Université de Montréal

Problem Hierarchies in Continual Learning

par

Fabrice Normandin

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

May 24, 2023

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Problem Hierarchies in Continual Learning

présenté par

Fabrice Normandin

a été évalué par un jury composé des personnes suivantes :

Gauthier Gidel

(président-rapporteur)

Irina Rish

(directeur de recherche)

Dhanya Sridhar

(membre du jury)

Résumé

La recherche en apprentissage automatique peut être vue comme une quête vers l'aboutissement d'algorithmes d'apprentissage de plus en plus généraux, applicable à des problèmes de plus en plus réalistes. De cette perspective, le progrès dans ce domaine peut être réalisé de deux façons: par l'amélioration des méthodes algorithmiques associées aux problèmes existants, et par l'introduction de nouveaux types de problèmes. Avec le progrès marqué du côté des méthodes d'apprentissage machine, une panoplie de nouveaux types de problèmes d'apprentissage ont aussi été proposés, où les hypothèses de problèmes existants sont assouplies ou généralisées afin de mieux refléter les conditions du monde réel. Le domaine de l'apprentissage en continu (Continual Learning) est un exemple d'un tel domaine, où l'hypothèse de la stationarité des distributions encourues lors de l'entrainement d'un modèles est assouplie, et où les algorithmes d'apprentissages doivent donc s'adapter à des changements soudains ou progressifs dans leur environnement. Dans cet ouvrage, nous introduisons les hiérarchiées de problèmes, une application du concept de hiérarchie des types provenant des sciences informatiques, au domaine des problèmes de recherche en apprentissage machine. Les hierarchies de problèmes organisent et structurent les problèmes d'apprentissage en fonction de leurs hypothéses. Les méthodes peuvent donc définir explicitement leur domaine d'application, leur permettant donc d'être partagées et réutilisées à travers différent types de problèmes de manière polymorphique: Une méthode conçue pour un domaine donné peut aussi être appliquée à un domaine plus précis que celui-ci, tel qu'indiqué par leur relation dans la hierarchie de problèmes. Nous démontrons que ce système, lorsque mis en oeuvre, comporte divers bienfaits qui addressent directement plusieurs des problèmes encourus par les chercheurs en apprentissage machine. Nous démontrons la viabilité de ce principe avec Sequoia, une infrastructure logicielle libre qui implémente une hierarchie des problèmes en apprentissage continu. Nous espérons que ce nouveau paradigme, ainsi que sa première implémentation, pourra servir à unifier et accélérer les divers efforts de recherche en apprentissage continu, ainsi qu'à encourager des efforts similaires dans d'autres domaines de recherche. Vous pouvez nous aider à faire grandir l'arbre en visitant `github.com/lebrice/Sequoia`.

Mots clés: apprentissage en continu, conception de logiciels de recherche, apprentissage profond, apprentissage automatique

Abstract

Research in Machine Learning (ML) can be viewed as a quest to develop increasingly general algorithmic solutions (*methods*) for increasingly challenging research problems (*settings*). From this perspective, progress can be realized in two ways: by introducing better methods for current settings, or by proposing interesting new settings for the research community to solve. Alongside recent progress in methods, a wide variety of research settings have also been introduced, often as variants of existing settings where underlying assumptions are removed to make the problem more realistic or general. The field of Continual Learning (CL), for example, consists of a family of settings where the stationarity assumption is removed, and where methods as a result have to learn from environments or data distributions that can change over time. In this work, we introduce the concept of *problem hierarchies*: hierarchical structures in which research settings are systematically organized based on their assumptions. Methods can then explicitly state their assumptions by selecting a target setting from this hierarchy. Most importantly, these structures make it possible to easily share and reuse research methods across different settings using inheritance, since a method developed for a given setting is also directly applicable onto any of its children in the hierarchy. We argue that this simple mechanism can have great implications for ML research in practice. As a proof-of-concept of this approach, we introduce *Sequoia*, an open-source research framework in which we construct a hierarchy of the settings and methods in CL. We hope that this new paradigm and its first implementation can help unify and accelerate research in CL and serve as inspiration for future work in other fields. You can help us grow the tree by visiting `github.com/lebrice/Sequoia`.

Keywords: continual learning, research software design, deep learning, machine learning

Contents

List of tables

List of figures

Acronyms

CL: Continual Learning. 4, 19, 21, 22, 27, 29–31, 41, 44, 51, 58, 59, 64, 69–71, 77, 78, 91, 92, 95, 104
CNN: Convolutional Neural Network. 21
CRL: Continual Reinforcement Learning. 30, 43, 44, 52, 64, 65, 67, 72
CSL: Continual Supervised Learning. 44, 52, 53, 64, 69–71
CTaCL: Continuous, Task-Agnostic Continual Learning. 64, 65, 68, 70

DAG: Directed Acyclic Graph. 21, 33, 85
DG: Domain Generalization. 20

EWC: Elastic Weight Consolidation. 69, 70

HM-MDP: Hidden-Mode Markov Decision Process. 39, 44
HS3MDP: Hidden Semi-Markov-Mode Markov Decision Process. 39

IID: Independently and Identically Distributed. 19

LSP: Liskov Substitution Principle. 32, 33

MDP: Markov Decision Process. 25, 39, 40, 42, 44
ML: Machine Learning. 4, 18–21, 27, 31, 58, 85
MLP: Multi-Layer Perceptron. 21

PL: PyTorch-Lightning. 55, 57
POMDP: Partially Observable Markov Decision Process. 12, 39, 42, 44

RL: Reinforcement Learning. 21, 22, 24, 25, 42, 49, 53, 63, 72, 77, 82

SB3: Stable-Baselines3. 72
SL: Supervised Learning. 21, 22, 24, 42, 49, 55, 63, 64, 77, 82
SSL: Self-Supervised Learning. 50, 55

Acknowledgements

I would also like to thank my research supervisor for her support and guidance throughout the process that lead to this document. I am also very grateful to Massimo Caccia, who has been with me on this project every step of the way.

I would also like to give special thanks to Xavier Bouthillier, Nicolas Gagné and Guillaume Alain for taking the time to read this and for their useful and constructive feedback.

I also thank Pau Rodriguez and Matthier Riemer for their guidance and feedback.

Additional thanks go to Ryan Lindeborg, Alexey Ostapenko, Luccas Cecchi and Livia Cavalcanti for their support and contributions to the Sequoia framework.

Intended audience

This document is meant to serve both as a introduction to problem hierarchies as well as to Sequoia, a software framework for the field of Continual Learning.

Python pseudocode is used throughout this document to illustrate ideas and concepts. These bits of code should hopefully help illustrate more clearly and explicitly the ideas described in the surrounding text.

The first portions of this document will describe how to construct such hierarchies, and are intended for readers with a minimal background in ML.

As a forewarning, I would like to declare explicitly here that I do not possess the necessary theoretical tools or knowledge that would be required in order to describe problem hierarchies in a thoroughly rigorous manner. Abuses of notation might be present, and I sincerely hope that they are not too severe or irritating to theory-oriented readers.

The later sections will then delve more deeply into the practical aspects of Sequoia: the core design decisions that were made, its features, components, etc. These later sections assume that readers have a minimal background in programming and applied machine learning.

My personal aim in writing this document is to make the best case I can for adopting problem hierarchies in the design of future research projects and frameworks, and to have a positive impact on how ML research is performed.

Additionally, if the Sequoia project survives, then this documents can hopefully be useful to people who want to get a better understanding of it, and 2) If Sequoia dies (as all things eventually do), then others are able to learn from my process and build upon this idea in the future.

Chapter 1

Introduction

The goal of ML research is to create algorithmic solutions to solve real-world problems. In practice, these real-world problems need to be slightly refined and simplified before they can be instantiated as research problems (*settings*) for the research community to try and solve. This is achieved through various assumptions, approximations, and design decisions that make it easier to study specific aspects of the problem, as well as to create algorithmic solutions (*methods*) for them.

Alongside recent progress in methods, a wide variety of research settings have also been recently introduced which relax these simplifying assumptions in order to better reflect the imperfections and messy nature of the real world.

The field of CL, for example, consists of a family of setting where the stationarity assumption is removed, and where methods as a result have to learn from environments or data distributions that can change over time. Methods for CL thus have to be able to adapt and learn from new data while also preserving knowledge acquired in previous tasks.

"Traditional" assumptions	"Real world" conditions	Associated field
Samples are IID.	Samples are often time-correlated and the environment changes over time	Continual Learning
Train / Test data come from the same distribution	The environment where agents are deployed often differ from the one they were trained in	Domain Generalization
Labels are available for each sample	Labelling is costly, unlabelled data is often abundant and cheap	[Un/Semi]-Supervised Learning
Training phase, followed by test phase	Agents often need to adapt and continue learning on-the-fly, and don't necessarily have access to the full training data	Online Learning

Table 1.1. Differences between "traditional" research conditions and those present in the real world, along with the field of research aimed at bridging that gap.

Domain Generalization (DG), also known as out-of-distribution generalization, is another example of such a field where the focus is on creating methods that are robust to spurious changes in their environments, and are therefore able to generalize to samples outside their original training distribution. Table 1.1 shows more examples of such fields, along with the aspect of the real world which they are concerned with.

From this description, one might naturally ask: *What do these different fields have in common? Can ideas from one field be reused in another?* Reality is, most research code is not written with such considerations in mind: each field has a dedicated set of tools and frameworks that are well suited to their particular needs. Interplay and code reuse across fields is difficult.

There is often no clear separation between settings and methods in research code: the implementation for the method might depend on characteristics of the setting or vice-versa. When attempting to build upon previous work, it is therefore often necessary to completely re-implement the method and/or setting of the prior work, assuming that the code is even available. This places the additional burden of verifying the correctness of the implementation on the researcher, which does not help reproducibility.

In this work, we introduce *problem hierarchies*: hierarchical structures in which various settings within a field are organized based on their assumptions. Problem hierarchies apply the concept of type hierarchies from computer science to the realm of ML research settings, in order to alleviate many of the issues associated with ML research today. Crucially, problem hierarchies allow methods to explicitly define their region of applicability and to be applied to different settings polymorphically. Figure 1.1 shows an example of a problem hierarchy.

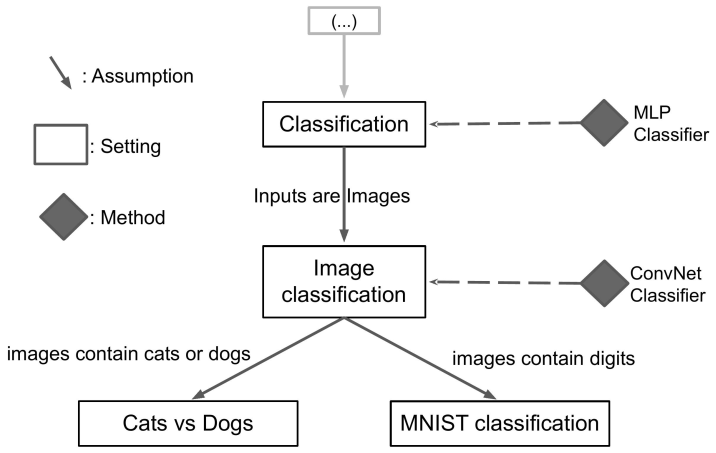


Fig. 1.1. Example of a simple problem hierarchy. Research settings can be organized into a hierarchy based on their assumptions. Methods have a *target* setting (dashed arrows), and can be applied onto any of their descendants. In this trivial example, both MLP and ConvNet classifiers can be compared in the image classification setting, even though one makes more assumptions than the other. Sequoia applies this principle to the field of continual learning.

A group of settings can be organized into a tree-shaped hierarchy (a Directed Acyclic Graph (DAG), to be precise) where the arrows between settings correspond to adding an assumption. The settings therefore become increasingly specific when moving down the hierarchy and conversely, travelling upward yields progressively more general settings. When a new method is created, it explicitly states its assumptions by selecting a *target* setting from the hierarchy. Methods become directly applicable on any setting at or below their target setting in the hierarchy.

In this simple example, a Multi-Layer Perceptron (MLP) classifier can be used in any classification setting, while a Convolutional Neural Network (CNN) classifier is applicable in classification settings where the inputs are images. The performance of both methods on the MNIST classification dataset can still be compared, even though they were created for slightly different settings. The hierarchy can be expanded upward from the root (top shaded box) to reach more and more general settings, or downward by adding more specific settings.

In this work, we argue that putting this simple idea into practice could have a great positive impact on how ML research is conducted today by alleviating many of the difficulties associated with reusing or extending prior work. The two main contributions of this work are the following:

(1) Problem hierarchies, an application of type hierarchies to the realm of ML research, as a mechanism for the systematic organization of research settings, and
(2) Sequoia, a software framework that serves as a proof-of-concept implementation of problem hierarchies for the field of Continual Learning, both in Continual Supervised Learning and Continual Reinforcement Learning.

This thesis is structured as follows. Chapter 2 begins by giving a high-level introduction to the terms and settings from the most relevant fields of research (Supervised Learning (SL), Reinforcement Learning (RL) and CL), as well as describe some of the common issues faced by researchers in these fields.

Problem hierarchies and their properties are then described in Chapter 3, along with a discussion of prior related works that propose a taxonomy or hierarchical organizations of research settings. Considerations for the implementation of problem hierarchies are given in Section 3.3, based on the lessons learned during the development of Sequoia.

Chapter 4 describes Sequoia, an implementation of problem hierarchies for the field of Continual Learning. Related work on the software side will be discussed in Section 4.1, which describes how existing frameworks and libraries tackle the common issues described in Chapter 2. A set of experiments are described in Section 4.7 as a demonstration of the Sequoia's effectiveness. We end with a discussion of future work in Chapter 5.

Chapter 2

Background

There are many different fields to choose from in machine learning. The focus of this work revolves around the settings in Supervised Learning (SL), Reinforcement Learning (RL), and Continual Learning (CL). This chapter will provide a brief introduction to some of the important concepts from each of these fields. An overview of the relevant software tools and libraries within each field will also be provided later in Section 4.1.

2.1. Prerequisites

Before we proceed further, it might be helpful to first provide an initial definition for the concepts of *setting* and *method*, which are central to this work. This will make it possible for the various fields of ML to be introduced using the same notation and perspective, which should hopefully prove to be natural and helpful. More formal definitions for each concept will be provided later in Section 3.1.

2.1.1. Setting: Type of Learning Problem

A setting describes a group of learning problems that share common characteristics. A setting can be thought of as a protocol for evaluating the performance of a learning algorithm (a method) on some type of data distribution(s). In short, settings:

- specify when, and on what data methods are trained;
- specify when, on what data, and exactly *how* method are evaluated;
- return an *objective*: a scalar or comparable value that denotes the performance of the method on that particular setting.

Settings do not specify *how* methods are trained: that is the responsibility of the method. This is in contrast with testing, which is fully specified by the setting. Applying a method onto a setting should return an *objective*, a scalar or comparable value that encodes the performance of that method on that particular setting.

```
class Setting:  # setting := class of learning problems.
    def apply(self, method: Method) -> float:
        """ Training and evaluation procedure for a method.
        This can basically be anything.
        Should return an "objective": a scalar or comparable value that
        represents how "good" a method is in this particular setting.
        """
# Create a setting
imagenet_classification: Setting = ImageClassificationSetting(dataset="imagenet")
# Create a method
some_learning_algorithm: Method = SomeImageClassificationMethod()
# Apply the method to the setting, get a performance.
performance = imagenet_classification.apply(some_learning_algorithm)
```

Listing 1. Pseudocode illustrating the concept of a setting. A method can be applied to a setting, and an objective/performance metric is returned, which describes how well the method performed in that setting.

Note that settings don't necessarily need to have to separate training and evaluation phases. Online Learning, for instance, is a setting where examples are observed only once, and where the metric of interest is the performance of the learning algorithm while it is training.

According to this formulation, the metric used as the objective is an important part of a setting's description. Changing the evaluation metric of a setting (e.g. from average test classification accuracy to test F1 score) yields a different setting. This seems reasonable, as methods might want to alter their behaviour based on what the objective of interest is in a particular setting.

2.1.2. Method - Solution to a problem

A method is a recipe to train a learning algorithm using data from a setting. Methods also provide a mechanism to perform inference, which can be used by the setting to query the method during evaluation.

The particular objective or loss function that a method uses to train is not prescribed by the setting. However, the *reward* or *task* is. To clarify, consider the example of a classification setting where the task consists of classifying images of cats and dogs. The setting describes the correct predictions for each image, and makes that information available to the method through the labels of the dataset. The method could choose to maximize the cross-entropy between its predictions and the correct class labels as its loss function.

The various settings described in this chapter will be illustrated in python pseudocode using this convention.

```
class Method:
    def train(self, data: Any):
        """ Called by the setting to let the method train using some data. """
    def predict(self, input: Any, output_space: Space[Output]) -> Output:
        """ Used by the setting for inference: Returns an output for the given input. """
```

Listing 2. Pseudocode illustration for the idea of a Method.

2.2. Research Settings

This section provides a brief introduction to the three fields most relevant to this work, namely Supervised Learning, Reinforcement Learning, and Continual Learning.

2.2.1. Supervised Learning

In Supervised Learning (SL) settings, the goal of a method is to learn $p(y|x)$, the mapping from inputs x to outputs y given a *dataset* $\mathcal{D}$ of examples consisting of inputs $x \sim p(X)$ and their associated outputs y. Methods for supervised learning are evaluated based on their generalization error, measured by their performance on a dataset D_{test} consisting of previously-unseen samples x_{test}, y_{test}, usually from the same distribution as the training data, which that have not been observed by the method during training.

Methods for SL learn the mapping from inputs to outputs by training a *model* f_θ (often a neural network), that, for a new input x, returns a predicted output for that sample, denoted y_{pred}. The model's parameters θ are optimized in order to reduce the error (or loss) on the training dataset. When trained successfully, the model becomes able to accurately predict the output for a new input from the same distribution.

There are many different frameworks and libraries designed to make it easier to create and train neural networks, not just for supervised learning, but in any kind of ML setting. These tools provide efficient implementations of the different matrix operations that form the basic building blocks of neural networks. They also support auto-differentiation, where the contribution of each model parameter on the training loss for a given example can be directly determined. The model parameters can then be learned efficiently using a gradient-based optimization algorithm such as Stochastic Gradient Descent (SGD), by repeatedly applying the change to the model parameters that would minimize the training loss. PyTorch[**63**], TensorFlow, and Jax are good examples of such frameworks.

2.2.2. Reinforcement Learning

Reinforcement Learning (RL) is a very large field of machine learning, which couldn't possibly be described in enough detail here. For a more detailed introduction to RL, we direct the reader to [**37**]. In RL settings, the goal of a method is to train an *agent*, by interacting with an *environment* in order to maximize the cumulative *rewards* provided by

```
X, Y = TypeVar("X"), TypeVar("Y")
Example = Tuple[X, Y]
Dataset = Sequence[Example]
class SupervisedLearning(Setting):
    dataset: Dataset
    def apply(self, method: Method) -> float:
        train_dataset, test_dataset = split(self.dataset)
        method.train(train_dataset)
        test_error = 0.
        for x_test, y_test in test_dataset:
            y_pred = method.predict(x_test)
            test_error += loss_criterion(y_pred, y_test)
        return test_error
```

```
Model = Callable[[X], Y]  # f(X) -> Y

class SupervisedLearningMethod(Method):
    model: Model
    def train(self, dataset: Dataset): ...
        for x, y in dataset:
            y_pred = self.model(x)
            train_loss = self.loss_function(y_pred, y)
            optimizer.minimize(train_loss)
    def predict(self, x_test: X) -> Y:
        y_pred = self.model(x_test)
        return y_pred
```

Listing 3. Pseudocode for a setting and method in Supervised Learning. The output space is omitted in this example, but is simply the space of possible outputs Y.

the environment. At each step, the agent receives an observation from the environment, then performs an *action* in the environment, and then gets back a *reward*, which essentially represents how good the action was, given the current *state* of the environment and the action that the agent performed in that state.

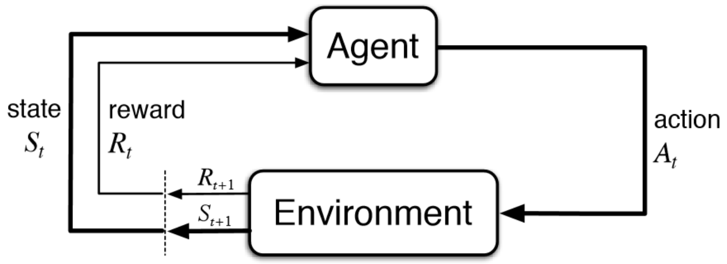


Fig. 2.1. The typical interaction loop in reinforcement learning. (Adapted from [**75**]).

RL methods learn a model (also called an actor, or *policy*) which outputs the action to take given an observation of the environment's state. How this model is trained varies greatly from one method to another. Generally speaking, methods in RL train their model by trying to increase the probability of selecting actions that lead to greater rewards, and, conversely, by lowering the likelihood of selecting actions that lead to negative or inferior rewards. RL methods can be categorized along many different axes: whether they consider a finite or infinite number of possible actions (discrete vs continuous action spaces), whether the policy is trained while interacting with the environment (on-policy) or trained using a set of collected experiences in the environment (off-policy), and many more.

Environments in RL are usually mathematically represented using some variant of a Markov Decision Process (MDP), where the environment has a set of possible states $s \in S$ (where S is called the state space), an action space A, a state transition function $p(s_{t+1}|s_t, a_t)$, and a reward function $r(s, a) \rightarrow \mathbb{R}$.

```
class Space(Generic[T]):
    def sample(self) -> T: ...
    def contains(self, value: Any) -> bool: ...

class Env(Generic[Observation, Action]):
    observation_space: Space[Observation]
    action_space: Space[Action]
    def reset(self) -> Observation: ...
    def step(self, action: Action) -> Tuple[Observation, float, bool, dict]: ...

env: Env = gym.make("CartPole-v0")  # create the environment, e.g. CartPole
done = False                        # episode termination signal
observation = env.reset()           # reset the environment, get the initial observation
while not done:                     # continue until the end of the episode
    action = agent(observation)     # Query an agent for the action to take at this step
    # OR, alternatively:
    action = env.action_space.sample()  # Choose a random action from the action space
    next_observation, reward, done, info = env.step(action)   # Update the environment
    ...
    observation = next_observation  # Update the observation before going to the next step.
```

Listing 4. Pseudocode for the `Env` and `Space` components of OpenAI Gym, as well as the python equivalent of the typical interaction loop from Figure 2.1.

In practice, the most popular library for reinforcement learning is OpenAI Gym[**7**]. Its very simple and intuitive environment interface (shown in Listing 4) is versatile enough to model all sorts of MDPs and real-world environments, ranging from simple classical control environments such as CartPole, all the way to self-driving cars.

```
Observation = TypeVar("Observation")
Action = TypeVar("Action")

class ReinforcementLearning(Setting):
  env: Env[Observation, Action]
  def apply(self, method: Method) -> float:
    method.train(self.env)
    # Test env is usually same as train env,
    # but seeded differently:
    env.seed(test_seed)
    total_reward = 0.
    for episode in range(n_test_episodes):
      obs = env.reset()
      done = False
      while not done:
        action = method.predict(obs)
        obs, reward, done, info = env.step(action)
        total_reward += reward
    return total_reward / n_test_episodes
```

```
# f(Observation) -> Action
Model = Callable[[Observation], Action]

class ReinforcementLearningMethod(Method):
  actor: Model[Observation, Action]
  def train(self, env: Env[Observation, Action]):
    for episode in range(self.n_train_episodes):
      observations, actions, rewards = [], [], []
      observation = env.reset()
      done = False
      while not done:  # gather one episode
        action = self.actor(observation)
        observation, reward, done, info = env.step(action)
        observations.append(observation)
        actions.append(action)
        rewards.append(reward)
      total_reward = sum(rewards)
      loss = some_function_of(observations, actions, rewards)
      # Minimize "loss" as a proxy for maximizing the rewards
      optimizer.minimize(loss)
  def predict(self, observation: Observation) -> Action:
      return self.actor(observation)
```

Listing 5. Pseudocode for a setting and method in Reinforcement Learning.

2.2.3. Continual Learning

Continual Learning (CL) is a field of ML, which can be roughly described as the study of learning problems that change over time. The objective of a method in CL is to learn from non-stationary distributions or environments, and to both adapt quickly to changes, as well as accumulate and retain knowledge over time.

In order to instantiate a problem in CL, one must first make assumptions about the data distribution, and set constraints to enforce non-stationary learning. [1] Assumptions are for example often made about the type and number of tasks, or the task boundaries, or the availability of task labels, while constraints often relate to memory, compute, or time allowed to learn a task. Combinations of assumptions, rules, and datasets have resulted in a multitude of settings [**38**], each of which often uses slightly different terminology, baselines, and evaluation procedures, making CL generally difficult for newcomers.

There are many different types of settings in CL, in large part due to this definition being very broad. A non-exhaustive list of such settings: Task-Incremental Learning, Incremental Learning, Multi-Task Learning, Continual Learning, Lifelong Learning. These are often very similar problem formulations, apart from slight modifications. For some of these terms, multiple different interpretations coexist in the literature. Figure 2.2 shows an example organization of some of the settings in CL based on their assumptions. This is further complicated by the fact that such problems are instantiated in the context of both Reinforcement Learning as well as Supervised Learning! Indeed, the *continual learning* aspect of these settings is usually independent from the particular type of task (whether supervised, unsupervised, reinforcement learning, or otherwise).

Incremental learning and its variants (class-incremental, task-incremental, domain-incremental learning, and others) are settings where an model or agent learns from a sequence of *tasks* (usually datasets in SL, and environments in RL). The goal of a method is to be able to learn each task separately, in sequence, and to then still perform adequately (i.e. generalize well) on held-out test data from all tasks seen so far.

In some incremental settings, the task identity is given to the method at test time, while in others it is not, and the agent has to determine which task a sample belongs to (also refered to as *task inference*). In some settings, the model/agent has to learn new classes/outputs/actions over time when encountering new tasks, while in others settings the output space remains the same across tasks.

While Incremental Learning learns multiple tasks in sequence (incrementally), Multi-Task Learning generally refers to a learning problem with more than one task, and where these tasks are learned at the same time, or sampled from a stationary distribution.

[1]It is often necessary to prevent methods from just gathering all previous data into a buffer and shuffling it, for example, since this would yield optimal performance, while totally bypassing the entire CL problem.

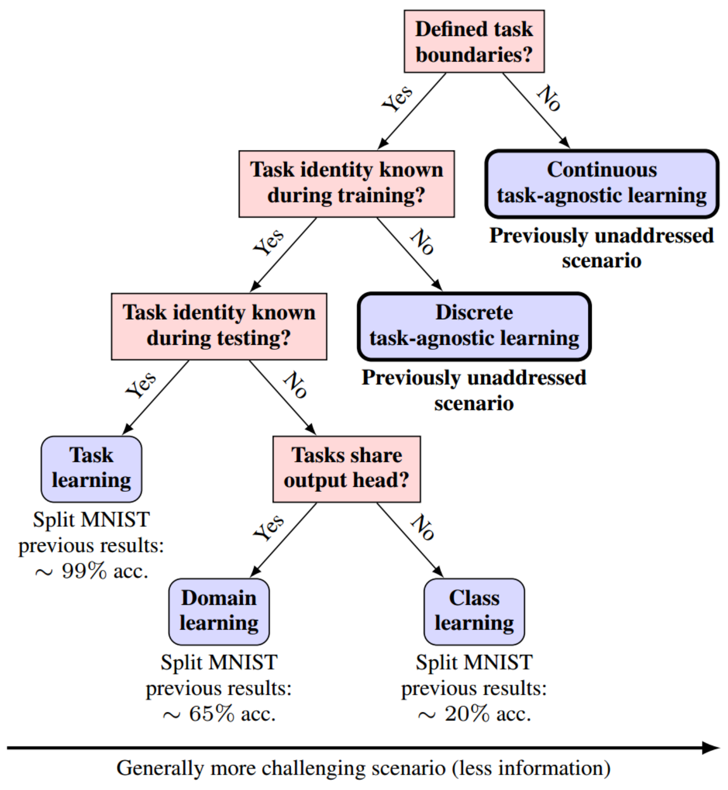

Defined task boundaries?
Yes
No
Task identity known during training?
Continuous task-agnostic learning
Previously unaddressed scenario
Yes
No
Task identity known during testing?
Discrete task-agnostic learning
Previously unaddressed scenario
Yes
No
Task learning
Split MNIST previous results: ∼ 99% acc.
Tasks share output head?
Yes
No
Domain learning
Split MNIST previous results: ∼ 65% acc.
Class learning
Split MNIST previous results: ∼ 20% acc.
Generally more challenging scenario (less information)

Fig. 2.2. Classification of continual learning settings (reproduction of figure 1 from [**89**]). This hints at a hierarchical organization of settings, which we formalize and introduce in Section 3.5.

```
# A "task" is either a dataset (SL) or an env (RL)
Task = Env | Dataset
class IncrementalLearning(Setting):
  tasks: List[Task]
  def apply(self, method: Method) -> float:
    # Create training and evaluation "tasks".
    train_tasks, test_tasks = split(self.tasks)
    # In this example, we train incrementally.
    for task in train_tasks:
      method.train(task)
    # Test after learning all tasks:
    performance_per_task = []
    for test_task in self.tasks:
      task_performance = self.evaluate(method, test_task)
      performance_per_task.append(task_performance)
    # Return average final performance.
    return sum(performance_per_task) / n_tasks
```

```
# f(Input) -> Output
Model = Callable[[Input], Output]
class IncrementalLearningMethod(Method):
    model: Model
    def train(self, task: Task):
        if not self.model:
            # First task: create the model.
            self.model = create_model(task)
        else:
            consolidate_previous_knowledge(self.model)
            self.model.prepare_for_new_task(task)
        # Train the model on this task.
        self.model.learn(task)
    def predict(self, input: Any) -> Any:
        return self.model(input)
```

Listing 6. Pseudocode for a continual learning setting and method, in this case, Incremental Learning (whether supervised or reinforcement learning).

2.2.4. Methods for Continual Learning

The objectives of Continual Learning are difficult to achieve in practice, in part because of the trade-off that exists between the two competing objectives of stability (the ability to retain knowledge over time) and plasticity (the ability to quickly learn new things). This problem is also referred to as the plasticity/stability dilemma.

One of the problems studied in CL is called *catastrophic forgetting*, and describes how training a neural network on a new task leads to a rapid degradation of its performance on previous tasks.[**56, 23**] The catastrophic forgetting problem can arguably be attributed to neural networks and the optimization algorithms used to train them, such as stochastic gradient descent (SGD) not having any inherent incentive or mechanism to preserve knowledge from data outside their current training distribution.

An effective way to mitigate forgetting is to preserve data from previous tasks and incorporate them in the training of the model on new tasks, such that the performance of the model on these tasks is preserved. Methods that employ some form of storage containing data from previous tasks are referred to as *rehearsal* or *replay-based* methods for CL. Exciting research avenues also explore the possibility of forgetting and re-learning previous tasks quickly, rather than simply preserving the model's performance on previous tasks[**8**].

Another approach towards reducing forgetting is to prevent the neural network from altering its weights in a way that would lead to a degradation of the performance on previously-learned tasks. These methods are referred to as *regularization-based* methods CL. Elastic Weight Consolidation[**40**] is an example of such methods.

Another way to tackle the CL problem is to use neural network architectures that can grow progressively over time, allocating new parameters when needed in order to learn new tasks. Methods of this sort are called *architecture-based*.

2.3. Common Issues in Continual Learning

There are several recurring issues that ML researchers have to deal with over the course of their work. The field of CL, with it's multitude of settings, assumptions, and methods, can be very difficult to study in practice. We will describe such difficulties here. It will hopefully become clear why CL is a prime candidate for the hierarchies that will be described in Chapter 3.

2.3.1. Reproducibility

The current incentive structure in ML is, roughly speaking, to publish as many papers as possible. It is very common for papers to be an isolated, somewhat disposable amount of coding work, involving a lot of redundant boilerplate logic to setup the setting, or the methods and the required baselines. Method implementations are often unnecessarily coupled to the problems they are used to solve, making them difficult to reuse outside of their original scope. Reproducing original results or reusing methods developed in that way as baselines is prohibitively difficult. In order to analyze specific properties of novel methods, researchers tend to re-implement baselines and adapt them to their particular needs [**33**]. These baselines are often not described in enough detail to ensure reproducibility, e.g. a prescribed hyper-parameter search strategy, computational requirements, open source libraries, etc.

2.3.2. Evaluation

Methods in CL are often studied under a small subset of the available settings, making it difficult to evaluate them, as their problem domains do not always overlap. Consequently, it is challenging to determine if a method will generalize beyond the setting it was designed for. To add to this, Continual Reinforcement Learning (CRL) poses further challenges in evaluation due to the lack of a clear distinction between training and testing phases [**36**]. Moreover, resource consumption is a critical factor for evaluation of CL methods which is often overlooked due to the lack of standardized evaluation protocols. Thus, there is a need for standardization of the infrastructure used to evaluate CL methods.

2.3.3. CSL and CRL evolve in silos

Continual supervised learning (CSL) and continual reinforcement learning (CRL) are often considered to be independent settings in the literature and thus tend to evolve separately from one another. However, most methods in one field can be instantiated in the other, resulting in duplicate efforts such as replay for CSL [**67, 47, 74, 48, 65**] and replay for CRL [**81, 70, 35**]. To this end, we advocate that the unification of both fields would greatly reduce these duplicate efforts and accelerate CL research.

In the following chapters, we present problem hierarchies as well as Sequoia, a unifying software framework for CL research, as a solution for jointly addressing these issues. We describe how *settings* differ from one another in terms of their *assumptions*. This perspective gives rise to a hierarchical organization of CL settings, where methods become directly applicable across different settings by polymorphism. This greatly reduces the amount of work required to develop general methods for CL, as well as to compare the performance of different methods.

Chapter 3

Problem Hierarchies - a Type Hierarchy for Learning Problems

In this chapter, we propose problem hierarchies as a solution to some of the issues described in Section 2.3. Problem hierarchies are an application of the concepts of type hierarchies and inheritance from computer science onto the field of ML. To be more precise, we claim that the principles behind well-designed type hierarchies, such as low coupling, high cohesion, separation of concerns, Liskov substitution and code reuse, can also be applied to the realm of ML research. We also argue that the benefits of following such principles in software, for instance, modularity, interchangeability, maintainability, expandability, and so-on, would also transfer over into the domain of ML, in the form of methods that are easy to use, reuse, maintain, expand, etc. We believe that adopting a hierarchical view of the research settings and methods we create could be a direct means of alleviating the reproducibility and effort duplication issues in ML.

This chapter begins with a definition of settings, methods, and assumptions in Section 3.1, followed by hierarchies and their properties in Section 3.2. Some useful considerations are provided in Section 3.3 for potential implementation of this idea in practice. Examples of problem hierarchies in various research fields are then described in Section 3.4. In Subsection 3.4.3, we describe related works where a taxonomy or hierarchical organization of the different settings is also created. We then conclude this chapter in Section 3.5 by applying problem hierarchies to CL, resulting in the hierarchical structure which will serve as the basis of Sequoia and Chapter 4.

3.1. Definitions

Let's begin by considering a hypothetical space $\mathcal{P}$ that contains all research problems imaginable. A point within that space is a concrete research problem or *benchmark*, and is denoted as p, where $p \in \mathcal{P}$.

3.1.1. Assumption

An assumption $A : \mathcal{P} \rightarrow \{0, 1\}$ is a predicate or indicator function that accepts any problem and returns whether the problem respects the assumption. The most intuitive way to describe an assumption is in natural language:

- «*The learning problem has images as inputs.* »
- «*The problem's data are i.i.d. samples from an unknown distribution.* »

Informally, assumptions that relate to the same aspect of a learning problem can be grouped into a *family* of assumptions. For instance, assumptions about the input space, or the output space, or the stationarity of the distributions, the level of supervision, the available computational resources, etc. could be viewed as belonging to different *families.* Assumption families are used to aggregate the assumptions of a setting into different subgroups, making comparisons between settings easier. Ideally, assumptions from different families should not interfere with each other or be mutually exclusive.

3.1.2. Setting

A setting $S \subset \mathcal{P}$ is a set of research problems, characterized by a set of assumptions A_S that are common to every problem within that setting: $S := \{p \in \mathcal{P} \mid a(p) = 1 \; \forall \; a \in A_S\}$ Settings are therefore a ***class*** of research problems. A setting can also be viewed as a particular region within this hypothetical space of all research problems $\mathcal{P}$.

Setting are described in the literature using a commonly-used name, whose components usually map directly to the assumptions of that setting. For instance, "ImageNet classification", "Semi-Supervised Learning", "Semi-Supervied ImageNet Classification".

A group of settings is itself also a setting, and is sometimes referred to as a *field.* The notion of *field* and *setting* are interchangeable. However, the term *field* is more often used to describe a large body of work with multiple distinct sub-groups, while *setting* is more commonly used to describe a more restricted set of problems. A setting S is considered a subtype of another setting S' if $S \subseteq S'$, that is, if all the assumptions of the *parent* (S') are respected in all members of the *child* setting S. More formally, settings should obey the Liskov Substitution Principle (LSP)[**51**]:

> «*Subtype requirement: Let $\phi(x)$ be a property provable about object x of type T, then $\phi(y)$ should be true for objects y of type S where S is a subtype of type T.* »

Where the "provable properties" in this case correspond to the setting's assumptions about the learning problem.

3.1.3. Method

Methods are solutions to research problems (settings). The domain of a method, called a *target setting*, defines what problems they can and cannot be used to solve. A method M_S can therefore be thought of as a function that applies to any problem p from the setting S: $M_S : (p \in S \to Y)$
Note, we do not explicitly define the image Y of a method, but one possibility could for example be to have $Y = \mathbb{R}$, where the output of a method is a *score* as a measure of how well the method performed in that setting.

An important consequence of settings adhering to the LSP is that methods should be covariant with respect to their input (setting) type. More precisely, given two types of problems $A, B \subset \mathcal{P}$ such that $B \subset A \subset \mathcal{P}$ - in other words, setting B makes all the same assumptions as setting A, but also makes additional assumptions - then any method M_A with A as its domain (i.e. created to solve problems of type A) should also be directly applicable to problems of type B: $M_A : (a \in A) \to Y, B \subset A \subset \mathcal{P} \Rightarrow M|_B : (b \in B) \to Y$

Methods may select one assumption from each family. If a method does not select an assumption from a given family, then it is effectively stating that it is agnostic to that particular characteristic of the learning problem.

For example, a general-purpose image classification method (M_A), and another more specialized method (M_B), designed to detect lung cancer from images of chest x-rays, can both be used to classify images of chest x-rays. However, M_B shouldn't be expected to work or be applicable to other types of image classification problems.

3.2. Hierarchies

A collection of settings can always be organized into a *hierarchy*, where more general settings, with fewer assumptions, are placed at the top, and more specific settings are placed at the bottom. This is essentially an inheritance hierarchy of classes of learning problems.

More formally, a hierarchy can be defined as a partially ordered set of assumptions, ordered by inclusion ($\subseteq$)[1].

Any problem hierarchy can be visually represented as a Directed Acyclic Graph (DAG)[2], where the empty set of assumptions is the root and corresponds to $\mathcal{P}$, and where arrows represent adding one or more assumptions. Nodes in the graph represent the settings (as sets of assumptions) obtained from making all assumptions encountered in the path from the root to that node.

[1]This concept of hierarchy *may or may not* be equivalent to that of a lattice, as studied in order theory or category theory.
[2]The DAG representation of a hierarchy *may or may not* correspond to the Hasse diagram (`https://en.wikipedia.org/wiki/Hasse_diagram`).

When an assumption is represented as an arrow from setting B to setting C in such a diagram, this indicates that setting C makes an additional assumption compared to B, and therefore that as in terms of settings, $C \subset B$, and in terms of assumptions, $A_C \supset A_B$.

There are four ways to grow hierarchies: specialization (creating a subtype), generalization (creating a supertype), extension (introducing an intermediate type) and combination (multiple inheritance).

3.2.1. Specialization - growing *downward*

Settings can always be subclassed, or further specified by adding additional assumptions, which results in a more restricted setting or subtype.

From any given setting $S \subset \mathcal{P}$, a new setting S^+ can always be created by adding an additional assumption to S. This creates a new hierarchy where $S^+ \subset S \subset \mathcal{P}$.

3.2.2. Generalization - growing *upward*

The base class of any hierarchy, also called the *root* setting, can always be de-throned by extracting an underlying assumption present in that setting, and creating a more general setting that does not make that assumption. In such a case, the previous root becomes a child of the new root.

More formally, from any given setting $S \subset \mathcal{P}$, a new setting S^- can always be created by extracting/removing an underlying (or unaccounted-for) assumption already present in S. This process creates a new ordering or structure where $S \subset S^- \subset \mathcal{P}$.

Likewise, for two classes of learning problems (settings) $A, B \subset \mathcal{P}$, if setting A can be reformulated as a particular case of setting B with additional assumption - that is, if an assumption f exists such that $(f(a) = 1 \ \forall a \in A)$ (it applies to all problems in A), and that $(\exists \ b \in B \mid f(b) = 0)$ (it doesn't apply to *all* problems in B) then $A \subset B \subset \mathcal{P}$.

3.2.3. Extension - growing *inward*

We posit that assumptions are never fully-qualified, and that it is always possible to decompose a given assumption into at least two different assumptions, which can be recombined to recover the initial assumption.

Therefore, given any hierarchy of settings $B \subset A \subset \mathcal{P}$, one can always create a new setting S such that $\exists S \subset P$ such that $B \subset S \subset A \subset \mathcal{P}$. This kind of expansion shouldn't affect the original settings and the methods that target them.

3.2.4. Composition - growing *outward*

So far, we've shown that we can take any hierarchy of settings, and grow it by either adding assumptions, which adds new settings at the bottom-end of the hierarchy, or by removing an underlying assumptions from the most general setting in the hierarchy, thus growing the hierarchy upward. However, there is also a third, very powerful way of growing hierarchies: to **combine** them.

Assumptions from different families are independent, and can be combined to create new settings where both assumptions are respected. This kind of combination of independent assumptions can be generalised as the cartesian product of their hierarchies.

The order in which they are combined is not important (i.e. combinig assumptions is commutative). This is often like adding a new prefix or suffix to the name of the setting, for example, going from a given hierarchy: {mnist, cifar, imagenet} × {classification, regression} to another: {Supervised, Semi-Supervised, Unsupervised} × {mnist, cifar, imagenet} × {classification, regression}

Summary

In summary, a setting S is defined by the set of *assumptions* that are respected by the problems within that setting A_S. A particular problem p is part of a given setting S if all of the assumptions in S are respected in p. A method M_A is a solution to a setting A, a particular class of problems. A group of settings can be organized into a directed graph with no cycles, where arrows add assumptions, and where nodes are settings.

3.3. Considerations

Creating abstractions for problems and solutions is an unavoidable step in the development of a research project with more than one setting or method. Problem hierarchies, or type hierarchies more generally, are a solution to this kind of problem. In this section, we describe some practical considerations for any potential implementations of problem hierarchies. In other words, we give a list of desired properties or objectives to consider when designing any research project that involves studying more than a one setting, or more than one method.

3.3.1. Avoiding Ambiguity

Creating a problem hierarchy for a given field begins by identifying the assumptions present in all the settings of interest and to organize them hierarchically based on these assumptions. Once the common assumptions are extracted, creating an ordering or hierarchy of the remaining assumptions can be a bit more challenging. For instance, there can often be some ambiguity in determining which setting is most general between two settings.

For example, consider a hypothetical research project in reinforcement learning, say, in the Atari setting, where the algorithm learns to play a game. Call this initial setting A. Suppose that we'd like to also develop an algorithm to learn to play multiple games, instead of a single game. For the purposes of this example, suppose that we'd want this new setting to involve having to learn by playing an even mix of the different games (i.e., to learn from a uniform, stationary distribution over all the games). Call this new setting B.

There are different ways to organize the two settings described here into a hierarchy:
$A \subset B.$: A is a single-task setting while B is a multi-task setting. A is a particular case of B, where we assume that there is only one task to learn. Any algorithm for setting B should be applicable to A, since to learn multiple games, you have to be able to learn a single game. On the other hand, an algorithm that can learn a single game cannot necessarily be applied to a setting where it is expected to learn multiple games at the same time.

However, could also make a case that:
$B \subset A.$: B is a multi-task setting, which is a particular case of A, a "traditional" setting, where the presence of multiple tasks is assumed, and each task is identified. Setting B assumes the presence of multiple sub-games or tasks, while setting A does not. Therefore, a method for setting A is designed to learn a given game, and B can be viewed as a different kind of game. Another equivalent perspective is to have the environment of setting A itself contain multiple different levels, which could be treated as different individual *games*. Therefore, setting B can be viewed as a subtype of A, where B makes additional assumptions compared to A.

One way to help solve this conundrum is to try to order settings based on the amount of information they could potentially make available to the method. In this case, setting B could potentially give its methods some information about the different games, or information about the transitions between tasks, which is not possible in setting A. Methods for B could depend on some contextual information, or on being able to explicitly model a distribution over tasks, in order to learn effectively, all of which would be impossible in setting A.

Therefore, this points in the direction of setting A being more general than B, since A cannot provide more information to methods than B. Methods for A that are applied to B could also potentially leverage the additional information to get better performance.

A useful consideration in such scenarios is that to assume that something *might* be true is always available as a more general assumption, than either choosing that it *is* or *isn't* true. For instance, it might not necessarily make sense for a method that assumes that there are multiple tasks to be applied to a setting where there is only one task, and vice-versa. The dichotomy between settings A and and B that is created from the two previous solutions could benefit from more specification. In order to account for this, we arrive at a third and final solution:
$A \subset C, B \subset C.$: Both settings A and B are distinct subtypes of a broader setting (setting C), where setting C simply assumes that there are $n \geq 1$ game(s) to learn. Setting A then restricts that assumption to $n = 1$, while setting B restricts that assumption to $n > 1$. Methods that can only learn through playing multiple different games simultaneously can

target setting B. Methods that are designed to learn, given a game with potentially many sub-games or tasks, can target setting C, and be applicable to both A and B. Lastly, methods that depend on only one game being present, without any sub-tasks, can target A.

An important advantage of this solution compared to the previous is that the definition of setting A does not need to change, or be denatured in any way. This ensures that algorithms specifically designed to learn a single game (A) are not considered applicable to setting C, where the complexity of the problem increases drastically compared to A: for example, learning to play multiple games simultaneously, without any explicit information about the different tasks.

3.3.2. Forward/Backward Compatibility

When growing a hierarchy downward following Subsection 3.2.1 - that is, when creating a new setting S^+ from an existing setting S such that $S^+ \subset S$ - The properties described Subsection 3.1.3 should be respected: all methods that were applicable to setting S should still be directly applicable to both setting S as well as the new setting S^+.

Concretely, this kind of expansion can be made easier by first identifying the difference in implementations between the two settings, and extracting the difference into a method or component that can be inherited and overridden in a new subclass.

Another practical way to enable this kind of forward-compatibility, in particular in the case where the new setting provides more information than its parent, is to design the major components (e.g. the model, training code, dataset, etc.) to accept / return structured (typed) objects that hold data, rather than the data itself. This makes it possible for the setting with the most information to create subclasses of such data types, adding new fields or properties. Designing the components, methods and functions to accept or return such objects makes it possible for these new fields or properties to be safely ignored by existing methods. This is an easy way of guaranteeing forward-compatibility when a new setting adds additional information. A simple illustration of such an approach can be seen in Listing 7.

3.3.3. Modularity

The software implementations of the learning problem(s) and their proposed solution(s) are often intertwined, such that using the method implementation outside of its original domain can be very difficult. We believe this is due to the absence of any real incentive for considering other problems than the one at hand when implementing a new method.

As a response, we suggest that research projects with more than one setting or method be designed in such a way that:

(1) new settings can be added in a way that doesn't break existing methods

```python
# Model that accepts tensors as inputs
Classifier: Callable[[Tensor], int]

@dataclass
class Inputs:
    x: Tensor

# Can accept any input that *contains* a tensor
StructuredClassifier: Callable[[Inputs], int]

# Can also accept:
@dataclass
class MultiTaskInputs(Inputs):
    x: Tensor
    t: int
```

Listing 7. Illustration of how using structured data types (e.g. python dataclasses) make it easier to create models and methods that are forward-compatible. In this example, a model with structured inputs can also be passed observations with additional information, which is safely ignored. On the other hand, other data elements would have to be discarded and some type of adapter be implemented in order for the model with tensor inputs to be applicable on observations containing additional information.

(2) new methods can be added in a way that doesn't interfere with other methods or require changes in the settings

Concretely, this involves creating a clear separation of concerns between settings and methods, as well as designing the settings and methods in a modular fashion, such that specific components can be replaced or extended by potential new subclasses.

3.3.4. Adaptability

In the worst case, a method for problems of type A, applied to a problem $b \in B$ where $B \subset A$, might not be an effective or efficient solution for problem b or other problems of type B, but should always *work*. In the best case, a method should be able to adapt itself to the particular problem they are being applied to by making use of the additional assumptions present in order to solve the problem more effectively and achieve better performance.

For example, it *should* be possible to use an algorithm designed for bandit settings (where the feedback only includes whether the prediction was correct or not, and doesn't contain the correct prediction) on a typical classification setting.

3.4. Hierarchies in the wild

This section shows examples of problem hierarchies from different fields of ML, as well as describe related works that introduce taxonomies or hierarchical organizations of research settings.

3.4.1. MDP variants

MDPs and POMDPs are some of the most widely-used theoretical frameworks in Reinforcement Learning. The relation between these abstractions - in other words, the differences in assumptions between these settings - are very clearly defined, and often already represented hierarchically (Figure 3.1 from [**14**] shows an example of such an organization).

A Markov Decision Process (MDP) is defined as a tuple $M = \langle S, A, r, p, \gamma \rangle$, where S is the state space, A is the action space, $r : S \times A \to \mathbb{R}$ is the reward function, $p : S \times A \to S$ the state transition function, and $\gamma \in [0, 1)$ is the discount factor. The goal of an agent is to maximize the average sum of discounted future rewards at a given step, also called the *return*.

POMDPs are an extension of MDPs, where the tuple also includes O, the observation space, and $x : S \to O$, the observation function. At each step, rather than return the next state s_{t+1}, POMDPs return an observation of the current state: $o_t = x(s_t)$. POMDPs therefore generalize MDPs by removing the assumption that the environment's state can be directly observed by the agent. Instead, the agent obtains *observations*, which are a function of the environment's hidden state: $o = x(s)$. Setting x to the identity function therefore simplifies a POMDP back to an MDP.

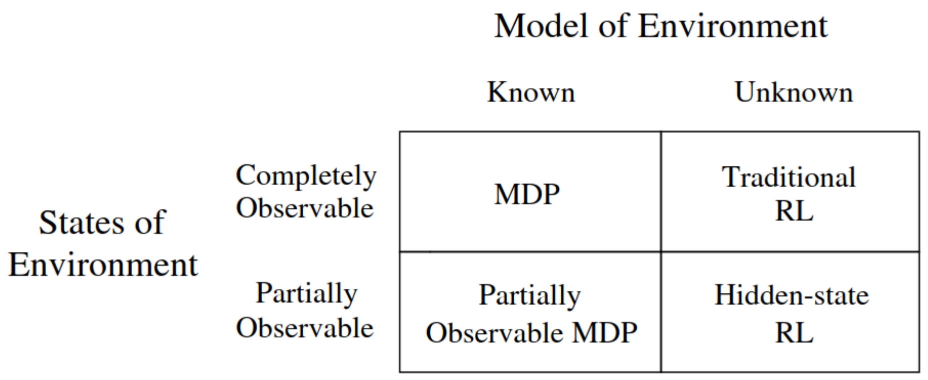
Hidden-Mode Markov Decision Processes (HM-MDPs) are introduced in [**14**] as a generalization of MDPs where the environment dynamics (i.e. the state transition function) can be non-stationary. The environment can be in a finite number of *modes* $z \in \mathcal{Z}$, which affect the transition function $p : S \times A \times \mathcal{Z} \to S$. Hidden-Mode Markov Decision Processes (HM-MDPs) are a subclass of POMDPs, where any pair of mode and observable state in a HM-MDP can be considered as the hidden state of a POMDP, and any observable state of a HM-MDP can be viewed as the observation of a POMDP[**14**]. HM-MDPs will be described in a bit more detail in Subsection 3.5.1.

Many other variants of MDPs and POMDPs exist, such as Hidden Semi-Markov-Mode Markov Decision Processes (HS3MDPs)[**30**], Continuous-Time MDPs, and many more.[3]

3.4.2. Discrete and Continuous action spaces

RL algorithms are typically designed to handle either discrete or continuous action spaces, with some algorithms implementations supporting both (for example, most of the algorithms provided in the Stable-Baselines3[**66**] library support both types of action spaces). Here, we make the claim that discrete and continuous settings can be organized into a hierarchy rather than considered separately. In other words, we claim that the "*discrete-actions*" setting can be viewed as a subtype of the "*continuous-actions*" setting.

[3]We direct readers to [**84, 34, 14, 37**] for a more thorough description of MDPs and their variants.



States of Environment \ Model of Environment	Known	Unknown
Completely Observable	MDP	Traditional RL
Partially Observable	Partially Observable MDP	Hidden-state RL

Fig. 3.1. Reproduction of Figure 1 and its caption from [**14**]: *"Categorization into four related problems with different conditions. Note that the degreee of difficulty increases from left to right and from upper to lower"*[**14**]

A counter-intuitive claim can be made as a consequence of this organization: methods for continuous-action settings, by construction, can always be applied to discrete-action settings. [4] Note that, following the principle of adaptability from Subsection 3.3.4, an algorithm designed for continuous action spaces shouldn't necessarily be expected to perform well in an environment with discrete actions, but it should nonetheless be at least applicable to such an environment.

We provide an informal proof of this claim here. Given any environment or MDP with a discrete action space, it is always possible to add an adapter or wrapper around the environment, such that it is presented as having a continuous action space. Such a transformation doesn't need to be entirely theoretically sound in order to prove this point. For instance, given an environment with a discrete action space with N possible values, we can apply a wrapper around the environment, that presents itself as having a continuous action space of N real values. When these values are provided to the adapter by the agent, they could for instance be used as the logits of a categorical distribution, passed through a softmax, and sampled from, resulting in a discrete value between 0 and $N - 1$, which is then used as the discrete action that is passed to the original environment.

This particular transformation serves as a simple, generic, and universally applicable example, but there are much more efficient ways of adapting specific continuous action algorithms for the discrete action case. One such example is the discrete variant of the Soft Actor Critic algorithm[**16**], where the soft state-value calculation, temperature and policy objectives can be simplified in the case of discrete actions, leading to performance comparable to the state-of-the-art algorithms for discrete actions in terms of sample-efficiency.

[4]Note: We're not claiming that they should, only that they could!

3.4.3. Problem Hierarchies in Continual Learning

Here we describe problem hierarchies in CL, which were already hinted at by Figure 2.2. We begin by describing related works in ML/CL theory that classify and organize different research settings from the literature. We then propose our own hierarchical organization of the settings in CL in Section 3.5. This hierarchy is then implemented concretely in Sequoia, which is described in Chapter 4.

3.4.4. Related work: Towards a theory of out-of-distribution learning

Having a solid theoretical foundation can be very helpful to more applied research in ML, as it informs future research, and deepens our understanding of the problems and solutions within that field. However, in many newer fields of ML, theory often lags far behind practical research. This is particularly the case in research fields such as continual/lifelong, meta, and multi-task learning, where the problem involves out-of-distribution and/or non-stationary learning, which makes it difficult to provide theoretical guarantees using the tools and frameworks from more conventional fields of ML.

In [**26**], the authors propose a new unified theoretical formalism for out-of-distribution learning problems. Their formalism is very flexible, and is able to describe virtually all forms of out-of-distribution learning, including continual, life-long, meta, multi-task, and transfer learning. The authors also suggest a hierarchical organization of all the settings covered by their formalism, as displayed in Figure 3.2, which is a replica of Figure 1 from [**26**].

The authors construct their framework through an iterative, bottom-up approach. Starting from an initial definition for an in-distribution learning problem, the authors iteratively expand this definition by introducing a new term or component to their definition, relaxing one assumption at a time. Each step along the way corresponds to a new, slightly more general setting, which can be simplified into the previous by specifying the added term to a particular value. Once the most general, all-encompassing definition of a learning problem is established, each setting can be recovered by specifying one or more components of the definition.

The authors of [**26**] do not claim that methods defined for one setting should be applicable to their children in their hierarchy. If there were to be a software implementation of the hierarchy of settings illustrated in Figure 3.2, having the learning problem broken down into its individual components could be of great benefit, from a software design perspective. For instance, introducing a new group of assumptions, all related to the same component (e.g. the statistical model or dataset) wouldn't have any effect on the structure of the overall hierarchy. Plans for future work related to implementations of problem hierarchies may

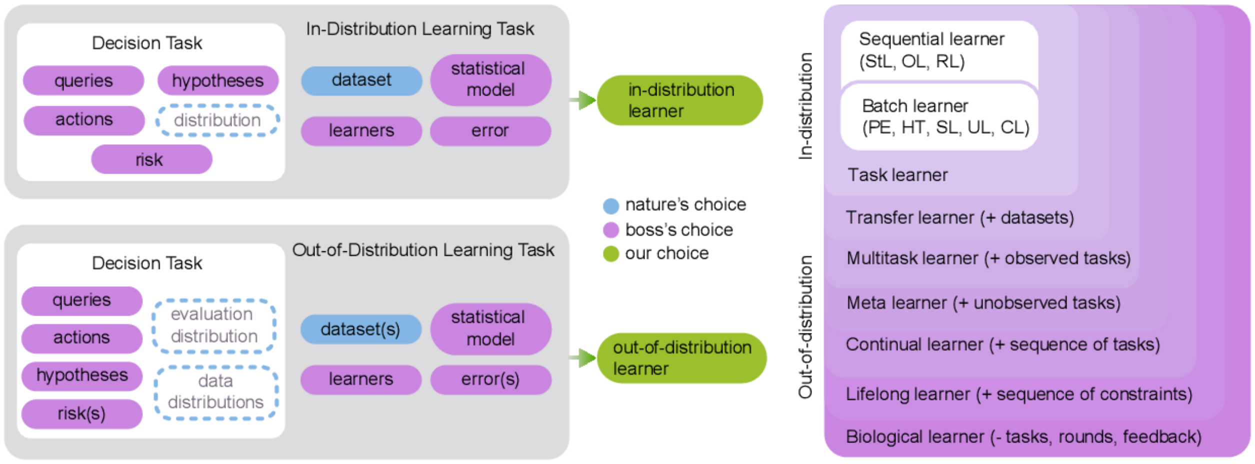

Decision Task
queries
hypotheses
actions
distribution
risk
In-Distribution Learning Task
dataset
statistical model
learners
error
in-distribution learner
Decision Task
queries
actions
hypotheses
risk(s)
evaluation distribution
data distributions
Out-of-Distribution Learning Task
dataset(s)
statistical model
learners
error(s)
out-of-distribution learner
nature's choice
boss's choice
our choice
In-distribution
Out-of-distribution
Sequential learner (StL, OL, RL)
Batch learner (PE, HT, SL, UL, CL)
Task learner
Transfer learner (+ datasets)
Multitask learner (+ observed tasks)
Meta learner (+ unobserved tasks)
Continual learner (+ sequence of tasks)
Lifelong learner (+ sequence of constraints)
Biological learner (- tasks, rounds, feedback)

Fig. 3.2. Figure 1 from [**26**]: (Left) Decision tasks (top) are composed of five components, and the goal is to choose a hypothesis based on the known distribution that minimizes risk. In an in-distribution learning task (middle), the distribution is not available, so a feasible in-distribution learner must leverage a data set, to find a hypothesis that minimizes error under an assumed statistical model. In out-of-distribution learning tasks (bottom), the distribution over queries need not be about the assumed distribution of the data, and there may be multiple data sets, risks, and errors. Each component is provided by one of three different actors: nature, the boss, or the machine learning practitioner. (Right) Schematic illustrating the nested nature of learning problems. PE = point estimation; HT = hypothesis testing; SL = supervised learning; UL = unsupervised learning; CL = causal learning; StL = streaming learning; OL = online learning; RL = reinforcement learning.[**26**]

include establishing a fully-qualified definition for a learning problem similarly as in [**26**], and are described further in Chapter 5.

3.4.5. Related work: Towards Continual Reinforcement Learning: A Review and Perspectives

In [**37**], the authors introduce a taxonomy of the different continual learning settings in RL, while also drawing parallels with equivalent settings in SL.

Recall, a MDP is defined as a tuple $M = \langle S, A, r, p, \gamma \rangle$, where S is the state space, A is the action space, $r : S \times A \rightarrow \mathbb{R}$ the reward function, $p : S \times A \rightarrow S$ the state transition function, and $\gamma \in [0, 1)$ the discounting factor. In a POMDP, the tuple expands to also include O, the observation space, and $x : S \rightarrow O$, the observation function. At each step, rather than return the next state s_{t+1}, POMDPs return an observation of the state: $o_{t+1} = x(s_{t+1})$. The authors of [**37**] first create the most general form of non-stationary RL problem imaginable, by assuming that any of the components of the POMDP could be time-dependent. This is illustrated in Figure 3.3.

The most general hypothetical setting considered by the authors is one where each one of four selected components of the POMDP, namely the state transition function, the reward function, the action space, and the observation function, might be non-stationary (and thus depend on time). For each component, the authors consider four possible levels of non-stationarity:

- Stationary: The outputs of the function are independent of time, or depend on a stationary task distribution with a fixed distribution over tasks.[**37**]:
- passive non-stationarity: The outputs of the function depend on time, independently of the other inputs. In other words, in a given context or task, the function can be considered stationary. In other words, situations where the evolution of the tasks does not depend on the agent's behaviour.[**37**]:
- active non-stationarity: The outputs of the function depend on time, and the non-stationarity of the function is depends on the agent's behaviour.
- Hybrid non-stationarity: There is both passive and active non-stationarity.

The Continual Reinforcement Learning (CRL) classification from [**37**] can be also described using a hierarchy of assumptions. More precisely, as the authors point out, the classification is the combination of a choice of the *degree* of non-stationarity, along with the *kind* of non-stationarity for each term. With four terms considered, and four types of non-stationarity, the total number of distinct hypothetical settings considered would be $4^4 = 256$, however the authors consider the number of non-stationarity terms, without describing each combination of non-stationary and stationary terms, limiting the number of potential settings to 16.

The authors do not appear to make any claims with regard to the applicability of algorithmic solutions across different types or degrees of non-stationarity. However, intuitively, one might imagine that, for instance, an algorithm that is able to solve CRL problems with hybrid non-stationarity of degree four, i.e. the most challenging type of CRL problem imaginable, would probably also be applicable to more restricted problems, where only some terms are non-stationary, or where the non-stationarity is of a simpler type.

One question that arises from this idea is whether passive non-stationarity is a necessary precondition for active non-stationarity. In other words, whether it is impossible for one term of the POMDP to be truly only *actively* non-stationarity, without any form of *passive* non-stationarity whatsoever.

We suggest that that this is the case, given that passive non-stationarity can be defined in terms of the function depending on time (or some context variable), while active non-stationarity is defined as the function depending on both time (or the context variable) as well as the agent's actions in the environment. Assuming this to be correct, we suggest that these assumptions can be organized into a problem hierarchy, and that an algorithm or method able to handle active non-stationarity in a given term, should also be able to handle passive non-stationarity.

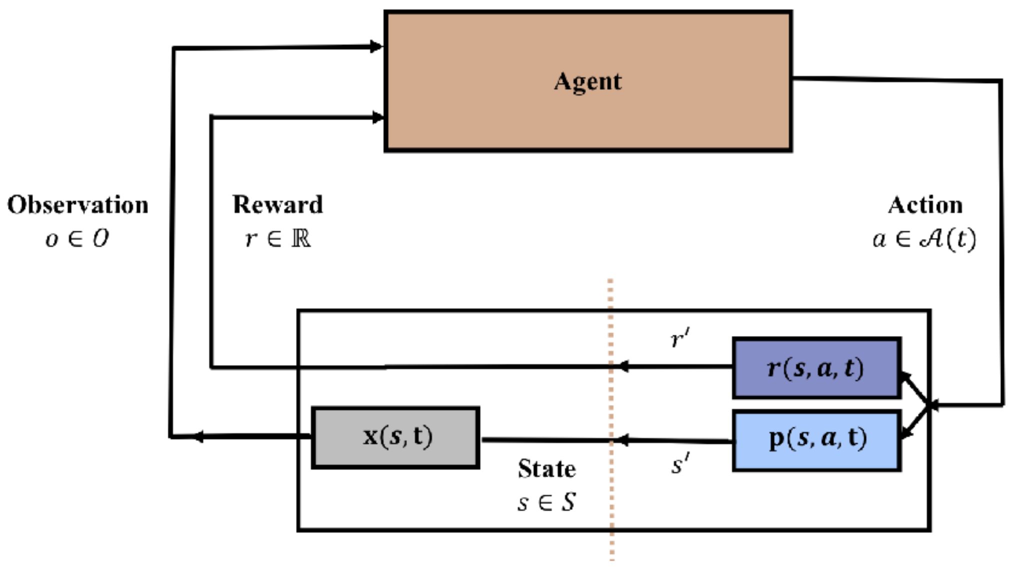


Fig. 3.3. Agent-Environment Interaction with Potentially Time Dependant Environment Components (Figure 1 from [**37**]). The environment here is a POMDP where each component may be time-dependant and non-stationary.

3.5. A Hierarchical Organization of Continual Learning Settings

In this section, we construct a problem hierarchy of CL settings. We begin by identifying the most widely used types of assumptions present in CL settings. We then introduce another two other families of assumptions, which are orthogonal to the CL problem and allow us to recover both CRL or Continual Supervised Learning (CSL). Combining these two major families of assumptions produces our hierarchy of CL settings, which can be seen in Figure 3.4. We present our CL hierarchy using a top-down approach, where assumptions are added progressively to recover more familiar, traditional settings.

We formalize the framework using a generalization of the Hidden-Mode Markov Decision Process (HM-MDP) [**14**], which, as described in Subsection 3.4.1, is itself a special case of a POMDP [**34**].

Recall from Subsection 3.4.1 that an MDP is defined as a tuple $M = \langle S, A, r, p, \gamma \rangle$, and that POMDPs add an observation space O and an observation function $x : S \mapsto O$.

A HM-MDP consists of the same components as a POMDP, with the addition of a context variable z and context space $\mathcal{Z}$ where $z \in \mathcal{Z}$. We also refer to contexts as tasks. The hidden context variable $\boldsymbol{z} \in \mathcal{Z}$ also has an effect on the dynamics of the environment or state transition function $p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a},\boldsymbol{z})$ for states $\boldsymbol{s}',\boldsymbol{s} \in \mathcal{S}$ and action $\boldsymbol{a} \in \mathcal{A}$. The feedback function $r(\boldsymbol{s},\boldsymbol{a},\boldsymbol{z})$ provides an agent $\pi(\boldsymbol{a}|\boldsymbol{o})$ (e.g. a supervised model) with a reward, which indicates the value of performing particular actions $\boldsymbol{a}$ after observing $\boldsymbol{o}$ in that environment. Subsection 3.5.2 explains how the output of this function corresponds to targets in SL and

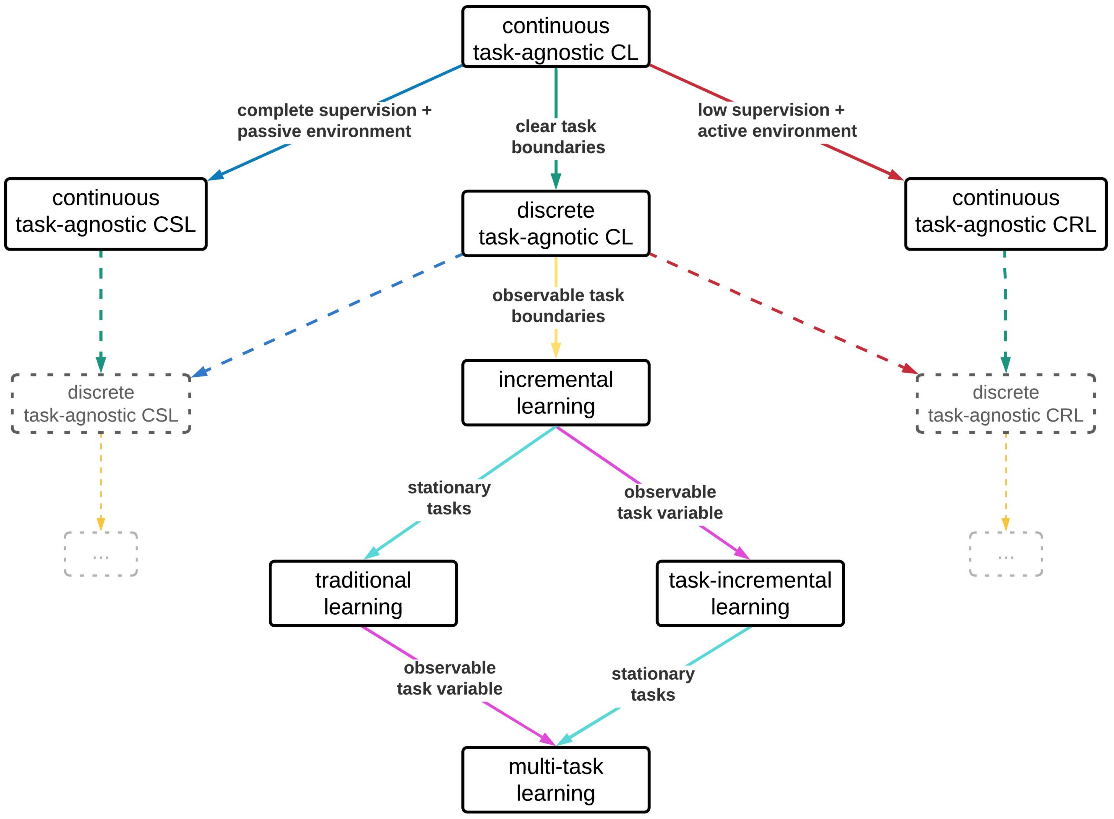

continuous task-agnostic CL
complete supervision + passive environment
clear task boundaries
low supervision + active environment
continuous task-agnostic CSL
discrete task-agnotic CL
continuous task-agnostic CRL
observable task boundaries
discrete task-agnostic CSL
incremental learning
discrete task-agnostic CRL
stationary tasks
observable task variable
...
traditional learning
task-incremental learning
observable task variable
stationary tasks
multi-task learning

Fig. 3.4. Hierarchy of Continual Learning Research Settings. (Figure 2 from our own previous work) Continual learning research settings can be organized into a tree, in which more general settings (parents) are linked with more restricted settings (children) by the differences in assumptions between them. Settings generally become more challenging the higher they are in this hierarchy, as less information becomes available to the method. The central portion of the tree shows the assumptions specific to CL, while the highest lateral branches indicate the choice of either supervised or reinforcement learning, which we consider to be orthogonal to CL. By combining either with the central assumptions, settings from Continual SL and Continual RL can be recovered to the left and right, respectively.

rewards in RL. The context variable follows a Markov chain $p(\boldsymbol{z}'|\boldsymbol{z})$, and therefore does not depend on the agent's performance (as in the *passive* non-stationarity case described by [**37**].) The non-stationarity of the context enables modelling the task/context-changes of CL settings. A change in the context variable is called a *task boundary*.

In the next section (Subsection 3.5.1) we show that by restricting the different elements of the HM-MDP we recover different CL settings. Then in Subsection 3.5.2 we discuss differences between continual supervised (CSL) and continual reinforcement learning (CRL). We end in Subsection 3.5.3, by presenting additional assumptions that are relevant to CL problems.

3.5.1. Continual Learning Assumptions

Assumptions related to CL can be arranged into a hierarchy, as illustrated in the the central portion of Figure 3.4. These settings cover most, if not all the current CL literature. We start from the most general setting: continuous task-agnostic CL [**89**] and add assumptions one by one.

3.5.1.1. Continuous Task-Agnostic CL. is our most general setting. The context variable is continuous $\mathcal{Z} \in \mathbb{R}$. This setting allows for different kinds of drifts in the environment, including smooth task boundaries, i.e. slow drift [**89**]. This setting is *task-agnostic*, meaning that the context variable $\boldsymbol{z}$ is unobserved. Because the context is allowed to drift slowly, it can be more challenging for the methods to infer when a task has changed enough to compartmentalize the recently acquired knowledge before adapting to the new task. In RL, this setting is analogous to the DP-MDP [**86, 11**]. In SL, it has also been studied in e.g. [**89, 2, 3**].

3.5.1.2. Discrete Task-Agnostic CL. assumes clear (or well-defined) task boundaries and so a discrete context variable $\mathcal{Z} \in \mathbb{N}$. In this setting the context can shift in a drastic way, still without the method or agent being explicitly informed. Some cases where this setting has been studied are [**15, 68**] for RL and [**8, 32, 31**] for SL.

3.5.1.3. Incremental Learning. (IL) relaxes the task-agnostic assumption: the task boundaries are observable. This is akin to augmenting the observation with a binary variable that is set to 1 when $\boldsymbol{z}' \neq \boldsymbol{z}$ and 0 otherwise. In doing so, the algorithm does not need to perform *task-boundary detection*. In SL, some well-known IL settings include class-IL and domain-IL distinguished by their *disjoint action space* and *shared action space*, respectively. This is discussed in Subsection 3.5.3.

At this point in the CL hierarchy, the tree branches in two directions, depending on the order of remaining assumptions (see Figure 3.4). We will first explain the right sub-tree.

3.5.1.4. Task-Incremental Learning. (task-IL) assumes a fully-observable context variable available to the agent $\pi(\boldsymbol{a}|\boldsymbol{x},\boldsymbol{z})$. In the literature, observing $\boldsymbol{z}$ is analogous to knowing the *task ID* or *task label*. In this simpler CL setting, forgetting can be prevented by freezing a model at the completion of each task and using the task-ID to retrieve it for evaluation.

The following settings remove the non-stationarity assumption in the contexts/tasks and are often used to set an upper-bound performance for CL methods.

3.5.1.5. Multi-task Learning. removes the non-stationarity in the environment dynamics and the feedback function as it assumes a stationary context variable $p(\boldsymbol{z}'|\boldsymbol{z}) = p(\boldsymbol{z}')$. When the contexts are stationary, there is no *catastrophic forgetting* (CF) [**24**] problem to solve. Multi-task learning assumes a fully-observable task variable.

3.5.1.6. Traditional Learning. branches off incremental CL and assumes a stationary environment. It is the vanilla setting machine learning defaults to. In our framework, it can be seen as a multi-task learning problem where the task variable isn't observable. However, a more natural view of this setting is to simply assume a single task/context.

3.5.2. Supervised Learning and Reinforcement Learning Assumptions

So far we have introduced settings and assumptions that revolve mainly around the type and presence of non-stationarity in the environment and the information observed by the agent. These have allowed us to define the CL problem. To bring all of CL research under one umbrella, we introduce two assumptions, orthogonal to the previous group, to recover RL and SL settings. Methods for a given CL setting that do not make these additional assumptions are therefore applicable to both its CSL and CRL versions, as in [**41, 22**]. Below we use the term observation as a *state* in RL parlance and the actions as *predictions* in SL parlance. We also assume a single context or task.

3.5.2.1. Level of feedback. In RL, the feedback function $r(\boldsymbol{s},\boldsymbol{a},\boldsymbol{z})$ returns a *reward* that informs the agent about the value of performing action $\boldsymbol{a}$ when in state $\boldsymbol{s}$ and context $\boldsymbol{z}$. In SL however, the feedback function is generally both directly known by the agent and differentiable, which allows the agent to simultaneously consider the value of all actions for a particular observation. This feedback is computed based on a *label* when the action space is discrete (classification) or a *target* when it is continuous (regression). The feedback level is a key differentiating feature between RL and SL.

3.5.2.2. Active vs passive environments. In RL, it is generally assumed that the agent's action has an effect on the next observation or state.[5] In other words, the dynamics of the environment $p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a},\boldsymbol{z})$ are action-dependant and we call this an *active* environment. In SL the agent is generally assumed to not influence the next state through its actions, i.e. $p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a},\boldsymbol{z}) = p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{z})$. The environment is thus referred to as being *passive* in these cases.

As seen in Figure 3.4, the two aforementioned assumptions are combined into a single assumption for SL (blue, left) and for RL (red, right). By combining either the RL or SL assumption along with those from the the central CL "trunk", settings from CSL and CRL are recovered. Future versions of Sequoia will decouple these assumptions to enable settings such as bandits and imitation learning, as exemplified in Table 3.1.

[5]The *bandit* setting is one notable exception to this rule.

	low supervision: $f_t := \{r_t\}$ reward only (or reward only for action performed).	*complete* supervision: $f_t := \{r_t, a_t^*\}$ Reward + "*optimal*" action (or reward for all possible actions)
Active environment: $p(s_{t+1} \vert s_t, a_t) \neq p(s_{t+1} \vert s_t)$ Actions affect the environment's state	Reinforcement Learning	Imitation Learning
Passive environment: $p(s_{t+1} \vert s_t, a_t) = p(s_{t+1} \vert s_t)$ Actions don't affect environment state	Bandits / Multi-Arm Bandits	Supervised Learning

Table 3.1. Assumptions related to the level of supervision and type of agent-environment interaction.

3.5.3. Additional Assumptions

Additional assumptions can be added on top of the ones described above to recover additional research settings. For example, a useful assumption in CL experiments is the one of disjoint versus joint action space, i.e. whether the contexts/tasks share a same action space, or whether that space is different for each task. In CSL, this assumption differentiates *class-incremental learning* from *domain-incremental learning* [**82**]. In [**21**], where it is referred as the *shared output space* assumption, a disjoint action space greatly increases the difficulty of a setting in terms of forgetting. In CRL however, the studied settings mostly have a joint action space, with the notable exception in the work of [**10**].

Other assumptions could also be relevant in defining a continual learning problem. For instance, the action space being either discrete or continuous, resulting in classification and regression CSL problems, respectively; a particular structure being required of the method's actions, as in image segmentation problems; an episodic vs non-episodic setting in RL; context-dependant [**8**] versus context-independent feedback functions; and many more.

3.6. Extensions to other fields: Dealing with ill-defined settings

Settings are usually loosely defined. Problem hierarchies, and categories more generally, require unambiguous mathematical definitions. At first glance, these two realities seem hard to reconcile. How can problem hierarchies provide a formal, rigorous definition for the messy and ambiguous types of research problems that are studied in practice? Additionally, there could be some concern that problem hierarchies, as a type system, might not be applicable or self-consistent when applied to more general research settings, with somewhat ill-defined tasks or evaluation procedures.

This section provides a partial answer to these concerns. Here, we extend problem hierarchies to the realm of unsupervised learning and some of its variants. We refer the reader to Chapter 5 for a more in-depth discussion of the practical challenges and lessons learned through implementing problem hierarchies in practice.

3.6.1. Unsupervised Learning

One way to define unsupervised learning is as a parent class of reinforcement learning, (and therefore also supervised learning, by extension), where we remove the assumption that the environment provides a reward signal to the method. This claim that RL and SL are both subtypes of the unsupervised learning setting has counter-intuitive implications. Notably, this implies that any method for unsupervised learning should be directly applicable to a supervised or reinforcement learning setting. This is made possible by the fact that under this definition of the unsupervised learning setting, it is still necessary for an agent to interact with its environment, i.e., to provide an action for each observation. Therefore, when applying an unsupervised learning method to a supervised or reinforcement learning setting, that method will, in the worst case - that is, ignoring the adaptability property of Subsection 3.3.4 - simply ignore the rewards or supervised labels.

There is no clear notion of *task* in the unsupervised learning setting. How best to compare the performance of two unsupervised learning algorithms remains unclear. Recall, we defined a setting as an evaluation procedure for a method or learning algorithm. We also suggested that the output of applying a method to a learning problem from a setting is a scalar *result* that represents the performance of that method on that particular learning problem. Unsupervised Learning can therefore initially be viewed as a counter-example to this definition of a setting, since it does not contain an explicit task. Assuming the above definition for the unsupervised learning setting, this ambiguity with respect to the evaluation of unsupervised learning methods can be addressed in different ways. For instance, we might assume that the environment in the unsupervised setting has no explicit task, and that the setting simply returns the same result for all methods that are applied onto it. This would be a sufficient, albeit uninformative way to define unsupervised learning while remaining consistent with the definition of the setting and method from Section 3.2. Another equivalent option would be to assume that there is a task in the environment, but that it is never observed by the method. This would mean that unsupervised learning methods would be evaluated according to a criteria they are not made aware of.

Alternatively, one could assume that there is a task, but that it is only observed at evaluation time. In other words, the environment only provides feedback during evaluation. In this scenario, the agent might then make use of the understanding of its environment acquired during unsupervised training in order to succeed better at a supervised evaluation task. This also assumes that methods are allowed to train/adapt themselves at test-time.

This option, while quite different to the other "purely unsupervised" variants, has the benefit of more accurately reflecting how many unsupervised learning methods are evaluated today. For instance, in the unsupervised representation learning setting, where we assume that the method has to learn a *model* or encoder, one such evaluation procedures is referred to as *linear probing*. In this procedure, after a method finishes training in an unsupervised environment, the model's representations of the dataset are extracted, and are fed to some simple output network (e.g. a single-layer classifier), as part of some supervised evaluation task, also referred to as a *downstream task*. The performance of the output network on this downstream task (e.g. classification) is used as a measure of the performance of the unsupervised method that learned the representations.

3.6.2. Semi-Supervised Learning

The Semi-Supervised Learning setting can be defined as a subclass of unsupervised learning, which adds the assumption that a supervised task is defined in the environment, and that feedback is sparsely available to the method. In other words, Semi-Supervised Learning can be viewed as an unsupervised learning setting where the environment *sometimes* provides feedback (as in Table 3.1).

3.6.3. Self-Supervised Learning

Self-Supervised Learning (SSL) can be defined as a subclass of unsupervised learning, where additional assumptions are made about the environment and its properties, allowing methods to create their own supervised tasks based on these assumptions. Methods (and their agents) use these innate properties or assumptions about their environment to construct additional self-supervised rewards, which can, for example, be used to learn representations or to learn a model of the environment.

Self-supervised learning methods can be very general and widely applicable. For instance, self-supervised learning objectives can be easily combined with supervised / reinforcement learning objectives as well other self-supervised objectives. (See Appendix A for an illustrative example of how self-supervised learning objectives can be used in the continual learning setting.) Model-based reinforcement learning is also arguably related to self-supervised learning, in that model-based reinforcement learning agents often learn a model of their environment through some form of self-supervision, for example by learning the dynamics of the environment, the effect of actions on future observations, etc. For example, some self-supervised reinforcement learning methods use self-supervised rewards in order to encourage the agent to visit unexplored regions of the state-space[**64**]. Other methods use self-supervision to learn the reversibility of potential actions[**28**] and use it as an auxiliary self-supervised reward signal, leading to better policies in certain environments where there is a potentially high risk associated with some actions, or where rewards are not available.

Summary

In this chapter, we described problem hierarchies, a system to create taxonomies of learning problems. Problem hierarchies, as type hierarchies, benefit from the same properties (e.g. polymorphism), which greatly facilitates the development of widely-applicable methods, the reuse of prior work, and the comparison of new methods with existing baselines. We first defined the main concepts of assumption, method, and setting, and provided a set of considerations for building problem hierarchies from any field of research. We then provided examples of existing problem hierarchies and showed how this system can be used to describe various fields of research. We then introduced our own hierarchical organization of the settings in the field of CL.

The following chapter will describe Sequoia, our implementation of this problem hierarchy for CL. Chapter 5 will then provide a discussion of the practical challenges involved with implementing problem hierarchies, as well as discuss future work related to Sequoia.

Chapter 4

Sequoia

Note to reader:
Portions of this chapter were adapted from a submission to the CoLLAs conference.

In this chapter, we introduce *Sequoia*, an open-source Python library based on problem hierarchies. Sequoia instantiates the hierarchy of CL settings described above in code, where each setting corresponds to a class in a tree-shape inheritance hierarchy. Sequoia is explicitly designed to address the issues associated with Continual Learning research previously described in Section 2.3 of Chapter 1.

To address these issues, we begin by establishing a clear, simple interface for both the Setting and Method classes. This interface is flexible enough to allow for different types of methods to be implemented, yet still rigid and clear enough that both settings and methods are safely and easily interchangeable. The design of the Setting and Methods of Sequoia are described in Section 4.1, Section 4.2 and Section 4.4.

Second, to help bridge the gap between the CRL and CSL domains, Sequoia uses the *Environment* as the interface between methods and settings. This class extends the familiar `Env` abstraction from OpenAI `gym` to also include supervised learning datasets. This makes it possible to develop methods that are applicable in both the CRL and CSL domains. This will be further discussed in Section 4.3, and a comprehensive list of the environments explicitly supported by Sequoia is given in Section 4.5.

Finally, Sequoia uses inheritance to make methods directly reusable across settings. By organizing research settings into an inheritance hierarchy, along with their environments, observations, actions, and rewards, Sequoia enables methods developed for any particular setting to be applicable onto any of their descendants through polymorphism. This mechanism has the potential to greatly improve code reuse and reproducibility in CL research. We illustrate this in Section 4.6, where we describe the methods available in Sequoia.

Section 4.7 will then provide a demonstration of the kind of large-scale empirical studies which are made possible through the use of this new framework.

Aside - Relation with other frameworks

We wish to emphasize that Sequoia is not competing with existing tools and libraries that provide standardized benchmarks, models, or algorithm implementations. On the contrary, Sequoia benefits from the development of such frameworks, since its primary function is to serve as an organized catalog of the algorithms (methods) and benchmarks (settings) that already exist in the research software ecosystem.

In the case of libraries that introduce standardized benchmarks, they can be used to enrich existing settings with additional datasets or environments, or even to create entirely new settings. This is for instance what is done in Sequoia with Continuum[**17**] and CTrL[**83**], which are used to create the datasets of their equivalent CSL settings in Sequoia.

Likewise, libraries or frameworks that introduce new models or algorithms can also be used to create new methods or to add new backbones to existing methods within Sequoia. This is the case for Avalanche [**52**], whose algorithms are directly reused and made available as methods targetting the CSL settings of Sequoia. In RL, the Stable-Baselines3[**66**] and Continual World [**85**] libraries also have their algorithms made available as methods in Sequoia.

External repositories can also register their own methods through a simple plugin system. The end goal for Sequoia is to provide the research community with a centralized catalog of the different research frameworks and their associated methods, settings, environments, etc. The following sections will show examples of such extensions.

4.1. Designing a Setting / Method API

Establishing a distinction between research settings and methods makes a lot of sense, in principle. To implement this idea in practice, however, requires making a number of important design decisions. There already exists a number of design approaches and software frameworks that create such a separation between learning algorithms and the data they are trained or evaluated on. This section will provide an overview of some of these approaches, describing their main abstractions and components of interest, as well as discuss their strengths. We then describe how we reuse and extend these ideas in order to design the Setting and Method classes of Sequoia in Section 4.2 and Section 4.4, respectively.

4.1.1. Related work: OpenAI Gym

OpenAI's Gym library was previously mentioned in Subsection 2.2.2. Sequoia's design is heavily inspired by OpenAI Gym. In this section, we describe the main abstractions of the gym library, as well as some of the design decisions that we believe have lead to its success.

The OpenAI Gym library [**7**] is ubiquitous in RL research. We argue that its wide adoption can be attributed to its simple yet flexible environment, space, and wrapper abstractions.

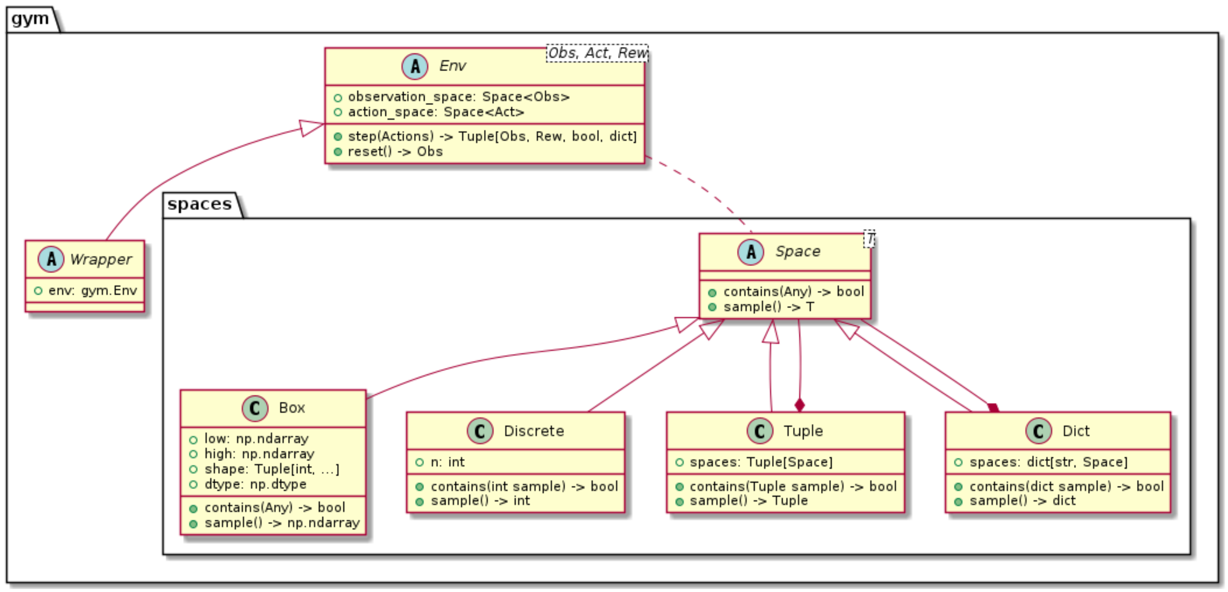


Fig. 4.1. UML diagram of the main components of the Gym API.

These abstractions enable the creation of general agents that can be easily applied on different types of environments. We refer readers to Listing 4, which showed type-annotated pseudocode for the environment and space abstractions of OpenAI gym. Figure 4.1 shows a diagram of the main components of Gym.

4.1.1.1. Environment. Gym environments have two main methods: `step` and `reset`. `step` receives actions from an agent, and performs a single step in the environment, returning the next observation, the rewards at that step, an episode termination signal (a.k.a. `done`) as well as a dictionary containing information useful for debugging (info). `reset` is used to start a new episode, and returns the initial observations.

By establishing an interface for the environment, rather than for the actor (as is/was done in other RL frameworks), Gym places effectively no constraints on the kind of methods that can be used on a particular environment.

4.1.1.2. Space. Environments use spaces to describe the domain, shape and type of the observations they produce as well as the actions that expect to receive from an agent. Gym includes simple spaces such as `Discrete` for integers and `Box` for arrays of floating-point values, as as well as the `Tuple` and `Dict` spaces, which can be used to construct arbitrarily nested structured spaces. These spaces can be used to describe the observations or actions of virtually any environment imaginable.

Models can be created based on the properties of the observation and action spaces, rather than for a specific environment. This has the great benefit of making these models agnostic to the particular choice of environment, as long as they have the right type of

```python
# Specific to CartPole:
env = gym.make("CartPole-v0")
actor_network = nn.Sequential(
    nn.Linear(4, 32),
    nn.ReLU(),
    nn.Linear(32, 2),
)
```

```python
# Works on all envs with Box observation and Discrete action spaces:
def make_actor_network(env: Env[np.ndarray, int]):
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(np.prod(env.observation_space.shape), 32),
        nn.ReLU(),
        nn.Linear(32, env.action_space.n),
    )
env = gym.make("CartPole-v0")
actor_network = make_actor_network(env)
```

Listing 8. Example of how Spaces of Gym enable the creation of more general RL methods. The type annotations in this example indicate the *assumptions* made by the actor about the kinds of environments it can interact with, namely, environments whose observations are arrays, and whose actions are integers.

observation or action space. For example, consider Listing 8, where an actor network for the CartPole environment is created.

An alternative and perhaps interesting view of Listing 8 is that the actor is actually describing a *space of environments* in which it is applicable, which itself is a subset of the *space of all environments*. Such abstractions can even be very easily and succinctly implemented as gym spaces, and are likely to play a critical role in future work related to Sequoia. This idea will be expanded upon in Chapter 5.

4.1.1.3. Wrapper. The Wrapper class from gym is also a great contributing factor to its success. Wrappers are objects that have an environment as an attribute, and delegate all method calls and attribute accesses to it by default, but can also modify the actions, observations, and rewards before they are passed to the wrapped environment. Wrappers are used to create reusable transformations or operations that can be applied on different environments. Wrappers can also modify the observation or action spaces to reflect their corresponding new outputs / inputs. They prevent boilerplate logic, for example, flattening or resizing observations, from having to be added to the agent or environment directly. The Wrapper class is a very elegant way to add transformations or adapters between the environment and the agent. The Gym library comes with a variety of wrappers that can be useful for transforming images, tracking episode statistics, recording videos, and much more.

4.1.2. Related work: PyTorch-Lightning

PyTorch-Lightning (PL)[**18**] is a framework used to structure and organize PyTorch[**63**] research code. It is rapidly gaining popularity, particularly in the SL and SSL research communities. When using PL, research code is split up into different simple components, each with its own distinct responsibilities:

(1) the `LightningModule`: contains the research logic (optimizer and loss calculation);

```
class Wrapper(Env[Observation, Action]):
    def __init__(self, env: Env):
        self.env = env
        self.observation_space = env.observation_space
        self.action_space = env.action_space
    def reset(self) -> Observation:
        return self.env.reset()
    def step(self, action: Action) -> Tuple[Observation, float, bool, dict]:
        return self.env.step(action)
```

Listing 9. Minimal reimplementation of the Wrapper class from Gym.

(2) the Trainer: performs training and evaluation. Contains all the engineering code required for high-performance training.
(3) the `LightningDataModule` (also refered to as *DataModule*): creates the training, validation, and testing dataloaders;

The LightningDataModule class is of particular interest to this work, since we build upon it to create the Setting class of Sequoia, as will be described later in Section 4.2. PyTorch-Lightning comes with a ton of features, but its arguably biggest selling-point is how well it enables scaling up research code to use multiple GPUs, mixed precision, model parallelism, and other high-performance features that would otherwise require significant changes to the model.

4.1.2.1. LightningDataModule. The LightningDataModule is an abstraction for an object that contains all the data-related logic. It's responsibilities are simple: to create the training, evaluation and testing data that is used by the Trainer to train the model. The minimal interface of this class is shown in Listing 10.

```
class LightningDataModule(Generic[T]):
    def prepare_data(self): ...
    def setup(self, phase: str): ...
    def train_dataloader(self) -> DataLoader[T]: ...
    def val_dataloader(self) -> DataLoader[T]: ...
    def test_dataloader(self) -> DataLoader[T]: ...
```

Listing 10. Interface of the LightningDataModule class from PyTorch-Lightning.

4.1.2.2. LightningModule and Trainer. The LightningModule class inherits from the familiar nn.Module class of PyTorch and adds methods and callbacks that make it simpler to scale up and distribute the training of the module. LightningModules are required to implement a forward pass (just like regular nn.Modules), as well as a training step, which should either produce a loss tensor for a given batch of inputs, or perform the optimization manually. Pseudocode for the LightningModule class is shown in Listing 11.

```
class LightningModule(nn.Module, Generic[T]):
    @abstractmethod
    def forward(self, input) -> Any:
    @abstractmethod
    def train_step(self, batch: T) -> Tensor: ...
    @abstractmethod
    def configure_optimizers(self) -> torch.optim.Optimizer: ...
    def val_step(self, batch: T) -> Tensor: ...
    def test_step(self, batch: T) -> Tensor: ...
```

Listing 11. Pseudo-interface for the LightningModule class.

```
datamodule: LightningDataModule = SomeDataModule()
model: LightningModule = SomeLightningModule()
# Create a Trainer that will train with 4 gpus:
trainer = Trainer(max_epochs=100, gpus=4, strategy="dp")
trainer.fit(model, datamodule)
# Evaluate the model
eval_results = trainer.validate(model, datamodule.val_dataloader())
print("Val Results: ", eval_results)
# Test the model
test_results = trainer.test(model, datamodule.test_dataloader())
print("Test Results: ", test_results)
```

Listing 12. Example usage of the LightningDataModule, LightningModule, and Trainer components of PL.

The Trainer contains all the engineering logic, as well as the training, evaluation, and test procedures. A pseudocode representation of the Trainer is available in Listing 26. Listing 12 contains a simple example of how these abstractions can be used.

These two classes are not exactly comparable with the Setting and Method of Sequoia, as it is entirely up to the Method to specify the training procedure, and as such, they are free to choose to use or not use PyTorch-Lightning. Sequoia does come with a “Base” method that uses PyTorch-Lightning that is easy to extend and makes use of the features of PL. This method is the subject of Subsection 4.6.1.

Summary

In summary, the Env from gym is a very flexible and generic abstraction for a source of data, and LightningDataModules have great cohesion, are self-contained, and make it easy to create data-agnostic models. We take advantage of both to create the `Setting` class, which is described next.

4.2. Setting

We implement the `Setting` as a configurable evaluation procedure for a `Method`. The responsibilities of a Setting are to 1) specify when, and on what kind of environments/datasets the method is to be trained and evaluated, and 2) to actually perform the training and evaluation procedure on a given method and to return results. Results should contain a single scalar, called the *objective*, which is to be a measure of how well the Method performed in that particular setting. This objective should be comparable between methods. Results could also contain other metrics relevant to the setting.

Sequoia defines a very simple interface for settings, the `AbstractSetting`, which is then implemented by the `Setting` class, the most general (a.k.a. *root*) setting in our hierarchy. The `AbstractSetting`, shown in Listing 13, has only one required method: `apply`, which should contain all the training and evaluation logic. The `AbstractSetting` doesn't impose any kind of form or restriction on the training/evaluation structure, and is provided as a simple starting point to create new settings or hierarchies for other fields of ML.

```python
class Setting:
    class Results:
        objective: float

    def apply(self, method: Method) -> Results:
        ...
```

Listing 13. The minimal interface for a Setting in Sequoia.

The `Setting` class of Sequoia is the root node of our hierarchy of Continual Learning settings. The `Setting` class reuses and builds upon good ideas from both Gym and PyTorch-Lightning, and is shown in Listing 14.

Concretely, settings in Sequoia create training, validation, and testing environments that a method interacts with, and coordinate the training/testing of the method following the protocol of the corresponding setting in the literature. The main abstractions of Sequoia, as well as the relationship between them are illustrated in the diagram of Figure 4.2.

Settings provide information about the data that Methods will encounter in these environments in advance using Gym spaces. The `observation_space`, `action_space` and `reward_space` properties of the settings reflect the corresponding spaces of their environments. These properties are implemented using a subclass of `gym.spaces.Dict` that produces dict-like objects of the corresponding type defined on the Setting (i.e., `<Setting>.Observations` for `<Setting>.observation_space`, etc). For settings where the environments and their spaces vary between tasks, for instance class-incremental learning where the number of potential *actions* (class predictions) grows over time, these properties on the Setting correspond to the union of the spaces from each environment. All of this makes it so Methods can create their models ahead of time, and these models will be compatible with all environments that will be encountered in that setting.

The Setting class also inherits from the `LightningDataModule` class, and implements the required methods (`train_dataloader`, `val_dataloader` and `test_dataloader`). These methods

```python
# Environment := Union[gym.Env, DataLoader]
class Setting(AbstractSetting, LightiningDataModule):
    class Observations:
        ...
    class Actions:
        ...
    class Rewards:
        ...
    class Results:
        objective: float

    observation_space: Space[Observations]
    action_space: Space[Actions]
    reward_space: Space[Rewards]

    # Inherited from LightningDataModule:
    def prepare_data(self) -> None: ...
    def setup(self, phase: str) -> None: ...
    def train_dataloader(self) -> Environment[Observations, Actions, Rewards]: ...
    def val_dataloader(self) -> Environment[Observations, Actions, Rewards]: ...
    def test_dataloader(self) -> Environment[Observations, Actions, Rewards]: ...

    @abstractmethod
    def apply(self, method: Method) -> Results:
        ...
```

Listing 14. The Setting class: the root of Sequoia's hierarchy of CL settings.

create `Environments` which are both DataLoaders and gym Envs and will be desribed next in Section 4.3. This interface also makes it easier for methods to use PyTorch-Lightning [**19**] to perform high-performance training of their models.[1]

Each setting is created by extending a more general setting and adding additional assumptions. This inheritance relationship from one setting to the next also extends to the objects produced / accepted by the setting's environments (`Observations`, `Actions`, and `Rewards`) they create. Listing 15 illustrates how this subtyping relation between settings and their objects is implemented in Sequoia.

Settings are available for each combination of the CL assumptions, along with the choice of one of RL / SL (as illustrated in Figure 3.4), for a total of 12 settings[2]. These two "branches" (one for CRL and the other CSL) form the basis of Sequoia's eponymous tree of settings. Each setting inherits from one or more parent settings, following the above-mentioned organization. A diagram showing the hierarchy of CL setting classes can be seen in Figure 4.3.

[1]It is important to note that methods are in no way required to use PyTorch-Lightning.
[2]Other common SL settings, such as Domain-Incremental and Class-Incremental learning are also available in Sequoia, but they rely on an additional family of assumption (fixed action space), and are thus omitted from the main portion of this paper.

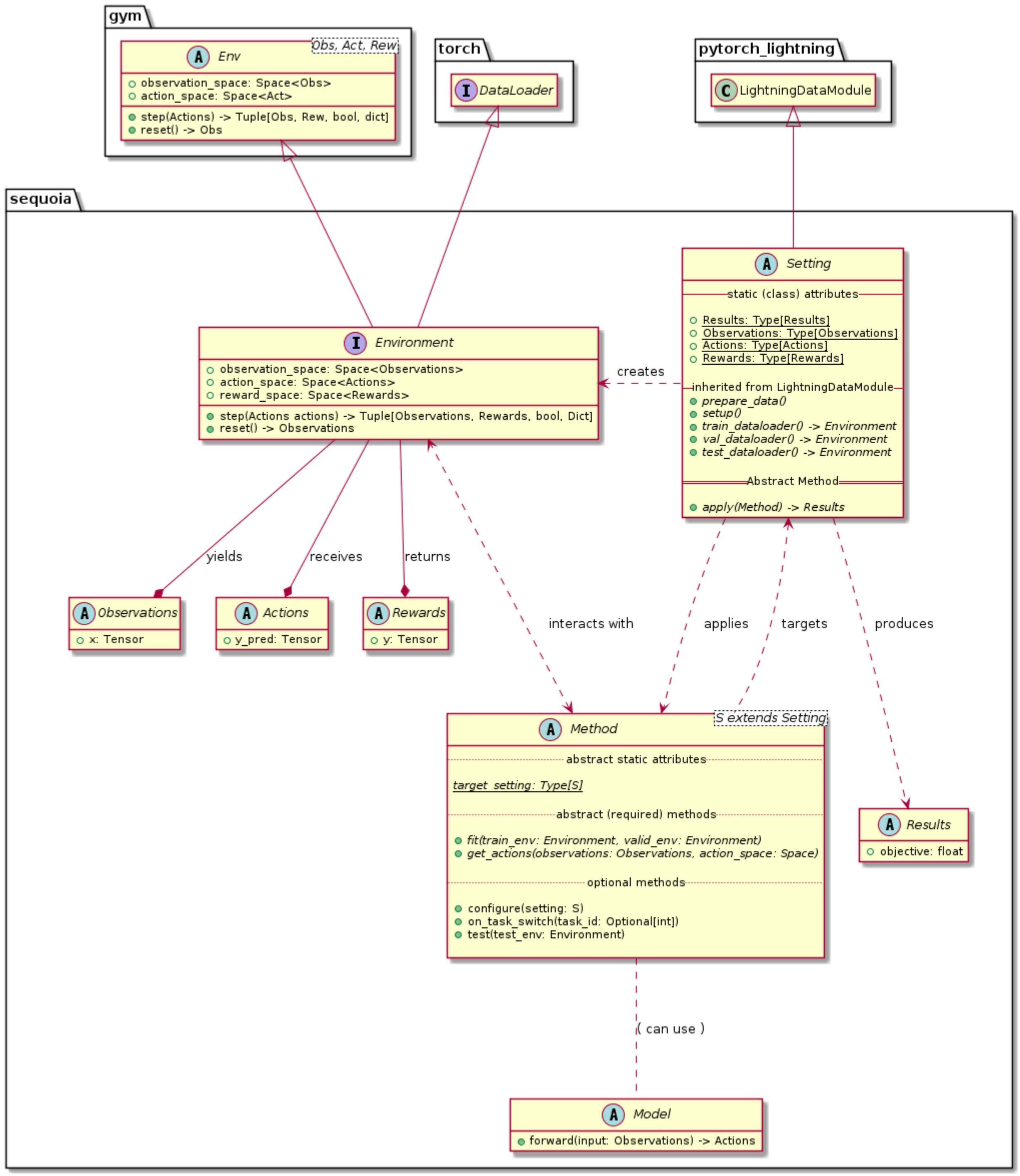

gym
Env
Obs, Act, Rew
observation_space: Space<Obs>
action_space: Space<Act>
step(Actions) -> Tuple[Obs, Rew, bool, dict]
reset() -> Obs
torch
DataLoader
pytorch_lightning
LightningDataModule
sequoia
Setting
static (class) attributes
Results: Type[Results]
Observations: Type[Observations]
Actions: Type[Actions]
Rewards: Type[Rewards]
inherited from LightningDataModule
prepare_data()
setup()
train_dataloader() -> Environment
val_dataloader() -> Environment
test_dataloader() -> Environment
Abstract Method
apply(Method) -> Results
creates
Environment
observation_space: Space<Observations>
action_space: Space<Actions>
reward_space: Space<Rewards>
step(Actions actions) -> Tuple[Observations, Rewards, bool, Dict]
reset() -> Observations
yields
receives
returns
Observations
x: Tensor
Actions
y_pred: Tensor
Rewards
y: Tensor
interacts with
applies
targets
produces
Method
S extends Setting
abstract static attributes
target_setting: Type[S]
abstract (required) methods
fit(train_env: Environment, valid_env: Environment)
get_actions(observations: Observations, action_space: Space)
optional methods
configure(setting: S)
on_task_switch(task_id: Optional[int])
test(test_env: Environment)
Results
objective: float
(can use)
Model
forward(input: Observations) -> Actions

Fig. 4.2. UML Diagram showing the main abstractions of Sequoia.

4.3. Environment

Settings in Sequoia create `Environments`, which adhere to both the `gym.Env` and the `torch.DataLoader` APIs. This makes it easy for SL researchers to transition to RL and vice-versa. These environments receive `Actions` and return `Observations` and `Rewards`. `Observations`

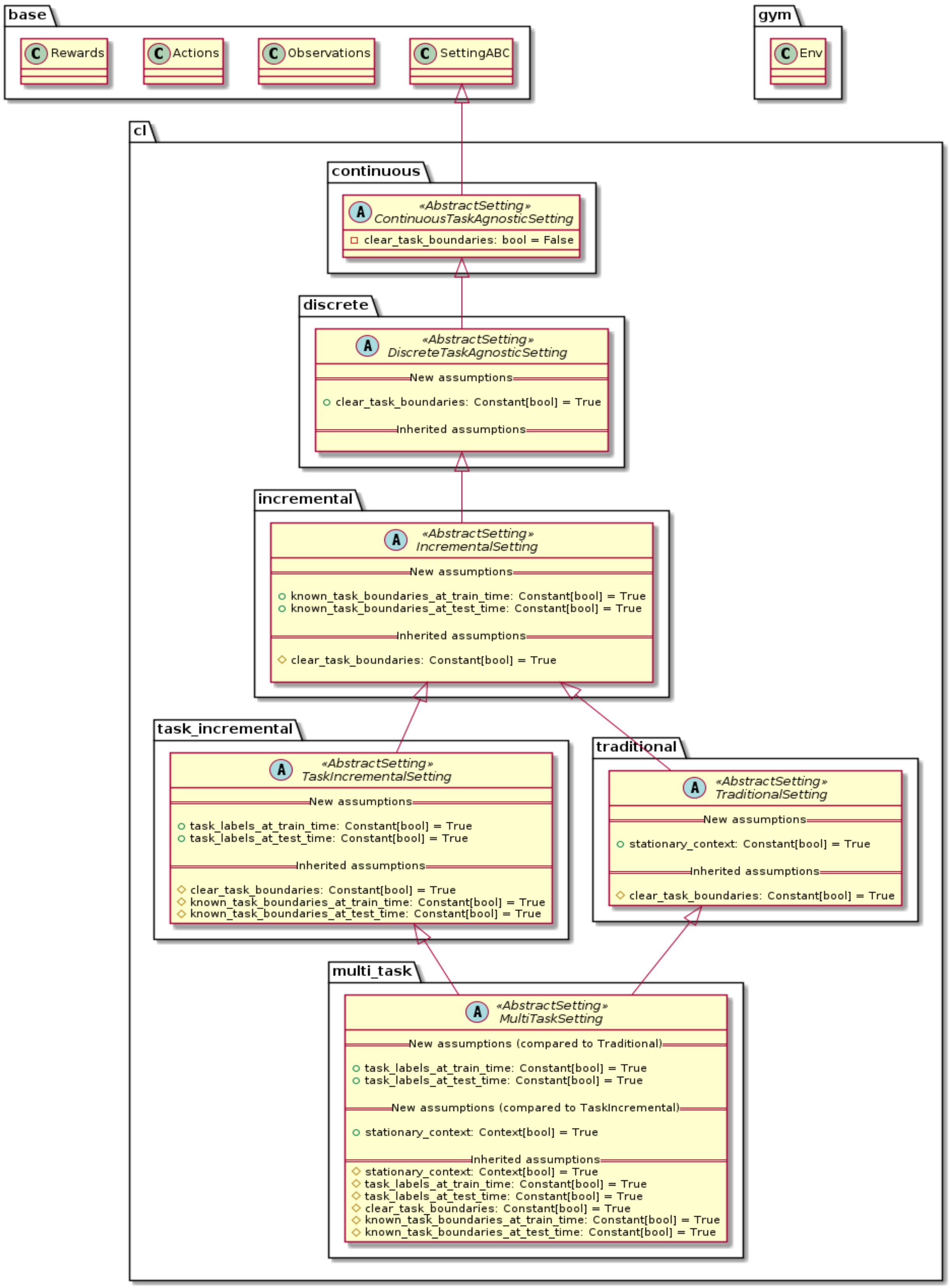

base
Rewards
Actions
Observations
SettingABC
gym
Env
cl
continuous
«AbstractSetting»
ContinuousTaskAgnosticSetting
clear_task_boundaries: bool = False
discrete
«AbstractSetting»
DiscreteTaskAgnosticSetting
New assumptions
clear_task_boundaries: Constant[bool] = True
Inherited assumptions
incremental
«AbstractSetting»
IncrementalSetting
New assumptions
known_task_boundaries_at_train_time: Constant[bool] = True
known_task_boundaries_at_test_time: Constant[bool] = True
Inherited assumptions
clear_task_boundaries: Constant[bool] = True
task_incremental
«AbstractSetting»
TaskIncrementalSetting
New assumptions
task_labels_at_train_time: Constant[bool] = True
task_labels_at_test_time: Constant[bool] = True
Inherited assumptions
clear_task_boundaries: Constant[bool] = True
known_task_boundaries_at_train_time: Constant[bool] = True
known_task_boundaries_at_test_time: Constant[bool] = True
traditional
«AbstractSetting»
TraditionalSetting
New assumptions
stationary_context: Constant[bool] = True
Inherited assumptions
clear_task_boundaries: Constant[bool] = True
multi_task
«AbstractSetting»
MultiTaskSetting
New assumptions (compared to Traditional)
task_labels_at_train_time: Constant[bool] = True
task_labels_at_test_time: Constant[bool] = True
New assumptions (compared to TaskIncremental)
stationary_context: Context[bool] = True
Inherited assumptions
stationary_context: Context[bool] = True
task_labels_at_train_time: Constant[bool] = True
task_labels_at_test_time: Constant[bool] = True
clear_task_boundaries: Constant[bool] = True
known_task_boundaries_at_train_time: Constant[bool] = True
known_task_boundaries_at_test_time: Constant[bool] = True

Fig. 4.3. UML Diagram of the CL assumptions hierarchy, as implemented in Sequoia. The CRL and CSL branches are not shown, but follow an identical structure. Observations, Actions and Rewards for each setting are also omitted here, but also follow the same structure.

```python
class Incremental(Setting):
    class Observations(Setting.Observations):
        x: Tensor
        t: Optional[Tensor]

    class Actions(Setting.Actions):
        action: Tensor

    class Rewards(Setting.Rewards):
        y: Tensor

    class Results:
        ...

    nb_tasks: int
    dataset: str
    # Fixed to True:
    task_labels_at_train_time: Final[bool] = True
    # Either True or False:
    task_labels_at_test_time: bool = False
```

```python
class TaskIncremental(Incremental):
    class Observations(Incremental.Observations):
        x: Tensor
        t: Tensor

    class Actions(Incremental.Actions):
        ...
    class Rewards(Incremental.Rewards):
        ...

    class Results:
          ...
    # inherited members (nb_tasks, etc) not shown
    ...
    # Now fixed to True:
    task_labels_at_test_time: Final[bool] = True
```

Listing 15. Illustration of the subtyping relation between Settings in Sequoia. In this example, the Incremental Learning setting has task labels available at training time, but not at test time (hence the `Optional[Tensor]` annotation in the observations for that Setting). Task-Incremental then adds the assumption that task labels are also available at test-time, and narrows the type of the `t`. Note that this inheritance relationship does not extend to the type of `Results` produced by the settings, as settings are allowed to use different objectives or metrics than their parents.

contain the input samples `x`, and may also contain task labels for each sample, depending on the setting. These objects have the same structure in both RL and SL settings. However, as described in Subsection 3.5.2, in SL, `Actions` contain the predictions, while `Rewards` additionally contain targets or labels. These objects are defined on the `Setting` and follow the same pattern of inheritance as the settings themselves.

4.4. Method

The `Method` class of Sequoia has a simple interface. Methods are required to implement `fit`, which is used for training and validation, as well as `get_actions`, which is used for inference at test-time. Both `configure` and `on_task_switch` are optional hooks: `configure` is called before training so that the method can adapt itself to the Setting, and `on_task_switch` may be called when a task boundary is reached, and may be passed the id of the new task, depending on the chosen setting.

```
class Method(ABC):
    target_setting: Type[Setting]
    def configure(self, setting: Setting):
        ...
    @abstractmethod
    def fit(self, train_env: Environment, valid_env: Environment):
        ...
    @abstractmethod
    def get_actions(self, observations: Observations, action_space: Space[Actions]) -> Actions:
        ...
    def on_task_switch(self, task_id: Optional[int]):
        ...
```

Listing 16. Basic interface for a CL Method in Sequoia.

4.5. Available Environments

One of the main objectives of Sequoia's design is to leverage existing frameworks as much as possible, such that its primary contribution remains to provide a useful organization of their settings and methods within a problem hierarchy.

In order to maximize the reuse of existing frameworks and libraries, Sequoia uses the Environment as its most general abstraction for a learning problem, in both RL as well as SL. The process of adding new environments to Sequoia settings is also designed to be as simple as possible.

Settings are designed in such a way that the environments they use for training, validation, or testing can be swapped out easily. Customized environments can be passed directly as arguments to the Settings, which will then use them as part of their training/evaluation protocol.

This section describes the environments available in Sequoia, both in SL as well as RL. A list of all the supported environments is shown in Table 4.1.

4.5.1. Supervised learning environments

SL settings can be passed one or more datasets, which they wrap and expose to methods as *passive* environments, which are both DataLoaders as well as gym environments. Sequoia uses two open-source packages to create supervised learning datasets: continuum and CTrL, which will be briefly described here.

4.5.1.1. Continuum. Sequoia uses the *Continuum* package [**17**], to create most of its supervised learning datasets. Continuum is a library used to create the datasets and scenarios commonly used in continual supervised learning. It can be configured to create many different types of incremental learning scenarios, including domain-incremental, class-incremental and many more.

4.5.1.2. Continual Transfer Learning benchmark. The Continual Transfer Learning benchmarks (CTrL)[**83**] package is a tool used to generate streams of SL tasks. It can be used to create various different types of CL scenarios, and is easy to configure. The CTrL package comes with a set of 6 benchmarks: S_+, S_-, S_{in}, S_{out}, S_{pl}, and S_{long}, each of which is meant to access a different kind of transfer in a lifelong learning setting. Each of these benchmarks can be used as the source of the datasets used by CSL settings in Sequoia.

4.5.2. Reinforcement Learning environments

Through its close integration with `gym`, Sequoia is able to use any gym-compatible environment as the "dataset" used by its RL settings. Setting create continuous or discrete tasks, depending on the choice of setting and environment.

For settings with continuous tasks, Sequoia samples a sequence of different environment configurations, and the environment is modified over time at each step or episode boundary, using a configuration that is the linear interpolation of the two nearest configurations in the schedule. Tasks can either be constructed automatically, given an environment and desired number of tasks, or passed manually as done in Listing 17. Adding support for generating tasks in other environments is easily done by registering a function to use to sample the tasks in that environment. Examples of how to add new continuous and discrete tasks in an environment are included in Appendix B.1.

```python
from sequoia.methods.stable_baselines3_methods import A2CMethod
from sequoia.settings.rl import ContinuousTaskAgnosticRLSetting

# Note: other types of settings could also be used here:
setting = ContinuousTaskAgnosticRLSetting(
    dataset="CartPole-v1",
    train_max_steps=20000,
    train_task_schedule={
        0: {"gravity": 10, "length": 0.2},
        10000: {"gravity": 100, "length": 1.2},
        20000: {"gravity": 10, "length": 0.2},
    }
)
# Create the method to use here:
method = A2CMethod()
results = setting.apply(method)
print(results.summary())
```

Listing 17. Example of creating a CRL setting by passing a task schedule. For settings with continuous tasks (i.e. the Continuous, Task-Agnostic Continual Learning (CTaCL) setting), the environment is updated continuously using an interpolation of the two nearest configurations. In settings with discrete tasks, the keys represent the task boundaries, as a number of environment steps.

Settings with discrete tasks - in other words, all settings except CTaCL - can use any gym environment whatsoever. This is thanks to the above-mentioned fact that a list of gym environments to use for each task can be passed as an argument to the setting constructor, and are then used as part of the training and evaluation procedure. This was also very useful while developing Sequoia, as it made it much easier to reformulate existing CRL benchmarks as settings.

The benchmarks and environments that are explicitly supported by Sequoia are described below and are also listed in Table 4.1.

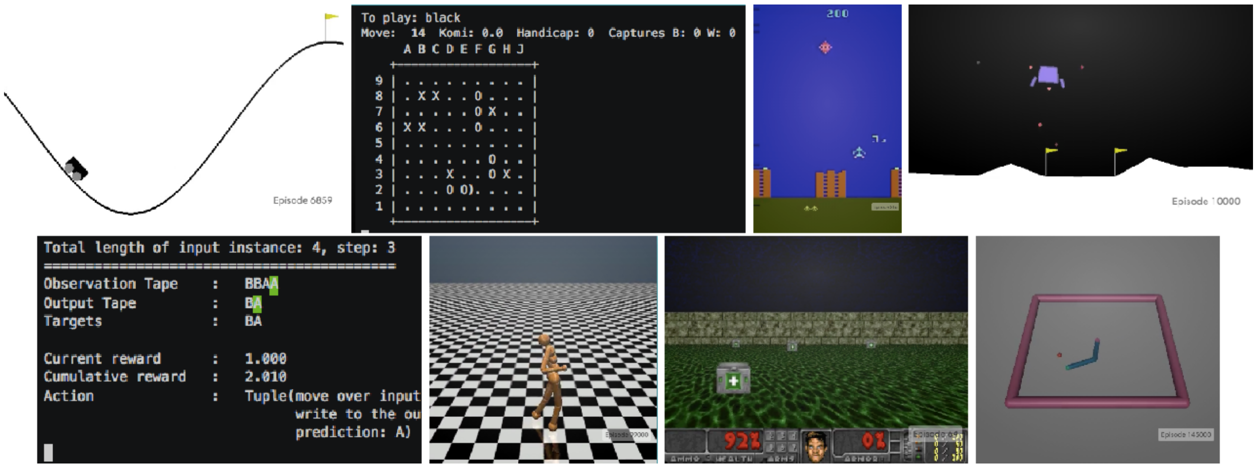
4.5.2.1. Classic control environments. The classic control environment of Gym[**7**] are easily configurable, and are thus a great starting point for CRL experiments. Sequoia supports sampling both continuous and discrete tasks in these environments namely the CartPole, MountainCar, Pendulum, MountainCarContinuous, and more. The tasks in these simple environments correspond to different environment configurations, i.e. different dictionaries mapping from environment attributes to the values to set for these attribute. When Sequoia generates the tasks, they are created by sampling from a normal random variable with configurable standard deviation, centered at the default value of that attribute.

For example, when using one of the classic-control environments from gym such as `CartPole`, tasks are created by sampling a new set of values for the environment constants such as the gravity, the length of the pole, the mass of the cart, friction coefficients, etc.

4.5.2.2. MuJoCO environments. Inspired by [**57**], Sequoia has explicit support for creating CRL settings using the widely-used MuJoCo[**80**]-based gym environments such as `HalfCheetah`, `Walker2d`, and `Hopper`. Similarly to the classic control environments, continuous tasks in these environments are created by introducing changes in the environmental constants such as gravity. Continuous tasks can thus easily be created in this case, as the environment is able to respond dynamically to changes in these values at every step, and the task can evolve smoothly by interpolating between different target values.

Discrete tasks in these environments can also include changes to the mass and scale of the different body parts of the model, in addition to changing the environmental constants such as gravity. These modifications only become available with discrete tasks, as they cannot be performed in real-time on a MuJoCo environment instance, and thus require instantiating one environment per task, each with different masses or limb sizes.

4.5.2.3. MonsterKong. Sequoia also introduces a non-stationary version of the *MonsterKong* Arcade learning environment[**6**], which can be used to create discrete tasks involving changing map configurations. The motivation and reasoning behind the creation of this environment is described in detail in Appendix D.1.



```
To play: black
Move:  14  Komi: 0.0  Handicap: 0  Captures B: 0 W: 0
     A B C D E F G H J
   +-------------------+
 9 | . . . . . . . . . |
 8 | . X X . . O . . . |
 7 | . . . . . O X . . |
 6 | X X . . . O . . . |
 5 | . . . . . . . . . |
 4 | . . . . . . O . . |
 3 | . . . X . . O X . |
 2 | . . . O O). . . . |
 1 | . . . . . . . . . |
   +-------------------+
```

```
Total length of input instance: 4, step: 3
==========================================
Observation Tape     :   BBAA
Output Tape          :   BA
Targets              :   BA

Current reward       :   1.000
Cumulative reward    :   2.010
Action               :   Tuple(move over input
                               write to the ou
                               prediction: A)
```

Fig. 4.4. Replication of Figure 1 from the original Gym paper[**7**]: "*Images of some environments that are currently part of OpenAI Gym.*" Since then, Gym environments have become ubiquitous in RL research.

Methods	SL	`BaseMethod`.{base, EWC, PackNet }, PNN, replay, HAT, CN-DPM
		`Avalanche`.{naive, AGEM, CWR*, EWC, GDumb, GEM, LWF, replay, SI}
	RL	`BaseMethod`.{base, EWC, PackNet }, PNN
		`stable_baselines3`.{A2C, DDPG, DQN, PPO, SAC, TD3}
		`continual_world`.{SAC, AGEM, EWC, VCL, PackNet, L2 reg., MAS, replay}
Environments	SL	`continuum`.{{K,E,Q,Fashion}MNIST, Cifar10(0), ImageNet100(0), Core50, Synbols}
		`CTrL`.{STL10, STL50}
	RL	`gym.envs.classic_control`.{CartPole, Pendulum, MontainCar{Continuous}}
		`gym.envs.mujoco`.{Hopper, Half-Chettah, Walker2d}
		`gym.envs.atari`.{Monsterkong}
		`metaworld`.{MT10, MT50}
		`continual_world`.{CW10, CW20}
Metrics		{Transfer Matrix, forward transfer, backward transfer,
		Average final performance, Online Training Performance} ×
	SL	{loss, accuracy}
	RL	{loss, total reward, average reward, episode length}

Table 4.1. Sequoia's methods, environments and metrics. Incremental settings in Sequoia can be passed custom environments to use for each task. This makes it possible to use virtually any gym environment to create custom incremental RL settings. Likewise, a custom dataset can be passed for each task to create arbitrary incremental SL settings. The environments listed here are those explicitly supported in Sequoia. Note that references for each supported method are omitted here, but included in Table 4.2.

4.5.2.4. Meta-World. Meta-World[**87**] is a library of Gym environments for robotic manipulation tasks, based on the MuJoCo physics engine[**80**]. It includes a wide variety of environments and tasks as well as a set of standardized benchmarks. These benchmarks, namely ML10, MT10, ML45, or MT50, consist in a pre-defined sequence of training environments and tasks, along with an associated testing environment, with the test environment or test tasks being different than those at train time. The difference between the training and testing conditions varies between benchmarks, which makes it possible to study the generalization ability of RL algorithms under different conditions.

Meta-World has become one of the most widely used benchmarks in continual and multi-task RL. The computational requirements of running continual or multi-task RL on the Meta-World benchmarks are quite substantial, however. The MT10, MT50 benchmarks from Meta-World [**87**] can be used in the Discrete CRL settings.

4.5.2.5. Continual World. Continual World[**85**] creates a set of CRL benchmarks based on Meta-World[**87**], by selecting a subset of the tasks, ordering them, and assigning a computational limit for each task. The CW10 and CW20 benchmarks introduced in [**85**] can be used with RL settings of Sequoia. The package also includes with a set of CRL methods, based on a Soft Actor-Critic backbone, which are also made available in Sequoia.

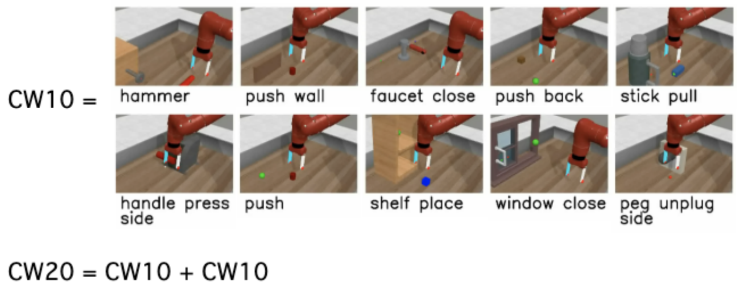


Fig. 4.5. Illustration of the CW10 and CW20 benchmarks of [**85**]. Image Source: `https://github.com/awarelab/continual_world/blob/main/assets/images/cw20.png`

4.6. Available Methods

One of Sequoia's biggest strength is how easy it is to extend. As mentioned earlier in Section 4, most methods in Sequoia are the result directly reusing existing implementations from other frameworks and repositories. This is the case for Avalanche [**52**], Stable-Baselines3 [**66**] and Continual World [**85**]. Table 4.2 shows all the methods currently available in Sequoia, along with their target setting.

4.6.1. The Base Method

As a proof of concept for the development of general methods, Sequoia features a highly customizable `BaseMethod`, which has the root Setting as its target setting, and is thus applicable onto any Setting in the tree, both in RL and in SL.

Method	Target setting
`BaseMethod`	`Setting` (all)
`BaseMethod`.EWC [**41**]	Incremental Learning (RL + SL)
`BaseMethod`.PackNet [**54**]	Incremental Learning (RL + SL)
replay	Incremental SL
CN-DPM [**46**]	CTaCL (all CSL)
HAT [**73**]	Task-Incremental SL
PNN [**71**]	Incremental SL
`Avalanche`.naive [**52**]	Incremental SL
`Avalanche`.AGEM [**12**]	Incremental SL
`Avalanche`.cwr_star [**52**]	Incremental SL
`Avalanche`.EWC [**41**]	Incremental SL
`Avalanche`.Gdumb [**65**]	Incremental SL
`Avalanche`.GEM [**53**]	Incremental SL
`Avalanche`.LWF [**49**]	Incremental SL
`Avalanche`.replay	Incremental SL
`Avalanche`.SI [**88**]	Incremental SL
`stable-baselines3`.A2C [**59**]	Incremental RL
`stable-baselines3`.DDPG [**50**]	Continual RL
`stable-baselines3`.DQN [**60**]	Continual RL
`stable-baselines3`.PPO [**72**]	Continual RL
`stable-baselines3`.SAC [**29**]	Continual RL
`stable-baselines3`.TD3 [**25**]	Continual RL
`continual_world`.SAC [**29**]	Incremental RL
`continual_world`.AGEM [**12**]	Incremental RL
`continual_world`.EWC [**41**]	Incremental RL
`continual_world`.VCL [**61**]	Incremental RL
`continual_world`.MAS [**1**]	Incremental RL
`continual_world`.L2 regularization	Incremental RL
`continual_world`.PackNet [**54**]	Incremental RL
`continual_world`.Replay	Incremental RL

Table 4.2. Sequoia's methods support. Each method specifies a target setting, listed on the right. Most methods are applicable in either RL or SL, while some can be applied to both. Methods also specify the "level of nonstationarity" they are prepared to handle, as a choice of one of Continual (which is also referred to as Continuous Task-Agnostic CL in Subsection 3.5.1), Discrete, Incremental, Task-Incremental, Traditional, and Multi-Task.

While developing a new Method in Sequoia, users are encouraged to separate the training logic from the networks used, the former being contained in the `Method`, and the latter in a model class, as advocated by PyTorch-Lightning [**19**] (PL), which we employ as part of this BaseMethod.

The BaseMethod is accompanied by the BaseModel, which acts as a modular and extendable model for CL Methods to use. This BaseModel adheres to PyTorch-Lightning's

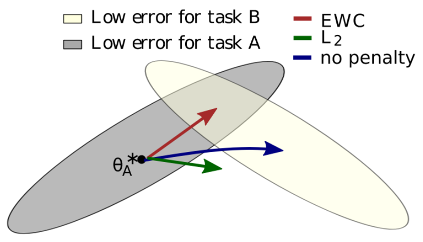


Fig. 4.6. Figure 1 from [**40**]: *"Elastic weight consolidation (EWC) ensures task A is remembered whilst training on task B. Training trajectories are illustrated in a schematic parameter space, with parameter regions leading to good performance on task A (gray) and on task B (cream). After learning the first task, the parameters are at θ_A^*. If we take gradient steps according to task B alone (blue arrow), we will minimize the loss of task B but destroy what we have learnt for task A. On the other hand, if we constrain each weight with the same coefficient (green arrow) the restriction imposed is too severe and we can only remember task A at the expense of not learning task B. EWC, conversely, finds a solution for task B without incurring a significant loss on task A (red arrow) by explicitly computing how important weights are for task A."*[**40**]

LightningModule interface, making it easy to extend and customize with additional callbacks and loggers. Likewise, the BaseMethod employs a `pl.Trainer`, which is able to train the BaseModel using the environments produced by any setting. Sequoia's Settings are also closely related to PL's DataModule abstraction.

Using this BaseModel when creating a new CL method can be particularly useful when transitioning from a CL Setting to its parent, as it comes equipped with most of the components required to handle such transitions (e.g. task inference, multi-head prediction, etc.). These components, as well as the underlying encoder, output head, loss function, etc. can easily be replaced or customized.

Additional losses can also easily be added to the `BaseModel` through a modular interface, which was originally designed to facilitate exploration of self-supervised learning research. Appendix A describes an experiment where the base method is used with combinations of different self-supervised auxiliary tasks, in order to mitigate catastrophic forgetting in different CSL settings.

4.6.1.1. EWC. Elastic Weight Consolidation (EWC)[**40**] is a regularization-based method for CL. When training on a new task, it adds a loss term that prevents weights that are judged to be important to the previous tasks from changing too much. The importance of the weights for each task is determined based on the loss gradients and the Fisher Information Matrix.

This regularization term is implemented in Sequoia as an auxiliary loss, which can be easily added to the `BaseModel`. This is the approach taken by the EWC method of Sequoia, which extends the `BaseMethod` and adds the EWC auxiliary loss to the `BaseModel`. The EWC

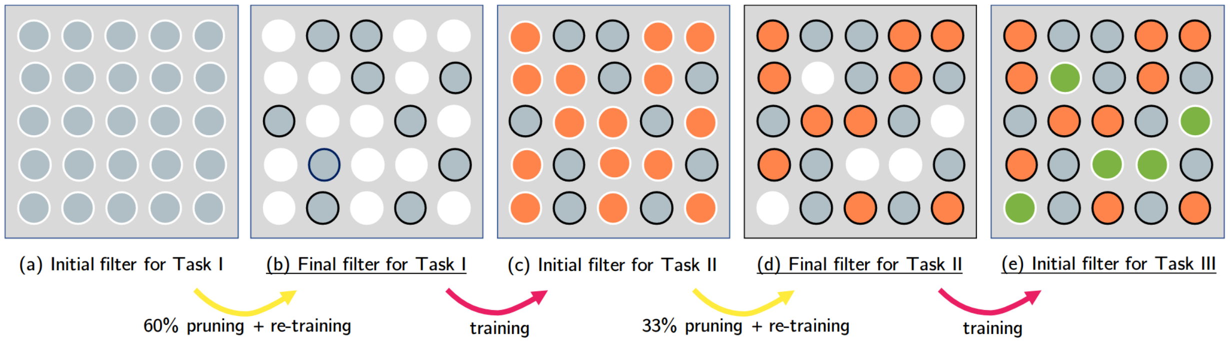

(a) Initial filter for Task I
(b) Final filter for Task I
(c) Initial filter for Task II
(d) Final filter for Task II
(e) Initial filter for Task III
60% pruning + re-training
training
33% pruning + re-training
training

Fig. 4.7. Figure 1 from [**54**], which shows how PackNet learns multiple tasks in the same network.

method requires some access to the task boundaries, and therefore cannot be applied to the CTaCL setting, where the environment changes smoothly over time and where there are no discrete task boundaries. The EWC method therefore restricts the domain of application of the BaseMethod slightly, by having the Incremental Learning setting as it's target setting.

4.6.1.2. PackNet. The PackNet method [**54**] is a parameter isolation method for CL, where multiple tasks can be learned within a single network by learning masks that prevent changing the weights of the previous tasks. After learning each task, the weights of the network are sorted based on their *importance*, as measured quite simply using their magnitude, and a pre-determined fraction of these weights, e.g. 50%, are kept, and the other weights are set to 0. The network is then retrained on the current task, using this sparse network, and the value of these weights are then frozen and perserved for future tasks, where this cycle starts over again. This process is illustrated in Figure 4.7, Figure 1 from [**54**]

The pruning mechanism of PackNet is very nicely decoupled from the actual training objective or downstream task, which makes it a very modular and widely-applicable method for CL in various domains. Following this idea, it is implemented in Sequoia as a PyTorch-Lightning Callback, that can be added to any LightningModule (including the BaseModel). Similarly as in EWC, we also make it available as a standalone method, which extends the BaseMethod and adds the PackNet callback to the BaseModel. Also simiarly to EWC, PackNet requires knowledge of the task boundaries. It therefore also has the Incremental Setting as its target setting.

4.6.2. Supervised Learning Methods

There already exists different libraries or frameworks that provide algorithms for CSL. As mentioned in Section 4, one of the main goals of Sequoia is to serve as a unified catalog of the existing methods and settings in CL, and to allow these methods to be used on different settings with as little overhead as possible.

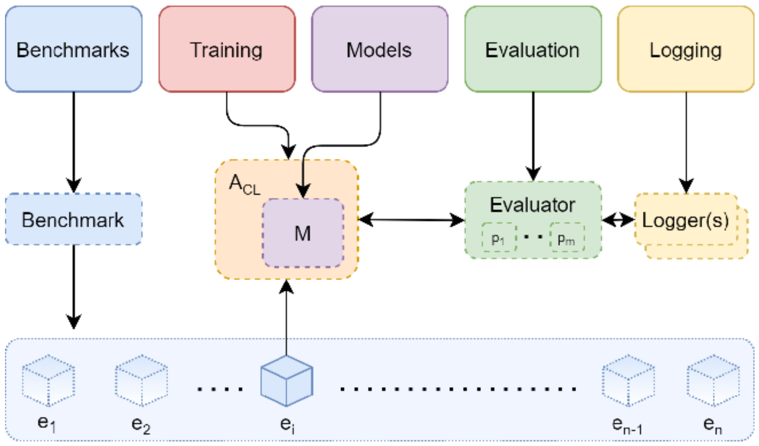


Fig. 4.8. Figure 1 from [**52**], showing the different components of Avalanche.

To that end, Sequoia benefits from recent developments in CL libraries such as Avalanche [**52**]. This subsection describes the methods applicable exclusively to the CSL settings of Sequoia.

4.6.2.1. Avalanche. Avalanche [**52**] is a feature-rich library for CL research that includes many CL algorithm implementations from the literature as well as various evaluation pipelines. Avalanche offers both standardized benchmarks as well as a growing set of CL methods, which are referred to as *strategies* in Avalanche. See Figure 4.8 for an illustration of the main components of Avalanche. There are interesting comparisons to be made between the approach taken by Sequoia and Avalanche. A more detailed description of the similarities and differences between them is included in Appendix C.

Many of the strategies of Avalanche [**52**] are available as Methods in Sequoia, and mostly target the Incremental CL setting. See Table 4.1 for a complete list the Avalanche methods in Sequoia.

4.6.2.2. CN-DPM. The Continual Neural Dirichlet Process introduced in [**46**] is a very general method for task-free CL. It learns a model consisting of a mixture of experts, that are spawned over the course of training. When the model encounters samples with a high uncertainty (or high training loss), these samples are stored in a buffer, called the *sort-term memory* buffer. Once filled, a new expert is spawned and trained on the contents of the short-term memory buffer. Experts are able to share parameters without interfering with each other using gradient-gated connections between their modules. Importantly, this method does not necessitate any task boundary information, and creates experts autonomously over time. This method is available as an optional add-on to Sequoia, and can be applied to any CSL setting.

4.6.3. Reinforcement Learning Methods

Libraries for RL can easily be used to create new methods, as we opted to uses the gym environment as the general abstraction for a learning problem. Similarly as in supervised learning, the goal here is to reuse existing libraries as much as possible, and make their algorithms and models available as methods that target the appropriate RL setting from our hierarchy.

4.6.3.1. Stable-Baselines3. The Stable-Baselines3 (SB3) library contains different RL algorithm implementations. They are not specifically designed for CRL problems, but can still be used as a foundation or backbone to create CRL methods.

4.6.3.2. Continual World. As mentioned previously, the Continual-World [**85**] package defines two benchmarks based on Meta-World [**87**]. It also includes different CRL methods based on SAC. This library required quite a bit of work to be usable in Sequoia, as the methods were all directly incorporated into the same model. These methods were disentangled, and each of them now inherits from an SAC base method. They are made available in Sequoia through a fork of the original work of [**85**], which is available at `https://www.github.com/lebrice/continual_world`.

4.7. Experiments

Sequoia's design makes it easy to conduct large-scale experiments to compare the performance of different methods on a given setting, or to evaluate the performance of a given method across multiple settings and datasets. We illustrate this by performing large-scale empirical studies involving all the settings and methods available in Sequoia, both in CRL and CSL. Each study involves up to 20 hyper-parameter configurations for each combination of setting, method, and dataset, in both RL and SL, for a combined total of ≈ 8000 individual completed runs. This section provides an overview of these experiments, which are also publicly available at `https://wandb.ai/sequoia/`.[3]

Sequoia also makes it possible to use methods originally designed for a general setting, and evaluate their performance when applied to a different type of sub-problem. For instance, representation learning methods are among the most general types of methods imaginable, as they only assume that some type of data (e.g. images) be present in order to be usable.

In Appendix A, we use this fact to study the performance of self-supervised learning methods in a continual learning setting.[4] We examine the effects of different combinations of self-supervised, supervised, and continual learning losses on the amount of forgetting incurred by a neural network in a task-incremental setting.

[3]We will update these sample studies periodically to reflect all future improvements made to the framework.
[4]This experiment was excluded from the main portion of this work for the sake of brevity.

4.7.1. Continual Supervised Learning

As part of our CSL study, we use some of the "standard" image classification datasets such as MNIST [**45**], Cifar10, and Cifar100 [**42**]. Furthermore, we also include the Synbols [**43**] dataset, a character dataset composed of two independent labels: the characters and the fonts. (See Appendix D.1.1 for the motivation for this dataset). Exhaustive results can be found online on the Weights and Biases platform at `https://wandb.ai/sequoia/csl_study`.

A sample of these results is illustrated in Figure 4.9, which shows results of various methods in the class-IL and task-IL settings in terms of their final performance and runtime. We note that some `Avalanche` methods achieve lower than chance accuracy in task-IL because they do not use the task label to mask out the classes that lie outside the tested task.

4.7.2. Continual Reinforcement Learning

We apply the RL methods from SB3 on multiple benchmarks built on HalfCheetah-v2, Hopper-v2, MountainCar-v0, CartPole-v0, MetaWorld-v2. We also introduce a new discrete domain benchmark, namely Continual-MonsterKong, that we developed to study forward transfer in a more meaningful way (see Appendix D.1 for a detailed discussion of this benchmark). Complete results are available at `https://wandb.ai/sequoia/crl_study`.

A sample of these results is in Figure 4.10. It presents various methods in the traditional and incremental learning settings with their final performance, online performance and normalized runtime.

4.7.3. HalfCheetah-gravity and Hopper-Bodyparts

In Table 4.3, we apply the `continual-world` methods, built on top of SAC, on a incremental RL benchmark inspired by [**58**]. Finally, Figure 4.11 shows the transfer matrix achieved by one such algorithm, namely PPO [**72**].

HalfCheetah-gravity and Hopper-Bodyparts are two benchmarks introduced in [**58**]. In the first, each task consist of a different gravity. In the latter, the agent's body parts are changing in size at each task. The gravity and body parts values are sampled as in [**58**]. The two benchmarks we study are each composed of 10 tasks. Figure 4.10 shows the results of this study.

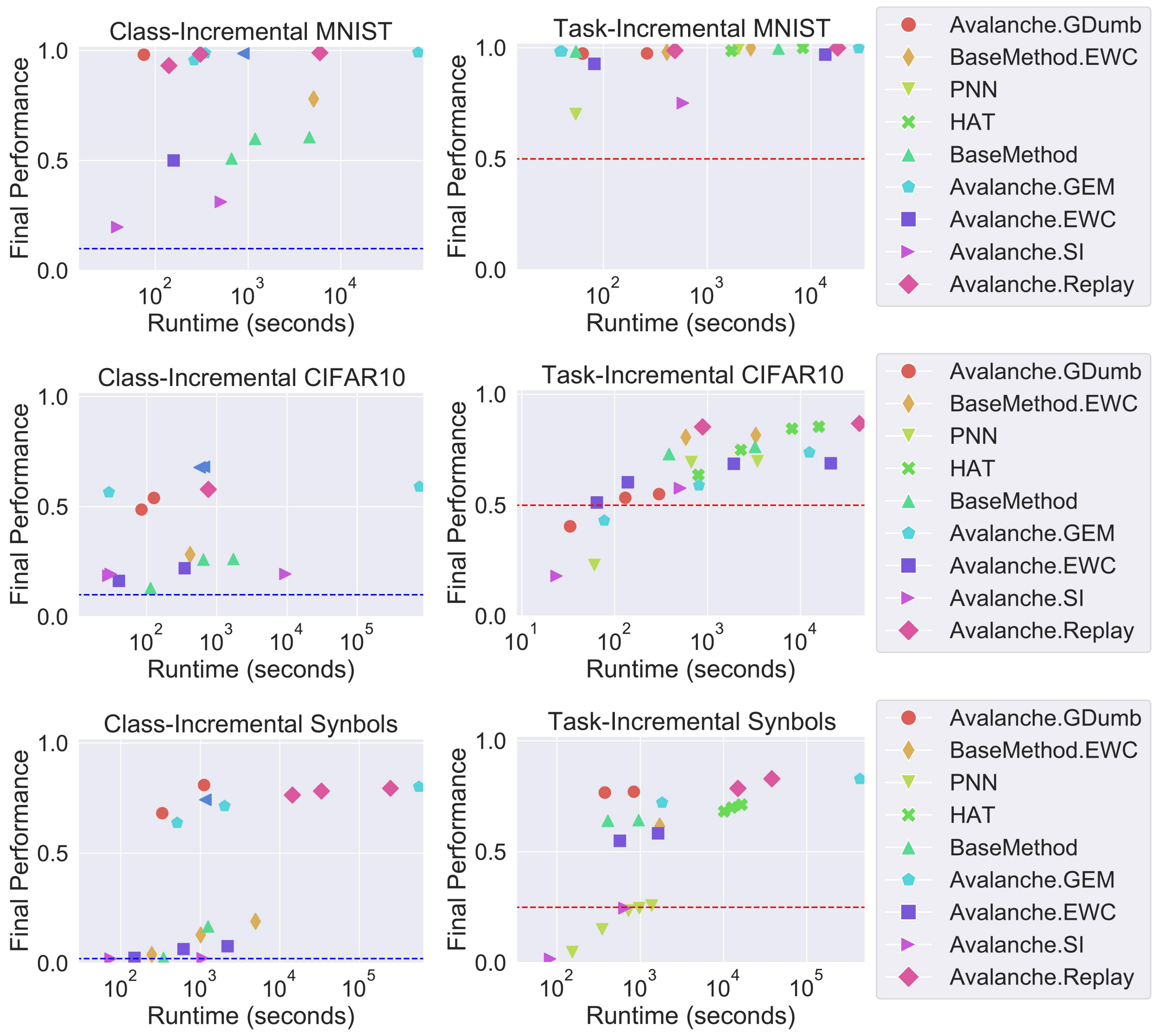


Fig. 4.9. Incremental Supervised Learning results. Final performance (vertical axis) is plotted against runtime (horizontal axis). The methods achieving the best trade-off lie closer to the top-left of the figures. The dotted line shows chance accuracy for each setting-dataset combination. For each methods, several trials are presented depending on metrics composed of linear combination of final performance and (normalized) runtime. Intuitively, better performance in CL normally comes at the cost of increased computation. This intuition is reflected in the presented results, as highlighted by the observed correlation between final performance and runtime. GEM and GDumb achieve the best tradoff, although the latter cannot make predictions in an online manner and thus serves more as a reference point.

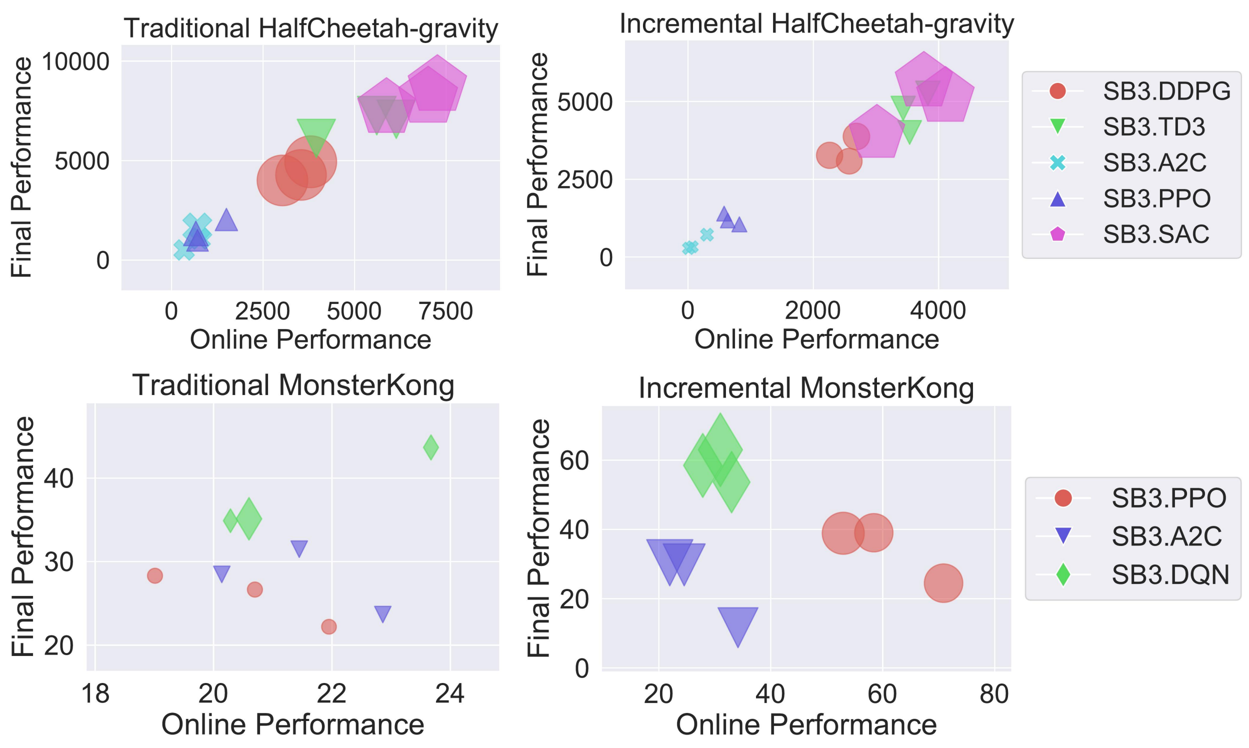
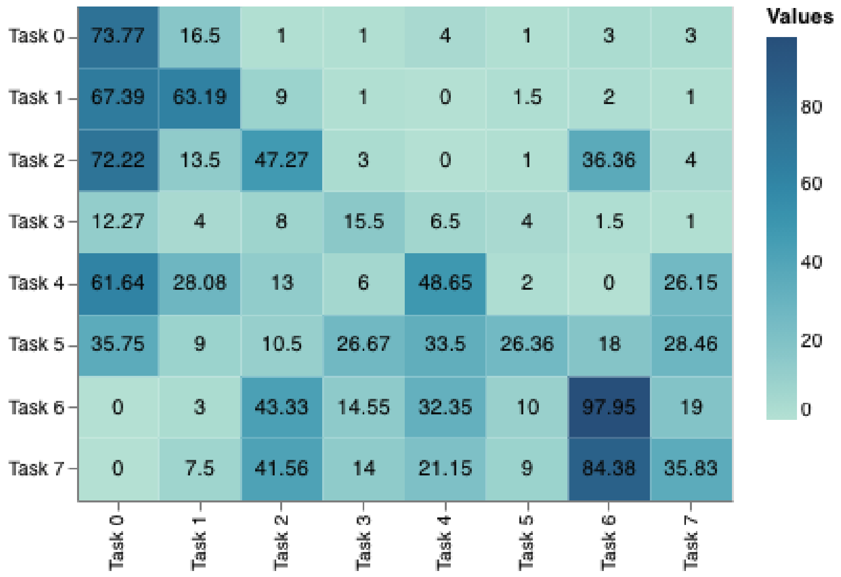


Fig. 4.10. Impact of the RL backbone algorithm in Traditional and Incremental RL. Final performance (vertical axis) is plotted against online performance (horizontal axis). The bubbles' size indicates the normalized runtime of the methods. Datasets are presented in each row and settings are presented in each column. For each method, several trials are presented depending on metrics composed of linear combination of final performance and online performance. The methods achieving the best trade-off lie closer to the top-right of the figures and have smaller bubble size. In general, we observe a trade-off between performance and runtime, a tendency also observed in Figure 4.9. Another interesting trade-off can be observed between final performance and online performance. E.g., in both MonsterKong benchmarks, DQN achieves the best final performance whereas PPO achieves the best online performance. Because the former is off-policy, it can re-use the previously acquired data to retain its performance on past tasks, increasing final performance. Contrarily, the latter, being on-policy, focuses on the current task and thus learns it faster, thereby increasing its online performance.

Method	Final Perf.	Online Perf.	Runtime (h)
SAC (base)	194 ± 105	254 ± 9	19.0±0.3
AGEM	787 ± 268	283 ± 14	24.9±2.6
EWC	616 ± 257	232 ± 17	19.3±2.5
L2	840 ± 224	245 ± 22	18.7±2.5
MAS	607 ± 236	236 ± 13	20.3±1.7
PackNet	1153 ± 325	**290 ± 41**	25.2±6.3
Perfect Memory	**1500 ± 399**	255 ± 7	**18.2±2.0**

Table 4.3. Incremental RL results. Multiple CRL methods, all built on top of SAC, are tested on the Hopper-Bodyparts benchmarks. All CRL methods outperformed the Fine-tuning SAC baseline, validating their efficacy. Experience replay with a Perfect Memory achieves the best retained performance on all tasks, followed closely by PackNet.



	Task 0	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7
Task 0	73.77	16.5	1	1	4	1	3	3
Task 1	67.39	63.19	9	1	0	1.5	2	1
Task 2	72.22	13.5	47.27	3	0	1	36.36	4
Task 3	12.27	4	8	15.5	6.5	4	1.5	1
Task 4	61.64	28.08	13	6	48.65	2	0	26.15
Task 5	35.75	9	10.5	26.67	33.5	26.36	18	28.46
Task 6	0	3	43.33	14.55	32.35	10	97.95	19
Task 7	0	7.5	41.56	14	21.15	9	84.38	35.83

Fig. 4.11. PPO's Transfer matrix in Continual-MonsterKong. Each cell at row *i* and column *j* indicates the test performance on task *j* after having learned tasks 0 through *i*. The contents of each cell correspond to the average reward per episode obtained in the test environment for the corresponding task. Positive numbers above the diagonal indicate generalization to unseen tasks, which is achievable by design in the Continual-Monsterkong benchmark.

Summary

In this chapter, a concrete implementation of problem hierarchies for the field of CL was introduced. Related works, in the form of software libraries and frameworks were described, and we showed how they contributed to Sequoia's design. The Setting, Method, and Environment abstractions of Sequoia were described and illustrated, followed by a list of their concrete implementations currently available in Sequoia.

To demonstrate Sequoia's effectiveness, a set of experiments were presented, where we perform large-scale comparisons of the performance of methods on settings in both SL and RL. In Appendix A we also describe an experiment where general self-supervised learning methods are applied to a continual learning setting, resulting in reduced forgetting in the model's representations.

The considerations of Section 3.3 were directly informed by the challenges faced while designing and implementing Sequoia. Next, in Chapter 5, some of the main limitations of Sequoia are described, along with a discussion of the plans for future work in overcoming these limitations.

Chapter 5

Future Work

Sequoia, as it stands, is a useful proof-of-concept for the implementation of a problem hierarchy for CL. This chapter provides a discussion of some of the issues and limitations of Sequoia's design and lays down potential avenues for future work.

The biggest limitation of Sequoia in its current form is its dependence on *nominal* subtyping, where settings explicitly list and inherit from all the assumptions they respect, resulting in a deep, fully-qualified inheritance hierarchies of settings. While effective at reducing code duplication, such pervasive use of inheritance often has the side-effect of reducing cohesion, which can be detrimental to the user and developer experience. This issue is exemplified in Listing 18.

Different potential solutions are described below. The first is composition, where the components of the hierarchy are broken down and reorganized into many smaller hierarchies for each component. Second, structural subtyping is briefly introduced, and discussed as an alternative. Finally, we introduce a novel form of structural subtying based on a novel use of Space abstractions, such as those introduced in Gym, which we refer to as *spatial* subtyping.

5.1. Composition

Another solution to this problem is to use a true composition of assumptions. This is part of the approach taken in Sequoia, where the central "trunk" for CL assumptions is implemented independently of whether the concrete setting is a supervised or reinforcement learning setting. This is achieved by maintaining a loose coupling between these settings and the environment they operate with: from the perspective of the CL setting, it doesn't matter if the environment is a supervised or reinforcement learning environment, the training and evaluation loop remains the same.

In practice however, settings are defined concretely as classes that inherit from both a CL setting of the central branch as well as either RL or SL. This was necessary for a variety of reasons, one example of which is the need of settings to declare which environments are

```python
from sequoia.settings.sl import *
from sequoia.settings.rl import *
from sequoia.methods import BaseMethod

method = BaseMethod(learning_rate=1e-3)

for setting in [
    ContinuousTaskAgnosticSLSetting("mnist"),
    ContinuousTaskAgnosticRLSetting("cartpole"),
    DiscreteTaskAgnosticSLSetting("mnist"),
    DiscreteTaskAgnosticRLSetting("cartpole"),
    IncrementalSLSetting("mnist"),
    IncrementalRLSetting("cartpole"),
    TaskIncrementalSLSetting("mnist"),
    TaskIncrementalRLSetting("cartpole"),
    MultiTaskSLSetting("mnist"),
    MultiTaskRLSetting("cartpole"),
    TraditionalSLSetting("mnist"),
    TraditionalRLSetting("cartpole"),
]:
    results = setting.apply(method)
    results.summary()
    results.make_plots()
```

Listing 18. The combinatorial explosion problem: As more assumptions families are added to Sequoia, having fully-qualified inheritance hierarchies of settings becomes untenable.

available, combined with the fact that some environments do not support continuous tasks, and thus only become usable after a given level has been reached when moving down the hierarchy.

One way to create a compositional implementation of problem hierarchies would be to use a fully-qualified specification of the components common to all learning problems, as done on the theoretical front by [**26**] (described previously in Subsection 3.4.4). In this solution, the deep hierarchy of settings would be replaced with multiple smaller hierarchies for the different components of the learning problem. However, existing software libraries or benchmarks, which might have a very simple structure, would have to be reorganized and re-framed in terms of this potentially complex new abstraction for a learning problem.

5.2. Structural subtyping

Structural subtyping is another approach to solving this problem. In structural subtyping, a type A is considered a subtype of another type B if the structure, properties and methods of B are implemented in A. A crucial difference with nominal subtyping is that A does not need to declare that it inherits from B, or even be aware of B's existence. Using structural subtyping, the properties and methods common to different types can be described

```
from itertools import product
import sequoia
from sequoia.methods import BaseMethod
from sequoia.assumptions import combine
from sequoia.assumptions.stationarity import NonStationary, Stationary
from sequoia.assumptions.task_type import Continuous, Discrete
from sequoia.assumptions.task_awareness import TaskAware, TaskAgnostic

stationarity = [Stationary, NonStationary]
task_type = [Continuous, Discrete]
task_awareness = [TaskAware, TaskAgnostic]
environment = ["mnist", "cartpole"]

for environment in environments:
    for assumptions in product(stationarity,
                               task_type,
                               task_awareness):
        setting = Setting(environment, assumptions=assumptions)
        method = BaseMethod()
        results = setting.apply(method)
        print(results.summary())
```

Listing 19. Pseudocode for a compositional solution to the class explosion problem. Assumptions are independent components that are combined with a concrete environment, resulting in a setting. Note, the `combine` function alluded to here does not correspond to multiple inheritance.

using an interface, without the need to change any of their structure. Listing 20 illustrates the differences between nominal and structural subtyping.

Protocols have been recently added to the Python language as a mechanism for structural subtyping. Protocols make it possible to statically check that a class or module respects a set of type constraints about its attributes or methods. This is in contrast with abstract base classes and nominal subtyping, where base classes have to be explicitly inherited from. There is also limited support for checking protocols at runtime: protocols, when used at runtime, only verify that the attributes of the protocol are present on the given object, without checking their types. This limitation greatly reduces the usefulness of protocols as a means of enforcing structural subtyping at runtime.

In the case of our problem hierarchies, where we might want to represent, for instance, different assumptions related to the type of non-stationarity present in an environment, it becomes very unclear how this might be described using a structural subtyping mechanism. This is one of the advantages of a nominal subtyping system: they are able to declare in advance properties that would be very expensive or difficult to otherwise verify. Inheritance is not the only means of declaring such properties in advance, however. For example, if there were to be a simple `assumptions` attribute on the environment class, where assumptions

```python
from typing import TypeVar, Generic, Iterator
from abc import ABC, abstractmethod
T = TypeVar("T")
```

```python
# Nominal subtyping:
from abc import ABC, abstractmethod

class Iterable(ABC, Generic[T]):
    @abstractmethod
    def __iter__(self) -> Iterator[T]:
        ...

class Bucket(Iterable[int]):
    ...
    def __len__(self) -> int:
        ...
    def __iter__(self) -> Iterator[int]:
        ...

def collect(items: Iterable[int]) -> None:
    ...

collect(Bucket())  # Passes type check
collect("bob")     # fails type check
collect([1, 2, 3]) # Also fails type check!
```

```python
# Structural subtyping:
from typing import Protocol

class Iterable(Protocol[T]):
    def __iter__(self) -> Iterator[T]:
        ...


class Bucket:
    ...
    def __len__(self) -> int:
        ...
    def __iter__(self) -> Iterator[int]:
        ...

def collect(items: Iterable[int]) -> None:
    ...

collect(Bucket())  # Passes type check
collect("bob")     # Fails type check
collect([1, 2, 3]) # Passes type check!
```

Listing 20. Comparison of Nominal and Structural subtyping in Python. Note that Python does not enforce these type constraints at runtime: they are verified by third-party type-checkers. In this example, the `Bucket` class needs to explicitly inherit from the Iterable class to be an appropriate argument to the `collect` function in nominal subtyping. In the structural subtyping alternative, the Bucket class does not need to explicitly inherit from Iterable: the requirements are met, and Bucket is therefore considered a subtype of the Iterable protocol.

were listed as flags, enumeration types, or otherwise, then checking for the presence of an assumption would be as simple as checking if the assumption is in the list. This shares the same limitation as nominal subtyping, namely that the assumptions need to be explicitly added in advance, to all environments (or learning problem components more generally), although it would not in this case be necessary to modify the implementation of these environments.

5.3. "*Spatial*" subtyping

So far, assumptions have either been represented as classes that are subclassed by settings or environments (nominal subtyping), or as modular components of the learning problem (composition). Structural subtyping is an interesting alternative, but it is yet unclear how assumptions can be implemented in that paradigm.

Classes were also always used as the intermediary model to describe problem hierarchies. However, settings are, by definition, sets of learning problems, or regions in the space of learning problems. In this section, we introduce and explore the very simple idea of implementing settings directly as spaces of learning problems, and assumptions directly as predicate functions that are used to constrain a space.

In this scenario, instead of having settings explicitly list their assumptions, the assumptions can instead be be implemented as predicate functions that are applied directly to a setting, and return whether they are respected or not. This is directly aligned with the definition of an Assumption from Subsection 3.1.1.

5.3.1. The Space / Class equivalence

An interesting fact to note is that spaces are conceptually equivalent to classes, and can serve the same role in practice. Samples from a space are equivalent to instances of a class. Sampling from a space is equivalent to creating an instance of a class. Checking for membership in a space (e.g. through `space.contains(v)`) is equivalent to checking if an object is an instance of a given type (e.g. using `isinstance(v, V)`). Multiple inheritance can be replaced with taking the intersection between multiple spaces. A space class, e.g. `gym.spaces.Discrete` can therefore be seen as a meta-class (a type of types) that, when invoked e.g. in `Discrete(10)`, gives back a space instance, which is equivalent to a class. This is also in-line with the classical definition of classes as sets of all possible members, in this case the digits from 0 to 9.

By pushing this idea one step further, we arrive at a very interesting realization. Problem hierarchies describe a space of learning problems. Sequoia uses classes and polymorphism to model the relationship between settings. However, why is there a need for this intermediate representation? **Why not model the Space of learning problems using spaces directly?**

5.3.2. Higher-order spaces: Environments, Datasets

Spaces can be created for more than just arrays, integers, dicts, and tuples. For instance, one can very easily create spaces of RL environments, or spaces of SL datasets. We refer to these new kinds of spaces as *higher-order* spaces. An example of such a space is shown in Listing 21. Higher-order spaces open the door to a more structural way of describing interfaces, including the one between learning problems and learning algorithms.

For example, an RL algorithm that is only applicable to environments where actions are discrete, could do so using a space, as shown in Listing 21. These spaces provide the same functionality as the classes in Sequoia: any given environment or dataset can be checked for membership by verifying if the assumptions hold in that environment.

```python
from gym import Env, Space
# Envs is a space of environments:
Envs: Space[Env]

all_envs: Space[Env] = Envs()

# Assumption:
def actions_are_discrete(env: Env) -> bool:
    return isinstance(env.action_space, Discrete)

# The subspace where the assumption holds:
discrete_action_envs: Space[Env[Any, int]] = all_envs.where(actions_are_discrete)

gym.make("CartPole-v1") in discrete_action_envs  # True
gym.make("Pendulum-v1") in discrete_action_envs  # False

discrete_action_envs.sample() # Sample a compatible environment, e.g:
# <TimeLimit<OrderEnforcing<MountainCarEnv<MountainCar-v0>>>>
```

Listing 21. A Space of Environments. In this example, an assumption is used to restrict the space to only the environments where the action space is discrete. Sampling from the space produces an environment with discrete actions. These kinds of spaces could potentially be used as a replacement for the nominal subtyping mechanism used to specify the target setting of methods in Sequoia.

Currently, new method classes specify a *target setting* (i.e. a setting class Setting) as an attribute. By default, we determine if a method is applicable in a given setting by checking whether that setting is a subtype of the method's target setting, using *isinstance* or *issubclass*. Methods can customize this behaviour by overriding the `is_applicable` class method. Spaces could potentially simplify this.

It is however still unclear how more complex assumptions would be expressed using these semantics. For instance, how would the assumption that the environment dynamics are stationary, or that the state is not affected by the actions of the agent be expressed? One way to do this might be to devise a lightweight procedure to check these kinds of properties functionally at runtime. For example, if we wanted to encode the assumption that an environment is *passive* (i.e. actions don't influence the state), then this could be implemented as a test where different actions are executed in the same starting state in the environment, and where we return whether the next state varies based on the action taken.

There could be some concern about the correctness or reliability of these checks. To use the same example, if the state was to be always the same for all actions, but only when in that initial state, e.g. an agent that is stuck in a corner of a room or fell down a hole, then actions might appear to have no influence on future state, while that would be incorrect in general. This approach would require that these assumptions be very lightweight, otherwise it might become computationally intensive to perform the simple check of whether a method

```
Settings: Space[Setting]

all_settings: Space[Setting] = Settings()

def only_one_task(setting: Setting) -> bool:
    # assuming that all settings have a list of environments, for instance.
    return len(setting.train_envs) == 1

single_task_settings = all_settings.where(only_one_task)
active_settings = all_settings.where(
    # Still unclear how to encode these richer assumptions:
    environment=lambda env: env.actions.have_an_effect_on(env.state)
)

single_task_active_settings = single_task_settings & active_settings
setting = single_task_active_settings.sample()
```

Listing 22. Pseudocode for a potential implementation for the space of settings.

is compatible with an environment. There would then necessarily be a trade-off to consider between the robustness of these evaluations and their efficiency.

Another important unanswered question is how to represent settings using this new use of spaces. One possibility might be to apply the same reasoning as was done for environments, and simply construct a space of settings, where assumptions can be applied to the components of the setting. For instance, if we assume that all settings have one or more environment(s), then some assumptions could be directed towards the environments, as shown in Listing 22. However, it's unclear how assumptions about the setting itself or its training/evaluation procedure should be represented.

In summary, the combinatorial explosion problem can be addressed in different ways. The first and most principled is to use composition rather than inheritance, and to break down the hierarchy of settings into smaller hierarchies of the setting's components. Structural subtyping is another alternative, where assumptions could be verified directly against a setting, rather than settings declaring their assumptions in advance. Lastly, we describe what we refer to as *spatial* subtyping, where higher-order spaces can be created to directly describe the space of environments or learning problems. How to effectively and succinctly express complex assumptions is an open question in both the structural and spatial subtyping approaches.

Conclusion

In this work, we propose to use type hierarchies from computer science and object-oriented programming to structure the types of problems in machine learning. We argue that putting these principles into practice constitutes an effective solution to some of the issues experienced by ML researchers today.

Problem hierarchies describe and formalize the process of creating taxonomies of the types of problems studied in any given field of research. These taxonomies and their visual representations as DAGs could be used as maps, to give both new and experienced researchers a lay-of-the-land, and help identify areas ripe for standardization.

Implementations of problem hierarchies also facilitate the reuse of prior work by establishing a clear separation between research problems and their solutions. This leads to the development of learning algorithms that are easier to disentangle from the problems they attempt to solve, and thus easier to extract and reuse in a different context or field. Furthermore, this hierarchical representation allows methods to declare their region of applicability, making them reusable polymorphically across different types of problems.

Sequoia is introduced as a proof-of-concept implementation of a problem hierarchy for the field of continual learning. Through it's reuse of existing frameworks and libraries in continual, supervised, and reinforcement learning, Sequoia acts as a repository of the benchmarks and algorithms offered by the existing libraries within this software ecosystem.

Finally, the assumptions that we make in ML research are often very different than the conditions present in the real world. Applied researchers and professionals tasked with solving real-world problems using ML greatly benefit from having good access to the solutions introduced by the research community, so they can be adapted and reused for their particular needs. Similarly, researchers also benefit from being informed of the types of problems that remain unsolved in practice. Problem hierarchies grants ML practitioners the ability to identify the research setting most closely related to the problem at hand, and to get access not only to the solutions for that specific type of problem, but to **all** research solutions applicable to more general problems. We believe that problem hierarchies and their implementations therefore open the way for more collaboration and sharing of solutions between researchers and ML practitioners.

References

[1] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget, 2018.

[2] Rahaf Aljundi, Klaas Kelchtermans, and Tinne Tuytelaars. Task-free continual learning. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11246–11255, 2019.

[3] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. Gradient based sample selection for online continual learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[4] Haitham Bou Ammar, Eric Eaton, Paul Ruvolo, and Matthew Taylor. Online multi-task learning for policy gradient methods. In *International conference on machine learning*, pages 1206–1214, 2014.

[5] André Barreto, Diana Borsa, John Quan, Tom Schaul, David Silver, Matteo Hessel, Daniel Mankowitz, Augustin Žídek, and Remi Munos. Transfer in deep reinforcement learning using successor features and generalised policy improvement. *arXiv preprint arXiv:1901.10964*, 2019.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] Massimo Caccia, Pau Rodriguez, Oleksiy Ostapenko, Fabrice Normandin, Min Lin, Lucas Page-Caccia, Issam Hadj Laradji, Irina Rish, Alexandre Lacoste, David Vázquez, et al. Online fast adaptation and knowledge accumulation (osaka): a new approach to continual learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[9] Daniele Calandriello, Alessandro Lazaric, and Marcello Restelli. Sparse multi-task reinforcement learning. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in neural information processing systems 27*, pages 819–827. Curran Associates, Inc., 2014.

[10] Yash Chandak, Georgios Theocharous, Chris Nota, and Philip S. Thomas. Lifelong learning with a changing action set, 2020.

[11] Yash Chandak, Georgios Theocharous, Shiv Shankar, Sridhar Mahadevan, Martha White, and Philip S Thomas. Optimizing for the future in non-stationary mdps. *arXiv preprint arXiv:2005.08158*, 2020.

[12] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. Efficient lifelong learning with A-GEM. In *International Conference of Learning Representations (ICLR)*, 2019.

[13] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv preprint arXiv:2002.05709*, 2020.

[14] Samuel Choi, Dit-Yan Yeung, and Nevin Zhang. Hidden-mode markov decision processes. 12 1999.

[15] Samuel PM Choi, Dit-Yan Yeung, and Nevin L Zhang. Hidden-mode markov decision processes for nonstationary sequential decision making. In *Sequence Learning*, pages 264–287. Springer, 2000.

[16] Petros Christodoulou. Soft actor-critic for discrete action settings, 2019.

[17] Arthur Douillard and Timothée Lesort. Continuum: Simple management of complex continual learning scenarios, 2021.

[18] William Falcon et al. Pytorch lightning. In *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* [**19**].

[19] William Falcon et al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

[20] Jesse Farebrother, Marlos C Machado, and Michael Bowling. Generalization and regularization in dqn. *arXiv preprint arXiv:1810.00123*, 2018.

[21] Sebastian Farquhar and Yarin Gal. Towards robust evaluations of continual learning. *arXiv preprint arXiv:1805.09733*, 2018.

[22] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[23] Robert M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128 – 135, 1999.

[24] Robert M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135, 1999.

[25] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[26] Ali Geisa, Ronak Mehta, Hayden S. Helm, Jayanta Dey, Eric Eaton, Jeffery Dick, Carey E. Priebe, and Joshua T. Vogelstein. Towards a theory of out-of-distribution learning, 2022.

[27] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. Unsupervised representation learning by predicting image rotations. *ArXiv*, abs/1803.07728, 2018.

[28] Nathan Grinsztajn, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist. There is no turning back: A self-supervised approach for reversibility-aware reinforcement learning. 2021.

[29] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

[30] Emmanuel Hadoux, Aurélie Beynier, and Paul Weng. Solving hidden-semi-markov-mode markov decision problems. In *SUM*, 2014.

[31] James Harrison, Apoorva Sharma, Chelsea Finn, and Marco Pavone. Continuous meta-learning without tasks. *ArXiv*, abs/1912.08866, 2019.

[32] Xu He, Jakub Sygnowski, Alexandre Galashov, Andrei A. Rusu, Yee Whye Teh, and Razvan Pascanu. Task agnostic continual learning via meta learning. *ArXiv*, abs/1906.05201, 2019.

[33] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019.

[34] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[35] Christos Kaplanis, Claudia Clopath, and Murray Shanahan. Continual reinforcement learning with multi-timescale replay. *arXiv preprint arXiv:2004.07530*, 2020.

[36] Khimya Khetarpal, Zafarali Ahmed, Andre Cianflone, Riashat Islam, and Joelle Pineau. Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms. 2018.

[37] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. *CoRR*, abs/2012.13490, 2020.

[38] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490*, 2020.

[39] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[40] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. In *Proceedings of the national academy of sciences* [**41**], pages 3521–3526.

[41] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[42] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[43] Alexandre Lacoste, Pau Rodríguez López, Frederic Branchaud-Charron, Parmida Atighehchian, Massimo Caccia, Issam Hadj Laradji, Alexandre Drouin, Matthew Craddock, Laurent Charlin, and David Vázquez. Synbols: Probing learning algorithms with synthetic datasets. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 134–146. Curran Associates, Inc., 2020.

[44] Nicholas C. Landolfi, Garrett Thomas, and Tengyu Ma. A model-based approach for sample-efficient multi-task reinforcement learning, 2019.

[45] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[46] Soochan Lee, Junsoo Ha, Dongsu Zhang, and Gunhee Kim. A neural dirichlet process mixture model for task-free continual learning, 2020.

[47] Timothée Lesort, Hugo Caselles-Dupré, Michael Garcia-Ortiz, Jean-François Goudou, and David Filliat. Generative Models from the perspective of Continual Learning. In *International Joint Conference on Neural Networks (IJCNN)*, 2019.

[48] Timothée Lesort, Alexander Gepperth, Andrei Stoian, and David Filliat. Marginal replay vs conditional replay for continual learning. In *International Conference on Artificial Neural Networks*, pages 466–480. Springer, 2019.

[49] HongLin Li, Payam Barnaghi, Shirin Enshaeifar, and Frieder Ganz. Continual learning using bayesian neural networks, 2019.

[50] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[51] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, nov 1994.

[52] Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Gabriele Graffieti, Tyler L. Hayes, Matthias De Lange, Marc Masana, Jary Pomponi, Gido van de Ven, Martin Mundt, Qi She, Keiland Cooper, Jeremy Forest, Eden Belouadah, Simone Calderara, German I. Parisi, Fabio Cuzzolin,

Andreas Tolias, Simone Scardapane, Luca Antiga, Subutai Amhad, Adrian Popescu, Christopher Kanan, Joost van de Weijer, Tinne Tuytelaars, Davide Bacciu, and Davide Maltoni. Avalanche: an end-to-end library for continual learning, 2021.

[53] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[54] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning, 2018.

[55] Andreas Maurer, Massimiliano Pontil, and Bernardino Romera-Paredes. The benefit of multitask representation learning. *J. Mach. Learn. Res.*, 17(1):2853–2884, January 2016.

[56] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109 – 165. Academic Press, 1989.

[57] Jorge A. Mendez, Boyu Wang, and Eric Eaton. Lifelong policy gradient learning of factored policies for faster training without forgetting, 2020.

[58] Jorge A. Mendez, Boyu Wang, and Eric Eaton. Lifelong policy gradient learning of factored policies for faster training without forgetting, 2020.

[59] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[61] Cuong V. Nguyen, Yingzhen Li, Thang D. Bui, and Richard E. Turner. Variational continual learning. In *International Conference on Learning Representations (ICLR)*, 2018.

[62] Emilio Parisotto, Jimmy Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. In *ICLR*, 2016.

[63] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[64] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017.

[65] Ameya Prabhu, Philip HS Torr, and Puneet K Dokania. Gdumb: A simple approach that questions our progress in continual learning. 2020.

[66] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[67] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Computer Vision and Pattern Recognition (CVPR)*, 2017.

[68] Matthew Riemer, Ignacio Cases, Robert Ajemian, Miao Liu, Irina Rish, Yuhai Tu, and Gerald Tesauro. Learning to learn without forgetting by maximizing transfer and minimizing interference. *arXiv preprint arXiv:1810.11910*, 2018.

[69] Mark B Ring. Child: A first step towards continual learning. *Machine Learning*, 28(1):77–104, 1997.

[70] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In *Advances in Neural Information Processing Systems*, 2019.

[71] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive Neural Networks. *ArXiv e-prints*, 2016.

[72] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[73] Joan Serrà, Dídac Surís, Marius Miron, and Alexandros Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. *CoRR*, abs/1801.01423, 2018.

[74] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[75] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[76] Norman Tasfi. Pygame learning environment. `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016.

[77] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

[78] Sebastian Thrun and Tom M Mitchell. Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2):25–46, 1995.

[79] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In *Advances in neural information processing systems*, pages 385–392, 1995.

[80] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[81] René Traoré, Hugo Caselles-Dupré, Timothée Lesort, Te Sun, Guanghang Cai, Natalia Díaz Rodríguez, and David Filliat. Discorl: Continual reinforcement learning via policy distillation. *CoRR*, abs/1907.05855, 2019.

[82] Gido M van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

[83] Tom Veniat, Ludovic Denoyer, and Marc'Aurelio Ranzato. Efficient continual learning with modular networks and task-driven priors, 2020.

[84] Wikipedia contributors. Markov decision process — Wikipedia, the free encyclopedia, 2021. [Online; accessed 15-November-2021].

[85] Maciej Wolczyk, Michal Zajac, Razvan Pascanu, Lukasz Kucinski, and Piotr Milos. Continual world: A robotic benchmark for continual reinforcement learning. *CoRR*, abs/2105.10919, 2021.

[86] Annie Xie, James Harrison, and Chelsea Finn. Deep reinforcement learning amidst lifelong non-stationarity. *arXiv preprint arXiv:2006.10701*, 2020.

[87] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. *CoRR*, abs/1910.10897, 2019.

[88] Friedeman Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning (ICML)*, 2017.

[89] Chen Zeno, Itay Golan, Elad Hoffer, and Daniel Soudry. Task agnostic continual learning using online variational bayes, 2019.

Appendix A

Self-Supervised Continual Learning Experiments

Note to reader: This experiment was conducted and presented as a course project in the IFT-6760B - Continual Learning course at Université de Montréal during the winter of 2020. The other contributors to this project are Rey Reza Wiyatno and Jérôme Parent-Lévesque.

In this experiment, we apply different self-supervised representation learning methods to a continual learning settings. We evaluate whether self-supervision can be an effective means of reducing catastrophic forgetting in neural networks. Our experiments in a task-incremental setting where different combinations of supervised, self-supervised, and CL regularization objectives seems to indicate that self-supervision can be used as an effective mean of enriching the representations of a neural network, by encouraging the network to learn features that generalise well across tasks, and are thus less susceptible to catastrophic forgetting.

A.1. Self-Supervised Learning

Self-supervised learning, as a form of unsupervised learning, is an approach whose goal is to learn rich representations from unlabeled data. It achieves this by learning one or more *auxiliary tasks*, which use intrinsic properties of the data itself to create auxiliary labels used to train the model's representations in a supervised fashion. Many forms of such auxiliary tasks exist. In the context of learning visual representations, most well-known auxiliary tasks usually involve applying a transformation to the input samples and then predicting the applied transformation given the representations of the transformed inputs. For example, one can transform an image dataset by applying random rotation to the images, and ask a model to predict the applied rotation [**27**]. Another popular and effective form of auxiliary task involves contrastive learning, which seeks to maximize the alignment between the representations of views of the same input example, while also

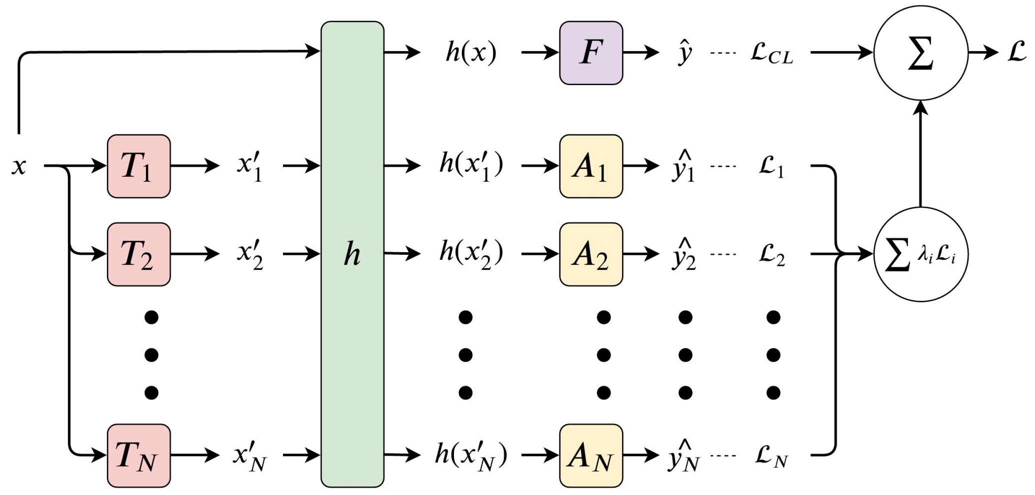


Fig. A.1. Illustration of the self-supervised learning architecture used in these experiments. Here, x denotes the original sample from, while T_i denotes the i-th input transformation function needed for the i-th auxiliary task. A shared neural encoder h is used to encode each input into its lower dimensional representation, which will then be passed to task specific networks F and A_i. Here, F represents a network responsible to solve the current task at hand, while A_i represents a network for the i-th auxiliary task. To train the networks, the objective is to minimize the combination of the loss L_{CL} with the auxiliary task losses L_i for all i.

minimizing alignment between representations of views of different examples. Regardless of which particular auxiliary task is used, the common goal is to learn representations that generalize well to other data and downstream tasks.

Our hypothesis is that forgetting in a model's representations can be reduced by training the model's representations jointly on both the supervised and self-supervised objectives. Our intuition is that self-supervised methods learn features that are independent from the supervised objective, and thus may be able to generalize better between tasks in a CL scenario. We believe that these methods and their losses can be used as regularization signals to prevent the model's representations from forgetting more general features learned from previous tasks that are relevant to the current task. This approach could therefore be classified under the regularization-based family of continual learning methods.

Furthermore, we predict that given sufficiently rich self-supervised tasks, it should be possible for a model to learn and adapt its representations continually, without using the supervised loss. We hypothesize that such a model would thereby learn features that are both general and rich enough to effectively perform a variety of downstream tasks, while also benefiting from considerably reduced forgetting. We attempt to produce such signals by combining a variety of effective auxiliary tasks. Fig. A.1 illustrates the proposed architecture.

Concretely, instead of updating the encoder model h using only the supervised loss signal, we either augment or entirely replace this loss signal with the losses from various self-supervised learning methods. In the case where we allow the supervised loss signal to backpropagate into the encoder, the self-supervised losses can be thought of as a regularization mechanism, preserving some task-invariant information in the representations. Otherwise, by detaching the hidden representations $h(x)$ from the output layer, the supervised signal is not backpropagated into the representation learner h, and thus the representations are learned solely through self-supervision. Formally, the optimization objective is to minimize the sum of the losses related to the continual learning problem and the auxiliary task losses:

$$\mathcal{L} = \mathcal{L}_{CL}(\cdot) + \sum_{i}^{N} \lambda_i \mathcal{L}_i(\cdot), \tag{A.1.1}$$

where $\mathcal{L}_{CL}(\cdot)$ denotes the continual learning task loss, $\mathcal{L}_i(\cdot)$ denotes the loss for the i-th auxiliary task, and λ_i denotes the weighting parameters for the auxiliary task losses.

A.1.1. Self-Supervised Learning methods

There are various auxiliary tasks that can be used in self-supervised learning. In this work, we use three different auxiliary tasks from different families: rotation prediction [**27**], input reconstruction with autoencoders, and contrastive learning with SimCLR [**13**]. We describe them below.

A.1.2. Rotation Prediction

We consider rotation prediction as an auxiliary task. Concretely, the original inputs obtained from the data stream are rotated randomly. The model is then asked to predict the angle of the applied rotation, given the rotated input. When the rotation angle is discretized, this auxiliary task can be posed as a classification problem. This task has been explored in [**27**], and has been shown to be beneficial in learning semantically meaningful representations.

A.1.2.1. Input Reconstruction. We also use input reconstruction as one of the auxiliary tasks. We believe the use of this task can positively impact the learned representations since the concept of reconstruction is universal. That is, the representation needed to perform reconstruction is likely to be reusable for image classification. Concretely, this corresponds to an autoencoder objective where the penalty is the mean squared error between the input and its reconstruction.

A.1.2.2. SimCLR. Contrastive learning is a self-supervised learning method that enforces similarity between representations of similar input pairs. In other words, an encoder model is trained to *contrast* between positive and negative pairs, where a positive pair is a pair of inputs that are deemed to be similar. SimCLR [**13**] is a contrastive learning framework for visual representation that maximizes similarity of two inputs in the latent space that are transformed differently. Learning representations with SimCLR has been shown to surpass the performance pretrained ImageNet models.

Given an input x, a set of transformation functions T, and an encoder f, we sampled two transformation functions, t_1 and t_2, from T, and apply them to the input x to obtain a pair of transformed inputs $\tilde{x_i} = t_1(x)$ and $\tilde{x_j} = t_2(x)$. We refer $\tilde{x_i}$ and $\tilde{x_j}$ as a positive pair since they come from the same input x. These inputs are then passed to the encoder to obtain two representations $h_i = f(\tilde{x_i})$ and $h_j = f(\tilde{x_j})$. We then apply non-linear function g (i.e., a fully connected neural network) to obtain $z_i = g(h_i)$ and $z_j = g(h_j)$. Both f and g are then updated by maximizing similarity between z_i and z_j. Nevertheless, we want the model to not only maximizing similarity in representations of a positive pair, but also to contrast the representations from different inputs (i.e., negative pairs). Concretely, when sampling a minibatch to train the models, we only sample one positive pair $\tilde{x_i}$ and $\tilde{x_j}$, and treat all other transformed inputs as negative ones. The model is then trained to optimize:

$$\mathcal{L}_{i,j} = -\log \frac{e^{sim(z_i,z_j)/\tau}}{\sum_{k=1}^{2N} \mathbb{1}_{k\neq i} e^{sim(z_i,z_k)}/\tau}, \tag{A.1.2}$$

where $sim(\cdot)$ is a function that measures similarity between two representations, N is the number of input samples, $\mathbb{1}_{k\neq i}$ is an indicator function that returns 1 whenever $k \neq i$, and τ is a temperature constant. Note that the minibatch size is $2N$ because we always apply two different transformations sampled from T for each input sample. We then discard g and only use the encoder f to get the input representation.

A.2. Experimental Setup

As previously discussed, there are different settings in continual learning that one may consider. In this experiment, we choose the task incremental learning setting. In this setting, we consider a stream of data from a sequence of tasks. The learner receives a batch of data coming from a particular task at a time to train the model until convergence. Once done with a task, the data from this task are discarded and will not appear again as the learner is presented with new tasks.

We evaluate the models on Fashion MNIST (F-MNIST), CIFAR10, and CIFAR100 datasets. For F-MNIST and CIFAR10, a continual learning task is defined as a 2-way classification task. Thus, we have in total 5 tasks for F-MNIST and CIFAR10, where each task is a 2-way classification task. For CIFAR100, a task is defined as a 20-way classification task, thus also resulting in 5 tasks in total. In all datasets, the classes in each task are chosen randomly.

We perform our experiments using models with multiple classification heads, where we train a new classification head using the data from the new task as the task changes, while the classification head from the previous tasks are frozen. For models trained using self-supervision, we first warm up the encoder h by training it using only the self-supervised signals until convergence before combining both training signals until they converge again. We train our models using Adam [**39**] as the optimizer. We ran grid search to determine the best coefficient λ_i for each of the auxiliary tasks. From this search, we use $\lambda_{reconstruction} = 0.01$, $\lambda_{rotation} = 1.0$, and $\lambda_{SimCLR} = 1.0$.

We run multiple experiments to study the contribution of each of the following factors to the performance of the network in this continual learning setting: 1) he supervised (classification) loss 2) self-supervised losses, and 3) an EWC[**40**] regularization CL penalty.

A.3. Results

For each experiment, the left-most plot illustrates the cumulative accuracy of the model during training on validation data from tasks the model have seen. For example, the data point for each line when the number of tasks learned is 4 represents the accuracy of each model on validation dataset from Task #1 to Task #4 (inclusive), after the model was trained on data from Task #4. These plots thus reflects both the forgetting and generalization errors of each model. The middle plot shows the final cumulative validation accuracy, where the validation set is the combination of validation data from all the tasks, after the each model has been trained on all tasks. Finally, the right-most plot reflects the accuracy distribution of the model on each validation set from each task at the end of training process. This plot allows us to see whether the model errors come from the tasks the model learned earlier or later during training process. If a model suffers from catastrophic forgetting, we will see that the model can only perform well on the last task, but not the earlier ones.

First, we evaluate the performance of the models when the encoder is trained using only self-supervised signals. In this experiment, the baseline is a model where the encoder h is fixed. Since we use multiple classification heads, the baseline is immune to forgetting by design. Thus, the other models can be considered to perform well if they can approach or even surpass the performance of the baseline. Similarly to other experiments, we perform each run 5 times with different seeds. The of this experiment are shown in Figure A.2

For F-MNIST, we found that `ae + simclr` combination performs the best and even surpasses the baseline. For CIFAR10, `ae + simclr` also performs the best and achieves comparable performance with the baseline. Interestingly, none of the models trained with self-supervision can approach the performance of the baseline model in CIFAR100. Perhaps this is caused by the more complex nature of the dataset. Nevertheless, these results also show how self-supervision helps to alleviate forgetting. Further evaluations are needed, especially on CIFAR100 dataset, which we leave as a future work.

We then train models using a mixture of supervised and self-supervised losses. The results of this experiment are shown in Figure A.3. From the left-most plot of Figure A.3, we see that models trained using self-supervision consistently perform better than the supervised `baseline` on all datasets. For F-MNIST, we found that the combination of input reconstruction with autoencoders and SimCLR (i.e., `ae + simclr`) performs the best, while `rotation + ae + simclr` performs the best for CIFAR10. The performance of `ae + simclr` is also the highest for CIFAR100, albeit close with `ae`, `rotation + ae`, and `rotation + ae + simclr`. At first glance, it seems that models trained with `ae` as one of the auxiliary tasks gain the most benefits. Nevertheless, to get the best possible performance, one should treat the choice of auxiliary tasks as another parameter to be "tuned". Perhaps one future research direction is to learn the best possible combination of auxiliary tasks.

Then, we combine self-supervised losses with the EWC[**40**] penalty. Comparing these results, in Figure A.5, with those from using only self-supervision from Figure A.2, we can observe that adding the EWC penalty increases the performance of these self-supervised learning methods, by preventing forgetting in these representations.

In the final experiment, we combine self-supervised, supervised, and the EWC losses when learning the representations.

Summary

In summary, these experiments show that self-supervised methods can indeed be used to mitigate catastrophic forgetting. We also found that multiple auxiliary tasks can sometimes work better than any individual task. However, there is no clear pattern as to what combination of auxiliary tasks performs consistently better than the others. Thus, one possible research direction is to develop a method that can learn the combination of auxiliary tasks that produce the best performance on some particular problems. Additionally, we only performed our experiments using three different auxiliary tasks. There exist many other self-supervised tasks, some of which might prove to be better for continual learning applications. It would be interesting to evaluate more of these tasks to see whether they can further increase the performance, and whether a pattern of good performing tasks emerges. Finally, further evaluations on other continual learning settings such as online learning would also be interesting. We believe this approach has the potential to be useful in continual learning settings where a lot of unlabelled data is available compared to labelled data, for example in the context of self-driving cars.

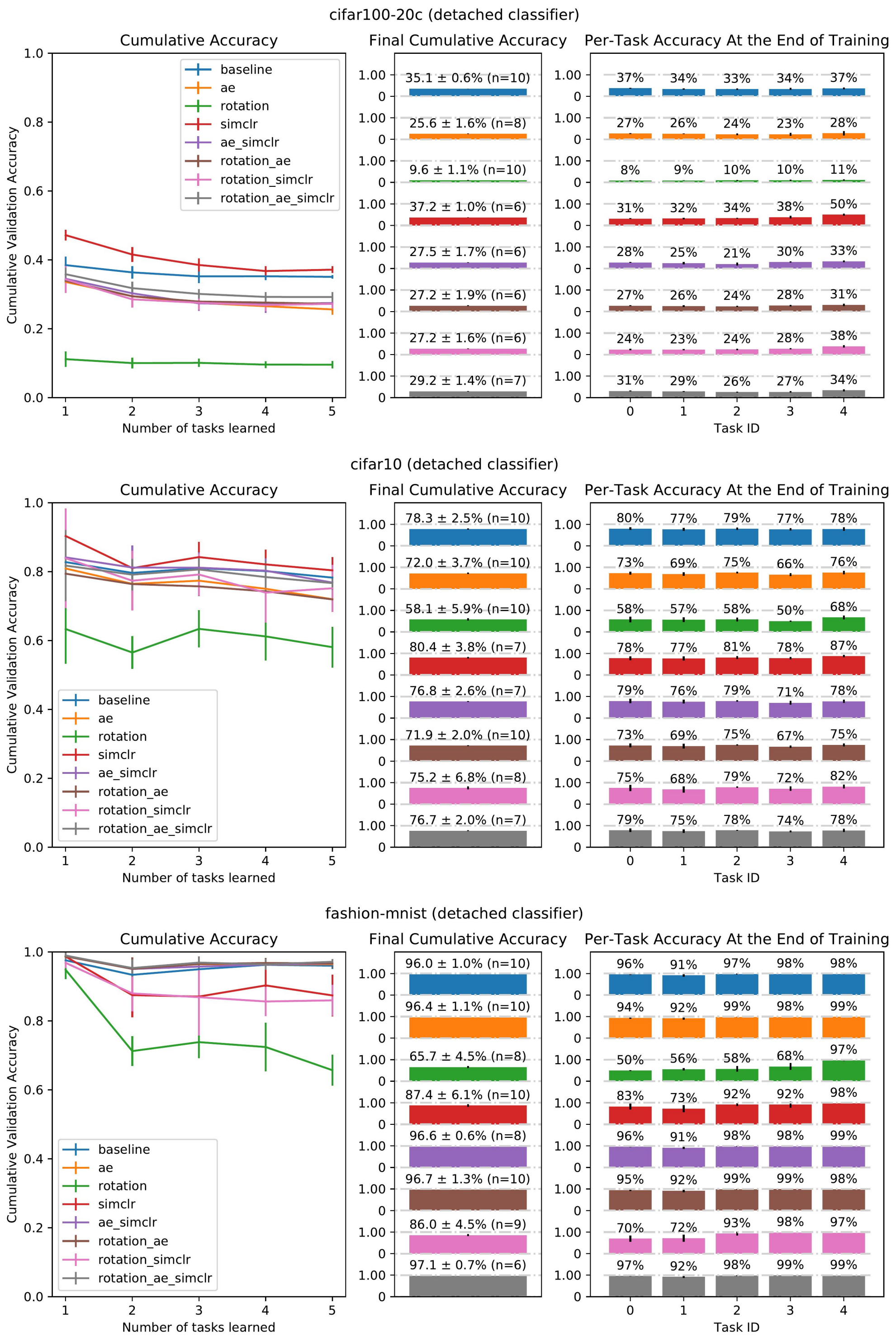


Fig. A.2. Continual Learning of representations using only Self-Supervised losses: Performance of all combinations of the selected auxiliary tasks on the Cifar100, Cifar10 and Fashion-MNIST datasets. Important to note is that the baseline in this case corresponds to a model where a classifier is trained for each task on the fixed outputs of the randomly initialized feature extractor, and as such the baseline incurs no forgetting, and is representative of the scenario where no online adaptation of the model's representations is performed.

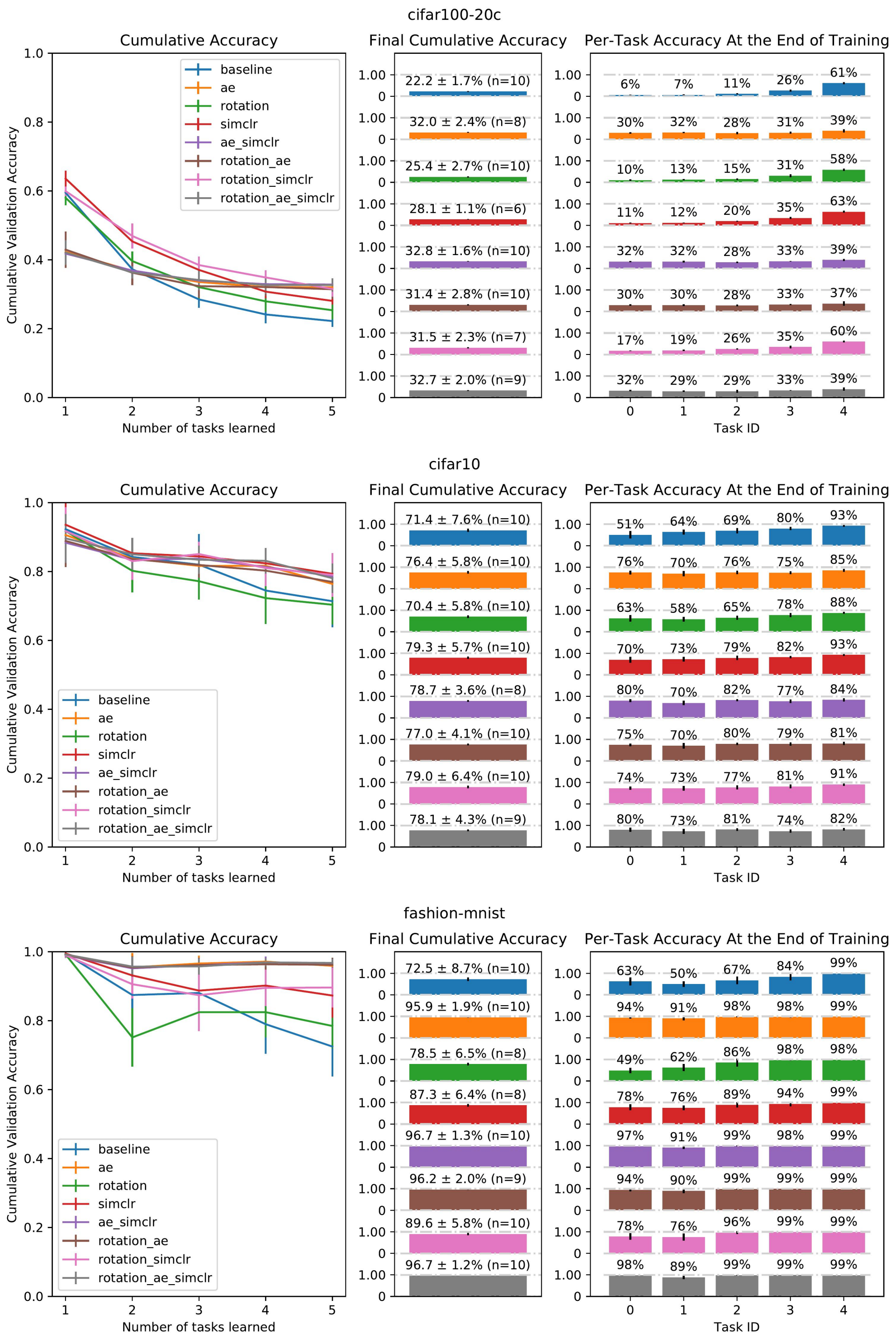


Fig. A.3. Continual Learning of representations using Self-Supervised and Supervised losses: Performance of all combinations of the selected auxiliary tasks on the Cifar100, Cifar10 and Fashion-MNIST datasets. The baseline here corresponds to a classifier without any CL-specific enhancements.

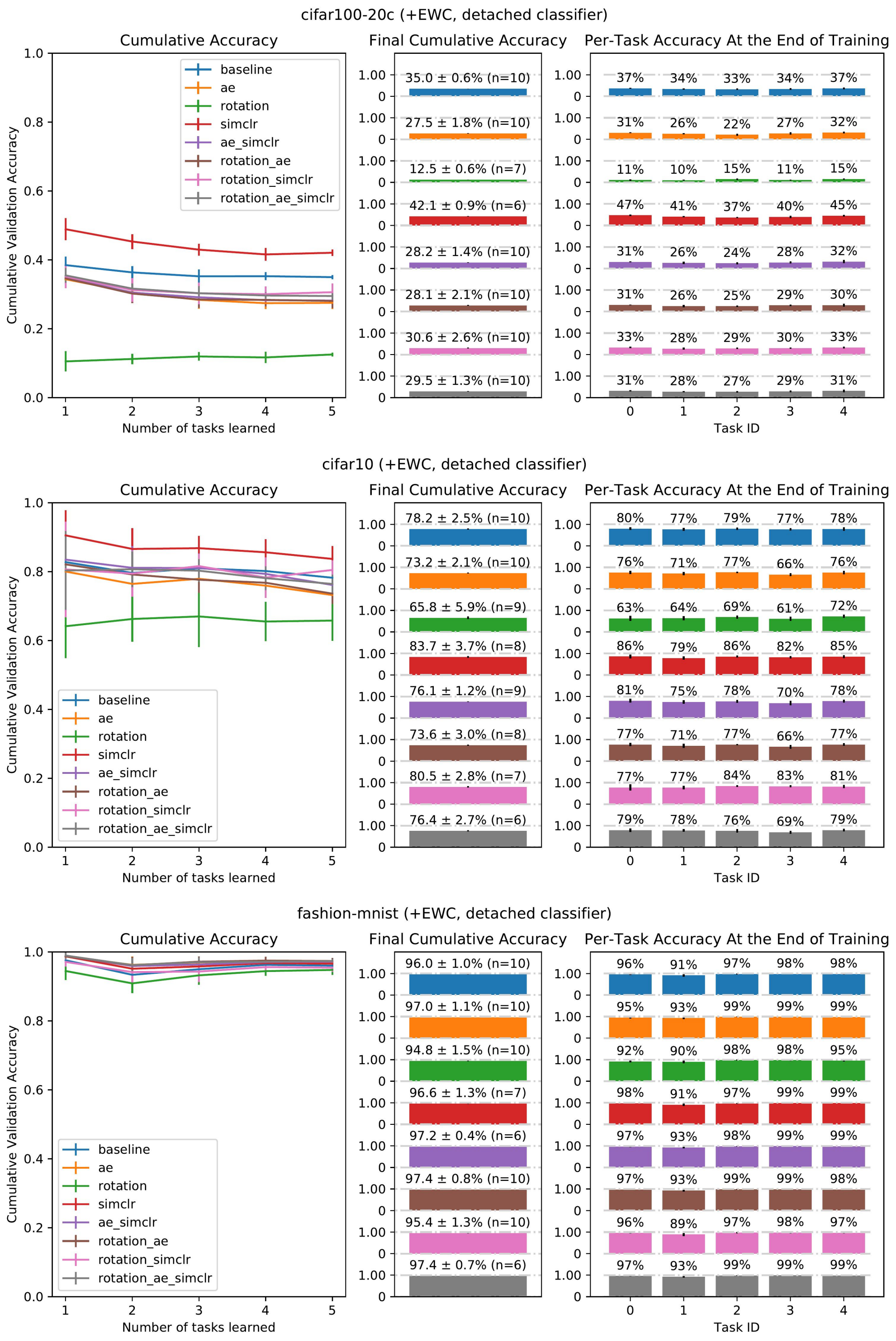


Fig. A.4. Continual Learning of representations using Self-Supervised and EWC losses: Performance of all combinations of the selected auxiliary tasks on the Cifar100, Cifar10 and Fashion-MNIST datasets, when the representations are learned using only self-supervision and an EWC penalty is introduced. This penalty effectively alleviates the catastrophic forgetting problem, at the expense of making the network less flexible.

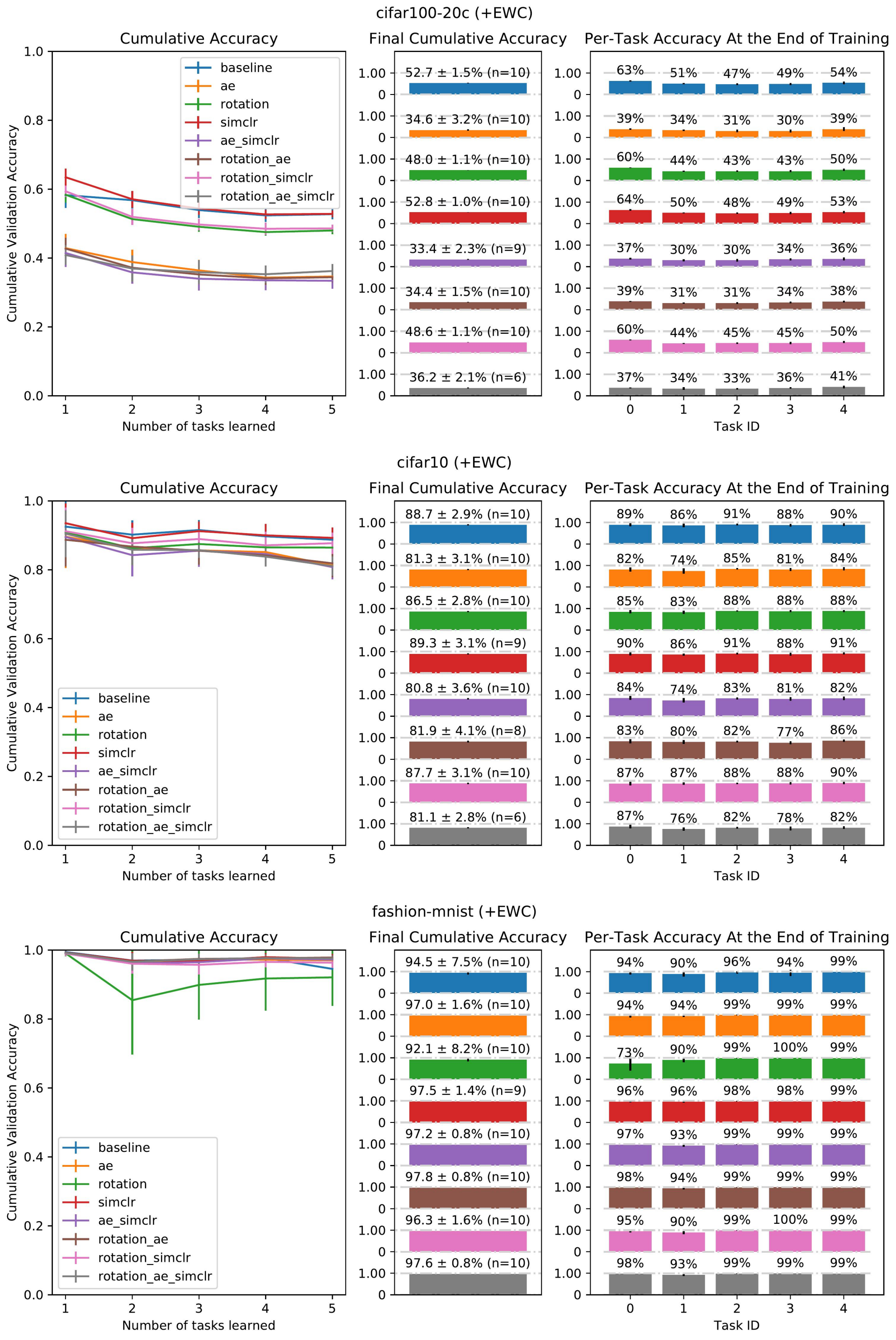


Fig. A.5. Continual Learning of representations using Self-Supervised, Supervised, and EWC losses: Performance of all combinations of the selected auxiliary tasks on the Cifar100, Cifar10 and Fashion-MNIST datasets, when an EWC penalty is introduced. This penalty effectively alleviates the catastrophic forgetting problem.

Appendix B

Extending Sequoia

B.1. Generating tasks in custom environments

As mentioned in Section 4.5, any gym-compatible environment can be used by the Settings in Sequoia, by simply passing an environment per task to the constructor of the Setting. Additionally, it is possible to add explicit support an environment, making it so Sequoia can create tasks in the environment automatically. This is done by registering a new handler to use to create tasks, as can be seen in Listing 23 for continuous tasks and Listing 24 for discrete tasks.

B.2. Adding new Settings to Sequoia

There are three ways to easily create a new setting:

(1) By extracting an assumption present in the root setting, therefore creating a new *root* or most general setting.
For example, the most general CL setting we consider in this work, Continuous, Task-Agnostic Continual Learning (CTaCL) makes an implicit assumption that the non-stationarity of the environment isn't affected by the actions of the agent. In this example, it could be argued that this kind of active non-stationarity is a more general form of non-stationarity than the passive variant. Therefore, it would follow that a method able to handle problems with active non-stationarity should also be applicable to problems with passive non-stationarity. This assumption could be extracted, to create a new (yet unamed) CL setting, with CTaCL as its child. Any Method that was previously declared to work in CTaCL will not be affected, and new methods aimed at this very challenging and general setting could be created to handle both types of settings. This is one of the major benefits of using an inheritance hierarchy to organize settings.

```
from typing import List, Dict
from sequoia.settings.rl import make_continuous_task
from gym.envs.classic_control import CartPoleEnv
import numpy as np

# A Continuous task is a dict mapping from attributes to the values
# to be set on the environment:
ContinuousTask = Dict[str, float]


@make_continuous_task.register(CartPoleEnv)
def make_task_for_my_env(
    env: CartPoleEnv, step: int, change_steps: List[int], seed: int = None, **kwargs,
) -> ContinuousTask:
    # NOTE: task sampling should be reproducible given a `seed`.
    step_seed = seed * step if seed is not None else None
    rng = np.random.default_rng(step_seed)
    return {
        "gravity": 9.8 * rng.normal(1, 0.5),
        "masscart": 1.0 * rng.normal(1, 0.5),
        "masspole": 0.1 * rng.normal(1, 0.5),
        "length": 0.5 * rng.normal(1, 0.5),
        "force_mag": 10.0 * rng.normal(1, 0.5),
        "tau": 0.02 * rng.normal(1, 0.5),
    }
```

Listing 23. Example of how to add new RL environments to Sequoia. In this example, we register a function which will be used to sample continuous tasks for this environment, allowing it to become used as part of the Continuous Task-Agnostic Continual RL Setting and all of its descendants.

(2) Creating new leaves in the tree, by adding an assumption or constraints to an existing setting. One example of this could be settings where more information is available in the observations/actions/rewards than their parents.
(3) Adding new intermediate nodes to the tree, by making the differences in assumptions between existing settings more fine-grained: for example, if a jump between two settings is too large, a new intermediary node can be introduced, also without impacting the methods that were created for the parent or the child setting.

Finally, there is another, albeit more involved way to create new settings: to introduce a new class of assumptions. This separate assumption hierarchy is then composed with existing settings, resulting in a large increase in the number of settings. For instance, if you consider the level of supervision, as-in, the availability of the rewards signal from the environment, you could recover unsupervised, semi-supervised, and “supervised”/traditional RL/SL settings. Through multiple inheritance, one could then create a new Setting for each combination of assumptions.

```python
import operator
from typing import List, Dict, Union, Callable
import numpy as np
import gym
from metaworld.envs.mujoco.sawyer_xyz.v2 import SawyerReachEnvV2
from metaworld import ML10
from sequoia.settings.rl.discrete import make_discrete_task

# In the case of Discrete RL settings, you can either return a
# 'continuous' task as before or a callable which will be
# applied onto the environment when a task boundary
# is reached:
ContinuousTask = Dict[str, float]
IncrementalTask = Union[ContinuousTask, Callable[[gym.Env], None]]

@make_discrete_task.register(SawyerReachEnvV2)
def make_discrete_task_for_metaworld_env(
    env: SawyerReachEnvV2,
    step: int,
    change_steps: List[int],
    seed: int = None,
    **kwargs,
) -> IncrementalTask:
    benchmark = ML10(seed=seed)
    rng = np.random.default_rng(seed)
    some_metaworld_task = rng.choice(benchmark.train_tasks)
    # NOTE: Equivalent to the following, but has the benefit of
    # being pickleable for vectorized envs:
    # return lambda env: env.set_task(some_metaworld_task)
    return operator.methodcaller("set_task", some_metaworld_task)
```

Listing 24. Example of how to add support for generating discrete tasks in new RL environments. These tasks are applied when a task boundary is reached. In this example, we register a function which will be used to sample discrete tasks for a common type of MuJoCO[**80**] environment, making it easier to use as part of the Discrete Task-Agnostic RL setting and its descendants.

The need for each of these variations to be defined as classes is one shortcoming of the Sequoia framework as it currently stands, and will be addressed in future work using structural, rather than nominal subtyping, as described in Chapter 5.

```
import gym
from sequoia.settings import Setting, Environment, Observations, Actions
from sequoia.methods import Method

class DemoModel:
    def forward(self, observations: Observations) -> Actions:
        ...

class DemoMethod(Method, target_setting=Setting):
    """ Pseudocode for a Method that targets a given Setting. """
    def configure(self, setting: Setting):
        # Called by the setting before training begins.
        self.model = DemoModel(setting.observation_space, setting.action_space,
                               setting.reward_space, nb_tasks=setting.nb_tasks)
        self.optimizer = Adam(...)

    def fit(self, train_env: Environment, valid_env: Environment):
        # Train a model using these environments from the setting.
        # Note: all Environments are gym environments. More on this later.
        for epoch in range(self.n_epochs):
            self.model.train_epoch(train_env)
            self.model.validation_epoch(valid_env)

    def get_actions(self, observations: Observations) -> Actions:
        # Called by the setting for inference (at test-time).
        actions = self.model(observations)
        return actions

    def on_task_switch(self, task_id: Optional[int]):
        # Gets called on task boundaries, depending on the setting.
        self.model.prepare_for_new_task(new_task=task_id)
```

Listing 25. Pseudocode example of how a CL method might be created in Sequoia.

Appendix C

Comparison with Avalanche

Avalanche is one of the most well-established software libraries for continual supervised learning research. Sequoia and Avalanche have similar goals: To unify and stardardize the research settings and methods in continual learning.

Before we discuss similarities and differences between the two frameworks in more detail, we must first describe how methods are implemented in Avalanche. The equivalent of methods in Avalanche are called *strategies*. All strategies in avalanche inherit from the BaseStrategy, which includes all the base training and evaluation logic, as well as various methods that serve as hooks so that subclasses can easily customize different parts of the training loop.

This BaseStrategy can also be customized using a modular plugin interface, such that multiple plugins can be added to the same strategy to combine different CL algorithms. These strategies interact with a series of *experiences* from a *stream*.

Here are some of the similarities between Sequoia and Avalanche.

- *streams* in Avalanche are somewhat similar to Settings in Sequoia.
- Sequoia's BaseMethod serves the same purpose as Avalanche's BaseStrategy.
- Sequoia's AuxiliaryTask serves the same purpose as Avalanche's StrategyPlugin:
  - Plugins in Avalanche are similar to callbacks in PyTorch Lightning.
  - Sequoia's AuxiliaryTask inherits from PL's Callback class
  - Plugins have but a few additional hooks compared to PL's Callbacks, for instance, they can modify the dataset before training.
- Adding plugins on top of the BaseStrategy is analogous to Adding auxiliary tasks on top of the BaseMethod. For example:
  - Avalanche's EWC Strategy adds the EWCPlugin to the BaseStrategy
  - Sequoia's EWC Method adds the EWC Auxiliary task to the BaseMethod

However, there are notable differences between Avalanche and Sequoia, the first of which lies is in their philosophy and design approaches:

- Sequoia places no constraints on the design of a Method.
- Sequoia delegates the responsibility of creating methods to other, more specialized libraries. It makes these methods applicable in a way that makes them reusable on any current or future applicable setting via polymorphism.

To illustrate this point, Sequoia could eventually include Avalanche scenarios as settings for CSL, if there were to be a clear hierarchical structure to their properties and assumptions. We instead decided to make Avalanche's methods applicable onto our own settings.

Along the same idea, Sequoia delegates the responsibility of doing high-performance training to a specialized framework, in this case PyTorch-Lightning. As a result, Sequoia is able to directly leverage all the nice features of PyTorch-Lightning such as training using multiple-GPUs and TPUs. This also leads to less code duplication. All other features of PL (mixed precision, DP, DDP, TPUs, logging, etc) are also already usable with the BaseMethod.

When adapting the methods from Avalanche to be applicable on the CSL settings in Sequoia, we also made some additions to them in order to widen their applicability. For example, we added a simple form of task inference to some methods, so that they could be applied to the CSL settings in Sequoia that do not provide task labels at test-time.

Another difference is the fact that Sequoia describes both SL and RL using the same abstractions. Avalanche-rl recently became available as an RL-version of Avalanche. It is however unclear how much overlap there is between the RL and SL versions of this library, and if it is possible to create a CL strategy that is applicable in both types of scenarios. In Sequoia, this kind of reuse is made possible in the BaseMethod by the BaseModel using a different type of output head for each setting (PolicyHead vs ClassificationHead).

One other difference is that, to the best of our knowledge, there current is no way to statically determine which scenarios a strategy can be applied to in Avalanche.

In summary, here is how to transform Avalanche's BaseStrategy into Sequoia's BaseMethod:

(1) Create RL Scenarios that use gym environments as the "experience source"/dataset.
(2) Add a way to create non-stationarity in the RL environments.
(3) Create structured objects for the data types that environments/datasets yield and accept. (E.g. Observation / Action / Reward dataclasses, for example)
(4) Refactor the BaseStrategy:
    - Make it use a Trainer from PyTorch Lightning instead of doing the training itself.
    - Make this BaseStrategy agnostic to RL or SL. A good way to go about this is to use a different type of output head for each setting, for instance. Could also use a completely different BaseModel, if that's simpler.
    - Refactor the BaseModel-equivalent in Avalanche to be a LightningModule.
(5) Refactor the "Plugin" interface:

- make it extend the Callback interface from PL, adding necessary hooks and methods specific to CL.
- refactor the CL plugins in Avalanche so that they inherit from this new Plugin interface. For the strategies that only apply in RL or only in SL, explicitly state that assumption programmatically using a property or registry of some sort.

Appendix D

Additional implementation details

D.1. Continual-MonsterKong environment

With rapid advancements in the field of deep RL, continual RL or *never-ending*-RL has witnessed rekindled interest towards the goal for broad-AI in recent years. While significant progress has been made in related domains such as transfer learning [**77**], multi-task learning [**4, 62, 9, 55, 5, 44**], and generalization in RL [**20**], an outstanding bottleneck is the lack of standard tools to develop and evaluate CRL agents [**38**]. A standardized benchmark will potentially enable rapid research and development of CRL agents. To this end, we propose a new CRL benchmark within the unified framework of Sequoia. In particular, we build the CRL benchmark leveraging the Pygame learning environment `MonsterKong` [**76**]. MonsterKong is pixel-based, lightweight and has an easily-customizable domain, making it a good choice for evaluating continual learning agents.

Specifically, we design *tasks* through a variety of map configurations. These configurations vary in terms of the location of the goal and the location of coins within each level. We introduce randomness across runs of a task by varying the start locations of the agent. To incorporate the ability to evaluate across specific CRL characteristics, we leverage tasks to define CRL *experiments*. We design families of tasks leveraging the following abstract concepts: *jumping tasks* which require the agent to perform jumps across platforms of different lengths in order to collect coins and reach the goal, *climbing tasks* which require the agent to competently navigate ladders in order to collect coins and reach the goal, and tasks that combine both of these skills. The specific tasks leveraged as part of the CRL competition are depicted in Figure D.1. The agent trains on each task for 200,000 steps.

Experiment Details: To evaluate the agents on the CRL benchmark, we follow the standard evaluation introduced above. Final performance reports accumulated reward per episode on all test environments, averaged over all tasks, after the end of training, whereas online performance is measured as the accumulated reward per episode on the training environment of the current task during training of all tasks. For the runtime score, we use

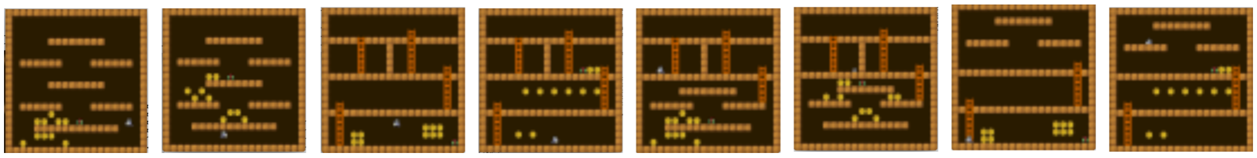


Fig. D.1. Continual-MonsterKong. We display the 8 tasks that constitute the benchmark in chronological order. The first two tasks test the agent's ability to jump between platforms, the second two test its ability to climb ladders and the last four combine both skills.

set *max_runtime* of 12 hours and *min_runtime* to 1.5 hours. Lastly, the agents are allowed a maximum of 200,000 steps per task.

Customization: Ideally, CRL agents must be able to solve tasks by acquiring knowledge in the form of skills, be able to use previously acquired behaviors, and build even more complex behaviours over the course of its lifetime [**69, 78, 79**]. While leveraging the MonsterKong environment, it is easy to introduce new environment layouts or modifications to existing layouts. Configurations could be customized to include arbitrary configurations of coins, ladders, platforms, walls, monsters, fire balls, and spikes. Making custom environment elements is straightforward as well, so the environment can be modified to aligned with the properties of the CRL agent that we would like to test.

While in our benchmark we mainly focused on three families of tasks within the Monsterkong domain, it is fairly straightforward to introduce variations of map configurations to the framework. Monsterkong provides two degrees of design choices 1. the task definitions and 2. the evolution of tasks referred to as experiment definitions. Due to the nature of how tasks are specified through simple matrices (map configurations), many layers of complexity can be added through the task specification. For example, object addition and removal can induce local variations in reward, nails can be penalizing, diamonds can be bonuses. Additionally, changes to the textures of the game like simple changes to the color of the walls, the coins, and the background as well as changes in the lighting are easy to add for users interested to test generalization of the policies learned.

D.1.1. Split-Synbols dataset

Currently employed datasets can't be used sensibly to construct domain-incremental learning problems. Some have used MNIST to construct Permuted-MNIST and Rotated-MNIST, however, [**21**] have explained and demonstrated why such benchmarks are flawed and bias their results unfairly towards some methods. Motivated by this, we introduce Split-Synbols. Based on the Synbols dataset [**43**], a character classification dataset in which examples have an extra label corresponding to their font, one can easily construct sensible domain-incremental benchmarks where e.g., a font would consist of a domain.

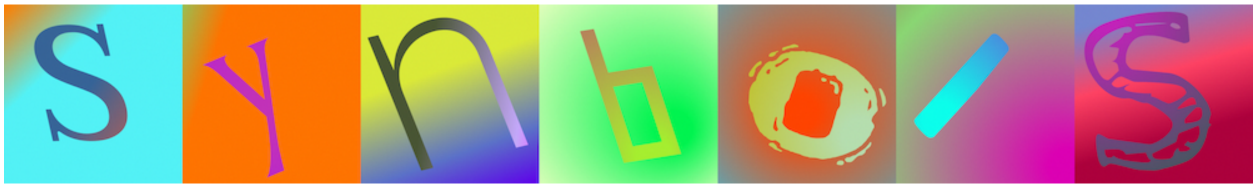


Fig. D.2. Split-Synbols. Example of Synbols character to classify.

For the experiments, however, we opted for a class-incremental version to increase the difficulty. We prescribe a segmentation into 12 tasks to be learned sequentially, each consisting of a 4-way classification problem. Examples of samples from the Synbols dataset are displayed in Figure D.2.

```python
class Trainer:
    """ Simplified Pseudocode for the Trainer class of Pytorch-Lightning. """
    def __init__(self, max_epochs: int, **lots_of_other_options):
        self.max_epochs = max_epochs
        ...
    def fit(self, model: LightningModule[T], datamodule: DataModule[T]) -> None:
        optimizer = model.configure_optimizers()
        for epoch in range(self.max_epochs):
            # training loop:
            model.train()
            for batch in datamodule.train_dataloader():
                optimizer.zero_grad()
                train_loss = model.train_step(batch)
                train_loss.backward()
                optimizer.step()
            # validation loop:
            model.eval()
            for val_batch in datamodule.val_dataloader():
                epoch_validation_loss += model.val_step(val_batch)
            ...
    def test(self, model: LightningModule[T], datamodule: DataModule[T]) -> dict:
        test_losses = []
        for test_batch in datamodule.test_dataloader():
            test_losses.append(model.test_step(test_batch))
        return {"Test loss": sum(test_metrics)}
```

Listing 26. Pseudocode for the Trainer class from PyTorch-Lightning