

Introduction to Deep Learning

Mnacho Echenim

Grenoble INP-Ensimag

2022-2023



Notes

Derivation of the backpropagation rules (simplified notations)

- **Goal:** get rid of cumbersome notations to represent partial derivatives
- Closer to what may be found in textbooks
- We want to compute $\frac{\partial C}{\partial w_k^j}$ and $\frac{\partial C}{\partial b^j}$
- We have the following equalities:

$$\begin{aligned}\frac{\partial C}{\partial w_k^j} &= p^{j+1} \cdot \frac{\partial a^j}{\partial w_k^j} \\ \frac{\partial C}{\partial b^j} &= p^{j+1} \cdot \frac{\partial a^j}{\partial b^j}\end{aligned}$$



Notes

Derivation of the backpropagation rules (part 2)

$$\frac{\partial \mathbf{a}^j}{\partial \mathbf{a}^{j-1}} = \begin{pmatrix} \Phi'(\zeta_1^j) \cdot [\omega_1^j]^T \\ \vdots \\ \Phi'(\zeta_{n_j}^j) \cdot [\omega_{n_j}^j]^T \end{pmatrix}$$

$$\frac{\partial \mathbf{a}^j}{\partial \mathbf{w}_k^j} = \begin{pmatrix} 0 \\ \vdots \\ \Phi'(\zeta_k^j) \cdot [\alpha^{j-1}]^T \\ \vdots \\ 0 \end{pmatrix} \leftarrow \text{line } k$$

$$\frac{\partial \mathbf{a}^j}{\partial \mathbf{b}^j} = \text{diag}(\Phi'(\zeta_1^j), \dots, \Phi'(\zeta_{n_j}^j))$$

The backpropagation equations are then derived as previously

Notes

Mini-batch gradient descent

- We are given M inputs $\overline{\alpha^0} \stackrel{\text{def}}{=} (\alpha_{(1)}^0, \dots, \alpha_{(M)}^0)$ and outputs $\overline{\rho} \stackrel{\text{def}}{=} (\rho_{(1)}, \dots, \rho_{(M)})$
- The parameters updates become

$$\Omega^j \leftarrow \Omega^j - \frac{\eta}{M} \cdot \sum_{k=1}^M \nabla_{\mathbf{w}^j} \mathcal{E}(\alpha_{(k)}^0, \dots, \Omega^L, \beta^L, \rho_{(k)})$$

$$\beta^j \leftarrow \beta^j - \frac{\eta}{M} \cdot \sum_{k=1}^M \nabla_{\mathbf{b}^j} \mathcal{E}(\alpha_{(k)}^0, \dots, \Omega^L, \beta^L, \rho_{(k)})$$

- **Goal:** compute these updates efficiently using matrix operations

Notes

Extensions to mini-batches

Inputs to the network: $\overline{\alpha^0}, \Omega_1, \beta^1, \dots, \Omega^L, \beta^L$. We let:

$$\overline{\beta^i} \stackrel{\text{def}}{=} (\beta^i, \dots, \beta^i) \quad (M \text{ columns})$$

$$\overline{\alpha}^i \stackrel{\text{def}}{=} (\alpha_{(1)}^i, \dots, \alpha_{(M)}^i) = f_i(\overline{\alpha}^{i-1}, \Omega^i, \beta^i)$$

$$\overline{\zeta^i} \stackrel{\text{def}}{=} (\zeta_{(1)}^i, \dots, \zeta_{(M)}^i)$$

$$\Phi'(\overline{\zeta^i}) \stackrel{\text{def}}{=} (\Phi'(\zeta_{(1)}^i), \dots, \Phi'(\zeta_{(M)}^i))$$

$$\overline{\mathcal{B}}^i \stackrel{\text{def}}{=} (\mathcal{B}_{(1)}^i, \dots, \mathcal{B}_{(M)}^i)$$

$$\nabla_{a^L} \mathcal{C}(\overline{\alpha^L}, \bar{\rho}) = (\nabla_{a^L} \mathcal{C}(\alpha_{(1)}^L, \rho_{(1)}), \dots, \nabla_{a^L} \mathcal{C}(\alpha_{(M)}^L, \rho_{(M)}))$$

Proposition

We have the following equalities:

- $\bar{\zeta}^i = \Psi(\bar{\alpha}^{i-1}, \Omega^i, \bar{\beta}^i) = [\Omega^i]^\top \cdot \bar{\alpha}^{i-1} + \bar{\beta}^i$ for $i = 1, \dots, L$
- $\bar{\mathcal{B}}^L = \Phi'(\bar{\zeta}^L) \odot \nabla_{\mathbf{a}^L} \mathcal{C}(\bar{\alpha}^L, \bar{\rho})$
- $\bar{\mathcal{B}}^j = \Phi'(\bar{\zeta}^j) \odot (\Omega^{j+1} \cdot \bar{\mathcal{B}}^{j+1})$

Other computations

Summary

Let $\mathbf{1}_M \stackrel{\text{def}}{=} (1, \dots, 1)^T \in \mathbb{R}^M$. For M inputs and outputs, parameter updates become:

$$\Omega^j \leftarrow \Omega^j - \frac{\eta}{M} \cdot \left(\overline{\alpha^{j-1}} \cdot [\overline{\mathcal{B}^j}]^T \right)$$

$$\beta_j \leftarrow \beta_j - \frac{\eta}{M} \cdot (\overline{\mathcal{B}}^j \cdot \mathbf{1}_M)$$

Notes

Notes

Mini-batch forward and backpropagation algorithms

Input: A network with L layers

Input: $\bar{\alpha}^0 = (\alpha_{(1)}^0, \dots, \alpha_{(M)}^0)$

```
1 for  $i \leftarrow 1$  to  $L$  do
2    $\bar{\zeta}^i \leftarrow [\Omega^i]^T \cdot \bar{\alpha}^{i-1} + \bar{\beta}^i$ ;
3    $\bar{\alpha}^i \leftarrow \Phi(\bar{\zeta}^i)$ ;
4 end
```

Algorithm 1: Forward propagation

Input: A network with L layers

Input: $\bar{\alpha}^0 = (\alpha_{(1)}^0, \dots, \alpha_{(M)}^0)$ that has been forward propagated

Input: $\bar{\rho} = (\rho_{(1)}, \dots, \rho_{(M)})$ the expected outputs

```
1  $\bar{B}^L \leftarrow \Phi'(\bar{\zeta}^L) \odot \nabla_{\alpha^L} \mathcal{C}(\alpha^L, \bar{\rho})$ ;
2  $[\mathcal{P}^L]^T \leftarrow \Omega^L \cdot \bar{B}^L$ ;
3 for  $j \leftarrow L-1$  to 1 do
4    $\bar{B}^j \leftarrow \Phi'(\bar{\zeta}^j) \odot [\mathcal{P}^{j+1}]^T$ ;
5    $[\mathcal{P}^j]^T \leftarrow \Omega^j \cdot \bar{B}^j$ ;
6 end
```

Algorithm 2: Backpropagation

Notes

Mini-batch gradient computation

Input: A network with L layers

Input: $\bar{\alpha}^0 = (\alpha_{(1)}^0, \dots, \alpha_{(M)}^0)$ that has been forward propagated

Input: $(\bar{B}^1, \dots, \bar{B}^L)$ that have been updated by backpropagation

```
1 for  $j \in \{1, \dots, L\}$  do
2   Gradient( $\Omega^j$ )  $\leftarrow \frac{1}{M} \cdot (\bar{\alpha}^{j-1} \cdot [\bar{B}^j]^T)$ ;
3   Gradient( $\beta^j$ )  $\leftarrow \frac{1}{M} \cdot (\bar{B}^j \cdot \mathbf{1}_M)$ ;
4 end
```

Algorithm 3: Mini-batch gradient computation

Note

By storing the necessary information, backpropagation and mini-batch gradient computations can be carried out in the same loop

Notes

Computed quantities in backpropagation

- At layer $j \in \llbracket 1, L \rrbracket$, two central quantities are computed
 - ▶ $\mathcal{P}^j \in \mathbb{R}^{1 \times n_{j-1}}$ and $\mathcal{B}^j \in \mathbb{R}^{n_j}$
- \mathcal{P}^j represents the information that is transmitted to layer $j - 1$
- \mathcal{B}^j represents the information necessary to compute gradients at layer j
- What does \mathcal{P}^j represent in the backpropagation rules?

Proposition

We have $\mathcal{P}^j = \frac{\partial \mathcal{C}}{\partial \mathbf{a}^{j-1}}$

Proof: by induction on j

Notes

What does \mathcal{B}^j represent in the backpropagation rules?

- Recall that $\nabla_{\mathbf{w}^j} \mathcal{C} = \alpha^{j-1} \cdot [\mathcal{B}^j]^\top$ and $\nabla_{\mathbf{b}^j} \mathcal{C} = \mathcal{B}^j$ for $j = 1, \dots, L$

Theorem

If \mathbf{z}^j is a formal parameter representing the net input of layer j , then $\mathcal{B}^j = \nabla_{\mathbf{z}^j} \mathcal{C}$

Proof (backward induction)

- $\mathcal{B}^L = \Phi'(\zeta^L) \odot \nabla_{\mathbf{a}^L} \mathcal{C}$, where $\zeta^L = (\zeta_1^L, \dots, \zeta_{n_L}^L)^\top$ and $\alpha^L = \Phi(\zeta^L)$, hence

$$\begin{aligned}\mathcal{B}^L &= \left[\frac{\partial \mathcal{C}}{\partial \mathbf{a}^L} \right]^\top \cdot \text{diag}(\Phi'(\zeta_1^L), \dots, \Phi'(\zeta_{n_L}^L)) \\ &= \left[\frac{\partial \mathcal{C}}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} \right]^\top \\ &= \left[\frac{\partial \mathcal{C}}{\partial \mathbf{z}^L} \right]^\top \\ &= \nabla_{\mathbf{z}^L} \mathcal{C}\end{aligned}$$

Notes

What does \mathcal{B}^j represent? (2)

- Assume $\mathcal{B}^{j+1} = \nabla_{\mathbf{z}^{j+1}} \mathcal{C}$; then we have

$$\begin{aligned}\zeta^{j+1} &= [\Omega^{j+1}]^T \cdot \alpha^j + \beta^{j+1} \\ &= [\Omega^{j+1}]^T \cdot \Phi(\zeta^j) + \beta^{j+1}\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \mathbf{z}^j} &= \frac{\partial \mathcal{C}}{\partial \mathbf{z}^{j+1}} \cdot \frac{\partial \mathbf{z}^{j+1}}{\partial \mathbf{z}^j} \\ &= [\mathcal{B}^{j+1}]^T \cdot \frac{\partial \mathbf{z}^{j+1}}{\partial \mathbf{z}^j}\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathbf{z}^{j+1}}{\partial \mathbf{z}^j} &= \frac{\partial \left([\Omega^{j+1}]^T \cdot \alpha^j + \beta^{j+1} \right)}{\partial \mathbf{z}^j} \\ &= \frac{\partial \left([\Omega^{j+1}]^T \cdot \alpha^j + \beta^{j+1} \right)}{\partial \mathbf{a}^j} \cdot \frac{\partial \Phi(\zeta^j)}{\partial \mathbf{z}^j} \\ &= [\Omega^{j+1}]^T \cdot \text{diag} \left(\Phi'(\zeta_1^j), \dots, \Phi'(\zeta_{n_j}^j) \right)\end{aligned}$$

Notes

Bringing it all together

We deduce that:

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \mathbf{z}^j} &= [\mathcal{B}^{j+1}]^T \cdot \frac{\partial \mathbf{z}^{j+1}}{\partial \mathbf{z}^j} \\ &= [\mathcal{B}^{j+1}]^T \cdot \left[[\Omega^{j+1}]^T \cdot \text{diag} \left(\Phi'(\zeta_1^j), \dots, \Phi'(\zeta_{n_j}^j) \right) \right] \\ &= [\Omega^{j+1} \cdot \mathcal{B}^{j+1}]^T \cdot \text{diag} \left(\Phi'(\zeta_1^j), \dots, \Phi'(\zeta_{n_j}^j) \right)\end{aligned}$$

Therefore:

$$\begin{aligned}\nabla_{\mathbf{z}^j} \mathcal{C} &= \text{diag} \left(\Phi'(\zeta_1^j), \dots, \Phi'(\zeta_{n_j}^j) \right) \cdot (\Omega^{j+1} \cdot \mathcal{B}^{j+1}) \\ &= \Phi'(\zeta^j) \odot (\Omega^{j+1} \cdot \mathcal{B}^{j+1}) \\ &= \mathcal{B}^j\end{aligned}$$

Notes

Foreword

- What follows is a list of techniques that are often used to improve gradient descent
- There are (currently) no formal proofs that these methods improve anything on neural networks
 - ▶ They may be viewed by some as nothing more than recipes
- **But**
 - ▶ Several originate from research on improving gradient descent on convex optimization problems
 - ▶ E.g., the Nesterov momentum method permits to obtain optimal convergence rates for convex optimization problems
 - ▶ It is not far-fetched to try them on non-convex optimization problems

Notes



Reducing noise for mini-batch gradient descent

Issues with mini-batch gradient descent

- As we get closer to an optimum, the noise in the gradient estimate can become a problem

How can this issue be handled?

- First idea: “denoise” the estimate by increasing the batch sizes
- Another idea: decrease the learning rate over time with decrease factor δ
 - ▶ Inverse decay: $\eta_k = \frac{\eta_0}{1+k\delta}$
 - ▶ Exponential decay: $\eta_k = \eta_0 \exp(-k\delta)$
 - ▶ ...

Note

It is still difficult to choose the initial rate, decrease factor, decrease schedule...

Notes



Is it guaranteed gradient descent will work?

- Problem: are we sure gradient descent will lead to a global minimum of the function?
- In general no: most of the time, the function to optimize is not convex
- Gradient descent could get stuck on
 - ▶ Local minima
 - ▶ Stationary points
- Can the algorithm be adapted to produce better results?

Notes

Adaptive learning rates

- Problem: fixing the right learning rate is difficult, even using decay strategies
- It is not clear whether the same learning rate should be used on every component
- Principle: use information about computed gradients to set learning rates individually for each component
- First approach: delta-bar-delta (Jacobs, 88)
 - ▶ Principle: check the sign of each partial derivative
 - ★ If it stays the same, then the optimization direction is correct: **increase the learning rate**
 - ★ Otherwise **decrease the learning rate**
 - ▶ **Nb:** This approach can only be applied to full gradient descent
 - ★ The noise in SGD can produce wrong signals
- Upcoming approaches: all operations on vectors and matrices are **componentwise**

Notes

AdaGrad (Duchi et al, 2011)

- Principle: keep track of aggregated squared magnitude of gradients; use this to scale the individual learning rates
 - ▶ Large partial derivatives lead to a fast decrease, and vice-versa
- This technique has good properties in a convex setting
 - ▶ But the fact that **all** gradients are accumulated may lead to an excessive decrease of the learning rate

Input: Step size η , numerical stabilizer δ

Input: Initial parameters θ

```
1 set gradient accumulation variable  $r$  to 0;  
2 for  $i \leftarrow 1$  to number of training steps do  
3    $g \leftarrow$  computation of  $\nabla_{\theta} \mathcal{C}(\theta)$  //full gradient, mini-batch...;  
4    $r \leftarrow r + (g \odot g)$ ;  
5    $v \leftarrow -\frac{\eta}{\delta + \sqrt{r}} \odot g$ ;  
6    $\theta \leftarrow \theta + v$ ;  
7 end
```

Notes

RMSProp (Hinton, 2012)

- Principle: accumulate squared magnitude of gradients, with an exponentially weighted moving average
 - ▶ Ancient history is discarded, as old gradients decay exponentially with time
- Example: on a slope with a locally convex bowl:
 - ▶ AdaGrad will likely get stuck
 - ▶ RMSProp will forget about the past and get out of the bowl

Input: Step size η , numerical stabilizer δ

Input: Exponential decay rate $\rho \in [0, 1[$

Input: Initial parameters θ

```
1 set gradient accumulation variable  $r$  to 0;  
2 for  $i \leftarrow 1$  to number of training steps do  
3    $g \leftarrow$  computation of  $\nabla_{\theta} \mathcal{C}(\theta)$  //full gradient, mini-batch...;  
4    $r \leftarrow \rho \cdot r + (1 - \rho) \cdot (g \odot g)$ ;  
5    $v \leftarrow -\frac{\eta}{\sqrt{\delta + r}} \odot g$ ;  
6    $\theta \leftarrow \theta + v$ ;  
7 end
```

Notes

AdaDelta (Zeiler, 2012)

- Principle: moving average, similarly to RMSProp
 - ▶ Main difference: **no input step size**
 - ▶ Step size is updated depending on previous changes

Input: Numerical stabilizer δ

Input: Exponential decay rate $\rho \in [0, 1[$

Input: Initial parameters θ

```
1 set gradient accumulation variable  $r$  to 0;  
2 set parameter update accumulation variable  $s$  to 0;  
3 for  $i \leftarrow 1$  to number of training steps do  
4    $g \leftarrow$  computation of  $\nabla_{\theta} \mathcal{C}(\theta)$  //full gradient, mini-batch...;  
5    $r \leftarrow \rho \cdot r + (1 - \rho) \cdot (g \odot g)$ ;  
6    $v \leftarrow -\frac{\sqrt{\delta+s}}{\sqrt{\delta+r}} \odot g$ ;  
7    $s \leftarrow \rho \cdot s + (1 - \rho) \cdot (v \odot v)$ ;  
8    $\theta \leftarrow \theta + v$ ;  
9 end
```

Notes

Adam (Kingma & Ba, 2014)

- Can be viewed as a combination of RMSProp and Momentum (coming up later)
 - ▶ Momentum is added by considering an exponentially weighted moving average of gradients
- Main difference: a bias correction term is applied to the first and second moment variables

Input: Step size η , numerical stabilizer δ

Input: Exponential decay rates $\rho_1, \rho_2 \in [0, 1[$

Input: Initial parameters θ

```
1 set first, second moment variable  $s, r$  to 0;  
2 for  $i \leftarrow 1$  to number of training steps do  
3    $g \leftarrow$  computation of  $\nabla_{\theta} \mathcal{C}(\theta)$  //full gradient, mini-batch...;  
4    $s \leftarrow \rho_1 \cdot s + (1 - \rho_1) \cdot g$ ;  
5    $r \leftarrow \rho_2 \cdot r + (1 - \rho_2) \cdot (g \odot g)$ ;  
6    $s' \leftarrow \frac{s}{1 - \rho_1^i}$ ;  
7    $r' \leftarrow \frac{r}{1 - \rho_2^i}$ ;  
8    $v \leftarrow -\eta \cdot \frac{s'}{\delta + \sqrt{r'}}$ ;  
9    $\theta \leftarrow \theta + v$ ;  
10 end
```

Notes

Momentum

- Principle: consider a *velocity* variable v
 - ▶ **Intuition:** v provides the direction and speed at which θ should move toward the optimum
 - ▶ **How:** by using information from previous gradient computations
 - ▶ For $\delta \in [0, 1[$, at each epoch, v is updated by $v \leftarrow \delta \cdot v - \eta \nabla_{\theta} \mathcal{C}(\theta)$
- Algorithm:
 - Input:** Learning rate η , momentum parameter δ
 - Input:** Initial parameters θ , initial velocity $v = 0$
 - 1 **for** $i \leftarrow 1$ **to** *number of training steps* **do**
 - 2 $g \leftarrow$ computation of $\nabla_{\theta} \mathcal{C}(\theta_k)$ // *full gradient, mini-batch...*;
 - 3 $v \leftarrow \delta \cdot v - \eta \cdot g$;
 - 4 $\theta \leftarrow \theta + v$
 - 5 **end**

Notes

Nesterov momentum

- Principle: use a *modified* velocity variable v
 - ▶ Intuition: look ahead with current velocity before computing gradient
 - ▶ For $\delta \in [0, 1[$, at each epoch, v is updated by $v \leftarrow \delta \cdot v - \eta \nabla_{\theta} \mathcal{C}(\theta + \delta \cdot v)$
- Algorithm:
 - Input:** Learning rate η , momentum parameter δ
 - Input:** Initial parameters θ , initial velocity $v = 0$
 - 1 **for** $i \leftarrow 1$ **to** *number of training steps* **do**
 - 2 $\tilde{\theta} \leftarrow \theta + \delta \cdot v$;
 - 3 $g \leftarrow$ computation of $\nabla_{\theta} \mathcal{C}(\tilde{\theta})$ // *full gradient, mini-batch...*;
 - 4 $v \leftarrow \delta \cdot v - \eta \cdot g$;
 - 5 $\theta \leftarrow \theta + v$;
 - 6 **end**

Notes

Implementing Nesterov Accelerated Gradient

- Update rules:

$$\begin{aligned}v &\leftarrow \delta \cdot v - \eta \nabla_{\theta} \mathcal{C}(\theta + \delta \cdot v) \\ \theta &\leftarrow \theta + v\end{aligned}$$

- The gradient computation is *forward looking*
- This does not fit well in a generic implementation of forward and backpropagation algorithms
- A solution:** “simplified” Nesterov momentum (Bengio et al.: *Advances in optimizing recurrent networks*. 2012)

Notes

Simplified Nesterov momentum

- Perform updates on the lookahead variable $\Theta \stackrel{\text{def}}{=} \theta + \delta \cdot v$

Notes

- Algorithm:

Input: Learning rate η , momentum parameter δ

Input: Initial parameters θ , initial velocity $v = 0$

```
1  $\Theta \leftarrow \theta$ ;  
2 for  $i \leftarrow 1$  to number of training steps do  
3    $g \leftarrow$  computation of  $\nabla_{\theta} \mathcal{C}(\Theta)$  // full gradient, mini-batch...;  
4    $\Theta \leftarrow \Theta + \delta^2 \cdot v - \eta \cdot (1 + \delta) \cdot g$ ;  
5    $v \leftarrow \delta \cdot v - \eta \cdot g$ ;  
6 end
```

- If the optimum is reached at step n ($v_n = 0$) then $\Theta_n = \theta_n$

Which optimization to choose?

- No clear answer
- Momentum and Adam are quite popular
- Recommendation: choose an optimizer and practice tuning its hyperparameters
- Some reading material
 - ▶ Momentum: <https://distill.pub/2017/momentum/>
 - ▶ Adam: <https://arxiv.org/abs/1412.6980>

Notes

Notes
