Cost-Efficient Hierarchical Knowledge Extraction with Deep Reinforcement Learning

Jaromír Janisch, Tomáš Pevný and Viliam Lisý
Artificial Intelligence Center, Department of Computer Science
Faculty of Electrical Engineering, Czech Technical University in Prague
{jaromir.janisch, tomas.pevny, viliam.lisy}@fel.cvut.cz

Abstract

We extend the framework of Classification with Costly Features to work with *structured* samples, that can no longer be represented as fixed-length vectors. Instead, the samples can only be represented as *trees of features*, with a variable and possibly unlimited depth and breadth, similar to a JSON file. We provide a method that, independently for each sample, sequentially selects features from the tree. The newly acquired features can be further expanded, until the algorithm terminates with a classification decision. Each piece of information has a real-valued cost, and the objective is to maximize the classification accuracy while minimizing the total cost of the selected features. The method targets data naturally occurring in many domains, e.g., targeted advertising, medical diagnosis, or malware detection. We demonstrate our deep reinforcement learning based algorithm in seven relational classification datasets.

Introduction

The motivation of this paper arises from real-world cases, where the structured data, available in form of trees, is key to optimally solving a problem. The goal is to sequentially select the features to maximize the classification accuracy, while minimizing the total cost of the selected features. Crucially, because the data is in the form of variable trees, the prior-art algorithms (Janisch, Pevný, and Lisý 2019a,b; Shim, Hwang, and Yang 2018), designed only to work with fixed-length vectors, cannot be used.

There are three possible approaches to the problem. *First*, it is possible to manually create a fixed set of features from the structured data. This step is laborious, suboptimal, and in many cases, very difficult to apply, because the structure of the data varies between individual samples. Yet, in practice, this approach is prevalent. *Second*, it may be possible to treat all features in the tree as tuples *(path, value)* and use set-based algorithms, e.g., Shim, Hwang, and Yang (2018). However, substantial issues remain. For example, it is unclear how to encode the a-priori unknown set of paths in possibly infinite trees. Moreover, set-based algorithms typically assume that all features have the same dimension, which is too strict. *Third*, it is possible to process the data in their natural, tree-structured form and directly select the features in the tree. We focus on this last approach.

arXiv preprint v3, 9/2020

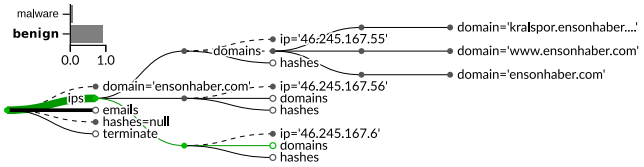


Figure 1: The key challenge is to identify the relevant features in the tree that are informative for the classification of this particular sample. Our method is to sequentially select features in the hierarchy, based on the already acquired information. The example comes from the *web* dataset, where the goal is to predict whether a given web domain is malicious. It illustrates a state in the process, with some acquired features (full nodes). The available actions are represented as empty nodes. Note that the breadth of the tree varies (e.g., different domains translate into different number of IP addresses) and the possible depth of the tree may be infinite. The algorithm gives weights to different features (visualized as line thickness) and select one of them (green line).

Below, we provide several examples of structured domains. In targeted advertising in social networks, the goal is to show an advertisement the user is likely to be interested in. The user is a part of a larger graph consisting of his friends, groups he joined, or applications he likes. The decision, which advertisement to show, requires the processing of this structured data, e.g., to probe the interests of the user's friends. The available data can be large and processing it requires resources, especially when doing it at scale. Any savings on the amount of required information results in substantially reduced cost.

These characteristics can be found in other domains – in medicine, the doctor makes decisions based on the medical history of the patient (of arbitrary length) and the results of the performed examinations. The doctor may directly diagnose the patient or send him for further testing. Not only the accuracy of the disease prediction is relevant, but also the total cost spent during the process. In the field of malware detection, companies process thousands or millions of new samples every day. The analysis of a sample often involves external services providing structured data. Usually, a service-level agreement restricts the number of requests per time unit or involves other costs. Time to process the sample is also important and depends on the number of queries. If the com-

pany can reduce the average cost to process one sample, it can lead to considerable savings.

To give a specific example, let's look at one of the domains used in this work (see Figure 1). It corresponds to requesting information from a real-world malware detection service.[1] The service provides information about which known malware communicates with a certain domain and what IP or email addresses are associated with it. Crucially, all acquired information can be further analyzed (e.g., a newly learned IP address can be probed to see its associated domains). In this case, the service is freely accessible, but in practice, it can be restricted in various ways. Not only there may be a requirement of payment, there also can be restrictions on the number of requests within a time period or varying delays based on the type of requested information. The goal is to predict whether the domain is malicious while minimizing the associated costs. Given a certain budget of the user (time, money or number of requests), the optimal behavior differs.

As already stated, we propose to work with the samples in their natural form. We frame the task as a Markov decision process (MDP), where a state consists of the currently observed feature tree, and at each step, the algorithm chooses a leaf for expansion, or terminates the feature collection and classifies. Our algorithm is based on Classification with Costly Features framework (Janisch, Pevný, and Lisý 2019a,b), in which the model is trained with deep reinforcement learning (Deep RL) to optimize average accuracy while using a defined per-sample budget. We modify the algorithm to accept the structured data and to work in the variable action space (i.e., for each sample, the set of available actions is different and potentially unbound).

The transition from fixed-length vectors to trees is non-trivial, with two main technical challenges. First, it is not obvious how to process the tree-structured data. Here we propose to use use the Hierarchical Multiple-Instance Learning architecture (Pevný and Somol 2016). Second, it is not clear how to select from the potentially unbound number of actions, corresponding to the tree leaves. Here we adopt a technique from natural language processing, and decompose the policy with a hierarchical softmax (Morin and Bengio 2005), which has not been used in the Deep RL context before. The resulting model is fully differentiable and it is trained with A2C (Mnih et al. 2016), a policy gradient method.

The contributions of this paper are:

1. We formalize a practical and novel problem of classifying structured data with costly features, common in real-world applications.
2. We provide an implementation of Deep RL based algorithm to solve this problem. The technical contribution consists of adapting the existing techniques to work in the new context and combining them together. Moreover, we show how to factorize the complex hierarchical action space efficiently.
3. We release seven new datasets to benchmark algorithms for this problem (six of which are adapted from previously public sources, one is completely new).
4. We provide an empirical evaluation of our algorithm along with two baselines.

[1]The service is accessible at threatcrowd.org.

Background

This section briefly describes the existing techniques used throughout this work. Our method is based on the Classification with Costly Features (CwCF) (Janisch, Pevný, and Lisý 2019a,b) framework to set the objective and to transform the problem into MDP solving. However, the structured data pose non-trivial challenges in the variable input size and the variable number of actions. To create an embedding of the hierarchical input, we use Hierarchical Multiple-Instance Learning (HMIL) (Pevný and Somol 2016). To select the performed actions, we use hierarchical softmax (Morin and Bengio 2005; Goodman 2001). To train our agent, we use Advantage Actor Critic (A2C) (Mnih et al. 2016), a reinforcement learning algorithm from the policy gradient family. The reader familiar with the techniques is advised to skip to Problem Definition and to refer to this section when needed.

Classification with Costly Features

Classification with Costly Features (CwCF) (Janisch, Pevný, and Lisý 2019a,b) is a problem of optimizing accuracy, along with a cost of features used in the process. Let (y_θ, k_θ) be a model where y_θ returns the label and k_θ the features used. Then the objective is:

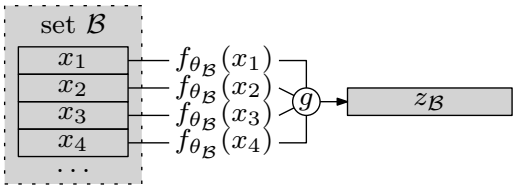
$$\min_{\theta} \mathop{\mathbb{E}}_{(x,y)\in\mathcal{D}} \left[\ell(y_\theta(x), y) + \lambda c(k_\theta(x))\right] \tag{1}$$

Here, (x, y) is a data point with its label, ℓ is the classification loss, c is the cost of the given features and λ is a trade-off factor between the accuracy and the cost. Minimizing this objective means minimizing the expected classification loss together with the λ-scaled per-sample cost. In the original framework, each data point $x \in \mathbf{R}^n$ is a vector of n features, each of which has has a defined cost.

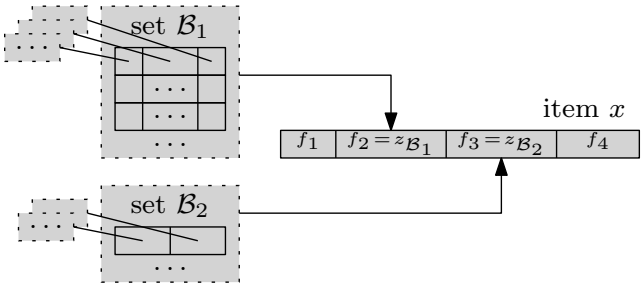
Eq. (1) allows a construction of an MDP with an equivalent solution to the original goal and so the standard reinforcement learning (RL) techniques can be used (Dulac-Arnold et al. 2011; Janisch, Pevný, and Lisý 2019b). In this MDP, an agent classifies one data point (x, y) per episode. The state s represents the currently observed features. At each step, it can either choose to reveal a single feature k (and receiving a reward of $-\lambda c(k)$) or to classify with a label $\hat{y}$, in which case the episode terminates and the agent receives a reward of $-\ell(\hat{y}, y)$. A common choice for the loss function ℓ is a binary loss (1 in case of mismatch, 0 otherwise). Janisch, Pevný, and Lisý (2019a) extend the framework to work with average or strict budget.

Hierarchical Multiple-Instance Learning

Pevný and Somol (2017) introduce Multiple-Instance Learning (MIL); alternatively, Zaheer et al. (2017) introduce Deep Sets – a neural network architecture to learn an embedding of an unordered set $\mathcal{B}$, composed of m items $x_{\{1..m\}} \in \mathbf{R}^n$ (see Figure 2). The items are simultaneously processed into their embeddings $z_i = f_{\theta_\mathcal{B}}(x_i)$, where $f_{\theta_\mathcal{B}}$ is a non-linear function with parameters $\theta_\mathcal{B}$, shared for the set $\mathcal{B}$. All embeddings are



(a) Embedding of a single set.



(b) Embedding of a hierarchy.

Figure 2: In HMIL, the items in a set are simultaneously processed with a non-linear function $f_{\theta_\mathcal{B}}$ and aggregated with g (a). In hierarchies, the individual sets are recursively processed as described and their embeddings are used as the feature values (b).

processed by an aggregation function g, commonly defined as a mean or max operator. The whole process creates an embedding $z_\mathcal{B}$ of the set, and is differentiable.

Hierarchical MIL (Pevný and Somol 2016) works with hierarchies of sets, corresponding to variable sized trees. In this case, the nested sets (where different types of sets have different parameters $\theta_\mathcal{B}$) are recursively evaluated as in MIL, and their embeddings are used as feature values. The soundness of the hierarchical approach is theoretically studied by Pevný and Kovařík (2019).

Hierarchical Softmax

This technique comes from natural language processing (Morin and Bengio 2005; Goodman 2001) and is new in the context of Deep RL. It decomposes a probability $p(y|x)$ into a tree, where each node represents a decision point, itself being a proper probability distribution. The sampling from p can be seen as a sequence of stochastic decisions at each node, starting from the root and continuing down the tree. If we label the probabilities encountered on the path from a root node to y with $p_1, \ldots, p_n$, then $p(y|x) = \prod_{i=1}^{n} p_i$. In our case, the main advantage of this approach is that we don't have to compute all the probabilities in the tree, only those which are needed at each decision point (see Figure 5a).

A2C Algorithm

Advantage Actor Critic algorithm (A2C), a synchronous version of A3C (Mnih et al. 2016), is an on-policy policy-gradient algorithm. It iteratively optimizes a policy π_θ and a value estimate V_θ with model parameters θ to achieve the best cumulative reward in a Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, r, t)$, where $\mathcal{S}$, $\mathcal{A}$ represent the state and action spaces, and r, t are reward and transition functions. Let's define a state-action value function $Q(s,a) = \mathbb{E}_{s' \sim t(s,a)}[r(s,a,s') + \gamma V_\theta(s')]$ and an advantage function $A(s,a) = Q(s,a) - V_\theta(s)$. Then, the policy gradient $\nabla_\theta J$ and the value function loss L_V are:

$$\nabla_\theta J = \mathop{\mathbb{E}}_{s,a \sim \pi_\theta, t} \Big[A(s,a) \cdot \nabla_\theta \log \pi_\theta(a|s) \Big]$$

$$L_V = \mathop{\mathbb{E}}_{s,a,s' \sim \pi_\theta, t} \Big[r(s,a,s') + \gamma V_{\theta'}(s') - V_\theta(s) \Big]^2$$

where in θ' is a fixed copy of parameters θ. To prevent premature convergence, a regularization term L_H in form of the average policy entropy is used:

$$L_H = \mathop{\mathbb{E}}_{s \sim \pi_\theta, t} \Big[H_{\pi_\theta}(s) \Big] \; ; \; H_\pi(s) = - \mathop{\mathbb{E}}_{a \sim \pi(s)} \Big[\log \pi(a|s) \Big]$$

The total gradient is computed as $G = \nabla_\theta(-J + \alpha_v L_V - \alpha_h L_H)$, with α_v, α_h learning coefficients. The algorithm iteratively gathers sample runs according to a current policy π_θ, and the traces are used as samples for the above expectations. Then, an arbitrary gradient descent method is used with the gradient G. Commonly, multiple environments are run in parallel to create a better gradient estimate. Mnih et al. (2016) use asynchronous gradient updates; in A2C the updates are made synchronously.

In our case, it is very expensive to compute all action probabilities. Hence we propose to estimate directly the gradient of the entropy as (Zhang et al. 2018):

$$\nabla_\theta H_{\pi_\theta}(s) = - \mathop{\mathbb{E}}_{a \sim \pi_\theta(s)} \Big[\log \pi_\theta(a|s) \cdot \nabla_\theta \log \pi_\theta(a|s) \Big]$$

and use the performed action to sample the expectation with zero bias; the variance can be lowered with larger batches.

Problem Definition

Let $\mathcal{R}$ be a dataset schema, describing a structure of a particular dataset (see Figure 3); all samples in the datasets follow the same schema. The schema is a tree with an empty root node. Children in the tree describe the features that belong to the parent node. The features consist of their label, data type and cost. A special data type *set* indicates that in a particular sample instance, there will be a collection of **arbitrary number** of same-typed items, described by the level below.

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1..n}$ be a dataset of samples (x, y) with a schema $\mathcal{R}$, where x is the complete tree with all features of the sample and y is the label. The structure of x follows the schema – a root node denotes the sample object and there is single node with a value for every feature described in $\mathcal{R}$. A node with *set* type contain multiple children, one for each object in the set (see Figure 1 for an example); a set can also be empty.

The algorithm works with partial observations of the complete feature tree. Let $\dot{x}$ denote a subtree of x with a shared root, a partial observation of the complete sample. Let a leaf node in $\dot{x}$ be *observed*, if it's feature value is known. The operation $reveal(\dot{x}, v)$, where v is an unobserved node, copies

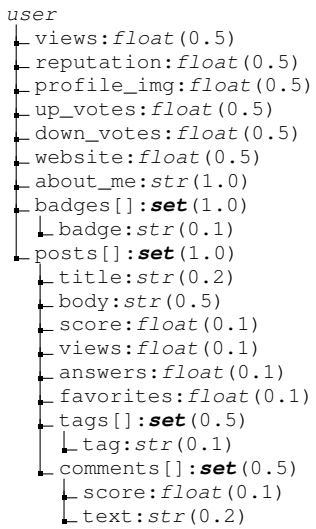


Figure 3: A schema of the *stats* dataset. A sample is an instance of *user* with several features of various data types. The features with *set* type are collections of same-typed items, described in one level below. The costs of the features are in the parentheses.

the true value of v from x to $\dot{x}$. For sets, this means that all children are revealed (without their values). As an exception, if a feature's cost is zero, it is automatically observed.

In one episode, the algorithm analyzes and classifies one sample x. At the beginning, the free and reachable features are automatically observed. At each step the algorithm can either select one leaf, $reveal$ it and pay the its cost, or it can terminate the process and classify.

We use the CwCF framework to set the objective. The overall goal is to optimize the eq. (1) over the training dataset $\mathcal{D}$, with a specific trade-off parameter λ. In this work, we only study the λ-target setting; the objective can be modified to a strict or average budget with a specific target (Janisch, Pevný, and Lisý 2019a). Apart from optimizing the eq. (1), we also want a model that generalizes well to unseen data.

Method

Following the CwCF method, the problem can be naturally represented as an MDP in which a random sample $(x, y) \in \mathcal{D}$ is processed during a single episode. The state s consists of the partially observed tree $\dot{x}$, the set of actions is composed of one terminal action a_t and one action per unobserved leaf in $\dot{x}$. The transition dynamics t and reward function r are defined as follows:

$$t(s,a) = \begin{cases} \mathcal{T} & \text{if } a = a_t \\ reveal(\dot{x}, a) & \text{otherwise} \end{cases}$$

$$r(s,a) = \begin{cases} -\ell(\hat{y}, y) & \text{if } a = a_t \\ -\lambda c(a) & \text{otherwise} \end{cases}$$

where $\mathcal{T}$ is a terminal state and $\hat{y}$ is a class prediction of the model. Since the agent does not know the true values

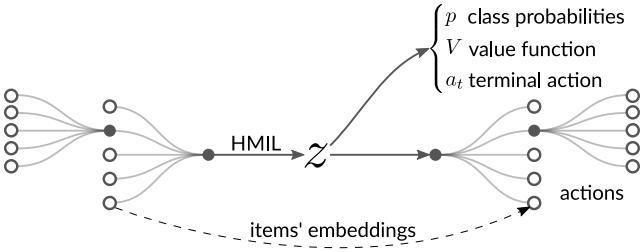


Figure 4: The architecture of the model. The hierarchical input is processed with HMIL to create the embedding z. Separate heads compute class probabilities p, value function V and the terminal action potential a_t. In a separate path, a hierarchical probability distribution over actions is computed (see Figure 5).

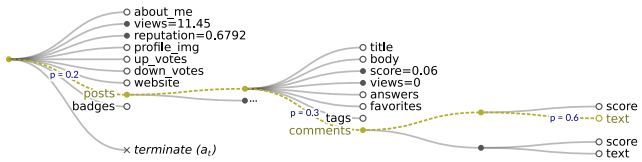
of x, y, which are randomly sampled from the dataset, the whole MDP is effectively stochastic in its view.

The policy is a neural network which takes a $\dot{x}$ as an input and outputs a probability distribution over all actions. The unobserved feature values are replaced with zeros. To be able to distinguish a valid zero value from an unobserved value, each feature is augmented with a a single float value, called a *mask*. The mask has a value 1 if the feature is observed and 0 if not. In case of sets, the mask signalizes what portion of the corresponding branch is revealed, with a value between 0 and 1. For limited trees, the set mask can be recursively computed as an average of masks in the corresponding subtree; otherwise its 0.

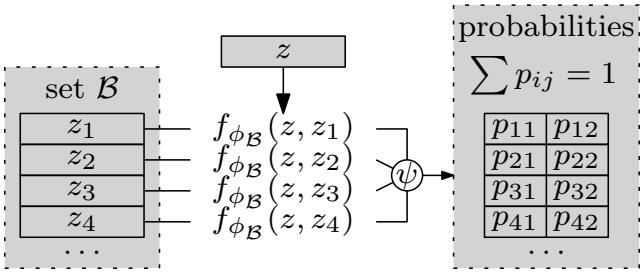
The data types present in the samples have to be transformed into floats prior to their processing by the model. For strings, we observed a good performance with a character trigram histograms (Damashek 1995). This hashing mechanism is simple, fast and conserves similarities between strings. Categorical features are translated into one-hot encoded array. The input, augmented with the masks, is processed with HMIL, which encodes it into an embedding z (see Figure 4). The further processing is split into separate streams.

First, separate heads output a class probability distribution $p(z)$, the value function $V(z)$ and pre-softmax energy of the terminal action $a_t(z)$, used later. The p output can be used to initialize the first part of the network by sampling random incomplete subtrees from the dataset and pretraining only the classification part. For the heads, we use single linear layers. More complex functions are likely to achieve better performance; however, it is not the goal of this paper.

Second, to obtain a probability distribution over all possible actions, we employ the hierarchical softmax (see Figure 5). Starting at the root of $\dot{x}$, stochastic decisions are made at each node, continuing down the tree. In sets, all items' features are considered at once, instead of choosing an item first and then continuing down (this is a deliberate implementation, both approaches seem to be viable). Each type of set $\mathcal{B}$ share parameters $\phi_{\mathcal{B}}$ (different from HMIL set parameters $\theta_{\mathcal{B}}$). The previously computed sample-level embedding z and embeddings of each item $z_i = f_{\theta_{\mathcal{B}}}(x_i)$ are used to compute pre-softmax energies for each feature in the item, with



(a) selecting an action



(b) feature probabilities in a set

Figure 5: (a) An action is selected with a series of stochastic decisions according to the computed probability distribution in each joint. The final probability of the performed action is a product of the partial probabilities on the path. (b) In a single set, the feature probabilities are computed by processing each item with a function $f_{\phi_\mathcal{B}}$ and a softmax function ψ.

$f_{\phi_\mathcal{B}}(z, z_i)$. Note that $f_{\theta_\mathcal{B}}$ and $f_{\phi_\mathcal{B}}$ are different functions, but both operate on an item in a set. All action energies across all items in the set are then passed through the softmax to obtain the final probabilities. Already observed features or completely explored branches (easily checked for their mask to be 1) are excluded from the softmax. At the root level, the terminal action's energy a_t is added to the softmax.

The complete probability distribution over all actions is never computed. Instead, only one action a in a state s is sampled with the hierarchical softmax to be performed. The final differentiable probability $\pi(a|s)$ is computed as a product of the partial probabilities of the path from the root to a leaf. The factorization of the policy π with hierarchical softmax plays three important roles. First, it is the computational savings as the whole distribution need not to be computed. Second, it enables effective learning (Tang and Agrawal 2019). Third, the model is much more interpretable, because the action can be factored into the selected objects and features.

The action selection mechanism can be interpreted as the model's attention to relevant objects and features at each level. For example, if posts seem to be relevant for the prediction, the probability of selecting this path will be high. If the set of posts has not been previously acquired, the action is to request all objects in the set. If it has, the process continues down to select a concrete post and its feature.

During training, multiple samples in a batch are processed in parallel and the model is synchronously updated with the A2C algorithm. Note that we use this algorithm to demonstrate the method without the ambition to achieve the best performance possible – any recent modification to the RL algorithm is likely to improve the performance. Exact implementation with hyperparameter settings is available in Supplementary Material A.

dataset	# samples	# classes	depth
carcinogenesis	329	2	2
hepatitis	500	2	2
mutagenesis	188	2	3
ingredients	39 774	20	2
sap	35 602	2	2
stats	8 318	3	3
web	1 171	2	3

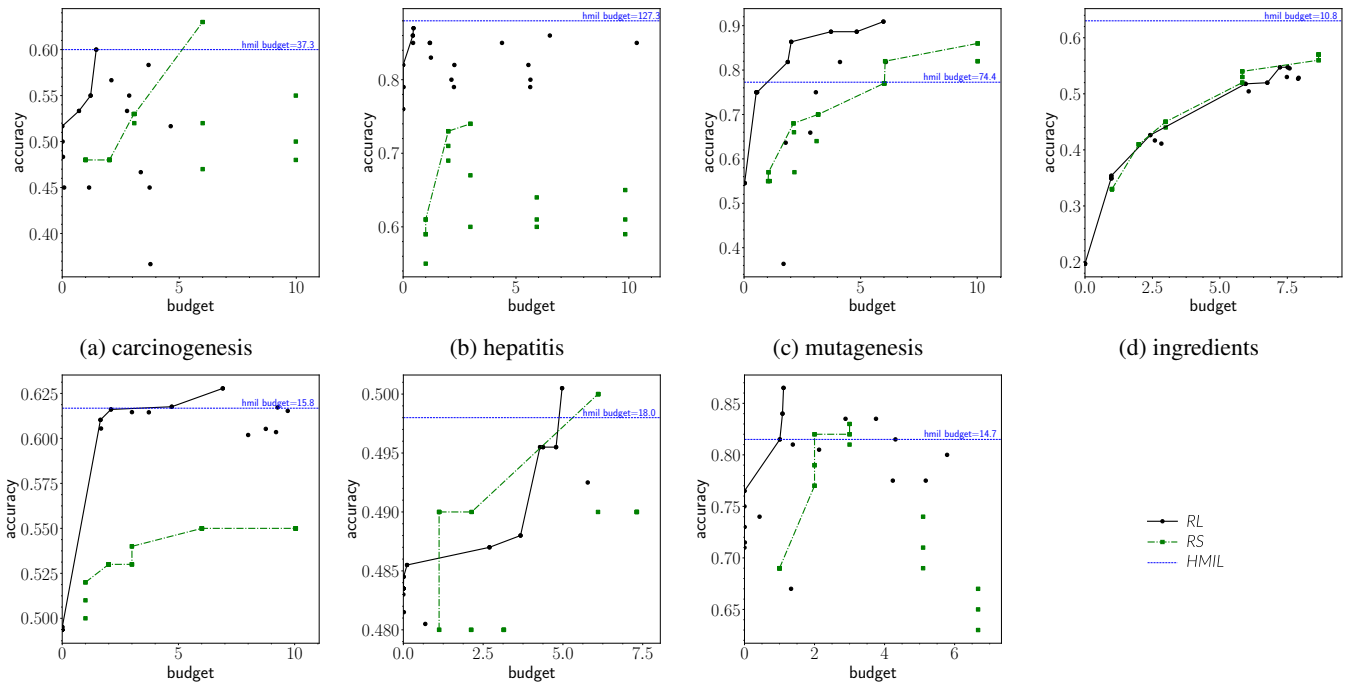
Table 1: Statistics of the used datasets.

Experiments

Used datasets We used seven distinct public relational classification datasets and compare to two baseline algorithms. We acquired six publicly available relational datasets, adapted them into hierarchical structure and added costs. The costs were assigned manually, in a non-uniform way, respecting that in reality some features are more costly then others (e.g., to get a patient's age is easier than doing a blood test). In practice, the costs would be assigned by the real value of the resources needed to retrieve them. One dataset was sourced from a real-world malware analysis service (the *web* dataset), with a permission to share. The costs here correspond to the API requests. To facilitate experimentation, we limited the depth for the datasets to fit into the memory. For reproducibility, we attach the processed versions, along with a library to load them. The datasets are summarized in Table 1 and the detailed descriptions, splits, structure and feature costs are available in Supplementary Material B.

Compared algorithms The scarcity of algorithms dealing with costly relational data also makes it hard to find suitable algorithms for comparison. Note it is not possible to preselect a set of features in the datasets, because each sample can contain a variable number of objects in its sets, with variable depth. We included two baseline algorithms. First, if we consider only the hierarchical data without any costs, we can use the HMIL (Pevný and Somol 2016) method with full samples. This approach should result in an upper bound on the accuracy for the given dataset, because it has access to complete information. We refer to this method as *HMIL*. Second, we can include the costs with a random sampling (refered to as *RS*). For each sample, we construct a partial observation such that it fits into a defined budget. Then we train HMIL with these partial samples. As this method is uninformed (i.e., it randomly samples features), it should result into a lower bound on the accuracy for a particular budget. We refer to the method described in this paper as *RL*.

Note that the method consists of several key components, none of which can be removed. Hence, we cannot perform an ablation analysis. At an extreme, the *HMIL* method can be thought of an ablation of our method without the feature selection and *RS* method can be thought as our method with random feature selections.



(a) carcinogenesis (b) hepatitis (c) mutagenesis (d) ingredients

(e) sap (f) stats (g) web

Figure 6: The raw performance of different algorithms on testing data, shown in the budget vs. accuracy plane. For each algorithm, we included the raw results as scattered points, as well as their pareto frontier. Each point is a result of a single model trained with a specific budget and a random seed. We show our method (*RL*), an algorithm trained on randomly sampled partial samples with a budget (*RS*) and HMIL trained with complete information (*HMIL*). The vertical line visualizes the accuracy of *HMIL* and the average cost of all features.

Results

For each dataset, we ran the algorithm with three different seeds for each of six different trade-off values λ. We made comparable number of runs with the baseline methods. The learned models are evaluated on the testing data and are plotted in the budget-accuracy plane, as seen in Figure 6. We include the pareto frontier to better visualize the best performance of the algorithms. When comparing the results, the whole range of budgets has to be taken into account.

Explainability In Figure 7, we visualized the feature acquisition process, with a trained model, for one sample in the *stats* dataset. The sequential nature of the algorithm makes it open for further analysis. The model decisions can be explained in the form of probabilities the model assigns to objects and their features and the changes in the class probability distribution after acquiring certain features. Note, that, as each sample is different, the assigned probabilities differ for different samples. The explicit choices the agent makes could be compared to a domain expert, either to verify the the agent's rationality or to improve the expert's decision-making. For more samples, see Supplementary Material C.

Performance The generalization to unseen testing data is challenging, especially in smaller datasets (carc., hepa. and muta.). Nevertheless, the *RL* algorithm greatly outperformed *RS* in these settings. *RL* also reached the performance of *HMIL* in carc. and hepa., with only a fraction of the cost ($1/18$ in carc.; $1/127$ in hepa). In muta., *RL* outperformed both other algorithms.

In ingredients, the performance of *RL* and *RS* is almost identical. Since this dataset contains a single set of ingredients, the main challenge is whether to continue identifying them or stop. The *RL* algorithm sees only an undisclosed list of ingredients and can only make a random selection, hence it cannot do better than *RS*.

In sap and web datasets, *RL* greatly outperforms both *RS* and *HMIL*, and it reaches the performance of *HMIL* again with a fraction of the cost ($1/8$ in sap; $1/15$ in web). In stats dataset, *RL* and *RS* perform similarly. Two runs of *RS* outperform *RL* in terms of cost-accuracy; the results may be explained with a random initialization. Again, *RL* reaches the performance of *HMIL* with $1/3$ of its cost.

In few datasets, *RL*, with a fraction of the cost, exceeds the performance of *HMIL*, which can access all features. We investigated the case and discovered that *HMIL* simply overfits, perfect accuracy on the training set, yet failing to generalize well.

A negative property of *RL* is its variance in the training runs. In practice, several runs would need to be executed and only the best one selected, based on the validation data.

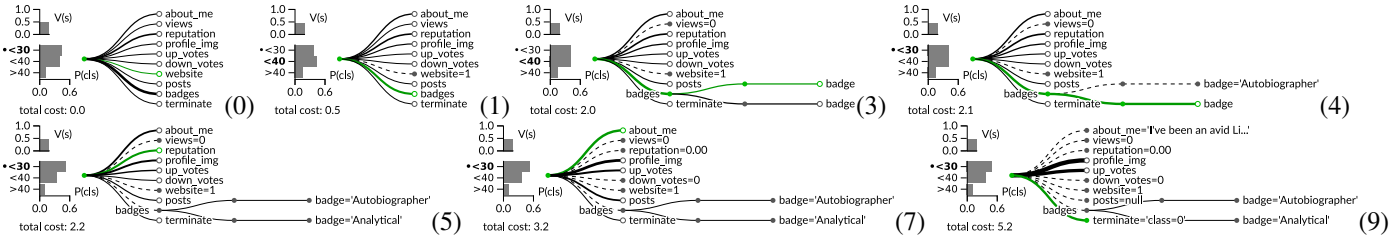


Figure 7: Visualization of a sequential feature selection with a sample from *stats* dataset. The example is artificial to some extent, but demonstrates the process well. The goal is to classify users into three age categories, based on their posts, plain text description, achieved badges and other metrics. Some steps are skipped, for clarity and space. Until step (4), the algorithm is undecided between class *<30* and *<40*. After learning that the user has a badge *Analytical* in step (5), the algorithm assigns 53% probability to the class *<30*. In step (9), the algorithm terminates the querying process and classifies correctly with the label *<30*. The line thickness visualizes the weight the algorithm assigns to different features; the dot shows the correct class.

Wall-clock measurements One downside of Deep RL based algorithms is the usually long time required to train them. To give an unbiased view, we measured the running times on a machine with 2 cores of Intel Xeon 2.60GHz and 4GB of memory (without GPU). The *RL* method needed about 1 hour to train in the datasets with shorter *epoch_length* parameter (carc., hepa., muta., stats and web). The other datasets (ingredients and sap) took about 24 hours to train. The *HMIL* method usually finished in order of minutes and *RS* method in tens of minutes. Convergence graphs for *RL* are available in Supplementary Material D.

Related Work

Our work is an extension of the Classification with Costly Features (CwCF) problem, originally defined by Dulac-Arnold et al. (2011) and lately advanced by Janisch, Pevný, and Lisý (2019a,b). The former algorithm is also based on reinforcement learning (RL), but work only with fixed-length vectors. Shim, Hwang, and Yang (2018) proposes a method for sets of features, but its application to trees of features remains unclear. Although it may be possible to create a set of *(path, value)* features from all tree leaves, the method does not specify how to do this. More specifically, the way to encode the paths, especially for deep or infinite trees, is not defined. Note that the set of all paths is not known until the tree is fully expanded.

In Deep RL, the action space is usually formed with orthogonal dimensions and can be factorized in some way (Tang and Agrawal 2019; Chen et al. 2019; Metz et al. 2017). In our method, we factorize the complex action space with hierarchical softmax (Morin and Bengio 2005; Goodman 2001), which was not used previously.

The problem is distantly related to graph classification algorithms (e.g., (Zhou et al. 2018; Hamilton, Ying, and Leskovec 2017; Perozzi, Al-Rfou, and Skiena 2014; Kipf and Welling 2016)). However, our case is very specific, because the samples in our work are trees and the optimization goal is accuracy in presence of a budget.

Conclusion

Driven by problems from real-world domains (e.g., malware detection or targeted advertising) in which the data are available only in the form of trees, we present a Deep RL based method capable of processing this data. The method accepts incomplete tree-structured input and directly selects more features from inside the hierarchy, in a sequential manner, with the goal of maximizing the classification accuracy and minimizing the costs of acquired features. Crucially, the method differs from previous approaches (which targeted fixed-length vector samples) with its ability to directly work with the tree-structured input, which vastly enhances its scope of possible application.

The method is based on a unique combination of several techniques with a Deep RL algorithm. On a set of seven datasets, we demonstrated its ability to work with different domains, its performance and generalization to unseen data. We also showed that our algorithm often exceeded, with a fraction of the cost, even the base algorithm which can access the complete data. In a single sample, we analyzed the taken decisions and showcased the explainability of our model.

References

Chen, Y.-E.; Tang, K.-F.; Peng, Y.-S.; and Chang, E. Y. 2019. Effective Medical Test Suggestions Using Deep Reinforcement Learning. *arXiv preprint arXiv:1905.12916* .

Damashek, M. 1995. Gauging similarity with n-grams: Language-independent categorization of text. *Science* 267(5199): 843–848.

Dulac-Arnold, G.; Denoyer, L.; Preux, P.; and Gallinari, P. 2011. Datum-wise classification: a sequential approach to sparsity. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 375–390. Springer.

Goodman, J. 2001. Classes for fast maximum entropy training. *arXiv preprint cs/0108006* .

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 1024–1034.

Janisch, J.; Pevný, T.; and Lisý, V. 2019a. Classification with Costly Features as a Sequential Decision-Making Problem. *arXiv preprint arXiv:1909.02564* .

Janisch, J.; Pevný, T.; and Lisý, V. 2019b. Classification with

Costly Features using Deep Reinforcement Learning. In *Proceedings of 33rd AAAI Conference on Artificial Intelligence*.

Kingma, D. P.; and Ba, J. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Kipf, T. N.; and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* .

Metz, L.; Ibarz, J.; Jaitly, N.; and Davidson, J. 2017. Discrete sequential prediction of continuous actions for deep rl. *arXiv preprint arXiv:1705.05035* .

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 1928–1937.

Morin, F.; and Bengio, Y. 2005. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, 246–252. Citeseer.

Motl, J.; and Schulte, O. 2015. The CTU prague relational learning repository. *arXiv preprint arXiv:1511.03086* URL https://relational.fit.cvut.cz/.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 701–710. ACM.

Pevný, T.; and Kovařík, V. 2019. Approximation capability of neural networks on spaces of probability measures and tree-structured domains. *arXiv preprint arXiv:1906.00764* .

Pevný, T.; and Somol, P. 2016. Discriminative models for multi-instance problems with tree structure. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, 83–91. ACM.

Pevný, T.; and Somol, P. 2017. Using neural network formalism to solve multiple-instance problems. In *International Symposium on Neural Networks*, 135–142. Springer.

Shim, H.; Hwang, S. J.; and Yang, E. 2018. Joint Active Feature Acquisition and Classification with Variable-Size Set Encoding. In *Advances in Neural Information Processing Systems*, 1375–1385.

Tang, Y.; and Agrawal, S. 2019. Discretizing continuous action space for on-policy optimization. *arXiv preprint arXiv:1901.10500* .

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J. 2017. Deep sets. In *Advances in neural information processing systems*, 3391–3401.

Zhang, Y.; Vuong, Q. H.; Song, K.; Gong, X.-Y.; and Ross, K. W. 2018. Efficient Entropy for Policy Gradient with Multidimensional Action Space. *arXiv preprint arXiv:1806.00589* .

Zhou, J.; Cui, G.; Zhang, Z.; Yang, C.; Liu, Z.; and Sun, M. 2018. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434* .

Supplementary Material:
Cost-Efficient Hierarchical Knowledge Extraction with Deep Reinforcement Learning

A Implementation and Hyperparameters

The model's parameters are initialized according to the provided dataset schema. Parameters $\theta_{\mathcal{B}}, \phi_{\mathcal{B}}$ are created for each set $\mathcal{B}$. We use a fixed embedding size of 64 (the output of $f_{\theta_{\mathcal{B}}}$), across all sets and datasets. ReLu is used as the activation function. The learning weight of the A2C algorithm were initialized as $\alpha_v = 0.5, \alpha_h = 0.05$, where the policy entropy controlling weight (α_h) exponentially decays by a factor 0.5 every *epoch_length* steps. We use Adam optimizer (Kingma and Ba 2015), with L2 regularization 10^{-4}. The gradients are clipped to a norm of 0.1. The network is initialized by pretraining the classifier with randomly generated partial samples from the dataset, with cross-entropy loss, learning rate 3×10^{-3} and early stopping. The learning rate of the main training exponentially decays from 3×10^{-3} by a factor of 0.5 every $10 \times epoch\_length$ steps. During the main training, the classifier is trained only in states where the agent decides to terminate. For each dataset, we run the algorithm for $100 \times epoch\_length$ steps, and select the best performing iteration based on its validation performance (reward). Hyperparameters for each dataset are in Table A.1, along with their training/validation/testing splits. The class distribution column displays the prior distribution of samples for each class.

Table A.1: Dataset hyperparameters and splits.

dataset	epoch_length	batch_size	#train	#test	#val	class distribution
carcinogenesis	100	128	209	60	60	0.45 / 0.55
hepatitis	100	128	300	100	100	0.41 / 0.59
mutagenesis	100	128	100	44	44	0.34 / 0.66
ingredients	1000	1024	29774	5000	5000	0.01∼0.20
sap	1000	1024	15602	10000	10000	0.5 / 0.5
stats	100	256	4318	2000	2000	0.49 / 0.38 / 0.12
web	100	256	771	200	200	0.27 / 0.73

B Dataset details

In this section, we provide some details about the nature of the used datasets, their source and the schemas in Figure B.1.

Carcinogenesis†: In this dataset, the sample is a molecule consisting of number of atoms, each connected with others through bonds. The bonds are divided into four categories and are provided as a list of relations between two atoms and their features. The task is to determine whether the molecule is carcinogenic or not.

Hepatitis†: A medical dataset containing patients infected with hepatitis, types B or C. Each patient has various features (e.g., sex, age, etc.) and three sets of indications. The task is to determine the type of the disease for each patient.

Mutagenesis†: This dataset consists of molecules which are tested on a particular bacteria for mutagenicity. The molecules themselves have several features and consist of atoms with features and bonds. The dataset is similar to carcinogenesis, but its structure is very different.

Ingredients: The dataset was retrieved from Kaggle[2] and contains recipes that have a single list of ingredients. The task is to determine the type of cuisine of the recipe. This is the simplest dataset we use and can be used for quick experiments and algorithm validation.

SAP†: In this artificial dataset, the task is to determine whether a particular customer will buy a new product based on the list of past sales. A customer is defined by various features and the list of sales. We rebalanced the dataset to contain roughly the same amount of both classes.

Stats†: This dataset is a anonymized content dump from a real Q&A website Stats StackExchange. We extracted a list of users to become samples and set artificial goal of predicting their age category. Each user has several features, a list of posts and badges. The posts also contain their own features and a list of tags and comments.

Web: This dataset is about malicious domain detection. We created it by querying ThreatCrowd[3], an online service providing malware analysis of web domains. Each domain contains its URL as a free feature and a list of associated IP addresses, emails and malware hashes. These objects can be further reverse-looked up for another domains.

The datasets marked with † symbol are retrieved from Motl and Schulte (2015) and processed into trees, usually by fixing a root and unfolding the graph into a defined depth. Float values in all datasets are normalized. Strings were processed with the tri-gram histogram method (Damashek 1995), with modulo 13 index hashing. The datasets were split into training, validation and testing sets.

[2]https://kaggle.com/alisapugacheva/recipes-data

[3]https://threatcrowd.org

For each dataset, two hyperparameters exist: *epoch_length* and *batch_size*. The first is the number of steps considered to be an epoch and controls other parameters, such as decay rate of learning rate. The latter controls the number of samples processed in parallel to form a batch. All other hyperparameters are shared across all datasets.

Figure B.1: Datasets schemas used in this work. The trees show the feature names, their types and their cost in parentheses. Features with a *set* type contain arbitrary number of same-typed items.

C Sample Runs

Below we show one selected sample from each dataset as processed by the proposed algorithm. Read left-to-right, top-to-down. At each step, the current knowledge tree is shown, the probabilities of actions are visualized by the line thickness. The green line marks the selected action. Left of the knowledge graph, there is a visualization of the current state value and class probabilities. The correct class is visualized with a dot, the current best prediction is in bold. All displayed models were trained with $\lambda = 0.001$, apart from the carcinogenesis dataset, trained with $\lambda = 0.01$. Interactive visualization with more samples can be found in the attached supplementary zip file.

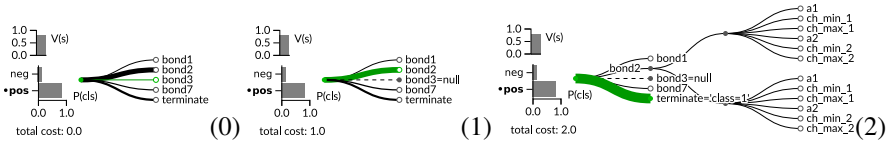


Figure C.2: Sample from dataset *carcinogenesis*

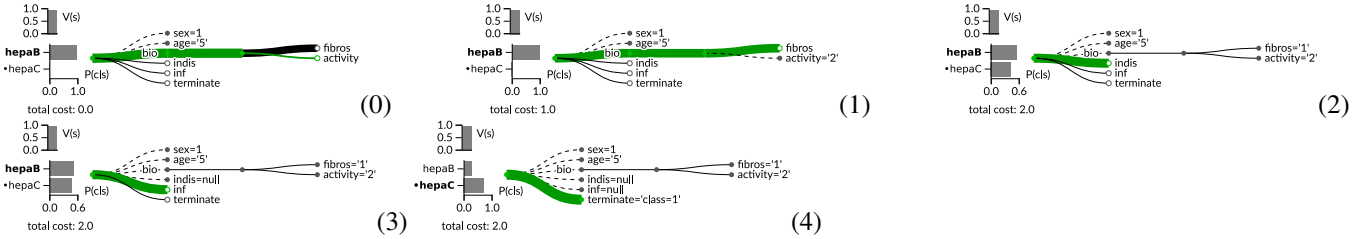


Figure C.3: Sample from dataset *hepatitis*

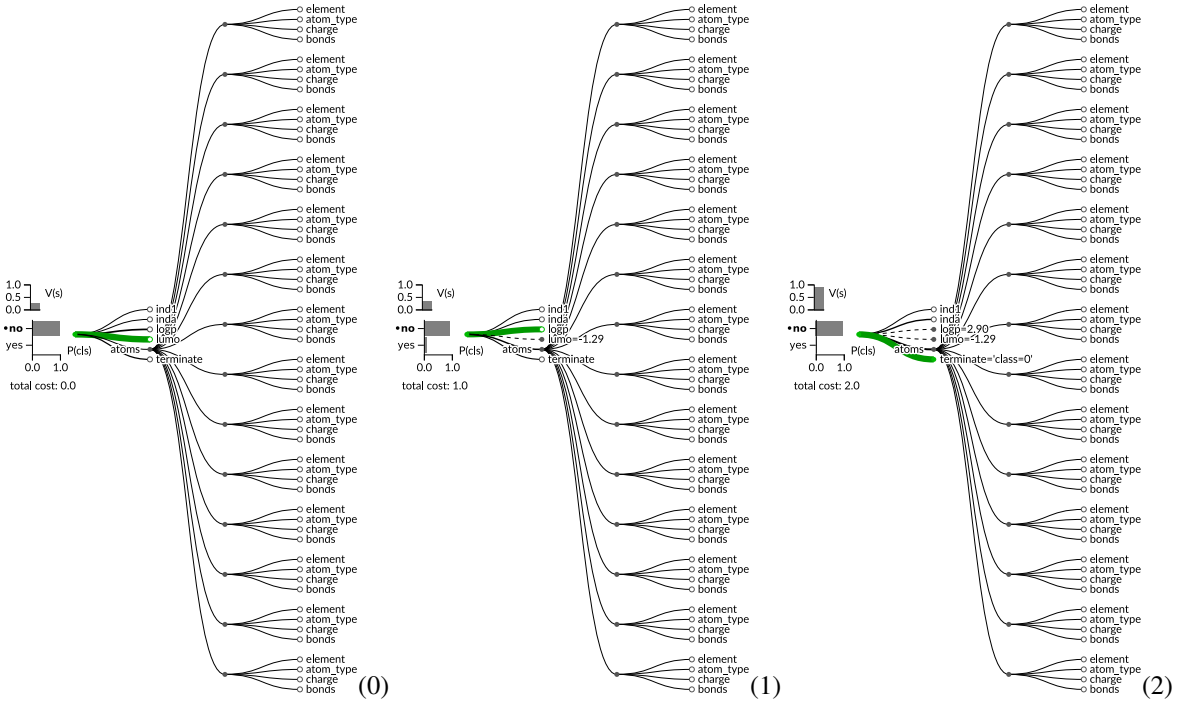


Figure C.4: Sample from dataset *mutagenesis*

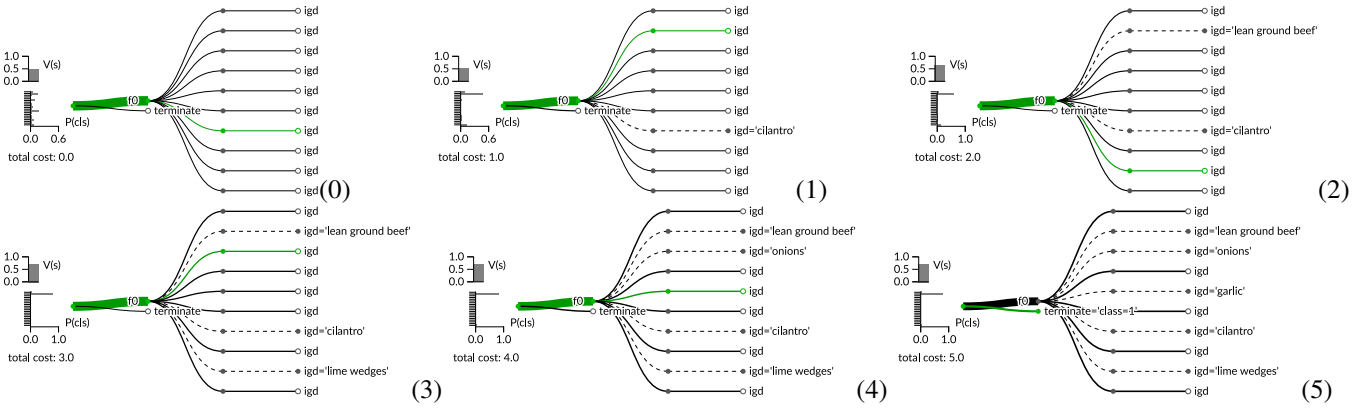


Figure C.5: Sample from dataset *ingredients*

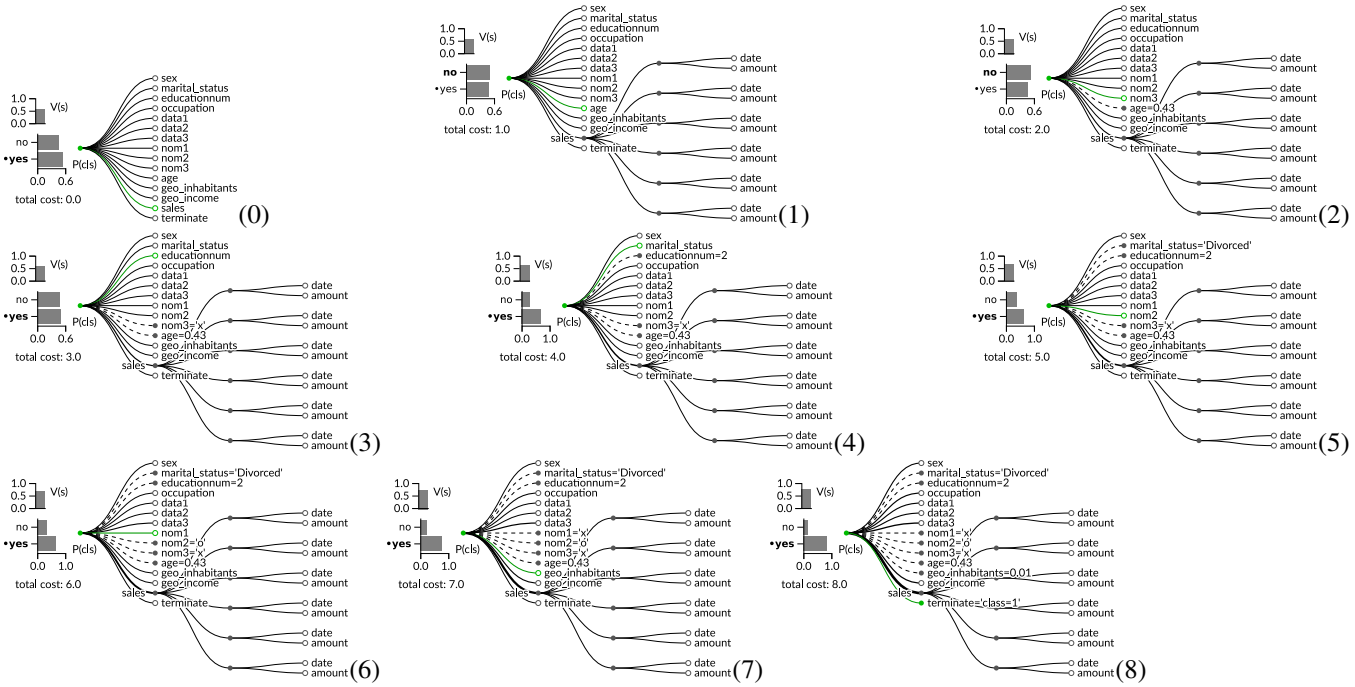


Figure C.6: Sample from dataset *sap*

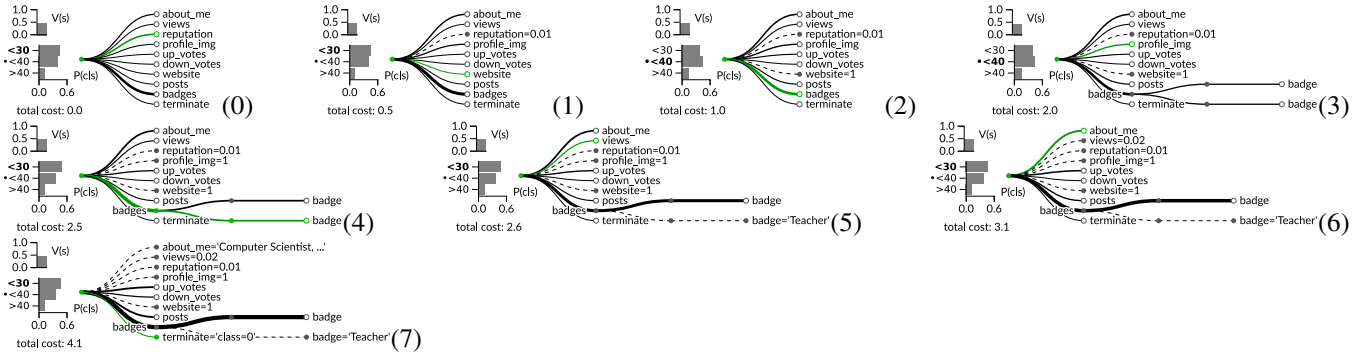


Figure C.7: Sample from dataset *stats*

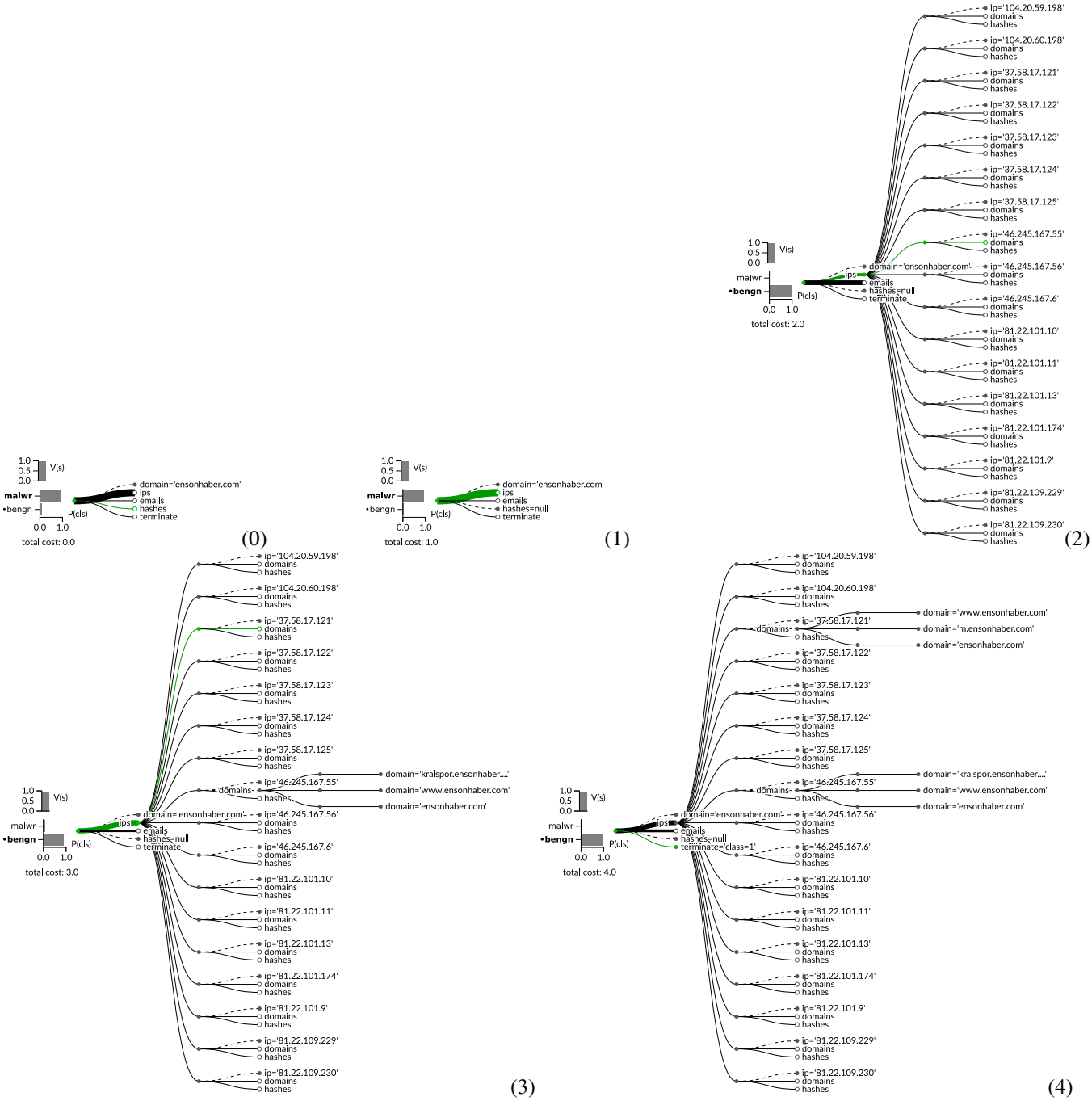


Figure C.8: Sample from dataset *web*

D Convergence Graphs

For each dataset, we present six different runs with $\lambda \in \{0.3, 0.1, 0.03, 0.01, 0.001, 0.0001\}$ in the following figures. Each subfigure, corresponding to a single run, is divided into three diagrams, showing the progression of *reward*, actually spent *cost* and *accuracy* during the training, averaged over over whole dataset. The reader can use the plots to examine behavior of the algorithm (mainly the importance of cost vs. accuracy) when run with different parameters λ. Evaluation on training and validation sets is shown. The best iteration (based on the validation reward) is displayed by the dashed vertical line.

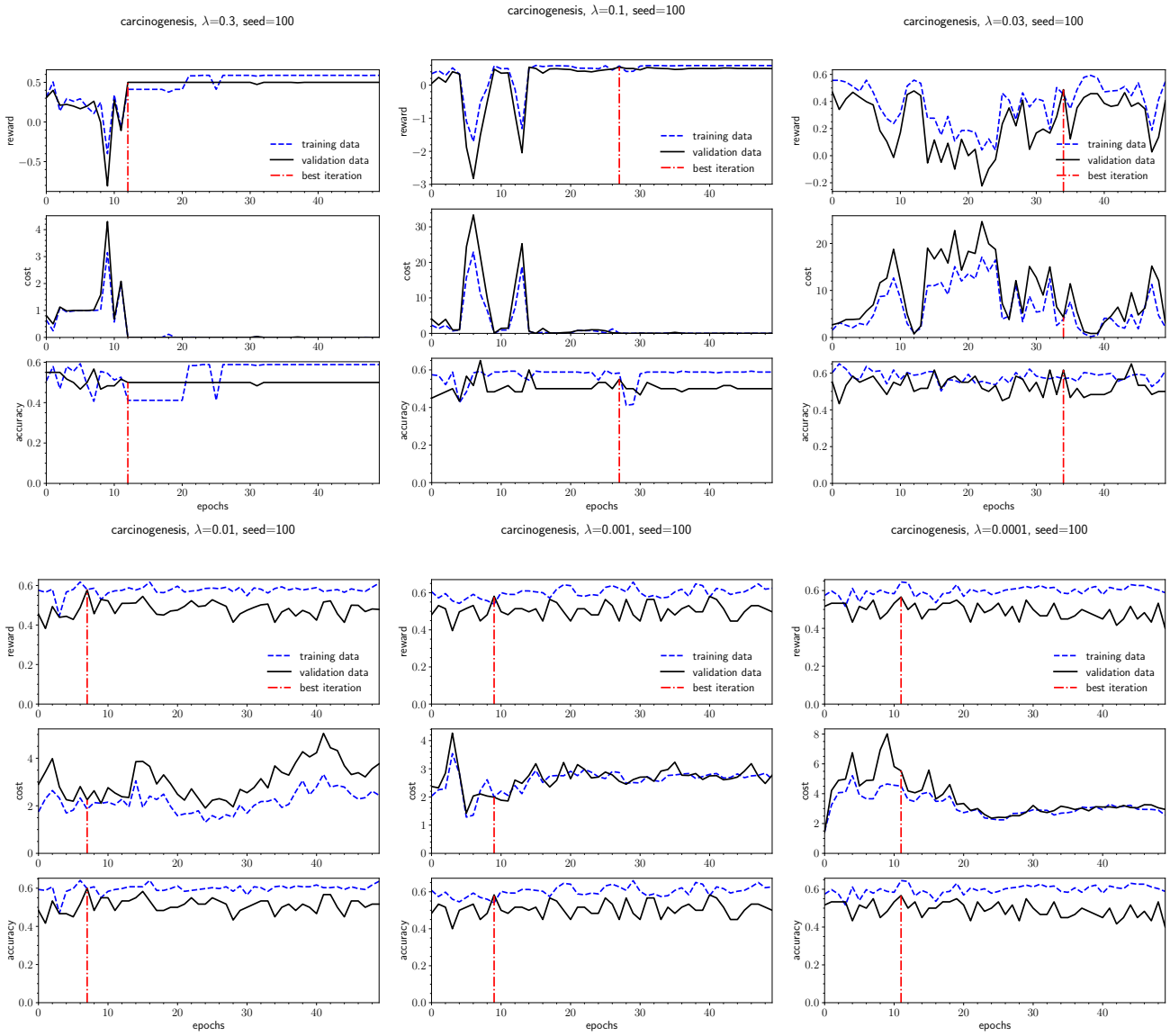


Figure D.9: Convergence graphs for *carcinogenesis* dataset.

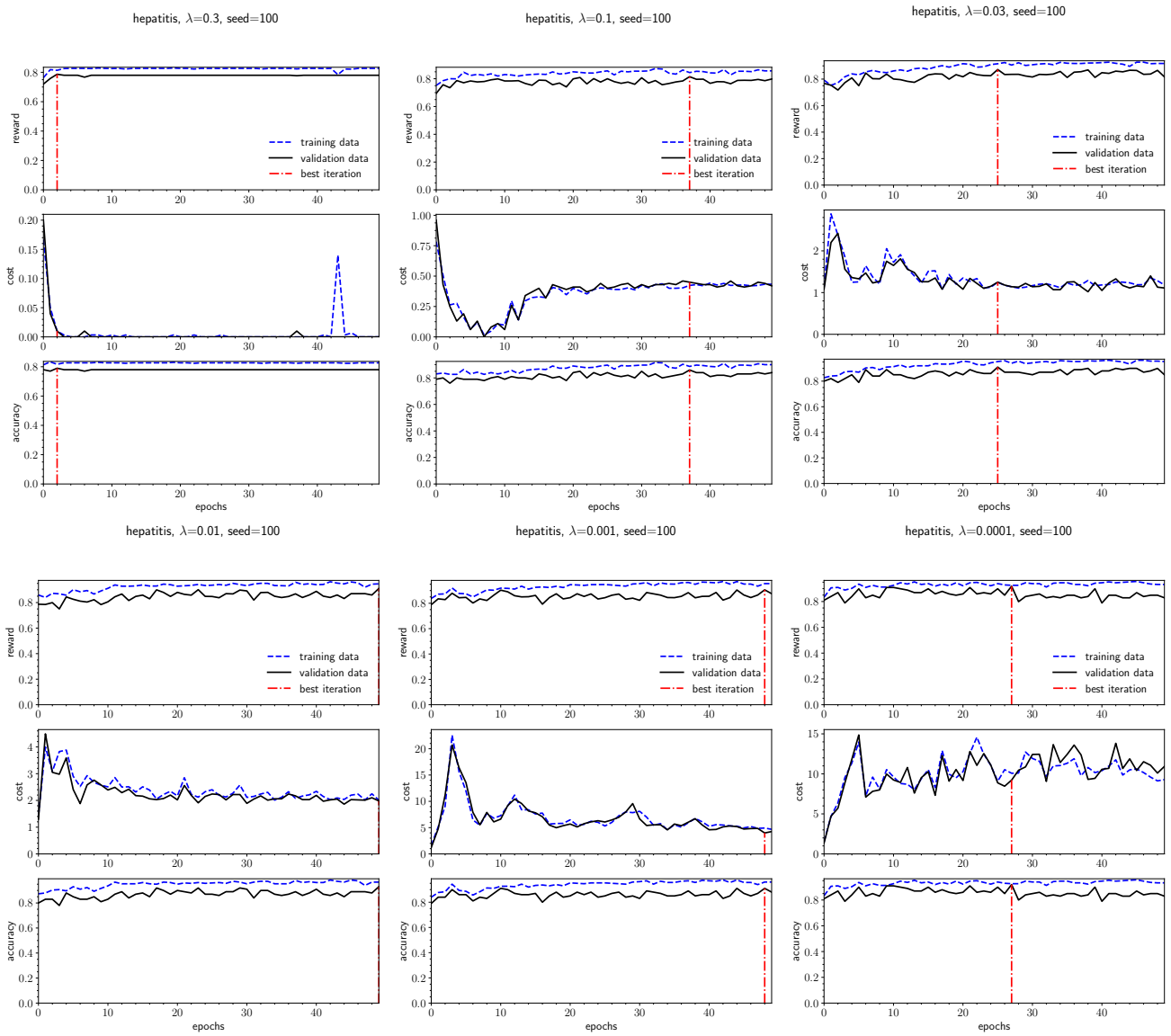


Figure D.10: Convergence graphs for *hepatitis* dataset.

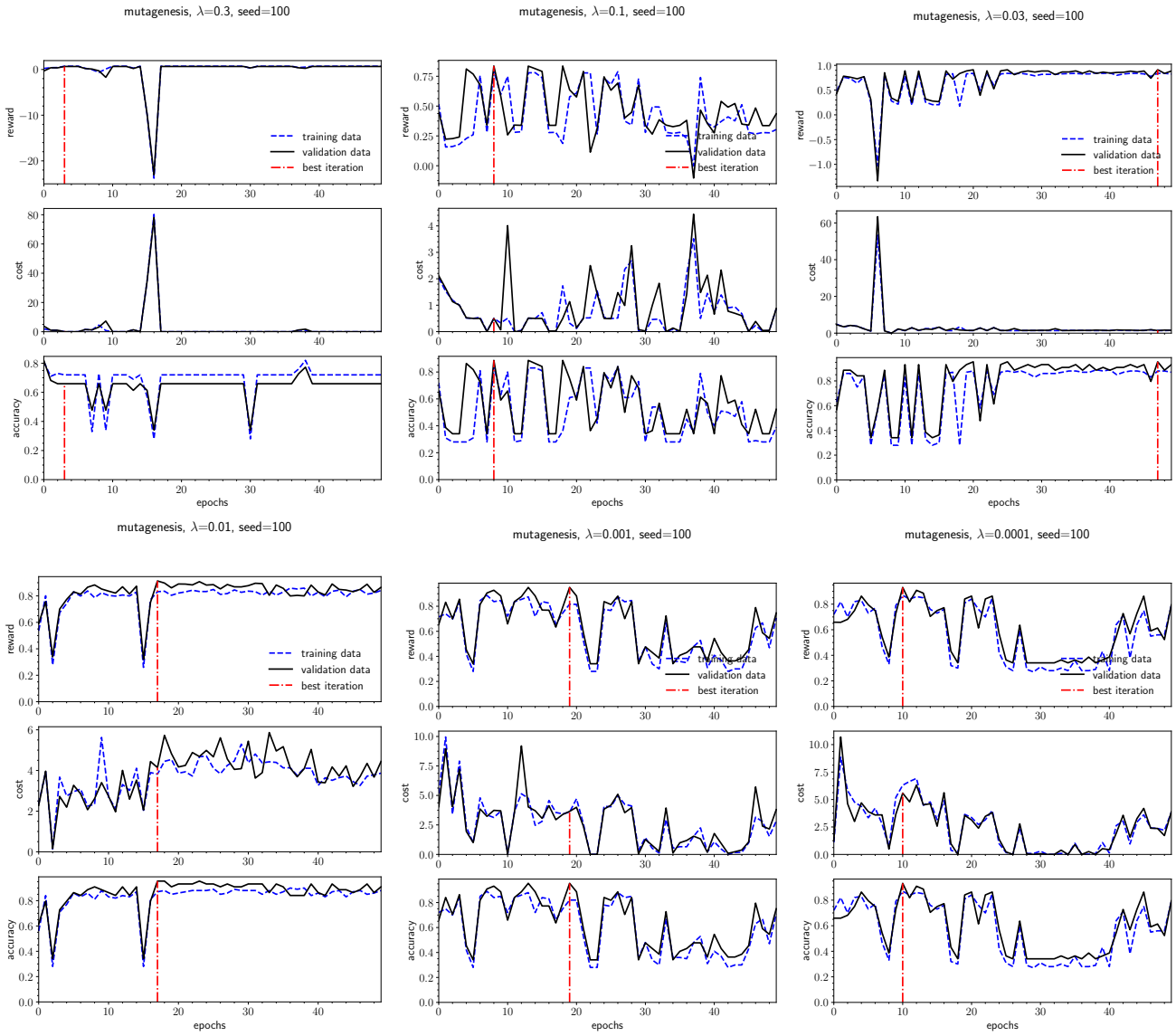


Figure D.11: Convergence graphs for *mutagenesis* dataset.

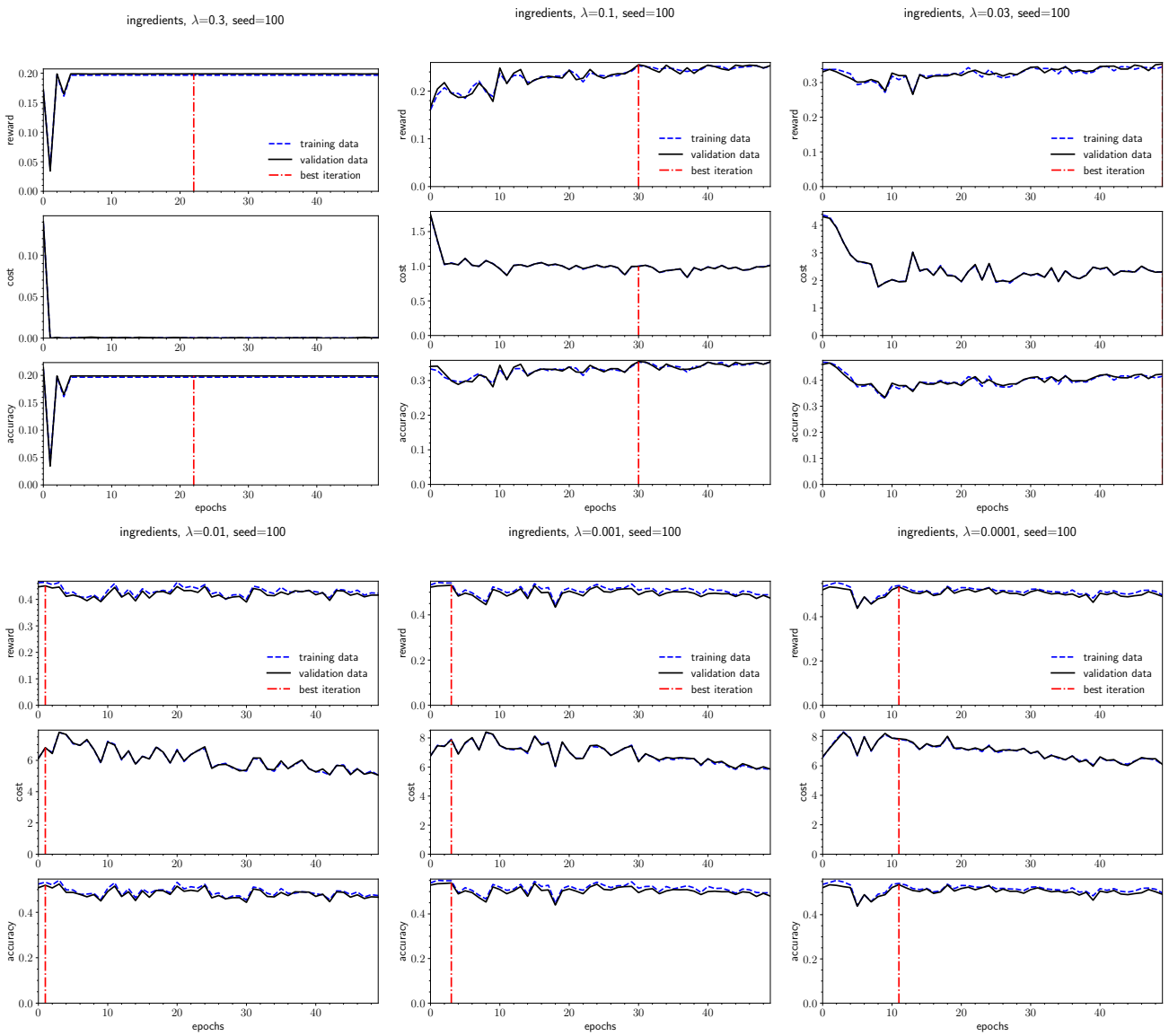


Figure D.12: Convergence graphs for *ingredients* dataset.

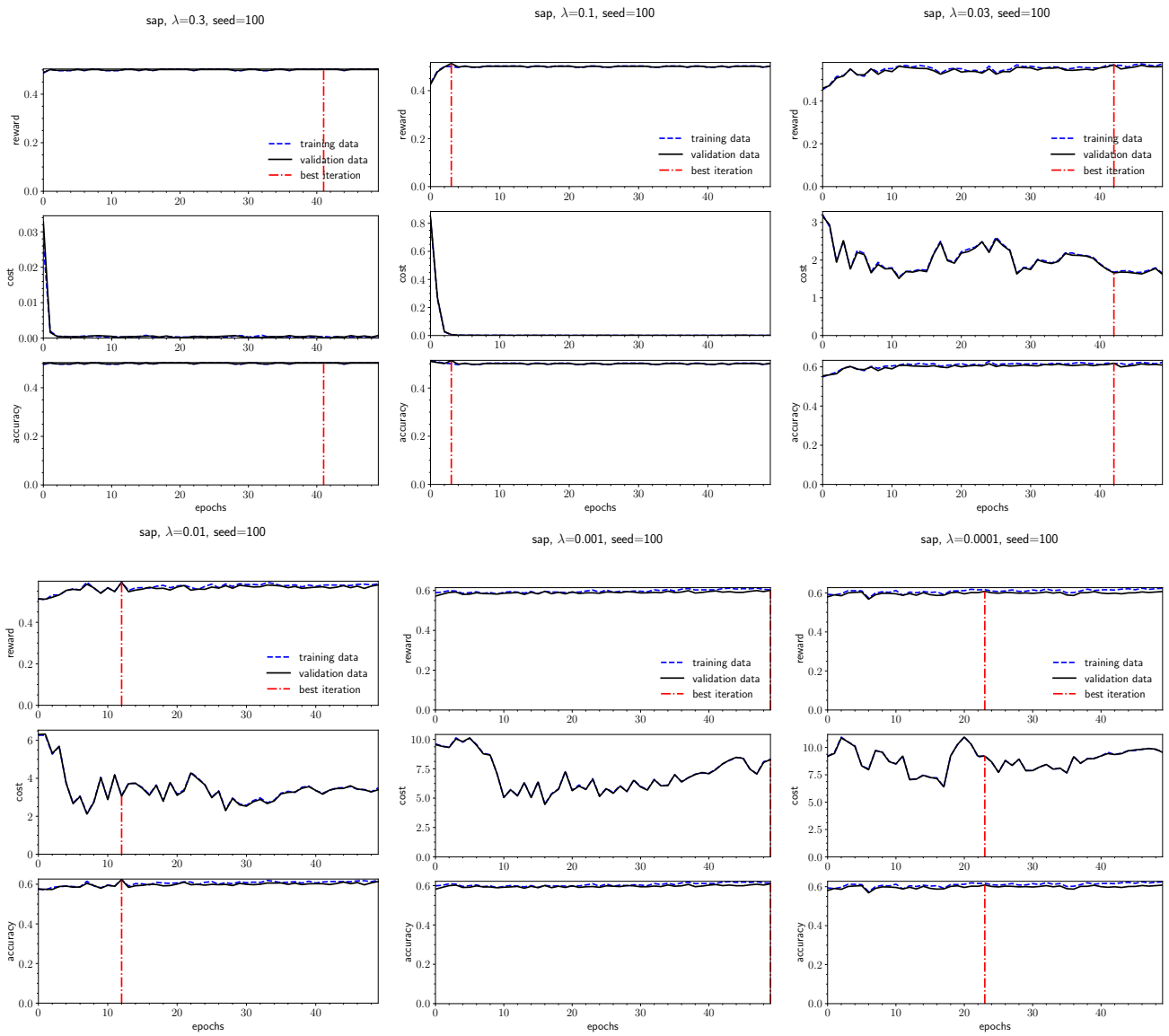


Figure D.13: Convergence graphs for *sap* dataset.

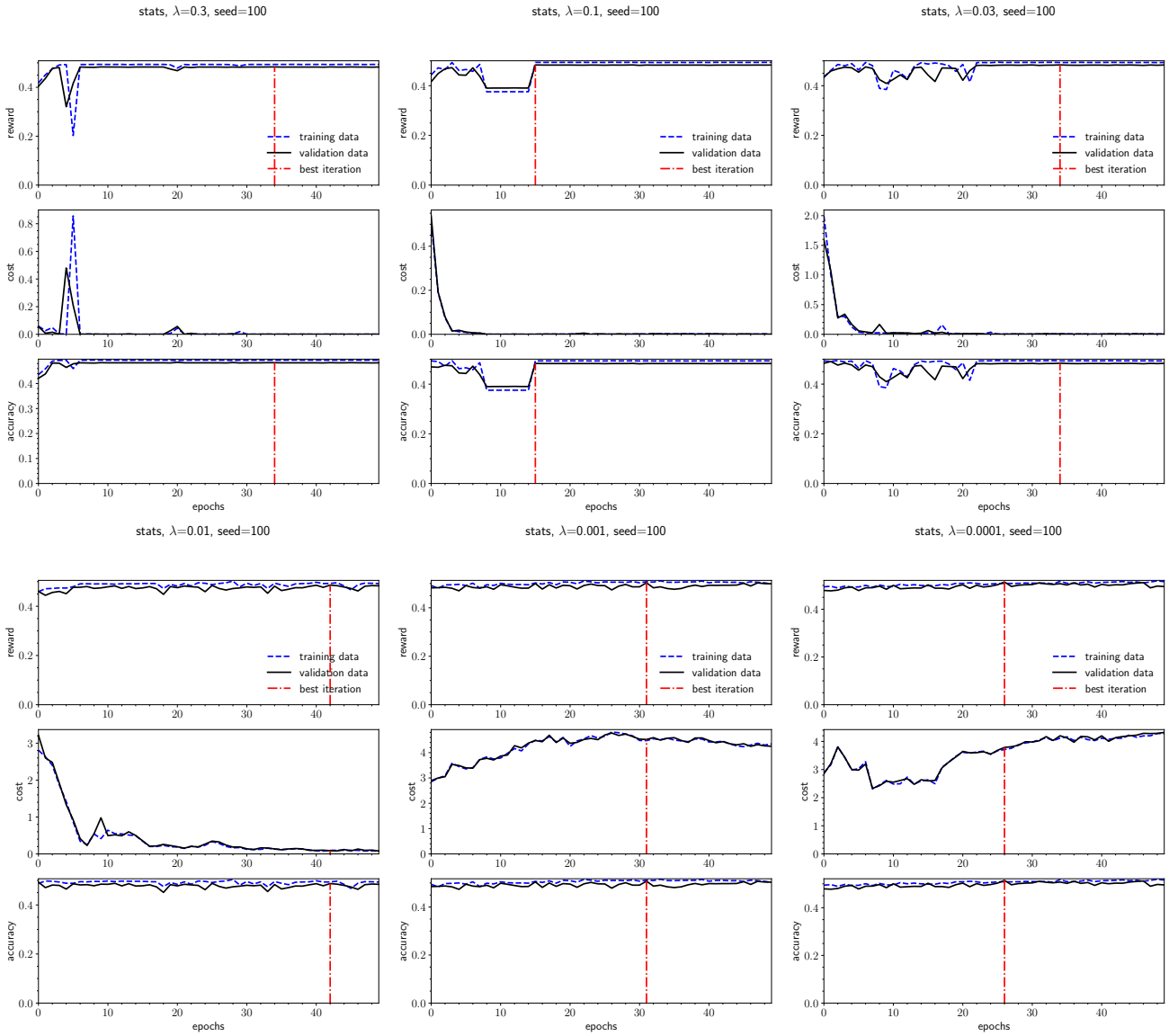


Figure D.14: Convergence graphs for *stats* dataset.

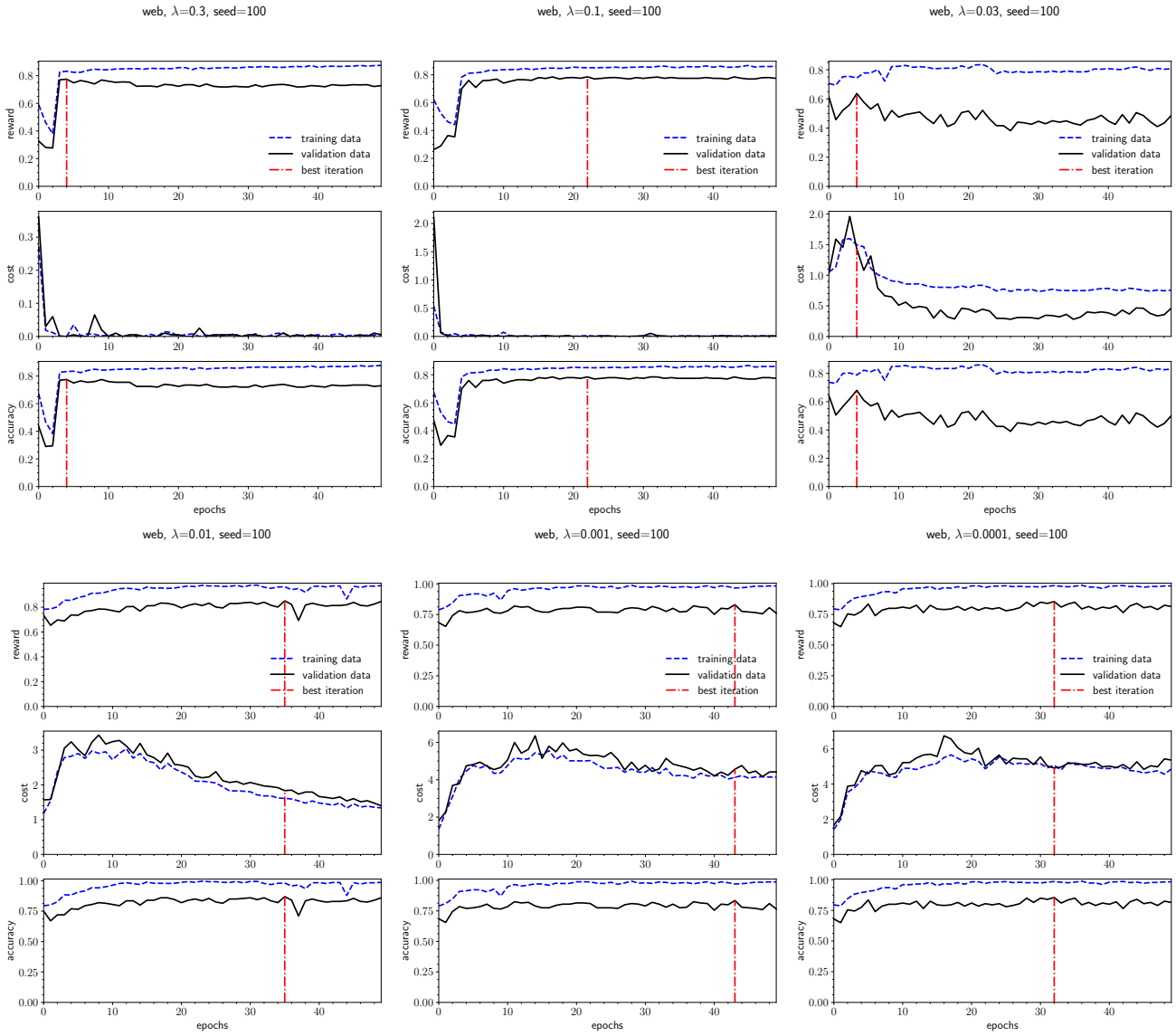


Figure D.15: Convergence graphs for *web* dataset.