Learning to learn by gradient descent as a discrete-time finite-horizon Markov decision process

A thesis submitted by

Matthew Stachyra

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Tufts University

February 2024

Advisor: Jivko Sinapov

Abstract

Meta-learning, often referred to as "learning to learn," is a paradigm within machine learning that focuses on developing models that generalize knowledge across tasks. Such a model learns across tasks as it learns within tasks, enabling the model to adapt quickly to new, unseen tasks. A model for a new task can be generated with near-optimal parameters in no or very few training steps. Meta-learning in machine learning is desirable in cases where there is limited data for a task or when a model needs to learn efficiently within few data samples. This thesis investigates whether Meta-learning can be cast as a Reinforcement learning (RL) problem for non-RL tasks. RL applied to Meta-learning has included hyperparameter search for neural architectures and meta-policies in RL task domains (Meta-Reinforcement Learning). RL has not been used to train models (parameter search) for non-RL tasks in the Meta-learning paradigm. This thesis asks: can Meta-learning be formulated as a sequential task? And if Meta-learning can be formulated as a sequential task, can Meta-learning be a discrete-time finite-horizon MDP solved by deep RL? Reinforcement Meta-Learning (REML) is proposed to formu-

late the Meta-learning task as composing neural networks layer by layer for sampled tasks with a shared parameter space across tasks. REML is evaluated according to the protocol proposed by Finn et al (2018) as used for the Model-agonistic meta-learning (MAML) algorithm. REML is first tested on regression tasks as a series of varying sinusoidal curves. Performance is evaluated looking at loss per step (convergence speed) for a model constructed by REML as compared to a model trained from scratch. REML is also tested on a few-shot learning task where it is given 5 and 10 samples and tested after 0, 1, and 10 gradient steps. Future work is to continue testing REML on classification and reinforcement learning tasks, and expanding the action space of REML to decide hyperparameters in addition to parameters.

Acknowledgements

Thank you to my parents for supporting me. You encouraged me to keep going and see where it led. You gave me the confidence to keep trying.

Thank you to Daniel Fuenmayor for being the first person to believe in this thesis. You indulged my ideas before they took shape.

Thank you to Michael Joseph for your eternal skepticism. You helped me think more clearly.

Contents

Chapter 1

Introduction

A machine learning model of a function is constrained by the available data in the function's domain. Limited data restricts the ability of the model to represent the target function at unknown points. Trying to learn under this constraint is known as few shot learning. One paradigm applied to manage this constraint is meta-learning or "learning to learn". Meta-learning has been defined in a number of ways, with an early definition proposed by Thrun et al. (1998): a model is learning to learn if its performance for each task (where there is more than 1 task, performance measure, and training experience) improves with the number of tasks and experiences [1]. Models trained to learn to learn generalize knowledge across task experiences, and leverage this general representation as a starting point for new tasks. The meta-learning model can generate a model for a new task with few data points and few training steps.

This thesis investigates whether Meta-learning can be cast as a Reinforcement learning (RL) problem for any task domain. There have been relatively fewer applications of RL to Meta-learning; more research focuses on solving RL tasks using the Meta-learning approach. This thesis seeks to implement RL as the meta-learner in a model agnostic manner akin to MAML (Model agnostic meta-learning) proposed by Finn et al 2018 [2]. Related research at the intersection of RL and Meta-learning include RL-driven hyperparameter search, where a recurrent policy is used to identify the most performant set of hyperparameters for a task. Another area of research was Meta Reinforcement Learning or RL^2, where a policy adapts to different tasks. This thesis proposes *Reinforcement* Meta Learning or REML, where a policy guides Meta learning for tasks outside the RL domain (i.e., is not an MDP solving another MDP). The hypothesis is an RL agent can generate a general representation across tasks in one policy, where the differences across tasks in a domain are understood as different states of the same changing environment rather than different, individual tasks.

The initial question is: can Meta-learning be formulated as a sequential task? If Meta-learning can be formulated as a sequential task, then Meta-learning as a task can be approached by a discrete-time finite-horizon MDP. REML proposes formulating REML as a task where the policy composing a neural network layer by layer. Each layer added to a target network is the action.

The state to the RL agent is information about the task and information about the network composed so far.

REML is first tested on regression tasks as a series of varying sinusoidal curves. Performance is evaluated looking at loss per step (convergence speed) for a model constructed by REML as compared to a model trained from scratch with the same architecture and training scheme. REML is also tested on a few-shot learning task where it is given 5 and 10 samples of an unseen task, and tested after 0, 1, and 10 gradient steps. Future work is to continue testing REML on classification and Reinforcement learning tasks, and expanding the action space of REML to decide hyperparameters in addition to parameters.

The research questions this work seeks to answer are: (1) can REML transfer knowledge from tasks it has learned to new tasks?, (2) how well does the policy adapt to new tasks while retaining previously learned knowledge (does REML generalize without catastrophic forgetting)?, (3) can REML generate models that learn near-optimal parameters from few examples for unseen tasks (can it do few-shot learning)?

To answer these research questions, preliminary implementation questions are: (1) how to compose a neural network using Reinforcement learning?, (2) how to train a neural network using Reinforcement learning?, and (3)

how to transfer learning (i.e., optimized weights) using Reinforcement learning?

Chapter 2

Related work

The REML algorithm proposed in this thesis is deep reinforcement learning for meta-learning. Related work includes the research where reinforcement learning is applied to meta-learning, and more generally, popular meta-learning algorithms in recent years. REML is unlike most meta-learning research in its use of RL as the meta-learning agent, rather than adapting a meta-learning agent to RL. Algorithms that have used RL as the meta-learning agent are typically limited to RL tasks (CITE). REML is unique in adapting parameters with RL for non-RL tasks. For this reason, REML is *Reinforcement* Meta-Learning and not Meta Reinforcement Learning.

RL^2 is perhaps the most similar work to REML in that it frames the learning process as the RL agent's objective [3]. It frames learning a new algorithm as a reinforcement learning problem. The difference is that the new algorithm

learned is limited to reinforcement learning tasks, hence the work is called RL^2.

REML has parallels to neural architecture search (NAS) in searching a space for a configuration to build a neural network with. The difference is that the configuration is hyperparameters in NAS versus parameters in REML. Neural architecture search with reinforcement learning is a specific method similar its use of in the use of an RL learner to generate networks where the loss of these generated networks is the reward signal to the RL learner [4]. The authors represent the model description as a variable length string that is generated by an RNN.

Model Agnostic Meta-Learning (MAML) shares with REML the ability to generalize to different learning tasks including regression, classification, and reinforcement learning [2]. A notable difference is MAML uses a single set of parameters that it adapts to new tasks and calculates the Hessian to update the global parameters with information on how these parameters changed for each new task. The intent was to enable the architecture to adapt with few gradient steps, to multiple tasks.

Chapter 3

Background

3.1 Artificial Neural Networks

Artificial neural networks (ANNs) are non-linear computational models that approximate a target function $f : \mathbb{R}^n \to \mathbb{R}^m$, where n and m are integers [5]. Given a set $X = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\} \subset \mathbb{R}^n \times \mathbb{R}^m$ of input-output pairs of size N the model is trained to approximate f such that $f(\mathbf{x}_i) = \mathbf{y}_i \forall i \in \{1, 2, \ldots, N\}$. By the Universal Approximation theorem, a neural network's approximation of a continuous function is theoretically guaranteed to be as precise as it needs to be given the network has at least one hidden layer with some finite number of nodes [6].

3.1.1 Starting from linear regression

Consider the problem of predicting the value of one or more continuous target variables $\mathbf{t} \subset \mathbb{R}^m$, or what is called regression. Given a set of N pairs $\{ (\mathbf{x}_n, \mathbf{t}_n) \}_{n=1}^{N}$, where $\mathbf{x}_n$ is a D dimensional vector such that $\mathbf{x}_n = (x_1, \ldots, x_D)^T$, the objective is to predict the value for any input vector $\mathbf{x}_n$ such that it is as as close as possible to the provided target value $\mathbf{t}_n$.

One approach is linear regression, or a linear combination over the components of an input pattern $\mathbf{x}_n$

$$y(\mathbf{x}_n, \mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_D x_D \tag{3.1}$$

where $w \in \mathbb{R}^{D+1}$ represents the parameters of the function, $w = (w_0, \ldots, w_D)$ and D is extended to $D + 1$ for the bias weight w_0.

As is, this regression function is limited to being a linear function over the input vector $\mathbf{x}_n$. Non-linear basis functions ϕ on the input variables make the function $y(\mathbf{x}_n, \mathbf{w})$ non-linear for an input $\mathbf{x}_n$:

$$y(\mathbf{x}_n, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i \phi_i (x_i) \tag{3.2}$$

This equation can be simplified further if we define a useful basis function for the bias $\phi_0(\mathbf{x}) = 1$ such that

$$y(\mathbf{x}_n, \mathbf{w}) = \sum_{i=0}^{D} w_i \phi_i \left(x_i \right) \tag{3.3}$$

Despite producing non linear outputs over the input $\mathbf{x}$ this is *linear regression* because it is linear with respect to $\mathbf{w}$.

3.1.2 Constructing neural networks

Basic ANNs can be seen as an extension to linear regression where the basis functions become parameterized. The basis functions continue to be non-linear functions over the linear combination of the input, but now the output of the basis function is dependent on the learned coefficients $\{ w_j \}$. In this construction, basis functions are known as *activation* functions h in the context of neural networks.

We start by rewriting equation 1.2 as a linear combinations over the input variables to produce a or the *activation*.

$$a = \sum_{i=1}^{D} w_i x_i + w_0 \phi \left(x_i \right) \tag{3.4}$$

The value a is transformed using a non-linear activation function h. This transformation produces z and is referred to as a *hidden unit*.

$$z = h\,(a) \tag{3.5}$$

The coefficients $\{\, w_j \,\}$ parameterizing this non-linear transformation are referred to as a *layer*.

An ANN has a minimum of two layers - an input layer and output layer. However, ANNs are not limited to 2 layers. ANNs can have l many layers where $l \in [2, +\infty)$. Networks with > 2 layers are referred to as *deep neural networks*. For the purposes of the background, we will continue with the simple 2-layer case to establish preliminaries.

The input layer operates on an input $(x_1, \ldots, x_D)$ to produce *activations* $a_j = (a_1, \ldots, a_M)$, where M denotes the number of parameters $\{\, w_j \,\}$ in the input layer. The parameters for the input layer are represented with a superscript (1) and the parameters for the output layer will be represented with a superscript (2).

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{3.6}$$

The activations are passed through a non-linear activation h

$$z_j = h\,(a_j) \tag{3.7}$$

The output layer then transforms the hidden units z_j to produce output unit activations a_k where $k \in (1, \ldots, K)$ and K is the number of outputs expected for this problem (i.e., appropriate to the target variable $\mathbf{t}_i$ for $\mathbf{x}_i$).

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{3.8}$$

The activation a_k is transformed by a different non-linear activation function that is appropriate for K. Here this activation function is represented as σ. A common choice of activation function h for non-output layers is the rectified linear unit $h(a) = \min(0, a)$. A common choice of activation function for σ is the sigmoid function $\sigma(a) = \frac{1}{1+e^{-a}}$ for classification problems and the identity $y_k = a_k$ for simple regression problems. We now present the equation for a *feed-forward* pass through a 2-layer ANN.

$$y_k(\mathbf{x}_n, \mathbf{w}) = \sigma \left(\sum_{j=1}^{M} w_{kj}^{(2)} h \left(\sum_{i=1}^{D} w_{ji}^{(1)} + w_{j0}^{(1)} \phi \left(x_i \right) \right) + w_{k0}^{(2)} \right) \tag{3.9}$$

A neural network then is a non-linear function over an input $\mathbf{x_n}$ to an output y_k that seeks to approximate $\mathbf{t_n}$ and is controlled by a set of adaptable parameters $\mathbf{w}$.

3.1.3 Training a neural network

The goal of learning for a neural network is to optimize the parameters of the network such that the loss function $E(X, \mathbf{w})$ takes the lowest value.

Continuing with the previous example for regression, we look at the sum-of-squares error function

$$E(X, \mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \|y\left(\mathbf{x}_n, \mathbf{w}\right) - \mathbf{t}_n\|^2 \tag{3.10}$$

There is typically not an analytical solution and iterative procedures are used to minimize the loss function E. The steps taken are

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{3.11}$$

where τ is the iteration step. An approach for the weight update step with $\Delta\mathbf{w}^{(\tau)}$ is to use the gradient of $E(X, \mathbf{w})$ with respect to the parameters $\mathbf{w}$. The weights are updated in the direction of steepest error function decrease or in the $-\nabla E(X, \mathbf{w})$ direction.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \alpha \nabla E\left(X, \mathbf{w}^{(\tau)}\right) \tag{3.12}$$

where $\alpha > 0$ is the learning rate controlling the size of update step taken. This iterative procedure is called *gradient descent optimization* [7].

3.1.4 Error function derivatives

The gradient $\nabla E(X, \mathbf{w})$ is calculated with respect to every $w \in \mathbf{w}$, for all $\mathbf{x}_n \in X$.

We start with one input pattern $\mathbf{x_n}$ and rewrite the error function as

$$\begin{aligned} E_n &= \frac{1}{2}\left(y(\mathbf{x_n}, \mathbf{w}) - \mathbf{t_n}\right)^2 \\ &= \frac{1}{2}\sum_{j=1}(w_{kj}z_j - t_{nk})^2 \end{aligned} \tag{3.13}$$

The calculation starts with the gradient of E_n with respect to each w_{kj} in the output layer (2) then continues backwards to layer (1) for w_{ji}. This method can extend to l-layer networks where $l = (1, \ldots, L)$ and $L \subseteq \mathbb{R}$.

Observe that

$$\frac{\partial E_n}{\partial w_{kj}} = \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} \tag{3.14}$$

We start by calculating the partial derivative of E_n with respect to the activation a_k. Recall that $a_k = \sum_k w_{kj}z_j$. By the chain rule:

$$\begin{aligned} \frac{\partial E_n}{\partial a_k} &= (h(a_k) - t_{nk})h'(a_k) \\ &= h'(a_k)(\hat{y}_n - t_{nk}) \end{aligned} \tag{3.15}$$

We introduce a new notation to call this partial derivative an *error*

$$\delta_k \equiv \frac{\partial E_n}{\partial a_k} \tag{3.16}$$

Next we calculate the partial derivate of a_k with respect to w_{kj}

$$\begin{aligned} \frac{\partial a_k}{\partial w_{kj}} &= \frac{\partial}{\partial w_{kj}} \left(\sum_k w_{kj} z_k \right) \\ &= z_k \end{aligned} \tag{3.17}$$

With which we can write

$$\frac{\partial E_n}{\partial w_{kj}} = \delta_k z_k \tag{3.18}$$

The procedure will continue in the same way for the remainder of the layers and their units, where we calculate the errors δ for the units in the layer and multiply error of that unit by its activation z. For layer (1) (input layer) we need to calculate

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{3.19}$$

starting with δ_j or $\frac{\partial E_n}{\partial a_j}$. Observe that

$$\frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{3.20}$$

where k is the number of outputs (here, k=1 for the continued regression example).

We calculated $\frac{\partial E_n}{\partial a_k}$ above. Continue with

$$\begin{aligned} \frac{\partial a_k}{\partial a_j} &= \frac{\partial}{\partial a_j}\left(\sum_k w_{kj} h(a_j)\right) \\ &= h'(a_j) w_{kj} \end{aligned} \tag{3.21}$$

We can finish calculating the error δ_j for equation 1.18

$$\begin{aligned} \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} &= h'(a_k)(\hat{y}_n - t_{nk}) h'(a_j) w_{kj} \\ &= \frac{\partial E_n}{\partial a_j} \\ &= \delta_j \end{aligned} \tag{3.22}$$

Thus we obtain the *backpropagation* formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{3.23}$$

where the error for a unit j is the result of backpropagating the errors in the units later in the network.

Calculating the gradient is backpropgating the errors. As we have seen, this procedure begins with a forward propagation of the input vectors x_n to calculate the activations of all units. Then, it involves calculating the errors δ_k in the output layer. Using δ_k we can calculate δ_j for the hidden units in previous layers. With all errors δ, the gradient is calculated by multiplying

the error by the activations a transformed by their non-linear function h where $h(a) = z$.

3.1.5 Recurrent Neural Networks

ANNs can be constructed as *directed graphs* $G = (V, E)$ where V is the set of vertices $\{ v_1, \ldots, v_n \}$ and E is the set of edges $\{ (u, v) \mid u, v \in V \}$. We show neural networks are directed becuause the edges are a set of ordered pairs. In comparison, an undirected graph would have edges $\{ \{u, v\} \mid u, v \in V \}$. In terms appropriate to neural networks, V corresponds to our hidden units $\{ z \}$ and output units and E corresponds to the parameters $\{ w \}$.

The 2-layer network we constructed above was a *directed acyclic graph*. G is acyclic if $\forall v \in V,$ there does not exist a cycle containing v. This means that for $\forall (u, v) \in E$, $u \neq v$.

ANNs can contain cycles however. A type of ANN that contains cycles is a *recurrent neural network* (RNN) [7]. An RNN is recurrent in that information persists in the network by being passed from one forward propagation step to the next. This ability to incorporate past network data makes RNNs useful for simulating dynamical systems.

RNNs model past network data as $\mathbf{h}_t$ or the *hidden state*

$$\mathbf{h}_t = f_h(\mathbf{x}_t, \mathbf{h}_{t-1}) \tag{3.24}$$

$$\mathbf{y}_t = f_o(\mathbf{h}_t) \tag{3.25}$$

where f_h is a transition function parameterized by θ_h and f_o is an output function parameterized by θ_o [8]. The transition function can be a non-linear function such as the rectified linear unit or the sigmoid function.

Datasets used with RNNs may include T_n many input patterns $\mathbf{x}^{(n)}$ where $T_n \in \mathbb{R}$ is the number of timesteps for which there is data for the datapoint

$$\{ (\mathbf{x}_1^{(n)}, \mathbf{y}_1^{(n)}), \ldots, (\mathbf{x}_{Tn}^{(n)}, \mathbf{y}_{Tn}^{(n)}) \}_{n=1}^{N} \tag{3.26}$$

The cost function is

$$E(\theta) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} d(\mathbf{y}_t^{(n)}, f_o(\mathbf{h}_t^{(n)})) \tag{3.27}$$

where θ is the parameters of the network, and $d(\mathbf{a}, \mathbf{b})$ is the divergence measure such as Euclidean distance used in the above sum of squares error.

The parameters θ are updated with a variant of backpropagation that works with sequential data called *backpropagation through time (BPTT)* [9]. This method "unrolls" the RNN into each a computational graph one time step

at a time. This unrolled RNN is equivalent to a deep neural network where the same parameters re-appear throughout the network per timestep. Back-propagation through time sums the gradient with respect to each parameter for all times the parameter appears in the network.

RNNs are prone to challenges during training including *exploding gradient* and *vanishing gradient*. During training with BPTT the gradients can become very large (i.e., exploding) or very small (i.e., vanishing). Calculating the errors involves multiplying the errors from later layers by the activations in earlier layers as defined above. RNNs can have long sequences in the unrolled network, meaning many multiplication operations over the gradients. Multiplying large or small numbers many times will lead to very large numbers and very small numbers, respectively.

A large gradient will cause large weight updates in the gradient update step, such as in gradient descent optimization, which will make training unstable. A small gradient will cause negligent or no weight updates such that no learning happens and hidden unit activation trend to 0. These activations are called *dead neurons* where "neuron" is another word for a hidden unit.

3.1.6 Long Short-Term Memory Networks

An extension of the RNN is the Long Short-Term Memory Network (LSTM), intended to address the exploding and vanishing gradient problems or "er-

ror back-flow problems"[10]. LSTMs introduce additional calculations called "gates" within the cells of an RNN. These gates control how much information is retained or discarded in each timestep. Each cell has state C_t and the gates responsible for modifying C_t across T_n for $(x_{1n}, \ldots, x_{Tn})$.

The *forget gate* controls the amount of information retained from the previous unit h_{t-1} and the input x_t to include in this state C_t

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3.28}$$

where W_f is a weight matrix and b_f is the bias term. The sigmoid is used as it outputs a value in $[0, 1]$, with 0 meaning to discard all previous network data and 1 meaning to keep all previous network data.

The *input gate* controls the amount of information to be included from the input x_t

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{3.29}$$

where W_i is the weight matrix and b_i is the bias term for this gate, respectively.

The output of the forget gate and input gate are composed as a proposal

vector that would be added element-wise to C_t.

$$\widetilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{3.30}$$

where $\widetilde{C}_t$ holds the amount of information to include from x_t and h_{t-1}. The tanh function is the hyperbolic tangent function $\frac{e^{2x}-1}{e^{2x}+1}$. It is used to transform x to a value within $[-1, 1]$ that results in more stable gradient calculations.

The cell state C_t is the sum of the two values we have constructed: some amount of the previous state C_{t-1} and some amount of the proposed $\widetilde{C}_t$.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{3.31}$$

The final calculation what to output - the new hidden state h_t. The cell calculates how much of the new cell state C_t to output for this timesteep.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{3.32}$$

$$h_t = o_t * \tanh(C_t) \tag{3.33}$$

3.2 Reinforcement Learning

An agent exhibiting reinforcement learning (RL) learns from interacting with an environment. The environment is represented in terms of states it can take on based on the agent's actions. This environment gives the agent a numeri-

cal reward signal based on the state-action pair. The agent continues to take actions within the environment, trying to maximize its cumulative reward.

In reinforcement learning, the learning objective of the agent then is to learn a *policy* π, a map from each state $s \in S$ and action $a \in A_s$ to the probability of $\pi(a|s)$ of taking action a in state s that maximizes cumulative reward over timesteps t.

Reinforcement learning is considered a distinct type of learning to *supervised learning* and *unsupervised learning*. The learning considered thus far has been *supervised learning* or learning from labeled examples where the ground truth is known. In this supervised context, the agent or model is given the answer after it acts. The model's task is instead to learn from labeled data such that it can generalize or approximate to unseen examples where a label does not exist. RL is not supervied learning because no answer is ever provided to the agent; the agent needs to discover its' own answer. RL is also not *unsupervised learning*, where the objective is to find patterns in unlabeled data; while the agent may build a model of the environment it interacts with as a kind of pattern recognition, the objective of RL is to maximize the numerical reward signal rather than to discover hidden structure.

A reason reinforcement learning is used over supervised learning is the answer may not be known for a sufficiently complex task. Another reason is

its often impractical to provided a full set of representative examples of all states the agent may experience.

3.2.1 Markov Decision Processes

Consider an agent that interacts with the environment over timesteps $t \in \mathbb{R}_{\geq 0}$. For each timestep $t = 0, 1, 2, \ldots$ the agent is in a state $s \in S$ where it can take an action $a \in A_s$ and recieve the reward signal r_{t+1} as it transitions from state $s = s_t$ to state $s' = s_{t+1}$. This sequence would look something like

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots \tag{3.34}$$

The state is the information the agent has about the environment. It is based on this state that the agent takes an action and enters a new state – and this process of taking an action from a state to a new state repeats. Therefore, the state needs to encode information that allows a decision in the context of how the agent is doing doing in the environment.

This information is more than the immediate sensations provided by the environment but must include relevant information about the sequence thus far (i.e., the history) of the agent's interactions with the environment. A state is said to be *Markov* if it encodes all previous states' information as relevant to take the next action in the current state.

In other words, $\forall s \in S$, s incorporates information (history) of the seqeuence to the current state. The probability of the next state s_{t+1} only depends on s_t and a_s. The past history of states and action transitions is not needed. This condition is the *Markov property*.

When a state is Markov, it has complete information on the dynamics of the environment

$$p(s', r|s, a) = P\{ R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t \} \tag{3.35}$$

where the probability of the environment continuing to state s' depends only on this state s_t and action a_t. For this to be true, $\forall s \in S$, s has a history of all seqeuences possible from that state.

RL problems that have the Markov property can be modelled as Markov Decision Processes (MDP) [11]. A markov decision process (MDP) is represented as a 4-tuple (S, A, P_a, R_a) where

- S is the set of states or *state space*
- $A = \{ A_s \mid s \in S \}$ is the set of actions or *action space*
- P is $P_a(s, s') = P(s_{t+1} = s'|s, a)$ or *transition function*
- $R_a(s, s') = \{ r \mid r \in \mathbb{R} \}$ is the reward upon transition from state s to state s' or *reward function*

The goal of the agent is to maximize the reward signal over timesteps t where $t \in T, \quad T \subseteq \mathbb{R}$. The reward value $r_t \in \mathbb{R}$ is rewarded by the environment to the agent every timestep. This accummulated value is called the *return*.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T \tag{3.36}$$

This equation for the return works if the time horizon of the agent's experience is finite. This is the case when experience have a termination condition that concludes the agent's trajectory. A learning experience that ends with a finite T is called an *episode* as in *episodic learning*. This type of learning fits tasks that have a natural endpoint, such as a car parking itself in a valid spot. A learning experience without a boundary like this is called a *continuing task*. However, if $T = \infty$ then the the return could be infinite.

Another problem with the above formulation for return is it provides equal weighting to all rewards. A *discount factor* is often added to the calculation to produce a finite sum

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T \tag{3.37}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.38}$$

where the discount factor γ is typically a number between 0 and 1. The addition of a discount factor transforms the reward contribution to the return

such that a reward k time steps into the future is only worth γ^{k-1} as much. The reward is in this sense "discounted".

Values closer to 0 will make the agent "myopic" in the sense the agent only takes into account the immediate reward and zeroes all subseqeuent rewards after s_{t+1}. In this configuration, the agent learns only from the next action it takes and is unable to learn sequences of actions that lead to some future state.

Values closer to 1, on the other hand, increasingly weigh future actions further in the trajectory. A value of 1, as discussed above, would mean each action is weighed equally; therefore, values closer to 1 such 0.95 or 0.90 provide more weight to actions near T.

The discount factor is tuned to reach a balance between a focus on near-term rewards and longer-term rewards. It helps address the problem of *temporal credit assignment* or the difficulty attributing credit to past action(s) for an observed return. Part of the challenge is the delay from the timestep t an actiona_t is taken at and when the return is calculated T. In other words, the environment may pass through multiple timesteps before the effect of an action is observed. The problem can be framed as figuring out the influence of each action on the return. Using a discount factor spreads the credit across timesteps as a measure of the reward and how far into the future actions are

from the current timestep.

3.2.2 Value Functions

As an agent interacts within its environment taking actions, the agent needs a way to decide what next action a_t to take in its state s_t. The value is decided in terms of the expected return discussed above.

There are two kinds of functions to approximate value depending on whether the input is the state s_t or the state-action pair (s_t, a_t).

The value of a state is

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma_k R_{t+k+1w}|S_t = s\right] \tag{3.39}$$

where $\mathbb{E}[\cdot]$ is the expected value of a random variable provided the agent follows policy π at timestep t.

Chapter 4

Method

4.1 Preliminaries

We define a discrete-time and finite-horizon Markov decision process (MDP) as a tuple $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, P_a, R_a, \gamma\}$, where $\mathcal{S}$ is the set of states or *state space*, $\mathcal{A} = \{ \mathcal{A}_s \mid s \in S \}$ is the set of actions or *action space*, P is $P_a(s, s') = P(s_{t+1} = s' | s, a)$ or *transition function*, $R_a(s, s') = \{ r \mid r \in \mathbb{R} \}$ is the reward upon transition from state s to state s' or *reward function*, $\gamma \in [0, 1]$ is the discount factor. The objective is to maximize the discounted expected return $G = \mathbb{E}_\tau[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)]$ where τ represents the trajectory of state-action pairs $\tau = (s_0, a_0, r_0, s_1, \ldots)$.

4.2 Formulation

REML casts "learning to learn" as an RL problem. The task given to REML is to learn regression algorithms, provided labeled regression datasets. The learning process across the set of datasets is represented as a discrete-time and finite-horizon MDP.

Each regression dataset is provided as a task to REML. Such a task $\mathcal{T}$ is defined as a dataset comprising N samples of input-output pairs represented by (x_i, y_i), where x_i and y_i are real numbers. Each task is randomly sampled from a distribution, and the set of all tasks is represented as $\mathcal{D}$. Specifically, each task $\mathcal{T}_i$ is defined as $\mathcal{T}_i = \{(x_{i1}, y_{i1}), (x_{i2}, y_{i2}), \ldots, (x_{iN}, y_{iN})\}$, where $i = 1, 2, \ldots, M$, and M is the total number of tasks. Additionally, for the purpose of evaluation, one task, denoted as $\mathcal{E}$, is chosen at random and left out. The set of such tasks then is

$$\mathcal{D} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{M-1}\} \quad \text{and} \quad \mathcal{E} = \mathcal{T}_M$$

Each task $\mathcal{T}$ is solved by approximation with a neural network N. REML constructs each N from neural network layers it sequences and trains from a pre-initialized pool of layers $\mathcal{L}$. This layer pool $\mathcal{L}$ consists of M input layers $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_M\}$, M output layers $\{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_M\}$, and a pre-defined number of hidden layers that is M times the depth of the composed network. Each input layer $\mathcal{I}_i$ is associated with a specific task $\mathcal{T}_i$, and each output

layer $\mathcal{O}_i$ corresponds to the output of the network for task $\mathcal{T}_i$. This layer pool $\mathcal{L}$ can be represented as

$$\mathcal{L} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_M, \mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_K, \mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_M\}$$

where $\mathcal{H}_k$ represents the k-th hidden layer, and K is the total number of hidden layers in the network. Each type of layer in $\mathcal{L}$ (i.e., $\mathcal{I}$, $\mathcal{H}$, and $\mathcal{O}$) has the same dimension for all tasks $\mathcal{T} \in \mathcal{D}$. Different sets of task $\mathcal{D}$ would have different types of layers and different dimensions for layer in the pool.

4.3 State Space

The state s is composed of information characterizing the task $\mathcal{T}$ and information characterizing the network built so far in the current environment. Information about the task is selected by the researcher to represent differences across tasks in the domain. For example, if the task is regression on varying sine curves, then the information could be what differentiates the curves such as the amplitude and phase shift. Information about the current environment always includes: a vector with the indices of layers chosen from the pool, a one-hot vector of layers chosen from the pool, the outputs of the network, target output (ground truth) of the network, the MSE loss of the network, the max of the absolute value outputs observed for this task (to characterize magnitude).

4.4 Action Space

The action space is discrete in the size of the global layer pool. As discussed in the formulation above, this pool includes the input, hidden, and output layers bespoke to the task (e.g., linear or dense layers for a regression task, convolutional layers for a classification task). Each action is an index into the global pool of layers; it is used to fetch the next layer to add to the network in the environment. If this potential next layer is not already in the network within the environment, then the layer is added. If this next layer is already within the network, this is an error and a penalty is calculated within the reward function.

4.5 Reward Function

The learning signal or reward function R_a is the negative MSE loss. The MSE loss captures "how good" the last few actions REML made are by assessing the performance of N on task $\mathcal{T}$. It is a highly interpretable metric with direct correlation to network performance as it is the measure of network performance. The loss is made negative because the objective of the "outer" network (REML) is to maximize a reward though the objective of the "inner" network N (sequenced by REML) is to minimize a loss.

The negative MSE loss was min-max scaled after it was observed to lead to unstable training across tasks. In addition, this performance dip coin-

cided with a drift in output values across tasks. The networks would anchor to one task (dataset) within an epoch and the other networks would output values in magnitude closer to the much smaller or larger target values for an earlier task.

Further testing showed that the Adam optimizer was responsible for a disproportionate degree of task performance in terms of the MSE loss of N. REML would choose different layers in different sequences and N would acheive the same, or in some cases better, MSE loss for the batch. This is because the Adam optimizer nullified the difference between sequences of layers, getting a better and worse set of layers (in terms of their initial, untrained MSE loss) to the same loss. This meant that even though the sequenced N performed well, the REML's policy wasn't learning effectively because it was choosing different layers with the same results.

The design is that the Adam optimizer trains the layers selected by REML for N; however, the intention was a balance in what Adam optimized in a sequenced network and in the network sequenced by REML. REML was to learn from Adam in terms of what sequence of layers (what sequence of actions) produced the highest return for the task. This finding was addressed by reducing the amount of gradient steps taken by the Adam optimizer in an episode (to credit the sequence of actions more). The number of gradient steps needs to be tuned to the task. The objective is to choose a value low

enough that Adam optimizer can't train any set of layers to the same loss, and high enough that a sequence of layers are trained to degree that differentiates one set of layers (i.e., they will have a lower loss and a higher return to be represented in the policy update).

Below is a figure showing the effect of the number of gradient steps on a regression task with sine curves using the same training setup (i.e., Adam optimizer with a 0.005 learning rate). This task requires between 30 and 50 gradient steps to begin fitting the curves at the peaks of the sine curve. This is the point at which it may be possible for the Adam optimizer to make negligible the difference between a "good" sequences of layers and a "bad" sequence of layers, as discussed above. Therefore, for a task like this, a value of 30 gradient steps would be an appropriate starting point.

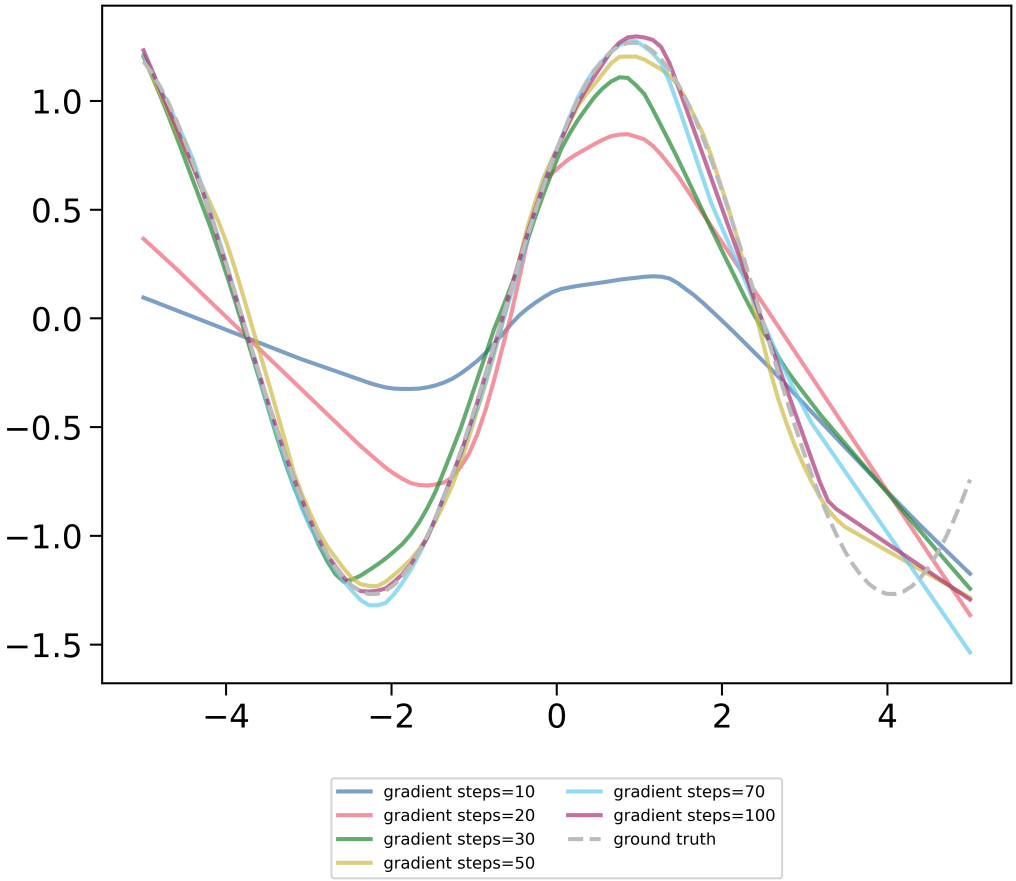

Figure 4.1: Sine curves after different amounts of gradient steps.

4.6 Policy Representation

The policy is represented as either a recurrent policy with a Long short-term memory (LSTM) network or a non-recurrent policy with a Multi-layer perceptron (MLP) network. An MLP is a feed-forward ANN without cycles that has multiple hidden layers. An LSTM is an MLP with cycles or recurrent connections that allow the architecture to use information from previous states observed in the network. These recurrent connections are composed of selective memory cells which learn if and how to incorporate past states' data. This mechanism enables the network to capture long-term dependencies in sequential data.

An LSTM was chosen for the recurrent policy to capture the patterns in the sequence of layers chosen by REML for a task. In this formulation, the sequence of layers chosen by REML are sequential data. The layers or paramemeters in a neural network are not interchangeable without changing the approximation. Therefore, the sequence of the selected layers by REML is as important as the layers that are chosen.

4.7 Policy Optimization

With the formulation as an RL problem, the policy can be optimized with RL algorithms. REML uses the Proximal policy optimization (PPO) algorithm with both MLP and LSTM policies, for reasons noted above [12]. PPO

was selected as it works with a discrete action space and was designed with training stability in mind. PPO clips updates calculated from a rollout of batch data such that the new policy is never too different from the old policy after a step. Stability was a priority given the policy would be trained across multiple different tasks and it would be possible for the policy to anchor to a particular task in a particular epoch, skewing the performance to that task.

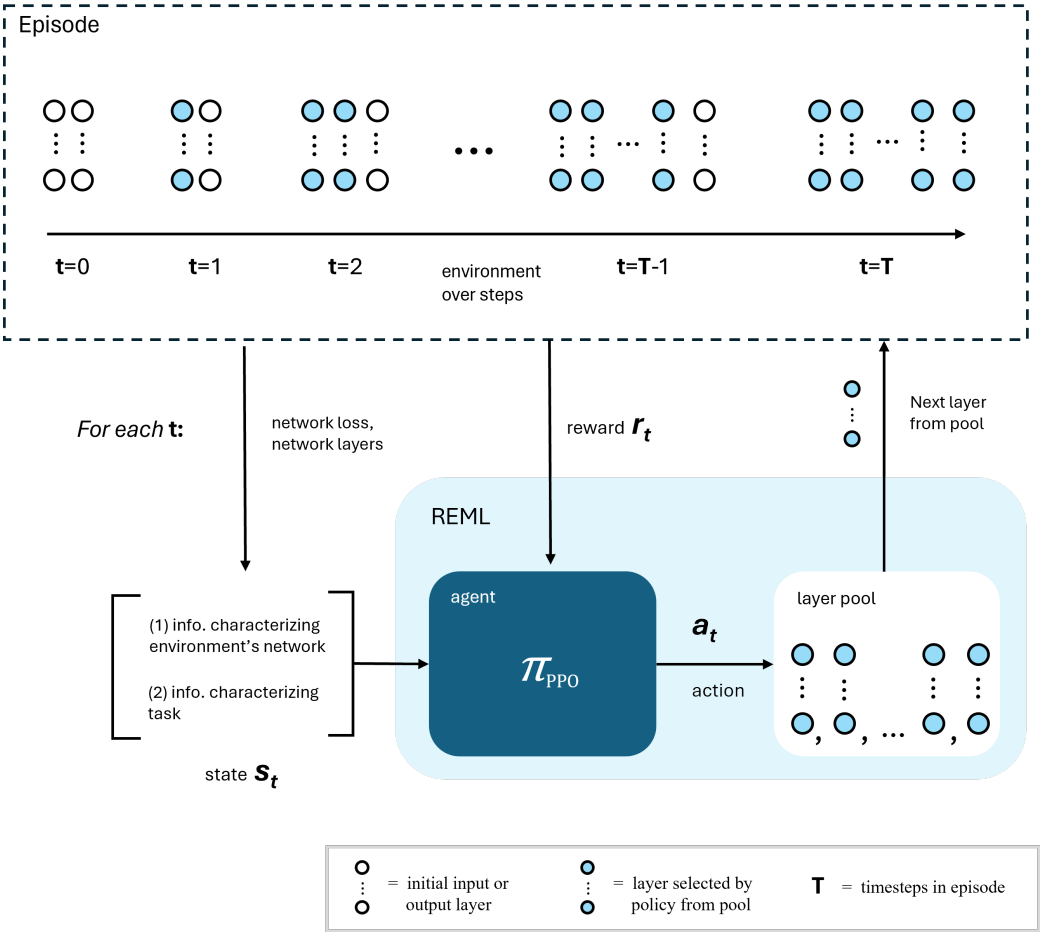

Figure 4.2: Overview of REML and meta-learning task formulation.

4.8 Algorithm

Algorithm 1 Reinforcement Meta-learning

Input: α, β: learning rate hyperparameters
Input: $p(\mathcal{T})$: distribution over tasks t
Input: $\mathcal{L}$: set of layers l
Input: f_{θ_o}: policy θ_o
Initialize all $l \in \mathcal{L}$ with θ_i
Sample training tasks $\mathcal{T}_i$ from $p(\mathcal{T})$
for all $\mathcal{T}_i$ **do**
  for all episodes **do**
    Initialize network N with initial input and output layers from $\mathcal{L}$
    Get initial state s_k
    while not done **do**
      Sample batch $\{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
      Get next layer $l_k \in L$ via $f_{\theta_o}(s_k)$
      Expand θ_i with l_k's parameters
      Pass $\mathbf{x}^{(j)}$ to N to get $\hat{\mathbf{y}}^{(j)}$
      Get MSE loss of N passing $\{\mathbf{x}^{(j)}, \hat{\mathbf{y}}^{(j)}\}$
      Get new state s_k incorporating loss and layer information
      Evaluate L_{θ_o} for chosen f_o
      Evaluate L_{θ_i} for chosen f_i using MSE loss
    end while
    $\theta_i \leftarrow \theta_i + \beta \nabla L_{\theta_i}$ using Adam optimizer
    $\theta_o \leftarrow \theta_o + \alpha \nabla L_{\theta_o}$
    Update $\mathcal{L}$ with θ_i
  end for
end for

Training starts with sampling training tasks $\mathcal{T}$ and initializing N with an input layer $\mathcal{I}$ and output layer $\mathcal{O}$ from $\mathcal{L}$. Each step, REML selects a layer from $\mathcal{L}$ to sequence in N based on the state $s \in \mathcal{S}$. Training consists of episodes where an episode terminates when the sequenced network N reaches

a pre-defined depth. At the end of an episode, layers are trained with the Adam optimizer [13]. A treatment of the negative mean squared loss error (MSE loss) is the reward signal. The trained layer copies are then updated in the layer pool $\mathcal{L}$. The idea is that knowledge acquired by REML is held by these updated layers in $\mathcal{L}$ with the policy serving as a map to rebuild a network it has made perform for the task or a related task in the past.

4.9 Time complexity

The time complexity is $O(\mathcal{D} * T * \frac{n}{b}(h * g))$ where $\mathcal{D}$ is the set of tasks, T is the number of timesteps, $\frac{n}{b}$ represents the number of batches trained over where n is the number of datapoints for a task and b is the batch size. The $h * g$ in the inner loop represents the gradient steps g and the network depth h of the constructed network by REML.

4.10 Space complexity

Memory is used to store the tasks $\mathcal{D}$, the layers in the layer pool $\mathcal{L}$, the parameters of the policy network θ_o, and copies of $l \in \mathcal{L}$ layers for the constructed network θ_i. The space required by $\mathcal{L}$ is $O(|\mathcal{D}| * h)$ where h is the network depth. We represent θ_o of the policy with m and θ_i with n. The space complexity of REML is $O(|\mathcal{D}| * h + m + n)$.

4.11 Implementation

REML was implemented using Pytorch for the layer pool and functions as part of the the constructed networks. Stablebaselines3 was used for policy with its open-source implementations of deep RL algorithms [14], [15]. Stablebaselines3 was developed by researchers with the intent to provide reliable baselines given recent findings that deep RL results are hard to reproduce. It ws found that the same RL algorithms produce different results depending on seed and other minor choices made in the implementation [16]. In fact, these differences were found to be greater in some cases than the differences between deep RL algorithms [17]. Stablebaselines3 benchmarks their implementations on common environments used in research and compares theirs to other implementations. Environments used in thesis are custom and created by the author using Gymnasium [18].

Computations were carried out using the Tufts University High Performance Cluster (HPC) with available gpu for batch jobs running about 15 hours for the regression tasks across 10 runs of 30 epochs and 7 tasks.

Evaluation data was captured and analyzed using Tensorboard and Weights & Biases (wandb) [19], [20]. Plots were generated using matplotlib with scienceplots [21], [22].

Chapter 5

Evaluation

The following research questions guided evaluation:

- Can REML transfer knowledge from tasks it has learned to new tasks?
- How well does this model adapt to new tasks while retaining previously learned knowledge (does REML generalize without catastrophic forgetting)?
- Can REML generate models that learn near-optimal parameters from few examples for unseen tasks (can it do few-shot learning)?

5.1 Task design

The research questions are evaluated with regression. REML is evaluated using varying sinusoidal curves, following the protocol proposed by Finn,

Abbeel, and Levine for Model Agnostic Meta-learning (MAML) [2]. Sine curves have an inherent periodicity for the meta-learner to learn across tasks. The testbed is defined as a distribution of sine curves that vary in amplitude and phase shift. The amplitude is chosen from [0.1, 5.0] and phase shift is chosen from [0, pi]. Each task is a dataset with 100 values linearly spaced within [-5, 5].

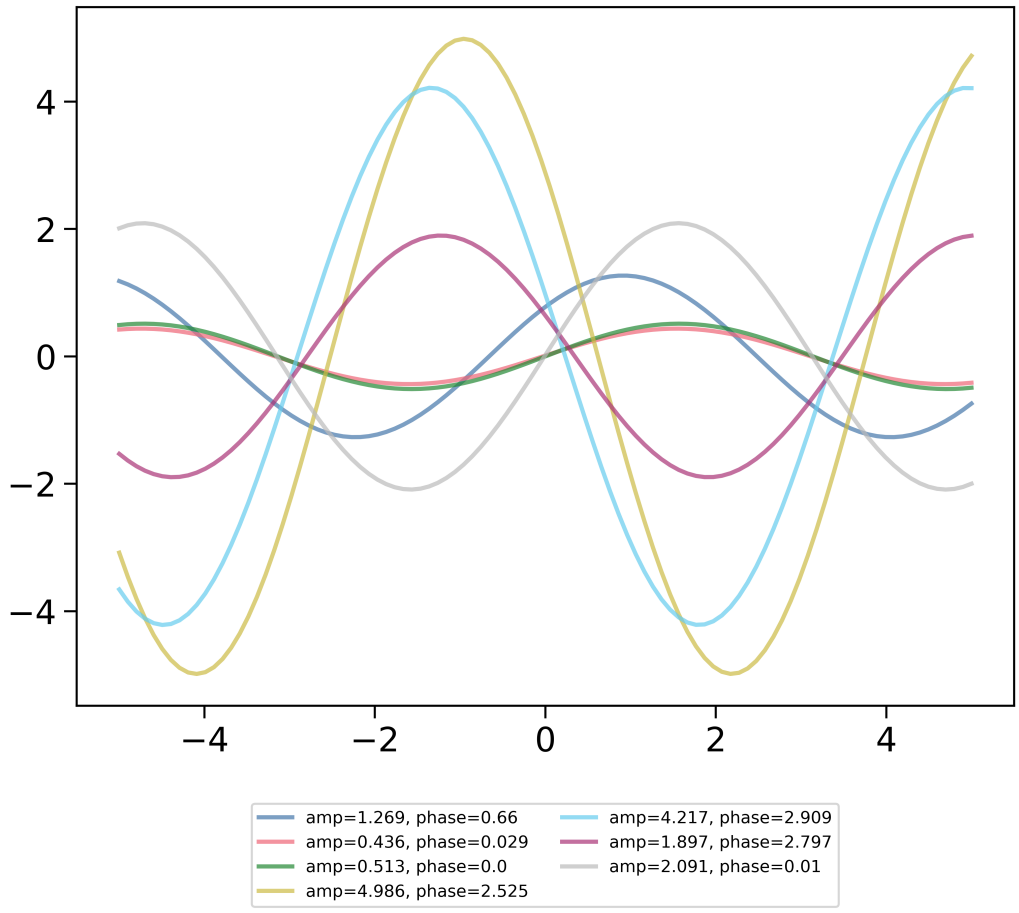

Figure 5.1: Testbed of varying sine curves with amplitude in [0.1, 5.0] and phase shift in [0, pi]

5.2 Setup

The layer pool $\mathcal{L}$ for this regression task is composed of linear layers with 40 nodes. Layers are initialized with Xavier Glorot initialization to minimize the chance of exploding or vanishing gradients, particularly in the recurrent policy case. Xavier Glorot initialization initializes the parameters with the aim to keep variance of activations similar across layers [23]. The max depth of each constructed N is 5 with 3 hidden layers. The activation function between hidden layers is the ReLU to introduce non-linearity in the approximation. The activation function for the output layer is just the identity function given this is regression. Leaky ReLU was tested with different alpha values but it generally slowed down convergence and there was no need without any observed irregularities in gradients. The hyperparameters for N are set ahead of time as the task is parameter search not hyperparameter search. The Adam optimizer is used to train each N at the end of an episode over 10 gradient steps [13]. A low value of gradient steps is employed to overcome the challenge discussed in the Method where Adam would compensate for REML's errors and train a "bad" sequence of layers to the same or better MSE loss performance than a sequence of layers observed with lower MSE loss without any gradient steps. A low value also mimics the few shot task where a maximum of 10 gradients steps are permitted. Each run begins by calculating the max and mix loss observed for the task. This first pass through the tasks does not update the policy. These min and max loss values

are used to min-max scale the reward per task and to calculate the additional discount factor in the learning signal. Each environment created for a task is normalized.

5.3 Policy selection and hyperparameter tuning

Prior to training, initial experiments were run to compare the performance of the recurrent and non-recurrent policies for a sampled set of tasks. Each policy was evaluated with different hyperparameter values for the learning rate, clip range, and horizon.

Learning rate controls the size of the gradient step taken, with higher rates using more gradient information in each update than lower rates. Learning rate values tested are 0.01, 0.001, and 0.0003 to observe performance over a large range of rates. If performance varys greatly around one rate over another, additional experiments to investigate sensitivity would be completed. The typical learning rate used with PPO is 0.0003.

Clip range is a hyperparameter unique to the PPO algorithm that controls the size of the update to the policy, preventing excessively large updates that can destabilize training [12]. The estimated policy gradient $\Delta\theta = \nabla_\theta J(\theta)$ is clipped in a range represented by $(-\epsilon, \epsilon)$ such that the gradient becomes

$\Delta\theta_{\text{clipped}} = \text{clip}(\Delta\theta, -\epsilon, \epsilon)$. Clip range values tested are 0.1, 0.2, and 0.3. The typical clip range used with PPO is 0.2.

Experience horizon is the number of experiences gathered before an update is made to the policy. In the stablebaselines3 implementation, the experience horizon is refered to as "n steps". Values tested were 5, 128, 512, 2048. The small value 5 was chosen as this is the typical length of an episode. The larger value 2048 is the typical value used. Here we are investigating whether a shorter experience horizon where the policy is more myopic considering one N architecture for one task gives better meta-learning performance over a longer experience horizon considered multiple N architectures for multiple tasks.

These hyperparameters were evaluated for PPO with a recurrent policy and with a non-recurrent policy. REML was run over 1000 steps on 7 tasks (6 training, 1 evaluation). Training averaged about 400 episodes per task for the non-recurrent PPO, as the algorithm made more errors. Training was about 200 episodes for the recurrent PPO which made fewer errors.

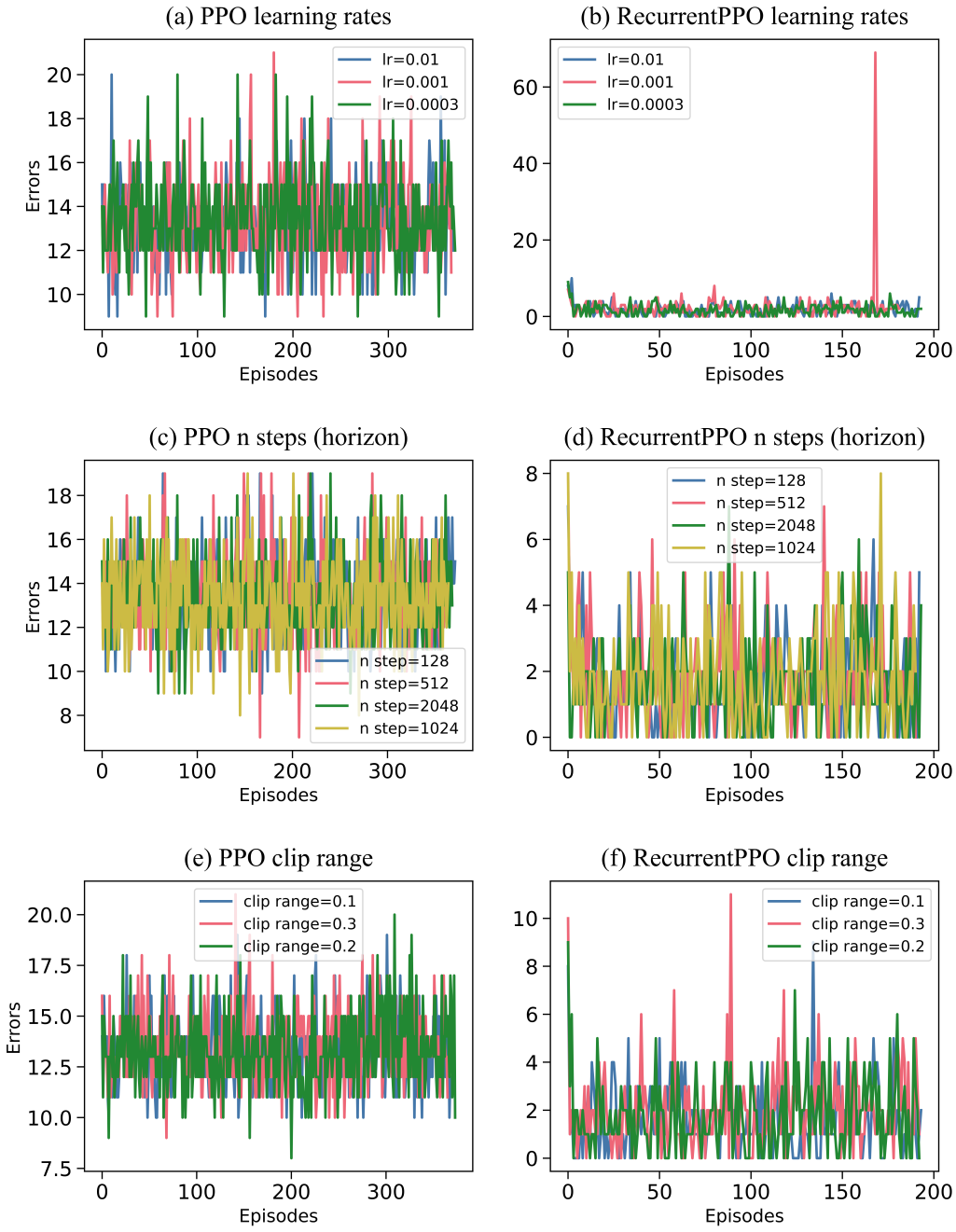

Figure 5.2: Errors made per episode across *training tasks*. REML trained on 6 tasks with 1000 steps per task.

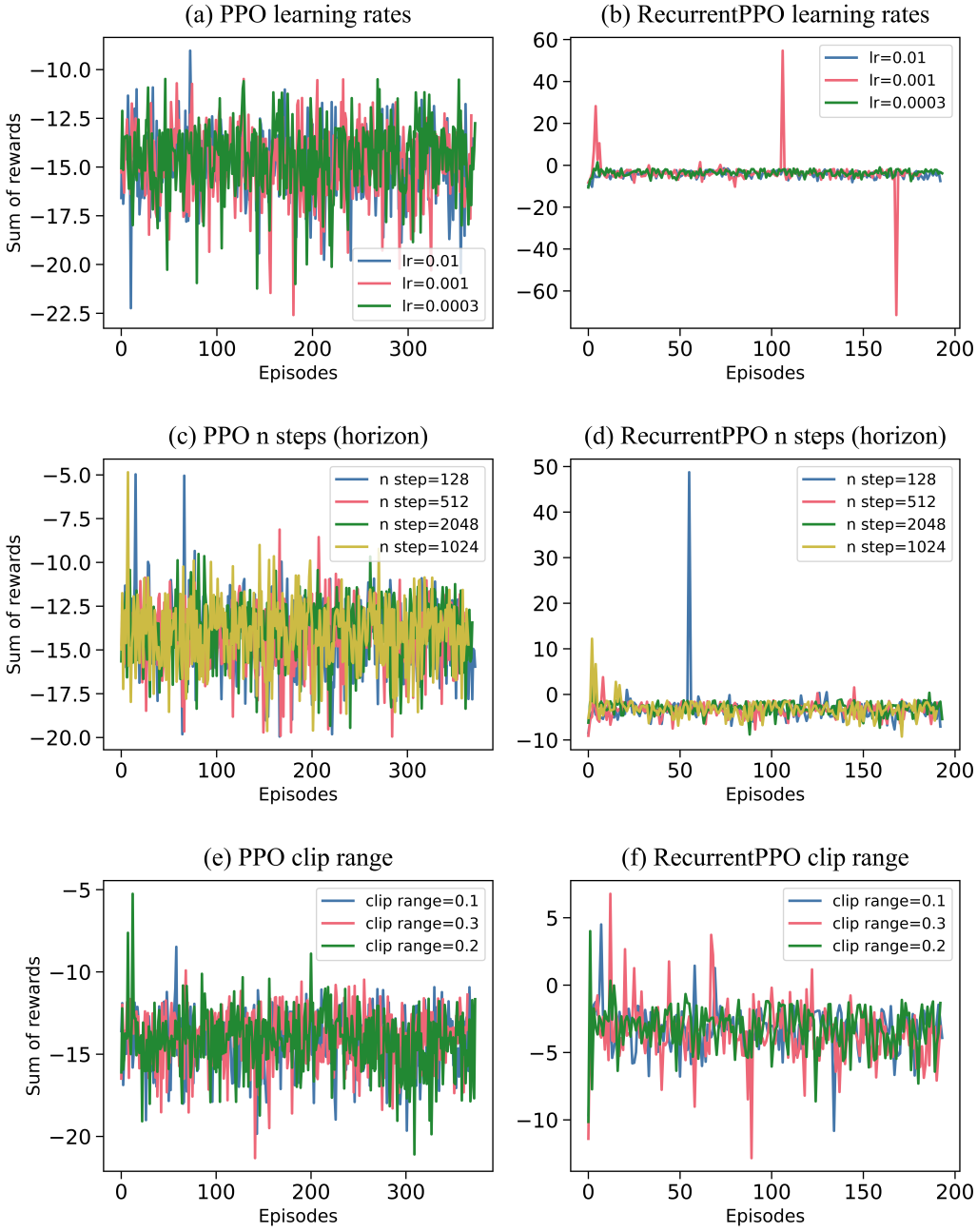

Figure 5.3: Sum of rewards per episode across *training tasks*. REML trained on 6 tasks with 1000 steps per task.

The recurrent policy outperforms the non-recurrent policy in all cases with lower errors and higher rewards per episode across hyperparameter values. Considering errors per episode, there is a single spike in errors for the recurrent policy at a higher learning rate of 0.001, though this may be noise. There don't appear to be any significant differences across other hyperparameter values for learning rate, clip range, or n steps (experience horizon) for number of errors per episode. A learning rate of 0.001 or a clip range of 0.3 appears to produce more noise and instability for the recurrent policy. The learning rate used in the original PPO implementation was lower with 0.0003. The clip range used in the original implementation was 0.2. These original hyperparameter values produce stable results.

The policies and hyperparameters were also tested on a held out task to evaluate meta-learning, or REML's ability to use prior experience to learn during new experiences more quickly. The metric is MSE loss per step during training. A network N constructed by REML is compared to a baseline network with the same architecture and layer initialization as REML's layer pool. Both network are trained with the Adam optimizer with a learning rate of 0.005. The intent is to observe whether REML generates a network for the task that has lower initial loss and converges in fewer steps than the baseline network. Lower loss values in fewer steps indicate faster learning. Lower loss values for the REML constructed network versus the baseline indicate meta learning. For both the recurrent and non-recurrent policies,

REML converges more quickly over the baseline, for all hyperparameter values. The recurrent policy outperforms the non-recurrent policy, starting at a lower loss before any additional training (i.e., at step 0), and converging in fewer steps to a minimum loss. Both policies take longer to converge with the n step hyperparameter set to 128. Results appear consistent across other hyperparameters. These results were consistent considering other randomly sampled tasks not part of the training set.

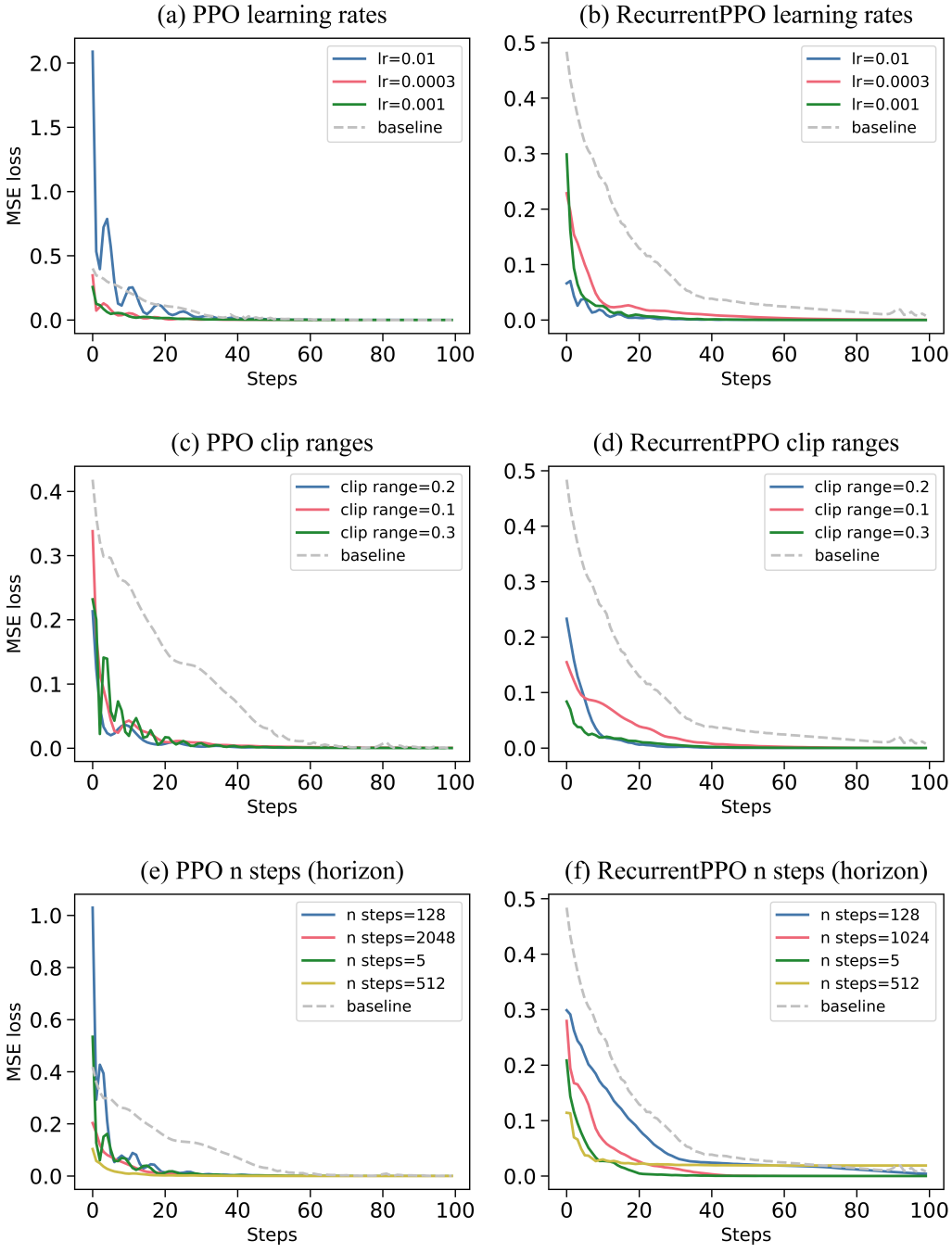

Figure 5.4: Loss per step on the held out evaluation task, comparing a network constructed by REML against a baseline with the same architecture trained from scratch.

5.4 Training the policy

Based on the policy and hyperparameter tests, REML was trained with a recurrent policy, a learning rate of 0.0003, clip range of 0.2, and n steps of 5. An n steps of 5 is chosen as convergence on unseen tasks appears faster, and more closely matches the episode length. Matching the experience horizon to be the length of the episode allows the policy to learn from each episode before continuing to next next episode.

Training is completed over 1000 steps per task for 6 tasks. Each task is trained in episodes where an episode terminates when the constructed N has reached 5 layers. This may be more than 5 steps if REML continutes to make errors; in the case of errors, no layer is added to N and REML receives a penalty.

Training performance is evaluated over 6 runs with different seeds to observe variance. Tasks are consistent across runs to enable direct comparison. Errors are tracked per episode, with a sum taken across tasks. See Figure 5.4 for errors per episode. Sum of rewards is likewise tracked per episode, with a sum taken over the tasks in Figure 5.6.

Figure 5.7 shows the sine curves generated by the policy compared to a random policy and the ground truth curve. The random policy chooses ran-

dom layers from the pool of trained layers to put into the network and then generates a curve. This is compared to a mean over 6 models generated by 6 different REML policies each trained with a different seed; the curves created by REML's models are done so after 5 steps for each task where each step is a layer. The REML policy generated models and their curvess outperform the random policy in all cases.

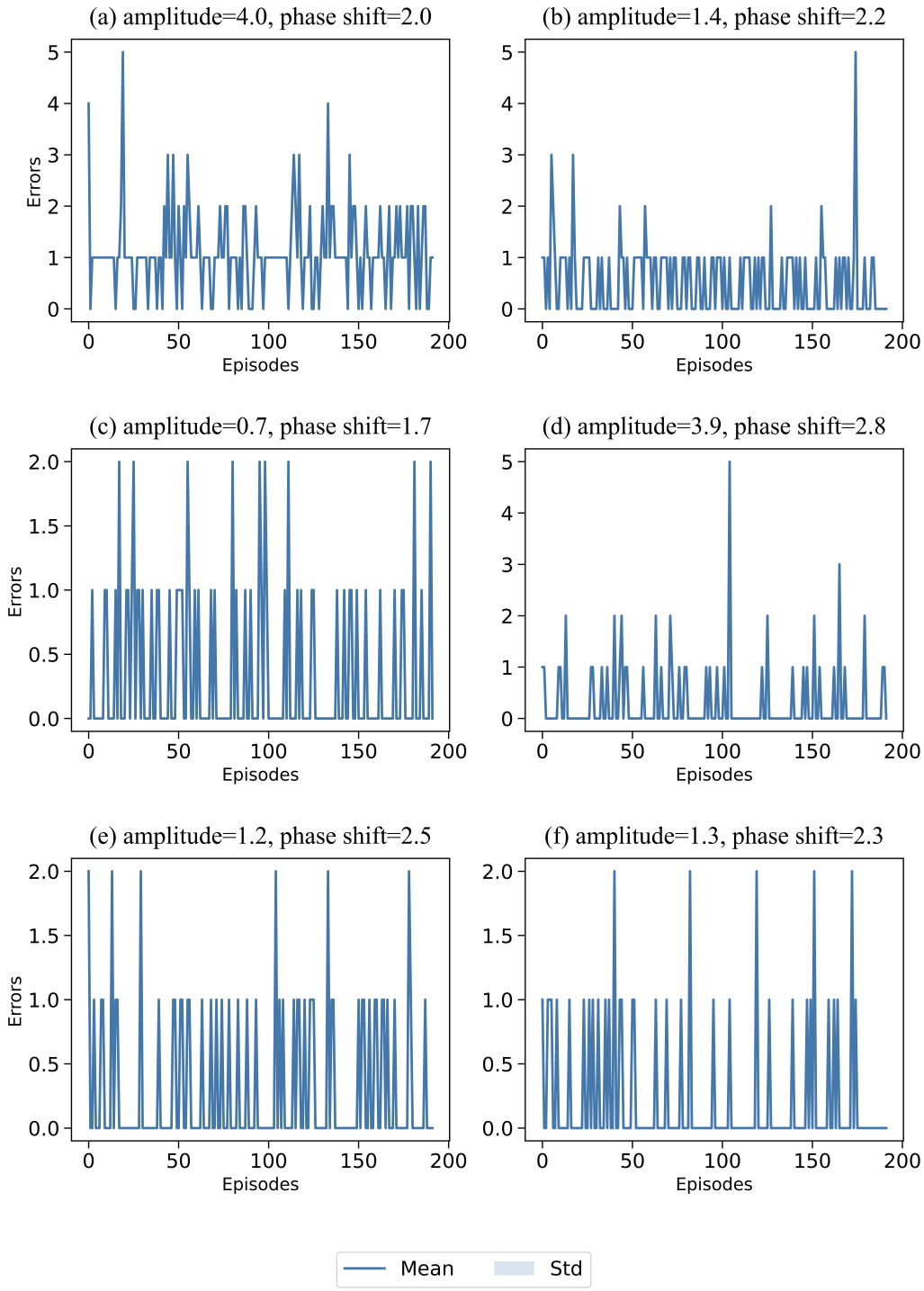

Figure 5.5: Errors per episode with recurrent policy, learning 0.0003, clip range 0.2, and n steps 5

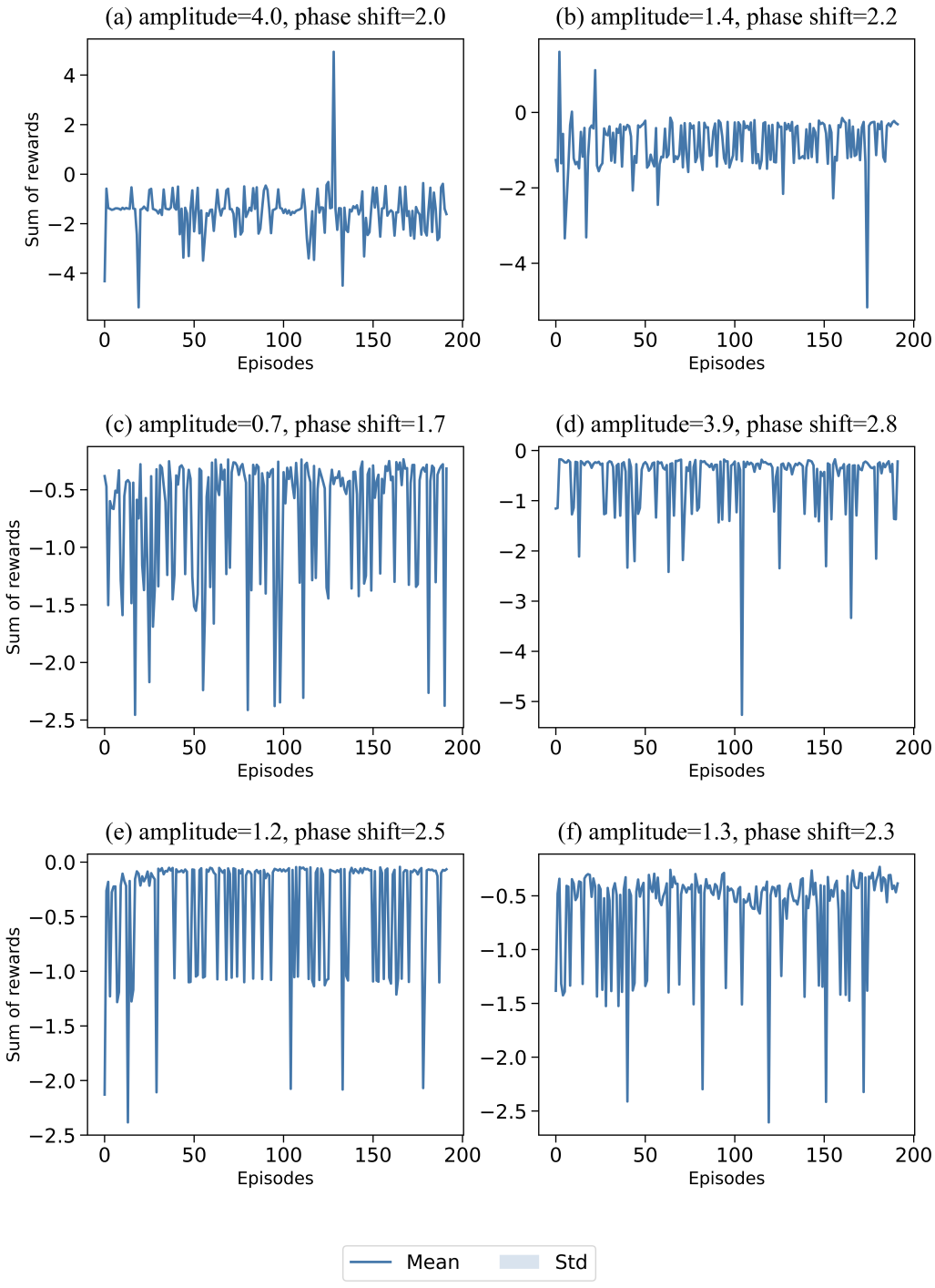

Figure 5.6: Sum of rewards per episode with recurrent policy, learning 0.0003, clip range 0.2, and n steps 5

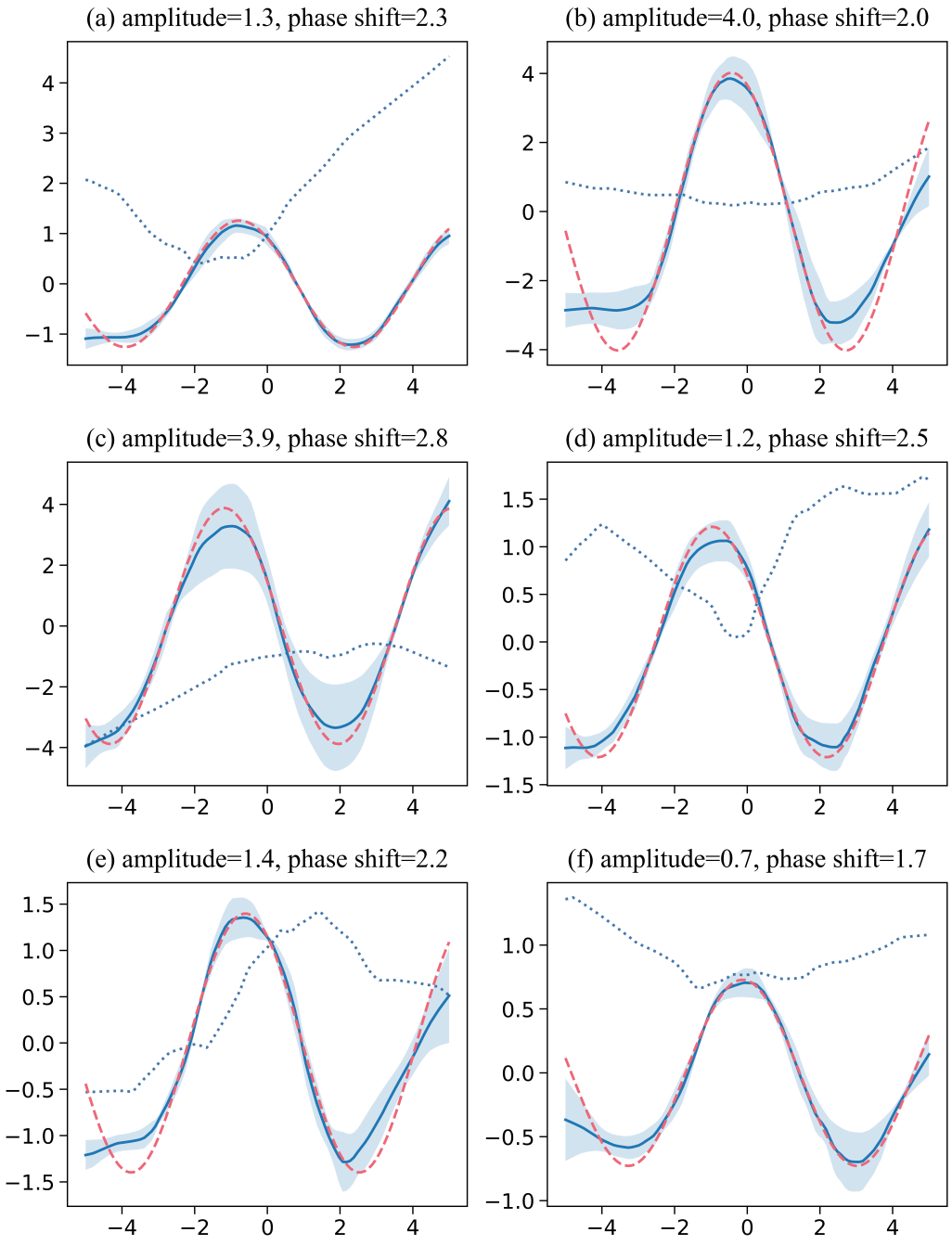

Figure 5.7: Sine curves of composed networks for training tasks

5.5 Meta-learning performance

REML's recurrent policy is evaluated on an unseen task. REML is asked to construct a network for the task. Convergence speed is tested as a MSE loss per step curve. This test is the same as above when comparing policy and hyperparameter performance. The baseline network has the same architecture, optimizer, and layer initialization. Both networks take 100 gradient steps. Results are in Figure 5.7. The network generated converges in 30 fewer gradient steps than the baseline.

The second test examines few-shot learning when $k = 5$ and $k = 10$, where k is the numbmer of datapoints x_i, y_i from the unseen task's data. For each k value , the composed network produces a sine curve at 0 gradient steps (pre-update), 1 gradient step, and 10 gradient steps trained on these k points. Results are in Figure 5.8. REML appears to have learned the general periodicity of sine curves within its layer pool. When $k = 5$, the randomly sampled pointed all lie within a narrow portion of the curve. Nonetheless, the rest of the curve shows signs of the periodicity inherent to sine curves. Considering the plots for $k = 10$, performance improves as expected given there are more data points to train on. The pre-update curves appear similar to the curves for the $k = 5$ plot after 10 gradient steps.

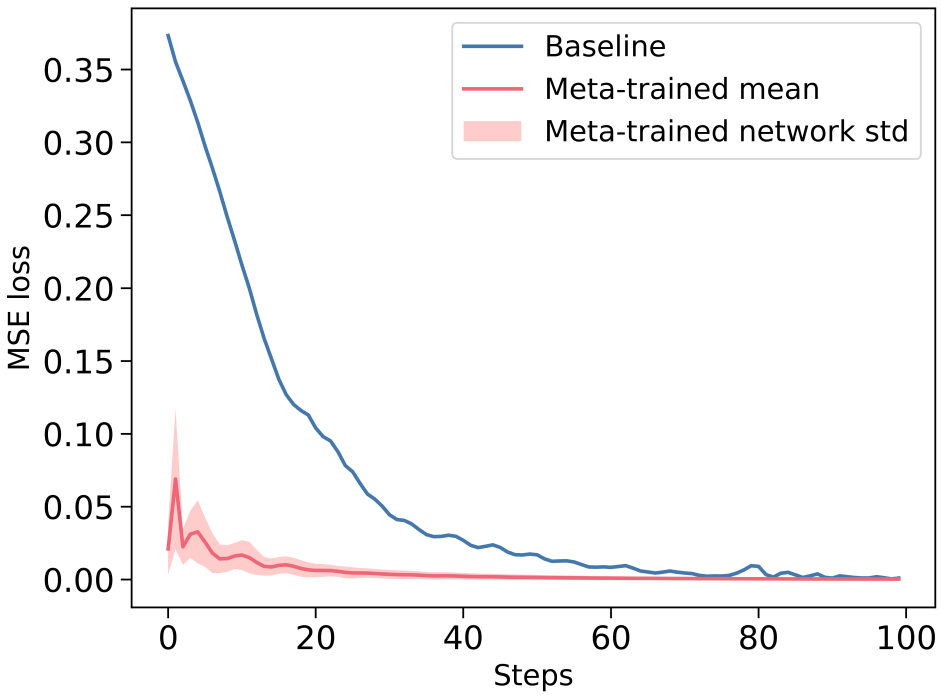

Figure 5.8: MSE loss across 100 gradient steps for an unseen task, compared to a baseline trained from scratch

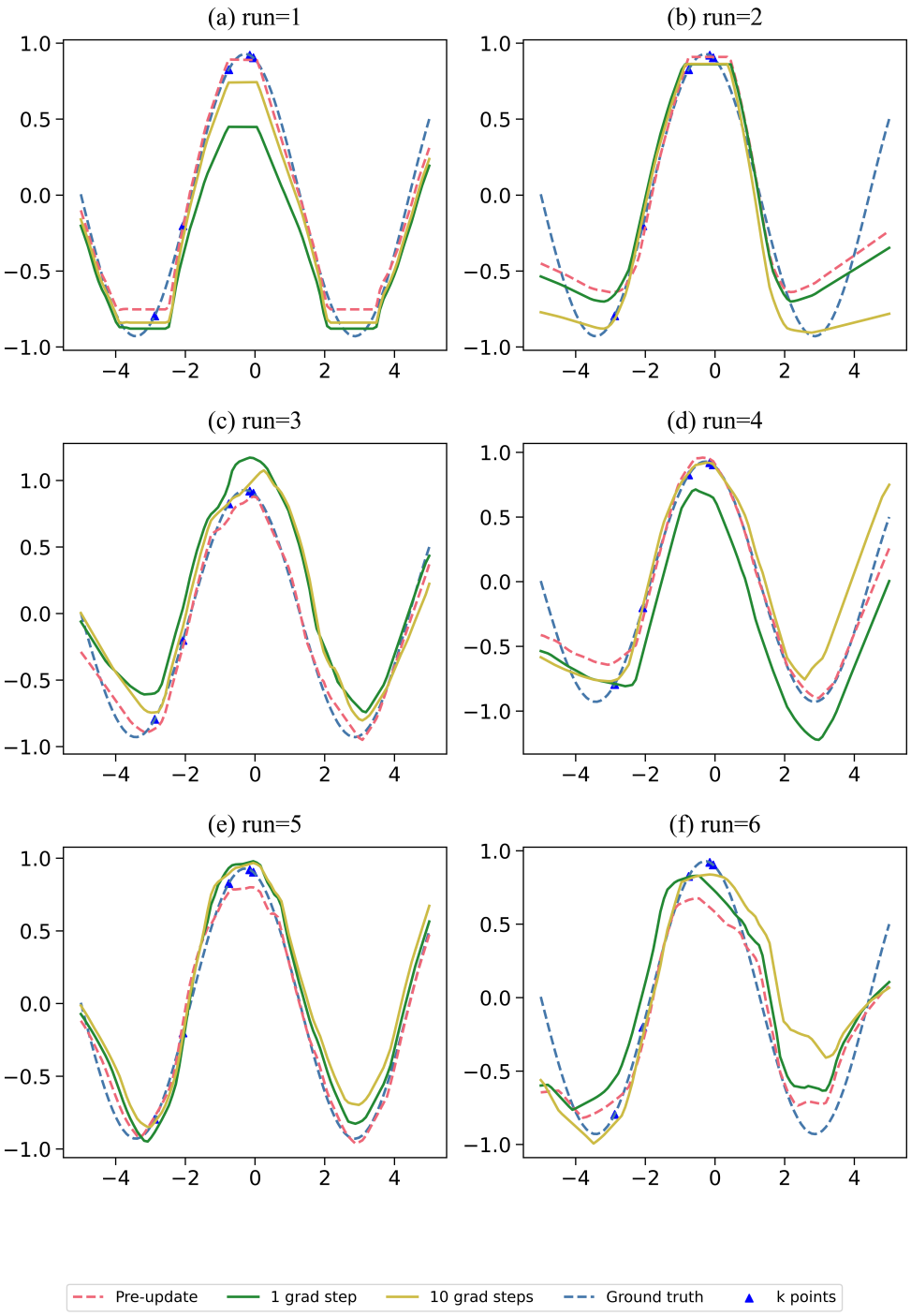

Figure 5.9: Few-shot learning for unseen task with k=5 data points

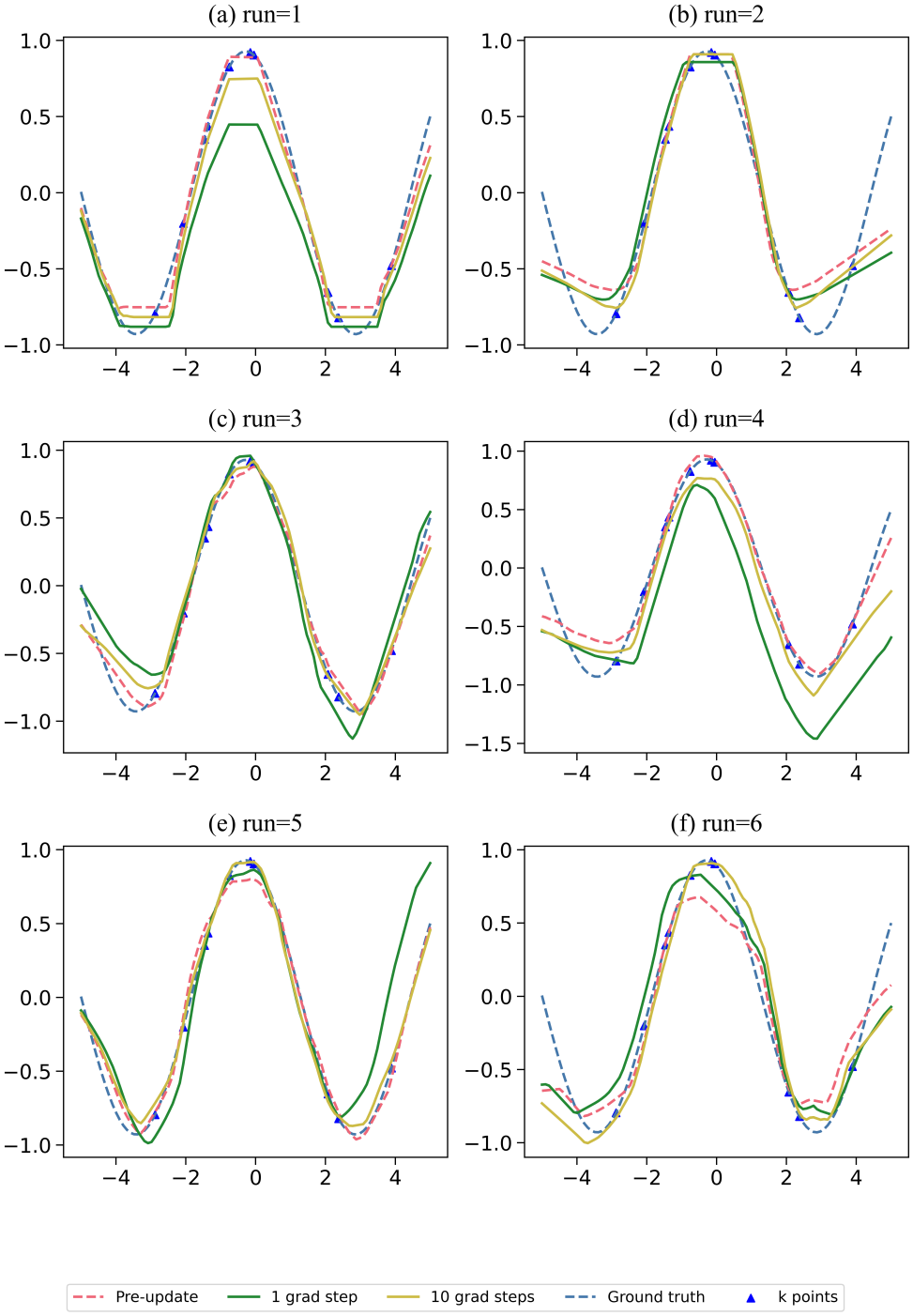

Figure 5.10: Few-shot learning for unseen task with k=5 data points

Chapter 6

Discussion

REML was proposed to cast Meta-learning as a Reinforcement learning problem. In this design, Meta-learning is a sequential task than can be solved as an MDP. The Meta-learning task is formulated as a sequence of decisions, where each decision is a layer to add to a network until some preset depth is reached. The available layers exist in a global pool that is shared across tasks; this pool is the acquired "knowledge". The objective of REML is to compose a network from trained layers in the pool such that this network provides a "warm start" with lower initial and loss and faster convergence than a network trained from scratch. This research direction was pursued to see if Meta-learning with RL can be domain agnostic by transforming the state and action space to operate on components of neural networks, where these components can change to the task (e.g., linear layers for regression and convolutional layer for classification). Let's consider each research question

in context of the results.

Can REML transfer knowledge from tasks it has learned to new tasks? As discussed, the knowledge to be transfered in REML is in the form of trained layers in a global pool. The test for *transfer learning* was comparing performance over training to see if the network composed by REML's policy has a lower starting loss and converges to some minimum loss in fewer gradient steps. In this way, REML provides this "warm start" for any task provided that task is similar enough to the tasks trained on. The baseline for comparison is a network with the same architecture (i.e., same depth, same number of nodes per layer, same Xavier initialization) that is trained over the same number of steps with the Adam optimizer using the same learning rate. The difference between the meta-trained network and the baseline network is the meta-trained network is composed of layers using REML's trained policy and pool. Plotting MSE loss versus gradient steps against this baseline indicates that the meta-trained network has a lower initial loss and converges to some minimum in fewer steps than a baseline network. These results indicate there is transfer of optimized weights by the policy.

How well does the network composed by REML adapt to new tasks while retaining previously learned knowledge? This next research question asks whether learning was robust to catastrophic forgetting. Catastrophic forgetting is observed when performance degrades on prior tasks with training on

new tasks. The model becomes overfit to more recent learning; it does not generalize across tasks. REML was designed to have enough layers in the pool such that each task had a full set of layers to encourage the system to pick different layers for each task. The reason for discouraging overlap in layers is to minimize overwriting layers trained in one task on a different task, while maintaining transfer as there are enough layers to capture knowledge for any task. If there was forgetting, it would be observed at the end of training if networks composed for earlier trained tasks performed relatively worse than networks for later trained tasks. In the evaluation we observed sine curves for all training tasks and observed consistent performance. The loss versus step curves also corroborate REML doesn't seem to forget prior tasks as the loss curves for all tasks outperform the corresponding scratch curve.

Can REML generate models that learn near-optimal parameters from few examples for unseen tasks? We have observed so far, in looking at the previous research questions, that REML shows transfer learning across training tasks. This question asks whether there is transfer learning for unseen tasks, when given few (e.g., 5 or 10) data points, or *few-shot learning*. Given the testbed for this task was taken from the MAML paper, results are compared to those observed for MAML [2]; if REML few-shot curves appear similar to those observed for MAML, then we can say there is few-shot learning as defined in their paper. The performance is compared visually, in part because

there is no loss data provided in the MAML paper and in part because we are interested in visual points of comparison (e.g., is the curve periodic?). Results show pre-update curves (i.e., policy is run to output a network that isn't further trained) provide a "warm start". These pre-update curves also have periodicity in both k=5 and k=10 cases. Additional gradient steps, with curves shown for 1 and 10 gradient steps, sometimes improve the curve and sometimes do not. This appears to be dependent on the information captured by the k data points. When k=5, the curves come closer to matching the ground truth in the domain where the k points are placed and the curves diverge where there isn't coverage. In the aggregate, across tasks, the results are similar to those observed in MAML where initial pre-update curves are a good start and improve with training and with data, though this is more dependent on the distribution of the k points in REML than in MAML.

6.1 Future Work

Future research includes testing REML on a more complex set of tasks starting with classification. The classification tasks will be the same as in the MAML paper and other meta-learning research with Omniglot and Miniimagenet [2], [24], [25]. The benefit of using these datasets is there is benchmark data across related works providing points of comparison. To test REML on

classification tasks with datasets such as these, the pool would be updated to include convolutional layers appropriate to the types of images in the dataset. The action space would likewise change such that 3 layers are added to the network in the environment: a convolutional layer followed by a pooling layer and batch normalization layer.

Another research direction is investigating different reward signals. The episode length in the depth of the network constrains the return. The experience horizon for an episode is very short, limited to be no greater than the number of layers, or depth, of the composed network - and this is typically fewer than 6 layers. This limits the size of the return, which limits how different returns are across tasks. This was visible in curves showing sum of rewards per epsiode wherre the curves are relatively flat indicating little differentiation. The problem when returns are similar is the policy doesn't have good information on how much better or worse one set of actions (layers) is over another. One design tested was shaping the reward to give a flat positive reward if the loss reduced step over step and a flat negative reward if it increased; however, flat rewards exacerbate the issue noted above with short experience horizons because returns could be equivalent in this design. More to the point, neural networks don't necessarily improve with the addition of a single layer; a network's performance depends on all layers in its architecture, and on their being trained together. Future research can tackle any of these challenges with the goal to provide the policy good in-

dication when an episode's chosen layers outperform another episode's layers.

Other research directions include expanding the action space. For this initial investigation, the action space was kept simple with only "add" actions. The action space could be expanded to include "delete" and "train" actions, where a "delete" action deletes the layer if it is in the network and "train" runs some bumer of gradient steps on all layers. These added actions do not make sense in the current episodic design because "train" would either be used on a network that isn't at full depth (requiring retraining of earlier layers with each add layer in future timesteps) or "train" would never have the opportunity to be chosen as an action, given the episode concludes when the network is full. Including these actions would requires moving to a continual learning paradigm where "train" and "delete" actions can happen at any time, and the agent continutes to choose actions after a network is full. These additional actions were briefly investigated in initial designs of REML. The delete action was removed because it seemed counterproductive in encouraging the agent to "second guess" itself, removing a layer to then add a different layer (why not pick this layer layer initially?). When the action space was tested with a "train" action, the agent learned to only pick the train action. The policy would learn to never choose "add" and the network in the environment would never grow from the initial 2 layers. This strategy gave a higher return because the loss after an "add" action will be higher than after a "train" action in the immediate term. That said, these

additional actions may make more sense in a continual learning design when the agent will have more steps to react to changes in the state.

The action space can further be expanded to incorporate hyperparameters, whether this is within the same policy or another policy dedicated to hyperparameter search. In this design, REML would be responsible for the hyperparameters as well as the parameters. This effort can draw on research dedicated to hyperparameter search with RL [4], [26].

Unbounded network depths or permitting the network to choose its architecture could be investigated too. In the design shown here, the composed networks all had the same depth, the same type of layer, and the same number of nodes per layer to keep comparison consistent to the baseline. Initial tests with unbounded network depths led to vanishing gradient even with as few as 10 layers. Future work could incorporate skip connections every few layers to mitigate the gradient problem.

Bibliography

[1] S. Thrun and L. Pratt, "Learning to learn: Introduction and overview," in *Learning to learn*, Springer, 1998, pp. 3–17.

[2] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International conference on machine learning*, PMLR, 2017, pp. 1126–1135.

[3] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "Rl2: Fast reinforcement learning via slow reinforcement learning," *arXiv preprint arXiv:1611.02779*, 2016.

[4] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[5] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4.

[6] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[8] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," *arXiv preprint arXiv:1312.6026*, 2013.

[9] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[11] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

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

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

[14] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[15] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 12 348–12 355, 2021.

[16] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[17] L. Engstrom, A. Ilyas, S. Santurkar, *et al.*, "Implementation matters in deep policy gradients: A case study on ppo and trpo," *arXiv preprint arXiv:2005.12729*, 2020.

[18] M. Towers, J. K. Terry, A. Kwiatkowski, *et al.*, *Gymnasium*, Mar. 2023. DOI: 10.5281/zenodo.8127026. [Online]. Available: https://zenodo.org/record/8127025 (visited on 07/08/2023).

[19] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[20] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020. [Online]. Available: https://www.wandb.com/.

[21] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[22] J. D. Garrett, "garrettj403/SciencePlots," version 1.0.9, Sep. 2021. DOI: 10.5281/zenodo.4106649. [Online]. Available: http://doi.org/10.5281/zenodo.4106649.

[23] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[24] X. Cao, *MLclf: The Project Machine Learning CLassiFication for Utilizing Mini-imagenet and Tiny-imagenet*, version v0.2.14, Oct. 2022. DOI: 10.5281/zenodo.7233094. [Online]. Available: https://doi.org/10.5281/zenodo.7233094.

[25] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[26] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu, "Graphnas: Graph neural architecture search with reinforcement learning," *arXiv preprint arXiv:1904.09981*, 2019.