Machine Learning (2020) 109:1587–1615
https://doi.org/10.1007/s10994-020-05874-8

Classification with costly features as a sequential decision-making problem

Jaromír Janisch[1] · **Tomáš Pevný**[1] · **Viliam Lisý**[1]

Received: 5 April 2019 / Revised: 29 January 2020 / Accepted: 17 February 2020 /
Published online: 28 February 2020

Abstract
This work focuses on a specific classification problem, where the information about a sample is not readily available, but has to be acquired for a cost, and there is a per-sample budget. Inspired by real-world use-cases, we analyze *average* and *hard* variations of a *directly specified* budget. We postulate the problem in its explicit formulation and then convert it into an equivalent MDP, that can be solved with deep reinforcement learning. Also, we evaluate a real-world inspired setting with sparse training datasets with missing features. The presented method performs robustly well in all settings across several distinct datasets, outperforming other prior-art algorithms. The method is flexible, as showcased with all mentioned modifications and can be improved with any domain independent advancement in RL.

Keywords Sequential classification · Costly features · Adaptive feature acquisition · Datum-Wise classification · Prediction on budget

1 Introduction

Classification with Costly Features (CwCF) is a family of classification problems with a cost of acquiring information. This cost can appear in many forms. Usually, it is about money or time, but it is present in any domain with limited resources. We view the problem as a sequential decision-making problem. At each step, based on the information acquired so far, the algorithm has to decide whether to acquire another piece of information (a *feature*) or to classify.

Editor: Hendrik Blockeel.

✉ Jaromír Janisch
jaromir.janisch@fel.cvut.cz

Tomáš Pevný
tomas.pevny@fel.cvut.cz

Viliam Lisý
viliam.lisy@fel.cvut.cz

[1] Artificial Intelligence Center, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic

Think about a doctor who is about to make a diagnosis for their patient. There are a number of examinations, tests or analysis which can be made, but each of them has a cost. As much as the doctor wants to make a reliable prediction, they are bound by their *average budget* they should not exceed. Naturally, patients with complicated diseases require more complicated and expensive tests, while trivial problems can be diagnosed with much fewer resources. Here, we use the medical domain only as an example, but there are several publications analyzing the use of CwCF in medicine, i.e., Peng et al. (2018) and Bayer-Zubek and Dietterich (2005).

As another motivating example, imagine an online service that analyzes computer files potentially infected with malware. The service is bound by a service-level agreement and has to provide a decision in a specified time, and this time cannot be exceeded. This is an example of a *hard budget*. The process can analyze the files in multiple ways and compute their features, and each computation takes a different, but known amount of time. The goal is to provide accurate predictions, while not violating the time constraint.

In other domains, different requirements arise. One domain contains a lot of missing data, other has imbalanced datasets. There can be a requirement of incorporating an existing classifier into the process. Misclassification errors can have outcomes with different impact, measured in the amount of lost resources. As we see, there are many variants of the CwCF problem. Techniques adapted for specific problems exist and it is difficult to modify these methods. In this article, we present a flexible reinforcement-learning based framework, that can work with all the mentioned instances. Should other needs arise, the method is easily modified to suit the problem. We mainly demonstrate our method in cases of average and hard budgets and also with missing features.

The power and generality of our method arises from the decoupling of the problem and the method itself. By using a general reinforcement learning algorithm, we are able to modify the problem specification, and the method will still provide a good result. The core of our algorithm is built on optimal methods, but we lose the guarantees by using function approximation (specifically, neural networks). Our method is also robust to hyperparameter selection, where the same set of hyperparameters usually works well across different domains and settings.

To the best of our knowledge, the presented method is the first that can work with both average and hard budgets, is flexible and robust. Formerly, CwCF with the average budget was approached with linear programming (Wang et al. 2014a), tree-based algorithms (Nan et al. 2016), gradient-based methods (Contardo et al. 2016) and recently reinforcement learning (Shim et al. 2018; Janisch et al. 2019). There are also several publications focusing on the hard budget problem—guided selection using a heuristic (Kapoor and Greiner 2005), and theoretical analyses (Cesa-Bianchi et al. 2011; Zolghadr et al. 2013). We present an overview of the related work in Sect. 6.

We directly build on recent work of Janisch et al. (2019), where the authors established a new state-of-the-art method in the average budget case. The authors proposed to solve the problem of minimizing expected classification loss, along with λ-scaled total cost:

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

where (x, y) are samples taken from the dataset $\mathcal{D}$, ℓ is a classification loss, λ is a trade-off parameter, y_θ is the classifier and z_θ returns the total cost of used features in the classification.

The definition only focuses on the average budget problem and it also introduces an unintuitive parameter λ. In the case the problem is well-specified, all the feature and

classification costs are precisely known and in the same units, solving the eq. (1) with $\lambda = 1$ will yield the optimal solution. However, in many cases, the user knows only the feature costs and desires a model that will achieve some trade-off between the accuracy and the cost or have a specified budget. In the case of eq. (1), the user has no option but to *try* different values and see whether the learned model corresponds to a targeted budget or not. This may require several runs of the algorithm and is inefficient. In this work we take a step back and propose the definition of the problem in its natural form. Given an explicit budget b, the problem for the average case is:

$$\min_{\theta} \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } \mathbb{E}\left[z_\theta(x)\right] \leq b \tag{2}$$

where the expectations are w.r.t. the distribution of the samples in the dataset. As we show in Sect. 2.4, definition (1) follows from (2) when using a Lagrangian framework and it allows us to derive an algorithm in which we remove the λ parameter from the user's control. In this case, the user directly sets a desired budget and the algorithm internally searches for a suitable λ. The method is based on an alternating optimization of the model's parameter θ and λ, similarly to Generative Adversarial Networks (Goodfellow et al. 2014); however, it is novel in the context of CwCF.

Next, we focus on the hard budget problem, which is defined as:

$$\min_{\theta} \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } z_\theta(x) \leq b, \forall x \tag{3}$$

Dulac-Arnold et al. (2011) showed that the minimization problem (1) could be transformed into an MDP formulation and solved through standard reinforcement learning techniques. The approach was later improved by Janisch et al. (2019) with deep learning. We follow this approach and modify the algorithm for both (2) and (3). In the case of the average budget, we use the mentioned alternating optimization. In the case of a hard budget, we show that simple modification to the MDP definition enforces the hard constraints. In exact settings, the method would yield an optimal solution and it only lacks guarantees due to the used function approximation.

In the last setting, we focus on the problem of missing features. We demonstrate that a simple modification of the algorithm performs comparably to a widely used MICE imputation method (Azur et al. 2011).

For each setting, we provide an experimental evaluation on several distinct datasets and show that the method achieves a state-of-the-art performance. Also, it is flexible, robust, easy to use and can be improved with any domain independent advance in RL itself.

The article is structured as follows. First, in Sect. 2, we present the main ideas for solving various definitions of the problem, along with the problem of missing features. We explain the implementation details in Sect. 3. In Sects. 4 and 5, we describe the performed experiments and their results. Section 6 summarizes the related work.

2 Problem variations

Before we delve into technical details, we present an overview of what CwCF is and how we view it. Then we start with the common notation which will be used for the rest of the article. In separate sections, we present the algorithms for the different cases. We start with the definition (1), an average budget case where the budget is specified indirectly through a

parameter λ. Next, still working with the average budget, we present a reformulation of the problem with a directly specified budget b and solving it with the Lagrangian framework. Then we modify the framework to work with hard budgets. Lastly, we focus on a problem of missing features, which appear in many real-world situations.

2.1 Classification with costly features

First, we would like to stress the *sequential* nature of the problem. Each sample is treated separately and the model sequentially selects features, one by one (see Fig. 1). Eventually, a decision to classify is made, and the model outputs a class prediction. Each decision is based on the knowledge acquired so far, hence different samples will result in completely different sequences of features and predictions. This important fact differentiates CwCF from feature selection methods, where the same subset of features is selected for each sample.

In real-world scenarios, there are many small modifications to the problem formulations. However, the presented method is very flexible and can be easily modified. For example, the prior knowledge can be included in the sample before starting the process (e.g., when a patient comes with known medical history). Multiple features can be grouped together and represented as one macro-feature. Different misclassifications can be treated with different weights through a particular choice of the loss function ℓ. The method can also make use of an external and independently pretrained classifier and automatically redirect samples if it is advantageous [shown by Janisch et al. (2019)].

2.2 Common notation

We assume that a sample can be represented as a real-valued vector, where each of its members we call *features*. Here we assume a feature is one real number, but presented algorithms can be trivially modified in the case of multi-dimensional features.

Let's start with common notation, which will be used for the rest of the article. Let $(x, y) \in \mathcal{D}$ be a sample drawn from a data distribution $\mathcal{D}$. Vector $x \in \mathcal{X} \subseteq \mathbf{R}^n$ contains feature values, where x_i is a value of feature $f_i \in \mathcal{F} = \{f_1, ..., f_n\}$, n is the number of features, and $y \in \mathcal{Y}$ is a class. Let $c : \mathcal{F} \to \mathbf{R}^+$ be a function mapping a feature f into its real-valued cost $c(f)$. For convenience, let's overload c to also accept a set of features and return the summation of their individual costs: $c(\mathcal{F}') = \sum_{f \in \mathcal{F}'} c(f)$. Let $b \in \mathbf{R}^+$ be the allocated budget per sample.

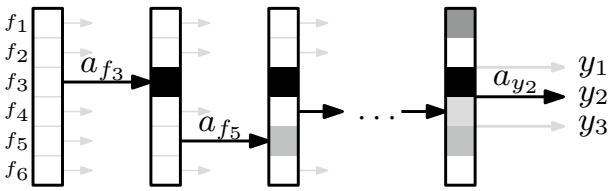


Fig. 1 Illustration of the sequential process with a sample with 6 features $(f_1, ..., f_6)$ and three classes (y_1, y_2, y_3). Feature values are acquired sequentially (actions a_{f_3}, a_{f_5}, ...) before making a classification (a_{y_2}). The particular decisions are influenced by the observed values—the model chooses different actions for different samples

Our method selects features *sequentially*, and is composed of a neural network with parameters θ. However, for convenience, we define a pair of functions (y_θ, z_θ) to represent the whole *process* of classifying one sample. In this notation, $y_\theta : \mathcal{X} \to \mathcal{Y}$ represents the classification output at the end of the process and $z_\theta : \mathcal{X} \to \mathbf{R}$ represents the total cost of all features acquired during the process.

2.3 Average budget with trade-off parameter λ

As we have seen in the medical example in the introduction, in some domains the user wants to target an average budget per sample. Let's start by writing the problem definition one more time:

$$\min_\theta \mathbb{E}\left[\ell(y_\theta(x), y) + \lambda z_\theta(x)\right] \qquad \text{(1 revisited)}$$

Here, the user has to specify a trade-off parameter λ which will result in an a priori unknown average budget. The approach is to create an MDP, where samples are classified in separate episodes and the expected reward R per episode is:

$$R = -\mathbb{E}\left[\ell(y_\theta(x), y) + \lambda z_\theta(x)\right] \tag{4}$$

Standard reinforcement learning techniques are then used to optimize this reward, thus solving (1). Illustration of the MDP is in Fig. 2.

We model the environment as a deterministic MDP with full information, which is easily implemented. The agent, however, solves a stochastic MDP which is created when you remove some of the information (namely, the unobserved feature values). Formally, the MDP consists of states $\mathcal{S}$, actions $\mathcal{A}$, transition function t and reward function r. State $s = (x, y, \bar{\mathcal{F}}) \in \mathcal{S}$ represents a sample (x, y) and currently selected set of features $\bar{\mathcal{F}}$. The agent receives only the selected parts of x without the label. Action $a \in \mathcal{A} = \mathcal{Y} \cup \mathcal{F}$ is either a classification action from $\mathcal{Y}$ that terminate the episode and the agent receives a reward of $-\ell(a, y)$, or a feature selecting action from $\mathcal{F}$ that reveals the corresponding value of x and the agent receives a reward of $-\lambda c(a)$. The set of available feature selecting actions is limited to features not yet selected. Reward and transition functions are specified as:

$$r(s,a) = \begin{cases} -\lambda c(a) & \text{if } a \in \mathcal{A}_f \\ -\ell(a,y) & \text{if } a \in \mathcal{A}_c \end{cases} \quad ; \quad t(s,a) = \begin{cases} (x, y, \bar{\mathcal{F}} \cup a) & \text{if } a \in \mathcal{A}_f \\ \mathcal{T} & \text{if } a \in \mathcal{A}_c \end{cases}$$

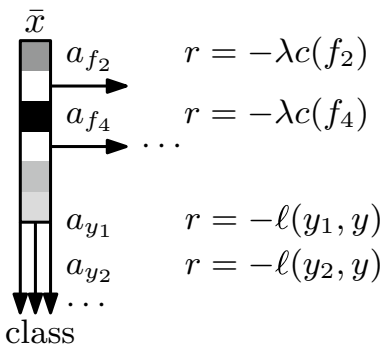


Fig. 2 The MDP. The agent sees a masked sample $\bar{x}$. At each step it chooses from feature-selecting actions (a_f) or classification actions (a_y) and receives a corresponding reward (either the cost of the selected feature or the classification loss)

For real datasets, there may be a specific cost for misclassification, expressed in the amount of lost resources. If such information is not available, we propose to use a binary classification loss ℓ:

$$\ell(\hat{y}, y) = \begin{cases} 0 & \text{if } \hat{y} = y \\ 1 & \text{if } \hat{y} \neq y \end{cases}$$

2.4 Average budget with specific target *b*

As we already mentioned, a manual specification of an unintuitive parameter λ, as used in the previous section, is not convenient. In real-world applications, the user wants to directly specify a budget b. Let's review the definition of the problem:

$$\min_{\theta} \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } \mathbb{E}\left[z_\theta(x)\right] \leq b \tag{2 revisited}$$

This constrained optimization problem can be transformed into an alternative Lagrangian form and solved with maxmin optimization. First, let's derive the Lagrangian, where $\lambda \in \mathbf{R}$ denotes a Lagrange multiplier:

$$L(\theta, \lambda) = \mathbb{E}\left[\ell(y_\theta(x), y) + \lambda(z_\theta(x) - b)\right] \tag{5}$$

The multiplier λ plays a similar role as in the previous approach. However, here it is a variable of our algorithm and is *not* specified by the user. The saddle point theorem in Bertsekas (1999, prop. 5.1.6) says that there exist parameters θ, λ which are optimal in (2) and are a solution of the following problem:

$$\max_{\lambda \geq 0} \min_{\theta} L(\theta, \lambda) \tag{6}$$

Inspired by an approach of Chow et al. (2017), we propose to iteratively perform gradient ascent in λ and descend in θ. For fixed θ, optimizing λ is easy, since the gradient is $\nabla_\lambda L = \mathbb{E}\left[z_\theta(x) - b\right]$. However, optimizing θ is not straightforward, since the model (y_θ, z_θ) is neither differentiable nor continuous (it is a sequential process). Let's look at the problem when λ is fixed, that is, minimizing Lagrangian L w.r.t. parameters θ:

$$\min_{\theta} L(\theta, \lambda) = \min_{\theta} \mathbb{E}\left[\ell(y_\theta(x), y) + \lambda z_\theta(x)\right] - \lambda b \tag{7}$$

In the search for optimal parameters θ, we can omit the term λb since it does not influence the solution. Note that the problem is then equal to (1) and thus we can directly apply RL through the method with fixed λ. However, we will only take small steps in θ, effectively estimating and following the gradient $\nabla_\theta L$. The summary can be seen in Algorithm 1.

Algorithm 1 Training with target budget b

$\lambda \leftarrow 0$, randomly initialize θ
loop
  Update λ by taking a gradient step with $\nabla_\lambda L = \mathbb{E}\left[z_\theta(x) - b\right]$ (maximize L)
  Update θ using RL (minimize L; Algorithms 2 and 3)
end loop

A similar approach was evaluated in work of Chow et al. (2017), where the authors used the Lagrangian framework together with policy gradients to solve a constrained problem and proved convergence. Note that for an optimal solution, a stochastic policy may be needed. Our method is based on Q-learning, which works only with deterministic policies and this can result in oscillations around the stable point. However, it is possible to detect when this happens, use it as a terminating condition and simply select the best-performing model satisfying the constraints.

2.5 Hard budget

In some domains, the resources are strictly restricted by a budget b per sample. The problem definition changes to:

$$\min_\theta \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } z_\theta(x) \leq b, \forall x \qquad \text{(3 revisited)}$$

Similarly to the previous case, we can construct an MDP where the expected reward per sample is $R = -\mathbb{E}\left[\ell(y_\theta(x), y)\right]$ and the episodes are restricted to end when the budget is depleted. Again, by solving this MDP with standard reinforcement learning techniques, we retrieve the solution to (3).

First, we change the reward function such that the costs of different features are ignored:

$$r(s,a) = \begin{cases} 0 & \text{if } a \in \mathcal{A}_f \\ -\ell(a,y) & \text{if } a \in \mathcal{A}_c \end{cases}$$

Second, we restrict the set of available feature-selecting actions at each step to those, which do not exceed the specified budget. That is, $a \in \mathcal{F}$ is available only if $c(\bar{\mathcal{F}} \cup a) \leq b$. This way, the environment itself enforces the constraint.

2.6 Missing features

In a lot of domains, there is a large amount of data that can be used to train our method. However, the data is often not complete; i.e., in the medical domain, patients are typically sent only to a few examinations before the diagnosis is made. When using past data, only this limited information will be present in the training set.

Here we present a principled method to deal with the issue, again by modifying our original algorithm. During training, a feature-selecting action is available only if the corresponding feature is present and the updates (see Eq. 10) are made only with the estimates of available actions. We experimented with another variation, where estimates of all actions (even for unavailable features) were used. Intuitively, it corresponds to a case

where we train with sparse data, but at test time, we have a full set. In our experiments, this approach underperformed the first one, hence we do not report it.

3 Method

In this section, we describe mainly the implementation of the reinforcement learning algorithm. Because we operate with large datasets with continuous features, the tabular approach is not feasible. Therefore, we employ neural networks as function approximators and use recent RL techniques. We experimented with a variety of different methods and found that incorporating recent insights from deep RL community is essential for the method to be stable, robust and perform well. After evaluating the implementation complexity and reported performance, we implemented Double Dueling DQN with Retrace as the RL solver. In the first part, we describe these RL methods. In the following part, we focus on the algorithm itself, and how it was implemented.

3.1 Deep RL background

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, t, r, \gamma)$, where $\mathcal{S}$ represent state space, $\mathcal{A}$ a set of actions, $t(s, a)$ is a transition function returning a distribution of states after taking action a in state s, $r(s, a, s')$ is a reward function and γ is a discount factor. In Q-learning, one seeks the optimal function Q^*, representing the expected total discounted reward for taking an action a in a state s and then following the optimal policy. It satisfies the Bellman equation:

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim t(s,a)} \left[r(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right] \tag{8}$$

A neural network with parameters θ takes a state s and outputs an estimate $Q^\theta(s, a)$, jointly for all actions a. It is optimized by minimizing MSE between the both sides of Eq. (8) for transitions (s_t, a_t, r_t, s_{t+1}) empirically experienced by an agent following a greedy policy $\pi_\theta(s) = \text{argmax}_a Q^\theta(s, a)$. Formally, we are looking for parameters θ by iteratively minimizing the loss function ℓ_θ, for a batch of transitions $\mathcal{B}$:

$$\ell_\theta(\mathcal{B}) = \mathop{\mathbb{E}}_{(s_t,a_t,r_t,s_{t+1}) \in \mathcal{B}} \left[q_t - Q^\theta(s_t, a_t) \right]^2 \tag{9}$$

where q_t is regarded as a constant when differentiated, and is computed as:

$$q_t = \begin{cases} r_t & \text{if } s_{t+1} = \mathcal{T} \\ r_t + \max_a \gamma Q^\theta(s_{t+1}, a) & \text{otherwise} \end{cases} \tag{10}$$

As the error decreases, the approximated function Q^θ converges to Q^*. However, this method proved to be unstable in practice (Mnih et al. 2015). Now, we briefly describe the techniques used in this work that stabilize and speed-up the learning.

Deep Q-learning (Mnih et al. 2015) includes a separate *target network* with parameters ϕ, which follow parameters θ with a delay. Here we use the method of Lillicrap et al. (2016), where the weights are regularly updated with expression $\phi := (1 - \rho)\phi + \rho\theta$, with some parameter ρ. The slowly changing estimate Q^ϕ is then used in q_t, when $s_{t+1} \neq \mathcal{T}$:

$$q_t = r_t + \max_a \gamma Q^{\phi}(s_{t+1}, a) \tag{11}$$

Double Q-learning (Van Hasselt et al. 2016) is a technique to reduce bias induced by the *max* in Eq. (10), by combining the two estimates Q^{θ} and Q^{ϕ} into a new formula for q_t, when $s_{t+1} \neq \mathcal{T}$:

$$q_t = r_t + \gamma Q^{\phi}(s_{t+1}, \operatorname*{argmax}_a Q^{\theta}(s_{t+1}, a)) \tag{12}$$

In the expression, the maximizing action is taken from Q^{θ}, but its value is estimated with the target network Q^{ϕ}.

Dueling Architecture (Wang et al. 2016) uses a decomposition of the Q-function into two separate value and advantage functions. The architecture of the network is altered so that it outputs two estimates $V^{\theta}(s)$ and $A^{\theta}(s, a)$ for all actions a, which are then combined to a final output $Q^{\theta}(s, a) = V^{\theta}(s) + A^{\theta}(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A^{\theta}(s, a')$. When training, we take the gradient w.r.t. the final estimate Q^{θ}. By incorporating baseline V^{θ} across different states, this technique accelerates and stabilizes training.

Retrace (Munos et al. 2016) is a method to efficiently utilize long traces of experience with truncated importance sampling. We store generated trajectories into an experience replay buffer (Lin 1993) and utilize whole episode returns by recursively expanding Eq. (8). The stored trajectories are off the current policy and a correction is needed. For a sequence $(s_0, a_0, r_0, \ldots, s_n, a_n, r_n, \mathcal{T})$, we implement Retrace together with Double Q-learning by replacing q_t with

$$q_t = r_t + \gamma \mathop{\mathbb{E}}_{a \sim \pi_{\theta}(s_t)} \left[Q^{\phi}(s_{t+1}, a) \right] + \gamma \bar{\rho}_{t+1} \left[q_{t+1} - Q^{\phi}(s_{t+1}, a_{t+1}) \right] \tag{13}$$

where we define $Q^{\phi}(\mathcal{T}, \cdot) = 0$ and $\bar{\rho}_t = \min(\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}, 1)$ is a truncated importance sampling between exploration policy μ that was used when the trajectory was sampled and the current policy π. The truncation is used to bind the variance of product of multiple important sampling ratios for long traces. We allow the policy π_{θ} to be stochastic—at the beginning, it starts close to the sampling policy μ but becomes increasingly greedy as the training progresses. It prevents premature truncation in the Eq. (13) and we observed faster convergence. Note that all q_t values for a whole episode can be calculated in $\mathcal{O}(n)$ time. Further, it can be easily parallelized across all episodes.

3.2 Training method

In this section, we describe the method of training the RL agent. At every step, the agent receives only an observation $o = \{(x_i, f_i) \mid \forall f_i \in \bar{\mathcal{F}}\}$, that is, the selected parts of x without the label. The observation o is mapped into a tuple $(\bar{x}, m)$:

$$\bar{x}_i = \begin{cases} x_i & \text{if } f_i \in \bar{\mathcal{F}} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad m_i = \begin{cases} 1 & \text{if } f_i \in \bar{\mathcal{F}} \\ 0 & \text{otherwise} \end{cases}$$

Vector $\bar{x} \in \mathbf{R}^n$ is a masked vector of the original x. It contains values of x which have been acquired and zeros for unknown values. Mask $m \in \{0, 1\}^n$ is a vector denoting whether a specific feature has been acquired, and it contains 1 at a position of acquired features, or 0. The combination of $\bar{x}$ and m is required so that the model can differentiate between a

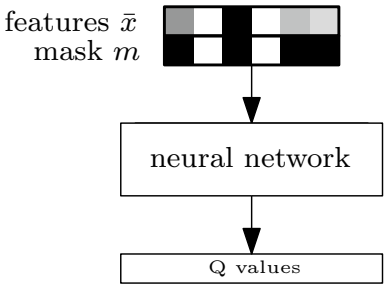


Fig. 3 The architecture of the model. The input layer consists of the feature vector $\bar{x}$ concatenated with the binary mask m, followed by a feed-forward neural network (FFNN). Final fully connected layer jointly outputs Q-values for both classification and feature-selecting actions

feature not present and observed value of zero. Each dataset is normalized with its mean and standard deviation and because we replace unobserved values with zero, this corresponds to the mean-imputation of missing values.

In our experiments, we use a feed-forward neural network, which accepts concatenated vectors $\bar{x}$, m and outputs Q-values jointly for all actions. There are three fully connected hidden layers, each followed by the ReLU non-linearity, where the number of neurons in individual layers change depending on the used dataset. The overview is shown in Fig. 3.

A set of environments with samples randomly drawn from the dataset are simulated and the experienced trajectories are recorded into the experience replay buffer. After each action, a batch of trajectories $\mathcal{B}$ is taken from the buffer, so that the total number of individual transitions matches the defined batch size. The transitions are then optimized upon with Adam (Kingma and Ba 2015), with Eqs. (9), (13). The gradient is normalized before back-propagation if its norm exceeds 1.0. The target network is updated after each step. Overview of the algorithm and the environment simulation is in Algorithm 2 and 3.

Algorithm 2 RL training

Randomly initialize parameters θ
Pretrain the classifier part $Q^\theta(s, a \in \mathcal{Y})$ with random states
Initialize target network $\phi \leftarrow \theta$
Initialize environments $\mathcal{E}$ with $(x, y, \emptyset) \in (\mathcal{X}, \mathcal{Y}, \wp(\mathcal{F}))$
Initialize replay buffer $\mathcal{M}$ with a random agent
loop
  for all $e \in \mathcal{E}$ **do**
    Simulate one step with ϵ-greedy policy π_θ:

$$a = \pi_\theta(s); \quad s', r = \textsc{step}(e, a)$$

    if $s' = \mathcal{T}$ (the episode terminated) **then**
      Store trajectory $(s_0, a_0, r_0, s_1, a_1, r_1, \ldots, \mathcal{T})$ of e into circular buffer $\mathcal{M}$
    end if
  end for

  Initialize batch $\mathcal{B} \leftarrow \emptyset$
  Sample random trajectories T from $\mathcal{M}$
  (so that the transition count matches the batch size)
  for trajectories $(s_{i,0}, a_{i,0}, r_{i,0}, \ldots, s_{i,n}, a_{i,n}, r_{i,n}, \mathcal{T}) \in T$ **do**
    for steps $t \in \{n \ldots 0\}$ **do**
      Compute targets $q_{i,t}$ according to eq. (13)
      Clip $q_{i,t}$ with maximum of 0
      Compute single-step MSE $e_{i,t} = [q_{i,t} - Q^\theta(s_{i,t}, a_{i,t})]^2$
      $\mathcal{B} \leftarrow \mathcal{B} \cup e_{i,t}$
    end for
  end for
  Perform one step of gradient descent on ℓ_θ w.r.t. θ, $\ell_\theta(\mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{e_{i,t} \in \mathcal{B}} e_{i,t}$
  Update target network parameters $\phi := (1 - \rho)\phi + \rho\theta$
end loop

Algorithm 3 Environment simulation

Operator $\odot$ marks the element-wise multiplication.
function $\textsc{Step}(e \in \mathcal{E}, a \in \mathcal{A})$
  if $a \in \mathcal{A}_c$ **then**

$$r = \begin{cases} 0 & \text{if } a = e.y \\ -1 & \text{if } a \neq e.y \end{cases}$$

    Reset e with a new sample $(x, y, \emptyset)$ from a dataset
    Return $(\mathcal{T}, r)$
  else if $a \in \mathcal{A}_f$ **then**
    Add a to set of selected features: $e.\bar{\mathcal{F}} = e.\bar{\mathcal{F}} \cup a$
    Create mask $m_i = 1$ if $f_i \in \bar{\mathcal{F}}$ and 0 otherwise
    Return $((e.x \odot m, m), -\lambda c(a))$
  end if
end function

Because all rewards are non-positive, the whole Q-function is also non-positive. We use this knowledge and clip the q_t value so that it is at most 0. Without this bound, the

predicted values sometimes rose to infinity, due to the *max* used in Q-learning. The definition of the reward function also results in optimistic initialization. A neural network with initial weights tends to output small values around zero. Effectively, the model tends to overestimate the real Q-values, which has a positive effect on exploration.

We do not use a discount factor ($\gamma = 1$), because we want to recover the original objectives. We use ϵ-greedy policy that behaves greedily most of the time, but picks a random action with a probability ϵ. The unavailable actions are ignored; in the greedy selection, the algorithm chooses an action with the highest Q-value among the available actions. Exploration rate ϵ starts at a defined initial value and it is linearly decreased over time to its minimum value.

Classification actions $\mathcal{A}_c$ are terminal and their Q-values do not depend on any following states. Prior to the main method, we pretrain the part of the network $Q^\theta(s, a)$, for classification actions $a \in \mathcal{A}_c$ with batches of randomly sampled states. We randomly pick samples x from the dataset and generate masks m. The values m_i follow the Bernoulli distribution with probability p. As we want to generate states with a different amount of observed features, we randomly select $\sqrt[3]{p} \sim \mathcal{U}(0, 1)$ for different states. The resulting distribution of states is shifted towards the initial state with no observed features. The main algorithm starts with accurate classification predictions and this technique has a positive effect on the speed of the training process.

In the case of the specified budget b, we also optimize the multiplier λ. In our experiments, we found that a simple gradient ascent with momentum works best. The learning rate schedule for both parameters θ and λ is exponential, in fixed steps, up to some minimal value.

4 Experiment setup

Here we describe the methodology, datasets, hyperparameters and methods we compare to in our experiments.

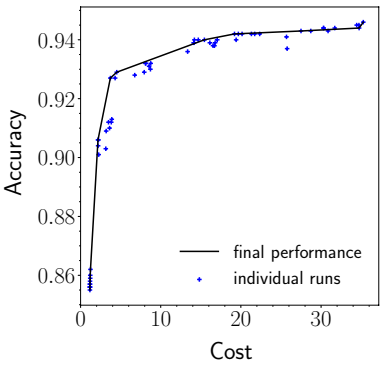


Fig. 4 Illustrative performance of different trained models and their trade-offs, measured on the cost-accuracy plane. *Validation set* is used to select the best performing models, hence the individual runs can sometimes exceed the final performance, which is reported on the test set

Table 1 Used datasets

Dataset	# Features	# Classes	Train size (k)	Val. size (k)	Test size (k)	Costs
Miniboone	50	2	45	19	65	U
Forest	54	7	200	81	300	U
Forest-2	54	2	200	81	300	U
Cifar	400	10	40	10	10	U
Cifar-2	400	2	40	10	10	U
Mnist	784	10	50	10	10	U
Diabetes	45	3	64	14	14	V

The cost is either uniform (U) or variable (V) across features

4.1 Evaluation metric

It is difficult to compare algorithms when we essentially optimize for two objectives - cost and accuracy. Thus, we adopt the following procedure. We train multiple instances of a particular algorithm, with varying parameters (this involves different settings of λ, budget b and seeds). The exact number of instances differs across datasets, settings and algorithms, but is comparable, with median of 20. In the cost-accuracy plane, we use the *validation set* to select the best performing model instances, which form a convex hull over all trained models. As an example, see Fig. 4, where we show several trained models and the selected ones. Note that because we select the best points on the validation set, occasionally some points can be higher that the final curve. For the final metric, we use the normalized area under this curve. By normalization we mean division by the area of the whole cost-accuracy plane, such that the best value is 1.0 (higher is better). We assume that, for each dataset, all models can achieve prior accuracy with no features and also the maximal accuracy of a particular model with all features.

4.2 Baseline method

We design a simple baseline method to compare with. First, we use a feature selection technique to select a *fixed order* of features, sorted from most important to least. Then, we iteratively add features, according to the list, and train separate neural network based classifiers. The resulting performance is visualized at the cost-accuracy graph as usual. Note that this baseline can be compared both to average and hard budget methods since for every budget, the set of used features is fixed. More specifically, we use Recursive Feature Elimination (Guyon et al. 2002) together with Ridge classifier (Hoerl and Kennard 1970) to select the feature order. The size of the neural network is comparable to the neural network used in the main method for a particular dataset.

4.3 Used datasets

In the following sections, we use several datasets, information about which is summarized in Table 1. They were obtained from public sources (Lichman 2013; Krizhevsky and Hinton 2009) and the Diabetes dataset was obtained from the authors of prior work (Kachuee et al. 2019). For datasets where there are no explicit costs, we use uniform costs for all

features. The Miniboone dataset is small and easy, from the classification perspective, and it is suitable for fast experimenting and evaluation. The Forest dataset contains categorical features (several features are one-hot encoded into multiple others) and many samples, making it hard to achieve good performance. The Cifar and Mnist datasets are challenging multi-class image recognition datasets, where we treat all pixels as separate features. We could leverage convolutions for the image datasets, but to make a fair comparison with other algorithms, we treat all datasets the same—as with features with no clear structure. The Forest-2 and Cifar-2 datasets are binarized version of the original datasets, where the classes were merged into two. The Diabetes dataset contains real-world medical data with expert-valued feature costs and we use its balanced and mean imputed version.

4.4 Compared algorithms

Let's review the algorithms we compare to in our experiments. We choose the following algorithms because they are recent, report impressive results and have their source code published on-line.

The first method we use is Adapt-Gbrt (*agbrt*) (Nan and Saligrama 2017), which is a random forest (RF) based algorithm that uses an external pretrained model (HPC). It jointly learns a gating function and Low-Prediction Cost model (LPC) that adaptively approximates HPC in regions where it suffices for making accurate predictions. The gating function then redirects the samples to either use HPC or LPC. The published implementation is able to work only in datasets with two classes. The hyperparameters were inspired by the original paper: Trade-off parameter $\gamma \in \{0.01, 0.1, 1.0, 10, 100, 1000\}$, learning-rate in $\{0.5, 1.0\}$, $P_{full} \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, trees depth 4 and number of trees 100 in the Miniboone dataset and 500 in the Forest dataset. We use RBF-SVM as the HPC model and initialize LPC model with GreedyMiser (Xu et al. 2012). For each combination of the described parameters, we perform one run. We aggregate all results and proceed according to the described metric.

The second method we use is Budget-Prune (*bprune*) (Nan et al. 2016), which is an algorithm that prunes an existing RF using linear programming to optimize for the cost vs. accuracy trade-off. First, we create a RF with BudgetRF algorithm (Nan et al. 2015) with 40 trees in Miniboone, Forest and Diabetes and 80 trees (we did not observe better performance with more trees) in Cifar and Mnist. Then we prune the resulting RF with Budget-Prune, with at least 12 different trade-off settings (more where necessary). The results are processed according to the evaluation metric.

In hard budget setting, we compare to recent heuristic-driven approach by Kachuee et al. (2019), called Opportunistic Learning (*oplearn*). In this algorithm, an auxiliary reward is defined as a change in prediction uncertainty, when some feature is added. Two separate networks are trained—one estimating class probabilities, the other predicting the auxiliary reward. During test-time, the features are greedily acquired according to the predicted reward, and classification is made when the target budget is reached. The method uses a heuristic that lacks the theoretical ground (in contrast with our method where we directly optimize the Eq. 1), but the experimental results indicate that it performs well. In Opportunistic Learning, an immediate reward is predicted ($\gamma = 0$), because of which the model loses the capacity to predict into the future. Nevertheless, the reported performance was impressive, hence we selected the method for comparison. We use a neural network with a comparable amount of parameters to our method. Because of the way this algorithm works,

Table 2 (a) algorithm-level parameters; (b) dataset-specific parameters

Symbol	Description	Value
(a) Global parameters		
$\lvert\mathcal{E}\rvert$	Number of parallel environments	1000
	Maximum number of steps	$100 \times$ ep_len
γ	Discount-factor	1.0
Retrace-λ	Retrace parameter λ	1.0
ρ	Target network update factor	0.1
$\lvert\mathcal{B}\rvert$	Number of steps in batch	50k
$\lvert\mathcal{M}\rvert$	Number of episodes in memory	40k
$\epsilon_{\text{-start}}$	Starting exploration	1.0
$\epsilon_{\text{-end}}$	Final exploration	0.1
$\eta_{\text{-start}}$	Starting η-greediness of target policy π	0.5
$\eta_{\text{-end}}$	Final η-greediness of target policy π	0.0
ϵ_{steps}	Length of exploration phase	$2 \times$ ep_len
LR-pretrain	Pre-training learning-rate	1×10^{-3}
LR-start	Initial learning-rate	5×10^{-4}
LR-min	Minimal learning-rate	5×10^{-7}
LR-update	Number of steps between learning-rate updates	$10 \times$ ep_len
LR-scale	Learning-rate multiplier	0.5
for the specific average budget		
λ-LR-start	Initial λ learning-rate	1×10^{-1}
λ-LR-min	Minimal λ learning-rate	1×10^{-4}
λ-LR-update	Number of steps between λ learning-rate updates	$10 \times$ ep_len
λ-LR-scale	λ learning-rate multiplier	0.5
λ-update	Number of steps between updates of λ	$0.1 \times$ ep_len

Table 2 (continued)

Dataset	Model size	ep_len	Specific
(b) Dataset parameters			
Mnist†	512	10k	$\lvert\mathcal{M}\rvert = 10\text{k}, \text{LR-pretrain} = 2 \times 10^{-5}, \text{LR-start} = 10^{-5}$
Cifar†	512	10k	$\lvert\mathcal{M}\rvert = 10\text{k}, \text{LR-pretrain} = 2 \times 10^{-5}, \text{LR-start} = 10^{-5}$
Forest	256	10k	
Miniboone	128	1k	
Diabetes	128	100	

† The replay size is lowered due to memory constraints

we train a single model for each dataset. It is then queried with different budgets to assess its accuracy. We then construct the convex hull above these points.

4.5 Methodology

All evaluated algorithms include a λ like trade-off parameter or a defined budget, which we sweep across different values and run the algorithms several times, with different seeds. We use the evaluation method described in Sect. 4.1 to present the results.

As for our method, we let it run for a pre-defined number of steps, according to the Algorithm 2. In one step, all the parallel environments advance for one step, a batch is sampled from the memory and a gradient step in θ is taken. For each dataset, we define a number of steps that constitute an epoch (*ep_len*; 100, 1000 or 10k steps). Several other parameters are dependent on the epoch length; namely the length of the exploration phase, the learning rate schedule and the frequency of λ updates (in the case of specific average budget). Also, for each dataset, we heuristically estimate the size of the neural network (NN) by training NN based classifiers with number of neurons selected from $\{64, 128, 256, 512\}$ in three layers with ReLu activation. We choose the lowest size that performs well on the task, without excess complexity. The hyperparameters stay the same across all versions of our algorithm, clearly featuring its robustness. Table 2 presents all used parameters.

5 Experiment results

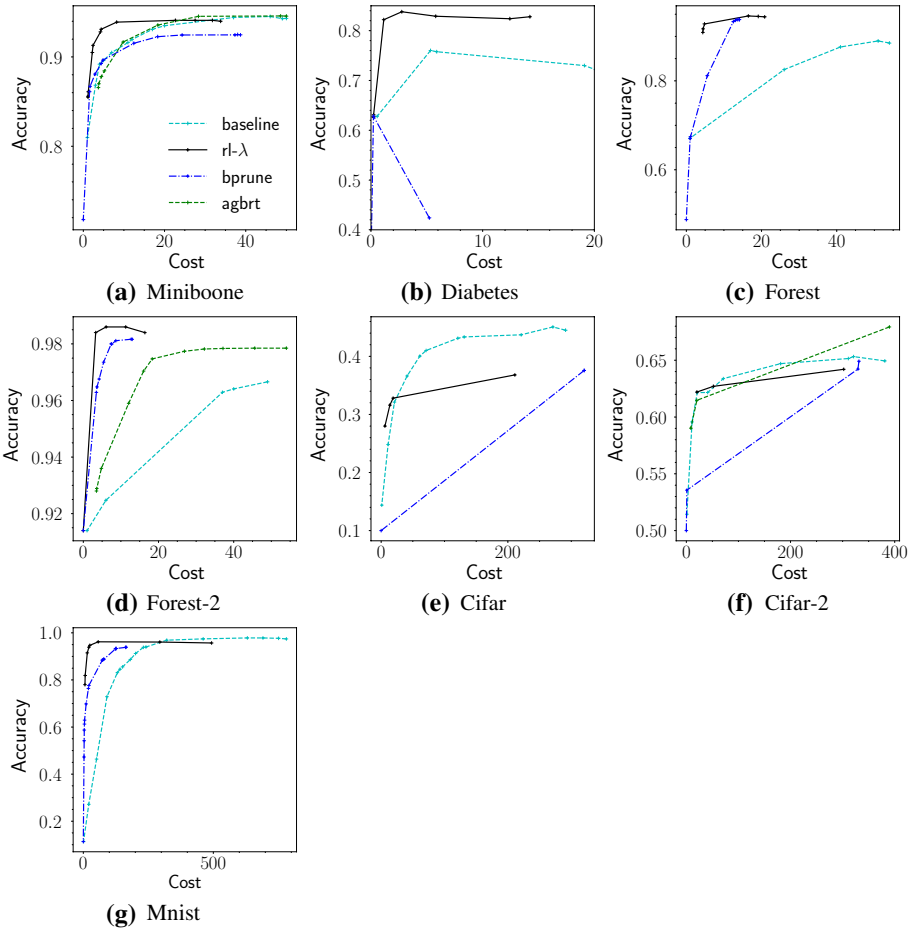
In this section, we describe the performed evaluation of the methods described in Sect. 2. The code used in this evaluation can be obtained at https://github.com/jaromiru/cwcf.

5.1 Time and memory requirements

Let us first discuss the time required for the algorithm to converge and the maximum memory required during the computation. We performed a test for each dataset for two

Table 3 Time and memory requirements until convergence in different datasets of two variations of the algorithm—average budget with trade-off λ and specified average budget b

Dataset	Steps/s	Total steps (k)	Total time	Memory required (GB)
Miniboone-λ	8.0	6	13 mins	6
Miniboone-b	6.0	30	83 mins	6
Diabetes-λ	3.0	3	17 mins	6
Diabetes-b	1.3	4	51 mins	6
Forest-λ	4.5	780	48 h	6
Forest-b	4.0	1500	104 h	6
Cifar-λ	1.2	600	136 h	32
Cifar-b	0.9	520	159 h	32
Mnist-λ	0.4	300	208 h	32
Mnist-b	0.4	500	347 h	32



(a) Miniboone **(b)** Diabetes **(c)** Forest

(d) Forest-2 **(e)** Cifar **(f)** Cifar-2

(g) Mnist

Dataset	baseline	rl-λ	agbrt	bprune
Miniboone	0.925	**0.935**	0.924	0.914
Diabetes	0.750	**0.834**	N/A	0.628
Forest	0.806	**0.924**	N/A	0.906
Forest-2	0.948	**0.984**	0.968	0.978
Cifar	**0.421**	0.356	N/A	0.265
Cifar-2	0.641	0.632	**0.644**	0.599
Mnist	0.888	**0.956**	N/A	0.922

Fig. 5 Comparison of the rl-λ algorithm trained through λ-specified budget, AdaptGbrt (agbrt) and BudgetPrune (bprune). The table shows the normalized area under the trade-off curve as the overall metric. AdaptGbrt algorithm cannot be evaluated in multi-class datasets. Seemingly malformed results of BudgetPrune in (**b**) are caused by overfitting of the algorithm

variations of the algorithm, average budget with λ and specific budget b, set to possibly acquire all features (λ close to 0 or budget b set to the number of features). The reported

memory consumption is an upper limit estimate—during our tests, a lower amount of memory occasionally resulted in an out-of-memory error. Evaluation of a trained model is fast and takes a negligible amount of time.

Each test was for performed on the following configuration: one core of Xeon E5-2650v2 2.60 GHz CPU with nVidia Tesla K20 5GB GPU. The amount of used memory varied across the datasets. The results are summarized in Table 3.

The results indicate that the variation with the directly specified budget b is roughly 2-times slower (depending on the dataset), as measured in the wall-clock time, than in the case with trade-off λ. In the former variation, the algorithm solves a non-stationary environment (λ update is part of the algorithm), hence the longer run-time is expected. However, if the user seeks a model that will achieve a particular budget, the former method is much more convenient. With the trade-off λ variation, the user would need to execute several runs to find a suitable λ, which may be slower at the end. However, the simpler trade-off λ method has its uses, i.e., in case the user has precisely quantified all the feature and misclassification costs in the same units. In such case, the user seeks a model that minimizes the Eq. (1) with $\lambda = 1$.

The run-time seems to be correlated with the number of features in the dataset, with one exception. In the Forest dataset, we attribute the long run-time to the one-hot encoding of the categorical features and the high number of samples—learning to select of meaningful features seems complex in this setting.

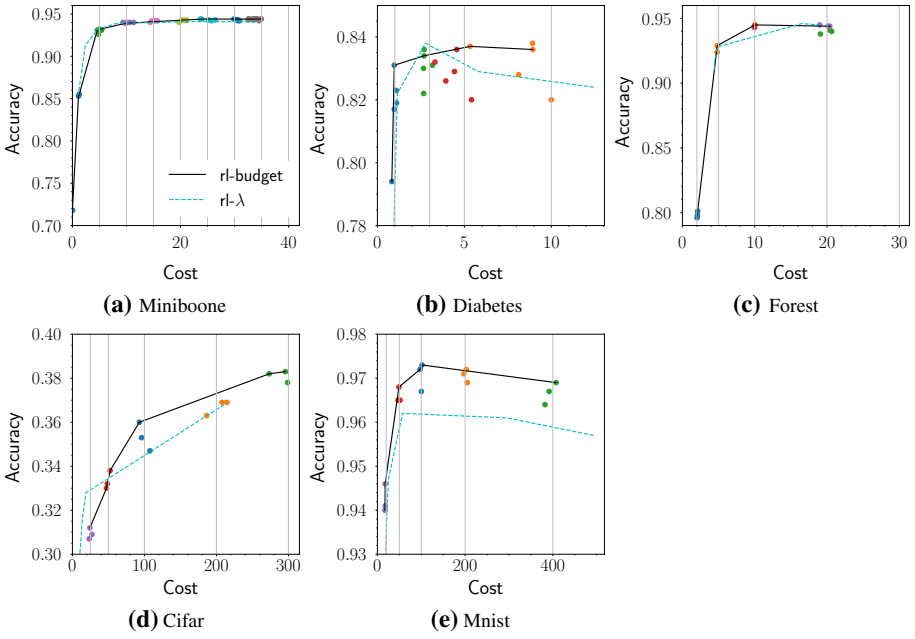


Fig. 6 Average budget setting with directly specified budget b in five different datasets. In each dataset, multiple runs were made with specific budgets (indicated with vertical lines). The runs with the same budget settings are plotted with the same color. For reference, the results of the λ-specified budget are included (dashed line) (Color figure online)

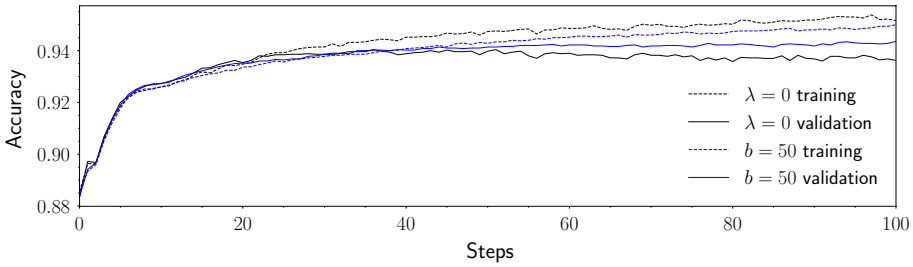


Fig. 7 Comparison of λ-set budget and specific target b. Theoretically equal settings, $\lambda = 0$ (meaning free features) and a specific budget target $b = 50$ (all features in the dataset), result into different behavior. All other settings are the same. Averaged over five runs, one step on the x axis corresponds to 100 training steps

5.2 Average budget with trade-off λ

In this section, we select several representative datasets and compare the RL method (*rl-*λ) in average budget settings with Adapt-Gbrt (*agbrt*) and Budget-Prune (*bprune*), where applicable. The results are shown in Fig. 5. The RL based algorithm performs robustly on all tested datasets, comparable or better to prior-art.

The Fig. 5b show a strange result in the case of the Budget-Prune in the Diabetes dataset. After a careful inspection, we found that the algorithm heavily overfits the training data while it performs poorly on the test set. This behavior results in the strange cost-accuracy curve (which is reported on the test set). In Miniboone, it is noteworthy that the Adapt-Gbrt and Budget-Prune algorithms do not exceed the performance of the baseline classifier. In the Cifar dataset, the baseline method provides well and consistent performance across all budgets, exceeding other methods by a large margin. It is only surpassed by RL when small budgets (up to 20 features) are targeted. We assume that the model capacity is the restricting factor here, as Cifar is a very hard dataset, especially when pixel relations are disregarded. Note that the baseline classifier solves much easier task—at each budget, there is a static set of pixels that are present in every sample. On the other hand, RL algorithm accesses a different set of pixels for each sample, which is a much harder task. Note that we do not use convolutions, which are common in image recognition tasks (to regard all datasets the same), but they could be incorporated into the algorithm, if needed.

It is noteworthy that in case of the Diabetes, Forest and Forest-2 datasets, the rl-λ algorithm's best performance outperforms the baseline classifier with all features, although both algorithms use neural networks of comparable sizes. This indicates that the RL algorithm generalizes much better, possibly because solving harder tasks may have a regularizing effect.

5.3 Average budget with target *b*

In this section, we discuss the results of the method trained with a user-specified budget b, while the λ is automatically learned as explained in the Sect. 2.4. The previous method with the λ defined budget is useful if the exact costs of features and classification are known—in this case, we simply set $\lambda = 1$ and let the algorithm find the best policy. Also, it can be used if we simply want to sweep across all spectrum of budgets, e.g., for comparison reasons. In the case we want to target a specific budget, the variant evaluated here is preferred, as

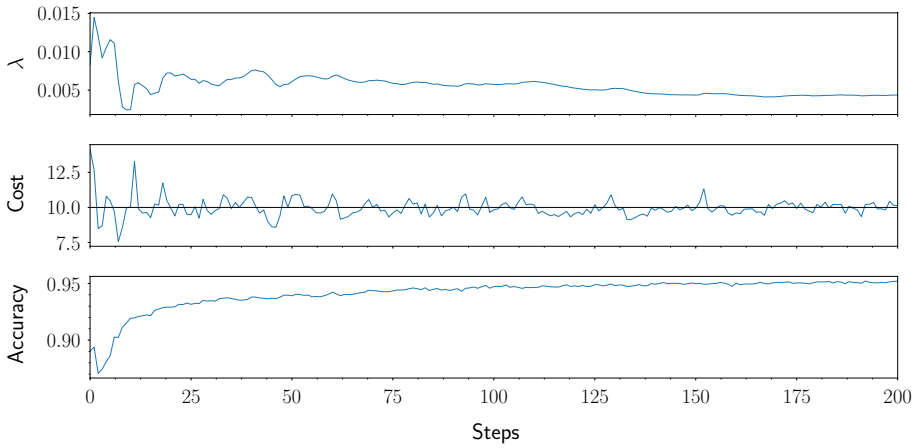


Fig. 8 The learning progress in the Miniboone dataset, with an average target budget $b = 10$. The plot shows changes in λ, spent budget and accuracy during learning (one run, not averaged). One step on the x axis corresponds to 100 training steps

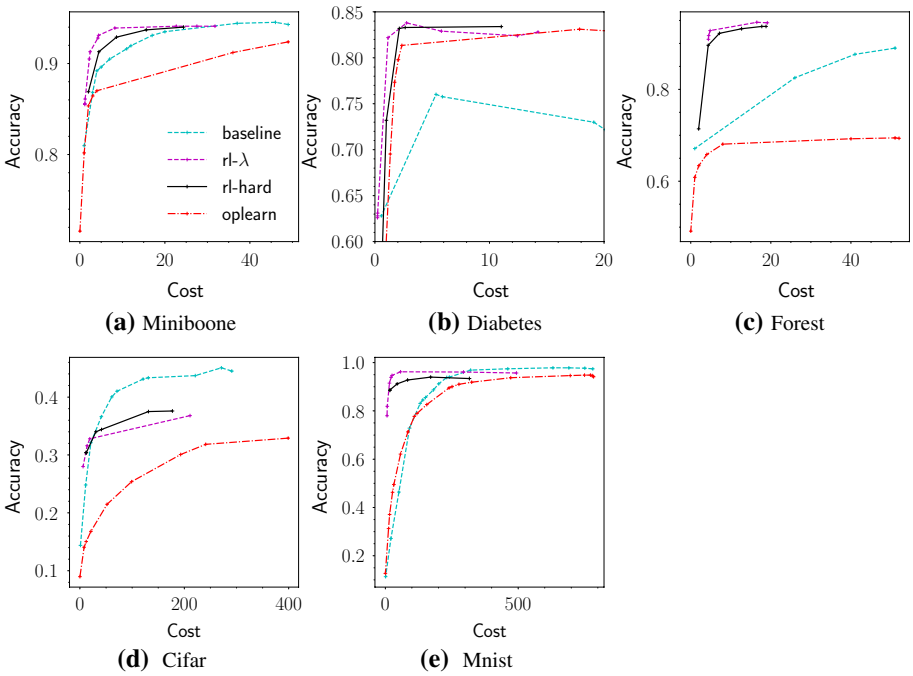
it removes the additional parameter and directly returns a model with a specified budget in one run (which saves computational resources).

For each evaluated dataset, we manually selected several distinct budget targets and ran the algorithm several times with different seeds for each budget. We plotted the results (see Fig. 6) on the cost-accuracy plane with different colors, to highlight the variance between different runs.

Comparing the raw results to the previous method with the λ defined budget, the results are similar and in some cases better. The learned models aligned to the specified budgets in most cases. However, there is some variance in both costs and accuracies, suggesting that in practice, the method should be run several times and only the best performing model should be chosen (based on the validation set).

We attribute the better performance to the normalization effect of the simultaneous optimization of λ—because the environment is effectively non-stationary, the task is slightly harder and the learned model generalizes better. In Fig. 7 we further explore this hypothesis. We analyze a training progress of the two methods, when a budget is specified directly and indirectly with λ. With the Miniboone dataset, we selected two, in theory, equal settings: $\lambda = 0$ and $b = 50$. With fixed λ, setting it to zero effectively means that all features are free and budget is infinite. In the other case, setting $b = 50$ means that all features can be acquired (there are 50 of them and the cost is uniformly 1.0). All other settings were equal and we conducted 5 runs of each algorithm and averaged their results. Figure 7 shows that the specific b-budget method is more resilient to over-fitting—while its performance increases slower than in λ-set budget, the validation performance monotonically raises as well. Also, the asymptotic average accuracy is better in the case of b-budget, about 0.943 in 20,000 steps while λ-budget reaches its top accuracy of 0.940 in about 3500 steps (in Fig. 7, the x-axis scale is different, it corresponds to 200 and 35 steps on the x axis respectively).

Another interesting fact is that the learned models always use the whole available budget up to some point, where it cannot strengthen its accuracy, even with more features. The observation is consistent with previous experiments with λ-targeted budget, where further



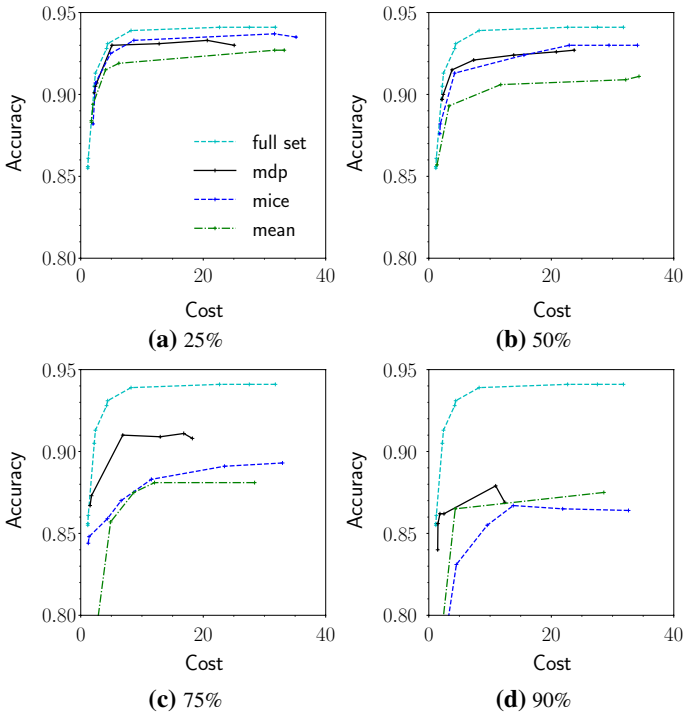
Dataset	baseline	rl-hard	oplearn
Miniboone	0.925	**0.929**	0.894
Diabetes	0.750	**0.826**	0.817
Forest	0.806	**0.916**	0.682
Cifar	**0.421**	0.364	0.281
Mnist	0.888	**0.927**	0.867

Fig. 9 Comparison of RL (*rl-hard*) and Opportunistic Learning (*oplearn*) algorithms in the hard-budget setting. For reference, we also plot the performance in the average budget setting (*rl-λ*). The table shows the normalized area under the trade-off curve metric

lowering λ did not improve accuracy nor depleted more budget. The Miniboone dataset has 50 features, and as it can be seen in the Fig. 6a, the model retrieved 35 at most. Similar effect can be seen in the Forest and Diabetes datasets (Fig. 6b, c).

In Fig. 8, we analyze the training progress. At first, the λ multiplier oscillates, until it converges to its optimal value. Similar oscillations can also be seen in the budgets spent by partially trained models, where the budget approaches the target value by the end of the training. We assume that a small deviation from the average target budget is acceptable. If not, we can simply select the last model that strictly meets the constraint.

In conclusion, using the specific budget b method has several advantages. It achieves slightly higher accuracy, displays better over-fitting resiliency and avoids a superfluous hyperparameter.



(a) 25% **(b)** 50% **(c)** 75% **(d)** 90%

Dataset	mean	mice	mdp
Miniboone-25	0.919	**0.929**	0.926
Miniboone-50	0.903	**0.921**	0.920
Miniboone-75	0.871	0.883	**0.905**
Miniboone-90	0.865	0.856	**0.874**

Fig. 10 The tested methods were trained with the sparse Miniboone dataset, where the stated percentage of features is missing. We compare performance with training on the *full set* without missing features, the altered *mdp* method and *mice* and *mean* imputation algorithms. Trained models were evaluated on the complete dataset with all features. The table shows the normalized area under the trade-off curve. Numbers after the dataset name identifies the percentage of missing features during training

5.4 Hard budget

In the hard budget setting, we compare to Opportunistic Learning (*oplearn*). We do not compare to the work of Kapoor and Greiner (2005), since it solves a slightly different problem. In their case, they do not use a per-sample budget, but rather a budget for the whole training process.

The results can be seen in Fig. 9, where the RL algorithm with hard budget setting is named *rl-hard*. For comparison reasons, we also plot the performance on the average budget task (*rl-λ*). This is to compare the *tasks* themselves, not the algorithms (that is, it cannot be said that one algorithm is better than other).

Generally, compared to average budget setting, the hard budget algorithm should achieve lower performance. In contrast to the hard budget setting, the average budget

method *can exceed* the target budget for selected samples. The experimental results indicate that the performance of both algorithms is similar, except for the Cifar dataset, where the hard budget algorithm is better for a range of costs. A similar result on the Cifar dataset was observed also in the average setting, with a specified budget. We noticed that with the *rl-λ* algorithm, all the resulting models fell either into the low-cost or high-cost region, but nowhere between. In other words, in Cifar, it is hard to select a λ so that the resulting model falls in the middle-cost region. However, with specific budget (both hard and average), we can force the algorithm to find such a model, which may work better. In Diabetes, the performance of the hard budget method is better for a range of costs, which we attribute to overfitting of the average budget method.

Compared to the Opportunistic Learning algorithm, our method achieves substantially better performance in all datasets. We attribute the result to the fact that RL method

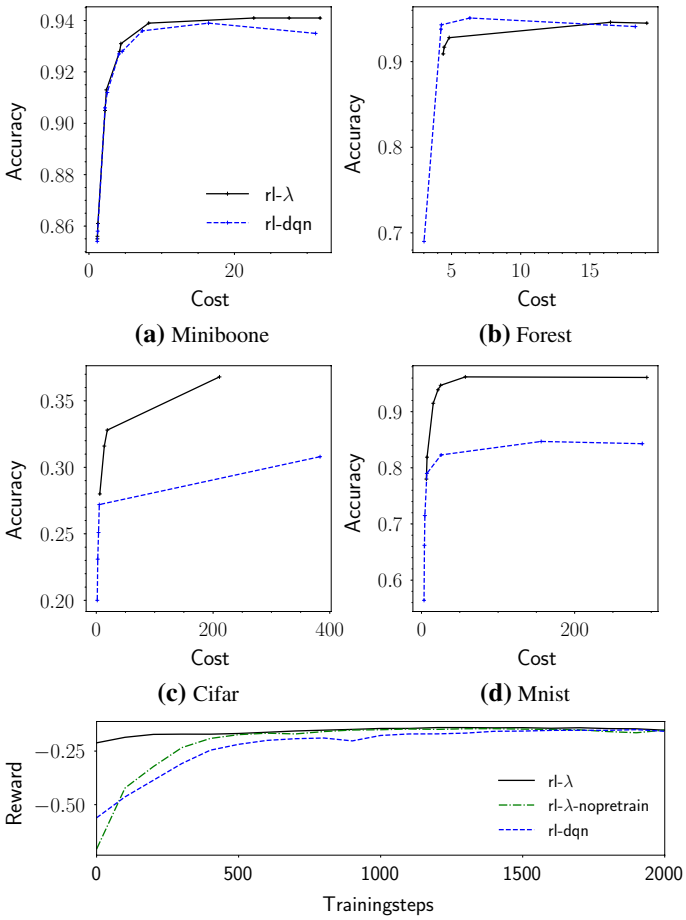


Fig. 11 Comparison of the plain DQN (*rl-dqn*) to a full method used in this work (*rl-λ*). Generally, the improved algorithm achieves a better score, especially in the datasets with a large amount of features (Cifar and Mnist). Subfigure **e** shows the training progression; here we include an ablation of the full algorithm without pretraining (*rl-λ-nopretrain*)

optimizes for the actual objective, while Opportunistic Learning method optimizes a heuristic objective, which is not exactly aligned with the actual goal.

We see a similar effect as in the average budget settings—if the increased budged does not result in increased accuracy, the model learns to stop acquiring features prematurely, to save resources. Note that in the hard setting, RL *never* exceeds the specified budget.

5.5 Missing features

In these experiments we assume that there is a sparse training set, while during the test time, the models can select any feature. That corresponds to the case of the mentioned medical domain, where past data can be elevated for training. However, it is difficult to obtain a real dataset with these attributes and therefore we decided to create a custom synthetic dataset. We artificially drop some percentage of features from the Miniboone dataset. We created four versions, with 25%, 50%, 75% and 90% of features missing. The synthetic datasets were created with an assumption that the features are missing completely at random (MCAR) and the fact that a feature is missing has no predictive power.

We implement the method described in Sect. 2.6 (*mdp*). We use two baseline methods—first, we simply impute the missing features with their *mean* and train the usual way. Second, we use *MICE* algorithm (Azur et al. 2011), which assumes linear dependencies between features. It works by iteratively predicting missing values with a linear regression over known or already predicted features and repeating this process several times. The imputed dataset is then regarded as complete and we train our method in a standard way. For comparison reasons, we also plot the performance on the *full set* without any missing features.

In Fig. 10 we present the results. We see incremental degradation of performance when an increasingly larger percentage of features is missing. The results show that the version with altered MDP performs robustly well. It performs comparably when less than 50% of the features are missing, and performs substantially better with sparser datasets. The mdp method does not involve any preparation and can be directly used in any sparse dataset. It also highlights the flexibility of the RL method. In the case of the MICE imputation method, it has to be noted that the preparation process takes a non-negligible amount of time (about 15 minutes in the Miniboone dataset).

Note that our method can be conveniently used in the case when there are also missing values in the test set, i.e., when some tests are unavailable. In this case, the algorithm simply cannot select the corresponding action and chooses the next best. We also experimented with two different training regimes connected to how we treat the Q-values for the actions corresponding to the missing features. First, we treat them the usual way—if the corresponding feature is missing, its value is unavailable in the computation of the Q-function update target. We hypothesized that this approach fits a setting in which we optimize for the fact, that the features may be missing also in the test set. On the other hand, if we knew that all the features will be available in the test set, it may make sense to include the values of the missing features in the computation of the Q-target (either the maximum in (11) or the expectation in (13)). However, in several experiments we made, the described approach did not seem to bring any benefit. On the contrary, in many experiments, it resulted in worse performance.

5.6 Effect of the RL algorithm

In this section, we demonstrate that the quality of the underlying RL algorithm plays a large role in the quality of the resulting model. Our framework can be easily modified to work with other RL algorithms, such as policy gradients (Mnih et al. 2016) or many different forms of Q-learning (Hessel et al. 2018). However, as we show below, the modifications should be done with care, as it influences the overall quality of the algorithm.

For this comparison, we selected the simple version of the technique we use in this work—plain DQN (see Sect. 3.1), without pretraining of the classification actions. We compare the resulting models to the complete algorithm used in this work (Double Dueling DQN architecture with Retrace, with pretraining). In Fig. 11, we show the results in four selected datasets. As it can be seen, the algorithms perform comparably when the amount of features is small (about 50). However, when applied to larger datasets (Mnist and Cifar), the simple algorithm is outperformed by a large margin. In Fig. 11e, we further study the effectivity of the algorithms and show that the plain DQN also learns much slower (even when we account for the pretraining).

6 Related work

Classification with Costly Features problem has been approached from many directions, with many different types of algorithms. But to our knowledge, there is no single framework that can work with both average and hard budgets, is flexible and perform robustly as our method. In the case of the average budget, usually some variation of the trade-off parameter λ is present. We are not aware of any work that would allow to set a target in the average budget setting.

Most closest works to this article are (Dulac-Arnold et al. 2011), which used Q-learning with limited linear regression, resulting in inferior performance. Recent works (Janisch et al. 2019; Shim et al. 2018) replace the linear approximation with neural networks and report superior performance. However, these methods focus only on the average budget problem and introduce an unintuitive trade-off parameter λ. In (Janisch et al. 2019) the authors showcase the flexibility of the network by incorporating an external classifier as a separate feature.

The following references focus only on the average budget problem. Contardo et al. (2016) use a recurrent neural network that uses attention to select blocks of features and classifies after a fixed number of steps. Mnih et al. (2014) presents an algorithm sequentially chooses image locations to observe; however, the presented algorithm is applicable only to image domains and is cost-agnostic. There is also a plethora of tree-based algorithms (Xu et al. 2012, 2013; Xu et al. 2014; Kusner et al. 2014; Nan et al. 2015, 2016; Nan and Saligrama 2017).

A different set of algorithms employed Linear Programming (LP) to this domain (Wang et al. 2014b, a). Wang et al. (2014a) use LP to select a model with the best accuracy and lowest cost, from a set of pre-trained models, all of which use a different set of features. The algorithm also chooses a model based on the complexity of the sample.

Wang et al. (2015) propose to reduce the problem by finding different disjoint subsets of features, that are used together as macro-features. These macro-features form a graph, which is solved with dynamic programming. In large problems, the algorithm can be used to find efficient groupings of features which would then be used in our method.

Trapeznikov and Saligrama (2013) use a fixed order of features to reveal, with increasingly complex models that can use them. However, the order of features is not computed, and it is assumed that it is set manually. Our algorithm is not restricted to a fixed order of features (for each sample it can choose a completely different subset), and it can also find their significance automatically.

Maliah and Shani (2018) focus on CwCF with misclassification costs, construct decision trees over feature subsets and use their leaves to form states of an MDP. They directly solve the MDP with value-iteration for small datasets with the number of features ranging from 4-17. On the other hand, our method can be used to find an approximate solution to much larger datasets. In this work, we do not account for misclassification costs, but they could be easily incorporated into the rewards for classification actions.

Benbouzid et al. (2012) presents a method to select a subset of available classifiers such that it maximizes the accuracy of the ensemble while it minimizes their count. The problem is formulated as an MDP, in which the model sequentially chooses either to *use* or *skip* a classifier in their fixed order, or to *stop* with a classification. The model only uses the outputs of the classifiers to choose actions and because of this, it can be much simpler and the problem can be solved with tabular methods. In a sense, our algorithm is a generalization of the principles—we do not restrict our model to a fixed order of features and we use their raw values to guide the process. Moreover, our model is able to work with hundreds of features with different costs and can be applied in broad range of domains. However, if the specific use-case fits the work of Benbouzid et al. (2012), their algorithm may be simpler and faster.

Peng et al. (2018) adapt the CwCF setting for a medical domain. They represent the problem as an MDP, which they solve with a policy gradient method. They augment the search with reward shaping and the training with auxiliary targets.

Bayer-Zubek and Dietterich (2005) also view the problem as an MDP with a similar structure. With discretized feature values, they present several methods based on the AO* algorithm to search the policy space (represented with a complete decision tree) for an optimal policy. Their approach is applicable in domains where the discretization of feature values is possible.

Tan (1993) analyzes a problem similar to our definition, but algorithms introduced there require memorization of all training examples, which is not scalable in many domains.

The hard budget case was explored in (Kapoor and Greiner 2005), who studied random and heuristic based methods. Deng et al. (2007) used techniques from the multi-armed bandit problem. There are also theoretical works (Cesa-Bianchi et al. 2011; Zolghadr et al. 2013). Kachuee et al. (2019) crafted a heuristic reward and used RL to maximize it.

7 Conclusion

In this work, we presented a flexible reinforcement learning (RL) framework for solving the Classification with Costly Features (CwCF) problem. We build on established work that already showcased the superior performance of RL in this problem. We modified it to work with a directly specified budget in average and hard budget cases. For the average case, we introduced the Lagrangian theory to automatically find suitable parameters. We also modified the framework in a principled way, to be able to work with datasets with missing features. All settings were evaluated on several diverse datasets and we report that our method robustly outperforms other algorithms in most settings.

The flexibility of the RL framework was successfully demonstrated by all mentioned versions of the algorithm. We showcased its robustness (it performs well across all datasets) and the ease of use (almost all hyperparameters stay the same across all datasets and algorithm variations). Moreover, the method is based on the standard RL algorithm and it can benefit from any improvement in the RL area itself.

Acknowledgements We thank Cheng Zhang for suggesting the baseline method. This research was supported by the European Office of Aerospace Research and Development (Grant No. FA9550-18-1-7008) and by The Czech Science Foundation (Grants No. 18-21409S and 18-27483Y). The GPU used in this research was donated by the NVIDIA Corporation. Computational resources were provided by the CESNET LM2015042 and the CERIT Scientific Cloud LM2015085, provided under the program Projects of Large Research, Development, and Innovations Infrastructures.

References

Azur, M. J., Stuart, E. A., Frangakis, C., & Leaf, P. J. (2011). Multiple imputation by chained equations: what is it and how does it work? *International Journal of Methods in Psychiatric Research*, *20*(1), 40–49.

Bayer-Zubek, V., & Dietterich, T. G. (2005). Integrating learning from examples into the search for diagnostic policies. *Journal of Artificial Intelligence Research*, *24*, 263–303.

Benbouzid, D., Busa-Fekete, R., & Kégl, B. (2012). Fast classification using sparse decision dags. In: *Proceedings of the 29th international conference on international conference on machine learning, Omnipress*, pp. 747–754.

Bertsekas, D. P. (1999). Nonlinear programming. Athena scientific Belmont.

Cesa-Bianchi, N., Shalev-Shwartz, S., & Shamir, O. (2011). Efficient learning with partially observed attributes. *Journal of Machine Learning Research*, *12*, 2857–2878.

Chow, Y., Ghavamzadeh, M., Janson, L., & Pavone, M. (2017). Risk-constrained reinforcement learning with percentile risk criteria. *Journal of Machine Learning Research*, *18*(167), 1–167.

Contardo, G., Denoyer, L., & Artieres, T. (2016). Recurrent neural networks for adaptive feature acquisition. In: *International conference on neural information processing*. Springer, pp 591–599.

Deng, K., Bourke, C., Scott, S., Sunderman, J., & Zheng, Y. (2007). Bandit-based algorithms for budgeted learning. In: *Seventh IEEE international conference on data mining (ICDM 2007)*. IEEE, pp 463–468.

Dulac-Arnold, G., Denoyer, L., Preux, P., & Gallinari, P. (2011). Datum-wise classification: A sequential approach to sparsity. In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer, pp. 375–390.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. In: *Advances in neural information processing systems*, pp. 2672–2680.

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine Learning*, *46*(1–3), 389–422.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In: *Thirty-second AAAI conference on artificial intelligence*.

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, *12*(1), 55–67.

Janisch, J., Pevný, T., & Lisý, V. (2019). Classification with costly features using deep reinforcement learning. In: *AAAI conference on artificial intelligence*.

Kachuee, M., Goldstein, O., Karkkainen, K., Darabi, S., & Sarrafzadeh, M. (2019). Opportunistic learning: Budgeted cost-sensitive learning from data streams. In: *International conference on learning representations.*

Kapoor, A., & Greiner, R. (2005). Learning and classifying under hard budgets. In: *European conference on machine learning*. Springer, pp. 170–181.

Kingma, D.P., & Ba, J. (2015). Adam: A method for stochastic optimization. In: *International conference on learning representations*.

Krizhevsky, A., & Hinton, G. (2009). Learning multiple layers of features from tiny images. Master's thesis, University of Toronto.
Kusner, M., Chen, W., Zhou, Q., Xu, Z., Weinberger, K., & Chen, Y. (2014). Feature-cost sensitive learning with submodular trees of classifiers. In: *AAAI conference on artificial intelligence*, pp. 1939–1945.
Lichman, M. (2013). UCI machine learning repository. http://archive.ics.uci.edu/ml.
Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In: *International conference on learning representations*.
Lin, L.J. (1993). Reinforcement learning for robots using neural networks. PhD thesis, Carnegie Mellon University.
Maliah, S., & Shani, G. (2018). Mdp-based cost sensitive classification using decision trees. In: *AAAI conference on artificial intelligence*, pp. 3746–3753.
Mnih, V., Heess, N., Graves, A., et al. (2014). Recurrent models of visual attention. In: *Advances in neural information processing systems*, pp. 2204–2212.
Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.
Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In: *International conference on machine learning*, pp. 1928–1937.
Munos, R., Stepleton, T., Harutyunyan, A., & Bellemare, M. (2016). Safe and efficient off-policy reinforcement learning. In: *Advances in neural information processing systems*, pp. 1054–1062.
Nan, F., & Saligrama, V. (2017). Adaptive classification for prediction under a budget. In: *Advances in neural information processing systems*, pp. 4730–4740.
Nan, F., Wang, J., & Saligrama, V. (2015). Feature-budgeted random forest. In: *International conference on machine learning*, pp. 1983–1991.
Nan, F., Wang, J., & Saligrama, V. (2016). Pruning random forests for prediction on a budget. In: *Advances in neural information processing systems,* pp. 2334–2342.
Peng, Y.S., Tang, K.F., Lin, H.T., & Chang, E. (2018). Refuel: Exploring sparse features in deep reinforcement learning for fast disease diagnosis. In: *Advances in neural information processing systems*, pp. 7322–7331.
Shim, H., Hwang, S.J., & Yang, E. (2018). Joint active feature acquisition and classification with variable-size set encoding. In: *Advances in neural information processing systems*, pp. 1375–1385.
Tan, M. (1993). Cost-sensitive learning of classification knowledge and its applications in robotics. *Machine Learning*, *13*(1), 7–33.
Trapeznikov, K., & Saligrama, V. (2013). Supervised sequential classification under budget constraints. In: *Artificial intelligence and statistics*, pp. 581–589.
Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In: *AAAI conference on artificial intelligence*, pp. 2094–2100.
Wang, J., Bolukbasi, T., Trapeznikov, K., & Saligrama, V. (2014a). Model selection by linear programming. In: *European conference on computer vision*. Springer, pp. 647–662.
Wang, J., Trapeznikov, K., & Saligrama, V. (2014b). An lp for sequential learning under budgets. In: *Artificial intelligence and statistics*, pp. 987–995.
Wang, J., Trapeznikov, K., & Saligrama, V. (2015). Efficient learning by directed acyclic graph for resource constrained prediction. In: *Advances in neural information processing systems*, pp. 2152–2160.
Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In: *International conference on machine learning*, pp. 1995–2003.
Xu, Z., Weinberger, K., & Chapelle, O. (2012). The greedy miser: learning under test-time budgets. In: *Proceedings of the 29th international conference on international conference on machine learning*, Omnipress, pp. 1299–1306.
Xu, Z., Kusner, M., Weinberger, K., & Chen, M. (2013). Cost-sensitive tree of classifiers. In: *International conference on machine learning*, pp. 133–141.
Xu, Z., Kusner, M., Weinberger, K., Chen, M., & Chapelle, O. (2014). Classifier cascades and trees for minimizing feature evaluation cost. *Journal of Machine Learning Research*, *15*(1), 2113–2144.
Zolghadr, N., Bartók, G., Greiner, R., György, A., & Szepesvári, C. (2013). Online learning with costly features and labels. In: *Advances in neural information processing systems*, pp. 1241–1249.