Neural Conditional Gradients

Patrick Schramowski [1] **Christian Bauckhage** [2 3] **Kristian Kersting** [1 4]

Abstract

The move from hand-designed to learned optimizers in machine learning has been quite successful for gradient-based and -free optimizers. When facing a constrained problem, however, maintaining feasibility typically requires a projection step, which might be computationally expensive and not differentiable. We show how the design of projection-free convex optimization algorithms can be cast as a learning problem based on Frank-Wolfe Networks: recurrent networks implementing the Frank-Wolfe algorithm aka. conditional gradients. This allows them to learn to exploit structure when, e.g., optimizing over rank-1 matrices. Our LSTM-learned optimizers outperform hand-designed as well learned but unconstrained ones. We demonstrate this for training support vector machines and softmax classifiers.

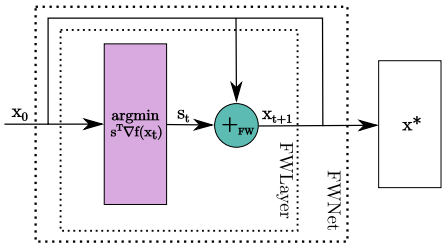
1. Introduction

Machine learning tasks can often be expressed as general constrained convex optimization problems of the form

$$\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x}\in\boldsymbol{S}} \; f(\mathbf{x}) \;, \tag{1}$$

where f is a convex and continuously differentiable function, and $\boldsymbol{S}$ is a compact convex subset of a Hilbert space. For such optimization problems, one of the simplest and earliest known iterative optimizers is given by the Frank-Wolfe (FW) algorithm (Jaggi, 2013), summarized in Fig. 1(bottom), also known as *conditional gradient*. In each iteration, it considers the linearization of the objective at the current position $\mathbf{x}$ and moves towards a convex minimizer of this linear function (taken over the same domain). In other words, Frank-Wolfe effectively turns the constrained convex optimization problem into a series of simple linear optimization problems. Recently, Bauckhage (2017) employed this view to implement Frank-Wolfe optimizing over the unit simplex—the convex hull $\boldsymbol{S} := \operatorname{conv}(\{\mathbf{e}_i | i \in [n]\})$ of the unit basis vectors—in terms of a recurrent neural network (RNN). Since the domain $\boldsymbol{S}$ is given as an intersection of linear constraints, the subproblems can be solved using softmin activation functions. This paper significantly extends our understanding of such neural conditional gradients.

Frank-Wolfe Network (FWNet)



Frank-Wolfe (FW) algorithm (1956)

1: Let $\mathbf{x}_0 \in \boldsymbol{S}$
2: **for** $t = 1, 2, 3, \ldots, T$ **do**
3: Choose step-size $\gamma_t \in [0, 1]$, e.g., $\gamma_t = \frac{2}{t+1}$
4: Compute $\mathbf{s}_t = \operatorname{argmin}_{s\in\boldsymbol{S}} \; \mathbf{s}^T \nabla f(\mathbf{x}_t)$
5: Update $\mathbf{x}_{t+1} = (1 - \gamma_t)\mathbf{x}_t + \gamma_t \mathbf{s}_t$

Figure 1. FWNets (top) implement the Frank-Wolfe algorithm (bottom) as recurrent neural networks. Unrolled over time, a FWNet layer takes the current state $\boldsymbol{x}_t$ as input, computes the linearization (purple layer) and moves the next internal state $\boldsymbol{x}_{t+1}$ towards a convex minimizer of $\boldsymbol{x}_t$ and this linearization (green layer).

As warm up, we show that the resulting Frank-Wolfe Networks (FWNets)—the generalized architecture is shown in Fig. 1(top)—allow one to implement (training) support vector machines directly within neural networks. Unfortunately, the resulting neural optimizer is too dense to scale to large classification problems: it hinges on the quadratic gram matrix. Consequently, as our second contribution, we introduce sparse FWNets for convex optimization over the unit ball of the *trace-norm*, i.e., $\boldsymbol{S} := \operatorname{conv}(\{\mathbf{u}\mathbf{v}^T | \mathbf{u} \in \mathbb{R}^n, ||\mathbf{u}||_2 = 1 \text{ and } \mathbf{v} \in \mathbb{R}^n, ||\mathbf{v}||_2 = 1\})$. Since the subproblems amount to approximating the unit left and right top singular vectors of the gradient matrix $\nabla f(\mathbf{w}_t)$, we replace the softmin activation functions by sparse RNNs that are structurally equivalent to the well known power iteration. This allows one to realize neural conditional gradients for, e.g., sparse softmax classifiers that scale well to large datasets.

[1]CS Depart., TU Darmstadt, Germany [2]B-IT, Univ. of Bonn, Germany [3]Fraunhofer IAIS, Sankt Augustin, Germany [4]Center for Cognitive Science, TU Darmstadt, Germany. Correspondence to: Patrick Schramowski <schramowski@cs.tu-darmstadt.de>.

The closest in spirit to FWNets are probably OptNets (Amos & Kolter, 2017). They integrate constrained optimization problems, in particular quadratic ones as individual layers into neural networks. This has also the potential of richer end-to-end training for complex tasks that require such optimization. However, OptNets do not cast the optimizer itself as a neural network. Instead, external optimizers are invoked to solve OptNets. This hampers a seamless integration with other deep learning concepts.

Consider e.g. *learning to learn* (L2L), which has a long history in psychology (Ward, 1937; Harlow, 1949; Kehoe, 1988) and has inspired many recent attempts within the machine learning community to build agents capable of learning to learn (Schmidhuber, 1987; Naik & Mammone, 1992; Thrun & Pratt, 1998; Hochreiter et al., 2001; Santoro et al., 2016; Chen et al., 2017; Wang et al., 2016; Ravi & Larochelle, 2017; Li & Malik, 2017). So far, however, learning to learn has mainly been considered for gradient(-free) optimizers (L2LG); *learning to learn by conditional gradients* (L2LC) has not been proposed. Our third contribution fills this gap. We show how to boost the performance of neural conditional gradients by learning parts of them instead of using hand-coded ones. Our learned conditional gradient optimizers, implemented by LSTMs, outperform hand-designed as well as unconstrained but learned competitors. We demonstrate this on a number of classification tasks, including training deep SVM and softmax classifiers.

We proceed as follows. We start of by reviewing L2L. Then we illustrate FWNets and use them to devise L2LC. Afterwards we introduce sparse FWNets for *trace-norm* problems. Before concluding, we present our experimental evaluation.

2. Learning to learn by gradients by gradients

Let us start off by briefly reviewing learning to learn by gradients by gradients (Chen et al., 2017). The goal is to optimize an objective function $f(\theta)$ defined over some domain $\theta \in \Theta$. To this end, we find the minimizer $\theta^* = \operatorname{argmin}_{\theta\in\Theta} f(\theta)$. While any method capable of minimizing this objective can be applied, the standard approach for differentiable functions is some form of gradient descent, resulting in a sequence of updates $\theta_{t+1} = \theta_t - \alpha_t \nabla(\theta_t)$. To realize L2L, Chen et al. (2017) proposed to replace hand-designed update rules with a learned update rule, called the optimizer g, specified by its own set of parameters. This results in updates to the optimizee f of the form $\theta_{t+1} = \theta_t + g_t(\nabla(\theta_t), \phi)$. More precisely, Chen et al. advocated to realize the update rule g using a recurrent neural network (RNN), which maintains its own state and hence dynamically updates as a function of its iterates.

Indeed this learning to learn by gradients by gradients is widely applicable due to the simplicity of gradient computations. When facing a constraint optimization problem, however, maintaining feasibility typically requires a projection step, which is potentially computationally expensive, especially for complex feasible regions in very large dimensions. To overcome this, we advocate the use of the Frank-Wolfe algorithm (Jaggi, 2013), which eschews the projection step and rather use a linear optimization oracle to stay within the feasible region. While convergence rates and regret bounds are often suboptimal, in many cases the gain due to only having to solve a single linear optimization problem over the feasible region in every iteration still leads to significant computational advantages. This may explain its popularity for problems such as computing the distance to a convex hull, computing a minimum enclosing ball, or training a support vector machine.

3. Neural support vector machines

Support Vector Machine (SVM) are working horses of machine learning. Frank Wolfe algorithms for training them (Ouyang & Gray, 2010) solve a quadratic program (QP) over the unit simplex, i.e., the convex hull $\boldsymbol{S} := \operatorname{conv}(\{\mathbf{e}_i | i \in [n]\})$ of the unit basis vectors.

To implement them as neural networks, we can proceed as follows. Consider, e.g., the l_2-SVM formulation for binary classification

$$\min \left(\frac{1}{2} w^2 - p + \frac{C}{2} \sum\nolimits_{i=1}^{N} \epsilon_i^2 \right) \; s.t. \; \mathbf{w}^T \mathbf{z}_i \geq p - \xi_i$$

where $\mathbf{z}_i = y_i \mathbf{x}_i$. The corresponding Lagrangian dual problem for SVMs can be expressed as:

$$\min\nolimits_{\boldsymbol{\alpha}\in\mathbb{R}^m} f(\boldsymbol{\alpha}) = \frac{1}{2} \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha}, \alpha_i \geq 0, \sum\nolimits_{i=1}^{N} \alpha_i = 1. \quad (2)$$

Here, $\boldsymbol{K}$ is a positive definite kernel matrix $\boldsymbol{K} = \boldsymbol{Z}^T \boldsymbol{Z} = (\boldsymbol{y} \circ \boldsymbol{x})^T (\boldsymbol{y} \circ \boldsymbol{x})$. As shown in (Frandi et al., 2014), $H(\boldsymbol{\Sigma}) = \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$, hence we have $\boldsymbol{s}_t = \boldsymbol{e}_t^{i^*}$ where

$$\begin{aligned} i_t^* &\in \operatorname{argmin}_{i=1,\ldots,m} \nabla f(\alpha_t)^{(i)} \\ &= \operatorname{argmin}_{i=1,\ldots,m} \sum\nolimits_{j|\alpha_t^{(j)}>0} K^{(i,j)} \alpha_t^{(j)}. \end{aligned}$$

So, the gradient for the new objective is $\nabla f(\boldsymbol{\alpha}_t) = \boldsymbol{K}\boldsymbol{\alpha}$. Therefore FW requires computing

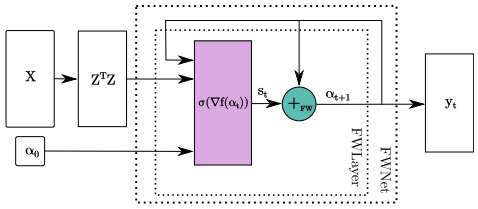
$$\boldsymbol{s}_t = \operatorname{argmin}_i \boldsymbol{e}_i^T (\boldsymbol{K}\boldsymbol{\alpha}) \approx \boldsymbol{\sigma}(\boldsymbol{K}\boldsymbol{\alpha}) \quad (3)$$

where the non-linear, vector-valued function $\boldsymbol{\sigma}(\boldsymbol{z})$ is the well known softmin operator defined as

$$\boldsymbol{\sigma}(\boldsymbol{z})_i = \frac{\exp(-\beta z_i)}{\sum_j \exp(-\beta z_j)} \quad (4)$$

for which we note $\lim_{\beta\to\infty} \boldsymbol{\sigma}(z) = \boldsymbol{e}_i = \operatorname{argmin}_j \boldsymbol{e}_j^T \boldsymbol{z}$. Plugging in the relaxed optimization step (3), we can now rewrite the Frank-Wolfe updates for SVMs as

Neural Support Vector Machine



Frank-Wolfe (FW) algorithm over the unit simplex

1: Let $\mathbf{w}_0 \in \boldsymbol{S}$
2: **for** $t = 1, 2, 3, \ldots, T$ **do**
3: Choose step-size $\gamma_t \in [0, 1]$, e.g., $\gamma_t = \frac{2}{t+1}$
4: Compute $\mathbf{s}_t = \boldsymbol{\sigma}(\nabla f(\boldsymbol{\alpha}_t))$
5: Update $\boldsymbol{\alpha}_{t+1} = (1 - \gamma_t)\boldsymbol{\alpha}_t + \gamma_t \mathbf{s}_t$

Figure 2. A FWNet (top) implementing Support Vector Machines (SVMs). The gram matrix $Z^T Z$ is treated as synaptic connections of n neurons and is kept fixed over time. Each unrolled FWNet layer takes it and the support vector coding $\boldsymbol{\alpha}_t$ as input, computes $\sigma(\nabla f(\boldsymbol{\alpha}_t))$ as linearization (purple layer) and moves the next support vector coding $\boldsymbol{\alpha}_{t+1}$ towards a convex minimizer of $\boldsymbol{\alpha}_t$ and this linearization (dark green layer). This is a neural instantiation of the FW algorithm for optimization over the unit simplex (bottom).

$$\begin{aligned} \boldsymbol{\alpha}_{t+1} &= \boldsymbol{\alpha}_t + \gamma_t \boldsymbol{d}_t = \boldsymbol{\alpha}_t + \gamma_t(\boldsymbol{s}_t - \boldsymbol{\alpha}_t) \\ &= (1 - \gamma_t)\boldsymbol{\alpha}_t + \gamma_t \boldsymbol{s}_t \approx (1 - \gamma_t)\boldsymbol{\alpha}_t + \gamma_t \boldsymbol{\sigma}(\boldsymbol{K}\boldsymbol{\alpha}_t) \ , \end{aligned}$$

where $\boldsymbol{K}\boldsymbol{\alpha}_t = \nabla f(\boldsymbol{\alpha})$. But this is then to say that by choosing an appropriate parameter for the softmin function the following non-linear dynamical system

$$\boldsymbol{\alpha}_{t+1} = (1 - \gamma_t)\boldsymbol{\alpha}_t + \gamma_t \boldsymbol{\sigma}(\nabla f(\boldsymbol{\alpha}_t)), \tag{5}$$

mimics Frank-Wolfe up to arbitrary precision.

The underlying FW over the unit simplex is summarized in Fig. 2(bottom). Structurally it is equivalent to the system of equations governing the dynamics of echo state networks, a particular from of recurrent neural networks (RNN), shown in Fig. 2(top) for training SVMs. For inference, we can unroll the RNN into a multi-layer neural network. Due to well known FW convergence results (Jaggi, 2013), we know that $\mathcal{O}(1/\epsilon)$ layers are likely to provide an ϵ-approximate solution to the SVM problem. Moreover, the neural view on training SVMs allows one to deepify SVMs: we replace the final classification layer of a deep network by a FWNet that trains an SVM. This enables end-to-end training akin to (Tang, 2013; Zhang et al., 2015) but in a simpler and fully neural fashion: the SVM parameters are updated via a forward-propagation only, and the parameters of the kernel neural network are updated by gradient descent using back-propagation of the error starting at the FWNet, cf. Fig. 3. Here, $\boldsymbol{h}_{n-1}$ denotes the input to the FWNets. During training, the FWNet computes the Kernel $\boldsymbol{K}$ at each iteration

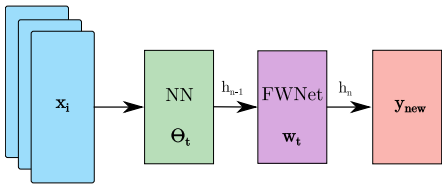


Figure 3. Deep SVMs: Stacking Frank-Wolfe Networks (FWNets) for SVMs on top of a deep neural network.

and the weights $\boldsymbol{w}$ (respectively $\boldsymbol{\alpha}$ of Eq. (5)) are updated as described above. To predict the class of a new example $\boldsymbol{x}_{new}$, we make one foward-pass through the network.

4. Learning to learn by conditional gradients by gradients (L2LC)

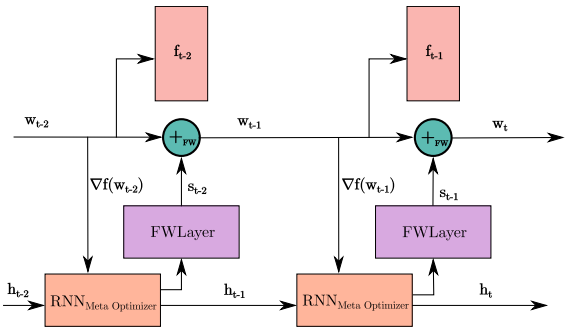
The performance of FWNets is hampered by the fact that they only makes use of the linearizations, ignoring other information such as curvature. To speed them up, we now introduce *learning to learn by conditional gradients by gradients* (L2LC) as depicted in Fig. 4.

The FWNet is unrolled over time t, and at each step t parts of the optimizer are trained using an RNN as optimize (orange). This way, the optimizee adapts the parts, which are then used to form a Frank-Wolfe update. Consider, e.g., learning to learn SVMs. Instead of using the typical hand-coded rule $\gamma = 2/(2 + t)$ or implementing a line-search, we learn to adapt, e.g., the learning-rate:

$$\begin{aligned} L(\boldsymbol{\gamma}) &= \mathbb{E}_f \left[\sum\nolimits_{t=1}^{T} f(\boldsymbol{w}_t) \right] \text{ where} \\ \gamma_t, \boldsymbol{h}_{t+1} &= RNN(\gamma_{t-1}, \boldsymbol{h}_t, \phi) \\ \boldsymbol{w}_{t+1} &= (1 - \gamma_t)\boldsymbol{w}_t + \gamma_t \boldsymbol{\sigma}(\nabla f(\boldsymbol{w}_t)) \end{aligned} \tag{6}$$

where $\boldsymbol{w}$ are the weights of the Frank-Wolfe layer and $\mathbf{h}$ the

Figure 4. Learning to learn by conditional gradients by gradients. The optimizer, an FWNet, is unrolled over time, resulting in FWLayers (purple boxes). They are trained by an RNN, the optimizee (orange boxes).



Frank-Wolfe algorithm for optimization over low-rank matrices using power iterations

1: Let $\boldsymbol{w}_0 \sim \mathcal{N}(0,1)$
2: **for** $t = 0, 1, \ldots, T-1$ **do**
3: Choose step-size $\gamma_t \in [0,1]$, e.g., $\gamma_t = \frac{2}{t+2}$
4: Compute $\nabla F(\boldsymbol{w}_t)$
5: Set $\boldsymbol{v}_0 \in R^m$ uniformly from unit sphere
6: **for** $k = 0, 1, \ldots, K-1$ **do**
7: $\boldsymbol{u}_{k+1} \leftarrow \nabla F(\boldsymbol{w}_t)\boldsymbol{v}_k$
8: $\boldsymbol{u}_{k+1} \leftarrow \boldsymbol{u}_{k+1}/||\boldsymbol{u}_{k+1}||$
9: $\boldsymbol{v}_{k+1} \leftarrow \nabla F(\boldsymbol{w}_t)^T\boldsymbol{u}_k$
10: $\boldsymbol{v}_{k+1} \leftarrow \boldsymbol{v}_{k+1}/||\boldsymbol{v}_{k+1}||$
11: $\boldsymbol{s}_t = -\mu\boldsymbol{u}_1\boldsymbol{v}_1^T$
12: Update $\boldsymbol{w}_{t+1} = (1-\gamma_t)\boldsymbol{w}_t + \gamma_t\boldsymbol{s}_t$

FWNets for optimization over low-rank matrices using neural power iterations

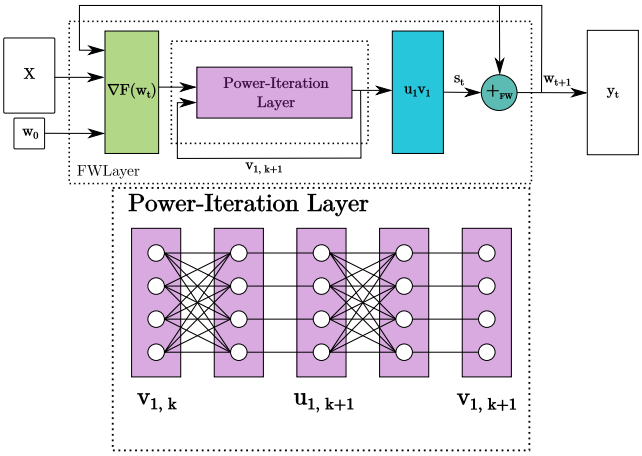


Figure 5. On bounded trace-norm domain, the subproblems of FW(left) amount to approximating the unit left and right top singular vectors of the gradient matrix $\nabla F(\mathbf{w}_t)$. To implement this within FWNets (right), RNNs implementing power iterations are used.

state of an RNN, e.g., an LSTM. Or, we learn to adapt the conditional gradient itself. For that, one has to be little bit more careful. We have to ensure that the predictions are on the unit simplex:

$$\begin{aligned} g_t, \boldsymbol{h}_{t+1} &= RNN(\nabla f(\boldsymbol{w}_t), \boldsymbol{h}_t, \phi) \\ \boldsymbol{w}_{t+1} &= (1-\gamma_t)\boldsymbol{w}_t + \gamma_t\boldsymbol{\sigma}(g_t) \end{aligned} \tag{7}$$

where $\gamma_t = 2/(t+2)$ or γ_t is constant. That is, the RNN predicts the unconstrained gradient, which is then projected onto the unit simplex using a sigmoid. Overall, the FWNet is unrolled over the learning iterations t, and at each step t the unconstrained gradient $\nabla f(w_t)$ is used as input to the RNN, the optimizee (orange). The prediction in then squeezed through a sigmoid and we update the weight vector.

5. Neural sparse softmax classifiers

Unfortunately, neural SVMs are not likely to scale well. The underlying SVM scales quadratically in the number of training examples due to the gram matrix. Indeed, one may resort to devise neural implementation of stochastic Frank-Wolfe algorithms (Jaggi, 2013) or frame the learning problem within L2LC, generalizing local FWNets to a global model (Vinyals et al., 2016; Ravi & Larochelle, 2017). Here we introduce FWNets for training large-scale, sparse softmax classifiers (Liu & Tsang, 2017), i.e., for optimization problems of the following form:

$$\min_{\boldsymbol{w}\in\boldsymbol{M}} F(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}) \tag{8}$$

where $\boldsymbol{M} = \{w \in \mathbb{R}^{h\times m} | \; ||\boldsymbol{w}||_* \leq \tau\}$ with $||\cdot||_*$ being the trace-norm (also called the nuclear- or Schatten l_1-norm). The trace-norm ball is the convex hull of the rank-1 matrices, which is also compact. The averaged multi-class objectives $f_i(\boldsymbol{w})$ are $f_i(\boldsymbol{w}) = \sum_{k=1}^{K} \boldsymbol{y}_k^{(i)} \log(p_k^{(i)})$ with $p_k^{(i)} = \exp(\boldsymbol{w}\boldsymbol{x}_i)/\sum_{j=1}^{N}\exp(\boldsymbol{w}\boldsymbol{x}_j)$. The individual gradients are $\nabla f_i(\boldsymbol{w}) = (p_k^{(i)} - \boldsymbol{y}_k^{(i)})\boldsymbol{x}^{(i)}$.

Since Schatten-norms are invariant under orthogonal transformations, we can employ the singular value decomposition (SVD) to minimize the induced linear subproblems. Therefore the main computational cost of a single FWLayer on a Schatten-norm domain remains the computation of the SVD of $\nabla F(\boldsymbol{w}_t)$, which is in $O(\min\{mn^2, m^2n\})$ (Jaggi, 2013). For bounded trace-norm, however, the subproblems can be solved by a single approximate eigenvector computation instead of a complete SVD, which is much more efficiently, especially if the matrix dimensions are large and the optimal solution is low-rank (Allen-Zhu et al., 2017). This gives Frank-Wolfe a significant computational advantage over projected and proximal gradient descent approaches. The vectors $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$ can be efficient computed via power iteration (Zheng et al., 2017). This results in a rank-1 solution of Eq. (8), which can be written as $-\mu\boldsymbol{u}_1\boldsymbol{v}_1^T$, where $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$ are the unit left and right top singular vectors of the gradient matrix $\nabla F(\boldsymbol{w}_t)$: $\boldsymbol{w}_{t+1} = (1-\gamma_t)\boldsymbol{w}_t - \gamma_t\mu\boldsymbol{u}_1\boldsymbol{v}_1^T$.

The Frank-Wolfe algorithm and corresponding FWNet for the trace-norm domain are shown in Fig. 5. Here, lines 4-11 instantiate the general Frank-Wolfe algorithm in Alg. 1 with k power iterations to compute the top singular vectors $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$ of $\nabla F(\boldsymbol{w}_t)$. Zheng et al. (2017) showed that a small number of power iterations $K(t) = \mathcal{O}(\log t)$ is sufficient to ensure a sublinear convergence in expectation and if the number of power iterations are constant (i.e. $K(t) = k$ for

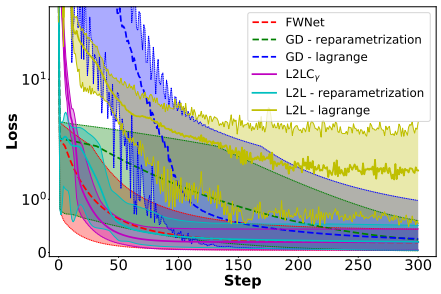


Figure 6. Loss while training of FWNet, GD, L2LC, and L2L on different Cifar-10 subsets with two classes (Cifar-2). The optimizee (L2LC and L2L) is trained on two fixed classes, but used to optimize all combinations of classes for the binary SVM and therefore transferring to a completely novel dataset.

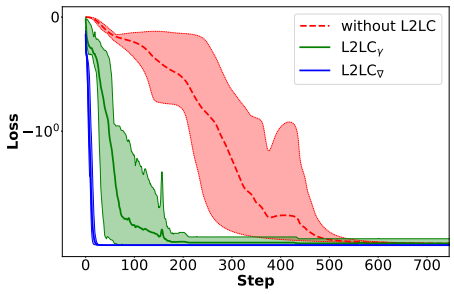


Figure 7. Loss while learning "concentric circles" using $\beta = 1.0$, $C = 1.0$, $\gamma_{FW} = 0.01$, $\gamma_{ADAM} = 0.01$. L2LC$_\gamma$ (green line) optimizes the step-size and L2LC$_\nabla$ (blue line) the kernel.

all t) the Frank-Wolfe algorithm converges in expectation to a neighborhood of the optimal solution whose size decreases with k. In any case, the power iteration can naturally be implemented within FWNets using an RNN as summarized in Fig. 5. Everything else remains conceptually the same and, in turn, we may even realize L2LC over low-rank matrices following similar arguments as for the unit simplex.

6. Experimental evidence

Our intention here is to evaluate neural conditional gradients by investigating the following questions: (**Q1**) Can FWNets compete with popular, non-neural gradient descent approaches such as ADAM (Kingma & Ba, 2014)? (**Q2**) Can we train CSVMs end-to-end using FWNets? (**Q3**) Can L2LC be faster than L2LG? (**Q4**) Do neural rank-1 softmax classifiers perform and scale well?

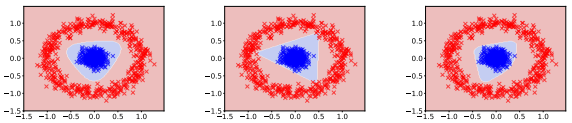
To this end, we implemented FWNets, neural SVMs, neural softmax classifiers, and L2LC using the TensorFlow API version 1.3 and the L2L implementation of (Chen et al., 2017). All experiments were ran on a Linux Machine with a NVIDIA GeForce GTX 1080 Ti with 11 GB memory and a AMD Ryzen Threadripper 1950X CPU with 16 physical cores having 32 threads in total. We considered several datasets. For comparing FWNets with classical, non-neural gradient optimization, we used both the synthetic datasets of "concentric circles" (Fig. 8) as well as the real-world datasets MNIST (LeCun et al., 1998) containing images of handwritten digits and Cifar-10 respectively Cifar-100 (Krizhevsky, 2009) containing images of different animals and vehicles. For the L2L experiments, we split the data into three disjoint sets. One split was used to train the optimizee, one for training the optimizer, and the final one to test the corresponding learned model. The neural SVMs are compared to ADAM gradient optimizers based on (1) reparameterization and (2) Lagrange multipliers to deal with the "sum to one" constraint. The objective of the Langrangian approach reads

$$f(\boldsymbol{\alpha}) = 0.5\boldsymbol{\alpha}^T \boldsymbol{K}\boldsymbol{\alpha} - \lambda \sum_{i=1}^{m} \alpha_i \text{ , s.t. } \alpha_i \geq 0. \quad (9)$$

Furthermore we evaluate the performance of learning to learn the optimizer for solving the given problem. For that we used an LSTM as optimzee. More precisley, following (Chen et al., 2017), we introduced an additional LSTM to optimize the step-size respectively two different LSTMs when optimizing the the fully connected and convolutional layers. In all experiments we used two-layer LSTMs with 20 hidden units in each layer, aiming at minimizing (2) respectively (9) using truncated backpropgation through time and early stopping in order to avoid overfitting.

Few-Shot Neural SVMs (Q1, Q3). In our first experiment we considered classes 1 and 2, denoted as Cifar-2, from the Cifar-10 dataset. We extracted their features from an inception-network and used them for training the base models (**Q1**) using a linear kernel. Additionally we train an optimizee for FW (**Q3**). A random search set $\beta = 10$. Fig. 6 summarizes the results. The optimizee is trained on the classes 1 and 2, but then used to optimize neural SVMs on all pairwise combinations (1-3,1-4,...,2-3,...) of classes from Cifar-10 and therefore transferring to a completely novel dataset. As one can see, FWNets and L2LC outperformed the other baselines. FWNets with an hand-design, adaptive stepsize can be slightly faster than L2LC$_\gamma$, but the LSTM learns to control FW in a similar way and shows a much smaller variance. This answers (**Q1**, **Q3**) affirmatively.

Deep SVMs (Q2, Q3). Next we considered training deep SVMs on the "concentric circles" dataset, i.e., we placed an FWNet as a last layer of a neural network, trained in an end-to-end (**Q1**) as well as in a learning to learn fashion (**Q4**). The neural network we used as kernel contained three layers. The first and second layer are fully-connected layer with four and two neurons each, trained using ADAM. Fig. 7 summarizes the results. As one can see, the learned optimizers outperformed the hand-coded ones. Moreover,



(a) FWNets 500 iters. (b) L2LC$_\gamma$ 200 iters. (c) L2LC$_\nabla$ 20 iters.

Figure 8. Hyperplanes achieving the same losses are found at different iterations. The hyperparameters used were $\beta = 1.0$, $C = 1.0$, $\gamma_{FW} = 0.01$, $\gamma_{ADAM} = 0.01$

as Fig. 8 illustrates, a learned optimizer may find smoother hyperplanes achieving the same loss in less many iterations, when also adapting the kernel: just using FWNets takes 500 iterations; when the LSTM controls the step size, it takes 200 iterations; when also adapting the kernel, it just takes 20 iterations and the hyperplane is considerably smoother. This answers (**Q2**, **Q3**) affirmatively.

To investigated deep SVMs further, we also considered Cifar-10. We split the labels of the dataset in two different classes, namely *natural* and *manmade*. The class *natural* contains the classes *bird*, *cat*, *deer*, *dog*, *frog* and *horse*, and the *manmade* class the classes *airplane*, *automobile*, *ship* and *truck*. As kernel we used a neural network with both convolutional and fully-connected layers: three convolutional layers with max pooling followed by a fully-connected layer with 32 hidden units; all non-linearities were ReLU activations with batch normalization. The final layer is a FWNet simulating to train an SVM, and the rest of the network was trained using ADAM. Fig. 9 summarize the results. As one can see, the stepsize-learned conditional gradient L2LC$_\gamma$ outperforms hand-coded optimizers even with adaptive stepsize; requiring less than half of the iterations to converge. Training also the kernel is harder as it is a non-convex problem; exploring this further is an interesting avenue for future work. In any case, the results answer (**Q2**, **Q3**) affirmatively.

Sparse Neural Softmax Classifiers (Q1, Q4). Finally, we investigated FWNets for training deep softmax classifiers on MNIST. We used a simple CNN with two convolutional layer and one fully-connected consisting of 16 neurons followed by a fully-connected softmax layer. For both FWnets and ADAM-based optimizers we used the same constant step size of $\gamma = 0.001$. The FWNets unrolled the power iteration networks for five steps in order to compute the left and right top singular vectors of the gradient matrix $\nabla F(\boldsymbol{w}_t)$. The step size was set $\mu = 50$. The results are summarized in Fig. 10. As one can see, the sparse FWNet outperformed ADAM, both in terms of convergence and predictive performance; the same top-1 accuracy in less than half of the iterations. This answers (**Q1**, **Q4**) affirmatively.

To investigate this further, we also considered wider and deeper CNNs on MNIST and also on Cifar-10 and Cifar-100. The more dense the network became, the better the

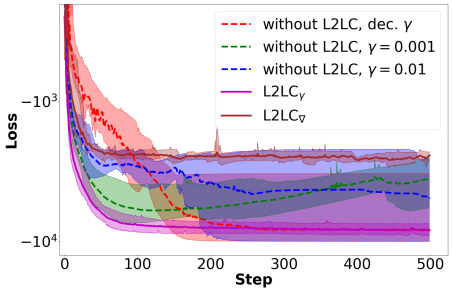


Figure 9. Loss while training on the Cifar-10 dataset using the two classes *manmade* and *natural*.

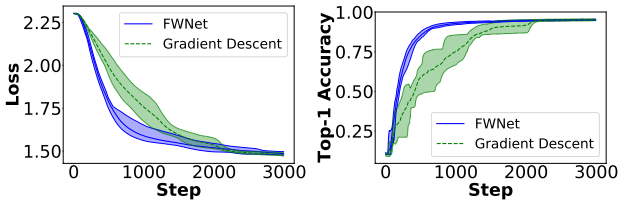


Figure 10. (Left) Learning curves of FWNets- (blue line) and ADAM-based (green line) softmax classifiers on the MNIST dataset. (Right) Top-1 accuracy on test-set on MNIST.

ADAM performed. This validates our assumption of low-rank solutions: if the low-rank assumption does not hold or is not required, there is no point in estimating a sparse model using the trace-norm constraint (Allen-Zhu et al., 2017).

7. Conclusion

We have introduced the *learning to learn by conditional gradients* (L2LC) framework based on Frank-Wolfe Networks (FWNets). This enables one to train sparse convex optimizers that are specialized to particular classes of problems. We illustrated this for training SVMs and sparse softmax classifiers. Our experimental results confirm that learned conditional gradients compare favorably against state-of-the-art optimization methods used in deep learning.

There are several interesting avenues for future work. One should develop FWNets for other ML tasks such as graph classification (Kersting et al., 2014) and Bayesian Quadrature (Briol et al., 2015) as well as for other FW approaches (Jaggi, 2013). One may also adapt the Power Iteration in an end-to-end fashion (Duvenaud et al., 2015). Finally, hierarchical RNNs (Wichrowska et al., 2017) have the potential to speed up *learning to learn by conditional gradients*.

Acknowledgments: This work was supported by the Federal Ministry of Food, Agriculture and Consumer Protection (BMELV) based on a decision of the German Federal Office for Agriculture and Food (BLE); grant nr. “2818204715”.

References

Allen-Zhu, Z., Hazan, E., Hu, W., and Li, Y. Linear Convergence of a Frank-Wolfe Type Algorithm over Trace-Norm Balls. In *Proc. NIPS*, 2017.

Amos, B. and Kolter, J.Z. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *Proc. ICML*, 2017.

Bauckhage, C. A Neural Network Implementation of Frank-Wolfe Optimization. In *Proc. ICANN*, 2017.

Briol, F.-X., Oates, C.J., Girolami, M.A., and Osborne, M.A. Frank-Wolfe Bayesian Quadrature: Probabilistic Integration with Theoretical Guarantees. In *Proc. NIPS*, 2015.

Chen, Y., Hoffman, M.W., Gomez Colmenarejo, S., Denil, M., Lillicrap, T.P., Botvinick, M., and de Freitas, N. Learning to Learn without Gradient Descent by Gradient Descent. In *Proc. ICML*, 2017.

Duvenaud, D.K., D.Maclaurin, Aguilera-Iparraguirre, J., Gómez-Bombarelli, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R.P. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Proc. NIPS*, 2015.

Frandi, E., Ñanculef, R., and Suykens, J.A. K. Complexity Issues and Randomization Strategies in Frank-Wolfe Algorithms for Machine Learning. *arXiv:1410.4062*, 2014.

Frank, M. and Wolfe, P. An Algorithm for Quadratic Programming. *Naval Research Logistics Quarterly*, 3(1–2): 95–110, 1956.

Harlow, H.F. The Formation of Learning Sets. *Psychological Review*, 56(1):51–65, 1949.

Hochreiter, S., Younger, A.S., and Conwell, P.R. Learning to Learn Using Gradient Descent. In *Proc. ICANN*, 2001.

Jaggi, M. Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization. In *Proc. ICML*, 2013.

Kehoe, E. A Layered Network Model of Associative Learning: Learning to Learn and Configuration. *Psychological Review*, 95(4):411–433, 1988.

Kersting, K., Mladenov, M., Garnett, R., and Grohe, M. Power Iterated Color Refinement. In *Proc. AAAI*, 2014.

Kingma, D.P. and Ba, J. Adam: A Method for Stochastic Optimization. In *Proc. ICLR*, 2014.

Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. Technical report, Univ. of Toronto, 2009.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Li, K. and Malik, J. Learning to Optimize Neural Nets. *arXiv:1703.00441*, 2017.

Liu, Z. and Tsang, I.W. Approximate Conditional Gradient Descent on Multi-Class Classification. In *Proc. AAAI*, 2017.

Naik, D.K. and Mammone, R.J. Meta-Neural Networks that Learn by Learning. In *Proc. IJCNN*, 1992.

Ouyang, H. and Gray, A.G. Fast Stochastic Frank-Wolfe Algorithms for Nonlinear SVMs. In *Proc. SDM*, 2010.

Ravi, S. and Larochelle, H. Optimization as a Model for Few-shot Learning. In *Proc. ICLR*, 2017.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T.P. Meta-Learning with Memory-Augmented Neural Networks. In *Proc. ICML*, 2016.

Schmidhuber, J. Evolutionary Principles in Self-Referential Learning. Master's thesis, TU Munich, 1987.

Tang, Y. Deep Learning Using Support Vector Machines. *arXiv:1306.0239*, 2013.

Thrun, S. and Pratt, L. (eds.). *Learning to Learn*. Kluwer Academic Publishers, 1998.

Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., and Wierstra, D. Matching Networks for One Shot Learning. In *Proc. NIPS*, 2016.

Wang, J.X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J.Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. Learning to Reinforcement Learn. *arXiv:1611.05763*, 2016.

Ward, L.B. Reminiscence and Rote Learning. *Psychological Monographs*, 49(4), 1937.

Wichrowska, O., Maheswaranathan, N., Hoffman, M.W., Gomez Colmenarejo, S., Denil, M., de Freitas, N., and Sohl-Dickstein, J. Learned Optimizers that Scale and Generalize. *arXiv:1703.04815*, 2017.

Zhang, S.-X., Liu, C., Yao, K., and Gong, Y. Deep Neural Support Vector Machines for Speech Recognition. In *Proc. ICASSP*, 2015.

Zheng, W., Bellet, A., and Gallinari, P. A Distributed Frank-Wolfe Framework for Learning Low-Rank Matrices with the Trace Norm. *arXiv:1712.07495*, 2017.