Final Project, Optimization Methods for Data Science

Simone Di Gregorio, Mattia Castaldo

1. Multilayer Perceptron

1.1. An organised backpropagation framework

When implementing something complex like backpropagation, an essential thing to consider is keeping everything as tidy and ordered as possible. Programming-wise, in Python, this means using object-oriented programming as much as possible; mathematically speaking, this means outlining all the details in advance.

In our case, we used as reference for most of the linear algebra for backpropagation the slides from Professor Galasso from his course of *Fundamentals of Data Science*, a compulsory course at the *Data Science Master's Degree* course at Sapienza Università di Roma.

1.1.1. The dense layer

Having outlined some terminology/notation, let's go deep with the math. In what follows (and also in the next sections), L is the unregularized loss.

First of all, we unravel a bit of dense layer backpropagation; we start with a batch size of N observations, a matrix of weights $W \in \mathbb{R}^{D \times M}$, for a layer with an input $\mathbf{X} \in \mathbb{R}^{N \times D}$ and an output $\mathbf{Y} \in \mathbb{R}^{N \times M}$. The upstream gradient is $\frac{\partial L}{\partial \mathbf{Y}}$, having the same shape as Y, since the loss is scalar valued (either by summing or averaging over the batch axis).

Notice that, for a single output, the following expression holds:

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

Therefore, by the chain rule, multiplying upstream and local derivatives:

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

By translating the above result in matrix form we get the following downstream gradient:

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{W}^\intercal$$

A similar reasoning leads to the gradient matrix for the weights:

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^\intercal \frac{\partial L}{\partial \mathbf{Y}}$$

Finally, the gradient w.r.t. the bias vector is simply the upstream gradient $\frac{\partial L}{\partial \mathbf{Y}}$ summed over the batch axis. By linearity of differentiation, the regularization term of the loss is considered for the gradient weight matrix by simply performing an element wise addition between the gradient matrix and $2\rho\mathbf{W}$.

The layer object needs the number of units to be initialised, but everything else is setup **automatically** when the layer is added to a `Model` object; adding the layer triggers the execution of a `model_setup` method, which initialises the required arrays for forward and backward pass exploiting information about batch size and output shape (at the previous layer) stored in the `Model` object.

1.1.2. The hyperbolic tangent activation

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \tanh_\sigma(x)} \frac{\partial \tanh_\sigma(x)}{\partial x}$$
$$\tanh_\sigma(x) = \frac{\exp(2\sigma x) - 1}{\exp(2\sigma x) + 1}$$
$$\frac{\partial \tanh_\sigma(x)}{\partial x} = \frac{4\sigma \exp(2\sigma x)}{(\exp(2\sigma x) + 1)^2}$$

.

Therefore, in order to get the local gradient it is enough to compute the final expression for every entry of the input matrix $\mathbf{X}$, getting the matrix $\mathbf{G}$. The downstream gradient is obtained as follows, with notation identical to the one used to illustrate the linear layer:

$$\frac{\partial L}{\partial \mathbf{X}} = \mathbf{G} \odot \frac{\partial L}{\partial \mathbf{Y}}$$
$$g_{i,j} = \frac{\partial \tanh_\sigma(x_{i,j})}{\partial x}$$

The hyperbolic tangent layer is handled in a similar way to the one of the linear layer, as a class with a forward and a backward method, updating/storing the input in the forward pass and with a `model_setup` method to setup the layer when added to an MLP model object. The σ hyperparameter is passed when initialising the layer object.

1.1.3. Cross-entropy loss and sigmoid activation

We are asked to consider the average cross-entropy loss: we take the gradient with respect to the summed loss, we then scale to get the gradients for the average loss. Let's get started by saying that the cross-entropy loss is problematic in nature since it uses the logarithm: possible underflows or overflows in the exponential of the sigmoid due to high absolute values of the logits may cause the prediction to be 0 or 1, leaving the cross-entropy undefined. Therefore, we use the following numerically stable version to address this:

$$L(\omega, \pi) = -\frac{1}{p}\sum_{i=1}^{P}\left[y_i \ln(p_i + \epsilon) + (1 - y_i)\ln(1 - p_i + \epsilon)\right] + \rho||\omega||^2$$

The ϵ term (set to 10^{-7}) prevents problems with the logarithm, and it is easily handled in backpropagation, since it impacts only the shape of $\frac{\partial L(p_i)}{\partial p_i}$.

$$\frac{\partial L(p_i)}{\partial p_i} = -\frac{y}{p_i + \epsilon} + \frac{1 - y}{1 - p_i + \epsilon}$$

The derivative of the sigmoid w.r.t. to its input is straightforward, notoriously being: $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$. Therefore, to get the downstream gradient for the rest of the backpropagation (an N-dimensional vector, where N is the batch size), we just need to element-wise compute the following:

$$\frac{L(x_i)}{\partial x_i} = \frac{\partial L(p_i(x_i))}{\partial p_i}\frac{\partial p_i(x_i)}{\partial x_i} \; \forall \, i \in \{1, \dots, N\}$$

.

Notice that we are using the sigmoid as final activation function since it is just the two-class version of a non-overparametrized softmax activation (receiving in input $K - 1$ dimensional vectors for K probabilities in output/classes to consider).

1.2. The model object

The overall logic is wrapped in a model object with a specific set of methods, the relevant ones are the following two:

- The `evaluate_loss` method is the method which is exposed to `scipy.minimize`, taking in input the parameter vector, the training data and the related labels. It also requires number of epochs and seed number to

allow replicability of the shuffling for each epoch which leads to batch indexes. This is due to the fact that training is performed through mini-batch stochastic gradient descent.
- The `backprop` method is called by the `evaluate_loss` one and calls the individual layers' `backprop` method, starting from the last layer and going in reverse order. It iteratively gets the downstream gradient for each of the layers and save in a list pointers to weight and bias gradients for the Linear layers. At the end, it updates the overall gradient vector by concatenating the flattened gradients from the Linear layers.

The actual optimiser is external to the `Model` object, being a function implementing Adam according to the signature required by `scipy.minimize`.

2. Radial Basis Function Network

2.1. Generalized RBF network

Since solving the $P \times P$ system of a proper regularised RBF may be too expensive as P grows, generalized RBF networks are introduced, where the number of centers $N \ll P$, the size of the training set. In this case, both weights w and centers have to be determined, so the output depends on a nonlinear relation with the input due to the Radial Basis Function, similar to an MLP with a nonlinear activation function.

2.2. Training the network

The variables are decomposed in 2 blocks: centers and weights. Two steps are iteratively performed:

- w_{k+1} is determined using Newton-Rapshon with respect to the current centers c_k.
- c_{k+1} is updated with a backtracking line search (*Armijo*) along the gradient direction evaluated with respect to the new value w_{k+1}.

2.2.1. Minimization for w

The minimization of $L(w, c)$ with respect to w consists of a simple logistic regression where the feature matrix is Φ, the matrix of RBF outputs. We can leverage the well-known closed form for $\frac{\partial L}{\partial w}$ and $\frac{\partial^2 L}{\partial w \partial w^T}$, Considering $p(\Phi; w) = \sigma(\Phi w)$ as the vector containing the probabilities of belonging to the first class, we can note, leveraging the chain rule of the gradient:

$$\frac{\partial L}{\partial w} = \Phi^T \left[\left[-\frac{y}{p(\Phi;w)} + \frac{1-y}{1-p(\Phi;w)} \right] \cdot [p(\Phi;w)(1-p(\Phi;w))] \right] + 2\rho_1 w =$$

$$= \Phi^T \left[-y + p(\Phi;w) \right] + 2\rho_1 w$$

To solve this minimisation problem, we use the Newton-Raphson algorithm, which requires the Hessian matrix with respect to w:

$$\frac{\partial L}{\partial w w^T} = \Phi^T W \Phi + 2\rho_1 I$$

where W is a $P \times P$ diagonal matrix with the i-th diagonal element as $p(\phi_i, w)(1 - p(\phi_i, w))$, and I is the identity $N \times N$ matrix. So, the i+1-th Newton step is:

$$w_{i+1} = w_i - \left(\Phi^T W \Phi + 2\rho_1 I \right)^{-1} \left[\Phi^T \left[-y + p(\Phi;w) \right] + 2\rho_1 w \right]$$

The final w_i that we get at convergence is w_{k+1}.

2.2.2. Minimization for the centers

Regarding the optimization for the centers, the centers matrix C is an $N \times M$ matrix, where N is the number of units in the hidden layer and M is the dimension of the feature vector x. It is initialized by randomly selecting N points from the training set. As Radial Basis Function, we choose the Multiquadric, $\phi(x)_j = \sqrt{\sigma^2 + ||x - c_j||^2}$. Now, in order to compute $\frac{\partial L}{\partial C}$, we apply the backpropagation framework as presented in the first paragraph of MLP.

$$\overset{N \times M}{\frac{\partial L}{\partial C}} = \overset{P \times 1}{\frac{\partial L}{\partial p(\Phi;w)}} \odot \overset{P \times 1}{\frac{\partial p(\Phi;w)}{\partial \sigma}} \overset{1 \times N}{\frac{\partial \sigma}{\partial \Phi}^T} \odot \overset{P \times N \times M}{\frac{\partial \Phi}{\partial C}}$$

The downstream gradient with respect to the sigmoid activation function follows similar reasoning to the one described in the Multilayer Perceptron paragraph. Let's dive deeper into the math for the last element of the chain rule. We focus on a single element of Φ to simplify the notation:

$$\nabla_{c_j} \phi_{i,j} = -(||x^i - c_j|| + \sigma^2)^{-1/2} (x^i - c_j) \in \mathbb{R}^M$$

As shown above, for each element of Φ the gradient is computed w.r.t to the vector c_j with $j = 1, ..., N$, so $\frac{\partial \Phi}{\partial C}$ is a 3-D tensor ($P \times N \times M$). Implementation-wise, we add one axis to $\frac{\partial L}{\partial \Phi}$ to make it compatible with the last element of the chain. By averaging along the batch axis, we obtain the average

gradient of the cross-entropy loss with respect to the matrix of centers C. As per Armijo, Larry (1966), c_{k+1} is found with a single line search to determine the optimal amount of movement along the gradient direction.

3. Support Vector Machines

3.1. The dual problem

First of all, to make things clear, we state once again the dual problem for a L1/hinge loss soft-margin SVM, with $e = -1_v$:

$$\begin{aligned} &\min_{\alpha} \frac{1}{2}\alpha^{\mathsf{T}} Q \alpha - e^{\mathsf{T}}\alpha \\ &\text{s.t. } y^{\mathsf{T}}\alpha = 0 \\ &\qquad 0 \leq \alpha_i \leq C \ \forall \ i \in \{1, \dots, \ell\} \end{aligned}$$

$Q = \mathbb{K} \odot yy^{\mathsf{t}}$, where $\mathbb{K}$ is the Gram matrix of dot products and y is the vector of responses, with $y^i \in \{-1, 1\}$; C is the inverse regularization, the linear scale term on the hinge loss in the primal formulation.

3.2. Convex optimization with CVXOPT

The CVXOPT Python framework has a function for quadratic programming, `qp`, which is exactly what we need. The function expects a problem specification identical to the standard one, explained here. We thus formulate P, q, G, h, A, b w.r.t. our needs and our problem.

For what concerns the practical implementation, we have chosen to use a Gaussian/RBF kernel to build the Gram matrix, a kernel implicitly mapping the observations to the unit sphere of the ℓ_2 sequence space. C and γ are chosen through cross-validation. C determines the smoothness of the separation surface, allowing errors in order to shrink the norm of the decision function belonging to the Reproducing Kernel Hilbert Space stemming from the kernel $K(x, y) = \exp(-\gamma||x - y||^2)$. Equivalently, we are talking about the L_2 norm of a theoretical weight vector in the sequence space. γ is a delicate hyperparameter due to the fact that lower and lower values of γ implicitly map the feature vectors to (sequence) embeddings which keep getting closer and closer to identity/linear dependence (rendering impossible a meaningful separation); on the contrary, a too high γ implies getting close to orthogonality in the arrival space, inducing trivial linear separability (and overfitting).

After the fitting, we only need to save the information related to the support vectors, related training data rows, labels and dual variables. For what concerns the intercept, the trick involves the primal complementary slackness condition: for vectors with $0 < \alpha_i < C$ we are sure that ξ_i, the slack variable, is null (the point is not misclassified or inside the margin). Using the h index for variables satisfying this condition, the complementary slackness becomes: $y_h = \sum_{i=1}^{\ell} y^i \alpha_i K(x_i, x_h) + b$. From the above expression, we can easily find the intercept b of the decision function $f(x) = \text{sgn}\left(\sum_{i=1}^{\ell} y^i \alpha_i K(x_i, x) + b\right)$. Even though this holds for any vector satisfying the condition, we take the entire set of relevant vectors and we compute the average of the resulting intercept values.

3.3. Sequential Minimal Optimization

Taking into consideration the dual formulation from above, the quadratic problem in two variables become, with $\{i, s\}$ as the working set of indexes and α^k as the vector of dual variables in the previous iteration:

$$\min_{\alpha_i,\, \alpha_j} \frac{1}{2}\left(q_{ii}\alpha_i^2 + 2q_{ij}\alpha_i\alpha_j + q_{jj}\alpha_j^2\right) + \sum_{\substack{s=1 \\ s\neq i,j}}^{\ell}\left(q_{js}\alpha_s^k\alpha_j + q_{is}\alpha_s^k\alpha_i\right) - \alpha_i - \alpha_j$$

$$\text{s.t.} \quad y^i\alpha_i + y^j\alpha_j = y^i\alpha_i^k + y^j\alpha_j^k \wedge\ 0 \leq \alpha_i \leq C\ \wedge 0 \leq \alpha_j \leq C$$

Setting $d = y^i\alpha_i^k + y^j\alpha_j^k$ to simplify notation a bit, we solve the problem with replacement, getting back to a univariate quadratic problem, for now ignoring the inequality constraints: we will consider them later on. In the following, $Q_{i\cdot}$ is the ith row of the matrix Q:

$$\begin{aligned}
&\min_{\alpha_i} \frac{1}{2}\left(q_{ii}\alpha_i^2 + 2q_{ij}\alpha_i y^j(d - y^i\alpha_i) + q_{jj}(d - y^i\alpha_i)^2\right) + \\
&+ \sum_{\substack{s=1 \\ s\neq i,j}}^{\ell}\left(q_{js}\alpha_s^k y^j(d - y^i\alpha_i) + q_{is}\alpha_s^k\alpha_i\right) - \alpha_i - y^j(d - y^i\alpha_i) = \\
&= \min_{\alpha_i}\left(\frac{1}{2}(q_{ii} + q_{jj}) - q_{ij}y^j y^i\right) + \\
&+ \alpha_i\Big[q_{ij}y^j d - q_{jj}y^i d - y^j y^i\left(Q_{j\cdot}^{\mathsf{T}} - q_{jj}\alpha_j^k\right) + \left(Q_{i\cdot}^{\mathsf{T}}\alpha^k - q_{ii}\alpha_i^k\right) - \\
&- q_{ij}\left(\alpha_j - y^i y^j\alpha_i^k\right)\Big] \\
&\qquad\qquad \alpha_j = (d - y^i\alpha_i)y^j
\end{aligned}$$

This univariate objective function/parabola is always convex (can be proved), and therefore we can just search for the vertex and take the usual $-\frac{b}{2a}$, where b is the linear term and a is the quadratic one, to get the solution out. We have isolated both terms above.

Now, the problem is that we also have the inequality constraints; for α_i it is easy, since we need to consider just the $[0, C]$ interval. This is not enough though, since in general there can be values of α_i that lead to inadmissible values of α_j, which is constrained to the same closed interval. The situation is well described in Platt (1998), the original paper describing SMO.

Taking from Cristianini and Shawe-Taylor (2000) and Platt (1998), the admissible interval for α_i can be tidily formulated in the following way:

$$U = \begin{cases} \max(0, \alpha_i^k - \alpha_j^k) \text{ if } y^i \neq y^j \\ \max(0, \alpha_i^k + \alpha_j^k - C) \text{ if } y^i = y^j \end{cases}$$
$$V = \begin{cases} \min(C, C - \alpha_j^k + \alpha_i^k) \text{ if } y^i \neq y^j \\ \min(C, \alpha_i^k + \alpha_j^k) \text{ if } y^i = y^j \end{cases}$$
$$U \leq \alpha_i \leq V$$

If the vertex of the parabola (the objective function quadratic in α_i) is outside the admissible interval, we need to pick for the update the boundary point closest to the vertex (since the parabola is convex).

For what concerns the maximum violating pair selection, we have to build the sets $R(\alpha)$ and $J(\alpha)$, as defined in Piccialli and Sciandrone (2018). These two sets need to be initialised and updated at each iteration (we use NumPy boolean masks for this).

The variables we are picking each time are (α_i, α_j), where:

$$i = \underset{i \in R(\alpha)}{\arg\max} \left\{ -\frac{(\nabla f(\alpha))_i}{y_i} \right\}$$
$$j = \underset{j \in S(\alpha)}{\arg\min} \left\{ -\frac{(\nabla f(\alpha))_i}{y_j} \right\}$$

With our initialisation, $\alpha^0 = 0_v$, $\nabla f(\alpha^0) = -1_v$. The maxima and minima in the sets could be more than one, of course (think about the first iteration), but we simply take the first one of the argmax/argmin set according to the order of observations given by the user (it is not really relevant).

The gradient is updated at each iteration according to:

$$\nabla f(\alpha^{k+1}) = \nabla f(\alpha^{k+1}) + (\alpha_i^{k+1} - \alpha_i^k)Q_{i\cdot} + (\alpha_j^{k+1} - \alpha_j^k)Q_{j\cdot}$$

3.4. Three classes problem: One vs. All

The three classes problem is straightforward once we have the classes defined for the binary SVM problem. We simply build another class which when initialised it also initialises three different binary SVM objects. The response vector having three distinct values is then transformed with a `LabelBinarizer` object from `scikit-learn`, resulting in three different columns in a 1-vs-all fashion.

The three models are then trained each with one of these three columns as the vector with the response labels. Once this is done, prediction is done for the three models independently, with the resulting vectors then column stacked. Notice that here we want to have the decision function score, not simply the 0-1 classification. This allows us to tackle the following cases:

- The observation is classified as positive by more than one classifier. The chosen class is the one for which we have the highest confidence, namely the decision function score for the corresponding one-versus-all classifier. Geometrically speaking, the chosen class is the one whose corresponding decision boundary is the farthest from the observation vector.
- The observation is classified as negative by the whole set of classifiers. The chosen class is the one for which we have the lowest confidence; lowest confidence for a negative classification means highest decision function score. Geometrically speaking, we are searching for the decision boundary closest to the observation vector.

In other words, using `np.argmax` on the second axis of the scores array is more than enough to handle all the possible cases.