Practical 2: Scalable Methods for Classification

1. Introduction

The objective of this practical is to familiarize yourself with stochastic optimization methods, and to compare custom solvers with black-box algorithms. Throughout the lab, we use classification of hand-written digits to illustrate these two points.

In the first part of the lab, we compare the performance of gradient descent with its stochastic counter-part. This will show the importance of a stochastic methods when the number of samples becomes large.

In the second part, we implement a custom solver for linear Support Vector Machines (SVMs). We compare its robustness and speed of convergence to stochastic sub-gradient descent.

2. Classification Problem

In the previous practical, we have defined and manipulated loss functions for regression. The difference with regression is that instead of predicting a value in $\mathbb{R}$, the model is required to predict a label in a discrete set of possible classes $\mathcal{C} = \{1, 2, ..., C\}$. The objective is to learn a linear model parameterized by $w \in \mathbb{R}^{C \times d}$. Given an input $x \in \mathbb{R}^d$, the prediction of the model is given by:

$$\hat{y} = \arg\max_{\bar{y} \in \mathcal{C}} w_{\bar{y}}^\top x.$$

Note that since $w \in \mathbb{R}^{C \times d}$, each row $w_{\bar{y}} \in \mathbb{R}^d$. As in the previous practical, the learning objective is the sum of a surrogate loss function and a regularization term:

$$\min_{w \in \mathbb{R}^{C \times d}} \frac{1}{N} \sum_{i=1}^{N} loss(w, x_i, y_i) + \frac{\mu}{2} \|w\|^2.$$

Note that we use $\| \cdot \|$ to denote the Frobenius norm: for a matrix $U = (U_{ij})_{i \in \mathcal{I}, j \in \mathcal{J}}$, $\|U\| = \sqrt{\sum_{i \in \mathcal{I}, j \in \mathcal{J}} U_{ij}^2}$.

In this practical we will explore two loss functions: the cross-entropy loss and the hinge loss.

3. The MNIST data set

Examples from the MNIST data set

We use the MNIST data set as our guiding example for classification. The MNIST dataset consists in 60,000 samples for training and validation, and 10,000 samples for testing. Each sample is a 28x28 gray-scale image representing a hand-written digit in $\{0, 1, ..., 9\}$. We flatten these images to obtain vectors of $\mathbb{R}^{784}$. The trainval data set is split into 50,000 samples for training and 10,000 samples for validation.

4. The Importance of Stochasticity: Illustration with (Sub-)Gradient-Based Approaches

4.1. The Cross Entropy Loss

The cross-entropy loss minimizes the Kullback-Leibler divergence between the empirical distribution of the data and the probability predicted by the model. More details are available in this Wikipedia article if needed.

The cross-entropy loss can be written down as:

$$L_{CE} : (w, x, y) \in \mathbb{R}^{C \times d} \times \mathbb{R}^d \times \mathcal{C} \mapsto \log \left(\sum_{k \in \mathcal{C}} \exp(w_k^\top x) \right) - w_y^\top x$$

When the cross-entropy loss is combined with quadratic regularization ($\frac{\mu}{2} \|w\|^2$ in our case), the resulting problem is called "logistic regression".

Question 1 (Optional): Show that the cross-entropy loss is convex.

Hint: log-sum-exp is convex (proof here), and composition with an affine map preserves convexity.

Question 2: Derive the gradient of cross-entropy.

Hint: it can be easier to derive this independently for each row w_k of w

Task 1: Implement the logistic regression objective and its gradient in `objective/logistic.py`. Feel free to use inefficient for loops as a first step. You can test this with `python run_test.py TestObj_Logistic_Gradient`. Once the tests pass, you can use a `torch` vectorized implementation as documented here.

4.1.1. Gradient Descent

With a cross-entropy loss, the objective function is smooth, therefore we can use gradient descent (GD) with a constant step-size. To begin with, we test the use of GD on a subset of MNIST:

Task 2: Run GD on a subset of MNIST (of size 1000) with the command `python main.py --dataset mini-mnist --opt gd --obj logistic`. Note that the step-size can be tuned with the command-line arguments, for instance by using `--init-lr 0.1` to set it to 0.1.

Question 3: How does the time complexity of one iteration of gradient descent grow with N the number of samples?

Task 3: Run GD on the full MNIST data set with the command `python main.py --dataset mnist --opt gd --obj logistic`.

You should be able to observe that the runtime of GD becomes very slow, because of its large cost per iteration. In machine learning applications, a data set of size 60,000 samples is relatively small and it is not uncommon to work with data sets with millions or even billions of samples. Clearly, this motivates the need for approaches whose cost per iteration is independent of the size of the data set N.

4.1.2. Stochastic Gradient Descent

Task 4: Implement the gradient descent step in the method `SGD._step` of `optim/sgd.py`. You can assume that the learning rate `lr` and the gradient `dw` are given by the `oracle_info` dictionary given in argument. This can be tested with `python run_test.py TestOpt_SGD`.

Task 5: Run SGD on the MNIST data set with the command `python main.py --dataset mnist --opt sgd --obj logistic`.

Note that you can choose the batch-size, for instance with the option '`--batch-size 64` if you wish to use a batch-size of 64.

You will observe that for appropriate choices of batch-size, the convergence of SGD is significantly faster than for GD.

As a side note, you may remark that we are monitoring an inexact estimate of the objective function. Indeed, observing the exact value of the objective function would require a pass on the entire dataset, which is computationally expensive. Instead, when performing an update on a mini-batch, we monitor the current value of the objective on this particular mini-batch (before the update). These values are then averaged over the mini-batches at each epoch, and form an estimate of the objective function. This estimate is inexpensive and most often good enough for monitoring purposes.

4.2. The Hinge Loss

The hinge loss aims at creating a margin of one between correct predictions and incorrect ones:

$$L_{hinge} : (w, x, y) \in \mathbb{R}^{C \times d} \times \mathbb{R}^d \times \mathcal{C} \mapsto \max_{k \in \mathcal{C}} \left\{ w_k^\top x + \Delta(k, y) - w_y^\top x \right\},$$

Where the function Δ is defined as follows: $\Delta(k, y) = 0$ if $k = y$ and $\Delta(k, y) = 1$ if $k \neq y$.

When the hinge loss is combined with a quadratic regularization, we obtain a "maximum-margin classifier" formulation, that is a formulation that maximizes the minimal distance between correct predictions and incorrect ones. Such a model is also known as Support Vector Machines (SVMs).

4.2.1. Stochastic Sub-Gradient Descent

Question 4: Why is the hinge loss not smooth? Derive a valid sub-gradient for the hinge loss.

Hint: it can be easier to derive this independently for each row w_k of w

Task 6 (optional): Implement the hinge loss function and its sub-gradient. Feel free to use inefficient for loops as a first step. You can test the implementation with `python run_test.py TestObj_SVM_SubGradient`. Once the tests pass, you can use a vectorized implementation provided here.

Task 7: Run SSGD on the MNIST dataset with the command `python main.py --dataset mnist --opt sgd --obj svm`. Note that again, you can tune the initial learning rate and the batch-size.

5. A Conditional Gradient Method

In this section, we will derive the dual of the SVM and a conditional gradient algorithm that exploits the special structure of this dual. In particular, we will see how an optimal step-size can be computed in closed-form at each iteration and how a stochastic approach in the primal translates into a block-coordinate approach in the dual.

5.1. SVM Dual

You have seen in the lecture that the SVM dual can be written as:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^{NC}} \quad & f(\alpha) \triangleq -\frac{1}{2}\alpha^\top Q \alpha + b^\top \alpha \\ \text{subject to: } & \forall\, i \in \{1, ..., N\}, \forall\, \bar{y}_i \in \mathcal{C}, \alpha_{i\bar{y}_i} \geq 0 \\ & \forall\, i \in \{1, ..., N\}, \sum_{\bar{y}_i \in \mathcal{C}} \alpha_{i\bar{y}_i} = 1 \end{aligned}$$

Note that we use the following indexing on the vector $\alpha \in \mathbb{R}^{NC}$: $\alpha_{i\bar{y}_i}$ is the dual variable corresponding to sample i and class $\bar{y}_i$. In addition, it will be simpler to use the following notation (this is valid because Q is defined as an outer product):

$$f(\alpha) = -\frac{\mu}{2}\|A\alpha\|^2 + b^\top \alpha.$$

We remind the following useful information:

Definition of A:

For each sample i and each class $\bar{y}$, we construct the augmented feature vector $\Phi_i(\bar{y}) \in \mathbb{R}^{Cd}$ as follows:

$$\Phi_i(\bar{y}) = \begin{pmatrix} (0) \\ x_i \text{ (at index } y_i) \\ (0) \end{pmatrix} - \begin{pmatrix} (0) \\ x_i \text{ (at index } \bar{y}) \\ (0) \end{pmatrix}$$

Then $A \in \mathbb{R}^{Cd \times NC}$ is the stack of (column) feature vectors for each sample and each class:

$$A = \frac{1}{\mu N}\Big(\Phi_1(1),\ \Phi_1(2),\ \ldots,\ \Phi_1(C),\ \Phi_2(1),\ \ldots,\ \Phi_N(C)\Big) \in \mathbb{R}^{Cd \times NC}$$

Definition of b:

$$b = \Big(\frac{\Delta(1, y_1)}{N},\ \frac{\Delta(2, y_1)}{N},\ \ldots,\ \frac{\Delta(C, y_1)}{N},\ \frac{\Delta(1, y_2)}{N},\ \ldots,\ \frac{\Delta(C, y_N)}{N}\Big)^\top \in \mathbb{R}^{NC}$$

KKT conditions:

$$w = A\alpha.$$

The objective $f(\alpha)$ is smooth (quadratic) and concave, and the the feasible set for α is a compact domain (that is, a closed and bounded domain). We can therefore apply the conditional gradient / Frank-Wolfe algorithm (these two names refer to the same algorithm). We detail this algorithm below (note that the description will apply to the minimization of $-f$ which is equivalent to the maximization of f).

5.2. The Conditional Gradient

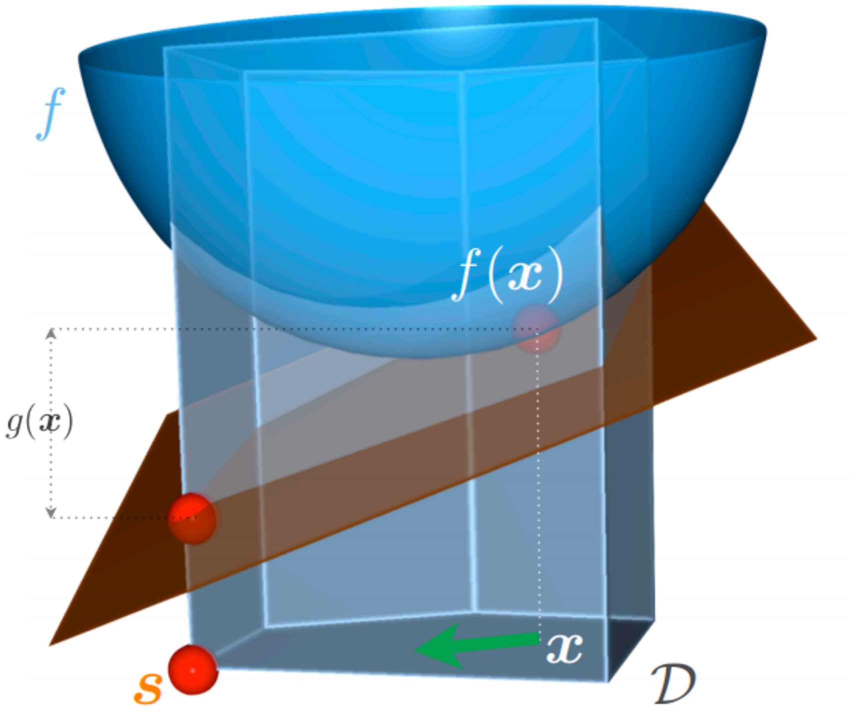

Illustration of the Conditional Gradient.

We suppose we are given a convex smooth objective function (blue bowl above), which is constrained over a compact domain $\mathcal{D}$ (in dark blue above). If we are at the current iterate $\alpha^{(t)} \in \mathcal{D}$ (x in figure), the conditional gradient linearizes the objective at the current point (brown plane), and finds the minimizer s (in red) of this linearized objective subject to the constraints $\mathcal{D}$. Then the next iterate is a convex combination of x and s (update with a green arrow): $\alpha^{(t+1)} = \gamma s^{(t)} + (1-\gamma)\alpha^{(t)}$. Note that since $\mathcal{D}$ is convex, necessarily $\alpha^{(t+1)} \in \mathcal{D}$. This is summarized in the following algorithm:

Algorithm 1: Conditional Gradient

```
• Initialization: $\alpha^{(0)} \in \mathcal{D}$, $t = 0$
• While not converged:
1. Linearization: $g = \frac{\partial f(\alpha)}{\partial \alpha}\Big|_{\alpha=\alpha^{(t)}}$
2. Compute the conditional gradient: $s^{(t)} = \arg\min_{s\in\mathcal{D}}\{s^\top g\}$
3. Pick a step size $\gamma \in [0,1]$.
4. Update: $\alpha^{(t+1)} = (1-\gamma)\alpha^{(t)} + \gamma s^{(t)}$
5. $t = t + 1$.
```

5.3. Conditional Gradient for Dual SVM: Optimal Step-Size

Question 5 (Optional): Derive the conditional gradient for the dual of the SVM (step 2 of the above algorithm). Can you relate it to the subgradient of the primal from Question 4?

Hint: the KKT conditions give a way to relate primal and dual variables. In addition, recall how the matrix A has been constructed from the x_i.

Question 6: We have to choose a step-size $\gamma \in [0,1]$ at step 3 of the algorithm. It turns out that the optimal value γ^* can be computed in closed-form. What equation does γ^* have to satisfy to be optimal?

Hint: the resulting objective value is given by $f((1-\gamma)\alpha^{(t)} + \gamma s^{(t)})$.

Question 7: Solve this equation in γ to obtain γ^* in closed form.

Let us look at the computational cost of this formulation. The largest element to store is A. The most expensive operation is a matrix-vector multiplication $A\alpha$. Since A is of size $Cd \times NC$ and α of size NC, this corresponds to a memory complexity of $\mathcal{O}(NdC^2)$ and a time complexity of $\mathcal{O}(NdC^2)$ per iteration.

In comparison, the SGD algorithm has a memory complexity of $\mathcal{O}(dC)$, and a time complexity of $\mathcal{O}(dC)$ per iteration.

Question 8: Convince yourself that SGD and BCFW have the memory and time complexities given above.

5.4. Improvement 1: Block-Coordinate Ascent

The algorithm derived so far has a time complexity per iteration that still grows linearly with N.

Fortunately, we can adapt our algorithm in a Block-Coordinate version. The resulting time complexity will not depend on N and yet we will still be able to compute an optimal step-size in closed form.

To see that, we derive an example with $N = 2$ and the result will extend easily to any value of $N \geq 2$. When $N = 2$, we can write the dual problem as:

$$\begin{aligned}
\max_{\alpha \in \mathbb{R}^{2C}} \quad & f(\alpha) \triangleq -\frac{\mu}{2}\|A\alpha\|^2 + b^\top \alpha \\
\text{subject to: } & \forall\, \bar{y_1} \in \mathcal{C}, \alpha_{1\bar{y_1}} \geq 0 \\
& \forall\, \bar{y_2} \in \mathcal{C}, \alpha_{2\bar{y_2}} \geq 0 \\
& \sum_{\bar{y_1} \in \mathcal{C}} \alpha_{1\bar{y_1}} = 1 \\
& \sum_{\bar{y_2} \in \mathcal{C}} \alpha_{2\bar{y_2}} = 1
\end{aligned}$$

We now use explicit block matrices to reveal the structure of the problem:

$$\alpha = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \quad A = \begin{pmatrix} A_1, A_2 \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

Then the dual problem can be decomposed as follows:

$$\begin{aligned}
\max_{\alpha_1, \alpha_2 \in \mathbb{R}^{C}} \quad & f(\alpha) \triangleq -\frac{\mu}{2}(\|A_1\alpha_1 + A_2\alpha_2\|^2) + b_1^\top \alpha_1 + b_2^\top \alpha_2 \\
\text{subject to: } & \forall\, \bar{y_1} \in \mathcal{C}, \alpha_{1\bar{y_1}} \geq 0 \\
& \forall\, \bar{y_2} \in \mathcal{C}, \alpha_{2\bar{y_2}} \geq 0 \\
& \sum_{\bar{y_1} \in \mathcal{C}} \alpha_{1\bar{y_1}} = 1 \\
& \sum_{\bar{y_2} \in \mathcal{C}} \alpha_{2\bar{y_2}} = 1
\end{aligned}$$

Note that we will keep using the notation $A\alpha = A_1\alpha_1 + A_2\alpha_2$.

Let us assume we want to perform an update on α_1 alone. We will find a search direction for the block of coordinates of α_1 (instead of the full α). Given this update, we will compute an optimal step size for the overall objective $f(\alpha)$:

Algorithm 2: Block-Coordinate Conditional Gradient

```
• Initialization: $\alpha_1^{(0)}, ..., \alpha_N^{(0)} \in \mathcal{D}$, $t = 0$
• While not converged:
• For i=1..N (in random order):
    1. Linearization: $g_1 = \frac{\partial f(\alpha_1, ..., \alpha_N)}{\partial \alpha_i}\Big|_{\alpha_1=\alpha_1^{(t)}, ..., \alpha_N=\alpha_N^{(t)}}$
```

```
2. Compute the conditional gradient: $s^{(t)} = \arg\max_{s\in\mathcal{D}}\{s^\top g\}$
3. $\gamma^* = \arg\min_{\gamma\in[0,1]} f(\alpha_1, ..., \underbrace{(1-\gamma)\alpha_i^{(t)} + \gamma s^{(t)}}_{\text{index } i}, ..., \alpha_N)$.
4. Update i: $\alpha_i^{(t+1)} = (1-\gamma^*)\alpha_i^{(t)} + \gamma^* s^{(t)}$
5. Maintain other coordinates: $\forall j \neq i$, $\alpha_j^{(t+1)} = \alpha_j^{(t)}$.
6. $t = t+1$.
```

Question 9: Derive the conditional gradient s (steps 1 and 2 of Algorithm 2) for a block-update on α_1.

Question 10: Derive the optimal step-size γ^* (step 3 of Algorithm 2) given this conditional gradient direction.

You should obtain the following algorithm:

Algorithm 3: Block-Coordinate Conditional Gradient for SVM (Dual)

```
- Initialization: $\alpha_1^{(0)}, ..., \alpha_N^{(0)} \in \mathcal{D}$, $t = 0$
- While not converged:
- For i=1..N (in random order):
    1. $s^{(t)} = \arg\max_{s\in\mathcal{D}}\{-\mu s^\top A_i^\top A\alpha + b_i^\top s\}$
    2. $\gamma^* = \dfrac{-\mu(A_i s - A_i\alpha_i)^\top A\alpha + b_i^\top s - b_i^\top \alpha_i}{\mu\|A_i s - A_i\alpha_i\|^2}$.
    3. Update i: $\alpha_i^{(t+1)} = (1-\gamma^*)\alpha_i^{(t)} + \gamma^* s^{(t)}$
    4. Maintain other coordinates: $\forall j \neq i$, $\alpha_j^{(t+1)} = \alpha_j^{(t)}$.
    5. $t = t+1$.
```

5.5. Improvement 2: Primal-Dual Approach

Look at the equations of Algorithm 3. Most of the cost stems from the explicit use of the matrices A_i and A, (of respective sizes $\mathcal{O}(C^2d)$ and $\mathcal{O}(NC^2d)$). In order to reduce the computational cost, we propose to use the following variables:

$$w = A\alpha, \quad w_j = A_j\alpha_j \text{ (for all blocks } j), \quad w_s = A_i s^{(t)} \text{ (when search at block } i), l = b^\top\alpha, \quad l_j = b_j^\top \alpha_j \text{ (for all b}$$

Question 11: Use the above definitions to write Algorithm 3 in terms of w, w_j, w_s, l, l_j, l_s. Can you simplify step 1 in terms of $L_{\text{hinge}}(w, x_i, y_i)$? Therefore, can you write w_s in terms of $\dfrac{\partial L_{\text{hinge}}(w, x_i, y_i)}{\partial w}$? Note how a stochastic estimate on a sample in the primal corresponds to a block of coordinate in the dual.

Such a formulation is usually called primal-dual, because it operates in the dual, but all the operations are expressed in terms of primal variables for efficiency reasons.

Since we maintain only primal variables instead of dual ones, we need to initialize these primal variables in a way that they correspond to feasible dual variables. For instance, we have that $w = A\alpha$, so we need to initialize w so that there exists a feasible vector $\alpha \in \mathcal{D}$ that satisfies $w = A\alpha$. It can be shown that initializing $w^{(0)}, w_1^{(0)}, ..., w_N^{(0)} = 0$ and $l^{(0)}, l_1^{(0)}, ..., l_N^{(0)} = 0$ satisfies this condition (we omit the derivation details for this, they are available in e.g. this paper).

You should obtain the following algorithm:

Algorithm 4: Block-Coordinate Conditional Gradient for SVM (Primal-Dual)

```
- Initialization: $w^{(0)}, w_1^{(0)}, ..., w_N^{(0)} = 0$, $l^{(0)}, l_1^{(0)}, .., l_N^{(0)} = 0$, $t = 0$
- While not converged:
- For i=1..N (in random order):
    1. $y_i^* = \arg\max_{k \in \mathcal{C}} \left\{ w_k^\top x_i + \Delta(k, y_i) - w_{y_i}^\top x \right\}$
    2. $w_s = \frac{-1}{\mu N} \frac{\partial L_{\text{hinge}}(w, x_i, y_i)}{\partial w}$ and $l_s = \frac{1}{N} \Delta(y_i^*, y_i)$
    3. $\gamma^* = \dfrac{-\mu (w_s - w_i)^\top w + l_s - l_i}{\mu \| w_s - w_i \|^2}$.
    4. $w_i^{(t+1)} = (1 - \gamma^*) w_i^{(t)} + \gamma^* w_s$
    5. $l_i^{(t+1)} = (1 - \gamma^*) l_i^{(t)} + \gamma^* l_s$
    6. $w^{(t+1)} = w^{(t)} + w_i^{(t+1)} - w_i^{(t)}$
    7. $l^{(t+1)} = l^{(t)} + l_i^{(t+1)} - l_i^{(t)}$
    8. $t = t + 1$.
```

Question 12: What are the memory and time complexities of the primal-dual BCFW (Algorithm 4)?

5.6. Improvement 3: Time-Memory Trade-Off with Mini-Batches

We can make two observations from the previous section: (i) there are N copies w_i of w and (ii) the algorithm processes only one sample at a time. This is ineffective for large data sets, the difference is noticeable for CPUs, but would be even more the case if were using GPUs. To solve both of these problems, we define a batch-size of b, and we divide $\{1, ..., N\}$ into M mini-batches:

$$B_1 = \{1, ..., b\},\ B_2 = \{b + 1, ..., 2b\},\ ...,\ B_M = \{b(M - 1) + 1, ..., N\}$$

Note: there are $M = \mathit{ceiling}(N/b)$ mini-batches.

Then for each mini-batch $j \in \{1...., M\}$, we can define $w_j = \sum\limits_{i \in B_j} A_i \alpha_i$. Note that when $b = N$, we have $M = 1$ mini-batch of size N and we retrieve the non block-coordinate version. Conversely, for $b = 1$ we have $M = N$ mini-batches of size 1 and we recover the previous block-coordinate method. This results in the following final algorithm:

Algorithm 5: Block-Coordinate Conditional Gradient for SVM (Primal-Dual) with Mini-Batches

```
- Initialization: $w^{(0)}, w_1^{(0)}, ..., w_M^{(0)} = 0$, $l^{(0)}, l_1^{(0)}, .., l_M^{(0)} = 0$, $t = 0$
- While not converged:
- For j=1..M (in random order):
    1. For $i \in B_j$, $y_i^* = \arg\max_{k \in \mathcal{C}} \left\{ w_k^\top x_i + \Delta(k, y_i) - w_y^\top x \right\}$
    2. $w_s = \frac{-1}{\mu N} \sum_{i \in B_j} \frac{\partial L_{\text{hinge}}(w, x_i, y_i)}{\partial w}$ and $l_s = \frac{1}{N} \sum_{i \in B_j} \Delta(y_i^*, y_i)$
    3. $\gamma^* = \dfrac{-\mu(w_s - w_i)^\top w + l_s - l_i}{\mu \|w_s - w_i\|^2}$.
    4. $w_i^{(t+1)} = (1 - \gamma^*) w_i^{(t)} + \gamma^* w_s$
    5. $l_i^{(t+1)} = (1 - \gamma^*) l_i^{(t)} + \gamma^* l_s$
    6. $w^{(t+1)} = w^{(t)} + w_i^{(t+1)} - w_i^{(t)}$
    7. $l^{(t+1)} = l^{(t)} + l_i^{(t+1)} - l_i^{(t)}$
    8. $t = t + 1$.
```

Question 13: What are the memory and time complexities of the primal-dual BCFW with mini-batches (Algorithm 5)? Compare this to the time and memory complexities of SGD with a similar batch-size.

You will observe that BCFW and SGD obtain the same time complexity per iteration, and BCFW is more expensive memory-wise. This is a trade-off between time and memory: the additional memory used by BCFW allows it to compute an optimal step-size, which allows it to converge in a lower number of iterations than SGD. After all these derivations, you can now (finally!) observe the difference between the two methods in practice on MNIST:

Task 8: Implement the `oracle` and `task_error` methods method for the `SVM_ConditionalGradient` Objective in `objective/svm.py`. You can test your implementation using `python run_test.py TestObj_SVM_ConditionalGradient`.

Hint: the task error is misclassification for an SVM, and the function `accuracy` is provided in `utils.py`

Task 9: Implement the `BCFW` optimizer in `optim/bcfw.py`: you need to implement the optimal step-size and the updates in `_step(oracle_info)`. You can test this by running `python run_test.py TestOpt_BCFW`.

Task 10: Run the BCFW algorithm on MNIST with the command `python main.py --dataset mnist --opt bcfw`. Note that compare to GD or SGD, there is no need to tune a learning rate. Compare the empirical speed of convergence with stochastic sub-gradient descent (for instance, look at the objective value after one epoch for each method).

Task 11 (Optional): Vary the value of μ as a power of 10, and observe how this affects the speed of convergence of BCFW. Do you have an intuition about why that is?

Hint: how difficult to solve does the dual look if μ becomes infinitely large? Infinitely small?