2. Comparisons with Curve Fitting

Setup. We first compare WNN and curve fitting approaches. In this experiment, we used a Vanilla CNN as a target network which is composed of four convolution layers and two fully connected layers. We trained the Vanilla CNN on CIFAR10 which consists of 50,000 images of 32×32 with 10 classes for training and 10,000 images for validation. For curve fitting model, we tested diverse models such as linear, polynomial, exponential, Xexponential, sigmoid, power, and Michaelis Menten (described in Appendix C). Inputs of the curve fitting methods were weights of past five epochs to predict future weights after five epochs. All the experiments were repeated 5 times to obtain average values.

Results. As shown in Figure 7(a), both WNN and curve fitting-based approaches showed faster convergence than naive training. Our WNN clearly outperformed all curve fitting approaches. WNN requires *the least the number of epochs to the same validation accuracy*. To sum up, those experimental results show 1) effectiveness of nowcasting concept for training DNNs, and 2) superiority of our WNN over curve fitting in both stability and speed.

Visualization. We verified the forecasting process through



(a) (b)

Figure 7. Comparisons of validation accuracy (a) of the proposed and curve fitting models, (b) of the proposed with previous methods (HyperNetworks, L2O, L2O-Amalgam, and Introspection). The shading represents the variation in validation accuracy of five trials.

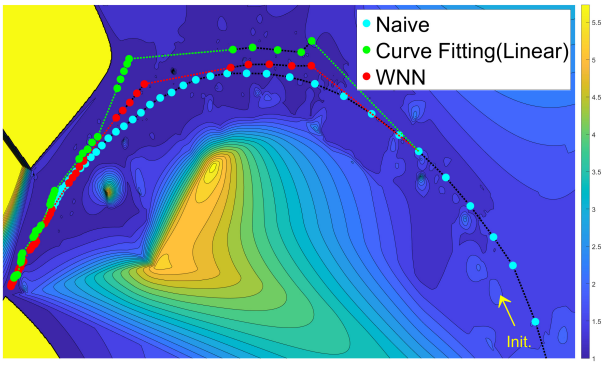


Figure 8. Landscape visualization of loss and trajectory of updates with Naive training, Curve Fitting, and WNN. WNN (red) accurately follows training trajectory of naive training (cyan) in the weight space.

landscape visualization for qualitative comparisons of our WNN, the curve fitting, and the naive training. We trained the Vanilla CNN once and did freeze all layers except the first convolution layer to focus on changes of weights in the layer. Then, we re-initialized and re-trained the layer of the network for each algorithm. All weights in the layer were saved per each epoch with corresponding validation loss during an entire training process. To fill the rest part of the landscape map, we added the Gaussian random noise and trained the network repeatedly. With accumulated weights and validation loss values, we build a landscape map that illustrate changes of weights on the loss landscape. Figure 8 shows the loss landscape which describes the training process of our WNN, the curve fitting, and the naive training. WNN allows weights to jump farther than the naive method during training process, resulting in faster convergence to the optimum solution than the naive one. The curve fitting algorithm also works well, but it shows less stable jumps at the early phase than them of WNN.

5.3. Comparisons with Similar Recent Methods

Setup. We conducted comparative experiments with HyperNetworks, L2O, L2O-amalgam, and Introspection which are representative efficient training methods. For HyperNetworks, a separate network was attached on the Vanilla CNN to approximate weights. In case of L2O, we used a pre-trained LSTM optimizer to train the Vanilla CNN. For L2O-Amalgam, we used the officially provided pre-trained model. For Introspection, we set forecasting points as the 20th and 40th epoch. For fair comparison, we trained the Introspection network using our collected dataset.

Results. Figure 7 (b) shows that all the algorithms, *i.e.*, WNN, curve fitting, L2O, L2O-Amalgam, HyperNetworks, and Introspection, improve convergence speed compared to the naive training. The linear curve fitting required only

Table 4. Time cost comparisons to train a Vanilla CNN by using various recent methods on the CIFAR10 dataset. NVIDIA TITAN Xp GPU was used to estimate time cost. "Converge" is the time to reach a validation accuracy of 59%

Method	Update (sec/batch)	meta-learning (hour)	forecasting (sec)	Converge (sec)	Speed Up
Naive Training	0.0245	-	-	52.82	$\times$1.00
HyperNetworks	0.0267	-	-	51.02	$\times$1.04
L2O	0.0290	20 (per task)	-	44.05	$\times$1.20
L2O-Amalgam	0.0291	6.35 (only once)	-	67.02	$\times$0.79
Introspection	0.0245	0.03 (only once)	0.015	43.23	$\times$1.22
CF (Linear)	0.0245	-	0.015	39.71	$\times$1.33
CF (Exponential)	0.0245	-	3.32	61.94	$\times$0.85
WNN	0.0245	0.08 (only once)	0.015	**25.27**	$\times$2.09

additional 0.015 seconds to forecast as presented in Table 4.

For HyperNetworks, since the number of trainable weights is small, the loss rapidly decreases at the beginning as shown in Fig 7 (b). However, the actual convergence is similar to or slightly worse than the naive approach due to the limitation of representation power. Furthermore, Table 4 shows that each update is slower because of additional cost for calculation of HyperNetworks' gradients.

Both L2O and L2O-Amalgam require fewer epochs to reach the validation accuracy of 59% than the naive training, but L2O-amalgam is slower than the naive training in actual time. However, as shown in Table 4, updates of L2O and L2O-Amalgam are slow because of additional computational time for the LSTM optimizer. Furthermore, L2O requires additional meta-learning hours to train the target network on a specific dataset or an architecture. L2O-Amalgam is a generalized version of L2O, so it can be used without an additional time cost for fine-tuning process. On the other hand, WNN only requires an additional 0.015 sec of weight nowcasting once per 5 epochs.

Finally, Introspection demonstrates slower convergence than our WNN due to its inefficient forecasting strategy as explained in Sections 2 and 3, and structural difference as described in Appendix D.3. This is because the Introspection forecasts far-future weights twice, but our work repeatedly forecast near-future weights. To reach a validation accuracy of 59%, which all methods could accomplish, WNN requires the least time (25.27 sec) outperforming all other approaches ($\times$ 1.71 faster than Introspection).

6. Experiments: Task-agnostic Applicability

In this section, we validate superior generalization ability of our WNN by training a target network in unseen training conditions and tasks. To be specific, *without fine-tuning WNN and further training data*, we verified our WNN is effective even in unseen tasks and unseen modalities.

Setup. On the basis of general tendency in history of weight changes across various tasks as described in Section 4.2, we try out WNN for training various models for a variety of tasks to validate its universal applicability to unseen tasks and modalities without fine-tuning WNN. In particular, experiments on language modeling and reinforcement learning show that our WNN (trained on only image classification) can even perform well with totally different modalities (*i.e.*, text and control). We set various unseen conditions as:

(i) **ImageNet Classification.** ImageNet (Deng et al., 2009) is a widely-used large-scale dataset with 1.3 million images from 1,000 classes. We trained the MobileNetV2 using the Adam with exponential decay.

(ii) **Image Segmentation.** DeepLabV3+ (Chen et al., 2018) with the MobileNetV2 backbone was trained on the PASCAL VOC 2012 dataset (Everingham et al., 2015) with the SGD, cosine decay, and cross entropy.

(iii) **Pose Estimation.** We adopted OpenPose (Cao et al., 2017) with an ImageNet-pre-trained VGG19 backbone on the MS COCO 2016 (Lin et al., 2014) that consists of over 100K persons' keypoints.

(iv) **Language Modeling.** The goal is to predict a masked word from a sequence of words, *i.e.*, text. The universal BERT (Dehghani et al., 2019) on the WikiText-2 dataset which consists of over 2 million words was trained with the Adam, the masked LM loss, and the penalized confidence (Pereyra et al., 2017).

(v) **Reinforcement Learning.** We applied WNN to the Pendulum problem, which is a famous reinforcement learning problem using the Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015). Two networks, an actor and a critic, were trained using the Adam for 200 episodes.

(vi) **Transfer Learning on Attention Model.** PVTv2-B0 (Wang et al., 2022) with ImageNet-pre-trained weights was trained on CIFAR100 for 50 epochs using the Adam, the warm-up scheduling, and decay.

(vii) **Diffusion Model.** We validated on the Denoising Diffusion Implicit Model (DDIM) (Song et al., 2020) on the Oxford Flowers dataset (Nilsback & Zisserman, 2008). The model was trained to minimize $\ell 1$ loss using the AdamW with the learning rate decay from $1e$-3 for 60 epochs. We evaluated it based on KID (Kernel Inception Distance) metric (Bińkowski et al., 2018).

Each experiment was replicated at least 3 times (up to 30) to obtain average performance and avoid bias. Training recipe of each task is described in Appendix. I.

Results. Table 5 summarizes the overall performance comparisons demonstrating that WNN outperforms all the other approaches in required epochs to reach $90 \sim 97\%$ of the best accuracy (right before convergence) for all the tasks. WNN, the curve fitting, and the Introspection obtained $\times 1.25$, $\times 0.99$, and $\times 0.94$ speed-up on average compared

Table 5. Experimental comparisons on various tasks with naive training, Introspection, a linear curve fitting, and the proposed WNN. WNN consistently outperforms the other methods on a variety of tasks.

Task (Metric)	Method	Best	Converge (epoch/episode)	Reach	Speed Up	Task (Metric)	Method	Best	Converge (epoch/episode)	Reach	Speed Up
ImageNet Classification (Val Acc↑)	Naive	69.40%	26	68.00%	× 1.00	RL (Episode Reward↑)	Naive	-139.10	116	-200.00	× 1.00
	Introspection	70.13%	24	68.00%	× 1.08		Introspection	-128.70	107	-200.00	× 1.08
	CF (Linear)	68.19%	41	68.00%	× 0.63		CF (Linear)	-138.79	111	-200.00	× 1.05
	WNN	70.57%	21	68.00%	× **1.24**		WNN	-123.55	88	-200.00	× **1.32**
Image Segmentation (Val Jaccard↑)	Naive	53.99%	55	48.00%	× 1.00	Transfer Learning on Attention Model (Val Acc↑)	Naive	83.22%	17	80.00%	× 1.00
	Introspection	53.97%	46	48.00%	× **1.20**		Introspection	81.52%	38	80.00%	× 0.45
	CF (Linear)	53.79%	50	48.00%	× 1.10		CF (Linear)	82.51%	14	80.00%	× **1.21**
	WNN	53.81%	45	48.00%	× **1.20**		WNN	83.09%	14	80.00%	× **1.21**
Pose Estimation (Val Loss↓)	Naive	623.54	43	673.00	× 1.00	Diffusion Model (Val KID↓)	Naive	0.1918	32	0.2000	× 1.00
	Introspection	627.17	46	673.00	× 0.94		Introspection	0.1786	36	0.2000	× 0.89
	CF (Linear)	614.80	30	673.00	× **1.43**		CF (Linear)	0.1878	54	0.2000	× 0.59
	WNN	615.79	30	673.00	× **1.43**		WNN	0.1732	26	0.2000	× **1.23**
Language Modeling (Val Perplexity↓)	Naive	31.25	55	100.00	× 1.00	Average	Naive	N/A	N/A	N/A	× 1.00
	Introspection	39.93	56	100.00	× 0.98		Introspection	N/A	N/A	N/A	× 0.94
	CF (Linear)	42.67	58	100.00	× 0.95		CF (Linear)	N/A	N/A	N/A	× 0.99
	WNN	31.26	48	100.00	× **1.15**		WNN	N/A	N/A	N/A	× **1.25**

to the naive training, respectively. Moreover, WNN consistently outperforms the naive training on a variety of tasks though it was trained only on weights from the image classification task. In contrast, the Introspection and the curve fitting method often underperform the naïve training. In this regard, our WNN is more robust to unseen tasks without a task-specific fine-tuning process than similar approaches. Further, without any architecture-specific fine-tuning, our WNN is applicable to transfer learning to a completely different architecture such as Transformer.

7. Conclusions

In this paper, we introduce a concept to periodically predict near future weights during training process, *i.e.*, Learning to Boost Training, which can directly save the number of epochs during training. The concept was first justified and proved through in-depth studies on weight prediction, which significantly improved training efficiency when realized as WNN. The WNN, when trained for various training cases for image classification, was able to aid training of various models with different optimizers for several unseen tasks such as image segmentation, pose estimation, word prediction, reinforcement learning, and even a generative task.

The limitation is that it requires additional memory and computation for using WNN, but those are far little compared to training efforts for large-scale models. WNN is also needed to be once trained with a collection of weight changes, but it is task-agnostic and we provide the pre-trained model and weight change dataset ($\sim$200 GB). Also, note that there is no additional benefit in task performance such as validation accuracy. Our WNN is very flexible and thus has significant potential to help lots of future researches that suffer from lack of computational cost.

Acknowledgement

This work was supported by Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-00842, Development of Cloud Robot Intelligence for Continual Adaptation to User Reactions in Real Service Environments)

References

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. Learning to learn by gradient descent by gradient descent. In *NeurIPS*, pp. 3988–3996, 2016.

Bińkowski, M., Sutherland, D. J., Arbel, M., and Gretton, A. Demystifying MMD GANs. In *ICLR*, 2018.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *NeurIPS*, 33:1877–1901, 2020.

Cao, Z., Simon, T., Wei, S.-E., and Sheikh, Y. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F., and Adam, H. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *ECCV*, pp. 801–818, 2018.

Dauphin, Y. N. and Schoenholz, S. Metainit: Initializing learning by learning to initialize. *NeurIPS*, 32, 2019.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, L. Universal transformers. In *ICLR*, 2019.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *CVPR*, pp. 248–255. IEEE, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Everingham, M., Eslami, S. A., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. The pascal visual object classes challenge: A retrospective. *International journal of computer vision*, 111(1):98–136, 2015.

Foret, P., Kleiner, A., Mobahi, H., and Neyshabur, B. Sharpness-aware minimization for efficiently improving generalization. In *ICLR*, 2020.

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

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *CVPR*, pp. 4700–4708, 2017.

Huang, T., Chen, T., Liu, S., Chang, S., Amini, L., and wang, Z. Optimizer amalgamation. In *ICLR*, 2022.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

Khailany, B., Ren, H., Dai, S., Godil, S., Keller, B., Kirby, R., Klinefelter, A., Venkatesan, R., Zhang, Y., Catanzaro, B., et al. Accelerating chip design with machine learning. *IEEE Micro*, 40(6):23–32, 2020.

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

Knyazev, B., Drozdzal, M., Taylor, G. W., and Romero Soriano, A. Parameter prediction for unseen deep architectures. *NeurIPS*, 34:29433–29448, 2021.

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

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

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *ECCV*, pp. 740–755. Springer, 2014.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. In *ICLR*, 2017.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *ICLR*, 2019.

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

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, pp. 116–131, 2018.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. In *ICLR*, 2017.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *NIPS Workshop*, 2013.

Nilsback, M.-E. and Zisserman, A. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, 2008.

Peebles, W., Radosavovic, I., Brooks, T., Efros, A. A., and Malik, J. Learning to learn with generative models of neural network checkpoints. *arXiv preprint arXiv:2209.12892*, 2022.

Pereyra, G., Tucker, G., Chorowski, J., Kaiser, Ł., and Hinton, G. Regularizing neural networks by penalizing confident output distributions. *arXiv preprint arXiv:1701.06548*, 2017.

Rodríguez, P., Gonzalez, J., Cucurull, G., Gonfaus, J. M., and Roca, X. Regularizing cnns with locally constrained decorrelations. In *ICLR*, 2017.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, pp. 4510–4520, 2018.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sinha, A., Mukherjee, A., Sarkar, M., and Krishnamurthy, B. Introspection:accelerating neural network training by learning weight evolution. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Hkg8bDqee.

Song, J., Meng, C., and Ermon, S. Denoising diffusion implicit models. In *ICLR*, 2020.

Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, pp. 6105–6114, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, W., Xie, E., Li, X., Fan, D.-P., Song, K., Liang, D., Lu, T., Luo, P., and Shao, L. Pvtv2: Improved baselines with pyramid vision transformer. *Computational Visual Media*, 8(3):1–10, 2022.

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

Zhu, C., Ni, R., Xu, Z., Kong, K., Huang, W. R., and Goldstein, T. Gradinit: Learning to initialize neural networks for stable and efficient training. *NeurIPS*, 34:16410–16422, 2021.

Appendix

A. Experimental Details of Section 3

This section describes experimental procedures and setting of Section 3. We adopted two model architectures, a Vanilla CNN and a ResNet, with Xavier Uniform and He Normal initialization methods on the CIFAR10 dataset from scratch. For both cases, we used the Adam optimizer with $1e$-3 learning rate and random data augmentation (*e.g.*, flip, rotation, scaling, translation). We first collected weight snapshots from initialization to the 100th training epoch for the Vanilla CNN and ResNet which were represented as $[\Theta_{init}^{CNN}, \Theta_{1}^{CNN}, ..., \Theta_{100}^{CNN}]$ and $[\Theta_{init}^{Resnet}, \Theta_{1}^{Resnet}, ..., \Theta_{100}^{ResNet}]$, and determine the target validation accuracies at the epoch where the weights were changed most extremely (Θ_{50}^{CNN}, Θ_{20}^{Resnet}).

A.1. The Future is Roughly Predetermined (Section 3.1)

To figure out the training trajectories, *i.e.*, histories of weight changes during training, roughly determined by initial weights, we repeatedly trained models 30 times with two different weight initialization conditions: 1) training started from the same initial weights (Θ_{init}^{CNN}, Θ_{init}^{ResNet}) and 2) from random initial weights. For each trial, we trained the models until reaching the target validation accuracies achieved by Θ_{50}^{CNN} for Vanilla CNN and Θ_{20}^{Resnet} for ResNet. We collected two sets of weight snapshots: (1) weights updated from the same initial weights ($\Phi^{CNN,i}$ and $\Phi^{Resnet,i}$, $i = 1, ..., 30$) and (2) weights updated from random initial weights ($\Psi^{CNN,i}$,$\Psi^{Resnet,i}$, $i = 1, ..., 30$). Then, we calculated cosine correlations between trials in each set (see Table A.1). We also report the correlations when training started from Θ_{15}^{ResNet}, not Θ_{init}^{ResNet}, until reaching the target validation accuracy at Θ_{20}^{Resnet} for verifying higher certainty of near future weights. The average correlation was 0.9841, which indicates that the consistency is higher from the near past weights.

Table A.1. Predetermined Convergence: the cosine correlations between weights after training when using random initial weights and using the same initial weight.

Architecture		Cosine Correlation	
		Random Initial Weights(Ψ)	The Same Initial Weights(Φ)
Vanilla CNN(from Θ_{init}^{CNN} to Θ_{50}^{CNN})	Conv	0.0131 $\pm$0.0345	0.9018 $\pm$0.0142
	Bias	0.0306$\pm$0.0218	0.9229 $\pm$0.0293
ResNet(from Θ_{init}^{Resnet} to Θ_{20}^{Resnet})	Conv	0.0594 $\pm$0.1712	0.8959 $\pm$0.0137
ResNet(from Θ_{15}^{Resnet} to Θ_{20}^{Resnet})	Conv	N/A	0.9841 $\pm$0.0046

Layer-wise correlations of Vanilla CNN are in the table below.

Table A.2. Layer-wise Convergence for Vanilla CNN

Initialization	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 5
Random	0.06023$\pm$0.1095	0.0025$\pm$0.0260	-0.005$\pm$0.0233	-0.0036$\pm$0.0064	-0.0011$\pm$0.0098	-0.0258$\pm$0.0320
Fixed	0.9853$\pm$0.0011	0.9391$\pm$0.0210	0.8862$\pm$0.0150	0.7612$\pm$0.0212	0.8952$\pm$0.0132	0.9438$\pm$0.0069

We also tested how strong data augmentation (e.g. mixup, random crop, large image rotation) and learning rate scheduling (e.g. cosine decay) affect the update paths when training the networks. The correlation with strong data augmentation was lower than one with weak augmentation only. However, they were still much higher than that with random initialization (0.0131 for Vanilla CNN and 0.0594 for ResNet).

Table A.3. Additional analysis about Predetermined Convergence for LR scheduling and strong data augmentation.

Architecture	Cosine Correlation	
	Weak Augmentation w/o LR scheduling	Strong Augmentation w/ LR scheduling
Vanilla CNN (Random Initialization)	0.0131 $\pm$0.0345	N/A
Vanilla CNN (Fixed Initialization)	0.9018 $\pm$0.0142	0.7424 $\pm$0.0343
ResNet (Random Initialization)	0.0594 $\pm$0.1712	N/A
ResNet (Fixed Initialization)	0.8959 $\pm$0.0137	0.8559 $\pm$0.0127

A.2. Beneficial Prediction of Future Weights (Section3.2)

To identify what predicted weights are beneficial, we simulated training processes with artificially generated predicted weights by adding noise to the target weights. In this simulation experiment, there were three components, starting weights

Θ_{start}, predicted weights $\Theta_{predicted}$, and target weights Θ_{target}. Note that we confirmed the existence of Θ_{target} in Appendix A.1. We set the $\Theta_{predicted}$ to $\theta_{target} + \epsilon$. Based on virtually generated predicted weights $\Theta_{predicted}$, we conducted training experiments on varying Θ_{start} and Θ_{target} as:

Train Simulation with Varying Θ_{target}: we verified that weight prediction can be worth for varying Θ_{target}, so we fixed Θ_{start} to [Θ_{init}^{CNN} and Θ_{init}^{Resnet}], and set Θ_{target} to [Θ_{5}^{CNN}, Θ_{50}^{CNN}, Θ_{100}^{CNN}] and [Θ_{5}^{Resnet}, Θ_{20}^{Resnet}, Θ_{100}^{Resnet}]. Then we followed the procedure as described in Algorithm 1. By this procedure, we can get 1) the number of required epochs

Algorithm 1 Train Simulation with Varying Θ_{target}

A target network $\mathcal{T}$, training dataset (X^{tr}, Y^{tr}), and test dataset (X^{te}, Y^{te})
Thresh = Accuracy($\mathcal{T}(X^{te}, \Theta_{target}), Y^{te}$)
Current = 0
Set weights $\Theta_{\mathcal{T}}$ of $\mathcal{T}$ as Θ_{start}
cnt=0
while Current $<$ Thresh **do**
  Update $\Theta_{\mathcal{T}}$ on (X^{tr}, Y^{tr}) one epoch
  Current = Accuracy($\mathcal{T}(X^{te}, \Theta_{\mathcal{T}})$
  cnt = cnt+1
end while
Set $Epoch_{start}$=cnt
Outputs = []
NumSample=0
while NumSample <500 **do**
  Set $\epsilon' \sim \mathcal{N}(0, I)$
  Set a relative distance $d \sim Unif(0, 8)$
  Set $\epsilon = d \circ ||\theta_{target} - \theta_{start}||_2 \circ \frac{\epsilon'}{||\epsilon'||_2}$ for all $\theta \in \Theta$
  Set $\theta_{Predicted} = \theta_{target} + \epsilon$ for all $\theta \in \Theta$
  Current = 0
  Set weights $\Theta_{\mathcal{T}}$ of $\mathcal{T}$ as $\Theta_{Predicted}$
  cnt=0
  while Current $<$ Thresh **do**
    Update $\Theta_{\mathcal{T}}$ on (X^{tr}, Y^{tr}) one epoch
    Current = Accuracy($\mathcal{T}(X^{te}, \Theta_{\mathcal{T}})$
    cnt = cnt+1
  end while
  Set $Epoch_{predicted}$=cnt
  Output.append([d, $\frac{Epoch_{predicted}}{Epoch_{start}}$])
  NumSample = NumSample+1
end while
return Output

from Θ_{start} and 2) the number of required epochs from $\Theta_{predicted}$. Figure 2 represents the ratio between the two numbers according to the relative distance.

Train Simulation with Varying Θ_{start}: we also conducted train simulations with varying Θ_{start} and the fixed Θ_{target}. For Θ_{target}, we used Θ_{50}^{CNN} and Θ_{20}^{Resnet}. For Θ_{start}, (Θ_{5}^{CNN}, Θ_{2}^{Resnet}), (Θ_{25}^{CNN}, Θ_{10}^{Resnet}), and (Θ_{45}^{CNN}, Θ_{15}^{Resnet}) were used. Then, we performed training experiments to identify the best times to predict future weights based on the categorization of the difference between the target epoch and the start epoch into three groups: short-term (from Θ_{t-5} to Θ_t), mid-term (from $\Theta_{t/2}$ to Θ_t), and long-term (from $\Theta_{t/10}$ to Θ_t), where t means the epoch number of Θ_{target}. The detailed procedure for this simulation is described in Algorithm 2. From the procedure, we obtained RRE, relative distance, and relative direction. Figure 3 represents the results of this train simulation experiment. those results indicate that the short-term prediction is more reliable and has a wider range of beneficial regions for weight prediction.

Algorithm 2 Train Simulation with Varying Θ_{start}

A target network $\mathcal{T}$, training dataset (X^{tr}, Y^{tr}), and test dataset (X^{te}, Y^{te})
Thresh = Accuracy($\mathcal{T}(X^{te}, \Theta_{target}), Y^{te}$)
Current = 0
Set weights $\Theta_{\mathcal{T}}$ of $\mathcal{T}$ as Θ_{start}
cnt=0
while Current $<$ Thresh **do**
  Update $\Theta_{\mathcal{T}}$ on (X^{tr}, Y^{tr}) one epoch
  Current = Accuracy($\mathcal{T}(X^{te}, \Theta_{\mathcal{T}})$
  cnt = cnt+1
end while
Set $Epoch_{start}$=cnt
Outputs = []
NumSample=0
while NumSample $<$1000 **do**
  Set $\epsilon' \sim \mathcal{N}(0, I)$
  Set a relative distance: $d \sim Unif(0, 8)$
  Set a relative direction: $a \sim Unif(-1, 1)$
  Find a plane P containing ϵ' and $\theta_{target} - \theta_{start}$
  Find ϵ that is on the P and orthogonal to $\theta_{target} - \theta_{start}$
  Set $\epsilon = d \circ \sqrt{(1-a)} \circ ||\theta_{target} - \theta_{start}||_2 \circ a \circ \frac{\epsilon'}{||\epsilon'||_2}$ for all $\theta \in \Theta$
  Set $\theta_{Predicted} = \theta_{target} + \epsilon$ for all $\theta \in \Theta$
  Current = 0
  Set weights $\Theta_{\mathcal{T}}$ of $\mathcal{T}$ as $\Theta_{Predicted}$
  cnt=0
  while Current $<$ Thresh **do**
    Update $\Theta_{\mathcal{T}}$ on (X^{tr}, Y^{tr}) one epoch
    Current = Accuracy($\mathcal{T}(X^{te}, \Theta_{\mathcal{T}})$
    cnt = cnt+1
  end while
  Set $Epoch_{predicted}$=cnt
  Output.append([$r, a, \frac{Epoch_{predicted}}{Epoch_{start}}$])
  NumSample = NumSample+1
end while
return Output

A.3. Weight Prediction is Feasible. (Section 3.3)

We tried to figure out the feasibility of weight prediction using a curve fitting method with two strategies, mid-term and short-term prediction. The mid-term one was utilized by Introspection (Sinha et al., 2017), and the short-term one was set from our observation. We applied the curve fitting to independently predict weight parameters by one-by-one. Then, the curve fitting model ($\mathcal{C}$) can be expressed as:

$$(mid-term) \qquad \mathbf{c}^* = \underset{\mathbf{c}}{\operatorname{argmin}} \sum_{\tau} (\theta_{\tau}^{l,i,j} - \mathcal{C}(\tau, \mathbf{c}))^2 + \rho \sum_{i=1}^{b} c_i^2, \quad c \in \mathbf{c} \quad and \quad \tau \in \{0, 0.2t, 0.35t, 0.5t\} \tag{11}$$

$$(short-term) \qquad \mathbf{c}^* = \underset{\mathbf{c}}{\operatorname{argmin}} \sum_{\tau=t-9}^{t-5} (\theta_{\tau}^{l,i,j} - \mathcal{C}(\tau, \mathbf{c}))^2 + \rho \sum_{i=1}^{b} c_i^2, \quad c \in \mathbf{c} \tag{12}$$

$$\hat{\theta}_t^{l,i,j} = \mathcal{C}(t, \mathbf{c}^*) \tag{13}$$

where t means the number of epochs, c_i^2 is ℓ_2-regularizer with a hyperparameter ρ, and b is the number of coefficients in the curve fitting model. Among the curve-fitting models, we chose a linear model because of its simplicity without optimizing it. The linear curve fitting-based predictor receives weight parameters from multiple epochs and returns a predicted weight parameter $\hat{\theta}_t^{l,i,j}$ by solving the Pseudo inverse.

Similar to Introspection, the mid-term strategy receives a [4×1]-shaped input which is composed of $[\theta_0^{l,i,j}, \theta_{0.2t}^{l,i,j}, \theta_{0.35t}^{l,i,j}, \theta_{0.5t}^{l,i,j}]$, and predicts $\theta_t^{l,i,j}$, where l, i, j mean the indices of layer, channel, spatial coordinate, and $\theta \in \Theta$. To be specific, the linear curve fitting model with the mid-term strategy can be described as:

$$\begin{aligned} A &= \begin{bmatrix} 0 & 0.2t & 0.35t & 0.5t \\ 1 & 1 & ... & 1 \end{bmatrix} \\ W_t &= [\theta_0^{l,i,j}, \theta_{0.2t}^{l,i,j}, \theta_{0.35t}^{l,i,j}, \theta_{0.5t}^{l,i,j}]^T, \\ \hat{\theta}_t^{l,i,j} &= W_t^T A^T (AA^T + \rho I)^{-1} [t, 1]^T. \end{aligned} \tag{14}$$

The short-term strategy also approximates the $\theta_t^{l,i,j}$, but differs in input. It receives a [5×1]-shaped input which is composed of $[\theta_{t-9}^{l,i,j}, \theta_{t-8}^{l,i,j}, \theta_{t-7}^{l,i,j}, \theta_{t-6}^{l,i,j}, \theta_{t-5}^{l,i,j}]$. The linear curve fitting model for short-term strategy can be described as:

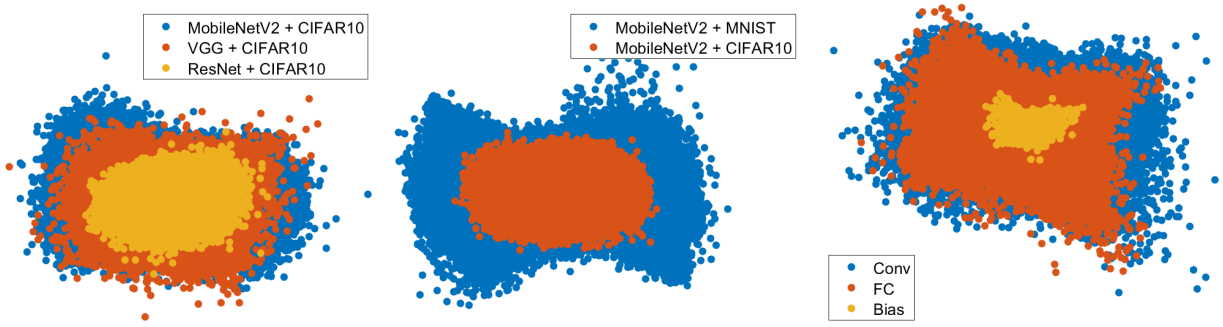
$$\begin{aligned} A &= \begin{bmatrix} t-9 & t-8 & ... & t-5 \\ 1 & 1 & ... & 1 \end{bmatrix} \\ W_t &= [\theta_{t-9}^{l,i,j}, \theta_{t-8}^{l,i,j}, ..., \theta_{t-5}^{l,i,j}]^T, \\ \hat{\theta}_t^{l,i,j} &= W_t^T A^T (AA^T + \rho I)^{-1} [t, 1]^T. \end{aligned} \tag{15}$$

The results are represented in Table 3. The linear curve fitting can predict the future weight parameters within the acceptable error, *i.e.*, the beneficial region depicted in Figure 3 in Section 3.1. Note that ResNet has Batch Normalization layers that make bias terms meaningless, so we didn't calculate the prediction errors for the bias of ResNet. We also tested the mid-term approach with the [5×1]-shaped input, but it slightly underperformed it with the [4×1]-shaped input.

B. Weight Change Tendency is Consistent Across Totally Different Tasks. (Section 4.2)

As explained in Section 4.2, we collected weights of the first 15 epochs of training processes from scratch for the three tasks: image classification, language modeling, and reinforcement learning. For each task, we trained the VGG16 (Simonyan & Zisserman, 2014) and the MobileNetV2 on CIFAR10 (Krizhevsky et al., 2009) for image classification, BERT (Devlin et al., 2018) on WikiText-2 (Merity et al., 2017) for language modeling, and DDPG (Lillicrap et al., 2015) on the Pendulum problem for reinforcement learning using the Adam with $1e$-3 learning rate. Experimental results are described in Figure 6. The individual history of weights formed clusters, indicating weight parameters are changed similarly in element-wise level regardless of target tasks. Note that it doesn't mean the converged weights, *i.e.*, a set of weight parameters, are the same for all the tasks because the element-wise trend doesn't consider layer indices, channel indices, and spatial coordinates of weights.

Additionally, we conducted the same experiments to visualize weight parameter change tendencies for diverse DNN architectures and datasets. We collected the history of weights of the first 15 epochs of a training process according to various architectures, datasets, and operation types. Then, we applied PCA for dimension reduction. As shown in Figure 9, these results illustrate that *single predictor of weights can be generalized not only to various tasks but also to various architectures and datasets.* Then, Figure 9(c) indicates that separate predictors for each operation type are mandatory to cover various types of operations.



(a) Architectures (b) Datasets (c) Operation types

Figure 9. Visualization of changes of weights across different architectures (Convolution of MobileNetV2, ResNet and VGGNet), datasets (CIFAR10 and MNIST), operations types(Convolution, Fully-connected, and Bias). Each dot is a history of changes of a weight for 15 epochs (compressed in 2D).

C. Details of Comparisons with Curve Fitting (Section 5.2)

In this section, we tested more diverse curve fitting models such as linear, exponential, and 2nd ordered polynomial, Xexponential, sigmoid, power, and Michaelis Menten as below: The curve fitting model ($\mathcal{C}$) can be expressed as:

$$\mathbf{c}^* = \underset{\mathbf{c}}{\operatorname{argmin}} \sum_{\tau=t-9}^{t-5} (\theta_\tau^{l,i,j} - \mathcal{C}(\tau, \mathbf{c}))^2 + \rho \sum_{i=1}^{b} c_i^2, \quad c \in \mathbf{c} \tag{16}$$

$$\hat{\theta}_t^{l,i,j} = \mathcal{C}(t, \mathbf{c}^*) \tag{17}$$

and

$$\mathcal{C}(x) = c_{11}x + c_{12} \qquad (Linear) \tag{18}$$

$$\mathcal{C}(x) = c_{21}x^2 + c_{22}x + c_{23} \qquad (Polynomial) \tag{19}$$

$$\mathcal{C}(x) = c_{31}e^{(-c_{32}x)} + c_{33} \qquad (Exponential) \tag{20}$$

$$\mathcal{C}(x) = c_{41}x\exp(-c_{42}x) + c_{43} \qquad (XExponential) \tag{21}$$

$$\mathcal{C}(x) = c_{51}sigmoid(c_{52}x) + c_{53} \qquad (Sigmoid) \tag{22}$$

$$\mathcal{C}(x) = c_{61}x^{c_{62}} + c_{63} \qquad (Power) \tag{23}$$

$$\mathcal{C}(x) = \frac{c_{71}x + c_{72}}{x + c_{73}} \qquad (MichaelisMenten), \tag{24}$$

where c means coefficient of each model, and x indicates their input.

Figure 10 shows the results using a Vanilla CNN and CIFAR10. The power and the Michaelis Menten functions cannot accelerate convergence, but the others lead to faster convergence than naive training. For early training, exponential, sigmoid, power, and Michaelis Menten functions are more effective than the linear and 2nd-ordered polynomial functions as shown in changes of validation loss of various curve fitting algorithms. However, the linear and 2nd-ordered polynomial functions outperform other functions from the 20th epoch in curve fitting algorithms. Our WNN consistently performs better than a variety of curve-fitting models.

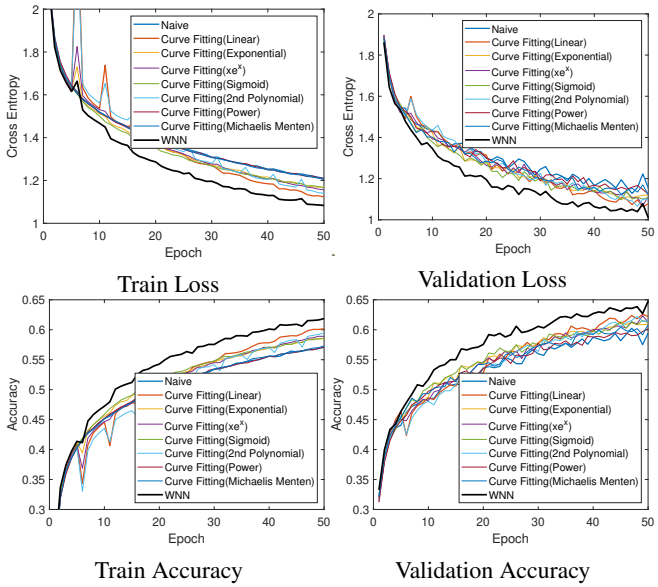


Figure 10. Comparisons of loss and accuracy with various curve fitting models

D. Weight Nowcaster Network

D.1. Network Architecture

We used a simple two-stream architecture, $\mathcal{F} = \{F_W, F_{dW}\}$, that is composed of a set of fully connected layers and an activation network as illustrated in Figure 5. Values of weights and their temporal differentials are normalized with bias term $\theta^{l,i,j}_{t-5}$ and scaled by min-max scaler $(max(W^o_t) - min(W^o_t))$ and scaling factor 2 for stable forecasting, and fed into each stream respectively:

$$W^o_t = [\theta^{l,i,j}_{t-9}, \theta^{l,i,j}_{t-8}, \theta^{l,i,j}_{t-7}, \theta^{l,i,j}_{t-6}, \theta^{l,i,j}_{t-5}]^T, \tag{25}$$

$$dW^o_t = [(\theta^{l,i,j}_{t-5} - \theta^{l,i,j}_{t-6}), ..., (\theta^{l,i,j}_{t-8} - \theta^{l,i,j}_{t-9})]^T, \tag{26}$$

$$W_t = \frac{W^o_t - \theta^{l,i,j}_{t-5}}{2 \cdot (max(W^o_t) - min(W^o_t))}, \tag{27}$$

$$dW_t = \frac{dW^o_t}{2 \cdot (max(W^o_t) - min(W^o_t))}. \tag{28}$$

where l, i, j mean the indices of layer, channel, spatial coordinates, t indicates the number of epochs, and $\theta \in \Theta$. We set the scaling factor to 2. Each branch has a structure as:

$$l(\mathbf{x}) = \sigma_1(\xi^l \cdot [\mathbf{x}, 1]^T) \tag{29}$$

$$act(\mathbf{x}) = (\xi^a_1 \cdot l(\mathbf{x})) \odot e^{(\xi^a_2 \cdot l(x)) \odot \mathbf{x}} + (\xi^a_3 \cdot l(\mathbf{x})) \tag{30}$$

$$F_b(\mathbf{x}) = \sigma_2(\xi^F_2 \cdot [act(\sigma_2(\theta_1 \cdot [\mathbf{x}, 1]^T)), 1]^T) \tag{31}$$

where $\mathbf{x}$ is an input vector, σ means activation function, $l(\mathbf{x})$ is an intermediate feature used in activation network, act is an activation network, and F_b is an output vector derived from an individual branch denoted by $b \in \{W, dW\}$. Each component here contain WNN's trainable parameters $\Xi_b = \{\xi^F_1, \xi^F_2, \xi^l, \xi^a_1, \xi^a_2, \xi^a_3\}$. Both outputs from the two branches, F_W and F_{dW}, are concatenated and passed through a fully-connected layer as:

$$\mathcal{F}(W_t) = \sigma_3(\xi_o \cdot [F_W(W_t)^T, F_{dW}(dW_t)^T, 1]^T). \tag{32}$$

The $\mathcal{F}(\cdot)$ is an update term to predict future weights and $tanh(\cdot)$ was used for σ_3 as the update can be both positive and negative within $2 \cdot (max(W^o_t) - min(W^o_t))$ range. Our WNN is trained to return the *difference* between the last input weight and its future weight. Finally, ℓ_1-norm is applied on the difference between the true updates (for 5 future iterations) and the model to train, i.e., $\mathcal{F}(W_t)$ to define a loss function as

$$\min_{\mathcal{F}}(\left| \mathcal{F}(W_t, dW_t) - \frac{\theta^{l,i,j}_t - \theta^{l,i,j}_{t-5}}{2 \cdot (max(W^o_t) - min(W^o_t))} \right|_{\ell_1}) \tag{33}$$

The ℓ_1-norm was used to obtain a larger gradient even when the differences are small. The optimal solutions, $\mathcal{F}$, from the two streams and a fully connected layer are then used to estimate the future weights of the target network. Then, WNN-based training can be given as:

$$\hat{\theta^{l,i,j}}_t = \theta^{l,i,j}_{t-5} + 2 \cdot (max(W^o_t) - min(W^o_t)) \cdot \mathcal{F}(W_t, dW_t) \tag{34}$$

which let us skip 5 epochs and directly jump to the t-th epoch from $(t-5)$-th epoch. The predicted future weights are modeled by adding the outputs into the last input weights. This is an element-wise process, so WNN needs to individually operate for all parameters. This property makes our approach much faster and more efficient with neglectable error in estimating trainable parameters.

To train WNN, in practice, weights in every epoch in various training conditions; architectures, datasets, and augmentation methods were collected so that the model can be adopted to even unseen cases. More details are to be explained in Section 5.1. We assumed that each type of weight; convolution weight, fully-connected weight, and bias, have their own characteristics, so individual forecasting networks for each type were separately trained.

D.2. Perioodic Short-term Nowcasting Strategy

Based on those observations in Section 3, we established our forecasting strategy as:

(i) Element-wise independent forecasting for each weight parameter.
(ii) Separate forecasting models for each mathematical operation.
(iii) Periodic short-term forecasting rather than few-shot mid-term forecasting of Introspection (Sinha et al., 2017).

As shown in Figure 4, every five epochs, our predictor forecasts the future weights element by element. A target network is trained with naive training for five epochs, while the weight parameters of each epoch are stacked to prepare WNN inputs. Next, WNN receives the stacked weight parameters and predicts the future parameters after five epochs. From the predicted weights, naive training is restarted while compensating the prediction error. The aforementioned procedure is repeated during an entire training process.

D.3. Ablation Study and Model Analysis of WNN

In this section, we performed various ablation study of the proposed method. All the experiments were repeated 5 times on CIFAR10 with a Vanilla CNN, and WNN was used as an add-on module to expedite the training. Average values of performances measures from the replicated experiments are reported.

(i) **Components of WNN.** We conducted ablation study to validate our network architecture of WNN. As shown in Table A.4, the performance of our WNN is improved by each component, periodic short-term forecast, normalized residual, two-stream design, and activation network. Note that WNN without the four components is almost similar to Introspection, only different in the network depth.

Table A.4. Actual time cost comparisons with WNN components on CIFAR10 with Vanilla CNN

Method	Periodic Nowcasting	Norm. Residual	Two Stream	Activation Network	Converge (epoch)	Converge (sec)
Naive	N/A	N/A	N/A	N/A	44	53.82
WNN	✗	✗	✗	✗	37	44.43
	✓	✗	✗	✗	34	40.91
	✓	✓	✗	✗	31	37.30
	✓	✓	✓	✗	29	34.89
	✓	✓	✓	✓	21	**25.27**

*Converge: Epoch/Time to reach a validation Acc of 59%

(ii) **Input and skip lengths.** Table A.5 shows experimental results on various input and skip lengths of WNN. WNN with five input length and five skip length demonstrates the least time to reach 59% validation accuracy. The longer input length leads the more stable results, but it limits the number of forecast. On the other hand, the shorter one makes more frequent, but unstable forecasting. For skip length, the further forecast can skip more epochs, but it may lead inaccurate regression results.

Table A.5. Convergence for various input/skip lengths.

Input Length	Skip Length 3	5	7	10
3	31.35	34.95	34.95	37.22
5	32.49	**25.27**	30.07	31.29
10	37.26	38.46	39.66	43.26

*Time to reach a validation Acc of 59%

(iii) **Costs for nowcasting various target architectures** We conducted experiments on the actual computational cost and time for nowcasting weight parameters of various architectures. Our WNN predicts the future weights by inference without calculating gradients. 1,000,000 weight parameters of a target network can be processed in one batch for inference by using parallel processing. That is, WNN can rapidly predict weight parameters of 5 epochs future. As shown in Table A.6, the most largest architecture (VGG19) requires only 10.6 seconds for nowcasting 5 epochs future. In most training cases, time cost for less than 10.6 seconds is negligible when compared to duration for training 5 epochs using entire training data. To be specific, it generally takes much longer time than 10.6 seconds to train 1 epoch

in practice. Our WNN can directly reduce training time. Furthermore, because it is not required to save every immediate outputs of whole hidden layers for inference, our WNN does not need large memory cost.

Table A.6. Required FLOPs/Nowcasting Time for Nowcasting Various Architectures.

Architecture (#Params)	FLOPs	Nowcasting Time(Sec)
MobileNetV2 (3.5M)	45G	0.3
VGG19 (143.7M)	1835G	10.6
ResNet50 (25.6M)	327G	1.9
ResNet152 (60.4M)	771G	4.4
ViT-B (86M)	1098G	6.4
Swin-B (88M)	1123G	6.4

*WNN occupies 9,425 parameters, and 0.04MB memory.

**WNN requires 13,713 FLOPs for nowcasting one parameter.

E. Analysis on Various Hyperparameters for Training of Unseen Dataset

Setup. This experiment applies the pre-trained WNN on various conditions of training (optimizers, batch size, and learning rate) and unseen dataset of image classification. Our WNN performs one prediction per one weight parameter independently regardless of the correlation between each weight, so we believed that the WNN is applicable to not only seen but also unseen conditions. Therefore, we performed experiments in various unseen conditions to validate **general applicability** in image classification of our method without any fine-tuning. We adopted Fashion MNIST on ResNet32 to validate how well a pre-trained WNN can adapt to unseen data. Fashion MNIST is composed of 28×28 gray images with 10 classes of fashion items, and it was not used to collect training data at Section 5.1. Therefore, this experiment validates that the WNN trained on limited condition (i.e. only CIFAR10 and MNIST with Adam optimizer) can boost training of unseen dataset (Fashion MNIST) with general conditions without any fine-tuning WNN. We applied varying conditions as follow:

E.1. Analysis on Various Optimizers

For collecting training data, the Adam optimizer was only used in Section 5.1. However, there are many kinds of optimizers, so we tested on four other optimizers. SGD, AdamW(Loshchilov & Hutter, 2019), SAM(Foret et al., 2020), and L2O-amalgam were used to verify effectiveness of WNN's prediction for future weights in various optimizing environments. Surprisingly, even though the WNN was not trained on Fashion MNIST, it worked well on the Fashion MNIST dataset. Further, WNN and curve fitting outperform the naive training for all optimizers. Notice the red line (WNN) converging faster than the validation accuracy from naive training for all optimizers (i.e., SGD, AdamW, SAM, and L2O-amalgam), even though WNN was not trained on them.

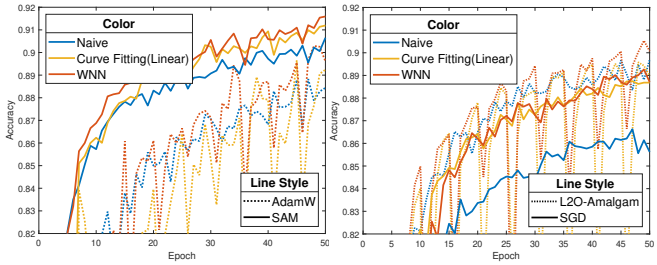
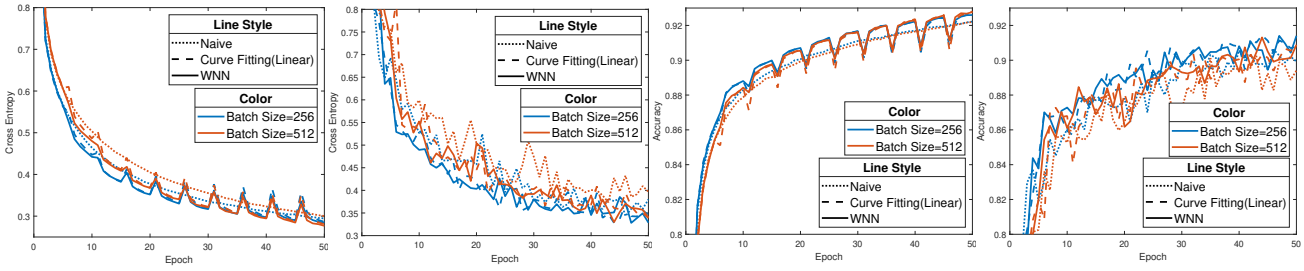


Figure 11. Comparisons of validation accuracy with unseen optimizers on Fashion MNIST (Unseen Dataset)

E.2. Analysis on Various Batch Sizes

To analyze our proposed work in detail, we conducted experiments with various batch size(256 and 512). It is widely known that a training process with large batch size is much stable because it updates using an average value of gradients with large samples. That is, the averaging operation with large batch size makes more global update in parameter space so that it can be more stable. Training stability is an important issue because it highly affects on prediction stability of WNN. As shown in Figure 12(a-d), the proposed method in 512-batch size shows better results than the naive training method in 256-batch size. If the input weights show oscillation and shaking, the predicted values would easily diverge and less reliable. This leads training with larger batch size to show bigger difference between the proposed and the naive training.



(a) Train Loss, Batch Size

(b) Validation Loss, Batch Size

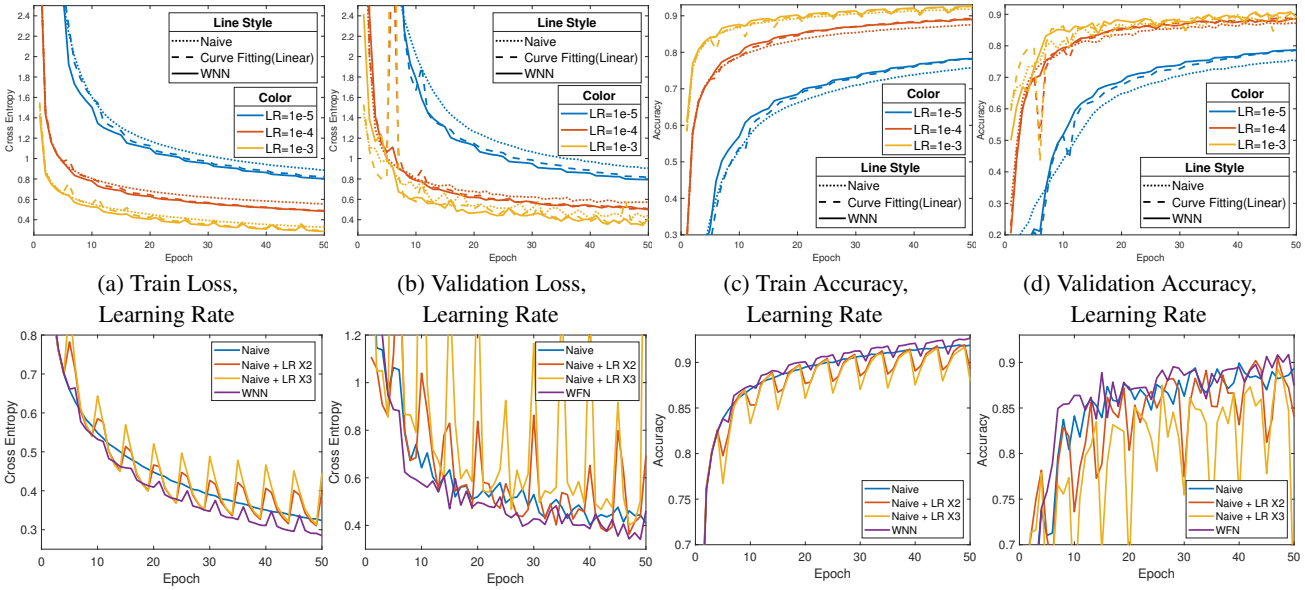
(c) Train Accuracy, Batch Size

(d) Validation Accuracy, Batch Size

Figure 12. Experimental results on Fashion MNIST (Unseen Dataset) with various batch size.

E.3. Analysis on Various Learning Rates

Similarly to analysis on various batch size, we performed experiments on various learning rates with the naive method, the linear curve fitting, and our WNN. We also performed comparison experiments with a naive training approach with varying learning rates (1e-3, 1e-4, and 1e-5). Figure 13(a-d) illustrate that training with WNN shows the fastest convergence among the three approaches. Because WNN skips five training epochs by prediction of weights based on previous five epochs, it looks like a naive training with varying learning rates from a learning rate value to the larger one per every five epochs. Although the naive approach with varying learning rates shows a tendency similar to our WNN, oscillation occurs when the learning rate value increases as shown in Figure 13(e-h). In particular, in early training phase, training with larger learning rate performs faster convergence than our proposed method, but the WNN shows increasingly good convergence in the mid and the latter training phase. This tendency shows that our WNN produces weights more directly toward global minimum rather than the naive method.



(a) Train Loss, Learning Rate (b) Validation Loss, Learning Rate (c) Train Accuracy, Learning Rate (d) Validation Accuracy, Learning Rate

(e) Train Loss (f) Validation Loss (g) Train Accuracy (h) Validation Accuracy

Figure 13. Experimental results on Fashion MNIST (Unseen Dataset) with varying learning rate. We applied 2 times and 3 times of learning rate value per five epochs to compare with our WNN which predicts future weights per five epochs.

F. Analysis on Weight Prediction Error

F.1. Visualization of Weight Prediction Error

To verify efficacy of our WNN, we visualize predicted weights. Figure 14 represents samples of convolution weights and absolute errors. The left column is samples of target convolution weights to be predicted. The middle column describes absolute differences between the current weights and the target future weights and the right column represents those between the predicted weights and the target future weights. Namely, the darker the color, the smaller the error in the middle and right columns. On the other hand, the left column contains both positive and negative values, so it is illustrated with color. The outputs of WNN show less errors than weights without WNN. Figure 15 illustrates average forecasting error for 3×3 filters extracted from validation dataset for training WNN. From these two figures, it is shown that the learning-based approach is better than curve fitting-based approach. Figure 16 shows visualization examples of weight forecasting. The black dots are previous 5 weights inputted to the weight prediction process and each method predicts five steps ahead (i.e., 'x' denotes predicted values from each approach). The curve fitting approaches showed reasonable predictions for simple cases (Figure 16 (mid)), but were inaccurate for complex and nonlinear cases (Figure 16 (left)). On the contrary, WNN worked well on both simple and complex cases.

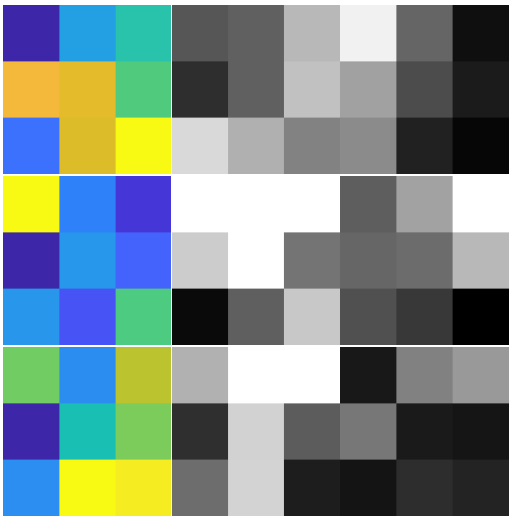
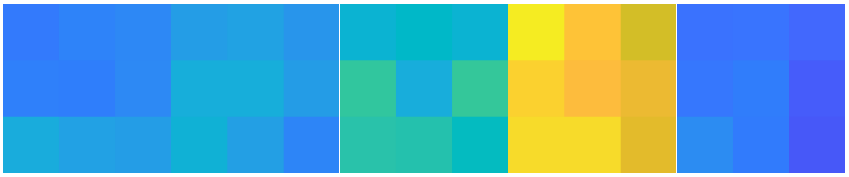


Figure 14. Weight samples and differences. The left column represents target future weights, the middle column describes absolute error between the current weights (without WNN) and the target future weights. The right column shows absolute error between predicted future weights (with WNN) and the target future weights.



(a) (b) (c) (d) (e)

Figure 15. Forecasting error between the last input weights and real future weights for 3×3 convolution filters. Note that blue color means lower error, and yellow means higher one. (a): w/o Forecasting, (b): Curve Fitting(Linear), (c): Curve Fitting(Exponential), (d): Curve Fitting(2nd Polynomial), and (e): WNN

F.2. Regression Error over Training Phase

It is known that training DNNs has a general tendency in phase analysis. In the beginning, training is generally unstable, so gradients are too steep and large changes in weight value occur too frequently. Training becomes more and more stable in the latter phase, so that gradients converge to zero and little change in weight value occurs. This tendency is generally observed in many cases. We tried to analyze our work from this point of view. Our WNN is applied repeatedly during

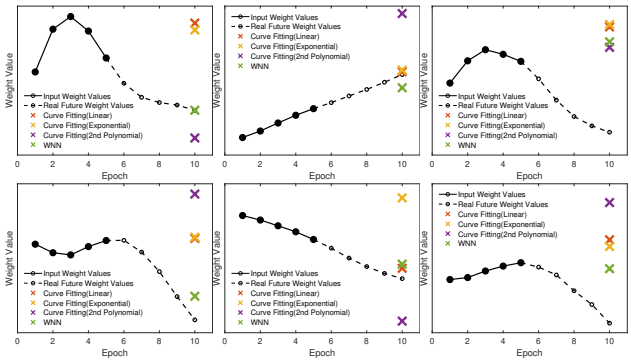
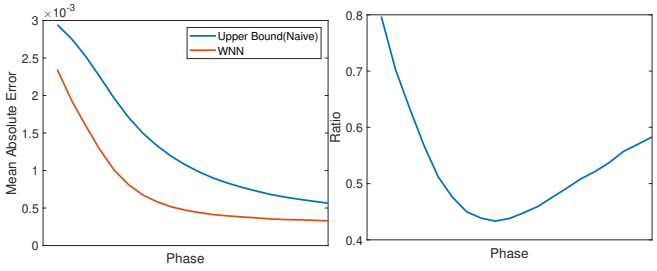


Figure 16. Examples of weight forecasting. Left: Successful cases for WNN, Middle: Easy cases for both Curve Fitting (Linear) and WNN, Right: Failure cases.



(a) MAE Comparison (b) Relative Comparison

Figure 17. Phase analysis of the proposed method. (a) represents mean absolute error between predicted weights and ground truth. (b) describes relative comparison between with and without our method.

training process, so we could compare effects of our method over training phase. Figure 17 (a) shows the effects of the proposed method during training. The blue line represents the naive training approach, so the mean absolute error (MAE) is differences between the current weight and the future weights without prediction. It can be an upper bound of MAE of the proposed method. At early phase, the training is unstable so it is hard to predict accurately. Then the training in mid-phase shows the biggest difference between MAE of our method and the upper bound. In addition, in the mid-term, training is mainly done, which results in a dramatic improvement in performance In other words, learning is stable in the mid-term and has a significant impact on performance. Our WNN, as shown in Figure 17 (a), allows to perform learning faster and more reliably. Figure 17 (b) is a graph for relative comparison through dividing the mae of the proposed method by one of the upper bound, so the smaller the value, the greater the difference. In the case of WNN-applied learning, it shows the best effect in the mid-term compared to the naive training.

G. Qualitative Results on Pose Estimation and Image Segmentation

Figure 18 shows qualitative results with and without WNN using the OpenPose model. The left column of images means input images to the model. The middle and right columns describe joint map resulted from the models without and with WNN respectively. Obviously, the predicted joint locations the model with WNN are more concentrated, which meansthe model with WNN is more converged than that without WNN. Furthermore, there are undetected joints in results from the model without WNN and the model with WNN shows robustness even when occluded with other objects.

Figure 18. Example outputs of OpenPose after training 100 Epochs without and with WNN. The left column describes input images. The middle and right columns show outputs of trained OpenPose model without WNN and with WNN respectively. Yellow circles describe improved results by WNN.

Figure 19 shows image segmentation results of DeepLabV3+ with and without WNN for qualitative visualization. Input images, outputs from DeepLabV3+ trained without WNN, and outputs from DeepLabV3+ trained with WNN are located from left to right columns. Each output is extracted at 50th train epoch from scratch. Output images from trained model without WNN (middle column) show mis-classified segmentation parts (yellow circles). On the contrary, output trained with WNN shows spatially consistent results with input images.

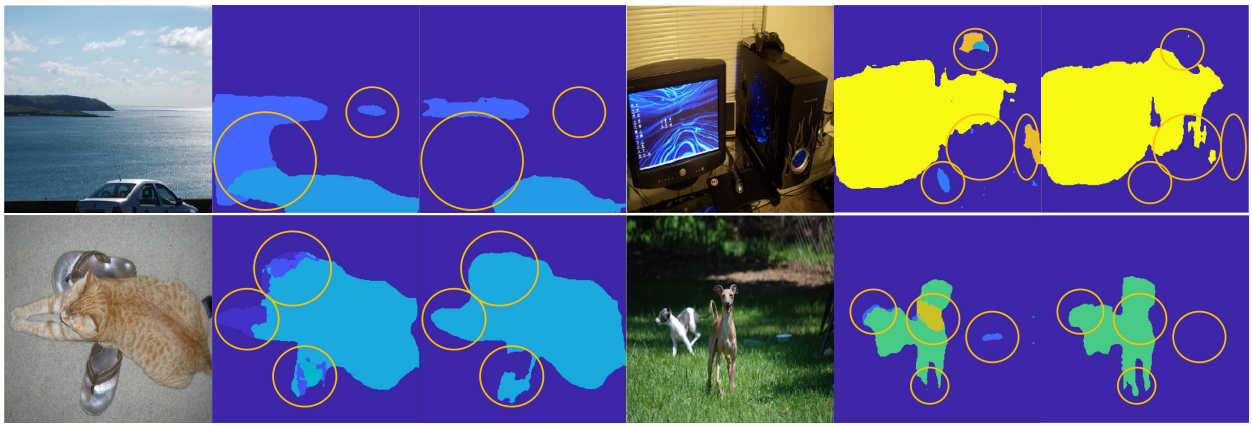
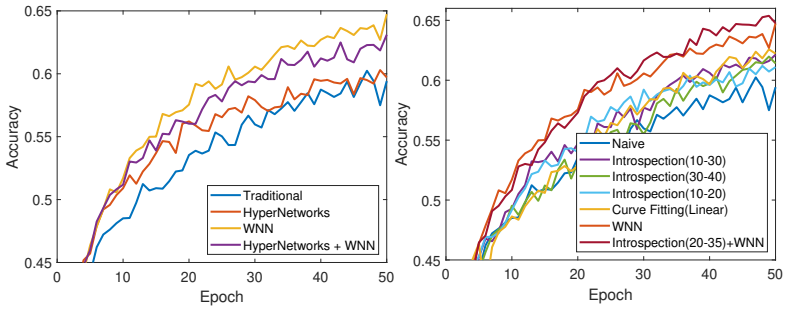


Figure 19. Example outputs of DeepLabV3+ after training 50 Epochs without and with WNN. The left column describes input images. The middle and right columns show outputs of trained DeepLabV3+ without WNN and with WNN respectively. Yellow circles describe improved results by WNN.

H. Algorithm Fusion using WNN with HyperNetworks and Introspection

Finally, we tried to fuse our WNN algorithm with similar approaches such as HyperNetworks and Introspection. Figure 20(a) shows validation accuracy of our WNN with HyperNetworks and Introspection when a Vanilla CNN is trained on CIFAR-10 dataset. In the case of HyperNetworks, training with the joint work (HyperNetworks + WNN) shows slower convergence than that with only WNN. As we mentioned in our main paper, HyperNetworks has limitations in representation power, resulting in performance drop.

Additionally, we performed training experiments with Introspection. Introspection forecasts mid-term future weights by analyzing global history of training from scratch to current epoch. On the other hand, our WNN periodically nowcasts near-future weights every 5 epoch when a target model is trained. Based on fundamental difference, we expected that periodic nowcast of our WNN constructs a new global tendency with global forecast of Introspection, resulting in faster convergence. As shown in Figure 20(b), WNN with Introspection outperforms the other methods when a Vanilla CNN is trained on CIFAR-10 dataset. This shows complementary relationship between WNN and Introspection, which improves training performance of a target network.



(a) WNN with HyperNetworks (b) WNN with Introspection

Figure 20. Experiments on algorithm fusion using WNN with HyperNetworks and Introspection. We trained a Vanilla CNN model on CIFAR10 dataset.

I. Detailed Training Recipes

I.1. Preliminary study

ResNet32 with [16,32,64] channels per each residual block and Vanilla CNN as target DNNs were used in our preliminary study. The Vanilla CNN has a conventional architecture, which consists of four convolution layers with [8,16,32,32] channels of 3×3 filters, a flattening layer, a fully-connected layer with 64 hidden neurons, and a classifier. For each convolution layer, we add a max pooling layer. Both ResNet and Vanilla CNN were trained by Adam optimizer with a fixed learning rate (LR) of 1e-3, 0.9 and 0.999 as two βs without LR scheduling, 1,024 batch size, simple data augmentations (random flip, rotation, translation) for totally 200 epochs. Also, we applied mean-subtraction and division by 255 on the CIFAR10 dataset as preprocessing. Then, this Vanilla CNN was reused in Section 5.2 and 5.3.

Section 3.1 (Table 1): we chose a starting epoch $Epoch_{start}$ and a target epoch $Epoch_{target}$ out of the 200 epochs and their corresponding weights (Θ_{start},Θ_{target}). We set a validation accuracy of Θ_{target} as a target accuracy, and the two networks were repeatedly retrained from Θ_{start} until reaching the target accuracy. After reaching the target accuracy, we calculated similarity between retrained weights and Θ_{target}. To compare with the fixed initialization from Θ_{start}, we also trained the two networks from random initialization.

Section 3.2 (Figure 2, 3, and Table 2): we added some noise to Θ_{start} as $\Theta_{start} + n$ and retrained from the $\Theta_{start} + n$ for simulating weight forecast with the error. Then, we measured direction and distance error using $\Theta_{start} + n$ and Θ_{target} as varying forecasting error and required epochs to reach the target accuracy from $\Theta_{start} + n$ and Θ_{start}. By analyzing those, we could find the relation between forecasting error and its corresponding speedup ratio.

Section 3.3 (Table 3): we applied three approaches, such as curve fitting, introspection, and WNN, with two strategies to real weight forecasting. Then, we measured direction and distance errors, and expected the speed up ratio based on the results of Figure 3.

Section 4 (Figure 6): we used VGG and MobileNetV2 for image classification, BERT for language modeling, and DDPG for RL. Details of the BERT, DDPG, and training recipe are the same with Section 6's.

I.2. Experiments in Section 5

CIFAR10: We trained a Vanilla CNN on CIFAR10 using Adam optimizer with a fixed learning rate of $1e-3$ for 50 epochs. For this experiment, Each batch was composed of 1,024 images, and random cropping, rotation, flip were used as data augmentation. Also, we applied mean-subtraction and division by 255 on the CIFAR10 dataset as preprocessing. The same recipe was applied to all the methods for fair comparison.

I.3. Experiments in Section 6

ImageNet: We trained MobileNetV2 with dropout for 60 epochs using Adam optimizer with exponential LR decay from $1e-2$ to $1e-5$, and 0.9, 0.999 as two βs. Each batch was composed of 64 images, and random cropping, rotation, flip, and color jittering were used as data augmentation. Also, we applied mean-subtraction and standard deviation normalization as preprocessing.

Image segmentation: DeepLabV3+ with a MobileNetV2 backbone was trained for 150 epochs from scratch on Pascal VOC 2012 dataset. As hyperparameters, we applied SGD with cosine LR from $3e-3$ to $1e-5$ with random flipping, scaling, blurring, and color jittering. Then, the batch size was 32 and objective function was cross entropy. We applied mean-subtraction and standard deviation normalization as preprocessing.

Pose estimation: OpenPose with ImageNet pre-trained VGG19 backbone was trained on MS COCO 2016 for 100 epochs. SGD with $1e-3$ LR, 0.9 momentum, 32 batch size without scheduling was used. As data augmentation, random rotation, flip, scaling and translation were adopted to minimize L2 loss. Also, we applied mean-subtraction and standard deviation normalization as preprocessing.

Language modeling: Universal transformer BERT were trained on WikiText2. For training 100 epochs, we used Adam optimizer with cosine learning decay from $2e-4$ to $1e-8$, 0.9 and 0.999 as two βs, 32 batch size, 256 max sequence length, and 512 dimensional word embedding. Then, masked LM loss with penalized confidence were used as an objective function.

Reinforcement learning: We trained Deep Deterministic Policy Gradient model which is composed of critic model to calculate Q-value, and actor model for predict action on Pendulum problem which is provided by PyGYM library. A sequence of fully-connected layers with [64,32,16] hidden neurons was used as the critic model, and a sequence of fully-connected layers with [32,32,32,16] hidden neurons was adopted as the actor model. Both were optimized by Adam optimizer with 0.9 and 0.999 as two βs for 200 episodes with 32 batch size. The LRs were $1e-4$ for critic, and $1e-3$ for actor.

Finetuning Pyramid Vision Transformer v2 (PVTv2): For finetuning, we applied resize on CIFAR100 images into 224 $\times$224 size, and preprocessing predefined by PVT V2. Adam optimizer with 0.9 and 0.999 as two βs for totally 50 epochs. In detail, it was composed of 5 epochs of burn-in (training only the randomly initialized classifier), 5 epochs of warm-up (training with small LR), 20 epochs with a fixed LR as $2.5e-3$, and 20 epochs with LR decay to $1e-4$. Also, we applied random rotation, flip, crop, color jittering as data augmentation, and 128 batch size.

Diffusion model We chose Denoising Diffusion Implicit Model on 64 $\times$ 64 images of Oxford Flower. AdamW with 0.9 and 0.999 as two βs, cosine LR decay from $1e-3$ to $1e-4$ during 60 epochs, and $1e-4$ weight decay was used as optimizer to minimize L1 loss. We set 64 as batch size, and adopted exponential moving averages. As preprocessing, we divided each image into 127.5 and subtracted 1.0.