Meta-Learning with Hypernetworks

Slides by: Paul Vicol

Overview

- A *hypernetwork* is a neural network that outputs the weights of another neural network (often called the *target network*)

- Hypernetworks are a useful means to solve bi-level optimization problems, as well as other “meta-learning” type tasks:
    - Hyperparameter optimization
    - Multi-objective optimization (MOO)
    - Continual learning
    - Efficient training of NN ensembles

Hyperparameter Optimization

Lorraine & Duvenaud, "Stochastic Hyperparameter Optimization through Hypernetworks." 2018
Mackay*, Vicol*, et al., "Self-Tuning Networks." ICLR 2019

Hyperparameter Optimization as Bilevel Optimization

- Hyperparameter optimization is a *bilevel optimization problem*:

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{\text{val}}(\lambda, \mathbf{w}^*) \qquad \text{subject to} \qquad \mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}_{\text{train}}(\lambda, \mathbf{w})$$

Outer loop over hyperparameters

Inner loop to optimize model parameters

```
while True:
    hparam = get_hyperparameter_value()
    W = init_weights()

    while not converged:
        W = gradient_step(W, hparam)
```

Gradient-Based Approaches to HO

Bilevel Problem: $$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\mathbf{w}^*(\lambda)})$$

$\mathbf{w}^*(\lambda)$ "Best-response" function

Goal: $$\lambda \leftarrow \lambda - \alpha \boxed{\nabla_\lambda \mathcal{L}_{val}}$$

$$\frac{d\mathcal{L}_{val}(\lambda, \mathbf{w}^*(\lambda))}{d\lambda} = \frac{\partial \mathcal{L}_{val}(\lambda, \mathbf{w}^*(\lambda))}{\partial \lambda} + \frac{\partial \mathcal{L}_{val}(\lambda, \mathbf{w}^*(\lambda))}{\partial \mathbf{w}^*(\lambda)} \boxed{\frac{\partial \mathbf{w}^*(\lambda)}{\partial \lambda}}$$

How the weights change in response to a small change in the hyperparameters

Approximating the *Best-Response Function*

- The *“best-response” function* maps hyperparameters to optimal weights on the training set:

$$\mathbf{w}^*(\lambda) = \arg\min_{\mathbf{w}} \mathcal{L}_{\text{train}}(\lambda, \mathbf{w})$$

- **Idea:** *Learn a parametric approximation* $\hat{\mathbf{w}}_\phi$ to the best-response function $\hat{\mathbf{w}}_\phi \approx \mathbf{w}^*$

- **Advantages:**
  - Since $\hat{\mathbf{w}}_\phi$ is differentiable, we can use *gradient-based optimization* to update the hyperparameters
  - By training $\hat{\mathbf{w}}_\phi$ we *do not need to re-train models from scratch*; the computational effort needed to fit $\hat{\mathbf{w}}_\phi$ around each hyperparameter is not wasted

Approximating the *Best-Response Function*

- Update the approximation parameters ϕ using the *chain rule*:

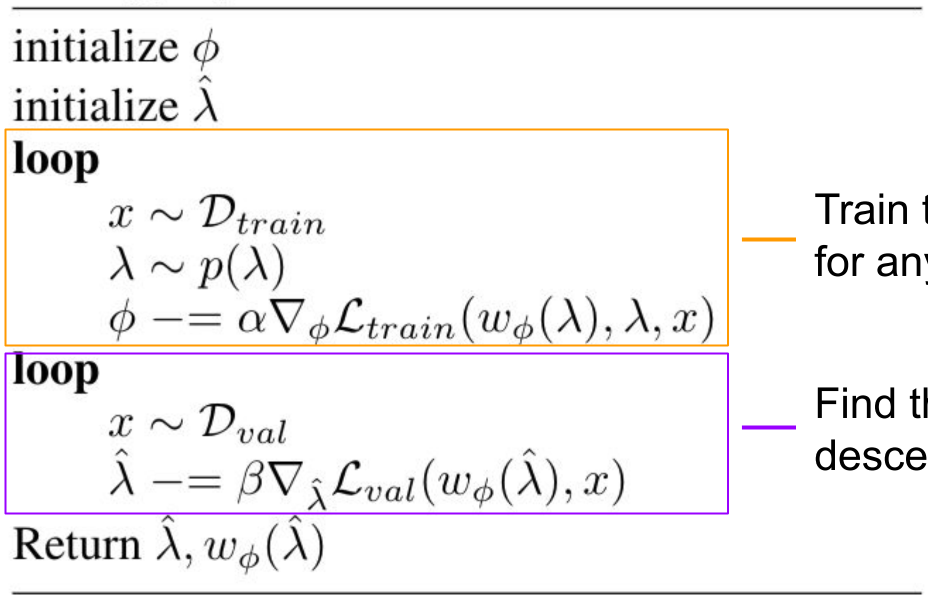
$$\frac{\partial \mathcal{L}_{\text{train}}(\hat{\mathbf{w}}_\phi)}{\partial \hat{\mathbf{w}}_\phi} \frac{\partial \hat{\mathbf{w}}_\phi}{\partial \phi}$$

- Update the hyperparameters using the *validation loss gradient*:

$$\frac{\partial \mathcal{L}_{\text{val}}(\hat{\mathbf{w}}_\phi(\lambda))}{\partial \hat{\mathbf{w}}_\phi(\lambda)} \frac{\partial \hat{\mathbf{w}}_\phi(\lambda)}{\partial \lambda}$$

Hypernetwork-Based Approaches to HO

Global Best-Response Approximation



```
initialize $\phi$
initialize $\hat{\lambda}$
loop
    $x \sim \mathcal{D}_{train}$
    $\lambda \sim p(\lambda)$
    $\phi \mathrel{-}= \alpha \nabla_{\phi} \mathcal{L}_{train}(w_{\phi}(\lambda), \lambda, x)$
loop
    $x \sim \mathcal{D}_{val}$
    $\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{val}(w_{\phi}(\hat{\lambda}), x)$
Return $\hat{\lambda}, w_{\phi}(\hat{\lambda})$
```

Train the hypernetwork to produce good weights for any hyperparameter $\lambda \sim p(\lambda)$

Find the optimal hyperparameters via gradient descent on $\mathcal{L}_{\text{val}}$

Scalability Challenges

- *Two core challenges to scale this approach* to large networks:

1. Intractable to model $\hat{\mathbf{w}}_\phi(\lambda)$ *over the entire hyperparameter space*, e.g., the support of $p(\lambda)$

   ➡ **Solution:** Approximate the best-response *locally* in a neighborhood around the current hyperparameter value

2. Difficult to learn a mapping $\lambda \to \mathbf{w}$ when $\mathbf{w}$ are the weights of a large network

   ➡ **Solution:** STNs introduce a *compact* approximation to the best-response by *modulating activations based on the hyperparameters*

Locally Approximating the Best-Response

- *Jointly optimize* the hypernetwork parameters and the hyperparameters by *alternating gradient steps on the training and validation sets*

Local Best-Response Approximation

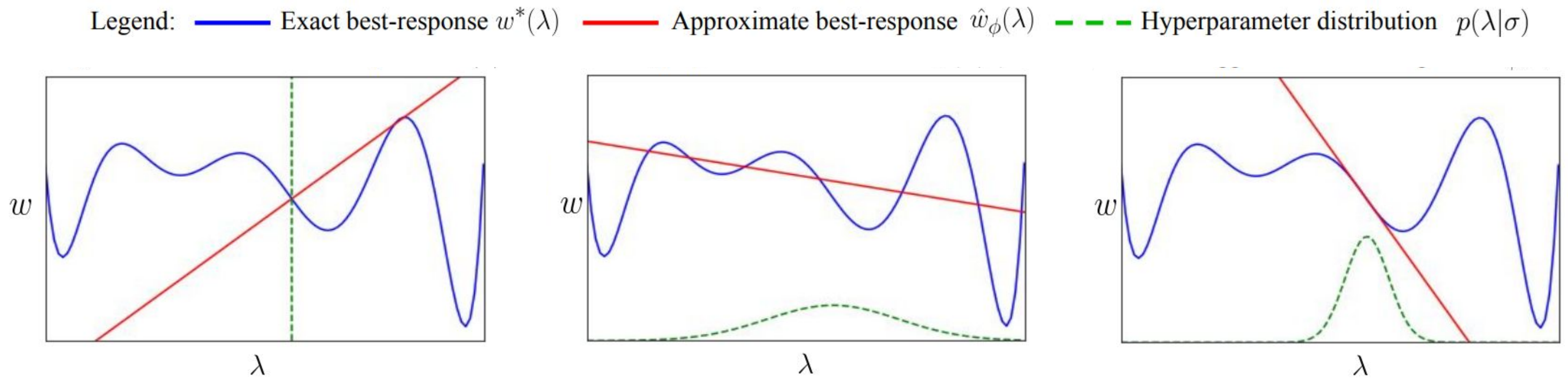
```
initialize $\phi$
initialize $\hat{\lambda}$
**loop**
    $x \sim \mathcal{D}_{train}$
    $\lambda \sim p(\lambda \mid \hat{\lambda})$
    $\phi \mathrel{-}= \alpha \nabla_{\phi} \mathcal{L}_{train}(w_{\phi}(\lambda), \lambda, x)$

    $x \sim \mathcal{D}_{val}$
    $\hat{\lambda} \mathrel{-}= \beta \nabla_{\hat{\lambda}} \mathcal{L}_{val}(w_{\phi}(\hat{\lambda}), x)$
Return $\hat{\lambda}, w_{\phi}(\hat{\lambda})$
```

Train the hypernet to produce good weights *around the current hyperparameter* $\lambda \sim p(\lambda \mid \hat{\lambda})$

Update the hyperparameters using the local best-response approximation

Effect of the Sampling Distribution



Too small	*Too wide*	*Just right*
The hypernetwork will match the best-response at the current hyperparameter, but may not be locally correct	The hypernetwork may be insufficiently flexible to model the best-response, and the gradients will not match	The gradient of the approximation will match that of the best-response

Compact Best-Response Approximation

- Naively representing the mapping $\lambda \to \mathbf{w}$ is intractable when $\mathbf{w}$ is high-dimensional

- We propose an architecture that computes the usual elementary weight/bias, plus an additional weight/bias that is scaled by a linear transformation of the hyperparameters:

$$\hat{\mathbf{W}}_\phi(\lambda) = \mathbf{W}_{\text{elem}} + (\mathbf{V}\lambda) \odot_{\text{row}} \mathbf{W}_{\text{hyper}}$$

$$\hat{\boldsymbol{b}}_\phi(\lambda) = \boldsymbol{b}_{\text{elem}} + (\mathbf{C}\lambda) \odot \boldsymbol{b}_{\text{hyper}}$$

- *Memory-efficient*: roughly 2x number of parameters and scales well to high dimensions

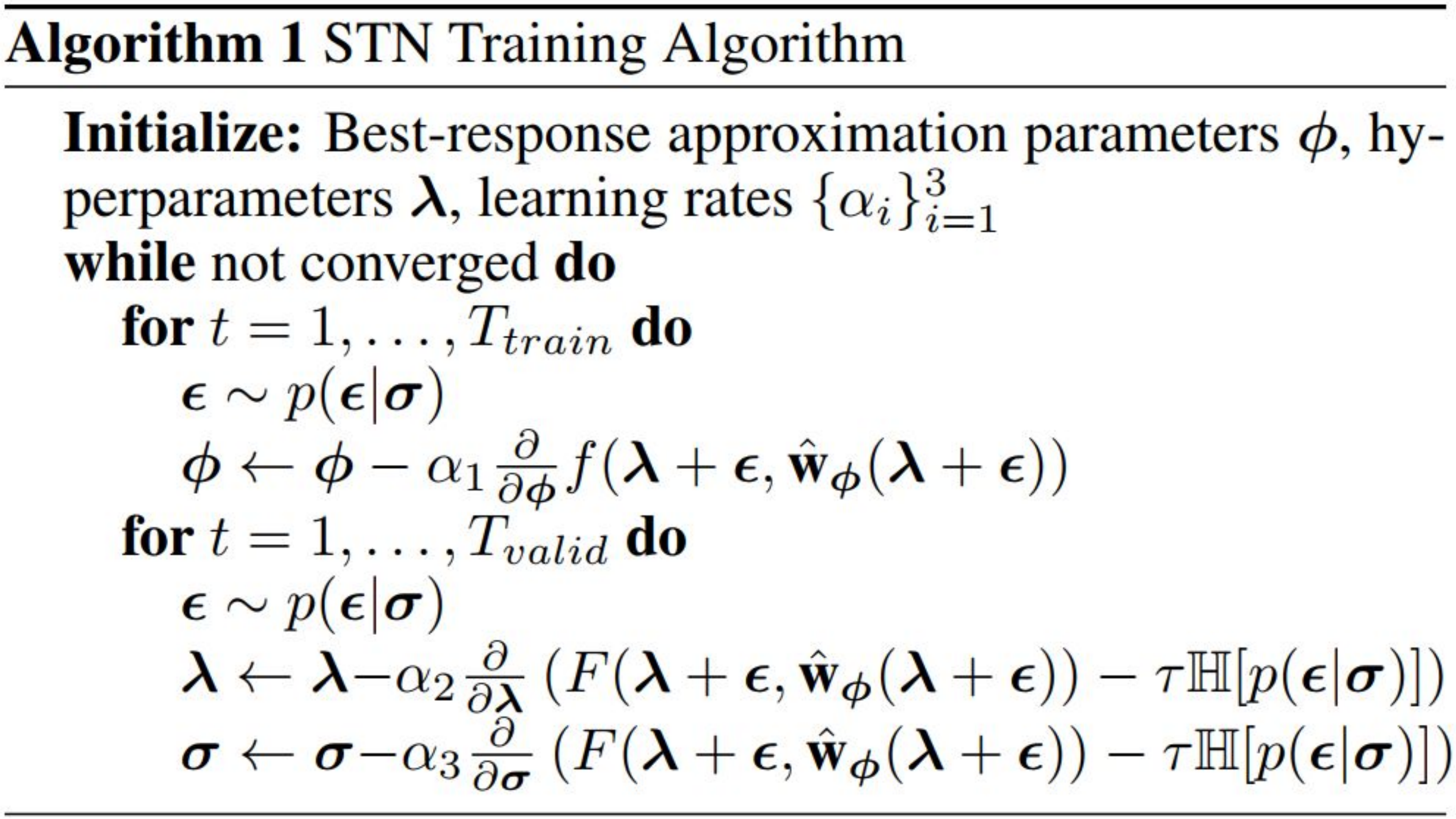
Compact Best-Response Approximation

- This architecture can be interpreted as *directly operating on the pre-activations of the layer*, and *adding a correction to account for the hyperparameters*:

$$\hat{\mathbf{W}}_\phi(\lambda)\boldsymbol{x} + \hat{\boldsymbol{b}}_\phi(\lambda) = \underbrace{[\mathbf{W}_{\text{elem}}\boldsymbol{x} + \boldsymbol{b}_{\text{elem}}]}_{\text{Usual computation of Linear layer}} + \underbrace{[(\mathbf{V}\lambda) \odot_{\text{row}} (\mathbf{W}_{\text{hyper}}\boldsymbol{x}) + (\mathbf{C}\lambda \odot \boldsymbol{b}_{\text{hyper}})]}_{\text{Correction term to account for the hyperparameters}}$$

- **Sample-efficient:** since the predictions can be computed by transforming pre-activations, the *hyperparameters for different examples in a mini-batch can be perturbed independently*
  - E.g., a different dropout rate for each example

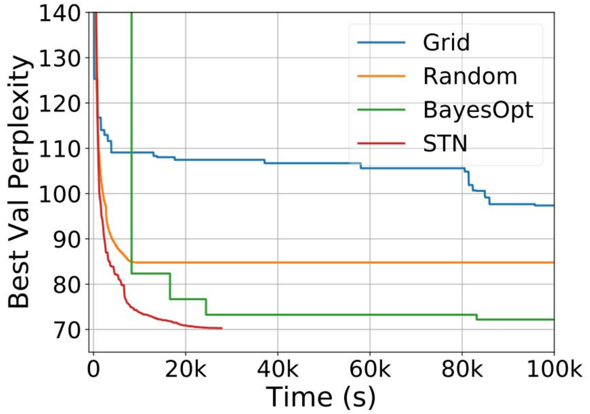
STN Algorithm



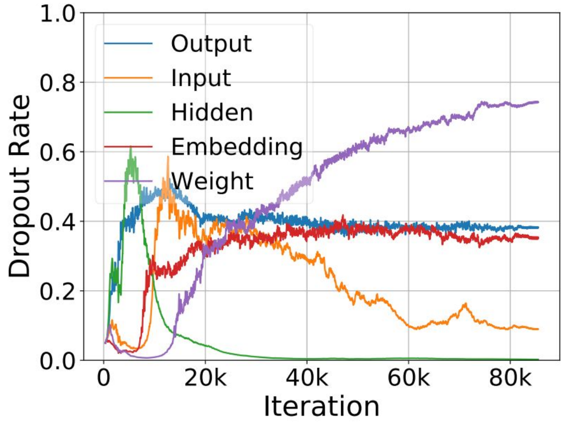
Algorithm 1 STN Training Algorithm

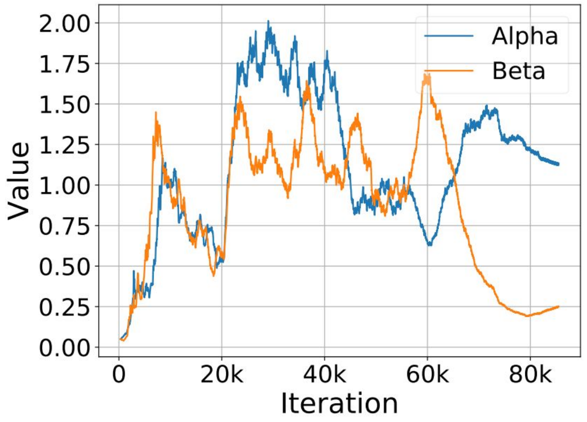
```
Initialize: Best-response approximation parameters $\boldsymbol{\phi}$, hyperparameters $\boldsymbol{\lambda}$, learning rates $\{\alpha_i\}_{i=1}^{3}$
while not converged do
    for $t = 1, \dots, T_{train}$ do
        $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
        $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \alpha_1 \frac{\partial}{\partial \boldsymbol{\phi}} f(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon}))$
    for $t = 1, \dots, T_{valid}$ do
        $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
        $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha_2 \frac{\partial}{\partial \boldsymbol{\lambda}} (F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})])$
        $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} - \alpha_3 \frac{\partial}{\partial \boldsymbol{\sigma}} (F(\boldsymbol{\lambda} + \boldsymbol{\epsilon}, \hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda} + \boldsymbol{\epsilon})) - \tau \mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})])$
```

STN - LSTM Experiments



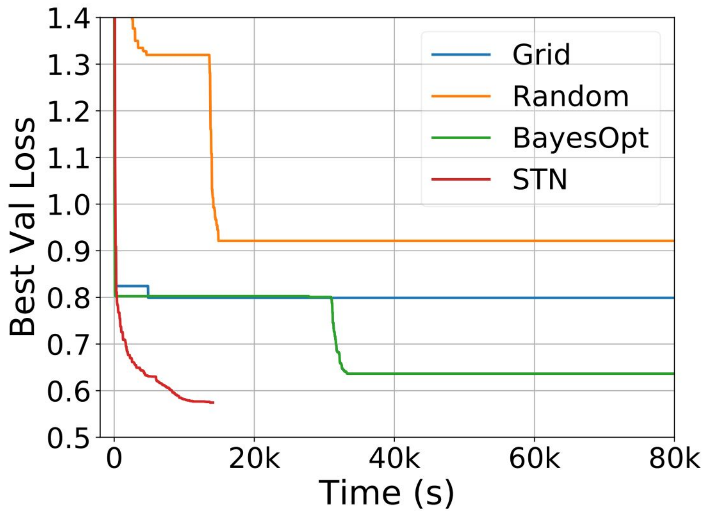
	PTB	
Method	**Val Perplexity**	**Test Perplexity**
Grid Search	97.32	94.58
Random Search	84.81	81.46
Bayesian Optimization	72.13	69.29
STN	**70.30**	**67.68**





STN - CNN Experiment Results

	CIFAR-10	
Method	**Val Loss**	**Test Loss**
Grid Search	0.794	0.809
Random Search	0.921	0.752
Bayesian Optimization	0.636	0.651
STN	**0.575**	**0.576**



- Again, STNs substantially outperform grid/random search and BayesOpt
  - Achieve *lower validation loss than BayesOpt in < ¼ the time*

What *can* we and what *can’t* we tune?

What can we tune?

- STNs can tune *most regularization hyperparameters* including
  - Dropout
  - Continuous data augmentation hyperparameters (hue, saturation, contrast, etc.)
  - Discrete data augmentation hyperparameters (# and length of cutout holes)

What can’t we tune?

- Because we collapsed the bilevel problem into a single-level one, *there is no inner training loop*

➡ We cannot tune inner optimization hyperparameters like *learning rates*

Gradient-Based Approaches to HO

Implicit Differentiation

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\frac{\partial \mathcal{L}_{train}(\lambda, \mathbf{w})}{\partial \mathbf{w}} = 0})$$

- Assuming *training has converged*, we can use the *implicit function theorem*

$$\frac{d\mathbf{w}(\lambda)}{d\lambda} = -\left(\frac{\partial^2 \mathcal{L}_{train}}{\partial \mathbf{w}^2}\right)^{-1} \frac{\partial^2 \mathcal{L}_{train}}{\partial \lambda \partial \mathbf{w}}$$

- *Expensive:* Solving the linear system with CG requires Hessian-vector products

Iterative Differentiation

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\text{Backprop through optimization steps}})$$

- Use autodiff to *backprop through training*
- Full optimization procedure or a truncated version of it

- *Expensive* when the number of gradient steps increases

Hypernet-Based

$$\lambda^* = \arg\min_{\lambda} \mathcal{L}_{val}(\lambda, \underbrace{\arg\min_{\mathbf{w}} \mathcal{L}_{train}(\lambda, \mathbf{w})}_{\hat{\mathbf{w}}_{\phi}(\lambda) \approx \mathbf{w}^*(\lambda)})$$

- Learn a hypernetwork $\hat{\mathbf{w}}_{\phi}(\lambda) \approx \mathbf{w}^*(\lambda)$ parameterized by ϕ to *map hyperparameters to network weights*

- Does not require differentiating through optimization
- Efficient, can also optimize discrete & stochastic hyperparameters

Multi-Objective Optimization

Navon et al., "Learning the Pareto Front with Hypernetworks." ICLR 2021.

Multi-Objective Optimization

- Multi-objective optimization problems are prevalent in ML
- **Constrained problems:** learn a single task while finding solutions that *satisfy certain properties, like fairness or privacy*
  - Usually by optimizing a main task with *auxiliary loss terms* to encourage the additional properties (e.g., fairness)
- The *set of optimal solutions* to a multi-objective problem is called the *Pareto front*
  - Each point on the front represents a different trade-off between possibly conflicting objectives.

The Pareto Front

- In many cases, we are interested in *more than one predefined preference*
  - Either because the trade-off is not known prior to training, OR
  - Because there are many possible trade-offs of interest
- **Pareto front learning (PFL):** design a model that can be applied *at inference time to any given preference direction*
- **Naive approach:** run a single-preference optimization multiple times
  - Scalability issue: number of models to be trained to cover the objective space grows exponentially with the number of objectives (computational cost), and we need to store all trained models in memory to switch between them later on (memory cost)
- **Hypernet approach:** learn a mapping from preference → network weights

Multi-Objective Optimization

Typical Approaches to MOO

- A *separate model* has to be trained for each point on the Pareto front
- The trade-off between objectives must be *specified a-priori*

Pareto Hypernetworks (PHN)

- Learns the *entire Pareto front simultaneously using a single hypernet*
- *Runtime efficient* compared to training multiple models
- Generalizes to new preference vectors

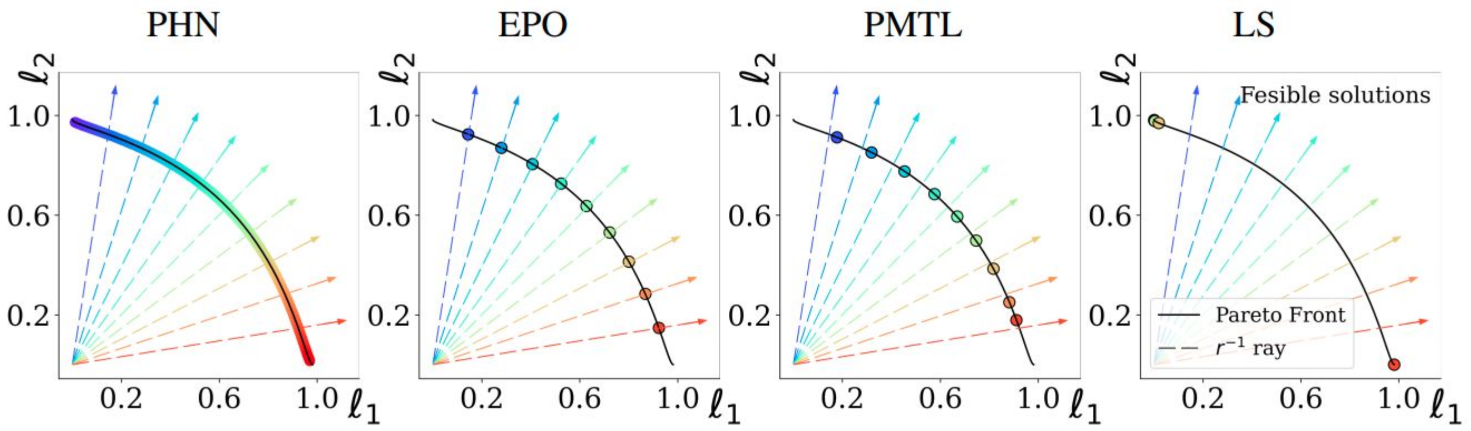


Figure 1: The relation between Pareto front, preference rays and solutions. Pareto front (black solid line) for a 2D loss space and several rays (colored dashed lines) which represent various possible preferences. **Left:** A single PHN model converges to the entire Pareto front, mapping any given preference ray to its corresponding solution on the front. See details in Appendix B.

Some Classic Approaches for MOO

- An MOO is defined by a *vector of m losses*: $\boldsymbol{\ell} = \begin{bmatrix} \ell_1 & \ell_2 & \cdots & \ell_m \end{bmatrix}$

- **Linear Scalarization (LS):**
  - Most straightforward approach to MOO, where we define a single loss based on a vector of weights for each loss term, $\mathbf{r}$:

$$\ell_{\boldsymbol{r}}(\theta) = \textstyle\sum_i r_i \ell_i(\theta)$$

- **Exact Pareto Optimal (EPO):** a more advanced MOO method that can converge to a desired ray in loss space

Pareto Hypernetworks

- *"Pareto hypernetworks" (PHNs)* are trained similarly to fitting a hypernet globally on the hyperparameter space---here the "hyperparameters" are preferences over loss terms

$$\mathbf{r} = (r_1, \ldots, r_m) \in \mathbb{R}^m \quad \text{s.t.} \quad \sum_j r_j = 1, r_j \geq 0$$

Algorithm 1 PHN

```
while not converged do
    $\boldsymbol{r} = Dir(\boldsymbol{\alpha})$
    $\phi(\theta, \boldsymbol{r}) = h(\theta, \boldsymbol{r})$
    Sample mini-batch $(x_1, y_1), .., (x_B, y_B)$
    if LS then
        $g_\theta \leftarrow \frac{1}{B} \sum_{i,j} r_i \nabla_\theta \ell_i(x_j, y_j, \phi(\theta, \boldsymbol{r}))$
    if EPO then
        $\beta = EPO(\phi(\theta, \boldsymbol{r}), \boldsymbol{\ell}, \boldsymbol{r})$
        $g_\theta \leftarrow \frac{1}{B} \sum_{i,j} \beta_i \nabla_\theta \ell_i(x_j, y_j, \phi(\theta, \boldsymbol{r}))$
    $\theta \leftarrow \theta - \eta g_\theta$
return $\theta$
```

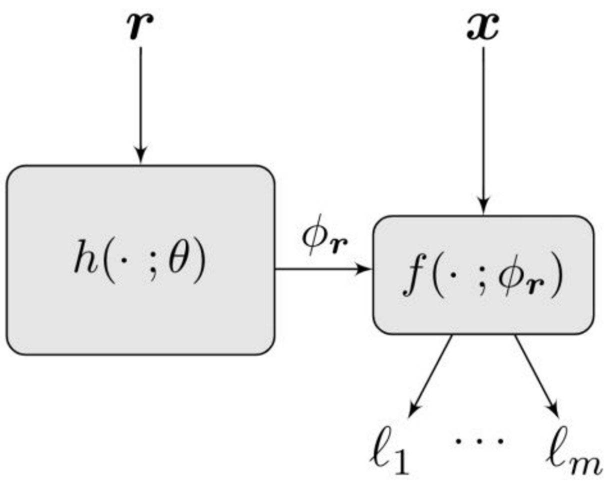
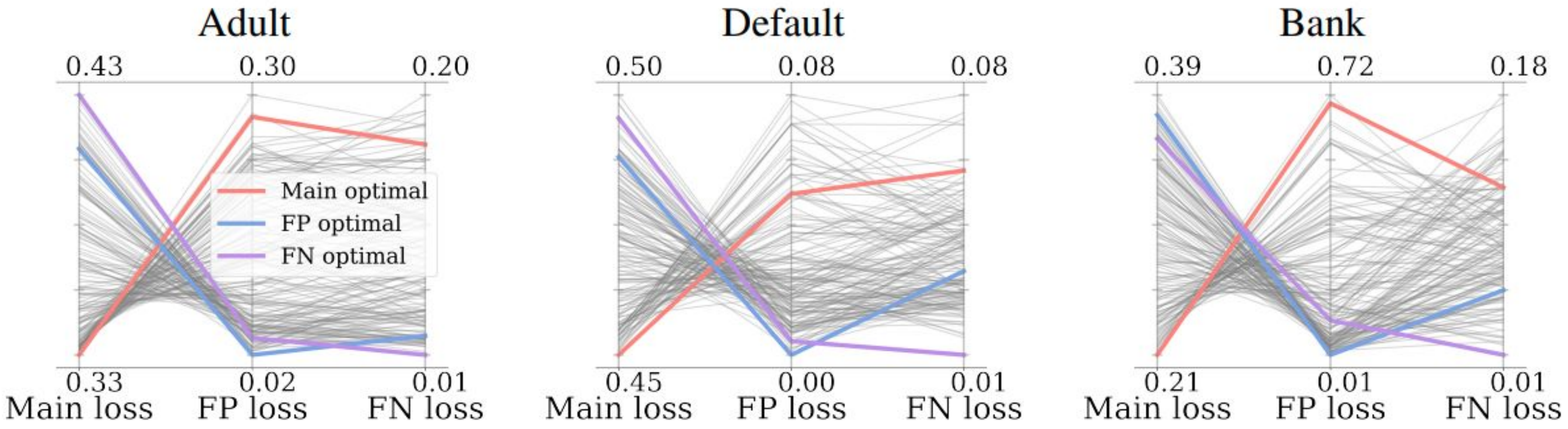


Figure 2: PHN receives an input preference and outputs the corresponding model weights.

PHN: Fairness

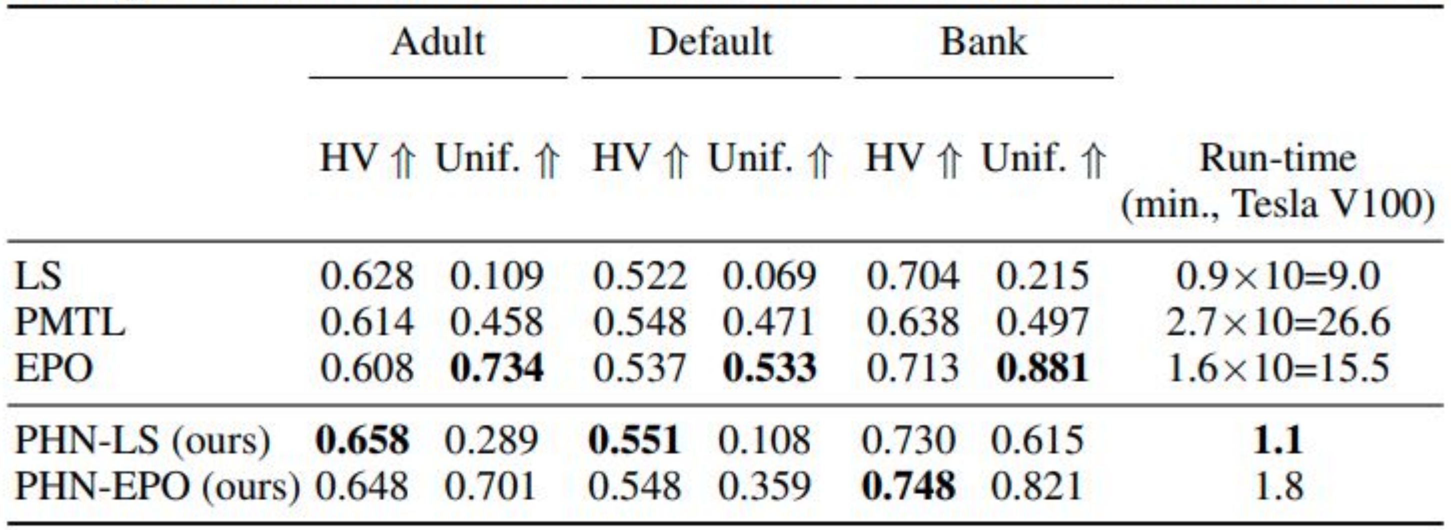
- 3-objective optimization problem with a classification objective and two fairness objectives: False-Positive (FP) fairness and False Negative (FN) fairness



PHN: Fairness

- 3-objective optimization problem with a classification objective and two fairness objectives: False-Positive (FP) fairness and False Negative (FN) fairness

Table 2: Comparison on three Fairness datasets. Run-time is evaluated on the Adult dataset.



	Adult		Default		Bank		
	HV $\Uparrow$	Unif. $\Uparrow$	HV $\Uparrow$	Unif. $\Uparrow$	HV $\Uparrow$	Unif. $\Uparrow$	Run-time (min., Tesla V100)
LS	0.628	0.109	0.522	0.069	0.704	0.215	0.9×10=9.0
PMTL	0.614	0.458	0.548	0.471	0.638	0.497	2.7×10=26.6
EPO	0.608	**0.734**	0.537	**0.533**	0.713	**0.881**	1.6×10=15.5
PHN-LS (ours)	**0.658**	0.289	**0.551**	0.108	0.730	0.615	**1.1**
PHN-EPO (ours)	0.648	0.701	0.548	0.359	**0.748**	0.821	1.8

Continual Learning

Oswald et al., “Continual Learning with Hypernetworks.” ICLR 2020.

The Continual Learning Problem

$$\{(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{(T)}, \mathbf{Y}^{(T)})\}$$

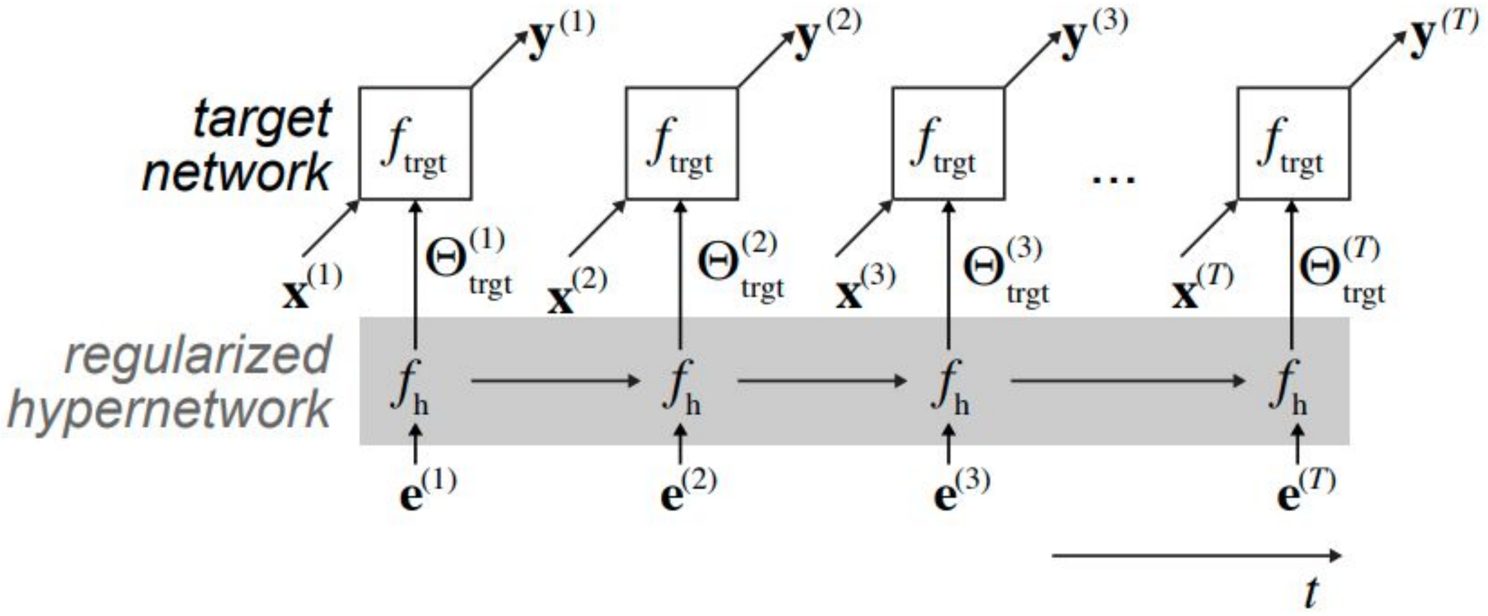
- In continual learning, we want to learn a new task $(\mathbf{X}^{(t)}, \mathbf{Y}^{(t)})$ starting with weights $\Theta^{(t-1)}$ the goal is to find new parameters $\Theta^{(t)}$ such that:

  1. It *retains or improves performance on the previous tasks*, and
  2. It *solves the new task*

- Key idea of this work: *address catastrophic forgetting at the meta-level*
  - Instead of retaining performance of a model on the previous data when it sees a new task, *get a hypernet to remember how to output the appropriate target network weights for each task*

Task-Conditioned Hypernets

- *Task-conditioned hypernet:* generate weights based on task identity
- A fixed embedding e^{(i)} is learned for each task



Rehearsing

- One approach to avoid catastrophic forgetting:
  1. Store data from previous tasks and the corresponding model outputs
  2. *Regularize the output* of the updated model so that its *predictions agree with the previous model* (e.g., before the new task):

$$\mathcal{L}_{\text{output}} = \sum_{t=1}^{T-1} \sum_{i=1}^{|\mathbf{X}^{(t)}|} \|f(\mathbf{x}^{(t,i)}, \Theta^*) - f(\mathbf{x}^{(t,i)}, \Theta)\|^2$$

- This is called **rehearsing**, and *requires storing and iterating over previous data*
- *Memory expensive and not strictly online learning!*

Hypernetwork Output Regularization

- First, compute the candidate change $\Delta\Theta_h$ that minimizes the task loss:

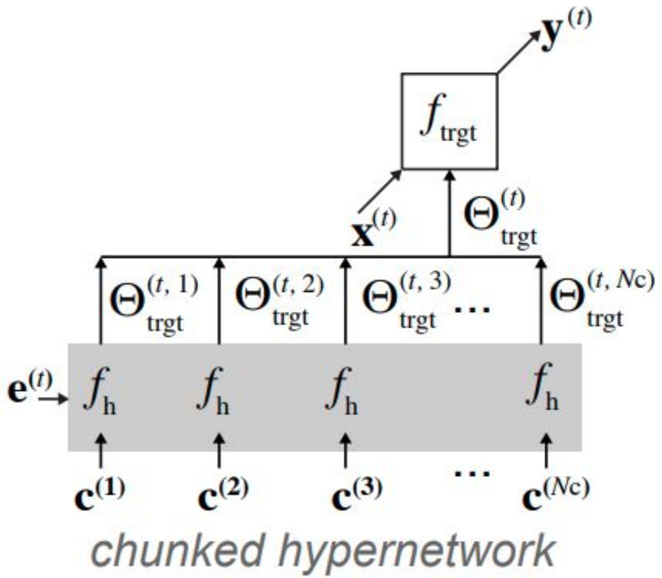
$$\mathcal{L}_{\text{task}}^{(T)} = \mathcal{L}_{\text{task}}(\Theta_{\text{h}}, \mathbf{e}^{(T)}, \mathbf{X}^{(T)}, \mathbf{Y}^{(T)})$$

- Then, *add a hypernet output regularizer* that ensures that the hypernet continues to output the same weights for old tasks that it previously learned:

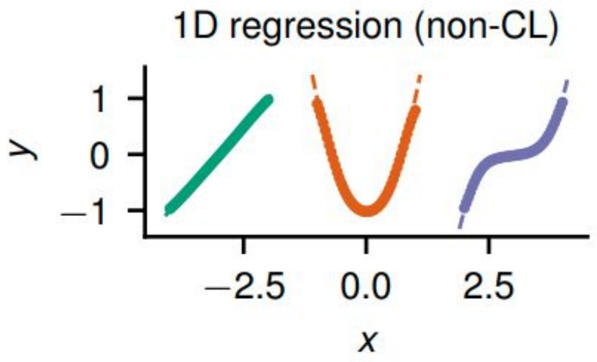
$$\begin{aligned}\mathcal{L}_{\text{total}} &= \mathcal{L}_{\text{task}}(\Theta_{\text{h}}, \mathbf{e}^{(T)}, \mathbf{X}^{(T)}, \mathbf{Y}^{(T)}) + \boxed{\mathcal{L}_{\text{output}}(\Theta_{\text{h}}^*, \Theta_{\text{h}}, \Delta\Theta_{\text{h}}, \{\mathbf{e}^{(t)}\})} \\ &= \mathcal{L}_{\text{task}}(\Theta_{\text{h}}, \mathbf{e}^{(T)}, \mathbf{X}^{(T)}, \mathbf{Y}^{(T)}) + \frac{\beta_{\text{output}}}{T-1} \sum_{t=1}^{T-1} \| f_{\text{h}}(\mathbf{e}^{(t)}, \Theta_{\text{h}}^*) - f_{\text{h}}(\mathbf{e}^{(t)}, \Theta_{\text{h}} + \Delta\Theta_{\text{h}})) \|^2\end{aligned}$$

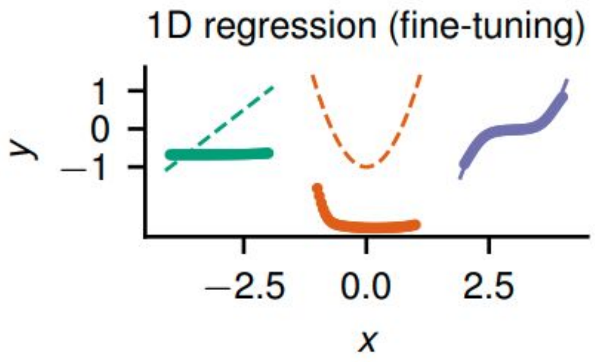
Slight Tangent: Chunked Hypernet Parameterization

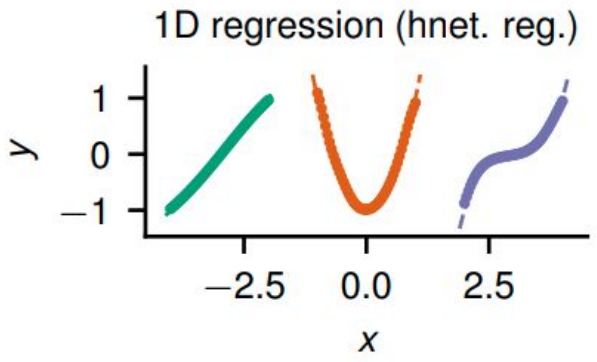
- For scaling to larger networks, they use a *chunked hypernetwork*
- Outputs a chunk of network weights (e.g., a layer) at a time, conditioned on the task embedding e^{(i)} and a particular chunk embedding c^{(k)}
  - Chunk embeddings are specific to each layer, but shared across all tasks.
  - So the hypernet learns to map (e^{(i)}, c^{(k)}) → weights for layer k of the model for task e^{(i)}
- Here, the number of learned hypernet params is *smaller than the number of target net params*



Experiment: 1D Regression







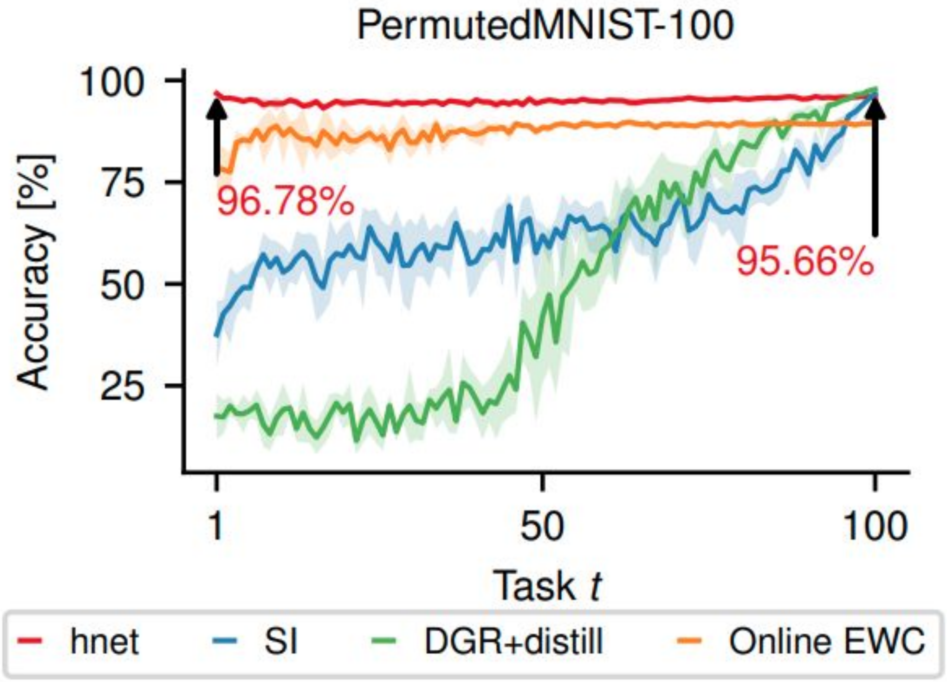
Fitting all polynomials simultaneously

Training on each polynomial sequentially causes forgetting of previous tasks

Using the hypernet with the memory-preserving regularizer retains performance on old tasks

Experiment: Permuted MNIST

- Each task is a *classification problem on a fixed random permutation* of MNIST pixels
- Low similarity between generated tasks → pMNIST studies the memory capacity of a continual learner



Q/A