Reinforcement Meta Learning

A thesis submitted by

Matthew Stachyra

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Tufts University

December 2023

Advisor: Jivko Sinapov

Thank you to my parents for supporting me. In their own words, they don't care what I am working on as long as I enjoy it. Perhaps that's for the best.

Thank you to Daniel Fuenmayor for being the first person to believe in my thesis. You indulged my ideas before they took shape, and gave me the confidence to shape them.

Thank you to Michael Joseph for your eternal skepticism.

Meta learning or "learning to learn" is often applied to training under the constraint where there are few example function evaluations, or *few shot learning*. This thesis contributes a new meta-learning algorithm called reinforcement meta-learning (REML). REML casts learning to learn as a markov decision process. It proposes a system with an outer network that constructs inner networks from a pool of layers for tasks. The outer network is implemented as a recurrent policy gradient method and the inner networks are appropriate to the task. This is to my knowledge the first work to use reinforcement learning as a meta-learner in a model agnostic fashion (i.e., can work for regression, classification, and reinforcement learning). Previous works have either user RL as a meta learner for other RL tasks or use RL to search hyperparameter space. REML searches parameter space for theoretically any task that can a neural network can approximate a function for. REML is evaluated on sinuosoidal in the same design as the MAML paper. REML shows transfer learning as it converges more quickly than a network trained from scratch across tasks. REML shows meta learning as it can approximate curves with k=5 and k=10 datapoints for unseen tasks. An advantage of REML over other meta-learning algorithms is in the access to past trajectories and the global layer pool, which enable it to generalize and handle unseen tasks.

Chapter 1

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 [1]. 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 [2]. 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 [3]. 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.

1.1 Title section 2.1

1.1.1 If needed

1.1.2 If needed

1.2 Title section 2.2

1.2.1 If needed

1.2.2 If needed

1.3 Title section 2.3

1.3.1 If needed

1.3.2 If needed

Chapter 2

Background

2.1 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 [4]. 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 [5].

2.1.1 Starting from linear regression

Consider the problem of predicting the value of one or more continuous target variables $\mathbf{t} \subset \mathbb{R}^m$ provided a D-dimensional vector $\mathbf{x}_n$ of input variables, or what is called regression. Given a set consisting of N observation and value pairs $\{ (\mathbf{x}_n, \mathbf{t}_n) \}_{n=1}^N$, 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 $\mathbf{x}_n$

$$y(\mathbf{x}_n, \mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_D x_D \tag{2.1}$$

where $\mathbf{x}_n$ has D dimensions, $\mathbf{x}_n = (x_1, \ldots, x_D)^T$ and $w \in \mathbb{R}^{D+1}$ 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 $y\left(\mathbf{x}_n, \mathbf{w}\right)$ non-linear for an input $\mathbf{x}_n$:

$$y(\mathbf{x}_n, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i \phi_i \left(x_i\right) \tag{2.2}$$

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

$$y(\mathbf{x}_n, \mathbf{w}) = \sum_{i=0}^{D} w_i \phi_i \left(x_i\right) \tag{2.3}$$

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

2.1.2 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*.

$$a = \sum_{i=1}^{D} w_i x_i + w_0 \phi \left(x_i\right) \tag{2.4}$$

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

$$z = h\left(a\right) \tag{2.5}$$

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).

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.6}$$

The activations are passed through a non-linear activation h

$$z_j = h\left(a_j\right) \tag{2.7}$$

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 $\mathbf{x}_i$).

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{2.8}$$

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.

$$y_k(\mathbf{x}_n, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} + w_{j0}^{(1)} \phi\left(x_i\right)\right) + w_{k0}^{(2)}\right) \tag{2.9}$$

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 $\mathbf{w}$.

2.1.3 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

$$E(X, \mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \| y\left(\mathbf{x}_n, \mathbf{w}\right) - \mathbf{t}_n \|^2 \tag{2.10}$$

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

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta \mathbf{w}^{(\tau)} \tag{2.11}$$

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.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \alpha \nabla E\left(X, \mathbf{w}^{(\tau)}\right) \tag{2.12}$$

where $\alpha > 0$ is the learning rate controlling the size of update step taken. This iterative procedure is called *gradient descent optimization* [6].

2.1.4 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

$$\begin{aligned} E_n &= \frac{1}{2} \left(y(\mathbf{x_n}, \mathbf{w}) - \mathbf{t_n}\right)^2 \\ &= \frac{1}{2} \sum_{j=1} (w_{kj} z_j - t_{nk})^2 \end{aligned} \tag{2.13}$$

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

$$\frac{\partial E_n}{\partial w_{kj}} = \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial w_{kj}} \tag{2.14}$$

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:

$$\begin{aligned} \frac{\partial E_n}{\partial a_k} &= (h(a_k) - t_{nk}) h'(a_k) \\ &= h'(a_k)(\hat{y}_n - t_{nk}) \end{aligned} \tag{2.15}$$

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

$$\delta_k \equiv \frac{\partial E_n}{\partial a_k} \tag{2.16}$$

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

$$\begin{aligned} \frac{\partial a_k}{\partial w_{kj}} &= \frac{\partial}{\partial w_{kj}} \left(\sum_k w_{kj} z_k \right) \\ &= z_k \end{aligned} \tag{2.17}$$

With which we can write

$$\frac{\partial E_n}{\partial w_{kj}} = \delta_k z_k \tag{2.18}$$

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

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{2.19}$$

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

$$\frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{2.20}$$

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

$$\begin{aligned}\frac{\partial a_k}{\partial a_j} &= \frac{\partial}{\partial a_j}\left(\sum_k w_{kj} h(a_j)\right) \\ &= h'(a_j) w_{kj}\end{aligned} \tag{2.21}$$

We can finish calculating the error δ_j for equation 1.18

$$\begin{aligned}\frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} &= h'(a_k)(\hat{y}_n - t_{nk}) h'(a_j) w_{kj} \\ &= \frac{\partial E_n}{\partial a_j} \\ &= \delta_j\end{aligned} \tag{2.22}$$

Thus we obtain the *backpropagation* formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{2.23}$$

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 $h(a) = z$.

2.1.5 Recurrent Neural Networks

ANNs can be constructed as *directed graphs*, formally defined as $G = (V, E)$ 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) [6]. 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*

$$\mathbf{h}_t = f_h(\mathbf{x}_t, \mathbf{h}_{t-1}) \tag{2.24}$$

$$\mathbf{y}_t = f_o(\mathbf{h}_t) \tag{2.25}$$

where f_h is a transition function parameterized by θ_h and f_o is an output function parameterized by θ_o [7]. 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

$$\{\,(\mathbf{x}_1^{(n)}, \mathbf{y}_1^{(n)}), \ldots, (\mathbf{x}_{Tn}^{(n)}, \mathbf{y}_{Tn}^{(n)})\,\}_{n=1}^{N} \tag{2.26}$$

The cost function is

$$E(\theta) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} d(\mathbf{y}_t^{(n)}, f_o(\mathbf{h}_t^{(n)})) \tag{2.27}$$

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)* [8]. 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.

2.1.6 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 "error back-flow problems"[9]. 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

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.28}$$

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

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.29}$$

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.

$$\widetilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{2.30}$$

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 $\widetilde{C}_t$.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{2.31}$$

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.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.32}$$

$$h_t = o_t * \tanh(C_t) \tag{2.33}$$

2.2 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 numerical 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.

2.2.1 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

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots \tag{2.34}$$

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

$$p(s', r|s, a) = P\{ R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t \} \tag{2.35}$$

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) [10]. 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.*

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T \tag{2.36}$$

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

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T \tag{2.37}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.38}$$

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.

2.2.2 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

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma_k R_{t+k+1w}|S_t = s\right] \tag{2.39}$$

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

2.2.3 Policy Gradient Methods

2.2.4 PPO

2.2.5 Recurrent Policy

2.3 Meta-Learning

Meta-learning or "learning to learn" is a general framework of using information learned from one task for future tasks. One definition extends the idea of reuse for future tasks by specifying that performance on future tasks should improve with each task learned [11]. Another definition...

2.3.1 Few shot learning

Chapter 3

Method

3.1 Preliminaries

3.2 Formulation

3.3 Policy Representation

3.4 Policy Optimization

3.5 Design

3.6 Algorithm

3.7 Implementation

REML was implemented using Pytorch for the layers and functions comprising the inner networks and Stablebaselines3 with its open-source implementations of deep RL algorithm for the meta-policy [12]. Stablebaselines3 was developed by researchers with the intent to provide reliable baselines given recent findings that deep RL results are hard to reproduce. The same algorithms produce different results depending on seed and other minor choices made in the implementation [13]. In fact, these differences were found to be greater in some cases than the differences between deep RL algorithms

[14]. Stablebaselines3 benchmarks their implementations on common environments used in research and compares theirs to other implementations.

Computations were carried out using the Tufts University High Performance Cluster (HPC) with 1 P100 GPU for batch jobs running about 4 hours for the regression and 12 hours for the classification tasks.

Data was captured using Tensorboard and Weights & Biases (wandb) [15], [16].

Chapter 4

Evaluation

The following research questions guided evaluation of REML:

- Can REML enable models to transfer knowledge from one domain to another (can it be sample efficient)?
- Can REML generate models that learn from a few examples (can it do few shot learning)?
- Can REML generate models for different modes of learning such as regression and classification (does it support multimodal learning)?

The first two questions (i.e., whether REML is sample efficient and can support few shot learning) are evaluated in context of the third question for both regression and classification tasks.

Each learining mode evaluates the performance of the outer network (i.e., the meta-policy) and the inner networks generated and trained by the meta-policy. The outer network's policy is evaluated according to its:

- return per epoch (i.e., a pass through all tasks)
- ability to converge in fewer training steps than a neural network trained without a meta-learner with the same hyperparameters for the same tasks (i.e., show *transfer learning*)
- ability to generalize to unseen tasks with few data points (e.g., k=5, k=10) (i.e., show *few-shot learning*)

The inner networks are evaluated according to the requirements of the task for the regression and for classification. For regression, the networks' loss is tracked per timestep to see if it is learning as well as the curve fit for the sine curve in the task. For classification, the networks' accuracy across classes is evaluated.

4.1 Regression

4.1.1 Sine curves

REML was first evaluated on relatively simple sine curves, following the same protocol 2018, Finn et all performed on MAML [3]. The task is defined as a sine curve within a distribution of curves that vary in their amplitude and phase shift. The amplitude is chosen from [0.1, 5.0] and phase shift is chosen from [0, pi].

The outer-network's role for the task is to construct a neural network that can provide a best fit curve for the sinusoidal function. The inner-network's role (as it is constructed by the outer network) is to predict a value as close as possible to the target output for the input.

The inner network generated by the outer network had set hyperparameters. Each network could have up to 5 layers, with the hidden layers represented as fully connected layers with 40 nodes. ReLU activation functions were used in between each hidden layer, and the identity function for the final output layer as this is regression. Leaky ReLU was tested with different alpha values but it generally slowed down convergence and there was no need to used it as this bounded design (i.e., a pre-set number of layers output by the outer network) prevented a vanishing gradient.

4.2 Classification

4.2.1 Miniimagenet

4.2.2 Omniglot

Appendix A

Title of the Appendix

Bibliography

[1] 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.

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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/`.

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