Frank-Wolfe and Coordinate Descent for Deep Learning

Federico Betti, Ioannis Mavrothalassitis, Luca Rossi
CS-439 Optimization for Machine Learning, EPFL Lausanne, Switzerland

***Abstract*—Training a deep neural network is a challenging, high dimensional, non-convex optimization problem. The current state-of-the-art optimizers are stochastic gradient descent (SGD) and other adaptive variants. In some cases, the structure of the problem at hand enables to employ other optimizers, and possible variants, which are valid alternatives to gradient-based methods in the convex and smooth setting. In this paper, we are interested in two such examples: the Coordinate Descent and the Frank-Wolfe algorithms. We show empirically that the *Block Coordinate Descent* algorithm performs comparably to the state-of-the-art on small datasets, while requiring a much lower cost per epoch. We combine it with commonly used optimizers to obtain faster training and more accurate results and we propose a gradient descent based variant which allows weaker assumptions on the objective. We then present the *Deep Frank-Wolfe* algorithm and we suggest a variant which yields quicker and more stable learning by solving more accurately a proximal problem. The implementation of our code is available at https://github.com/johnmavro/Machine-Learning-Optimization.**

I. Introduction

Stochastic gradient descent is the most commonly used optimization algorithm for neural network training, because it provides the best generalization error [Zhou et al., 2020]. One of its main downsides is that it requires a hand-designed schedule for the learning rate. To tackle this issue, many empirically successful adaptive methods were introduced [Duchi et al., 2011; Kingma and Ba, 2014]. Consequently, other optimizers have rarely been studied in deep learning settings. For instance, the Coordinate Descent algorithm is a valid optimization method if the objective satisfies the Polyak-Łojasiewicz (PL) inequality and is coordinate-wise smooth with convergence in $\mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\right)$ steps [Karimi et al., 2016]. Similarly, the Frank-Wolfe algorithm minimizes in $\mathcal{O}\left(\frac{1}{\epsilon}\right)$ steps a convex and smooth function over a convex and compact set [Frank and Wolfe, 1956]. However, in non-convex settings, their usage is still limited. Under suitable assumptions on the activation function of all the layers, an attempt of using Coordinate Descent in deep learning has been proposed in [Zeng et al., 2019]. We present this algorithm, which we refer to as *Block Coordinate Descent*, in Section 2. Later, we propose a gradient descent based variant which weakens the assumptions on the objective, contrarily to the closed form solution implemented by the authors, which works only for particular losses. Finally, we combine the cheap cost per iteration of this algorithm with the better performance of other optimizers to obtain faster convergence and better results. In Section 3, we present the *Deep Frank-Wolfe* algorithm proposed in [Berrada et al., 2018], which optimizes at each epoch a loss preserving proximal problem with the Frank-Wolfe algorithm. We show empirically that solving more accurately this proximal problem in the early stages of training leads to better results in our tests, while containing the additional computational cost.

II. Coordinate Descent

A. The Block-Coordinate Descent Algorithm in a nutshell

We consider a neural network with $N-1$ hidden layers. Let us denote by $\mathcal{W} = \{W_i \in \mathbb{R}^{d_i \times d_{i-1}}\}_{i=0}^{N}$ the set of weight matrices, $X = (x_1, x_2, ..., x_n) \in \mathbb{R}^{d_0 \times n}$ and $Y = (y_1, y_2, ..., y_n) \in \mathbb{R}^{d_N \times n}$ the feature vectors and the labels, respectively. A DNN training problem can be reformulated as

$$\min_{\mathcal{W}} R_n(\Phi(X, \mathcal{W}), Y), \tag{1}$$

where $R_n(\Phi(X, \mathcal{W}), Y) = \frac{1}{n}\sum_{j=1}^{n} l(\Phi(x_j, \mathcal{W}), y_j)$ is the empirical risk for some loss function l and $\Phi(x_j, \mathcal{W}) = \sigma_N(W_N\sigma_{N-1}(W_{N-1}\dots W_2(\sigma_1(W_1 x_j))))$ is the neural network model. By introducing additional variables $\{U_i, V_i\}_{i=1}^{N}$, [Taylor et al., 2016] reformulate (1) as

$$\begin{aligned} &\min_{\mathcal{W}} R_n(V_N, Y) \quad \text{subject to} \\ &U_i = W_i V_{i-1}, \quad V_i = \sigma(U_i) \quad i = 1, \dots, N. \end{aligned} \tag{2}$$

In a quadratic penalty method fashion, it has been proposed in the literature to solve

$$\min_{\mathcal{V},\mathcal{W},\mathcal{U}} R_n(V_N, Y) + \frac{\gamma}{2}\sum_{i=1}^{N}\left[||V_i - \sigma(U_i)||_F^2 + ||U_i - W_i V_{i-1}||_F^2\right], \tag{3}$$

as a surrogate objective to (2), where $\mathcal{U} = \{U_i\}_{i=1,\dots,N}$ and $\mathcal{V} = \{V_i\}_{i=1,\dots,N}$. Given a regularization parameter $\alpha \in \mathbb{R}^+$, [Zeng et al., 2019] update the variables $\{V_i, U_i, W_i\}$ by minimizing the objective in (3) recursively variable-wise from the output to the input layer, while keeping the others fixed to their last updated values and enforcing the update to be close to the previous one. By construction, this becomes an easier subproblem for each variable which can be solved in closed form when σ_i is the ReLU activation function for $i = 1, \dots, N-1$ (while in our setting $\sigma_N == \mathrm{Id}$) and l is the MSE. The complete algorithm is presented in Algorithm 1.

B. A more general way to solve the V_N update

The update of V_N is not always trivial and heavily depends on $R_n(V_N, Y)$ (the choice of the loss function). While using the squared loss as in [Zeng et al., 2019] enables to update V_N by a closed form solution, the latter choice of the cost function is sub-optimal for classification tasks. As a solution for other differentiable losses, the authors suggest to use a prox-linear update strategy. We present in Appendix B the case in which l is the cross-entropy loss. Alternatively, we can minimize the objective in line 2 of Algorithm 1 by employing gradient descent for T iterations. Given a step size η_t, the update rule when l is the cross-entropy loss is given by (see Appendix A)

$$V_N^k = V_N^{k-1} - \eta_t\left[\gamma(V_N^{k-1} + U_N) + \frac{exp(V_N^{k-1})}{\sum_i exp((V_N^{k-1})^i)} - y\right], \tag{4}$$

where y is the one-hot encoding of the target output.

Algorithm 1: Block Coordinate Descent

Data: $X \in \mathbb{R}^{d_0 \times n}$,$Y \in \mathbb{R}^{d_N \times n}$

Initialize$\{W_i^0, V_i^0, U_i^0\}_{i=0}^N$,$V_0^k \equiv V_0 = X$

while *not converged* **do**

  $V_N^k = \text{argmin}_{V_N}\{R_n(V_N;Y) + \frac{\gamma}{2}\left\|V_N - U_N^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|V_N - V_N^{k-1}\right\|_F^2\}$

  $U_N^k = \text{argmin}_{U_N}\{\frac{\gamma}{2}\left\|V_N^k - U_N\right\|_F^2 + \frac{\gamma}{2}\left\|U_N - W_N^{k-1}V_{N-1}^{k-1}\right\|_F^2\}$

  $W_N^k = \text{argmin}_{W_N}\{\frac{\gamma}{2}\left\|U_N^k - W_N V_{N-1}^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|W_N - W_N^{k-1}\right\|_F^2\}$

  for *i*=$N-1,\ldots,1$ **do**

    $V_i^k = \text{argmin}_{V_i}\{\frac{\gamma}{2}\left\|V_i - \sigma(U_i^{k-1})\right\|_F^2 + \frac{\gamma}{2}\left\|U_{i+1}^k - W_{i+1}^k V_i\right\|_F^2\}$

    $U_i^k = \text{argmin}_{U_i}\{\frac{\gamma}{2}\left\|V_i^k - \sigma(U_i)\right\|_F^2 + \frac{\gamma}{2}\left\|U_i - W_i^{k-1}V_{i-1}^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|U_i - U_i^{k-1}\right\|_F^2\}$

    $W_i^k = \text{argmin}_{W_i}\{\frac{\gamma}{2}\left\|U_i^k - W_i V_{i-1}^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|W_i - W_i^{k-1}\right\|_F^2\}$

  end

end

C. Combining the best of the two worlds

According to our experiments in the next subsection, the *Block Coordinate Descent* algorithm is faster than gradient-based methods, but with a slightly worse performance. We propose a hybrid variant which trains with *Block Coordinate Descent* in the first 60% of the epochs and with usual optimizers (e.g. SGD, Adam) towards the end of training. This optimizer combines the best of both worlds and thus reaches consistently a higher accuracy than the standard algorithm, while being faster than the state-of-the-art optimizers.

D. Results

Table I shows the obtained results on the MNIST dataset. We present for space reasons the results for FMNIST, CIFAR10 in Appendix F). The hyper-parameters used are listed below (for the hybrid optimizers we use their combination):

- *Block Coordinate Descent*: $\gamma = 0.1$, $\alpha = 4$.
- *Adam*: $\gamma = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$.
- *SGD*: $\gamma = 0.01$, $\mu = 0.9$, scheduler *StepLR(stepsize=15, gamma=0.2)*.
- *Block Coordinate Descent with Prox Linear V_N update*: $\gamma = 0.1$, $\alpha = 4$.
- *Block Coordinate Descent with GD update* (4) *for V_N*: $\gamma = 0.1$, $\alpha = 4$, $T = 250$, $\eta_t = \mathcal{O}\left(\frac{1}{T}\right)$.
- *Block Coordinate Descent*: $\gamma = 0.1$, $\alpha = 4$.

	Test Accuracy	Average Time Per Epoch
BCD	93.71	0.98
Adam	96.41	9.47
Prox Linear	93.46	0.98
BCD (GD update)	93.68	1.84
SGD (with schedule)	94.57	9.33
BCD + Adam	95.93	4.25
BCD + SGD	95.56	4.29

TABLE I

RESULTS FROM 50 TRAINING EPOCHS ON THE *MNIST* DATASET USING A 4 LAYER PERCEPTRON AS AN ARCHITECTURE.

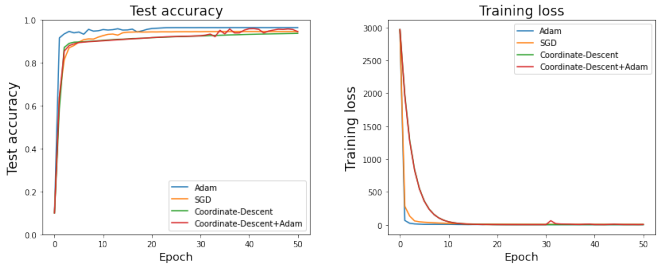

Fig. 1. Test accuracy and training loss on the MNIST dataset using a 4 layer perceptron for some selected optimizers

III. FRANK-WOLFE

A. The Deep Frank-Wolfe Algorithm in a nutshell

Consider a given dataset $(x_i, y_i)_{i\in[N]}$, where $x_i \in \mathbb{R}^d$ is a sample with label y_i. For an input x_i and given $w \in \mathbb{R}^p$ the network parameters, the model predicts $f(x_i, w) := f_i(w) \in \mathbb{R}^{|\mathcal{Y}|}$, a vector with one score per element of the output space $\mathcal{Y}$. We consider the minimization task

$$\min_{w\in\mathbb{R}^p} \frac{1}{N}\sum_{i=1}^N \mathcal{L}_i(f_i(w), y_i), \tag{5}$$

where $\mathcal{L}_i$ is a convex and piecewise linear loss function, such as the Multi-Class Hinge Loss

$$\mathcal{L}_{\text{hinge}} : (u, y) \in \mathbb{R}^{|\mathcal{Y}|}\times\mathcal{Y} \mapsto \max\{\max_{\bar{y}\in\mathcal{Y}\setminus y}\{u_{\bar{y}}+1-u_y\}, 0\}, \tag{6}$$

where $(\cdot)_y$ refers to the component for $y \in \mathcal{Y}$. Stochastic gradient descent, chosen a sample j, performs an update on the parameters by setting ([Bubeck et al., 2015])

$$w_{t+1} = \text{argmin}_w\{\frac{1}{2\eta}||w - w_t||^2 + T_{w_t}[\mathcal{L}_j(f_j(w))]\}, \tag{7}$$

where $\mathrm{T}_{w_t}(\cdot)$ denotes Taylor linearization centered in w_t and η is the learning rate. Therefore, SGD minimizes the first order approximation of the objective, while enforcing w to be close to w_t. On the other hand, the *Loss Preserving Linearization* (LPL) proposed in [Berrada et al., 2018] linearizes the model f_j but not the objective $\mathcal{L}_j$: thus, the update in this case is given by

$$w_{t+1} = \text{argmin}_w\{\frac{1}{2\eta}||w - w_t||^2 + \mathcal{L}_j(T_{w_t} f_j(w))\}. \tag{8}$$

[Lacoste-Julien et al., 2013] showed that problem (8), when $\mathcal{L}_j$ is a hinge loss, is amenable to optimization in the dual employing the Frank-Wolfe algorithm. Given $(\bar{y}, y) \in \mathcal{Y}^2$, let us define $\Delta(\bar{y}, y) = 1$ if $\bar{y} = y$, 0 otherwise. Moreover, for $\bar{y} \in \mathcal{Y}$, let us introduce the quantities

$$a_{\bar{y}} = \partial f_{x,\bar{y}}(w)|_{w_0} - \partial f_{x,y}(w)|_{w_0}, \tag{9}$$

$$b_{\bar{y}} = f_{x,\bar{y}}(w_0) - f_{x,y}(w_0) + \Delta(\bar{y}, y). \tag{10}$$

Then we can recast (8) as [Berrada et al., 2018]

$$w_{t+1} = \text{argmin}_w\left\{\frac{1}{2\eta}||w - w_t||^2 + \max_{\bar{y}\in\mathcal{Y}}\{a_{\bar{y}}^T(w - w_0) + b_{\bar{y}}\}\right\}. \tag{11}$$

Using (11), the Lagrangian dual of (8) reads as:

$$\min_{\alpha\in\mathcal{P}}\{\frac{1}{2\eta}||A\alpha||^2 - b^T\alpha\}, \tag{12}$$

where $A = (\eta a_{\bar{y}})_{\bar{y}\in\mathcal{Y}}$ and $b = (b_{\bar{y}})_{\bar{y}\in\mathcal{Y}}$, and $\mathcal{P}$ denotes the probability simplex $\mathcal{P} = \{\alpha \in \mathbb{R}_+^{|\mathcal{Y}|} : \sum_{\bar{y}\in\mathcal{Y}} \alpha_{\bar{y}} = 1\}$. Note that, indeed, problem (12) corresponds to the minimization of

a convex function over the convex set $\mathcal{P}$. Moreover, by the relation $w - w_0 = -A\alpha$, w_0 denoting the initial weights, we can perform directly all the updates in the primal variables. The proximal Frank-Wolfe algorithm is presented in Algorithm 2.

Algorithm 2: Proximal Frank-Wolfe Algorithm

Data: Proximal coefficient η, initial point $w_0 \in \mathbb{R}^p$

$w_1 = w_0$
$\lambda_1 = 0$
$t = 1$
while *not converged* **do**
  Choose direction $s_t \in \mathcal{P}$
  $w_s = -As_t$
  $\lambda_s = b^T s_t$
  $\gamma_t = \dfrac{(w_t - w_0 - w_s)^T(w_t - w_0) + \eta(\lambda_s - \lambda_t)}{||w_t - w_0 - w_s||^2}$
  $w_{t+1} = (1 - \gamma_t)w_t + \gamma_t(w_s + w_0)$
  $\lambda_{t+1} = (1 - \gamma_t)\lambda_t + \gamma_t\lambda_s$
  $t \leftarrow t + 1$
end

B. A variant which ensures quicker learning

In [Berrada et al., 2018], the authors perform at each epoch a single step of the proximal Frank-Wolfe Algorithm 2 by taking as feasible dual direction the gradient of $\mathcal{L}_i$, claiming that this is sufficient to retain the desired accuracy. For space reasons, we present the full algorithm in Appendix D. However, we observed empirically that in this setting learning was unstable and rather slow in the early stages, even though the final performance of the optimizer was comparable to the state-of-the-art. We obtained better training trends by performing multiple steps of Algorithm 2 in the first 20% of the training epochs. To make our algorithm computationally competitive with the original one and with the other optimizers, we restrict ourselves to performing only one additional step. Moreover, we take as feasible dual direction for the second proximal step the same used in the first one (i.e. the gradient).

C. Results

The *Deep Frank-Wolfe* Algorithm achieves state-of-the-art accuracy while not requiring a schedule for the learning rate, since the latter is automatically adapted. On the other hand, we discussed how performing multiple proximal steps in the early stages of training helped us to stabilize and speed up learning. In this section, we present the empirical results supporting our thesis. The hyper-parameters used are the following:

- *DFW and DFW multistep*: $\eta = 0.1$, $\mu = 0.9$.
- *Adam*: $\gamma = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$.
- *SGD*: $\gamma = 0.1$, $\mu = 0.9$, scheduler *StepLR(stepsize=20, gamma=0.2)*.

	DenseNet	GoogLeNet	WideResNet
DFW	79.79	90.89	91.45
Adam	85.59	91.45	91.31
SGD (with schedule)	82.26	92.79	90.57
DFW multistep	**77.21**	**92.13**	**92.40**

TABLE II
TEST ACCURACY (%): RESULTS FROM 50 TRAINING EPOCHS ON THE *CIFAR10* DATASET USING DIFFERENT ARCHITECTURES.

	DenseNet	GoogLeNet	WideResNet
DFW	56.69	71.71	69.37
Adam	55.57	69.62	68.29
SGD (with schedule)	51.10	73.13	68.51
DFW multistep	**56.78**	**72.32**	**71.56**

TABLE III
TEST ACCURACY (%): RESULTS FROM 50 TRAINING EPOCHS ON THE *CIFAR100* DATASET USING DIFFERENT ARCHITECTURES.

	DenseNet	GoogLeNet	WideResNet
DFW	58.21	158.54	309.34
Adam	61.56	161.43	310.78
SGD (with schedule)	59.12	155.44	308.22
DFW multistep	**66.72**	**165.67**	**321.46**

TABLE IV
AVERAGE TIME PER EPOCH (SECONDS): RESULTS FROM 50 TRAINING EPOCHS USING DIFFERENT ARCHITECTURES.

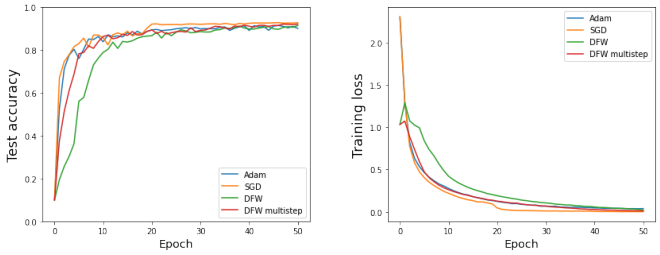

Fig. 2. Test accuracy and training loss on the CIFAR10 dataset using a GoogLeNet architecture. We see the superior and more stable behaviour of the multistep algorithm compared to the standard one.

Table II and Table III show that our multistep algorithm consistently outperforms the original DFW algorithm proposed in [Berrada et al., 2018]. Table IV shows that we achieve this while requiring a computational time per epoch which is comparable to the one of the single step algorithm. Our intuition is that, by solving more accurately the proximal problem in the early stages, we get learning "on the good track" and this yields better results. Figure 2 shows the most clear example of the advantage of our approach. Complete plots of the training trends are shown in Appendix E for space reasons.

IV. CONCLUSIONS

We have introduced the *Block Coordinate Descent* algorithm as an efficient and quick alternative to the state-of-the-art-optimizers in the training of deep neural networks. We combined it with commonly used optimizers and we proposed an alternative update for the additional variable V_N which generalizes the algorithm to the case of any differentiable loss function. We then presented the *Deep Frank-Wolfe* algorithm and we showed that more stable training trends can be obtained by performing multiple proximal steps in the dual. To conclude, these two algorithms can be employed to train deep neural networks, but only in very limited settings. The *Deep Frank-Wolfe* algorithm requires the loss function to be convex and piece-wise linear, which is not often the case. On the other hand, the *Block Coordinate Descent* algorithm has closed form solution for every update only for ReLU activation functions, squared loss objective and fully connected layers. Moreover, we showed in Appendix C the difficulties that the algorithm faces when we add sparse layers (e.g. convolutional ones).

REFERENCES

L. Berrada, A. Zisserman, and M. P. Kumar. Deep frank-wolfe for neural network optimization. *arXiv preprint arXiv:1811.07591*, 2018.

S. Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8 (3-4):231–357, 2015.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL http://jmlr.org/papers/v12/duchi11a.html.

M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval research logistics quarterly*, 3(1-2):95–110, 1956.

H. Karimi, J. Nutini, and M. Schmidt. Linear convergence of gradient and proximal-gradient methods under the polyak-łojasiewicz condition. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 795–811. Springer, 2016.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

S. Lacoste-Julien, M. Jaggi, M. Schmidt, and P. Pletscher. Block-coordinate Frank-Wolfe optimization for structural SVMs. In *Proceedings of the 30th International Conference on Machine Learning*, pages 53–61, 2013. URL https://proceedings.mlr.press/v28/lacoste-julien13.html.

G. Taylor, R. Burmeister, Z. Xu, B. Singh, A. Patel, and T. Goldstein. Training neural networks without gradients: A scalable admm approach, 2016. URL https://arxiv.org/abs/1605.02026.

J. Zeng, T. T.-K. Lau, S. Lin, and Y. Yao. Global convergence of block coordinate descent in deep learning. In *International conference on machine learning*, pages 7313–7323. PMLR, 2019.

P. Zhou, J. Feng, C. Ma, C. Xiong, S. Hoi, and W. E. Towards theoretically understanding why sgd generalizes better than adam in deep learning, 2020. URL https://arxiv.org/abs/2010.05627.

APPENDIX

A. Update of V_N using gradient descent

The update of V_N in the *Block Coordinate Descent* algorithm is given by

$$V_N^k = \text{argmin}_{V_N}\{R_n(V_N;Y) + \frac{\gamma}{2}\left\|V_N - U_N^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|V_N - V_N^{k-1}\right\|_F^2\}. \tag{13}$$

In the specific case in which the loss function l is the cross-entropy loss we can easily compute the gradient of the objective in (13) with respect to V_N. If C denotes the size of the output layer and Y denotes the one-hot encoding of the target output we can write

$$\nabla_{V_N}(l(V_N;Y) + \frac{\gamma}{2}\left\|V_N - U_N^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|V_N - V_N^{k-1}\right\|_F^2) = \frac{\exp(V_N)}{\sum_{i=1}^C \exp(V_N^{k-1,i})} - Y + \gamma(V_N + U_N^{k-1}) + \alpha(V_N - V_N^{k-1}),$$

so the update reads

$$V_N^k = V_N^{k-1} - \eta_k\left[\frac{\exp(V_N^{k-1})}{\sum_{i=1}^C \exp(V_N^{k-1,i})} - Y + \gamma(V_N^{k-1} + U_N^{k-1})\right], \tag{14}$$

where $V_N^{k-1,i}$ denotes the value for the i^{th} node of the output layer at the $(k-1)$-th iteration.

B. Alternative Approaches for the update of V_N

In contrast with our approach in Appendix A, in Zeng et al. [2019] V_N is updated alternatively by employing a prox-linear strategy. Upon the notation of Appendix A, calculating the closed form solution in the case of the squared loss gives

$$\begin{aligned}&\frac{\partial}{\partial V_N}(< \nabla_{V_N^{k-1}}\left\|V_N^{k-1} - Y\right\|_F^2, V_N - V_N^{k-1} > + \\ &\frac{\gamma}{2}\left\|V_N - V_N^{k-1}\right\|_F^2 + \frac{\alpha}{2}\left\|V_N - U_N^{k-1}\right\|_F^2) = 0 \\ &\Rightarrow \nabla_{V_N^{k-1}}\left\|V_N^{k-1} - Y\right\|_F^2 + \\ &\gamma(V_N - V_N^{k-1}) + \alpha(V_N - U_N^{k-1}) = 0.\end{aligned}$$

Now solving for V_N we get:

$$V_N = \frac{1}{\gamma+\alpha}(\gamma V_N^{k-1} + \alpha U_N^{k-1} + Y - V_N^{k-1}).$$

C. Convolutions, Perceptrons and Block Coordinate Descent

In the original publication for block Coordinate Descent it is mentioned that the algorithm can be used for Convolution networks, since they can be modeled as fully connected layers with sparse matrices for the weights. However, empirically we found the performance for a convolutional network to be way lower than that of a simpler perceptron, as shown in Table V.

	MultilayerPerceptron	Convolution
Test Accuracy	95.16	72.18

TABLE V
RESULTS FROM 100 TRAINING EPOCHS ON THE *MNIST* DATASET USING A PERCEPTRON AND A CONVOLUTION AS AN ARCHITECTURE.

Even though the result might seem peculiar at first glance we can notice that maintaining the graph of weights sparse hinders the training process, since this is an algorithm that employs a closed form solution and not a gradient method. So after updating the weights with the algorithm the new weights will more than likely no longer be sparse, since the closed form solution does not take into consideration this requirement. More formally, we can express the new optimization problem for a Convolution layer as follows:

$$\begin{aligned}\text{minimize} \quad & \frac{\gamma}{2}\left\|U_i^{iter} - W_i V_{i-1}^{iter-1}\right\|_F^2 + \frac{\alpha}{2}\left\|W_i - W_i^{iter-1}\right\|_F^2 \\ \text{subject to} \quad & \\ & \langle T, W_i^+\rangle_F = 0 \\ & \langle T, W_i^-\rangle_F = 0 \\ & W_i = W_i^+ + W_i^- \\ & \text{All elements of } W_i^+ \text{ are non-negative} \\ & \text{All elements of } W_i^- \text{ are non-positive,}\end{aligned} \tag{15}$$

where $\langle\cdot,\cdot\rangle_F$ denotes the Frobenius inner product, the matrix T corresponds to the binary complement of the filter that defines the sparse matrix of the convolution and we use the notation and the objective function of line 8 in Algorithm 1. We can easily notice that the previously stated program is a Quadratic

Program and therefore it does not have any trivial closed-form solutions, since its sets of constraints is non-trivial. This implies that we necessarily observe a loss of information when directly updating a convolutional layer with the algorithm, which uses a closed form solution.

A slightly more formal statement is that our quadratic program has no closed form solution, comes from the fact that the following quadratic program is known to be NP-hard and ours comes as a direct generalization to it.(D is any matrix)

$$\begin{aligned} &\text{minimize} && x^T D x \\ &\text{subject to} && x \geq 0, \end{aligned} \tag{16}$$

D. Single step proximal Frank-Wolfe

In this section, we show for the sake of completeness the complete *Deep Frank-Wolfe* algorithm in the case of a single proximal step in Algorithm 2. Because the conditional gradient δ_1 is always a feasible direction in the dual (see Berrada et al. [2018]) and we don't add any regularization term, the expression for the optimal step-size becomes

$$\gamma_1 = \frac{\eta \lambda_s}{||w_s||^2} = \frac{\lambda_s}{\eta ||\delta_1||^2}, \tag{17}$$

where we used the fact that $\lambda_1 = 0$ and

$$\begin{aligned} w_s &= -A s_1 \\ &= -\eta \left[\left((\partial f_{x,\bar{y}}(w)|_{w_0} - \partial f_{x,y}(w)|_{w_0})_{\bar{y} \in \mathcal{Y}} \right)^T s_1 \right] = -\eta \delta_1. \end{aligned}$$

This leads to the *Deep Frank-Wolfe* algorithm presented in Algorithm 3.

Algorithm 3: Deep Frank-Wolfe Algorithm

Data: Proximal coefficient η, initial point $w_0 \in \mathbb{R}^p$, momentum coefficient μ

$t = 0$
$z_0 = 0$
$t = 1$
for *each epoch* **do**
  for *each mini-batch* **do**
    Receive mini-batch of data $(x_i, y_i)_{i \in \mathcal{B}}$
    $\forall i \in \mathcal{B}$, $b_t^{(i)}(w_t) \leftarrow$ $(f_{x_i,\bar{y}}(w_t) - f_{x_i,y_i}(w_t) + \Delta(\bar{y}, y_i))_{\bar{y} \in \mathcal{Y}}$
    $\forall i \in \mathcal{B}$, compute a dual direction $s_t^{(i)} \in \mathcal{P}$
    $\delta_t = \partial \left(\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (s_t^{(i)})^T b_t^{(i)} \right) |_{w_t}$
    $\gamma_t = \frac{\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (s_t^{(i)})^T b_t^{(i)}(w_t)}{\eta ||\delta_t||^2}$ clipped to $[0, 1]$
    $z_{t+1} = \mu z_t - \eta \gamma_t \delta_t$
    $w_{t+1} = w_t - \eta \gamma_t \delta_t + \mu z_{t+1}$
    $t \leftarrow t + 1$
  end
end

E. Frank-Wolfe - complete training results

In this section we present the complete training results for the *Deep Frank Wolfe* algorithm in comparison to the state-of-the-art optimizers. In the figures below the training loss is intended to be the cross entropy loss for SGD and Adam and Multi-class Hinge loss for the DFW algorithms.

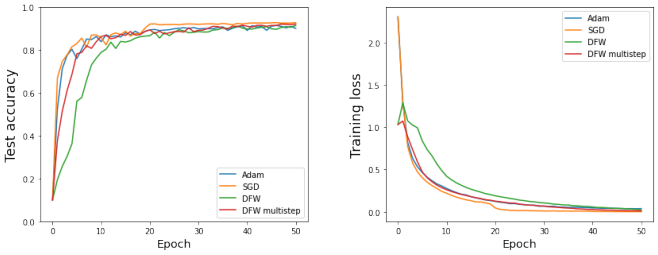

Fig. 3. Top-1 test accuracy and training loss on the CIFAR10 dataset using a GoogLeNet architecture.

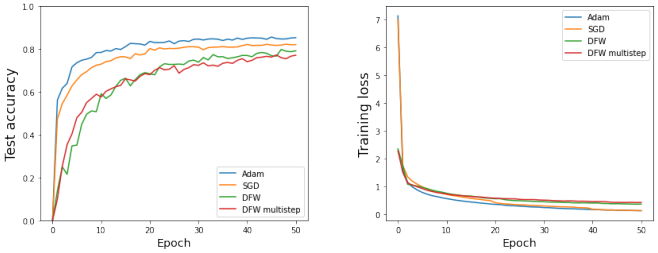

Fig. 4. Top-1 test accuracy and training loss on the CIFAR10 dataset using a DenseNet architecture.

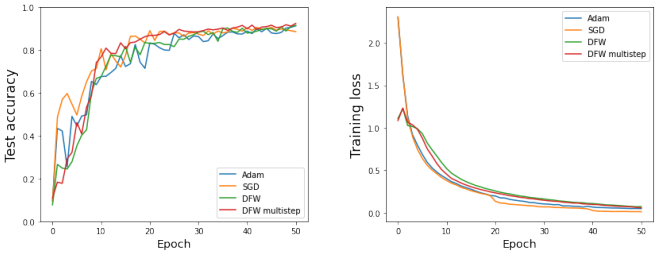

Fig. 5. Top-1 test accuracy and training loss on the CIFAR10 dataset using a WideResNet architecture.

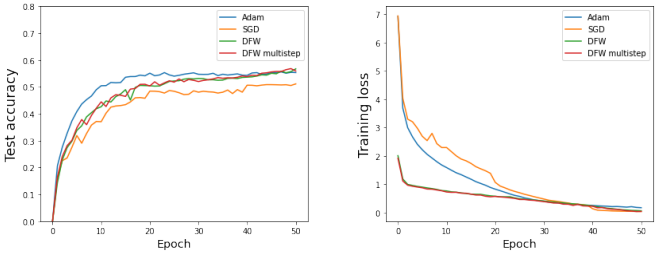

Fig. 6. Top-1 test accuracy and training loss on the CIFAR100 dataset using a DenseNet architecture.

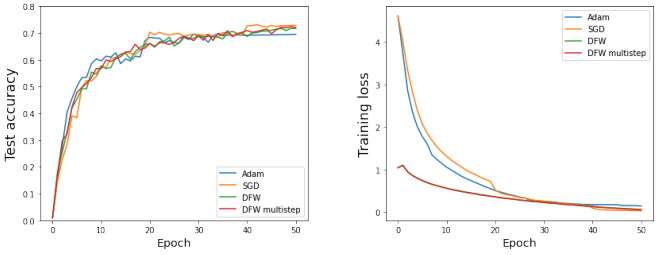

Fig. 7. Top-1 test accuracy and training loss on the CIFAR100 dataset using a GoogLeNet architecture.

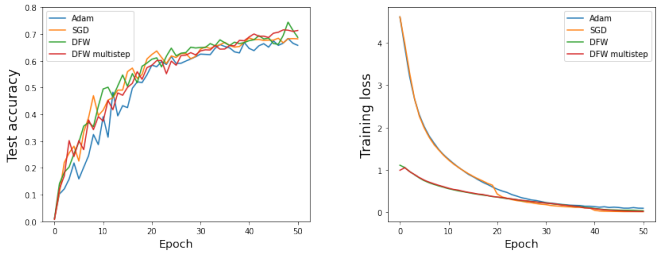

Fig. 8. Top-1 test accuracy and training loss on the CIFAR100 dataset using a WideResNet architecture.

F. Coordinate Descent - complete training results

In this section we present the complete training results for the *Block Coordinate Descent* algorithm in comparison to the state-of-the-art optimizers. Below we provide the accuracies, the average time and the plots for training loss and test accuracy for $FMNIST$ and $CIFAR10$. We also present the training loss and the test accuracies for the Block Coordinate Descent + GD optimizer and the prox linear update in Figure F and Figure F, respectively.

	Test Accuracy	Average Time Per Epoch
BCD	84.23	0.98
Adam	87.59	9.50
Prox Linear	84.14	0.98
BCD (GD update)	84.29	1.01
SGD (with schedule)	86.51	9.34
BCD + Adam	87.17	4.36
BCD + SGD	85.59	4.32

TABLE VI
RESULTS FROM 50 TRAINING EPOCHS ON THE *FMNIST* DATASET USING A 4 LAYER PERCEPTRON AS AN ARCHITECTURE.

	Test Accuracy	Average Time Per Epoch
BCD	48.46	9.31
Adam	42.92	11.43
BCD (GD update)	47.75	9.31
Entropy	47.33	9.34
SGD (with schedule)	48.70	11.25
BCD + Adam	47.33	10.17
BCD + SGD	48.33	10.09

TABLE VII
RESULTS FROM 50 TRAINING EPOCHS ON THE *CIFAR10* DATASET USING A 4 LAYER PERCEPTRON AS AN ARCHITECTURE.

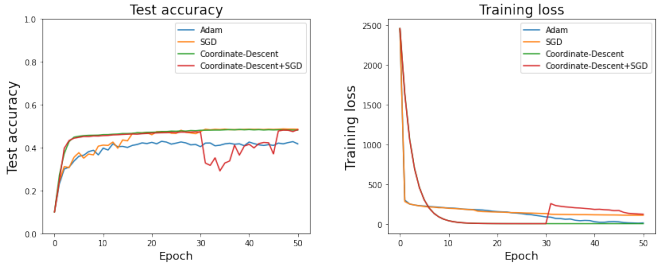

Fig. 9. Test accuracy and training loss on the CIFAR10.

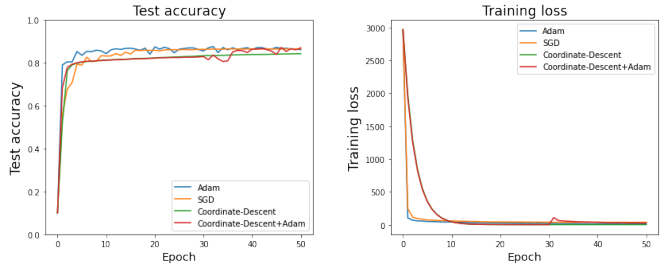

Fig. 10. Test accuracy and training loss on the FMNIST.

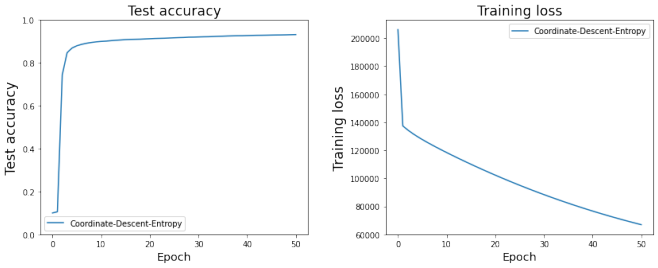

Fig. 11. Test accuracy and training loss on the MNIST dataset using Block Coordinate Descent + GD update

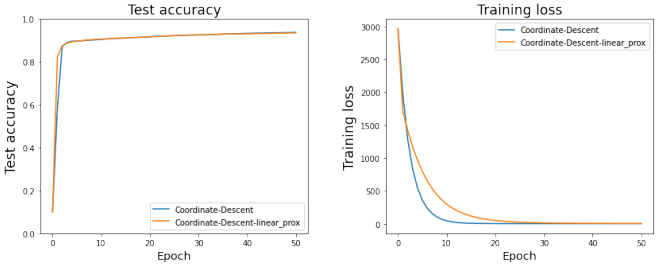

Fig. 12. Test accuracy and training loss comparison on the MNIST dataset using Block Coordinate Descent and Block Coordinate Descent with the prox linear update