Bachelor Thesis

Planning experiments with meta-learned priors

Jan Olivetti Auladell

Advisor (MIT): Ferran Alet

Tutor (UPC): Javier Béjar

June 2022

In partial fulfillment of the requirements for the

Bachelor's Degree in Mathematics

Bachelor's Degree in Computer Engineering

SUMMARY

We consider a setting where experiments of widely different costs can be performed to obtain several properties of an entity and a reward is obtained for confirming that the entity does satisfy a property. In that case, it is sometimes beneficial to perform a cheap experiment in order to obtain partial information on the desired property, if corresponding experiment is much costlier. Our goal is to train a neural network that can incorporate new information from experiments in order to output better predictions, and then use this neural network in a planner that will determine what sequence of experiments will maximize the expected value. In this project, the entities will be molecules.

We introduce the concept of transposed meta-learning, which is similar to meta-learning but instead we aim to create a model that, for unseen inputs, can generalize to unseen tasks after seeing just a few example tasks. We consider an architecture-based approach and an optimization-based approach and find that the latter yields better accuracy. We employ a meta-learning technique called CNGrad to meta-train on several inputs in parallel.

In the planner, we perform tree search to find the best policy for each set of already known properties. We consider best-first search with several hard-coded heuristics as well as a planner inspired by AlphaZero with the intention that it would learn the best tree exploration method. Out of the heuristics that we try, the best one is exploring the combinations of experiments with the lowest total cost first. Although the AlphaZero planner improves its performance compared to not doing planning by learning, its performance is not as good as best-first search.

This work shows the potential usefulness of incorporating new information to obtain better predictions even if that additional information comes at a cost, and it also provides a novel meta-learning framework that can be used in other problems aside from this one.

Keywords: deep learning, meta-learning, planning, tree search

AMS Code: 68T05

RESUM

Considerem una situació on experiments de costos molt diferents poden ser realitzats per obtenir diverses propietats d'una entitat i una recompensa és obtinguda per confirmar que la entitat satisfà una propiteat. En aquest cas, de vegades és beneficiós realitzar un experiment barat per obtenir informació parcial sobre la propietat desitjada, si el corresponent experiment és més car. El nostre objectiu és entrenar una xarxa neuronal que pot incorporar informació nova dels experiments per donar millors prediccions, i després utilitzar aquesta xarxa neuronal en un planificador que determinarà quina sequència d'experiments maximitza el valor esperat. En aquest projecte, les entitats seran molècules.

Introduïm el concepte de meta-aprentatge transposat, que és semblant al meta-aprenentatge però volem crear un model que, per entrades no vistes, pot generalitzar a tasques no vistes després de veure només unes poques tasques d'exemple. Considerem un enfocament basat en arquitectura i un altre en optimització i descobrim que el darrer dóna millor precisió. Utilitzem una tècnica de meta-aprentatge anomenada CNGrad per meta-entrenar en diverses entrades en paral·lel.

En el planificador, realitzem cerca en arbre per trobar la millor decisió per cada conjunt de propietats ja conegudes. Analitzem best-first search amb diverses heurístiques i un planificador basat en AlphaZero amb la intenció d'entrenar el millor mètode d'exploració de l'arbre. De les heurístiques que provem, la millor és explorar les combinacions d'experiments amb el menor cost total primer. Malgrat que el planificador d'AlphaZero millora el rendiment respecte a no planificar, el seu rendiment no és tan bo com best-first search.

Aquest treball mostra la potencial utilitat d'incorporar informació nova per obtenir millors prediccions encara que aquesta informació addicional incorri un cost, i també planteja un marc de meta-aprenentatge nou que pot ser utilitzat en altres problemes.

Paraules clau: aprenentatge profund, meta-aprenentatge, planificació, cerca en arbre

Codi AMS: 68T05

RESUMEN

Consideramos una situación en la cual tenemos experimentos de costes muy distintos que pueden ser realizados para obtener diversas propiedades de una entidad y recompensa es obtenida por confirmar que la entidad satisface una propiedad. En este caso, a veces es beneficioso realizar un experimento barato para obtener información parcial sobre la propiedad deseada, si el experimento correspondiente es más caro. Nuestro objetivo es entrenar una red neuronal que puede incorporar información nueva de los experimentos para producir mejores predicciones, y luego utilitzar esa red en un planificador que determinará qué secuencia de experimentos maximiza el valor esperado. En este proyecto, las entidades serán moléculas.

Introducimos el concepto de meta-aprendizaje transpuesto, que es similar al meta-aprendizaje pero con el objetivo de crear un modelo que, para entradas no vistas, generaliza a tareas no vistas tras ver unas pocas tareas de ejemplo. Consideramos un enfoque basado en arquitectura y otro en optimización y descubrimos que el segundo tiene mejor precisión. Usamos una técnica de meta-aprendizaje llamada CNGrad para meta-entrenar sobre varias entradas en paralelo.

En el planificador, realizamos búsqueda en árbol para encontrar la mejor regla para cada conjunto de propiedades ya conocidas. Analizamos best-first search con varias heurísticas y un planificador basado en AlphaZero con la intención de entrenar el mejor método de exploración en el árbol. De las heurísticas que probamos, la mejor es explorar las combinaciones de experimentos con el coste total más bajo primero. Pese a que el planner de AlphaZero mejora su rendimiento comparado con no planear, su rendimiento no es tan bueno como best-first search.

Este trabajo muestra la potencial utilidad de incorporar información nueva para obtener mejores predicciones incluso si esta información adicional incurre un coste, y planteamos un marco de meta-learning nuevo que puede ser utilizado en otros problemas.

Palabras clave: aprendizaje profundo, meta-aprendizaje, planificación, búsqueda en árbol

Código AMS: 68T05

Contents

Chapter 1

Introduction

Suppose we have an entity and an agent that can perform several experiments on that entity. Each of these experiments comes at a cost, and the costs of these experiments vary greatly. We obtain a reward for confirming that the entity satisfies a particular property. If the corresponding experiment is particularly expensive, it might be better to try one of the much cheaper experiments first to hopefully gain some information to decide whether the expensive experiment is worth it.

For example, suppose we have a robotic butler whose job is to prepare meals for its owner. This butler could, for instance, choose to serve an expensive wine in the hopes that the owner will really like it. However, before taking the risk, it might be wise to serve a cheaper wine first to check if the owner likes wine at all. Or maybe the butler can figure out that liking certain foods is correlated with liking wine so maybe it tries serving one of those foods instead to gain information about the owner's preferences at the lowest cost.

In another scenario, imagine that a pharmaceutical company has just discovered a new drug that is able to cure a certain disease, but they need to test whether that drug is safe on humans. Doing the experiment would be very expensive, but if successful, the profits would be large. Instead, the company might decide to test whether the drug is safe on mice first at a much lower cost, in order to gain information on the likelihood that the drug is safe on humans. There could be other properties you could analyze of the drug that might give less information about safety on humans but since the cost is even cheaper it might be worth it.

1.1 Formal definition of the problem

We have an entity E and a set of experiments $\mathbf{f}e_1, e_2, ..., e_n\mathbf{g}$ we can perform on that entity. Each experiment e_i has a cost c_i and its true value t_i which is a binary result that can be either 0 or 1 and is hidden from us. One of the n experiments, e_g, is the *goal* experiment, and it will give us a reward R_g if we perform e_g and we get $t_g = 1$ as the result. We define the vector $k \in \mathbf{R}^n$ as the vector of known properties, which contains the outcomes of experiments we already performed. $k_i = -1$ indicates that we have not performed e_i yet and as such we do not know its true value, but if we have performed e_i then $k_i = t_i$, the true value. In the initial step, $k_i = -1 \forall i$, indicating that we do not know the outcomes of any of the experiments at the start. At each step, we can either stop, giving us a total payout of $R(k) = R_g - \sum_{k_i \neq -1} c_i$ if $k_i = 1$ or $R(k) = -\sum_{k_i \neq -1} c_i$ otherwise, or choose one of the remaining experiments e_i and learn the value of t_i by paying the

cost c_i.

Our goal is to create a model f that takes E and k as an input and outputs $p \in \mathbb{R}^n$, a probability vector where p_i represents the probability of t_i being 1. Then, we will create a planner that uses the predictions of model f to create a policy of which action to take for each k that will maximize the expected value of the payout. In this report, we will tackle the case in which the entity is a molecule and the each experiment determines one property of the molecule.

Chapter 2

Related work

When it comes to molecular property prediction, large datasets have already been created to be used to train neural networks. In this project, we will use the ChEMBL20 dataset Bento et al., which is a curated dataset of over 400,000 molecules and 902 properties chosen such that each property would be known for at least 128 molecules of the dataset. This amount of data points for each task is sufficient to train a multitask model.

Research done in Nguyen, Kreatsoulas, and Branson leverages this dataset in order to meta-learn initializations that can be used to adapt to new tasks by seeing just a few molecules in that task. They use the GGNN architecture from Li et al. Cho et al., which we also use to make predictions on the molecules.

The concept of meta-learning has also been studied extensively, and different approaches have been devised. One of the most well-known, MAML Finn, Abbeel, and Levine, uses second-order optimization in order to find initializations that can perform few-shot learning for unseen tasks. This framework is the basis for our

optimization-based approach. In Sun et al., they explore a scenario where the training and validation samples come from different distributions, and how they train on a single training sample to obtain better predictions for the rest. Some meta-learning approaches change the architecture of the neural network, such as Flennerhag et al. adding warp layers in the model and a different optimization approach to extend meta-learning beyond just few-shot learning, or Alet et al. proposing the idea of adding a conditional normalization layer to meta-learn multiple tasks in parallel. In Alet et al., meta-learning is used to train a tailoring loss that aims to find a conserved quantity between frames of a video in order to make better predictions of future frames. Datasets to test the effectiveness of meta-learning have also been created, such as Triantafillou et al.

A lot of work has been done studying the use of neural networks in planning as well. For example, Wu, Say, and Sanner proposes a method to use a planner with a neural network for continuous problems instead of discrete ones. Pu, Kaelbling, and Solar-Lezama formulates a new architecture that can incorporate new information and use it to perform better predictions, which can be used to plan which queries to perform next.

AlphaZero Silver et al. has been used in perfect information two-player games to learn by playing against itself using Monte Carlo Tree Search and achieve superhuman performance. This framework can be adapted to train an agent that computes a policy in a wide variety of problems. In MuZero Schrittwieser et al., they change the architecture so it can learn to play games with an environment that can affect the game. In our problem, this would correspond to the outcome

of an experiment not being known to us until we perform it. EfficientZero Ye et al. expands on this framework even further to make learning more efficient. The work in Ozair et al., also based on AlphaZero, learns how to play games when the environment is stochastic or partially hidden.

Chapter 3

Modeling

3.1 Transposed meta-learning

In the classic formulation of meta-learning, we have some known tasks, and for each of these tasks we have some known inputs and their corresponding outputs. The goal is to learn from this dataset in such a way that if we receive a new, unseen task, we can generalize to unseen inputs after receiving just a few example inputs of that task. Thrun and Pratt Vilalta and Drissi Vanschoren By itself, this framework is not suitable to our problem because in the planning stage we will be receiving an unseen input and we will need to generalize to unseen tasks after receiving just a few example tasks of that input. What we can do, however, is to transpose the original formulation of meta-learning, switching the roles of inputs and tasks. We transpose our dataset so that we have some known inputs, and for each of these inputs we have some known tasks and their corresponding outputs. We will be learning from this dataset such that when we receive a new, unseen

input, we can generalize to unseen tasks after receiving just a few example tasks of that input. This means that, in the planning stage, we will be able to update our predictions for the remaining tasks (including the goal task) every time we perform an experiment and obtain new information about one of the tasks of that input. In our specific example, the inputs are molecules and the tasks are the different properties of the molecule. This new formulation will be the basis of our training of the models.

From a data collection perspective, it is easy to obtain data from cheap tasks but difficult to obtain data from expensive tasks. Therefore, we will know the cheap tasks for many inputs but the expensive tasks for only a few inputs. As such, we can train on the inputs for which we know both the cheap tasks and the expensive tasks by treating the cheap tasks as the few "known" example tasks and using them to generalize to the "unseen" expensive tasks. The hope is that for the rest of molecules where we do not know the expensive tasks, we can generalize to them after only seeing the results of the cheap tasks. Therefore, this formulation can be useful for other meta-learning problems and not just the exact one we are tackling in this project.

3.2 Gated Graph Neural Network architecture

Each input molecule will be represented as the adjacency matrix of a graph G with V nodes, with each node representing an atom and each edge representing a bond between two atoms. Each atom in the input will also have a feature vector

with F features associated with it. To make predictions, we use a variant of the Gated Graph Neural Network Li et al. Nguyen, Kreatsoulas, and Branson, which operates in two phases: the message passing phase and the readout phase. In the message passing phase, each node v's hidden representation h_v at layer t will be updated according to the hidden representation of neighboring nodes $N(v)$ using the following formula:

$$m_v^{t+1} = A_{e_{vv}} h_v^t + \sum_{w \in N(v)} A_{e_{vw}} h_w^t$$

$$h_v^{t+1} = GRU(h_v^t, m_v^{t+1})$$

where $A_{e_{vw}} \in \mathbb{R}^{F \times F}$ is a learnable weight matrix, $m_v \in \mathbb{R}^F$ is the message passed from the neighbors of v to update the hidden representation of v and GRU is the gated recurrent unit. Cho et al. A total of 7 such layers are used in the message passing phase, each with a different weight matrix. In the readout phase, the feature vectors from all nodes are summed to obtain the representation of the molecule $h = \sum_{v \in G} h_v^T$. To obtain predictions for each property of the molecule, we will add additional layers in the readout phase after obtaining the feature representation of the molecule. In some models we will also add layers between the input and the first message passing layer.

3.3 Multitask model

One way to obtain predictions for each property is to add a 2-layer multilayer perceptron (MLP), followed by a sigmoid layer that takes the molecule's hidden representation h as an input and outputs the probabilities p of each property of the molecule being a 1. We use the Binary Cross Entropy loss (BCE) to train all the models we will propose, including this one.

This multitask model is unable to incorporate new information about the experiments we perform in the planning phase, and as such it is only good to obtain an initial prediction p_g and perform the goal experiment if $E = R_g p_g \quad c_g > 0$. In theory, this model should be enough to obtain perfectly accurate predictions since the input is Markovian and it is possible to deduce all the properties by only looking at the molecule. However, in practice, it is unrealistic to expect a perfectly accurate model and as such finding a way to incorporate extra information from the experiments can improve prediction accuracy.

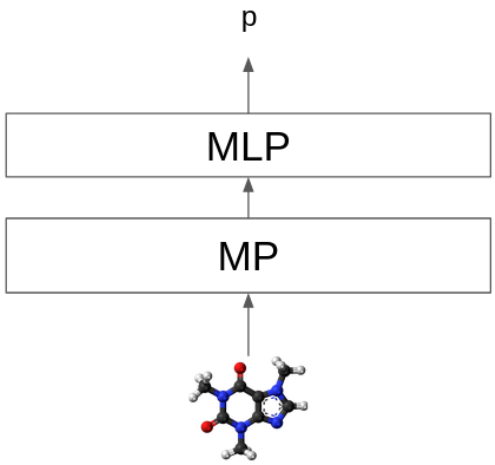


Figure 3.1 Architecture of the multitask model. MP is a black box containing all the layers of the message passing phase of the GGNN. After the MLP, a sigmoid layer is used to clamp the probabilities to between 0 and 1.

3.4 Multitask model with extended input

A simple way of incorporating properties we already know is to include them as part of the input. We will use the approach described in Pu, Kaelbling, and Solar-Lezama as the *implication model*. In the multitask model with extended input, we represent the known properties as a vector $r \in \mathbb{R}^{2k}$, where k is the number of tasks. For each task i, if $k_i = 0$ we set $r_{2i} = 1$ and $r_{2i+1} = 0$, and if $k_i = 1$ we set $r_{2i} = 0$ and $r_{2i+i} = 1$. If $k_i = -1$, meaning that we have not performed experiment i yet, we set both positions of v to 0. We feed r to a 2-layer MLP to obtain a condensed representation $h_r \in \mathbb{R}^{F_r}$, concatenate it to the feature vector of each node which means each node will have $F + F_r$ features, and then feed those extended feature vectors to another 2-layer MLP to obtain new feature vectors with only F features. These new feature vectors will be used as the input for the message passing layers. In the planner we will not be performing many experiments on the same molecule because the cost of doing so increases while the information we gain from each additional experiment decreases, so to better simulate the conditions of the planner, for each training molecule we will only incorporate between 0 and 6 known properties from the set of cheap experiments (chosen at random) instead of all the known ones from the ChEMBL20 dataset.

While this model is able to incorporate information from the experiments, there are two shortcomings with it. The first one is that we need a large weight matrix for the first MLP since there are almost a thousand properties in the dataset, but we will be using at most 6 of them for each input. This also means that the input will be flooded with zeros, which might make learning more difficult. The second

is that there is a generalization gap: since the known properties are tied to each input molecule, this means that for new inputs the relationship between molecule and properties has to be retrained.

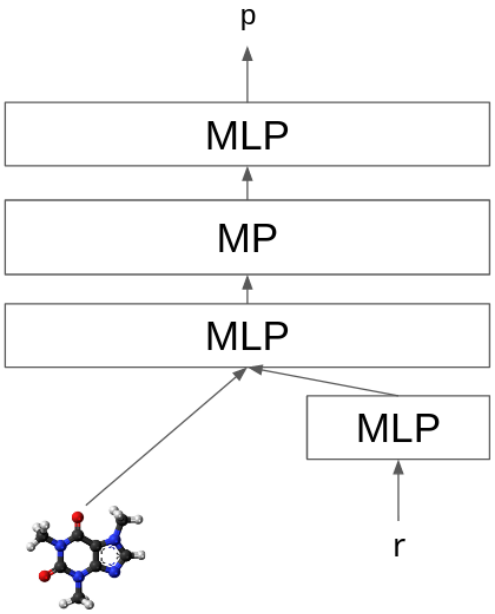


Figure 3.2 Architecture of the multitask model with extended input. We feed the vector k of known properties into an MLP before concatenating it with the feature vector of each atom.

3.5 Transposed MAML

Instead of using an architecture-based approach like in the last section, we can use an optimization-based approach instead to address those shortcomings. In the classic formulation of MAML, we split tasks into a meta-training set T^{tr} and a meta-validation set T^{val}, and each task will contain a set of training samples $D_{T_i}^{tr}$ and a set of validation samples $D_{T_i}^{val}$. The goal is to learn an initialization θ_0 from which we can adapt to new tasks by seeing only a few samples of that task. Meta-training occurs in two stages: the inner loop and the outer loop. Finn,

Abbeel, and Levine

In the inner loop, we use the training samples $D_{T_i}^{tr}$ to adapt θ_0 to a task $T_i \sim p(\mathcal{T}^{tr})$ by performing N gradient descent steps on a neural network f parametrized by θ using the training samples:

$$\theta_N^i = \theta_{N-1} - \alpha \nabla_\theta \mathcal{L}_{D_{T_i}^{tr}}(f_{\theta_{N-1}^i})$$

where α is the inner learning rate. We repeat this process for a batch of tasks in $D_{T_i}^{tr}$ to obtain θ_N^i, the parameters of a model f that can make predictions on new inputs of task T_i.

In the outer loop, we learn the initialization θ_0 by using the validation samples $D_{T_i}^{val}$ to calculate the meta-loss, which is the sum of the losses of all the tasks in the batch:

$$\mathcal{L}_{meta}(\theta_0) = \sum_i \mathcal{L}_{D_{T_i}^{val}}(f_{\theta_N^i})$$

and then we use the meta-loss to optimize θ_0 using gradient descent:

$$\theta_0 \quad \theta_0 - \beta \nabla_\theta \mathcal{L}(\theta_0)$$

where β is the outer learning rate.

Meta-validation is the same process, but we use tasks $T_i \sim p(\mathcal{T}^{val})$ and we do not update θ_0 after the outer loop.

However, as discussed in Section 3.1, we are using transposed meta-learning instead of classic meta-learning, which means that $\mathcal{T}^{tr}$ and $\mathcal{T}^{val}$ will be sets of

molecule, and that each molecule T_i will contain a set of training properties $D_{T_i}^{tr}$ and a set of validation properties $D_{T_i}^{val}$. We choose the training properties to be the cheap ones and the validation properties to be the expensive ones, with the goal that the model learns to generalize to expensive properties after only seeing some of the cheap ones. Meta-training and meta-validation are the same as in the classic formulation of MAML but with this transposed dataset. The architecture used is the same as the multitask model (Section 3.3). Like in the multitask model with extended input (Section 3.4), instead of using all the training properties in ChEMBL20, we will randomly select between 0 and 6 of them to use for each molecule. Since we allow the possibility of a molecule having no training properties, this means that the initialization θ_0 must already provide good predictions for the properties of the molecules, which is necessary to ensure that the first step of the planner (i.e. before performing any of the available experiments) is correct. We use the Python module `higher` in order to perform second-order optimization that is necessary to correctly train in both loops. Grefenstette et al.

By training in this way, we hope to obtain a model that, for unseen molecules, is able to adapt by knowing just a few properties and provide more accurate predictions on the rest of properties. However, this method has one major shortcoming: we cannot train different molecules in parallel. That is because, after the inner loop, the parameters θ_N^i will be different for each molecule, which requires us to store a different model for each molecule that can only receive that particular molecule as the input and nothing else. In classic MAML that might not be a problem, as we might only have a few tasks, but in transposed MAML we have

tens of thousands of molecules and as such this method is far too slow.

3.6 Transposed CNGrad

In order to train on several molecules in parallel, we use the meta-learning version of the CNGrad method described in Alet et al. Instead of updating all the parameters θ in the inner loop and the outer loop, we add a Conditional Normalization (CN) layer in the architecture, which consists of a Batch Normalization (BN) layer followed by an affine layer (parametrized by γ, β 2 R^F, where F is the number of hidden features of the input of the CN layer). In our model, we add it before the second fully connected layer of the MLP in the readout phase. The γ and β parameters will be different for different molecules, and in the inner loop we initialize them to 1 and 0 respectively for all molecules. In the inner loop we only update γ and β, while in the outer loop we update the rest of the parameters θ^0. This means that the majority of the neural network will be the same for all molecules, and all the difference between the molecules will be captured in the CN layer. Since the BN layer treats each molecule separately, and the affine layer only performs element-wise operations, it is possible to compute the inner loop update of several molecules in parallel by just making the shape of γ and β equal to B F_h, where B is the meta-training batch size and F_h is the number of hidden features in each molecule. This means that we can also compute the outer loop update in parallel, which lets us greatly speed up the meta-training algorithm without losing much expressiveness. We found experimentally that the performance of the model is

better if we remove the BN layer and only add the affine layer, so our architecture only includes the latter.

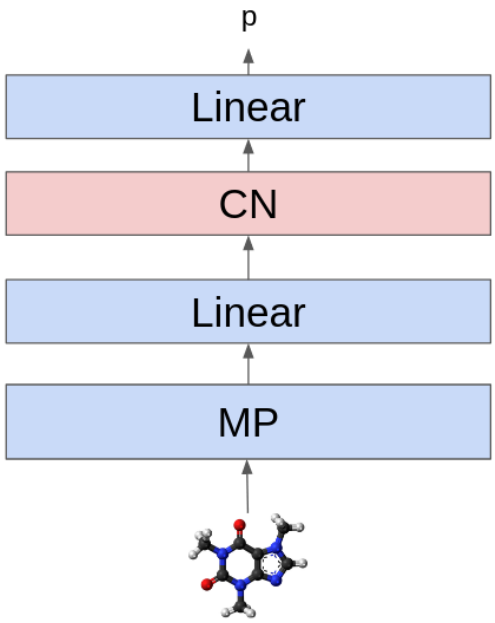


Figure 3.3 Architecture of the CNGrad model. In the inner loop, the output p is compared to the training tasks and only the parameters in red are updated. In the outer loop, the output p is compared to the validation tasks and only the parameters in blue are updated. The CN layer is reset every time we receive a new input.

3.7 Conservation loss

We know that $P(k_i = 1) = P(k_i = 1 \boldsymbol{j} k_j = 0)P(k_j = 0) + P(k_i = 1 \boldsymbol{j} k_j = 1)P(k_j = 1)$. Define the left hand side as the *prior probability of* k_i and the right hand side as *posterior probability of* k_i. We do not train the model to fulfill this equality in any the training frameworks we have discussed, which might be problematic during the planning phase. If the posterior probability of the goal experiment after performing experiment j is much greater than the prior probability according to the model, the planner might decide to perform experiment j to "increase" the

odds of success in the goal experiment even if experiment j might give us no useful information.

To correct this, we add a second term to the meta-loss in the outer loop which we call the *conservation loss*. For each molecule T_i, in addition to $D^{tr}_{T_i}$ we create two new datasets $D^{tr:0}_{T_i}$ and $D^{tr:1}_{T_i}$ by choosing a random property j (which might not be in the dataset for that molecule) by adding the input/output pair $(j, 0)$ to the first new dataset and $(j, 1)$ to the second. We compute the inner loop three times, one for each of the three datasets for that molecule to obtain θ^i_N, $\theta^{i:0}_N$ and $\theta^{i:1}_N$. The meta-loss will be computed using θ^i_N as usual, but we compute the predictions p, p^0 and p^1 using the validation dataset $D^{tr}_{T_i}$ with all three resulting models. The conservation loss will be the Kullback-Leibler divergence between p (the prior probabilities) and $(1 \quad p_j)p^0 + p_j p^1$ (the posterior probabilities). This loss is multiplied by a constant c, determined experimentally, before being added to the meta-loss.

This additional loss function does not guarantee that the prior and posterior probabilities of the model will match exactly, but it will make them much closer than if we did not use the conservation loss.

Chapter 4

Planning

4.1 Overview

In the planning phase, we will receive M (molecule), $c \in \mathbb{R}^n$ (vector containing the cost of each experiment, with n being the number fo available experiments), $g \in \mathbb{N}$ (the goal experiment) and $R \in \mathbb{R}$ (the reward if $k_g = 1$) as an input. We can use any of the models we discussed in Chapter 3 to make predictions on M, additionally using $k \in \mathbb{R}^n$ (vector containing the outcomes of experiments we already performed) as an input for all models except the multitask model.

To obtain the policy that maximizes E (the expected value), we will use tree search. Each node represents a possible value of k, and the root corresponds to the starting k (with $k_i = \quad 1/8k$, representing that we do not know the outcome of any experiments yet), and each node has two children $k^{i:0}$ and $k^{i:1}$ for each available experiment i at that node, the former with $k_i = 0$ and the latter with $k_i = 1$ (in both cases, the rest of k is the same as the parent node). Each node has one

additional child corresponding to stopping. The stop nodes and the nodes where $k_g \neq 1$ are terminal nodes, and we can compute the payout $R(k)$ of those nodes directly. Otherwise, we can compute the expected value of a node recursively using the predictions of the model p for that node as such:

$$E(k) = \max\left(R(k), \max_i (1 \quad p_i) E(k^{i:0}) + p_i E(k^{i:1})\right)$$

$R(k)$ would correspond to stopping at that node. Then, the policy at each node k would be to choose the action (performing one of the experiments or stopping) with the highest expected value.

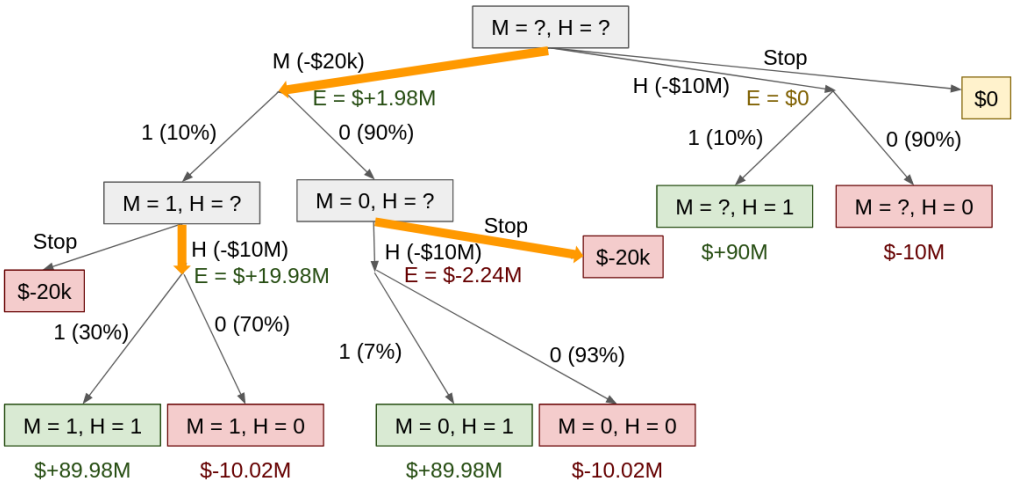


Figure 4.1 An example of tree search. In this simplified scenario we only have 2 experiments: M which costs $20,000 but gives no reward, and H which costs $10 million but gives a reward of $100 million if the experiment results in a 1. At each node we expand the possible actions and we compute the expected value of each action. The action resulting in the highest expected value will be our policy for that node, represented by the thick orange arrows.

To reduce the search space somewhat, we consider all the nodes with the same k as being one single node with multiple parents. We can do this because the order in which we perform the experiments leading to that k does not affect the

total cost we have to pay or the future predictions of the model and costs of the experiments. That still leaves us with 3^n possible nodes if there are n available experiments (k_i can be 1, 0 or 1), and in practice we cannot explore all of these nodes. For that reason, we need an algorithm to decide which nodes to expand first, and every time we expand a node we update the expected value of all the ancestors of that node. We will explore two algorithms: one using best-first search with several heuristics the other inspired by AlphaZero. Silver et al.

4.2 Best-first search

When we expand a node, the cost of calculating predictions using our model is the dominant factor in terms of computation time. Therefore, we must be able to identify which child nodes are the best to explore next without knowing what the predictions of the model look like in the child node. Despite not having access to those predictions, we have access to the following information that could be useful to decide which node to expand next:

- **Cost of the experiment.** The cost of the experiment gives us both an upper bound of the maximum expected value ($R_g \quad \sum_{k_i \neq 1} c_i$) and a rough indication of how significant an experiment needs to be to be worth performing. For example, if we could only perform one experiment, one whose cost is $\frac{1}{1000}$ of the goal experiment could be worth it if p_g changes by 0.01 after that experiment, while the same is not true for an experiment whose cost is $\frac{1}{10}$ of the goal experiment. While this is not entirely true in our actual prob-

lem because non-goal experiments also give us information on other non-goal experiments we can perform, it is still an important consideration.

- **Probabilities of the current node.** An experiment i with p_i close to $\frac{1}{2}$ has much higher entropy than one with p_i close to 0 or 1, which roughly means that the former gives us more information than the latter. Furthermore, performing a non-goal experiment is only useful if its outcome would change whether you perform the goal experiment in the end or not. Therefore, an experiment with p_i close to $\frac{1}{2}$ gives you a sizeable chance of not performing the goal experiment in a case where you otherwise would have but it would have been suboptimal, greatly increasing your expected value. As p_i gets closer to 0 or to 1, the chance of "changing your mind" decreases, making the increase in expected value lower. Similarly, if p_g is close to $\frac{1}{2}$ it means there is potential to increase or decrease that probability by performing additional experiments, while a p_g close to 0 or 1 means that the model does not really need additional information and that you can stop or perform the goal experiment, respectively.

- **Changes in probabilities between the current node and its parent.** Measuring the difference between the current node's p and the parent node's p gives us an idea of how useful the experiment performed by the parent node is. If the difference is negligible, it means that the experiment we performed did not give us much information for its price. Despite having wasted some computing time on calculating the predictions after a fruitless

experiment, we can avoid expanding further from that node and instead explore other nodes with more impactful experiments. This metric fails in some corner cases: for example, consider a scenario with 3 experiments where $t_3 = t_1 \quad t_2$, $g = 3$, with e_1 and e_2 being much cheaper than e_3. In the case where $t_1 = t_2 = \frac{1}{2}$, this metric would tell us that it is not worth it to perform e_2 after e_1 despite the fact that doing so would determine the result of e_3. Additionally, since computing the difference between p can be expensive, we will only calculate the difference between p_g instead.

With these three pieces of information at our disposal, we propose 6 different heuristics that emphasize different factors:

- **I. Total cost.** Here we just explore all the nodes in order from lowest total cost of the experiments to highest total cost, not including the cost of the goal experiment. We already discussed that the total cost gives us an upper bound of the expected value, so in cases where we can find a cheap policy with a high expected value early, we will be able to immediately prune all the nodes where $R \quad \sum_{k_i \in} {}_1 c_i$ is lower than the expected value we obtained. This advantage does not apply in cases where there is not a policy with a high expected value, perhaps because p_g was low to begin with.

- **II. Upper bound of the expected value after one experiment.** We can incorporate information from p_i and p_g to calculate a better upper bound of the expected value if we were only allowed to perform one experiment before deciding whether to perform the final experiment or stopping. The

best case will be when $p_g^{i:0}$ is as low as possible and $p_g^{i:1}$ is as high as possible, or the other way around, since the experiment will only be useful if we stop in the case where p_g is lowest and perform the goal experiment in the case where p_g is highest. We can calculate $p_{max}^{i:0} = \min\{1, \frac{p_g}{p_0}\}$ and $p_{max}^{i:1} = \min\{1, \frac{p_g}{p_1}\}$, the upper bounds of p_g if $t_i = 0$ or $t_i = 1$ respectively, and then calculate the expected values $E_j = p_j(R_g p_{max}^{i:j} \quad c_g) \quad \sum_{k_i \in \quad 1} c_i$ and take the maximum to get our upper bound. While this upper bound will not be accurate in the real planner because we are allowed to perform more than one experiment, it gives us a good indication of what experiments can be most relevant.

- **III. Maximize information to cost ratio.** With this heuristic we are interested in performing experiments that can give us the most information at the lowest price. We define information in terms of bits of entropy, and the entropy of a Bernoulli distribution with a probability p_i of having value 1 and a probability $1 \quad p_i$ of having value 0 is $S(p_i) = \quad p_i \log p_i \quad (1 \quad p_i) \log 1 \quad p_i$. Since both the entropy of the goal experiment and the auxiliary experiment are important, we expand the node with the highest $S = \frac{S(p_i)S(p_g)}{c_i}$ first.

- **IV. Maximize information to cost ratio, weighted by probability** We can also compute the probability P of being in a certain node after performing all the experiments you need to get there. Since gaining information is most useful in nodes with the highest probability, we expand the node with the highest SP.

- **V. Maximize information to cost ratio, weighted by relevance** We

can also compute $\Delta = (p_g^{\text{current node}} \quad p_g^{\text{parent node}}) p_{k_i=t_i}^{\text{parent node}}$, quantifying the "impact" of the previous experiment performed. We multiply by the probability of the previous experiment leading to the current node so that both children of the parent node have the same Δ regardless of p_i in the parent node. We expand the node with the highest $S\Delta$.

- **VI. Combine the previous two ideas.** We expand the node with the highest $SP\Delta$.

To further prune the search space, we do not expand a node if the probability P_k of reaching that node is less than p_{node_min} and we do not try an experiment if $p_i < p_{exp\_min}$ or $1 \quad p_i < p_{exp\_min}$. We do this to avoid wasting compute time on nodes that are far too unlikely to be reached or experiments with results so predictable that they are unlikely to provide useful information.

Additionally, since we are able to compute several molecules in parallel (or in this case, the same molecule but with different known properties), instead of expanding the node with the highest heuristic, we expand the B nodes with the highest heuristic all at once. When we visit a node, we immediately calculate the expected value of stopping and performing the goal experiment, and then we calculate the heuristic for all its remaining children. All of these heuristics are put in a priority queue so that, after all the B nodes have been expanded, we can choose an additional batch of B nodes with the best heuristic from the priority queue.

Since we are merging nodes with the same k_i but different paths to get there into one node, it is possible that a potential child node gets considered by multiple

parents and has a different heuristic for each one. In this case, we put the child node in the priority queue each time the calculated heuristic is better than the current best, and once we extract the node from the priority queue, any subsequent instances of that node in the priority queue will be ignored.

4.3 AlphaZero

The motivation behind this approach is that, instead of hand-crafting an algorithm to expand the tree most effectively, we train a neural network to do it for us

4.3.1 Policy and value network

In AlphaZero, a deep neural network is trained from self-play that takes a board position s as an input and returns a policy vector $q \in \mathbb{R}^a$ (where a represents the number of available actions in position s) and a scalar value $v \in \mathbb{R}$. The policy vector indicates the probability of the agent to perform each move, while the value indicates the probability of winning in that position. However, for our problem we will need a different architecture and definitions. Our neural network will take M, $c \in \mathbb{R}^n$, $k \in \mathbb{R}^n$, $g \in \mathbb{N}$ and $p \in \mathbb{R}^n$ as an input, representing the state of the current node, and it will return a policy vector $q \in \mathbb{R}^{n+1}$, with q_i indicating the probability of performing experiment i and q_{n+1} representing the probability of stopping, and a scalar value $v \in \mathbb{R}$ indicating the expected value of that node.

Since our input is more complex than the traditional board state, we will create a new architecture to calculate q and v. We start with the message passing

network discussed in Section 3.1 that will take M as an input and return the molecule hidden features $h \in \mathbb{R}^F$, but now it is not enough to feed them into a single MLP since we need to calculate two different variables now and we need to incorporate the rest of the information from the board state. What we will do is create an embedding for each available experiment using c, k, and p. We discretize c into C different buckets that are logarithmically spaced, and each of those buckets will be assigned a trainable embedding $h_c \in \mathbb{R}^{F_c}$. Each value of $k_i \in \{-1, 0, 1\}$ is also assigned a trainable embedding $h_k \in \mathbb{R}^{F_k}$. Finally, we assign each experiment a unique trainable embedding $h_i \in \mathbb{R}^{F_e}$ and we concatenate those three embeddings along with p_i repeated F_p times to form a task embedding $H_i \in \mathbb{R}^{F_c+F_k+F_e+F_p}$. We calculate v by concatenating h and H_g and feeding it as an input to a 2-layer MLP followed by a hyperbolic tangent function, and we calculate q by concatenating h, H_g and H_i for each experiment e_i to form a full task embedding $\in \mathbb{R}^{F+2(F_c+F_k+F_e+F_p)}$ and we feed each of these task embeddings to a 2-layer MLP. We also assign a trainable embedding $H_s \in \mathbb{R}^{F_c+F_k+F_e+F_p}$ to the action of stopping and we also feed it to the 2-layer MLP, obtaining $n+1$ values. We apply the softmax function to these values to obtain the policy vector q.

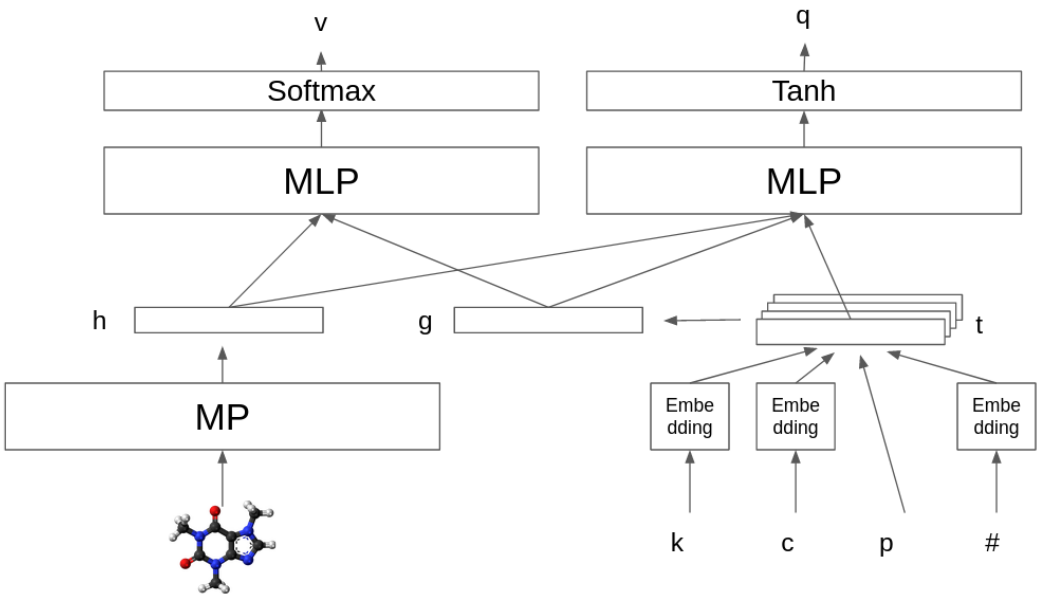


Figure 4.2 Policy and value neural network architecture for the molecule problem. We construct the representation of a task t_i using k_i, c_i, p_i and the index i, and we concatenate each t_i with h and g (the representation of the goal task) to obtain the task vector. We use h and g to calculate the value, and we use the task vectors to calculate the policy.

4.3.2 Monte Carlo Tree Search

The basis of self-play in the AlphaZero algorithm is Monte Carlo Tree Search (MCTS). In order to improve the neural network's predictions of the policy and value, we use those predictions as part of the MCTS algorithm in order to generate new games with better play than the neural network alone would be able to manage. Each game is initialized with the initial board state and a blank MCTS tree. To decide what action to play from each position, we perform a fixed number N of MCTS iterations. Each MCTS iteration generates a path in the MCTS tree that starts with the node containing the current board and ends either in a terminal state or in a node we never visited before. In the first case, we can calculate the outcome of the game directly using the rules of the game. In the second case, we

call the neural network to return the policy and value for that node. In both cases, the value of the last visited node in the path (calculated from a terminal state or from the neural network) is used to update the estimate of the actual value of all the nodes in that path.

At each node s, MCTS chooses an action a according to two terms: Q, the exploitation factor, and U, the exploration factor. Let $N(s,a)$ be the number of times that action a has been selected at node s, and $q(s,a)$ the probability of selecting action a at node s according to the policy vector q. Then, $U(s,a)$ is calculated as follows:

$$U(s,a) = c_{puct}\frac{q(s,a)}{1+N(s,a)}$$

where c_{puct} is a constant that is determined experimentally and controls the rate of exploration of the agent. A higher c_{puct} increases exploration while a lower c_{puct} increases exploitation. U gives a higher priority to actions that have been explored less often than the policy would otherwise suggest.

On the other hand, $Q(s,a)$ is equal to the average value of the iterations that chose action a at node s, or 0 if MCTS has never chosen that action at that node. The value of an iteration is that of its leaf node, and the value of that node is either the outcome of the game for a terminal node, or the neural network prediction for non-terminal nodes. Q gives a higher priority to actions that we already know result in the highest chance of winning.

At each node, MCTS chooses the action with the highest $Q+U$. After all the N iterations are complete, MCTS will choose the action a at the root with the highest visit count $N(s,a)$, the chosen move is played, the turn passes to the opponent,

and the whole process is repeated again to decide the next move. The MCTS tree used to decide what move to play is maintained after each move, leading to more accurate information in the future without having to perform additional MCTS iterations. Once the game is finished, the moves that were actually played in the game get used as training samples for that epoch. In those training samples, the target policy at node s will be determined by the relative $N(s, a)$ counts while the target value will be the final outcome of the game.

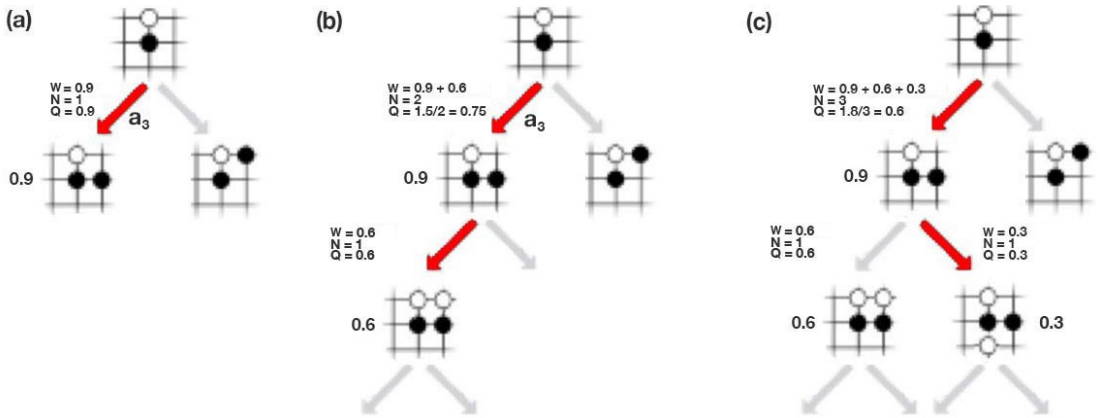


Figure 4.3 An example of three consecutive MCTS iterations. In iteration (a), we explore the node on the left, which we have never seen before, and compute the value which is 0.9. Therefore, the root's Q is updated to 0.9. In iteration (b), we explore the node on the left again, so we continue and hit a node we have never seen before. Its value is 0.6 which means all the nodes on its path get their Q updated accordingly. Iteration (c) is analogous to (b). Note that after each iteration, the Q of the root is the average of all the values computed at the leaves. Image taken from "Monte Carlo Tree Search (MCTS) in AlphaGo Zero".

AlphaZero is generally used in two-player games where both players are competing against one another, and the neural network plays actions for both players. After choosing an action in a node, the resulting child node will be the opponent's move, and as such MCTS will choose the best move for the opponent, and the current player alternates on each subsequent node. However, this formulation does

not work for our problem because there is only one agent trying to maximize the expected value. We can consider that the other player in the "game" is nature deciding which outcome each experiment has, but this other player is not trying to win or anything. It instead can be considered as being purely random. Therefore, it is not correct to use the MCTS algorithm outlined in this section without any modifications, as it assumes that both players are playing to win.

Instead, what we do is to only have nodes in the tree for the first player (the agent trying to maximize the expected value) in the same way as the heuristic tree search, with each node representing a possible vector of known properties k. Whenever MCTS chooses to perform an experiment i, we use the CNGrad model discussed in Section 3.6 to calculate the predicted probability of $t_i = 1$ and we use that probability to randomly choose a value for k_i. The only exception is if we perform the goal experiment, as there will be no further nodes to explore. In that case, we use p_g to calculate the expected value of the node and back up that value. Since we are trying to train the planner to generate the policy to the highest expected value, we never give the neural network access to the actual outcomes of the experiments. Instead, we simulate them using the CNGrad model since that is the closest equivalent to planning with heuristics using the predictions of the CNGrad model, which is what we did in the previous section.

Unlike in the heuristic sample, in the MCTS version the cost of making predictions using the neural network is vastly dominated by the cost of making MCTS iterations. However, these iterations are easy to parallelize using multiple CPUs, as each iteration will only contribute to either 0 or 1 inputs to the neural network

and we can batch those inputs. What we do is play many games concurrently on each CPU. We do one iteration for each game, figure out which queries we need to make to the neural network for each iteration, make a batch with all the queries and feed them to the neural network. By using many CPUs in parallel, we can generate many games to use as training samples using only a single GPU.

Chapter 5

Experiments

5.1 Dataset

In the ChEMBL20 dataset, we have 456903 molecules and 902 tasks that are known for at least 128 molecules in the dataset. In order to perform meta-learning, we will randomly split the tasks into training and validation, and we will randomly split the molecules into meta-training, meta-validation and planning. For a molecule to stay in our final dataset, we require knowing at least 3 training properties and 1 validation property. With these restrictions, we are left with 113072 meta-training molecules, 37846 meta-validation molecules and 37288 planning molecules.

The meta-training and meta-validation molecules will be used in the training stage, while the planning molecules will be used to test the performance of the planner. We will also reuse the meta-validation set to train the AlphaZero planner. We can do this as none of our meta-learning formulations actually use the meta-validation set to update the model, so we can use the information to train

something else while still having a fresh dataset (the planning set) on which to test the performance of the planner.

5.2 Model training

The learning curves for each of the models we trained are shown in Figures 5.1, 5.2 and 5.3 respectively. We did not train the MAML model since, as we discussed, it is very slow to train due to its inability to batch several molecules. Here it is worth noting that there is a noticeable gap between training and validation in the multitask with extended input model that does not exist in the other two models, despite the accuracy being higher than the multitask model. This might be because the extended input model receives more information during training than the bare multitask model, but is not able to learn from it as well as the CNGrad model, which does not have such a pronounced gap. However, we notice that in the CNGrad model, the training accuracies are lower than the validation accuracies. This is because the model has no additional information to use to predict the training properties, but when it is time to predict the validation properties, it has already learned from the training properties, leading to better predictions.

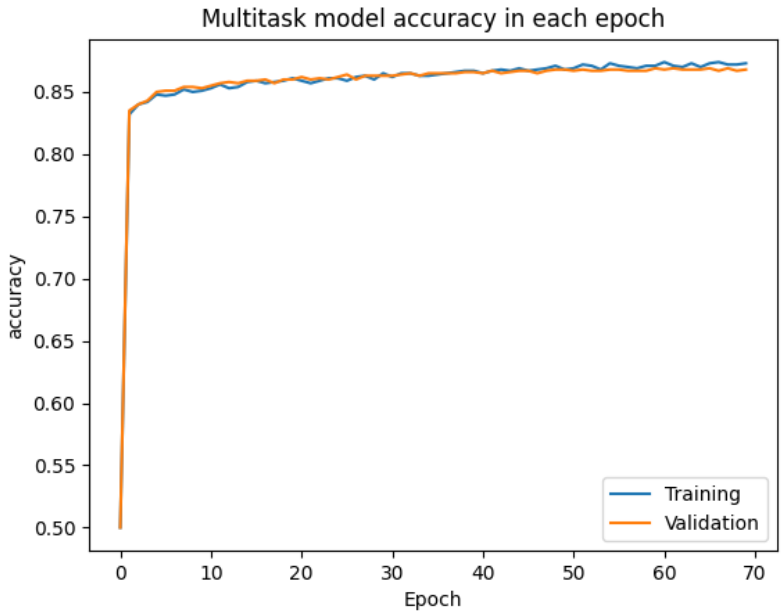


Figure 5.1 Accuracy over time of the multitask model.

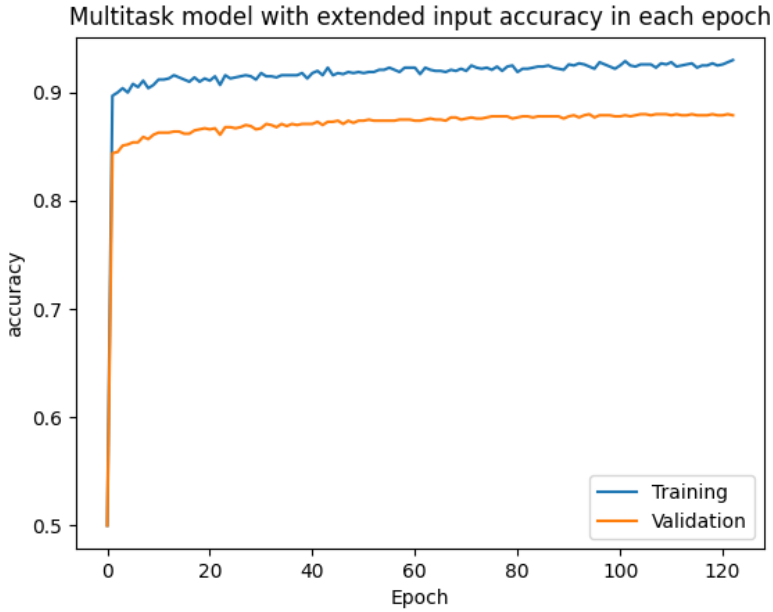


Figure 5.2 Accuracy over time of the multitask with extended input model.

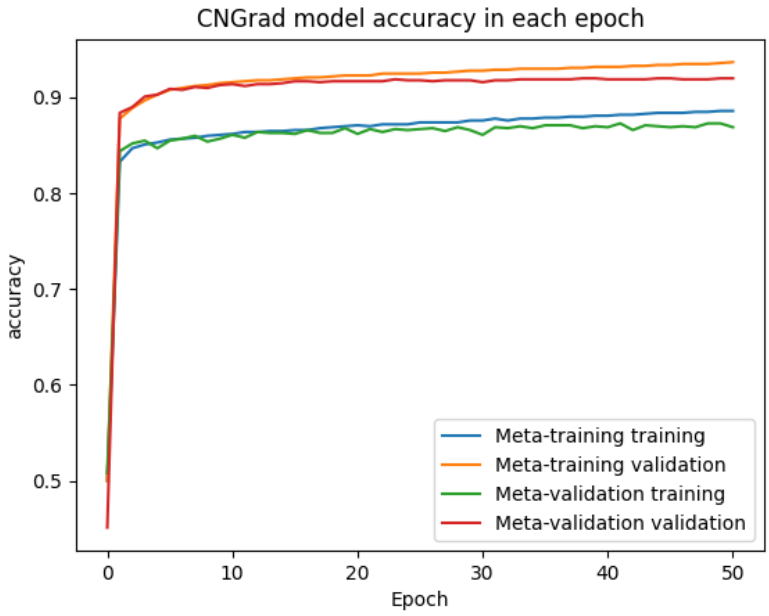


Figure 5.3 Accuracy over time of the CNGrad model.

The (meta-)validation accuracy of each of the models we trained is shown in Table 5.1. Comparing multitask to multitask with extended input, the latter performs slightly worse if we give it no known properties, but its accuracy jumps over that of the multitask model as soon as we incorporate even one property. This means it has the potential to outperform the multitask model in the planner. However, the CNGrad model is the clear winner in terms of accuracy with any number of known properties in the training dataset, indicating that it is the best method we have tried to incorporate information from additional known properties. It may seem strange that CNGrad is able to outperform the multitask model even when they are given the exact same input with no known properties, but we must keep in mind that the meta-learning models (which include CNGrad) train using more data than the bare multitask model (in the form of additional known properties in the input) which means that it can learn some additional information

that the multitask model simply cannot infer.

Model	Accuracy		0 known		1 known		2 known	
Multitask	.868	.003						
Extended input	.878	.003	.860	.007	.875	.007	.879	.007
CNGrad	.920	.003	.909	.003	.915	.003	.919	.003

Model	3 known		4 known		5 known		6 known	
Extended input	.879	.007	.887	.007	.890	.007	.890	.007
CNGrad	.923	.003	.925	.003	.926	.003	.926	.003

Table 5.1: Validation (or meta-validation validation) accuracies for the models we trained. In models that incorporate meta-learning, we also include the accuracy when the training dataset contains a certain number of samples, from 0 to 6. The 95% confidence intervals are also shown.

5.3 Planning

In order to test the performance of the planner over the planning molecules, we need to assign a cost to each experiment we can perform on the molecule. However, this information is not possible for us to find, as pharmaceutical companies typically keep that information hidden. Mercado Therefore, we will assign costs to the experiments arbitrarily, but keeping in mind that the training properties are the properties that are easy to measure and thus cheap, while the validation properties are the ones we are interested in generalizing to because they are hard to measure and thus expensive.

In order to test the generated policy with real data, we will restrict the set of experiments we can perform only to those that are present in the dataset for that molecule. We number those experiments from 0 to n 1, with the training properties taking the lowest numbers and the validation properties taking the highest numbers. The goal property will always be n 1. Note that this goal

property will be different from molecule to molecule, as different molecules have different known properties in the dataset. We choose parameters R, the reward for performing the goal experiment and getting 1 as a result, and c_{max}, the cost of the most expensive experiment (which will be the goal experiment). In our tests, $R = 2000$ and $c_{max} = 1000$. These parameters make the goal experiment worth it if $p_g > \frac{1}{2}$ and not worth it otherwise, and therefore checking whether the planner has made the right decision can be seen as an equivalent to calculating the accuracy. Experiment i will be assigned a cost $c_i = c_m a x^{\frac{i}{n-1}}$. This means that training properties will cost less to discover than validation properties, which is the cost distribution we wanted.

For the heuristic-based planners, we will expand 8 batches of nodes, with each batch containing up to 128 nodes, giving us the possibility to explore a total of 1024 nodes. Given that the full tree contains 3^n nodes, and there can be over $n = 100$ available experiments for a molecule, the nodes we are able to explore will only be a tiny minority of the total amount, which will test the heuristics' capability to explore the most promising nodes quickly. To prune the tree further, we choose $p_{node\_min} = 0.001$ and $p_{exp\_min} = 0.02$, which prevents us from expanding nodes that are too unlikely to be reached or experiments that are too unlikely to give us new information, respectively. In the AlphaZero-inspired model, we will perform 250 MCTS iterations for each action.

With those parameters, we use a planner and a trained model to obtain a policy that tells us which action to take first, and which actions to take next depending on the results of previous experiments. Once we have the policy, we

can calculate what the actual result of applying that policy would be by using the real data in the ChEMBL20 dataset to simulate what would happen at each step, and find out the actual payout of that policy. The results can be found in Table 5.2. Additionally, since we are updating the heuristics-based planner live every time we explore a new node, we can calculate what the policy, and therefore expected value and payout, at each time step would be. This information is shown for the CNGrad model in Figures 5.4 and 5.5, respectively.

Model	No planning		Heuristic I		Heuristic II		Heuristic III	
Multitask	260.16							
Extended input	258.07	2.60	265.68	2.57	263.38	2.53	264.49	2.53
CNGrad	261.77	2.42	269.39	2.42	269.01	2.41	268.86	2.41
Always	-283.42	6.29						

Model	Heuristic IV		Heuristic V		Heuristic VI		AlphaZero	
Extended input	264.74	2.53	264.02	2.54	264.38	2.53		
CNGrad	268.85	2.41	268.92	2.41	268.86	2.41	262.49	5.37

Table 5.2: Average payout per molecule of the planner for each combination of predicting model and planning algorithm when $R = 2000$ and $c_{max} = 1000$. The row labeled "Always" represents the average payout if our policy is to always perform the goal experiment. The column labeled "No planning" represents the average payout of only using the model to decide whether to perform the goal experiment or stop (we are not allowed to perform other experiments in this case). The 95% confidence intervals to test the hypothesis that the performance is different from the bare multitask model is shown.

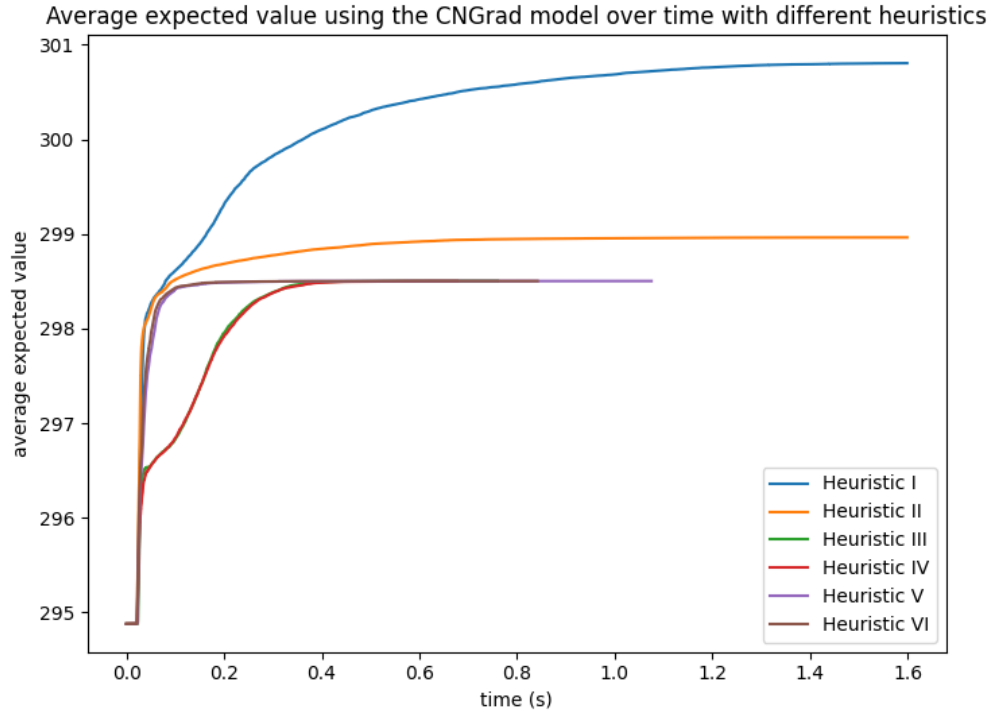


Figure 5.4 Average expected value per molecule as a function of how long the planner is allowed to run for the different heuristics using the CNGrad model.

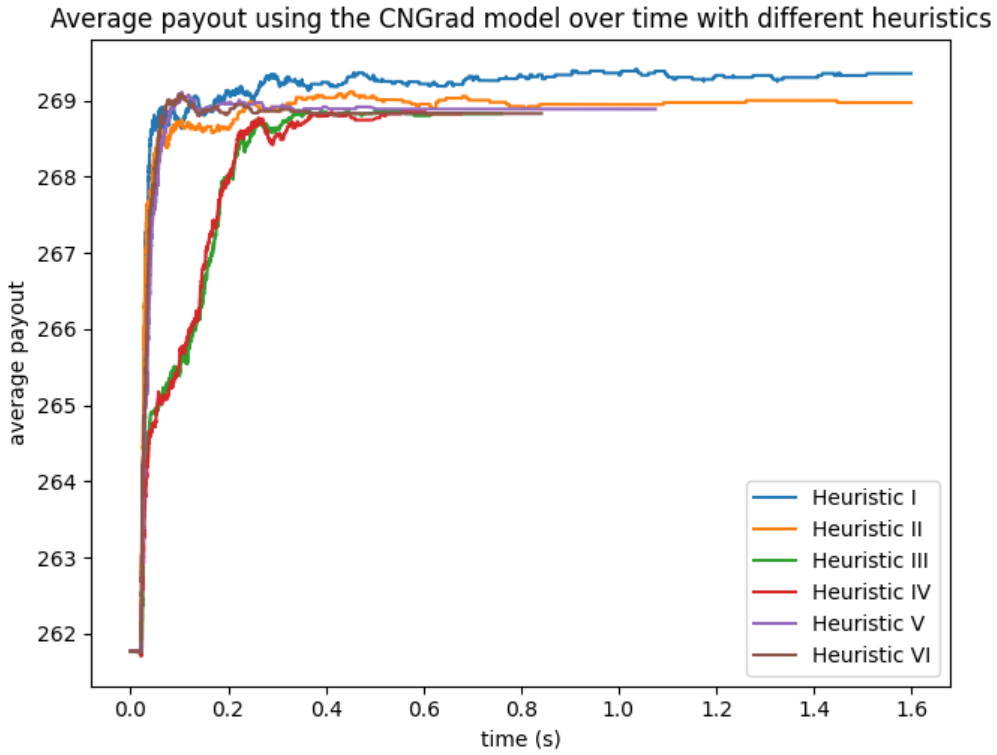


Figure 5.5 Average payout per molecule over as a function of how long the planner is allowed to run for the different heuristics using the CNGrad model.

Just as we expected from the validation accuracies with 0 known properties, extended input without planning performs slightly worse than multitask, but CNGrad without planning performs slightly better, although neither of these are statistically significant results. What is statistically significant, however, is that the meta-learning approaches we devised outperform the multitask model by incorporating information from the experiments we perform. Both methods outperform the multitask model, but CNGrad outperforms extended input by a statistically significant margin. One thing that is important to keep in mind is that most molecules will already be predicted with a high confidence by the model and as such the planner will not help. The planner only helps in the minority of molecules

for which the model does not have such a clear prediction for the molecule and additional information might be useful.

Comparing different planning methods instead of predictive models yields results that are perhaps a bit more surprising. In particular, heuristic I (always expand the nodes with the lowest total cost first) outperforms the other, more intricate heuristics. The difference is not statistically significant, but it can be noticed by looking at the graphs. There are two reasons why heuristic I might be the best. The first reason is that it is the only heuristic that only uses information that we 100% know is true: the costs of the experiments. The probabilities of each outcome are just predictions by the model and there is a chance that they are wrong, which would cause the heuristic to expand the wrong nodes and hurt performance. Therefore, this might indicate that in planning problems like these it is better to only rely on information that must be true instead of relying on predictions that might not always be accurate. The second reason is that it is the only heuristic that takes into account the entire path it took to get to the current node. For instance, in heuristics III and IV we only look at the entropy of the experiments in the current node, and the expected value for these two approaches rises substantially slower in the early stages of planning, as evidenced by Figure 5.4. This might be because these approaches might spend a lot of time expanding the same experiment in many different nodes that might have high entropy but not be relevant. Including the Δ term from heuristics V and VI seems to address this problem, as the expected value rises as quickly as heuristics I and II in the early stages. Furthermore, given that heuristics III and IV perform very similarly,

as well as V and VI, we can deduce that taking into account or not the probability of being in the current node does not impact the performance of the planner.

The results of the planning stage also highlight the importance of the conservation loss, discussed in Section 3.7. When we tried to generate policies with a model that was not trained with conservation loss, we would frequently get policies like the one shown in 5.6. If the model predicts that the posterior probability is higher than the prior, it will choose to perform additional experiments in order to exploit the increased posterior probability to increase the expected value. The greater the difference between the posterior and the prior, the more expensive the experiments it might prescribe will be, and the more complex the erroneous subtrees can become. To test the effectiveness of the conservation loss, we took a sample of 250 planning molecules and took note of all the differences between the posterior and the prior (called Δ) and their magnitudes. The results are shown in Figure 5.7. While it is not possible to perfectly train the model to always output perfectly consistent predictions, we show that it is possible to greatly reduce the magnitude of the differences, which means that it is much harder for the planner to find an experiment that can "exploit" this difference. It is worth noting that absurd policies like the one shown in 5.6 still happen when using a model trained with a conservation loss, albeit much less frequently and involving fewer and cheaper experiments. To further mitigate the issue, after we generate the policy, we inspect all its subtrees and prune all the subtrees for which all of its leaves lead to the same final action (stopping or performing the goal experiment).

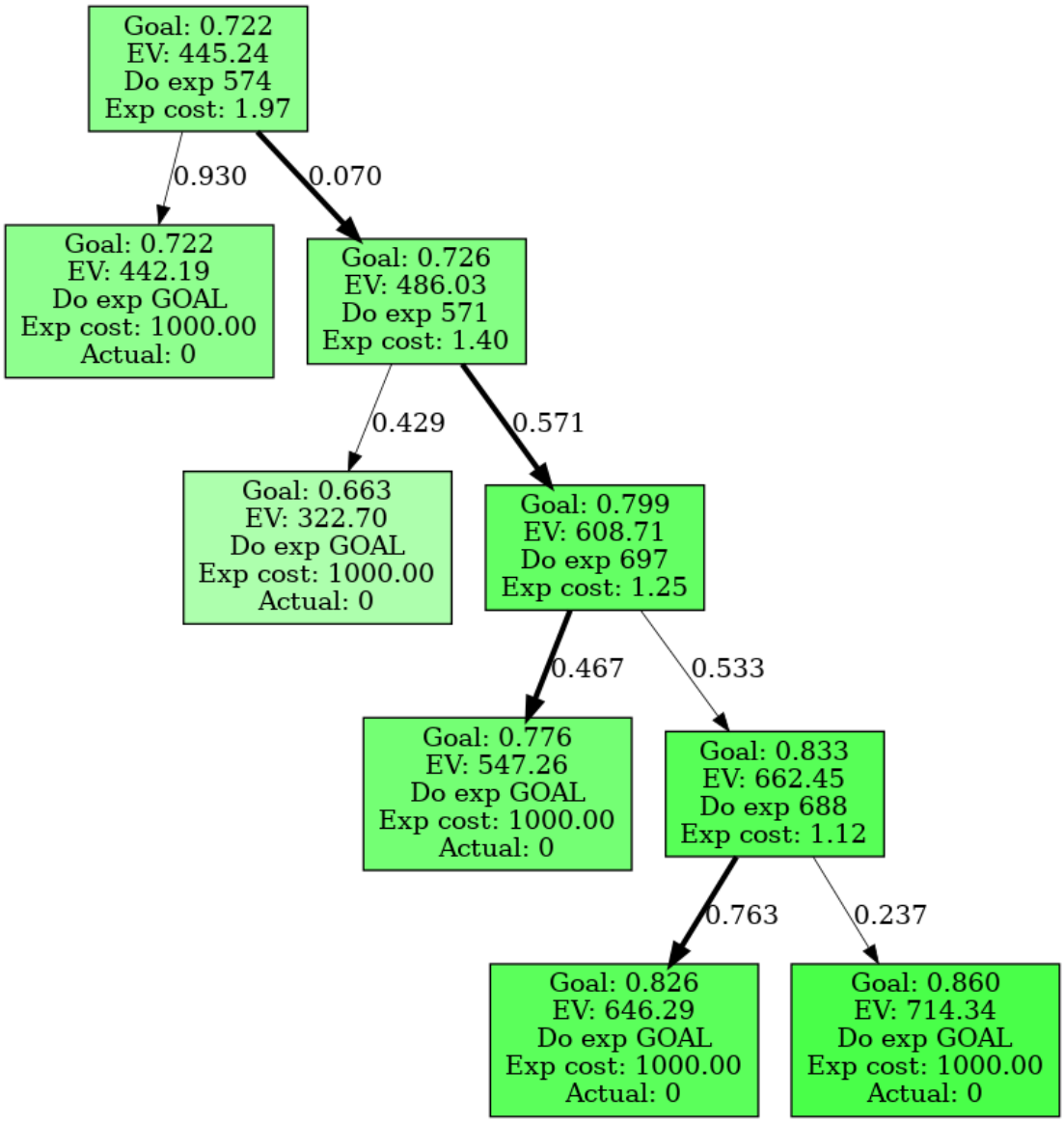
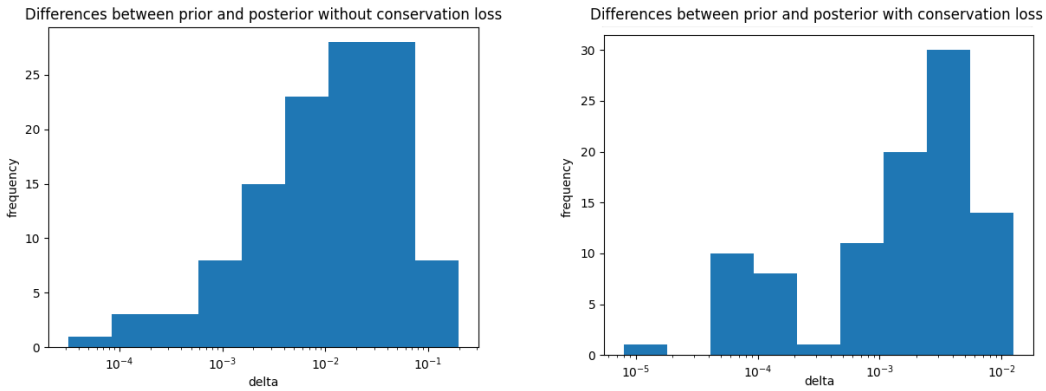


Figure 5.6 Example of a policy generated by the CNGrad model trained without a conservation loss term. The arrows represent the probabilities of landing on each child node after performing the experiment prescribed by the policy, and the bolded arrow represents the actual outcomes according to the dataset. This means that in reality we would perform a total of three unnecessary experiments, wasting 4.62 units of payout. Due to the posterior probabilities of the goal being higher than the prior probabilities, the policy tells us to perform additional experiments that according to the model will increase the expected value, but we know that cannot be true since we will perform the goal experiment no matter what outcomes we get. This means that the optimal policy would be to perform the goal experiment directly and avoid paying the cost of the other unnecessary experiments.

Training the neural network that predicts policy and value in AlphaZero is



(a) Differences between prior and posterior on 250 sample molecules using a model trained without a conservation loss.

(b) Differences between prior and posterior on 250 sample molecules using a model trained with a conservation loss.

Figure 5.7 By comparing the two histograms, we can deduce that, despite the amount of times we obtain an inconsistent prediction between prior and posterior is more or less the same, the magnitude of the difference is greatly reduced. Without a conservation loss, most differences fall in the 1-10% range which is quite substantial, as it can allow for unnecessary experiments totaling up to a tenth of the maximum reward. However, with a conservation loss, most differences fall in the 0.1-1% range, a full order of magnitude lower, which greatly limits the maximum cost, and therefore potential amount, of unnecessary experiments.

computationally expensive, and randomly selecting inputs will not be enough because, as we discussed earlier in this section, the CNGrad model will already have high-confidence predictions for the molecule and will not need planning. Therefore, we construct a special dataset from the meta-validation set that primarily contains molecules with properties for which the CNGrad model predicts a probability between 0.2 and 0.8. We also include 3% of the properties with predictions outside of this range so that the AlphaZero model can also learn when to directly perform the goal experiment or stop. The training curves of this method can be seen in Figures 5.8 and 5.9

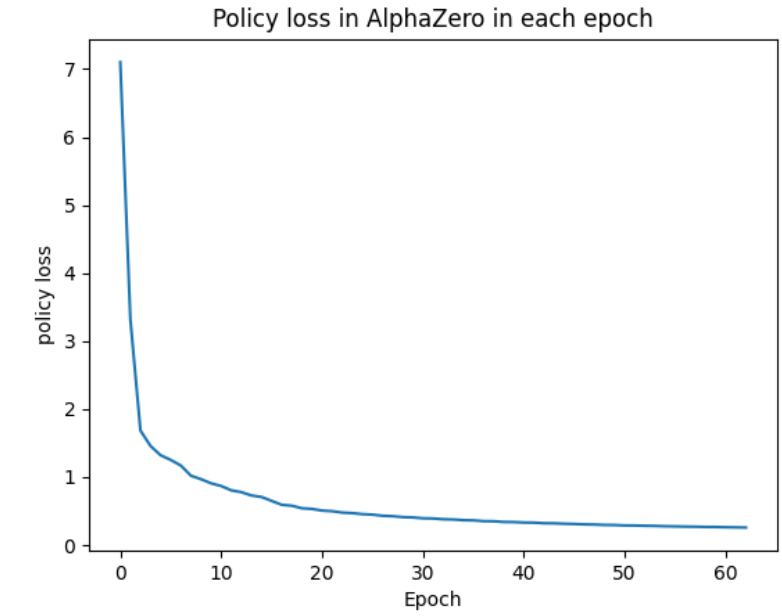


Figure 5.8 Policy loss of the AlphaZero neural network at each epoch

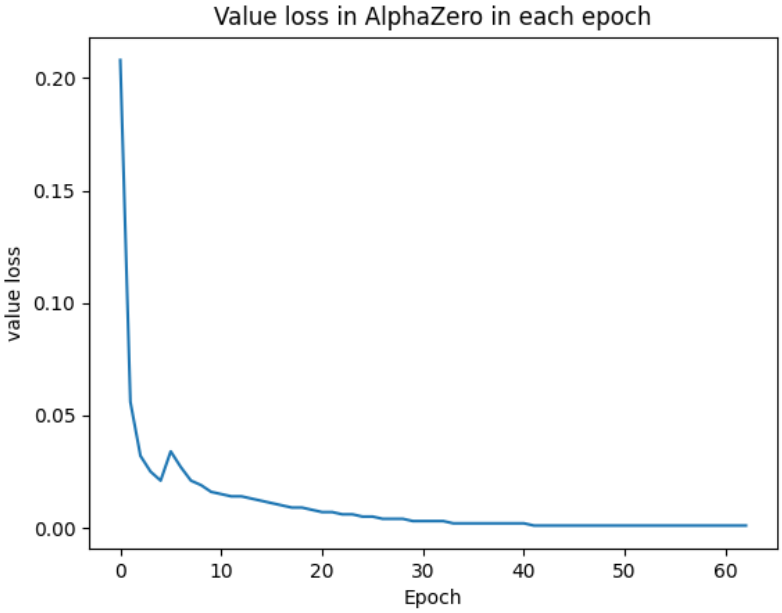


Figure 5.9 Value loss of the AlphaZero neural network at each epoch.

We observe that the model trains the value quite quickly, but that might be because it is easy to predict, as the CNGrad model predictions are part of the

input and it is easy to predict high values for high-probability goals and low values for low-probability goals. The loss curve for the policy trends downward too. However, the AlphaZero model fails to improve on the CNGrad model without planning or even the multitask model by a statistically significant margin.

Chapter 6

Conclusions

In this project, we posed a problem in which we have an entity on which we can perform several experiments, each one with a cost, and one of them, the *goal* experiment, will give us a large reward if successful, and our goal is to choose a policy of experiments to perform in order to maximize the expected value. We introduced the transposed meta-learning framework. In traditional meta-learning, for unseen tasks, we want to generalize to unseen inputs after seeing a few sample inputs, but in transposed meta-learning, we switch the roles of task and input so that for unseen inputs, we want to generalize to unseen tasks after seeing a few sample tasks.

We used previous research to propose two models that take advantage of meta-learning to learn from new information: one that is architecture based and one that is optimization based. We determined that the optimization based method obtained the best validation accuracy, and we used the CNGrad technique to be able to meta-learn several tasks (or, in the case of transposed meta-learning, inputs) in parallel and with the same expressiveness as meta-learning techniques that update the entire model.

We created a planner that would use the trained models to create policies for new, unseen molecules determining which experiments depending on the outcome of previous experiments. We used best-first search to explore the different combinations of experiments we can perform, and we tried several heuristics to decide which nodes to explore first. We discovered that, in this particular problem, the best heuristic is to simply visit the nodes in order from the lowest total cost of the experiments to the highest, and we concluded that the reason for that is that the total cost is the only piece of information that we have that is guaranteed to be accurate and not dependent on the predictions of a model that might be inaccurate for certain inputs. Additionally, the total cost captures information about the entire tree of possibilities instead of just the current node we are visiting or only the nodes in its vicinity.

We also tried to modify the AlphaZero algorithm to create a neural network that could learn what the best way to traverse the tree is instead of having to hard-code the heuristics ourselves, but the results were worse than with the handcrafted heuristics and we conclude that additional work is needed to create a model that learns the best way to traverse the tree.

Chapter 7

Future work

An important piece of information we are missing in our project are the real-world costs of performing each experiment. We have already discussed how pharmaceutical keep this information hidden, but there might be some other way to estimate the costs or at least simulate a more realistic distribution of costs. In any case, having this information would greatly help the project in having more real-world applicability.

One shortcoming of our work is that we have not been able to obtain satisfactory results with the AlphaZero model, but the results we obtained are at least somewhat promising because the model did learn enough to outperform the CNGrad model without planning, even if not by a statistically significant margin or to the same extent as the hard-coded best-first search did. Perhaps additional computation time is needed to make the approach work, or perhaps a different architecture or training method is needed to obtain better performance, but in any case it is clear that more work needs to be done in that area. There might also be a different method that is more suitable to automatically learn the best method of planning in this particular problem.

Lastly, the problem we posed can be applied to any type of entity from which we can obtain information at a certain cost, not just molecules and molecular properties. In particular, it would be interesting to apply the approaches we have devised on problems that require deeper planning, as we have already showed that knowing 6 properties for a molecule is about as much extra information as we will need, and it will most likely be less than that (for most inputs it is 0). Even a somewhat artificial problem might help test the limits of our approach. For example, consider the problem of image classification where we receive handwritten digits from the MNIST dataset and we must predict what digit that is, but the entire image is hidden from us and instead we must pay a certain cost to see each pixel. This problem would be solved in the same way as the problem with the molecules we formulated, but it would require deeper planning as knowing just a handful of pixels will likely not be enough to discern the digit in the image.

References

[1] Alet, F. et al. *Noether Networks: Meta-Learning Useful Conserved Quantities*. 2021. DOI: 10.48550/ARXIV.2112.03321. URL: https://arxiv.org/abs/2112.03321.

[2] Alet, F. et al. *Tailoring: encoding inductive biases by optimizing unsupervised objectives at prediction time*. 2020. DOI: 10.48550/ARXIV.2009.10623. URL: https://arxiv.org/abs/2009.10623.

[3] Bento, A. P. et al. "The ChEMBL bioactivity database: an update." In: *Nucleic Acids Research* (2014). ISSN: 0305-1048. DOI: 10.1093/nar/gkt1031.

[4] *Learning phrase representations using RNN encoder-decoder for statistical machine translation*. Conference on Empirical Methods in Natural Language Processing. 2014.

[5] Finn, C., Abbeel, P., and Levine, S. *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*. 2017. DOI: 10.48550/ARXIV.1703.03400. URL: https://arxiv.org/abs/1703.03400.

[6] Flennerhag, S. et al. *Meta-Learning with Warped Gradient Descent*. 2019. DOI: 10.48550/ARXIV.1909.00025. URL: https://arxiv.org/abs/1909.00025.

[7] Grefenstette, E. et al. *Generalized Inner Loop Meta-Learning*. 2019. DOI: 10.48550/ARXIV.1910.01727. URL: https://arxiv.org/abs/1910.01727.

[8] Li, Y. et al. *Gated Graph Sequence Neural Networks*. 2015. DOI: 10.48550/ARXIV.1511.05493. URL: https://arxiv.org/abs/1511.05493.

[9] Mercado, R. Personal communication. 2022.

[10] *Monte Carlo Tree Search (MCTS) in AlphaGo Zero*. 2018. URL: https://jonathan-hui.medium.com/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a.

[11] Nguyen, C. Q., Kreatsoulas, C., and Branson, K. M. *Meta-Learning GNN Initializations for Low-Resource Molecular Property Prediction*. 2020. DOI: 10.48550/ARXIV.2003.05996. URL: https://arxiv.org/abs/2003.05996.

[12] Ozair, S. et al. "Vector Quantized Models for Planning". In: (2021). URL: http://proceedings.mlr.press/v139/ozair21a/ozair21a.pdf.

[13] Pu, Y., Kaelbling, L. P., and Solar-Lezama, A. *Learning to Acquire Information*. 2017. DOI: 10.48550/ARXIV.1704.06131. URL: https://arxiv.org/abs/1704.06131.

[14] Schrittwieser, J. et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%5C%2Fs41586-020-03051-4.

[15] Silver, D. et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815.

[16] Sun, Y. et al. *Test-Time Training with Self-Supervision for Generalization under Distribution Shifts*. 2019. DOI: 10.48550/ARXIV.1909.13231. URL: https://arxiv.org/abs/1909.13231.

[17] Thrun, S. and Pratt, L. *Learning to Learn*. Springer US, 1998. ISBN: 978-0-7923-8047-4. DOI: 0.1007/978-1-4615-5529-2.

[18] Triantafillou, E. et al. "Meta-Dataset: A Dataset of Datasets for Learning to Learn from Few Examples". In: (2019). DOI: 10.48550/ARXIV.1903.03096. URL: https://arxiv.org/abs/1903.03096.

[19] Vanschoren, J. *Meta-Learning: A Survey*. 2018. DOI: 10.48550/ARXIV.1810.03548. URL: https://arxiv.org/abs/1810.03548.

[20] Vilalta, R. and Drissi, Y. "A Perspective View and Survey of Meta-Learning". In: *Artificial Intelligence Review* 18.1 (2002), pp. 77–95. ISSN: 1573-7462. DOI: 0.1023/A:1019956318069.

[21] Wu, G., Say, B., and Sanner, S. *Scalable Planning with Deep Neural Network Learned Transition Models*. 2019. DOI: 10.48550/ARXIV.1904.02873. URL: https://arxiv.org/abs/1904.02873.

[22] Ye, W. et al. *Mastering Atari Games with Limited Data*. 2021. DOI: 10.48550/ARXIV.2111.00210. URL: https://arxiv.org/abs/2111.00210.