Artificial Intelligence - EDAP01
Assignment 2 - Machine Learning

Linear Regression using Gradient Descent
Classifiers using Perceptron Algorithm and Logistic Regression

Hicham Mohamad (hi8826mo-s)

February 17, 2021

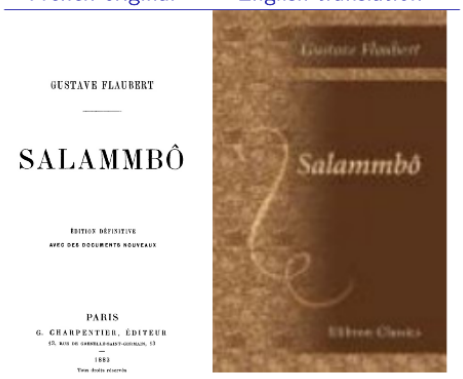
Introduction

In this assignment we are going to work on three different **Machine Learning** algorithms that involve two datasets corresponding to letter counts in the 15 chapters of the french and english versions of *Salammbô*, shown in Figure 1b. We first use the **gradient descent** method to compute regression lines. Then we need to program the **perceptron** algorithm and run it on the same dataset. Finally, we will improve the threshold function with the **logistic** curve for implementing the **logistic regression** from our perceptron program. In this way, we will try various configurations and study their influence on the learning speed and accuracy. At the end, we need to read the paper of Sebastian Ruder (2017) [7] "An overview of gradient descent optimization algorithms" and summarize the main characteristics of all the optimization algorithms the author describes.

User guide

All the objectives of this assignment are now solved but I only got problem in the **Keras task** of the last part for the **Logistic Classifier**. This can be a problem of version compatibility of Anaconda, which I updated using `pip` as suggested. The implementation is found in the provided jupyter notebook.

French original | English translation



(a)

Chapter	French		English	
	# characters	# A	# characters	# A
Chapter 1	36,961	2,503	35,680	2,217
Chapter 2	43,621	2,992	42,514	2,761
Chapter 3	15,694	1,042	15,162	990
Chapter 4	36,231	2,487	35,298	2,274
Chapter 5	29,945	2,014	29,800	1,865
Chapter 6	40,588	2,805	40,255	2,606
Chapter 7	75,255	5,062	74,532	4,805
Chapter 8	37,709	2,643	37,464	2,396
Chapter 9	30,899	2,126	31,030	1,993
Chapter 10	25,486	1,784	24,843	1,627
Chapter 11	37,497	2,641	36,172	2,375
Chapter 12	40,398	2,766	39,552	2,560
Chapter 13	74,105	5,047	72,545	4,597
Chapter 14	76,725	5,312	75,352	4,871
Chapter 15	18,317	1,215	18,031	1,119

(b)

Figure 1: a) The data set: Salammbô in french and english which it is a corpus, i.e. a collection or a body of texts. b) Letter counts from our dataset Salammbô

Learning

Machine learning algorithms can be classified along two main lines: **supervised** and **unsupervised** learning. the most common supervised learning tasks are *regression* (predicting values) and *classification* (predicting classes).

In supervised learning, given a **training set** of N example input-output pairs

$$(x_1, y_1), (x_2, y_2), \cdots (x_N, y_N)$$

where each y_i was generated by an unknown function $y = f(x)$, we need to discover a function h, called **hypothesis** that approximates the true function f. Learning is a search through the space of possible hypotheses for one that will perform well, *even on new examples beyond the training set.*

To measure the **accuracy** of a hypothesis we give it a **test set** of examples that are distinct from the training set. In this case, we say a hypothesis "*generalizes*" well if it correctly predicts the value of y for novel examples. In general, there is a **tradeoff** between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better.

1 Linear regression - Gradient Descent learning

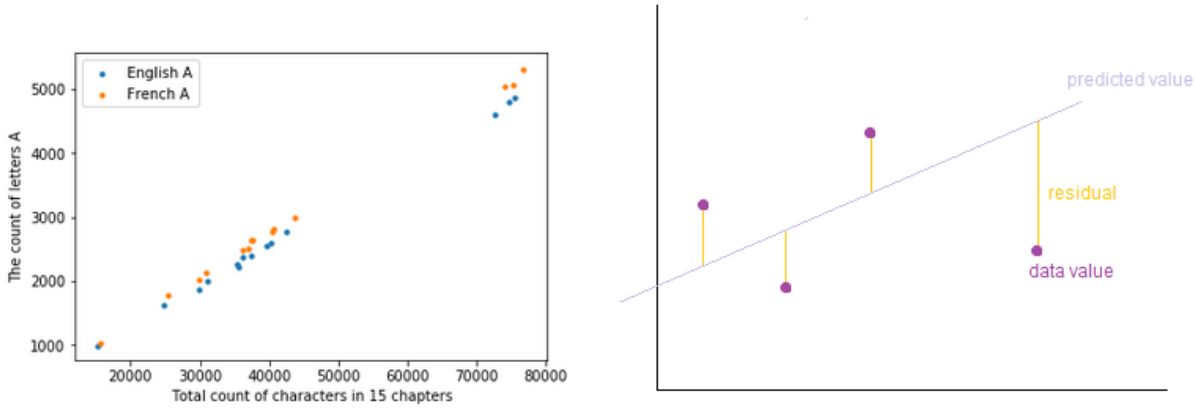
In this task we need to implement a linear regression program using gradient descent algorithm for computing the regression line linking counts of letters in a the text of Salammbô book to counts of letter A. We will implement the **stochastic** and **batch** versions of the algorithm. Then, we will apply it on two data sets corresponding to **letter counts** in the 15 chapters of the French and English versions of Salammbô, where the first column is the total count of characters and the second one, the count of A's.

Hypothesis of Linear Regression

The linear regression model can be represented by the following equation

$$Y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \theta^T x$$

where θ is the model's **parameter vector** including the bias term θ_0 and x is the **feature vector** with $x_0 = 1$. Training of the model here means to find the parameters so that the model best fits the data. How do we determine the best fit line? The line for which the error between the predicted values and the observed values is minimum is called the **best fit line** or the regression line. These errors are also called as **residuals**. The residuals can be visualized by the vertical lines from the observed data value to the regression line, as shown in Figue 2b.



(a) (b)

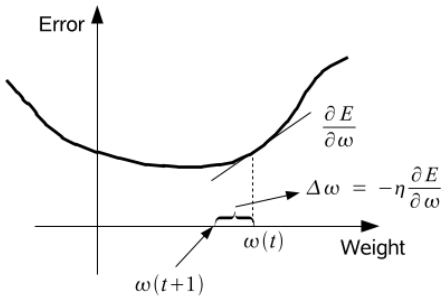
*Figure 2: a) The plot for the data set from Salammbô. b) The line for which the the error between the predicted values and the observed values is minimum is called the **best fit line** or the regression line..*

Gradient Descent learning

The gradient descent is a numerical method to find the minimum of $y = f(x_0, x_1, x_2, \cdots, x_n)$ when there is no analytic solution. The minimization can be summarized with the following **update rule** for a given weight w_k, as shown in Figure 3a

$$\Delta w_k = -\eta \frac{\partial E}{\partial w_k} \tag{1}$$

where E is the error based on the difference between the y and $\hat{y}$. Δw_k in Equation 1 is called **steepest descent**.



(a)

```
1: function Regression(DS)
2:     w ← w^0
3:     while ||∇Loss(w)|| > ε do
4:         w ← w − α∇Loss(w)
5:     return w
```

(b)

Figure 3: a) Illustration of the gradient descent method. b) Gradient descent learning algorithm.

Univariate regression In the case of a univariate linear function (a straight line) with input x and output y, we have the form

$$y = w_1 x + w_0 \tag{2}$$

where w_0 and w_1 are real-valued coefficients to be learned. Here we define $\mathbf{w}$ to be the **weights vector** $[w_0, w_1]$ and $\hat{y} = h(x) = w_1 x + w_0$. To fit a line to the data, all we have to do is finding the values of the weights $[w_0, w_1]$ that minimize the **loss function** L. It is traditional to use the **squared loss function**, L_2, summed over all the training examples:

$$Loss(w) = \sum_{j=1}^{N} L_2(y_j - h(x_j))^2 = \sum_{j=1}^{N} L_2(y_j - (w_1 x_j + w_0))^2 \tag{3}$$

We would like to find

$$\mathbf{w^*} = \arg\min_w Loss(w) \tag{4}$$

The sum in Equation 3 is minimized when its partial derivatives with respect to w_0 and w_1 are zero. We choose any starting point w^0 in weight space, here, a point in the (w_0, w_1) plane, and then move to a neighboring point that is downhill, repeating until we converge on the minimum possible loss:

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w}) \tag{5}$$

where α is usually called the **learning rate**. It can be a fixed constant or it can decay over time as the learning process proceeds. We work out the partial derivatives, i.e. the slopes, in the simplified case of only one training example, (x, y)

$$\begin{aligned} \nabla Loss(\mathbf{w}) = \frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h(x))^2 \\ &= 2(y - h(x)) \times \frac{\partial}{\partial w_i}(y - h(x)) \\ &= 2(y - h(x)) \times \frac{\partial}{\partial w_i}(y - (w_1 x + w_0)) \end{aligned} \tag{6}$$

Applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} Loss(\mathbf{w}) = -2(y - h(x)); \quad \frac{\partial}{\partial w_1} Loss(\mathbf{w}) = -2(y - h(x)) \times x \tag{7}$$

then, plugging this back into the update equation number 4, shown in the method in Figure 3b and folding the 2 into the unspecified learning rate α, we get the following *learning rule* for the weights:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h(x_j)); \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h(x_j)) \times x_j$$

These updates constitute the **batch gradient descent** learning rule for univariate linear regression. *Convergence to the unique global is guaranteed as long as we pick α small enough but may be very slow: this because we have to cycle through all the training data for every step.*

There is another possibility, called **stochastic gradient descent**, where we carry out the updates using one single training point at a time:

$$w_i \leftarrow w_i + \alpha \cdot x_i^j \cdot (y^j - (w_0 x_0^j + w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j))$$

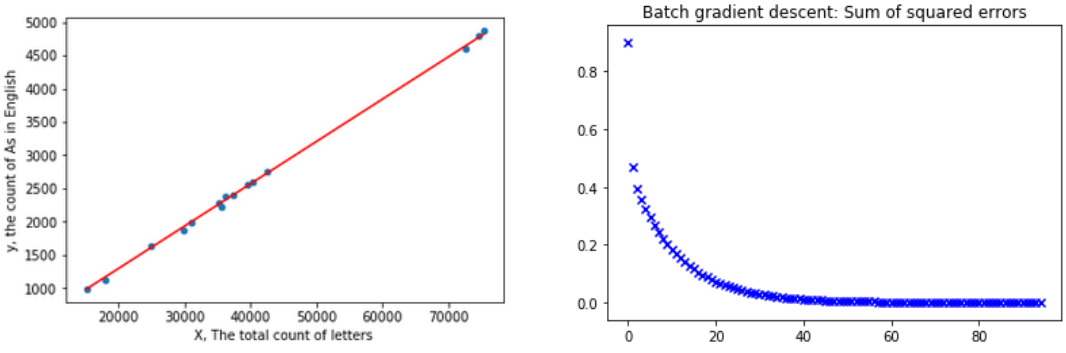
Thus, in this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. Whereas batch gradient descent has to scan through the entire training set before taking a single step — a costly operation if n is large — stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at.

Convergence to the minimum Often, stochastic gradient descent gets the weights w "close" to the minimum much faster than batch gradient descent. Note however that it may "never converge" to the minimum, and the parameters w will keep oscillating around the minimum of the cost function $J(w)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum. For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

Regression line implementation

Before starting the computation, we scale the data so that they fit in the range [0, 1] on the x and y axes. After implementing the gradient descent algorithm in Python and running it on the English data set we get the **regression line** plot visualized in Figure 4a and the errors in Figure 4b. After training the model we find the following **parameters/weights** such that the model best fits the data

```
1  ===Batch descent===
2
3  Epoch 236
4  the normalized Weights [[-7.31744724e-04]
5   [ 9.94697306e-01]]
6
7  Summed squared error, SSE = [[0.00086294]]
8
9  Restored weights [[-3.56432855]
10  [ 0.06430049]]
```



(a) (b)

Figure 4: a) Visualization of the points and the regression line after running the implemented Gradient Descent program, with the batch version, on the english dataset corresponding to letter counts in the 15 chapters of Salammbô. b) Plot of the sum of squared errors.

2 Linear classifier - Perceptron

In this task, we use the same dataset as before, but this time to classify a chapter as French or English. Given a pair of numbers corresponding the letter count and count of A, we will predict the language.

Now we put the two datasets together, English and French such that the dataset consists of arrays X and y. Array X contains the counts of letters, counts of "A" and a column of ones for the intercept; Whereas array y contains the classes, where 0 is for English and 1 for French.

In this way, we need to implement **linear classifiers** using the Perceptron algorithm and run it on Salammbô dataset, where we use the number of misclassified examples as a **stop criterion**. At the end, we evaluate the perceptron using the **leave-one-out cross validation** method.

The perceptron Learning algorithm (Digression)

Actually the perceptron learning algorithm just moves the decision plane ($\mathbf{w}$) in such a way that all inputs x_n n are correctly classified. There is a simple weight update rule that converges to a solution - that is, a **linear separator** that classifies the data perfectly - provided the data are **linearly separable**.

Here we can write the classification hypothesis as

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

which corresponds to the **threshold function**. For a single example (x, y), we have a simple weight update rule that converges to a solution

$$w_i \leftarrow w_i + \alpha(y - h(\mathbf{x})) \cdot x_i \tag{8}$$

which is essentially identical to the update rule for linear regression we have mentioned above; But this is not the same algorithm, because $h(\mathbf{x})$ is now defined as a non-linear function of $\mathbf{w} \cdot \mathbf{x}$. This rule is called the **Perceptron learning rule**. Typically this learning rule is applied on one example at a time, choosing examples at random as in stochastic gradient descent.

Linear classifier implementation using the perceptron

In this task, we implement the perceptron learning algorithm in similar way to the implementation in the previous section, i.e. the gradient descent method, but this time using the weight update in Equation 8.

Then we evaluate the stochastic perceptron classifier using the **leave-one-out cross validation** method. In this way, we train and run 30 models. In each train/run session, we train on 29 samples and evaluate on the remaining sample.

Plotting the decision boundary

As there are two features in our dataset, the linear equation can be represented by

$$h(x) = w_0 + w_1 x_1 + w_2 x_2$$

The decision boundary can be found by setting the weighted sum of inputs to 0. Equating h(x) to 0 gives us,

$$x_2 = -\frac{w_0 + w_1 x_1}{w_2}$$

Results of the Perceptron classification

After running the perceptron program with the stochastic version we get the following results:

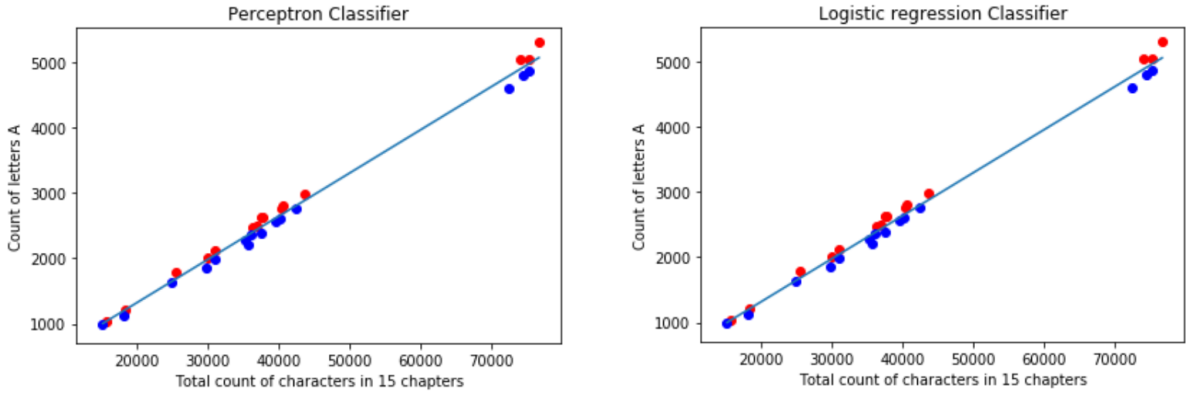
```
1 ===Stochastic Descent Perceptron ===
2
3 Misclassified observations:  0
4 Epochs:  30
```

```
Weights:  [[0.0, -3.805630498533728, 3.9828689759036155]]

Restored weights [array([0.]), array([-0.26348008]), array([3.98286898])]
Weights with y set to 1 [array([0.]), array([-0.06615334]), array([1.])]
```

After running the implemented **leaveOneOut cross-validation** function we get the following output:

```
Epochs:  61
Fold  29 on 30:
Correct classifications:  29
Cross-validation accuracy (stochastic): 0.9666666666666667
```



(a) (b)

Figure 5: a) Classification of the dataset using the stochastic perceptron. b) Classification using logistic regression.

3 Logistic Regression

Here we need to implement a **linear classifier** using the **logistic regression** in the perceptron learning method. This is done by implementing the stochastic and/or the batch version of the algorithm. Then we should run the resulting program on our dataset. At the end we evaluate the logistic regression using the **leave-one-out cross validation** method.

Linear classification with logistic regression

For the simple perceptron task, we were dealing with a **hard threshold activation function** for the classification problem, which is not differentiable. That is why it is more common to use the **logistic activation function**. The logistic function, also called Sigmoid, defined by

$$logistic(z) = \frac{1}{1 + e^{-z}} \tag{9}$$

has more convenient mathematical properties. By using this logistic function, we replace and improve the **threshold function** used in the Perceptron learning method. In this way, we get the process of *fitting the weights to minimize the loss on a dataset*, i.e. **logistic regression**. With this function, we now have

$$h(\mathbf{x}) = logistic(\mathbf{w}.\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}.\mathbf{x}}} \tag{10}$$

Logistic regression attempts to model the probability of an **observation** x to belong to a **class**. As we can see in Figure 6, this forms a **soft boundary** in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

Using the chain rule for the derivation, we get a useful property of the derivative of the sigmoid function which satisfies

$$g\prime(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h(\mathbf{x})(1 - h(\mathbf{x})) \tag{11}$$

so the weight update for minimizing the loss is

$$w_i \leftarrow w_i + \alpha(y - h(\mathbf{x})) \cdot h(\mathbf{x})(1 - h(\mathbf{x})) \cdot x_i \tag{12}$$

Using the property of the derivative in Equation 11 gives us the **stochastic gradient ascent** rule

$$w_i \leftarrow w_i + \alpha(y^{(j)} - h(\mathbf{x}^{(j)})) \cdot x_i^{(j)} \tag{13}$$

In general form we can write

$$w_i \leftarrow w_i + \alpha \cdot \delta \cdot x_i^{(j)} \tag{14}$$

Where in this case

$$\delta = (y - h(\mathbf{x}))$$

In real-world applications, logistic regression has become one of the most popular classification techniques for problems in medicine, marketing and survey analysis, credit scoring, public health, and other applications.

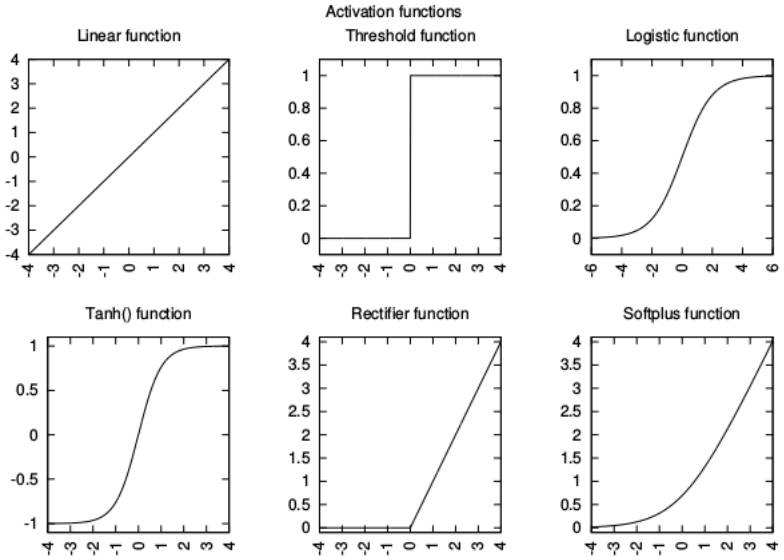


Figure 6: The different activation functions

Linear classifier implementation using logistic regression

In this task, we implement the perceptron learning algorithm in similar way to the implementation in the previous section, but this time using the logistic activation function (sigmoid), $h(\mathbf{x})$ in Equation 10, in the weight update rule.

Results of the logistic classification

We get the following parameters results using the implemented algorithm in the stochastic version:

```
1  === Stochastic Descent Logistic ===
2
3  Epoch 596
4  Logistic Weights:
5   [[2.073225839414742, -3769.1505527855234, 3952.204257002531]]
6
7  Logistic Restored Weights
8   [array([2.07322584]), array([-0.04912546]), array([0.74401436])]
9  Weights with y set to 1
10  [array([2.7865401]), array([-0.06602756]), array([1.])]
```

We evaluate the logistic regression using **leave-one-out cross validation** method as with the perceptron, and obtain the following results:

```
Epochs 528
Fold  29 on 30:
Correct classifications:  29
Cross-validation accuracy (batch): 0.9666666666666667
```

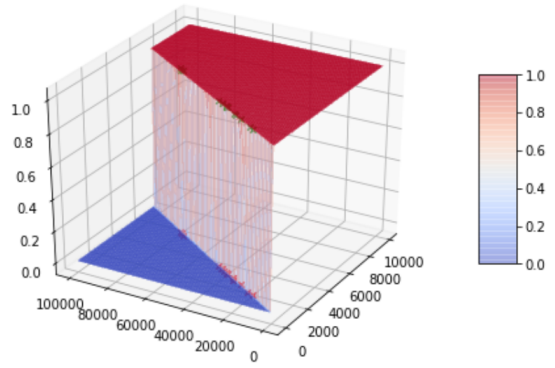


Figure 7: Visualization of the logistic surface.

4 Conclusion

As as we have described above, it is preferable and more common to run stochastic gradient descent. In the algorithm, we have also tried to assume a fixed learning rate $\alpha = 100$ and by slowly letting the learning rate α decrease to zero as the algorithm runs, by multiplying by 0.99. In this way, it is possible to ensure that the parameters will converge to the **global minimum** rather than merely oscillate around the minimum.

Shuffling and normalization used in our assignment actually helped to achieve good performance with accuracy 96%. In general, by using a suitable **regularization term/optimizer**, as mentioned in the paper below, may actually lead also to improved performance and better convergence, like `RMSprop` or `NAG` with the **cross-entropy** error function in the case of a logistic regression classifier. Because tuning/choosing the learning rate α can also improve the results but it is a challenging task.

5 Reading

In this task, we need to read the article of Ruder (2017) *An overview of gradient descent optimization algorithms* and summarize the main characteristics of all the **optimization algorithms** the author describes.

The article addresses the gradient descent optimization algorithms which are often used as black-box optimizers, as practical explanations of their strength and weakness are hard to come by. Thus, the authors present and investigate the following principal points:

Three variants of gradient descent In this section, they describe how three variants of gradient descent differ in how much data is used to compute the gradient of the objective function. Here we have trade off between the accuracy of the parameter update and the time it takes to perform an update depending on the amount of the data.

- **Batch gradient descent**: computes the gradient of the cost function w.r.t. to the parameters for the entire training dataset. It can be very slow and is intractable for datasets that do not fit in memory.
- **Stochastic gradient descent**: This in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$. It is usually much faster and can also be used to learn online.
- **Mini-batch gradient descent**: Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of training examples.

However, Vanilla mini-batch gradient descent does not guarantee good convergence, but offers a few **challenges**: First, it is that choosing a proper learning rate can be difficult. Additionally, the same learning rate is applied to all parameter updates. Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal **local minima**.

Algorithms for optimizing SGD The authors outline some algorithms that are widely used by the Deep Learning community to deal with the aforementioned challenges. Here we summarize the main characteristics of these algorithms:

1. **Momentum**: Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the update vector of the past time step to the current update vector.

2. **Nesterov accelerated gradient** NAG: Nesterov accelerated gradient (NAG) is a way to give momentum term some **prescience**. NAG first makes a big jump in the direction of the previous accumulated gradient, measures the gradient and then makes a correction.

3. **Adagrad**: It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. Adagrad's main weakness is its accumulation of the squared gradients.

4. **Adadelta**: Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size ω.

5. **RMSprop**: RMSprop divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

6. **Adam**: Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{15}$$

   m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively.

7. **AdaMax**: The v_t factor in the Adam update rule scales the gradient inversely proportionally to the l_2 norm of the past gradients (via the v_{t-1} term) and current gradient $|g_t|^2$. With AdaMax we can generalize this update to the l_p norm.

8. **Nadam**: Nadam (Nesterov-accelerated Adaptive Moment Estimation) combines Adam and NAG. In order to incorporate NAG into Adam, we need to modify its momentum term m_t.

Additional strategies for optimizing SGD Here they introduce additional strategies that can be used alongside any of the previously mentioned algorithms to further improve the performance of SGD.

- **Shuffling and curriculum learning**: it is often a good idea to shuffle the training data after every epoch, in order to avoid providing the training examples in a meaningful order to our model, as this may bias the optimization algorithm. However, sometimes supplying the training examples in a **meaningful order** may actually lead to improved performance and better convergence, i.e. Curriculum Learning method.

- **Batch normalization**: To facilitate learning, we typically normalize the initial values of our parameters by initializing them with **zero mean** and **unit variance**.

- **Early stopping**: We should always monitor error on a validation set during training and stop if the validation error does not improve enough.

- **Gradient noise**: They show that adding noise makes networks more robust to poor initialization and helps training particularly deep and complex networks.

6 References

References

[1] Pierre Nugues, Github. Classification with the perceptron and logistic regression https://github.com/pnugues/edap01/tree/master/labs

[2] Pierre Nugues, Github. Linear regression. Available: https://github.com/pnugues/ilppp/tree/master/programs/ch04/python

[3] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 3/e, Prentice Hall Series in Artificial Intelligence, 2010. ISBN-10: 0132071487 or 1292153962. Available: https:\aima.cs.berkeley.edu.

[4] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning.* MIT Press, 2016, ISBN: 9780262035613. Available: https://www.deeplearningbook.org/.

[5] Aurélien Géron: Hands-On Machine Learning with Scikit-Learn and TensorFlow, Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, 2017, ISBN: 9781491962299.

[6] Andrew Ng, Stanford course CS229: Machine Learning. Available: cs229.stanford.edu/notes/cs229-notes1.pdf

[7] Sebastian Ruder. An overview of gradient descent optimization algorithms. Insight Centre for Data Analytics NUI Galway Aylien Ltd., Dublin, 2017.

[8] Cauchy, Original article on gradient descent in french: https://gallica.bnf.fr/ark:/12148/bpt6k2982c/f540.item