Machine Learning

Neural Networks and Deep Learning: Training

Gabriele Russo Russo   Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Training Neural Networks

- In principle, training an ANN is not different than training the other parametric models you studied so far
- The goal is determining the model parameters $\boldsymbol{\theta}$ that optimize a cost function $J(\boldsymbol{\theta})$
  - e.g., via stochastic gradient descent
- In this case, $\boldsymbol{\theta}$ includes:
  - the connection weights, $\boldsymbol{W}^{(\ell)}, \forall 1 \leq \ell \leq L$
  - the bias vectors, $\boldsymbol{b}^{(\ell)}, \forall 1 \leq \ell \leq L$

Training Neural Networks: Cost function

Which is the cost/objective function to optimize?

- Let's consider a regression task and the squared error as the loss function
$$\mathcal{L} = \frac{1}{2}(t - y)^2 \tag{1}$$
where t is the prediction target and y is the NN prediction.
- The cost $J(\boldsymbol{\theta})$ may comprise a regularization term
$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \frac{1}{2}(t^{(i)} - y^{(i)})^2 + \lambda\Omega(\boldsymbol{\theta}) \tag{2}$$
- e.g., assuming ℓ_2 regularization:
$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \frac{1}{2}(t^{(i)} - y^{(i)})^2 + \lambda\|\boldsymbol{\theta}\|_2 \tag{3}$$

Training Neural Networks: SGD

Cannot we just keep using SGD?
Short answer: Yes

NN training via SGD (a naive algorithm!)

```
1 for epoch ← 1, ..., $N_{epochs}$ do
      /* Loop over training samples                              */
2     for $i = 1, ..., N$ do
3         $\hat{g} \leftarrow \nabla_{\boldsymbol{\theta}} J^{(i)}(\boldsymbol{\theta})$        /* Compute the gradient */
4         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \hat{g}$
5     end
6 end
```

Training Neural Networks: SGD (2)

Cannot we just keep using SGD?

- A few more challenges involved…
- Main issue: nonlinearity of NNs cause loss functions to become nonconvex
  - We must use gradient-based methods to drive the cost function to a low value
  - No global convergence guarantees!
- How to (efficiently) compute the gradient?
  - For single-layer NNs (e.g., Perceptron), the loss is just a function of the weights: easy
  - For multi-layer NNs, loss is a composite function of the weights of the previous layers
  - Deriving an analytical expression is easy, but evaluation can be computationally expensive

Backpropagation

- Backpropagation (or, backprop) is an algorithm to compute the gradient of a function, including the gradient of the NN cost function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
  - Rumelhart, Hinton, Williams (1986). *Learning representations by back-propagating errors*, Nature, 323 (6088): 533–536.
- The term “backpropagation” is often used loosely to refer to the entire training algorithm – including how the gradient is used, such as by SGD – but this is not strictly correct

Backpropagation: Overview

The algorithm consists of 2 phases:

- Forward phase: A training instance is fed into the NN as input, resulting in a cascade of computations across the layers. The final output is used to compute the cost function.
- Backward phase: The gradient of the cost function with respect to the parameters is computed, traversing the NN in the opposite direction (starting from the output layer).

To efficiently compute the gradient, backprop leverages dynamic programming and the chain rule of calculus.

Forward Propagation

- Forward propagation (or forward pass): computation of intermediate variables and outputs in order, from the input layer to the output layer
- We saw forward propagation in action in the xor notebook

For a NN with a single hidden layer, forward prop. computes:

- $\boldsymbol{z} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$
- $\boldsymbol{h} = \phi(\boldsymbol{z})$
- $\boldsymbol{y} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}$

Recall: Chain Rule of Calculus

- Let $f : \mathbb{R} \to \mathbb{R}$, $g : \mathbb{R} \to \mathbb{R}$
- The derivative of the composite function $f(g(x))$, according to the chain rule, is computed as:

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x) \tag{4}$$

- Intuitively: increasing x by some infinitesimal quantity h_1 “causes” g to change by the infinitesimal $h_2 = g'(x)h_1$. This in turn causes f to change by $f'(g(x))h_2 = f'(g(x))g'(t)h_1$.
- Equivalently, letting $y = g(x)$ and $z = f(y)$:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{5}$$

Recall: Chain Rule of Calculus (2)

- Consider a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, and $g_i : \mathbb{R} \to \mathbb{R}$

$$\frac{\partial f(g_1(x), \ldots, g_n(x))}{\partial x} = \sum_{k=1}^{n} \frac{\partial f(g_1(x), \ldots, g_n(x))}{\partial g_k(x)} \cdot \frac{\partial g_k(x)}{\partial x} \tag{6}$$

- Intuitively: changing x slightly causes each g_k to change slightly. Each of these effects causes a slight change to f. For infinitesimal changes, these effects combine additively.
- Alternatively, $g : \mathbb{R} \to \mathbb{R}^n$ and

$$\frac{\partial f(g(x))}{\partial x} = \sum_{k=1}^{n} \frac{\partial f(g(x))}{\partial g_k(x)} \cdot \frac{\partial g_k(x)}{\partial x} \tag{7}$$

Recall: Chain Rule of Calculus (3)

- Consider $f : \mathbb{R}^n \to \mathbb{R}$, and $g : \mathbb{R}^m \to \mathbb{R}^n$
- In this case, we consider the partial derivatives w.r.t. each x_i

$$\frac{\partial f(g(\boldsymbol{x}))}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial f(g(\boldsymbol{x}))}{\partial g_k(\boldsymbol{x})} \cdot \frac{\partial g_k(\boldsymbol{x})}{\partial x_i} \tag{8}$$

- In vector notation:

$$\nabla_{\boldsymbol{x}} f(g(\boldsymbol{x})) = \left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{g}} f(\boldsymbol{g}(x)) \tag{9}$$

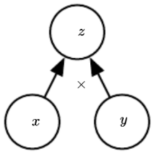
where $\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}}\right)$ is the Jacobian matrix of $g(\boldsymbol{x})$.

- Same rule exploited if we have multi-dimensional tensors (just think that you can flatten the tensor into a vector).

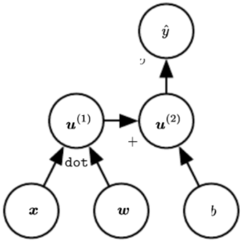
Computational Graph

- It is useful to describe NNs using a more precise formalism, namely computational graphs
- Each node indicates a variable
  - A scalar, a vector, a matrix, or a tensor
- An operation is a function of 1+ variables
  - We assume operations to return a single output variable (which may be a vector though!)
- An edge (x, y) indicates that y is computed by applying an operation to x
  - We may annotate y with the operation

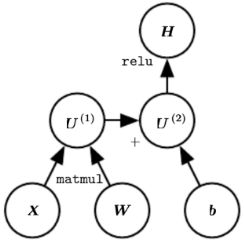
Examples



(a)



(b)



(c)

Example: Chain Rule

$$\begin{tikzcd} w \arrow[r, "f"] & x \arrow[r, "f"] & y \arrow[r, "f"] & z \end{tikzcd}$$

$$z = f(y) = f(f(x)) = f(f(f(w)))$$

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} =$$

$$= f'(y)f'(x)f'(w) =$$

$$= f'(f(f(w)))f'(f(w))f'(w)$$

Applying the rule, we often need to evaluate subexpressions more than once. We either store them or re-compute them

Backpropagation with Scalar Variables

- Consider a computational graph with nodes $u^{(1)}, u^{(2)}, ..., u^{(n)}$, in topological order, all representing scalar variables
- The first $k < n$ variables $u^{(1)}, ..., u^{(k)}$ represent input variables
- $u^{(n)}$ is the output variable
- We want to compute the gradient w.r.t. input

$$\frac{\partial u^{(n)}}{\partial u^{(i)}}, \forall i \in \{1, ..., k\}$$

- Note: when training a NN, we usually are not interested in computing the gradient w.r.t. input, but to parameters

Backpropagation with Scalar Variables (2)

- Each node $u^{(i)}$ is associated with an operation $f^{(i)}$

$$u^{(i)} = f^{(i)}(\mathcal{A}^{(i)}), \quad \mathcal{A}^{(i)} = \{u^{(j)} | j \in Parent(u^{(i)})\}$$

Forward pass (with scalar variables only)

```
for $i = 1, ..., k$ do
    $u^{(i)} \leftarrow x_i$                    /* Input values */
end
for $i = k + 1, ..., n$ do
    $u^{(i)} \leftarrow f^{(i)}(\mathcal{A}^{(i)})$
end
return $u^{(n)}$
```

Backpropagation with Scalar Variables (3)

- We use a lookup table `grad` to store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for every $u^{(i)}$
- For convenience, let $children(j) = \{i | j \in Parent(u^{(i)})\}$

Backward propagation (with scalar variables only)

```
1 Run forward pass (previous slide) to evaluate variables
2 Initialize empty lookup table grad
3 $\texttt{grad}[u^{(n)}] \leftarrow 1$
4 for $j$=$n$-1,$n$-2,...,1 do
      /* Apply chain rule                                  */
5     $\texttt{grad}[u^{(j)}] \leftarrow \sum_{i \in children(j)} \texttt{grad}[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
6 end
7 return $\{grad[u^{(i)}], i = 1, \dots, k\}$
```

Backpropagation for NNs

- We now focus on the specific case of backpropagation for multi-layer neural networks
- To keep things simple, we assume that a single training instance $\boldsymbol{x}$ and the associated label $\boldsymbol{t}$ are used to compute the cost $J(\boldsymbol{\theta})$
- We aim to compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

Forward Pass

Forward pass

```
1 $\boldsymbol{h}^{(0)} \leftarrow \boldsymbol{x}$
2 for $k$=1,..,$L$ do
3     $\boldsymbol{a}^{(k)} \leftarrow \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}$
4     $\boldsymbol{h}^{(k)} \leftarrow f^{(k)}(\boldsymbol{a}^{(k)})$
5 end
6 $\boldsymbol{y} \leftarrow \boldsymbol{h}^{(L)}$
7 return $J = L(\boldsymbol{y}, \boldsymbol{t}) + \lambda\Omega(\boldsymbol{\theta})$
```

Backward Pass

Backpropagation for multi-layer NNs

```
1 $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{y}} J = \nabla_{\boldsymbol{y}} L(\boldsymbol{y}, \boldsymbol{t})$
2 for $k$=$L$,..,$2$,$1$ do
3     $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f^{(k)\prime}(\boldsymbol{a}^{(k)})$
4     $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
5     $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
      /* Propagate the gradients                    */
6     $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
7 end
8 return $\nabla_{\boldsymbol{W}^{(k)}} J, \nabla_{\boldsymbol{b}^{(k)}} J, \forall k = 1, \ldots, L$
```

Backward Pass (2)

```
1 $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{y}} J = \nabla_{\boldsymbol{y}} \mathcal{L}(\boldsymbol{y}, \boldsymbol{t})$
```

- $\boldsymbol{g}$ is initialized as the gradient of the cost function with respect to the output vector $\boldsymbol{y}$
- The cost function may include a regularization term, which does not depend on the output
- We need just differentiate the loss function
- This step can be performed analytically and depends on the loss function in use
  - e.g., if $\mathcal{L} = \frac{1}{2}\| \boldsymbol{y} - \boldsymbol{t} \|_2^2$, we get $\nabla_{\boldsymbol{y}} \mathcal{L} = (\boldsymbol{y} - \boldsymbol{t})$

Backward Pass (3)

```
2 for $k$=$L$,..,$2$,$1$ do
3     $g \leftarrow \nabla_{a^{(k)}} J = g \odot f^{(k)\prime}(a^{(k)})$
4     ...
```

- We iterate backward from the L-the layer to the first hidden layer
- At the beginning of each iteration, g stores the gradient of the cost with respect to the k-th layer
- e.g., at first iteration, g is the gradient w.r.t. the output layer

Backward Pass (4)

```
2  for $k=L,..,2,1$ do
3      $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f^{(k)\prime}(\boldsymbol{a}^{(k)})$
```

- We compute the gradient w.r.t. the pre-activation value
- Being $f^{(k)}$ the activation function of the k-th layer:

$$\boldsymbol{h}^{(k)} = f^{(k)}(\boldsymbol{a}^{(k)})$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(k)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{a}^{(k)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(k)}} \odot f^{(k)\prime}(\boldsymbol{a}^{(k)}) = \boldsymbol{g} \odot f^{(k)\prime}(\boldsymbol{a}^{(k)})$$

- Note: $\odot$ denotes the element-wise product

Remark: Softmax

```
2 for k=L,..,2,1 do
3     $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f^{(k)\prime}(\boldsymbol{a}^{(k)})$
```

- The simple expression of line 3 above is not correct if $f^{(k)}$ is the softmax
- In that case, the output of the layer depends on **all** the pre-activation variables. Therefore, the Jacobian must be used

$$\boldsymbol{g} \leftarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(k)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{a}^{(k)}} = \left(\frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{a}^{(k)}} \right)^T \cdot \boldsymbol{g}$$

Backward Pass (5)

4 $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$

5 $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$

- Recall: $\boldsymbol{g}$ is the gradient w.r.t. $\boldsymbol{a}^{(k)} = \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}$
- We now take into account the regularization term as well:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(k)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(k)}} + \lambda \frac{\partial \Omega(\boldsymbol{\theta})}{\partial \boldsymbol{W}^{(k)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(k)}} \frac{\partial \boldsymbol{a}^{(k)}}{\partial \boldsymbol{W}^{(k)}} + \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta}) =$$

$$= \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$$

- The same reasoning holds for $\boldsymbol{b}^{(k)}$

Backward Pass (6)

```
6 $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T}\boldsymbol{g}$
```

- We use $\boldsymbol{g}$ to back-propagate the gradient
- At the next iteration $\boldsymbol{g}$ must hold the gradient w.r.t. $\boldsymbol{h}^{(k-1)}$

$$\boldsymbol{a}^{(k)} = \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}$$

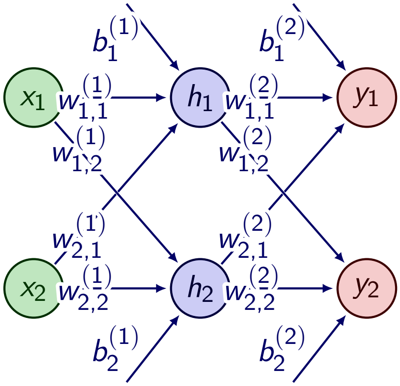
$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(k-1)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(k)}} \frac{\partial \boldsymbol{a}^{(k)}}{\partial \boldsymbol{h}^{(k-1)}} = \boldsymbol{W}^{(k)T}\boldsymbol{g}$$

Derivative of Activation Functions: Examples

	$f(x)$	$f'(x)$
Logistic (sigmoid)	$\frac{1}{1+e^{-x}}$	$f(x)(1-f(x))$
ReLU	$\max\{0, x\}$	$\begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$
Hyp. tan	$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$1-f(x)^2$

Exercise

Given the following NN specification and the input vector $\boldsymbol{x} = \{0.5, 0.1\}$, apply backprop algorithm



$$W^{(1)} = \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix}$$

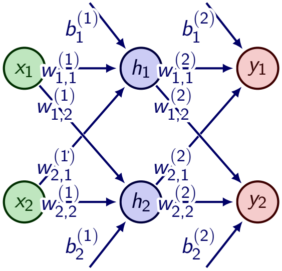
$$W^{(2)} = \begin{bmatrix} 0.4 & 0.5 \\ 0.5 & 0.55 \end{bmatrix}$$

$$\boldsymbol{b}^{(1)} = [0.35, 0.35]$$
$$\boldsymbol{b}^{(2)} = [0.6, 0.6]$$
$$\boldsymbol{t} = [0.01, 0.99]$$
$$\mathcal{L} = \frac{1}{2} \parallel \boldsymbol{y} - \boldsymbol{t} \parallel_2^2$$

All the units use the logistic activation.

ex_backprop.ipynb (solution)

Exercise: Solution (partial)

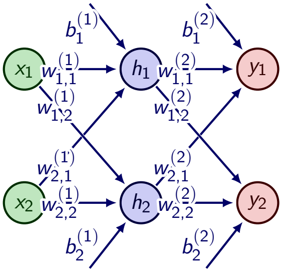


$$
\begin{aligned}
\boldsymbol{a}^{(1)} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} = \\
&= \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \\
&= \begin{bmatrix} 0.095 \\ 0.155 \end{bmatrix} + \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \begin{bmatrix} 0.445 \\ 0.505 \end{bmatrix}
\end{aligned}
$$

$$
\boldsymbol{h}^{(1)} = \sigma(\boldsymbol{a}^{(1)}) = \begin{bmatrix} 0.6094 \\ 0.6236 \end{bmatrix}
$$

Note: numerical results may be inaccurate due to approximations...check the notebook for correct results!

Exercise: Solution (partial)



$$\boldsymbol{a}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)} =$$

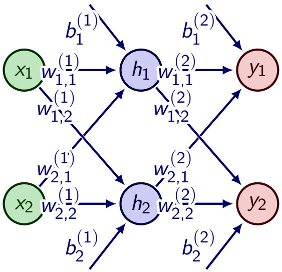
$$= \begin{bmatrix} 0.4 & 0.5 \\ 0.5 & 0.55 \end{bmatrix} \cdot \begin{bmatrix} 0.6094 \\ 0.6236 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.5244 \\ 0.6478 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 1.1244 \\ 1.2478 \end{bmatrix}$$

$$\boldsymbol{y} = \boldsymbol{h}^{(2)} = \sigma(\boldsymbol{a}^{(2)}) = \begin{bmatrix} 0.7548 \\ 0.7770 \end{bmatrix}$$

$$\mathcal{L} = \frac{1}{2} \parallel \boldsymbol{y} - \boldsymbol{t} \parallel_2^2 = \frac{1}{2} \left\| \begin{matrix} 0.7448 \\ -0.213 \end{matrix} \right\|_2^2 = 0.3$$

Exercise: Solution (partial)



$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{y}}\mathcal{L} = \boldsymbol{y} - \boldsymbol{t} = \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix}$$

k=2

$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(2)}}\mathcal{L} = \boldsymbol{g} \odot \sigma\prime(\boldsymbol{a}^{(2)}) =$$

$$= \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix} \odot \begin{bmatrix} \sigma(1.1244)(1-\sigma(1.1244)) \\ \sigma(1.2478)(1-\sigma(1.2478) \end{bmatrix} =$$

$$= \begin{bmatrix} 0.7448 \\ -0.213 \end{bmatrix} \odot \begin{bmatrix} 0.1851 \\ 0.1733 \end{bmatrix} = \begin{bmatrix} 0.1378 \\ -0.0369 \end{bmatrix}$$

$$\nabla_{\boldsymbol{W}^{(2)}}\mathcal{L} = \boldsymbol{g}\boldsymbol{h}^{(1)\,T} = \begin{bmatrix} 0.1378 \\ -0.0369 \end{bmatrix} \cdot \begin{bmatrix} 0.6094 & 0.6236 \end{bmatrix} =$$

$$\begin{bmatrix} 0.0840 & 0.0860 \\ -0.0225 & -0.0230 \end{bmatrix}$$

Computational Demand

- Let's consider a generic computational graph
- Assume each operation to have roughly the same cost (actually, an operation may be a matrix multiplication!)
  - We can reason in terms of operations
- A naive application of the chain rule would have an exponential runtime because of repeated evaluation of subexpressions
- With n nodes, the forward pass of backprop executes $\mathcal{O}(n)$ operations
- Backward pass adds an operation for every edge in the original graph: $\mathcal{O}(n^2)$

Computational Demand for NNs

- For the specific case of feedforward fully-connected NNs, we can be more precise
- Consider a NN with L hidden layers, each with m neurons
- We drop the assumption of atomic operations (i.e., a matrix multiplication is not $\mathcal{O}(1)$)
- Forward pass: an add-and-multiply operation for every weight: $\mathcal{O}(Lm^2)$
- Backward pass: more operations to perform, but still: $\mathcal{O}(Lm^2)$
- Note: this is just one step in the whole training process!

Generalizing Backpropagation: Input Batches

- So far, we considered the gradient of the cost function computed on a single input instance x
- In practice, training is more efficient if minibatches of input are considered, because computational parallelism can be exploited (e.g., vectorization)
- The same algorithm can be applied: just replace x with a matrix $\boldsymbol{X}$, where each row is a different training instance
- Operations involving gradients do not change: just imagine to “flatten” matrices (or tensors) into a vector

Training with Minibatch SGD

- Recall Minibatch SGD from the first part of the course

NN training via Minibatch SGD

```
while stopping criterion not met do
    Sample minibatch $\mathcal{B}$ randomly from the training set
    $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J^{\mathcal{B}}(\boldsymbol{\theta})$        /* Compute the gradient */
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{g}$
end
```

Minibatch Size

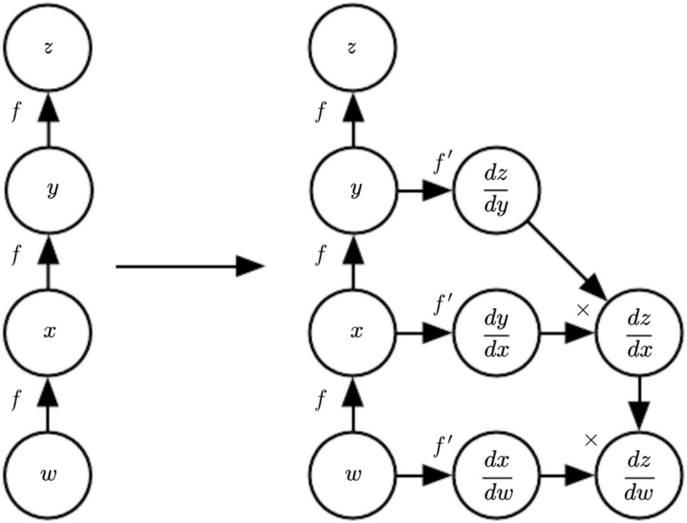
How to choose the size of the minibatch?

- Larger batches generally make training faster, but smaller batches favor convergence
- Batch size is usually a power of 2 (the reason should be obvious...)
- Typical values are, e.g., 8, 16, 32, 64
- The choice depends on the hardware in use
- Tip: keep it fixed while optimizing other hyperparameters, and optimize it at the end (we will come back to this issue later!)

Symbol-to-symbol Differentiation

- Computational graphs (as well as algebraic expressions) are defined over symbols (i.e., variables that do not have a specific value)
- An approach to backpropagation implementation: given a set of numerical values for the inputs in the graph, return the gradient at those input values
  - Called symbol-to-number differentiation
  - Implemented in Torch and Caffe
- Alternative approach (symbol-to-symbol): extend the original graph with new nodes that provide a symbolic description of the desired derivatives
  - Implemented in Theano and TensorFlow

Example: Symbol-to-symbol Differentiation



General Backprop and Custom Operations

- Backpropagation can be implemented in a way that enable easy integration of new operations on vectors, matrices and tensors
- The algorithm does not require a specification of which operations are performed, nor how to compute their derivatives
- It only needs a suitable `op` interface, providing a `backprop()` method to backpropagate the gradient
- Deep learning libraries provide built-in implementations of most used operations, but also support custom operations
- More details: Goodfellow–Bengio–Courville Sec. 6.5.6

Example with scikit-learn

- scikit-learn provides `MLPClassifier` and `MLPRegressor`, based on multi-layer feedforward NNs
- Very easy to use, but not recommended to work with NNs
  - No GPU support
  - Less flexibility compared to other frameworks (e.g., TensorFlow)
- We see a simple classification example on the Iris data set

mlpclassifier.ipynb

Exercise

Spend some time on

`http://playground.tensorflow.org`

3/10/2023

Introduction to Keras and TensorFlow

TensorFlow

- Library for ML focused on DNNs
- Developed by Google Brain team
  - Many scientists and developers involved, including Geoffrey Hinton (2018 Turing Award winner)
- First released in 2015; major update (TensorFlow 2) in 2019
- Built-in support for GPU execution, as well as distributed execution

TensorFlow

Installing TensorFlow

- Main version available for Linux; Windows; macOS (no GPU)
  - tensorflow.js for browsers
  - TensorFlow Lite for embedded/mobile devices
- A command should be enough:

```
pip install tensorflow
```

- If you use Google Colab, it is already installed

Keras

Keras semplifica la definizione di modelli ed architettura rete, ma non offre training per la rete, per questo ha bisogno di un backend (TensorFlow)

- Keras is an open-source library, first released in 2015, that provides a Python API for ANNs
- The goal is easing the definition of deep models
- Keras requires a backend for actual execution
  - TensorFlow, PyTorch, JAX, …
- Since 2017, TensorFlow has integrated its own implementation of Keras (`tf.keras`)
  - we will use this Keras implementation
  - no need to install it explicitly

Quindi installando TensorFlow, avremo incluso anche una versione di Keras (tf.Keras)

TensorFlow APIs

Abbiamo tre diverse interfacce per definire modelli:
Le prime due vengano da Keras, e useremo quasi esclusivamente loro.
La Core Api è di più basso livello, e la uso se devo introdurre qualcosa di particolare, quindi in generale sarà poco usata.

- As part of `tf.keras`:
  - Sequential API: easiest to use; OK for most apps
  - Functional API: more flexibility
- Core API: only for advanced stuff

- We will mostly use the Sequential API

Basics

- Let's start with the basics of TensorFlow
  - Tensors
  - Variables
  - Graphs
  - Automatic differentiation
  - Modules

è un array multidimensionale (tipo numpy). Se voglio inserire dei numeri ho prefisso "constant". "shape" = size, dtype = tipo

se metto una variabile in una funzione, tensorflow calcola automaticamente la derivata. Ciò porta alla creazione di un graph.

Modules: se spengo server, non voglio perdere dati, allora salvo il modulo, cioè tutte le variabili, pesi etc... che posso ricaricare quando voglio.

tf_basics.ipynb

abbiamo anche tf.nn.softmax, applicabile ai livelli di uscita.

Il tensore è IMMUTABILE, ovvero, ad ogni operazione viene creato un nuovo tensore, quindi ciò che posso fare è aggiornare la variabile associata al tensore.
Oppure usare una "Variable", tensore mutabile.

Keras

con @tf.function si ottimizza la funzione, creando un grafo delle operazioni che verrà richiamato le volte successive per velocizzare il codice, usando tale grafo. Se passo da interi a virgola mobili, il grafo viene ricompilato. Queste operazioni le vediamo poco, lavoriamo a livello più alto.

- 2 core data structures: layers and models
- Layer: simple input/output transformation
  - encapsulates a state (weights) and some computation
  - may represent a layer of a DNN, but also a data preprocessing step
- Model: directed acyclic graph (DAG) of layers
  - e.g., Sequential model: a linear stack of layers
  - `fit()` and `predict()` methods (similar to scikit-learn)

(no neuroni, è scatola nera in cui metto input e produce output)

grado diretto aciclico di livelli, quindi grafo dei livelli.

tf_keras.ipynb

"Dense", indica livello fully connected. (vedi foto)

tali metodi sono quelli offerti dagli stimatori in scikit learn, addestramento (con loop di addestramento visto prima) e predizione

NNs for Regression: Hints 7/11/2023

come impostare hyperparametri modello, vediamo REGOLE GENERALI, o LINEE GUIDA.
Se lavoro in 3 dimensioni -> 3 neuroni. Se parlo di automobili, metto un neurone per ogni feature (cavalli, posti, anno...). Il feature engineering va fatto anche con reti neurali, anche se abbiamo visto che alcune cose "le fanno da sole". Se posso applicarne qualcuna, perchè sono facili ad es, meglio farlo.

Hyperparameter	Possible/typical values
# input neurons	1 per input feature
# hidden layers	?? (1-5 for many tasks)
# hidden units per layer	?? (10-100 for most tasks)
# output units	1 per predicted dimension
Hidden activation	ReLU (quantomeno per iniziare uso lei)
Output activation	None; ReLU for positive outputs; tanh for bounded outputs
Loss	MSE or MAE MAE = errore medio assoluto MSE = pesa di piu se sbaglio di tanto

Quanti livelli nascosti? da 1 a 5 sono normalmente sufficienti, dipende dalla complessità.
Ciò si lega anche alle unità nel livello nascosto. Se ho 1000 feature, non sembra sensato mettere solo 10 unità per livello. Spesso si va a tentativi, detto anche "hyperparameter optimization".

NNs for Classification: Hints

Hyperparameter	**Binary**	**Multiclass**
Input and hidden layers	*Same as regression*	
# output units	1	1 per class
Output activation	Sigmoid	Softmax
Loss	Cross entropy	Cross entropy

Sigmoid per Binary, perchè output compreso tra 0 e 1, ok per probabilità.
Se multiclasse, usiamo la softmax, prende i valori pre attivazione e produce vettore in uscita tra 0 e 1.

Remark: Cross-Entropy Loss

- With 2 classes, in Keras you have `BinaryCrossentropy`

t è 0 o 1, solo un termine sarà diverso da 0. Log(y) è la probabilità, se y=1 loss è 0 (ln(1)=0).
Altrimenti piu mi allontano
piu il logaritmo diventa piccolo,
mettiamo il meno davanti e allora cresce.

$$\mathcal{L} = -\left(t \log y + (1-t)\log(1-y)\right)$$

- With $C > 2$ clssses and, hence, $\mathrm{y} = (y_1, \ldots, y_C)$, the loss is:

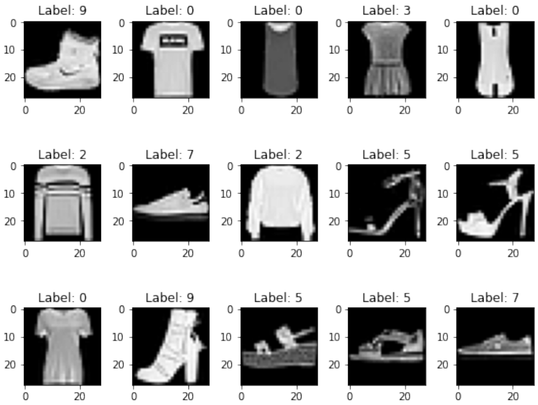
solo un termine sarà diverso da 0

$$\mathcal{L} = -\sum_{c=0}^{C} t_c \log y_c$$

- `CategoricalCrossentropy` if targets are given as one-hot vectors
- `SparseCategoricalCrossentropy` if targets are integers between 0 and $(C-1)$

Example: Fashion MNIST

- We consider the Fashion MNIST dataset
- 60,000 grayscale images of 28×28 pixels each to classify, with 10 classes



tf_fashion.ipynb

Tuning Hyperparameters

- Flexibility of NNs is also one of their drawbacks: many hyperparameters to tweak!
  - Number of layers, units
  - Optimization algorithm (and its hyperparameters!)
  - …
- Exhaustive explorations (i.e., grid search) usually not admissible
- You can perform a randomized search using scikit-learn facilities

  tf_fashion_hyper.ipynb
- Specialized libraries for hyperparameter tuning, e.g., Hyperopt. Talos, Spearmint, …

References

- Goodfellow et al.: 6.5, 7.1, 7.4, 7.8, 7.11, 7.12, 8.1–8.5, 8.7.1
- D2L: 5.3–5.6, 8.5, 12
- Hands-on ML: Chapters 10-11