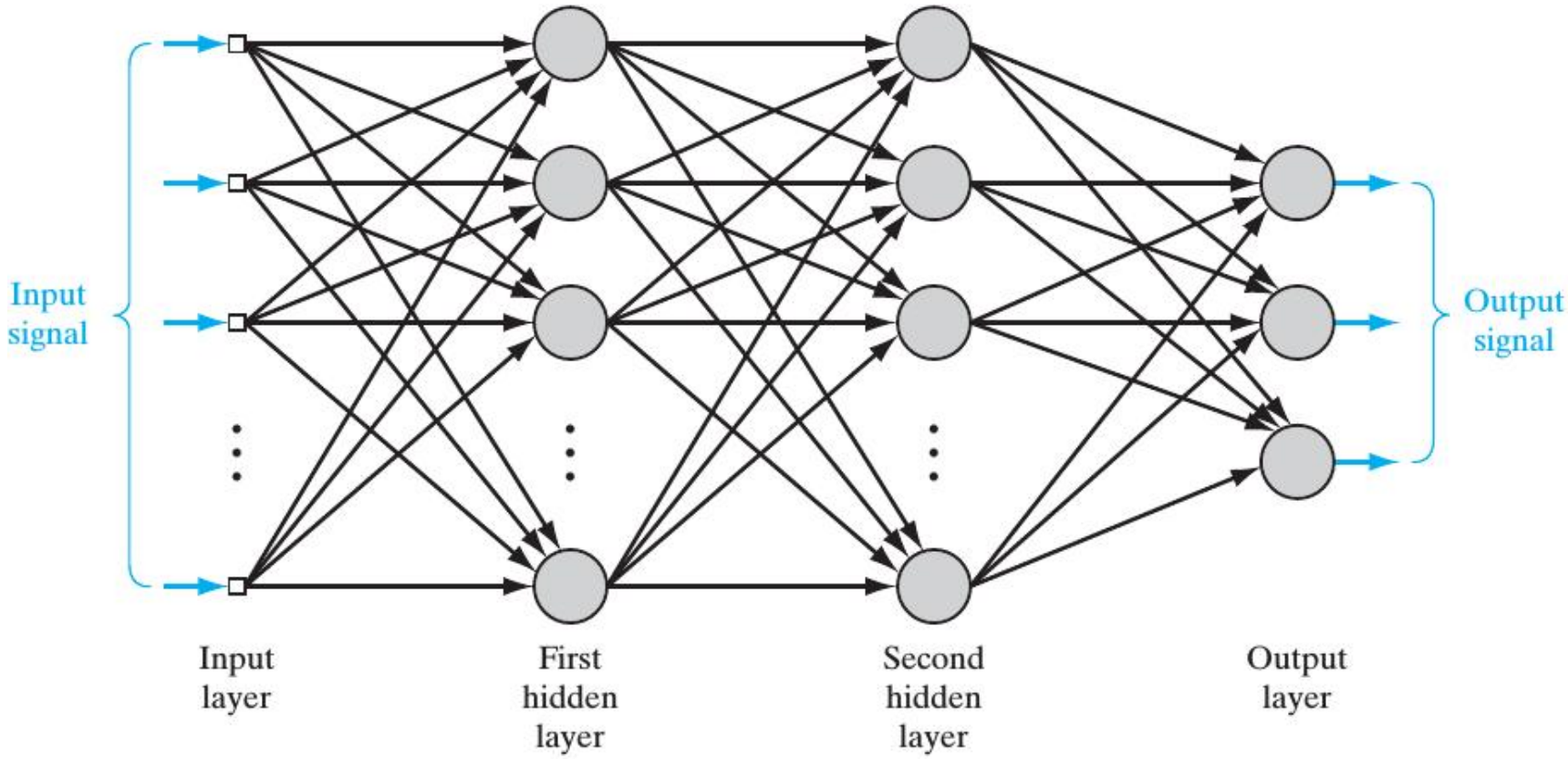
Optimization Techniques for Big Data Analysis

Chapter 4. First-Order Methods Applied to Neural Networks

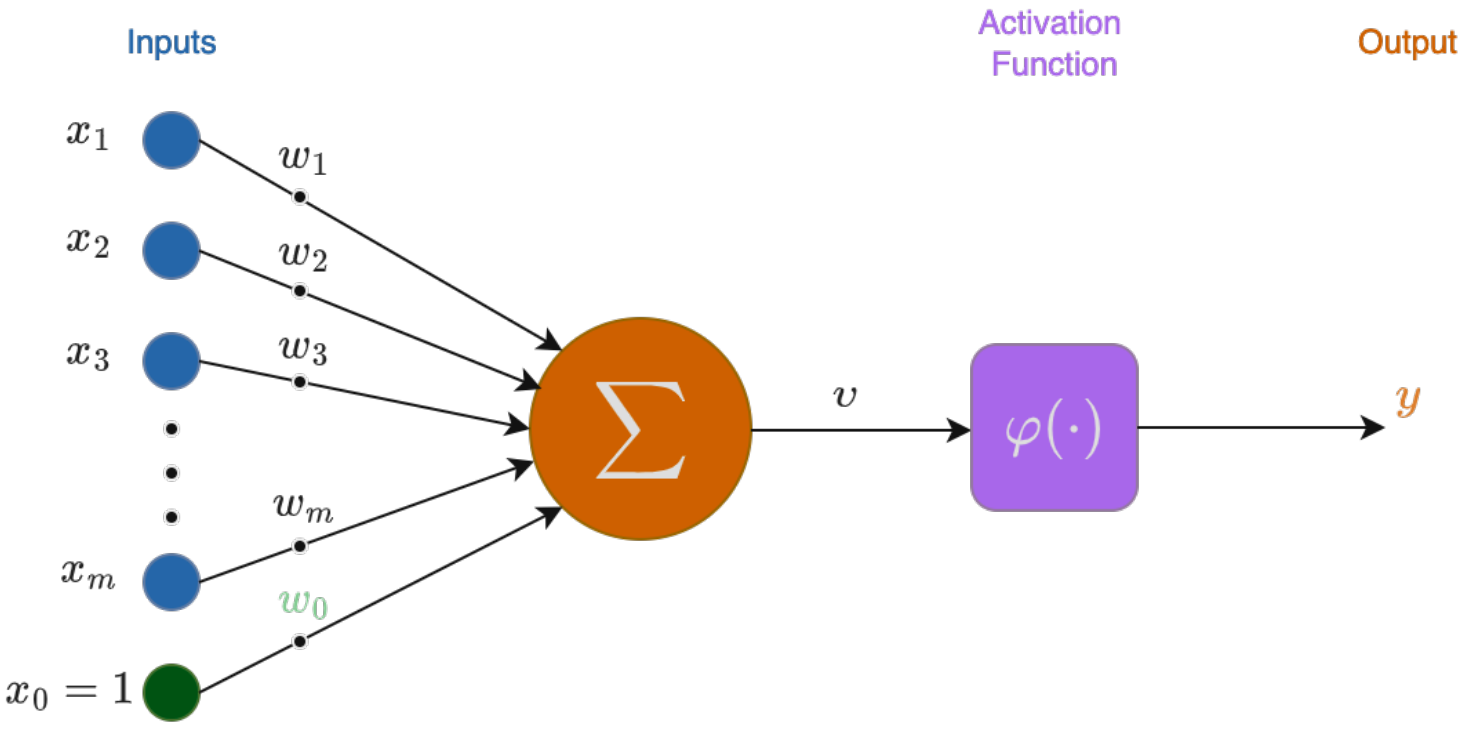
Master of Science in Signal Theory and Communications
Dpto. de Señales, Sistemas y Radiocomunicaciones
E.T.S. Ingenieros de Telecomunicación
Universidad Politécnica de Madrid

2024

Scheme of a DNN



Basic unit: the neuron



The basic operation of a single neuron corresponds to:

$$y = \varphi\left(\sum_{j=1}^{m} x_j w_j + w_0\right) = \varphi\left(\mathbf{w}^T \mathbf{x}\right)$$

DNN output function

For a neural network of with one hidden layer, the p-th output for one input $\mathbf{x}_i = [x_{1,i}, x_{2,i}, \cdots, x_{j,i}, \cdots, x_{d,i}]^T$ is given by,

$$o_p(\mathbf{x}_i) = \varphi^{(2)} \left(\sum_{l=1}^{m_1} w_{pl}^{(2)} \varphi^{(1)} \left(\sum_{j=1}^{d} w_{lj}^{(1)} x_{j,i} + w_{l0}^{(1)} \right) + w_{p0}^{(2)} \right)$$

DNN output function

For a neural network of with one hidden layer, the p-th output for one input $\mathbf{x}_i = [x_{1,i}, x_{2,i}, \cdots, x_{j,i}, \cdots, x_{d,i}]^T$ is given by,

$$o_p(\mathbf{x}_i) = \varphi^{(2)} \left(\sum_{l=1}^{m_1} w_{pl}^{(2)} \varphi^{(1)} \left(\sum_{j=1}^{d} w_{lj}^{(1)} x_{j,i} + w_{l0}^{(1)} \right) + w_{p0}^{(2)} \right)$$

To estimate it, the forward pass does the following:

$$\text{Layer 1} \rightarrow v_{l,i}^{(1)} = \sum_{j=0}^{d} w_{lj}^{(1)} x_{j,i} \quad z_{l,i}^{(1)} = \varphi^{(1)} \left(v_{l,i}^{(1)} \right)$$

$$\text{Layer 2} \rightarrow v_{p,i}^{(2)} = \sum_{l=0}^{m_1} w_{pl}^{(2)} z_{l,i}^{(1)} \quad z_{p,i}^{(2)} = \varphi^{(2)} \left(v_{p,i}^{(2)} \right)$$

Cost function

Depending on the task, a specific loss function $\mathcal{L}$ is required.
For the sake of simplicity, we are going to use MSE without loss of generality.

Cost function

Depending on the task, a specific loss function $\mathcal{L}$ is required. For the sake of simplicity, we are going to use MSE without loss of generality.

$$\mathcal{L}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} \ell_{\mathbf{W}}(\mathbf{x}_i, y_i)$$

Cost function

Depending on the task, a specific loss function $\mathcal{L}$ is required. For the sake of simplicity, we are going to use MSE without loss of generality.

$$\mathcal{L}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\mathbf{W}}(\mathbf{x}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} e_i^2$$

Cost function

Depending on the task, a specific loss function $\mathcal{L}$ is required. For the sake of simplicity, we are going to use MSE without loss of generality.

$$\mathcal{L}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\mathbf{W}}(\mathbf{x}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} e_i^2 = \frac{1}{n}\sum_{i=1}^{n} (o(\mathbf{x}_i) - y_i)^2$$

Cost function

Depending on the task, a specific loss function $\mathcal{L}$ is required. For the sake of simplicity, we are going to use MSE without loss of generality.

$$\mathcal{L}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\mathbf{W}}(\mathbf{x}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} e_i^2 = \frac{1}{n}\sum_{i=1}^{n} (o(\mathbf{x}_i) - y_i)^2$$

Note that the DNN could have multiple outputs, in that case be $\mathbf{y}_i = [y_{1,i}, \cdots, y_{p,i}, \cdots, y_{q,i}]$, the q-dimensional vector with the labels for the i-th sample.

$$\mathcal{L}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{p=1}^{q} e_{p,i}^2 = \frac{1}{n}\sum_{i=1}^{n}\sum_{p=1}^{q} (o_{p,i} - y_{p,i})^2$$

Weights update rule

During one iteration of the learning algorithm:

$$w^{(t)}[k+1] = w^{(t)}[k] - \eta \frac{\partial \mathcal{L}}{\partial w^{(t)}}$$

Weights update rule

During one iteration of the learning algorithm:

$$w^{(t)}[k+1] = w^{(t)}[k] - \eta \frac{\partial \mathcal{L}}{\partial w^{(t)}}$$

Considering the former example, if $t = 2$ the updating rule looks like:

$$w_{pl}^{(2)}[k+1] = w_{pl}^{(2)}[k] - \frac{\eta}{n} \sum_{i=1}^{n} \left(\frac{\partial \ell_{p,i}}{\partial e_{p,i}} \frac{\partial e_{p,i}}{\partial \varphi_{p,i}^{(2)}} \frac{\partial \varphi_{p,i}^{(2)}}{\partial v_{p,i}^{(2)}} \frac{\partial v_{p,i}^{(2)}}{w_{pl}^{(2)}[k]} \right)$$

Weights update rule

During one iteration of the learning algorithm:

$$w^{(t)}[k+1] = w^{(t)}[k] - \eta \frac{\partial \mathcal{L}}{\partial w^{(t)}}$$

Considering the former example, if $t = 2$ the updating rule looks like:

$$w_{pl}^{(2)}[k+1] = w_{pl}^{(2)}[k] - \frac{\eta}{n} \sum_{i=1}^{n} \left(\underbrace{\frac{\partial \ell_{p,i}}{\partial e_{p,i}} \frac{\partial e_{p,i}}{\partial \varphi_{p,i}^{(2)}} \frac{\partial \varphi_{p,i}^{(2)}}{\partial v_{p,i}^{(2)}}}_{\delta_{p,i}} \frac{\partial v_{p,i}^{(2)}}{\partial w_{pl}^{(2)}[k]} \right)$$

Weights update rule

During one iteration of the learning algorithm:

$$w^{(t)}[k+1] = w^{(t)}[k] - \eta \frac{\partial \mathcal{L}}{\partial w^{(t)}}$$

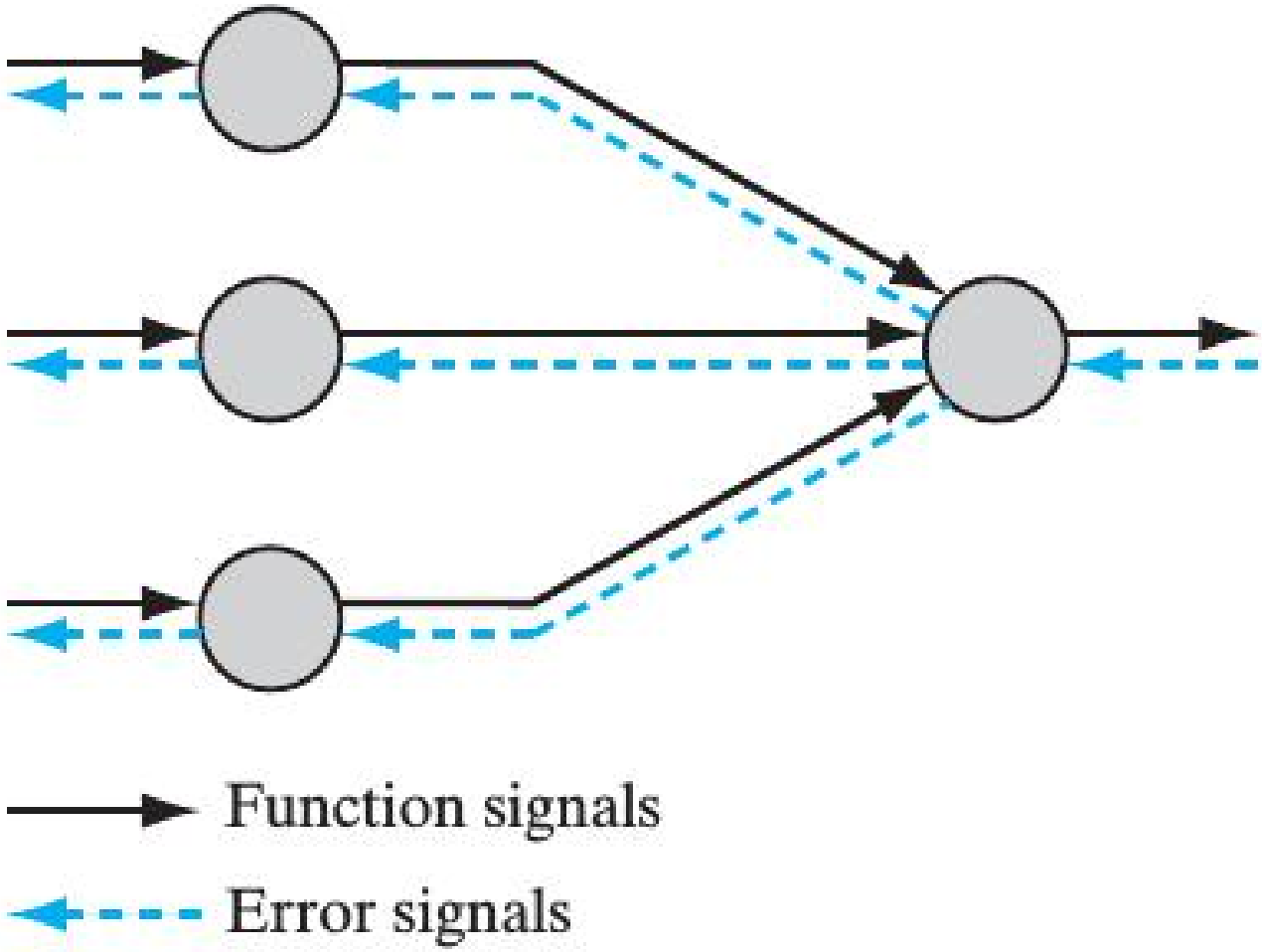
Considering the former example, if $t = 2$ the updating rule looks like:

$$w_{pl}^{(2)}[k+1] = w_{pl}^{(2)}[k] - \frac{\eta}{n} \sum_{i=1}^{n} \left(\underbrace{\frac{\partial \ell_{p,i}}{\partial e_{p,i}} \frac{\partial e_{p,i}}{\partial \varphi_{p,i}^{(2)}} \frac{\partial \varphi_{p,i}^{(2)}}{\partial \upsilon_{p,i}^{(2)}}}_{\delta_{p,i}} \frac{\partial \upsilon_{p,i}^{(2)}}{\partial w_{pl}^{(2)}[k]} \right)$$

The updating rule for $t = 1$ (backward pass) looks like this:

$$w_{lj}^{(1)}[k+1] = w_{lj}^{(1)}[k+1] - \frac{\eta}{n} \sum_{i=1}^{n} \sum_{p=1}^{q} \left(\delta_{p,i} \frac{\partial \upsilon_{p,i}^{(2)}}{z_{l,i}^{(1)}} \frac{\partial z_{l,i}^{(1)}}{\partial \varphi_{l,i}^{(1)}} \frac{\partial \varphi_{l,i}^{(1)}}{\partial \upsilon_{l,i}^{(1)}} \frac{\partial \upsilon_{l,i}^{(1)}}{\partial w_{lj}^{(1)}[k]} \right)$$

Signal flows



Gradient calculation

All factors in the updating rules can be calculated easily:

$$\frac{\partial \ell_{p,i}}{\partial e_{p,i}} \rightarrow 2(o_{p,i} - y_{p,i})$$

$$\frac{\partial e_{p,i}}{\partial \varphi_{p,i}^{(2)}} \rightarrow 1$$

$$\frac{\partial \varphi_{p,i}^{(2)}}{\partial \upsilon_{p,i}^{(2)}} \rightarrow \varphi^{'(2)}\left(\upsilon_{p,i}^{(2)}\right)$$

$$\frac{\partial \upsilon_{p,i}^{(2)}}{\partial w_{pl}^{(2)}} \rightarrow z_{l,i}^{(1)}$$

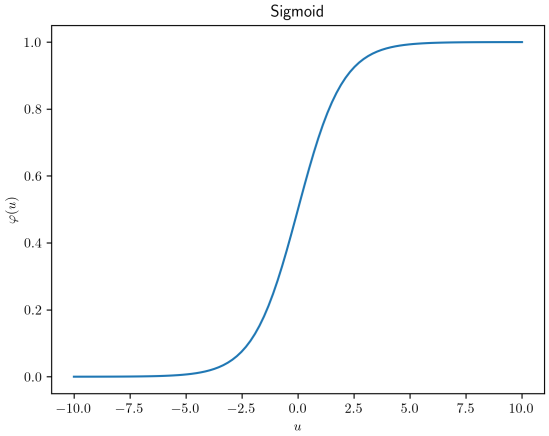
$$\frac{\partial \upsilon_{p,i}^{(2)}}{\partial z_{l,i}^{(1)}} \rightarrow w_{p,l}^{(2)}$$

$$\frac{\partial z_{l,i}^{(1)}}{\partial \varphi_{l,i}^{(1)}} \rightarrow 1$$

$$\frac{\partial \varphi_{l,i}^{(1)}}{\partial \upsilon_{l,i}^{(1)}} \rightarrow \varphi^{'(1)}(\upsilon_{l,i}^{(1)})$$

$$\frac{\partial \upsilon_{l,i}^{(1)}}{\partial w_{lj}^{(1)}} \rightarrow x_{j,i}$$

Activation Functions

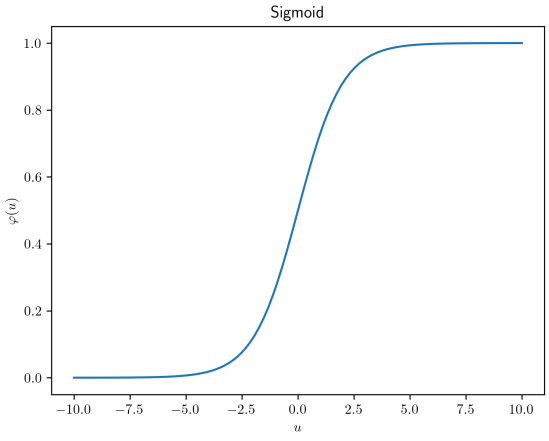


Sigmoid

$$\varphi(u) = \frac{\exp(u)}{1+\exp(u)} = \frac{1}{1+\exp(-u)}$$

$$\varphi^{'}(u) = \varphi(u)(1-\varphi(u))\partial u$$

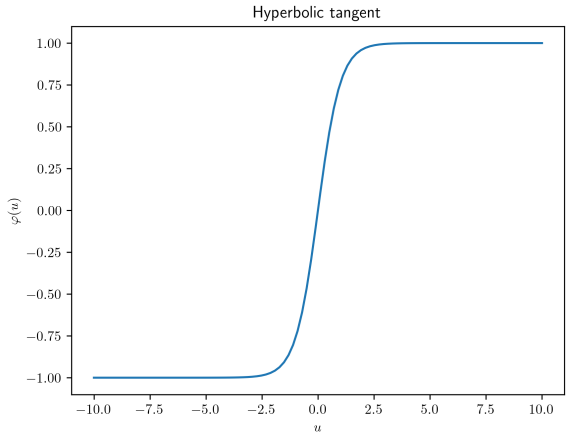
Activation Functions



Sigmoid

$$\varphi(u) = \frac{\exp(u)}{1+\exp(u)} = \frac{1}{1+\exp(-u)}$$

$$\varphi^{'}(u) = \varphi(u)(1-\varphi(u))\partial u$$

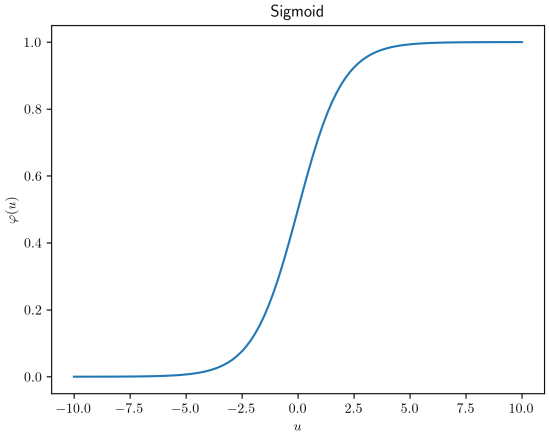


Hyperbolic tangent

$$\varphi(u) = \frac{\exp(u)-\exp(-u)}{\exp(u)+\exp(-u)}$$

$$\varphi^{'}(u) = (1-(\varphi(u))^2)\partial u$$

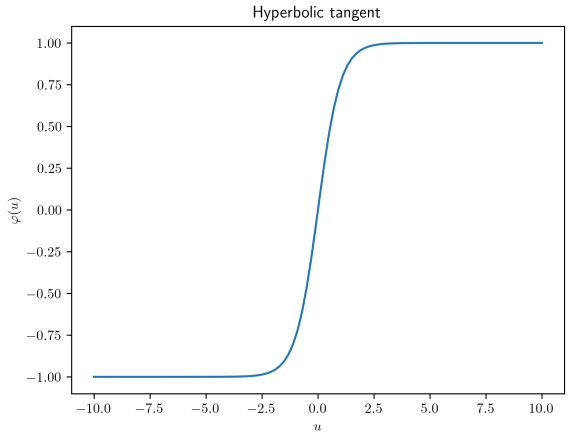
Activation Functions



Sigmoid

$$\varphi(u) = \frac{\exp(u)}{1+\exp(u)} = \frac{1}{1+\exp(-u)}$$

$$\varphi^{'}(u) = \varphi(u)(1-\varphi(u))\partial u$$



Hyperbolic tangent

$$\varphi(u) = \frac{\exp(u)-\exp(-u)}{\exp(u)+\exp(-u)}$$

$$\varphi^{'}(u) = (1-(\varphi(u))^2)\partial u$$

Softmax

$$\varphi(u_p) = \frac{\exp(u_p)}{\sum_{l=1}^{C}\exp(u_l)}, \quad \frac{\partial\varphi(u_p)}{\partial u_l} = \varphi(u_p)(\iota_{pl}-\varphi(u_l)), \quad \iota_{pl} = \begin{cases} 1 & \text{if} \quad p = l \\ 0 & \text{if} \quad p \neq l \end{cases}$$

Remarks

It is worth highlighting the following:

- The objective function is non-convex.
  - It's most likely to have saddle points than local minima in high-dimensional spaces.
  - There exists a negative curvature for every saddle point.

Remarks

It is worth highlighting the following:

- The objective function is non-convex.
  - It's most likely to have saddle points than local minima in high-dimensional spaces.
  - There exists a negative curvature for every saddle point.
- The theoretical analysis of the NN is hard to undertake.

Remarks

It is worth highlighting the following:

- The objective function is non-convex.
  - It's most likely to have saddle points than local minima in high-dimensional spaces.
  - There exists a negative curvature for every saddle point.
- The theoretical analysis of the NN is hard to undertake.
- The computational complexity (in terms of time and memory) depends on the number of samples n, the dimensionality of the input space d, and the number of hidden layers.

Remarks

It is worth highlighting the following:

- The objective function is non-convex.
  - It's most likely to have saddle points than local minima in high-dimensional spaces.
  - There exists a negative curvature for every saddle point.
- The theoretical analysis of the NN is hard to undertake.
- The computational complexity (in terms of time and memory) depends on the number of samples n, the dimensionality of the input space d, and the number of hidden layers.

Think about what the algorithm needs to keep in memory for a whole epoch (forward and backward passes)

Stochastic Gradient Methods

$$f(\mathbf{w}) = \mathbb{E}[\ell_{\mathbf{w}}(\mathbf{x})] + r(\mathbf{w})$$

For the standard gradient descent, during iteration k, we need access to the actual gradient vector $\nabla_{\mathbf{w}} f(\mathbf{w}_k)$.

Stochastic Gradient Methods

$$f(\mathbf{w}) = \mathbb{E}[\ell_{\mathbf{w}}(\mathbf{x})] + r(\mathbf{w})$$

For the standard gradient descent, during iteration k, we need access to the actual gradient vector $\nabla_{\mathbf{w}} f(\mathbf{w}_k)$.

Let's assume it is not available (because data is too big or cannot be centralized).

Stochastic Gradient Methods

$$f(\mathbf{w}) = \mathbb{E}[\ell_{\mathbf{w}}(\mathbf{x})] + r(\mathbf{w})$$

For the standard gradient descent, during iteration k, we need access to the actual gradient vector $\nabla_{\mathbf{w}} f(\mathbf{w}_k)$.

Let's assume it is not available (because data is too big or cannot be centralized).

In this case, the expectation $f(\mathbf{w})$ must be replaced by an instantaneous approximation of it, $f_s(\mathbf{w}) = \ell_{\mathbf{w}}(\mathbf{x}) + r(\mathbf{w})$. This is what we label it as the stochastic approach:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla_{\mathbf{w}} f_s(\mathbf{w}_k)$$

Stochastic Gradient Methods

Working with gradient estimates instead of the gradient itself implies certain impairments that we have to take into account.

$$s_k\left(\mathbf{w}_k\right) = \nabla_{\mathbf{w}} f_s\left(\mathbf{w}_k\right) - \nabla_{\mathbf{w}} f\left(\mathbf{w}_k\right)$$

This noise perturbation is known as *Gradient Noise Process*

1. Prevents the stochastic iterate from converging to the optimum $\mathbf{w}^*$ when constant step sizes are used (less convergence level).
2. Some deterioration in performance occurs since the iterate $\mathbf{w}_k$ will instead fluctuate close to $\mathbf{w}^*$ in the steady state regime.

Stochastic Gradient Methods.Performance metrics

1. *Mean Square Deviation (MSD)* that refers to the fluctuations level of the sequence of the coefficients around the optimum value.
2. On the other hand the *Excess Risk (ER)* refers to the mean deviation at the end of convergence of the objective function along the sequence of coefficients with respect to the objective function particularized at the optimum value.

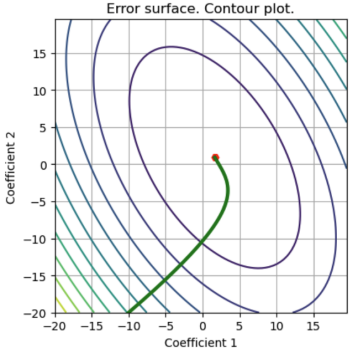
For fixed step size, we have (let us remind that for decaying step size $MSD = ER = 0$ but paying extra convergence time):

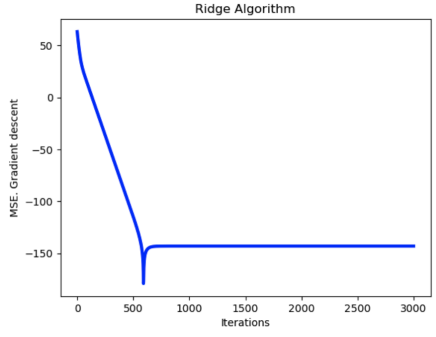
$$MSD \triangleq \lim_{k\to\infty} E\left\{\|\mathbf{w}_k - \mathbf{w}^*\|_2^2\right\}$$
$$ER = \lim_{k\to\infty} E\left\{f\left(\mathbf{w}_k\right) - f\left(\mathbf{w}^*\right)\right\}$$

Examples

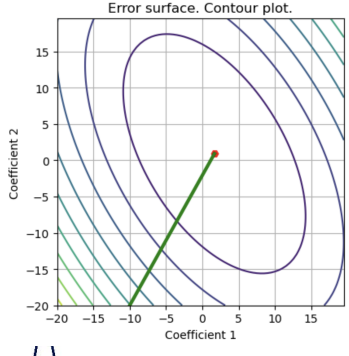
Have a look at the examples; in all of them, the addressed problem corresponds to ridge regression:

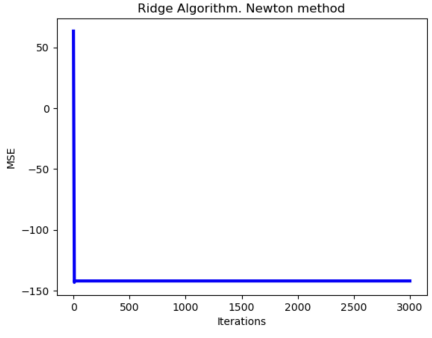
`Example_4_1`: Learning curve and error surface for FOM.



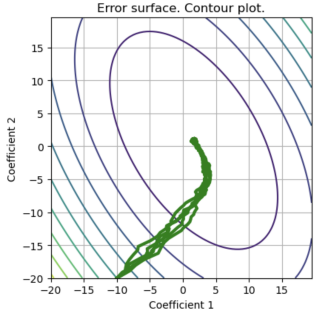


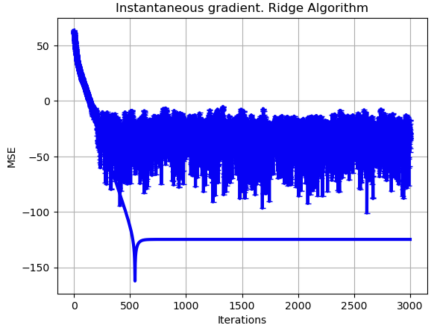
`Example_4_2`: Learning curve and error surface for SOM.





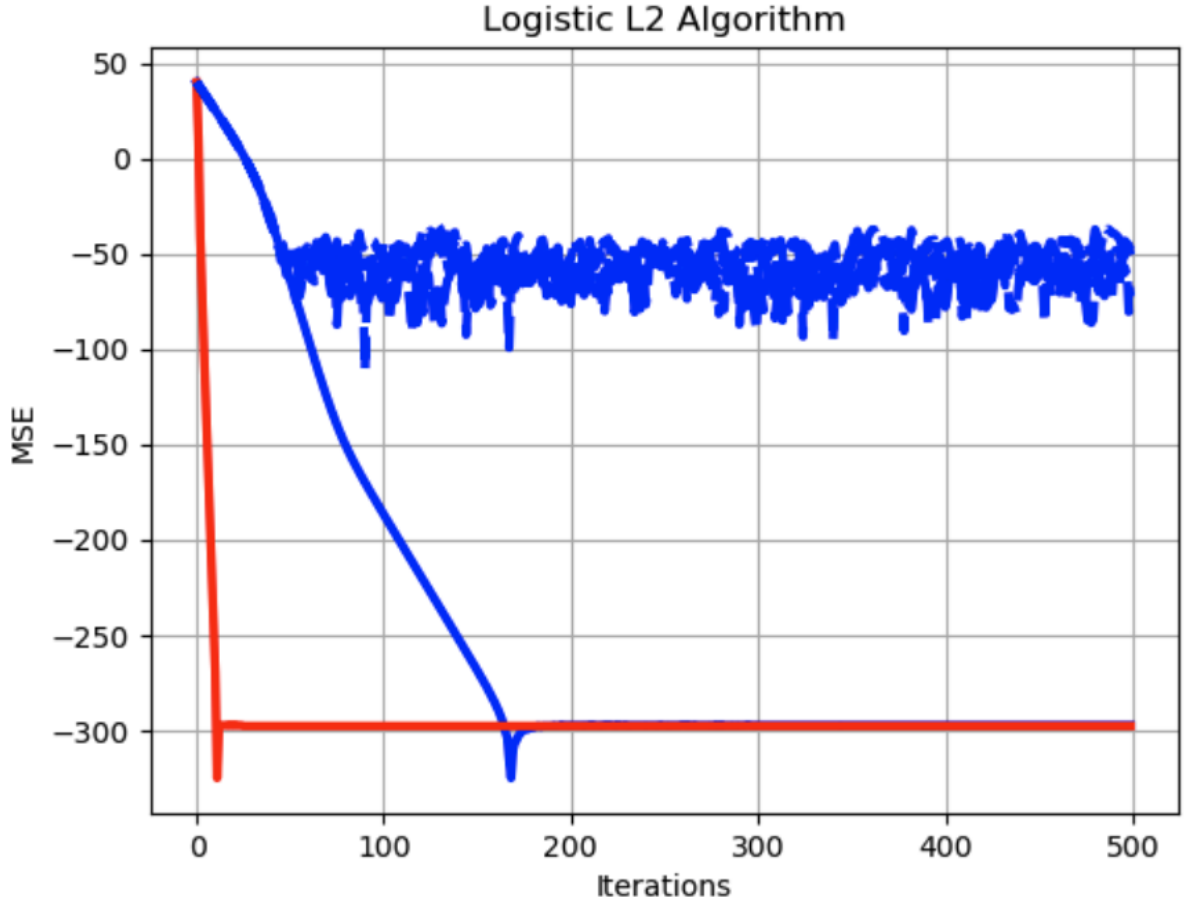
`Example_4_3`: Learning curve and error surface for SGD.



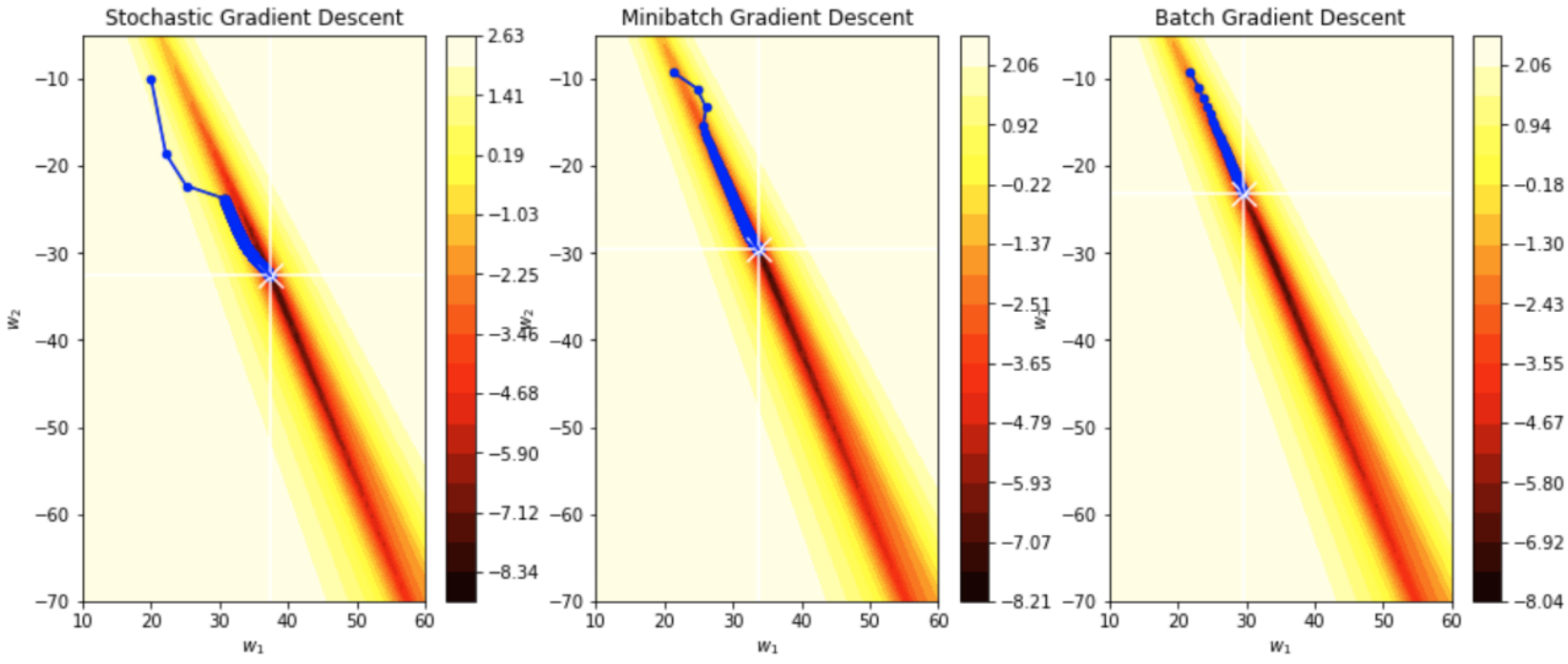


Case studies

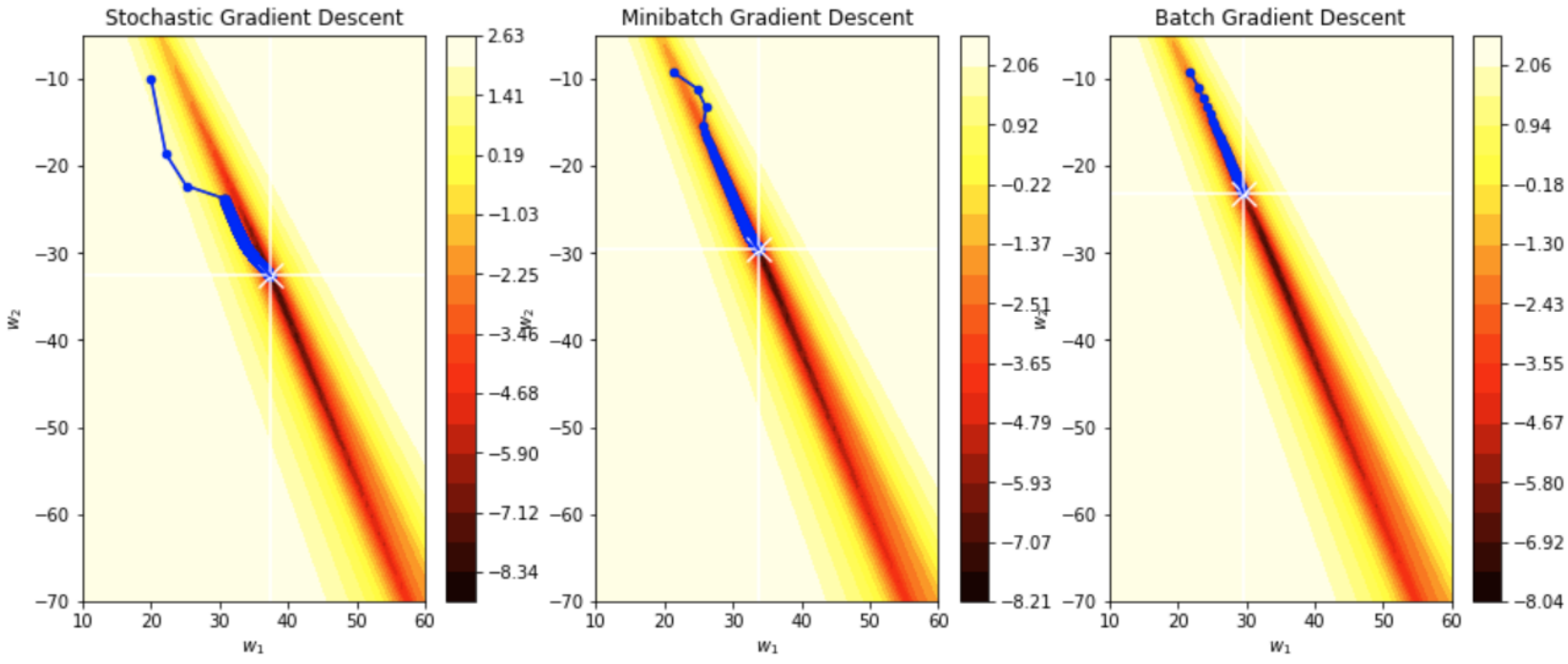
- `Case_study_4_1`: This is just an overview of the previous examples putting all together.
- `Case_study_4_2`: This is a similar exercise but focuses on the logistic functions instead.



Minibatch


Stochastic Gradient Descent
Minibatch Gradient Descent
Batch Gradient Descent
w_1
w_2
−10
−20
−30
−40
−50
−60
−70
10
20
30
40
50
60
2.63
1.41
0.19
−1.03
−2.25
−3.46
−4.68
−5.90
−7.12
−8.34
2.06
0.92
−0.22
−1.37
−2.51
−3.65
−4.79
−5.93
−7.07
−8.21
2.06
0.94
−0.18
−1.30
−2.43
−3.55
−4.67
−5.80
−6.92
−8.04

Minibatch



Stochastic behaviour has positive effects regarding saddle points!

Setting the learning rate

- Theoretical approaches regarding the estimation of L and mu are unfeasible in Big Data applications.

Setting the learning rate

- Theoretical approaches regarding the estimation of L and mu are unfeasible in Big Data applications.
- Classical approaches to update the learning rate at each step.

Setting the learning rate

- Theoretical approaches regarding the estimation of L and mu are unfeasible in Big Data applications.
- Classical approaches to update the learning rate at each step.
  - Exact Line Search: Chose $\eta_k = \arg\min_{\eta} f(\mathbf{w}_k - \eta \nabla f(\mathbf{w}_k))$

Setting the learning rate

- Theoretical approaches regarding the estimation of L and mu are unfeasible in Big Data applications.
- Classical approaches to update the learning rate at each step.
  - Exact Line Search: Chose $\eta_k = \arg\min_{\eta} f(\mathbf{w}_k - \eta \nabla f(\mathbf{w}_k))$
  - Backtracking Line Search: Reduce η_k by a factor γ until the smooth condition $f(\mathbf{w}_{k+1}) \leq f(\mathbf{w}_k) - \delta \eta_k \|\nabla f(\mathbf{w}_k)\|_2^2$ is satisfied.

Setting the learning rate

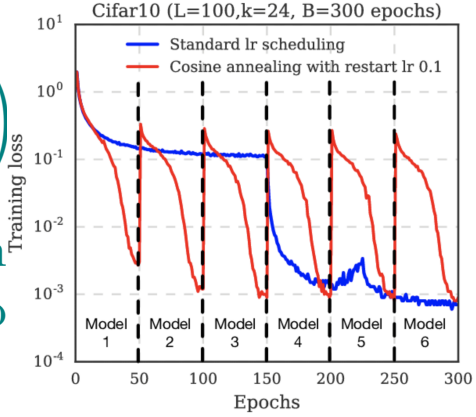
- Theoretical approaches regarding the estimation of L and mu are unfeasible in Big Data applications.
- Classical approaches to update the learning rate at each step.
  - Exact Line Search: Chose $\eta_k = \arg\min_{\eta} f(\mathbf{w}_k - \eta \nabla f(\mathbf{w}_k))$
  - Backtracking Line Search: Reduce η_k by a factor γ until the smooth condition $f(\mathbf{w}_{k+1}) \leq f(\mathbf{w}_k) - \delta \eta_k \|\nabla f(\mathbf{w}_k)\|_2^2$ is satisfied.
- Learning rate schedulers:

Setting the learning rate

- Theoretical approaches regarding the estimation of L and mu are unfeasible in Big Data applications.
- Classical approaches to update the learning rate at each step.
  - Exact Line Search: Chose $\eta_k = \arg\min_{\eta} f(\mathbf{w}_k - \eta \nabla f(\mathbf{w}_k))$
  - Backtracking Line Search: Reduce η_k by a factor γ until the smooth condition $f(\mathbf{w}_{k+1}) \leq f(\mathbf{w}_k) - \delta \eta_k \|\nabla f(\mathbf{w}_k)\|_2^2$ is satisfied.
- Learning rate schedulers:
  - Cosine annealing (warm restart) [1]:

$$\eta_k = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \left(1 + \cos\left(\frac{T_{cur}}{T}\right)\right)$$

T_{cur} accounts for how many epochs have been performed since the last restart. You can also increase T by a T_{mul} factor every restart.



Adaptive Gradient Algorithm (Adagrad)

Decay the learning rate for parameters in proportion to their update history (more updates means more decay).

The update rule used is:

$$\mathbf{g}_{k+1} = \mathbf{g}_k + (\nabla f(\mathbf{w}_k))^2$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{\sqrt{\mathbf{g}_{k+1}} + \varepsilon} \odot \nabla f(\mathbf{w}_k)$$

with $\mathbf{g}_0 = \mathbf{0}$. The step size is adjusted automatically: parameters with large accumulated gradient have a smaller step, and parameters with small accumulated gradient have a larger update.

This feature of Adagrad makes it also useful for dealing with sparse data.

RMSProp / Adadelta.

In Adagrad, the sum of the gradients is always increasing; thus the algorithm stops learning eventually.

Unlike Adagrad, in **RMSProp** instead of allowing this sum to increase continuously over the training period, we allow the sum to decrease.

$$\mathbf{g}_{k+1} = \beta \mathbf{g}_k + (1-\beta)\left(\nabla f(\mathbf{w}_k)\right)^2$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{\sqrt{\mathbf{g}_{k+1}} + \varepsilon} \odot \nabla f(\mathbf{w}_k)$$

with $\mathbf{g}_0 = \mathbf{0}$, $\beta \simeq 0.9$. RMSProp exhibits the same property of speeding up the updating of the weights along one dimension and slowing down the motion along the other.

A similar algorithm to RMSProp, developed independently, is **Adadelta** [3].

Adaptive moment estimator (Adam)

Adam intuitively consists in adding momentum to RMSProp to improve its convergence speed.

Adaptive moment estimator (Adam)

Adam intuitively consists in adding momentum to RMSProp to improve its convergence speed.

$$\mathbf{v}_{k+1} = \beta_1 \mathbf{v}_k - (1 - \beta_1) \nabla f(\mathbf{w}_k)$$

$$\mathbf{g}_{k+1} = \beta_2 \mathbf{g}_k + (1 - \beta_2) \left(\nabla f(\mathbf{w}_k)\right)^2$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{\sqrt{\mathbf{g}_{k+1}} + \varepsilon} \odot \mathbf{v}_{k+1}$$

Adaptive moment estimator (Adam)

Adam intuitively consists in adding momentum to RMSProp to improve its convergence speed.

$$\mathbf{v}_{k+1} = \beta_1 \mathbf{v}_k - (1-\beta_1)\nabla f(\mathbf{w}_k)$$

$$\mathbf{g}_{k+1} = \beta_2 \mathbf{g}_k + (1-\beta_2)\left(\nabla f(\mathbf{w}_k)\right)^2$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{\sqrt{\mathbf{g}_{k+1}} + \varepsilon} \odot \mathbf{v}_{k+1}$$

To address the giant update steps happening at the beginning of training, Adam applies a bias correction: $\hat{\mathbf{v}}_{k+1} = \frac{\mathbf{v}_{k+1}}{1-\beta_1^k}$ and $\hat{\mathbf{g}}_{k+1} = \frac{\mathbf{g}_{k+1}}{1-\beta_2^k}$. Usually $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

Adaptive moment estimator (Adam)

Adam intuitively consists in adding momentum to RMSProp to improve its convergence speed.

$$\mathbf{v}_{k+1} = \beta_1 \mathbf{v}_k - (1-\beta_1)\nabla f\left(\mathbf{w}_k\right)$$

$$\mathbf{g}_{k+1} = \beta_2 \mathbf{g}_k + (1-\beta_2)\left(\nabla f(\mathbf{w}_k)\right)^2$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{\sqrt{\hat{\mathbf{g}}_{k+1}} + \varepsilon} \odot \hat{\mathbf{v}}_{k+1}$$

To address the giant update steps happening at the beginning of training, Adam applies a bias correction: $\hat{\mathbf{v}}_{k+1} = \frac{\mathbf{v}_{k+1}}{1-\beta_1^k}$ and $\hat{\mathbf{g}}_{k+1} = \frac{\mathbf{g}_{k+1}}{1-\beta_2^k}$. Usually $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

Case_studies and examples

Go to the repository and take a look to following notebooks:

- `Example_4_4`: Ridge regression using Adam optimizer.
- `Case_study_4_3`: Vanilla NN for a regression problem using SGD.
- `Case_study_4_4`: Same network than before but using Adam optimizer.
- `Case_study_4_5`: Vanilla NN for solving the XOR problem.
- `Case_study_4_6`: Vanilla NN for a nonlinear classification problem.

Parallel and distributed gradient

Gradient methods are also amenable to parallel and distributed implementations:

- Parallel refers to using multiple processors in the same computer.
- Distributed is a more general concept, in which the computations are distributed among many different computers. It has to deal with:
  - Latency problems
  - Synchronicity problems

Parallel and distributed gradient

Gradient methods are also amenable to parallel and distributed implementations:

- Parallel refers to using multiple processors in the same computer.
- Distributed is a more general concept, in which the computations are distributed among many different computers. It has to deal with:
  - Latency problems
  - Synchronicity problems

Let us recall the gradient update rule for the least squares problem:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mu_k \left[\frac{2}{n} \sum_{i=1}^{n} \left(\mathbf{w}_k^T \mathbf{x}_i - y_i \right) \mathbf{x}_i \right]$$

Parallel and distributed gradient

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mu_k \left[\frac{2}{n} \sum_{i=1}^{n} \left(\mathbf{w}_k^T \mathbf{x}_i - y_i \right) \mathbf{x}_i \right]$$

If we have a computer with $M = 1$ processor, for each iteration k, the processor must:

- Obtain the n vector operations of the sum
- Perform the sum
- Perform the update

Parallel and distributed gradient

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mu_k \left[\frac{2}{n} \sum_{i=1}^{n} \left(\mathbf{w}_k^T \mathbf{x}_i - y_i \right) \mathbf{x}_i \right]$$

If we have a computer with $M = 1$ processor, for each iteration k, the processor must:

- Obtain the n vector operations of the sum
- Perform the sum
- Perform the update

The most time-consuming part is obtaining the n sum terms, which you may notice are independent.

Parallel and distributed gradient

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mu_k \left[\frac{2}{n} \sum_{i=1}^{n} \left(\mathbf{w}_k^T \mathbf{x}_i - y_i \right) \mathbf{x}_i \right]$$

If we have a computer with $M = 1$ processor, for each iteration k, the processor must:

- Obtain the n vector operations of the sum
- Perform the sum
- Perform the update

The most time-consuming part is obtaining the n sum terms, which you may notice are independent.

So, if we have hardware with $M > 1$, we can split the computations of the sum terms across them.

Map - Reduce

Map - Reduce is a programming paradigm developed for processing data in distributed file systems (Hadoop). Its logic is based on two operations: 1) Map and 2) Reduce.

Map - Reduce

Map - Reduce is a programming paradigm developed for processing data in distributed file systems (Hadoop). Its logic is based on two operations: 1) Map and 2) Reduce.

Applied to GD it would work like this:

- Map step: split the n-sized training dataset into M subsets (partitions) with L training samples each, and let each processor obtain $\mathbf{p}_{m,k}, m \in 1, 2, ..., M$:

$$\mathbf{p}_{m,k} = \sum_{j=(m-1)L}^{mL-1} \left(\mathbf{w}_k^T \mathbf{x}_j - y_j\right) \mathbf{x}_j$$

Map - Reduce

- Reduce step: update the gradient:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - 2\frac{\mu_k}{n}\sum_{m=1}^{M}\mathbf{p}_{m,k}$$

This step collects the computations done in the Map step to obtain an update.

Map - Reduce

- Reduce step: update the gradient:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - 2\frac{\mu_k}{n}\sum_{m=1}^{M}\mathbf{p}_{m,k}$$

This step collects the computations done in the Map step to obtain an update.

If Reduce computation time is negligible compared to the Map computation time, the parallel version may speed up the computation by a factor up to M.

This, however, is hard to reach in real life due to other possible bottlenecks, such as memory access.

Distributed gradient

- The Map-Reduce scheme can also be used in a distributed setting, but we must consider communication-related problems (latency), especially in **synchronous** schemes, as we need all computers to wait for the others to make an update.
- It is possible to develop more advanced, **asynchronous** approaches, such as HOGWILD! [2], that only requires nodes (processors) access to shared memory with the possibility of overwriting each other work.
- If the optimization problem is sparse (most gradient updates only modify small parts of the decision variable), HOGWILD! achieves an early optimal rate of convergence.

Acknowledgments

I would like to acknowledge several sources I have used to create slides

- Akshay L Chandra, "Learning Parameters, Part 5: AdaGrad, RMSProp, and Adam". `https://towardsdatascience.com/learning-param...`
- Artem Oppermann, "Optimization in Deep Learning: AdaGrad, RMSProp, ADAM". `https://artemoppermann.com/optimization-in-deep...`

Questions?

References

[1] Ilya Loshchilov and Frank Hutter. "Sgdr: Stochastic gradient descent with warm restarts". In: *arXiv preprint arXiv:1608.03983* (2016).

[2] Benjamin Recht et al. "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems* 24 (2011).

[3] Matthew D Zeiler. "Adadelta: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).

Thank You

Julián D. Arias-Londoño
julian.arias@upm.es