Machine Learning

Neural Networks and Deep Learning: Training

Gabriele Russo Russo   Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2024/25

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Training Neural Networks

- In principle, training a NN is not different than training other parametric models you studied so far
  - e.g., linear or logistic regression
  - in practice, you might face more challenges
- We search for model parameters $\boldsymbol{\theta}$ that optimize a cost function $J(\boldsymbol{\theta})$
  - e.g., via stochastic gradient descent (SGD)
- In this case, $\boldsymbol{\theta}$ includes:
  - the connection weights, $\boldsymbol{W}^{(\ell)}, \forall 1 \leq \ell \leq L$
  - the bias vectors, $\boldsymbol{b}^{(\ell)}, \forall 1 \leq \ell \leq L$

Cost Function: Example

Which is the cost/objective function to optimize?

- Similar to those you have seen so far
- e.g., for a regression task, with 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 function $J(\boldsymbol{\theta})$

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \frac{1}{2}(t^{(i)} - y^{(i)})^2 \tag{2}$$

Training with SGD

Can we just keep using SGD for NNs?

Yes …but better algorithms exist (discussed later)

NN training via SGD (simplified)

Data: training dataset $\mathcal{D}$

for *epoch* $\leftarrow 1, \ldots, N_e$ **do**

  foreach *random minibatch* $\mathcal{B} \in \mathcal{D}$ **do**

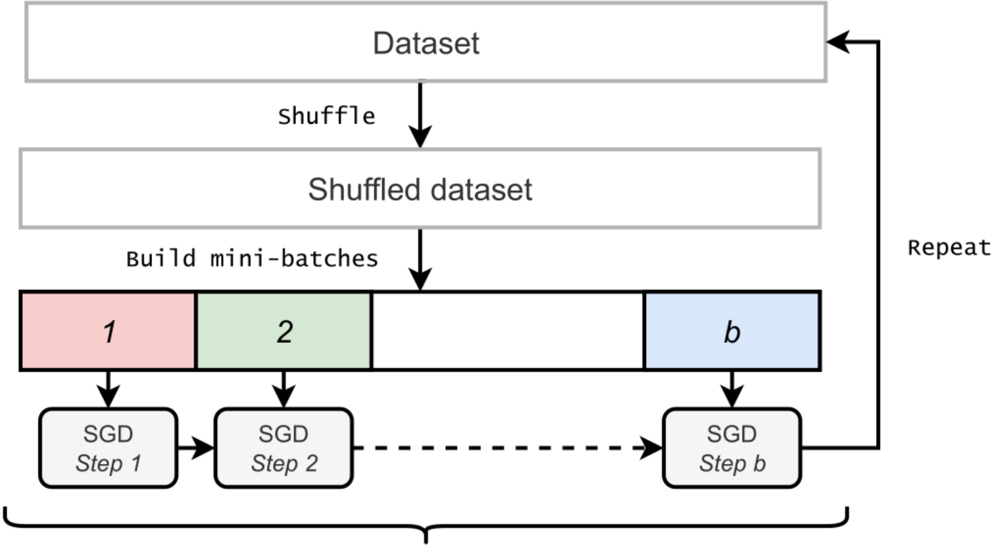
    $\hat{\mathbf{g}} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J^{\mathcal{B}}(\boldsymbol{\theta})$ `/* Compute the gradient */`

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \hat{\mathbf{g}}$

  end

end

Training with SGD (2)



Training with SGD (3)

- How many epochs?
  - It is a hyperparameter
  - Tens of epochs are usually required for convergence (possibly many more!)
  - We will discuss a strategy to automatically stop the training based on a convergence criterion (early stopping)
- How to choose the size of the minibatch?
  - Larger batches may enable processing speedup (e.g., exploiting hardware parallelism); smaller batches may be better for convergence
  - Ideal size depend on hardware & implementation
  - Usually a power of 2 (e.g., 16, 32, 64)

Example

Spend some time on `http://playground.tensorflow.org`

Training: Issues

Unfortunately, training NNs is more challenging compared to basic models

- NNs lead to nonconvex cost functions
  - We can use gradient-based methods to drive the cost function to a low value
  - But we have no global convergence guarantees!
  - Still, many local minima empirically shown to have acceptable quality
- How to efficiently compute the gradient?
  - Cost is a composite function of the weights of all the layers
  - Evaluation can be computationally expensive

Example: Gradient Computation

- We have seen that the output of a NN (and, hence, the cost function) is the result of composing several functions
- Let's consider a very simple univariate composite function $f(x)$

$$f(x) = \sqrt{x^2 + e^{x^2}} + \cos\left(x^2 + e^{x^2}\right)$$

- How to compute $\frac{\mathrm{d}f}{\mathrm{d}x}$?

Recall: Chain Rule of Calculus

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

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x) \tag{3}$$

- Intuition: perturbing 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 $f'(g(x))h_2 = f'(g(x))g'(x)h_1$.
- Equivalently:

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

Example: Gradient Computation (2)

$$f(x) = \sqrt{x^2 + e^{x^2}} + \cos\left(x^2 + e^{x^2}\right)$$

- How to compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ through the chain rule?

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{2x + 2xe^{x^2}}{2\sqrt{x^2 + e^{x^2}}} - \sin\left(x^2 + e^{x^2}\right)\left(2x + 2xe^{x^2}\right)$$

- Note that we end up with a longer expression than f (and more expensive to evaluate)
- We must be careful (and smart) in implementing the chain rule

Automatic Differentiation

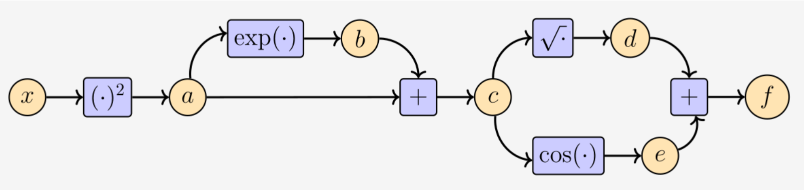
- Automatic differentiation (AD): techniques to numerically evaluate the gradient of a function through the chain rule
- Applied to general computer programs implementing (complicated) functions
- Programs formally represented as computational graphs before applying AD
  - Nodes indicate variables
  - Edges indicate operations among variables

Example

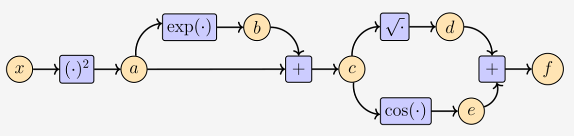
$$f(x) = \sqrt{x^2 + exp(x^2)} + \cos\left(x^2 + exp(x^2)\right)$$

- It is useful to introduce variables

$$a = x^2 \qquad b = exp(a) \qquad c = a + b$$
$$d = \sqrt{c} \qquad e = \cos c \qquad f = d + e$$



Example (2)



$$\frac{\mathrm{d}f}{\mathrm{d}c} = \frac{\mathrm{d}f}{\mathrm{d}d}\frac{\mathrm{d}d}{\mathrm{d}c} + \frac{\mathrm{d}f}{\mathrm{d}e}\frac{\mathrm{d}e}{\mathrm{d}c}$$

$$\frac{\mathrm{d}f}{\mathrm{d}b} = \frac{\mathrm{d}f}{\mathrm{d}c}\frac{\mathrm{d}c}{\mathrm{d}b}$$

$$\frac{\mathrm{d}f}{\mathrm{d}a} = \frac{\mathrm{d}f}{\mathrm{d}b}\frac{\mathrm{d}b}{\mathrm{d}a} + \frac{\mathrm{d}f}{\mathrm{d}c}\frac{\mathrm{d}c}{\mathrm{d}a}$$

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}a}\frac{\mathrm{d}a}{\mathrm{d}x}$$

- Green indicates derivatives of elementary functions (easy)
- We apply a reverse-mode AD algorithm: we propagate gradients traversing the graph backwards (from f towards a)
- No redundant computations!

Backpropagation

- Backpropagation (or, backprop for short) is a reverse-mode automatic differentiation algorithm, mostly used to compute the gradient of the NN cost function
  - 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 pass: A minibatch of training instances 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 pass: 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).

Forward Propagation

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

e.g., for a NN with a single hidden layer, forward prop. computes:

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

Forward Pass

- To keep things simple, we assume that a single training instance x is used to compute the cost $J(\boldsymbol{\theta})$ (we will generalize to minibatches later)

Forward pass

$\boldsymbol{h}^{(0)} \leftarrow \boldsymbol{x}$

for k=1,..,L **do**

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

  $\boldsymbol{h}^{(k)} \leftarrow \phi^{(k)}(\boldsymbol{a}^{(k)})$

end

$\boldsymbol{y} \leftarrow \boldsymbol{h}^{(L)}$

return $J = L(\boldsymbol{y}, \boldsymbol{t})$

Backward Pass

Backpropagation for multi-layer NNs

```
g ← ∇_y J = ∇_y L(y, t)
for k=L,...,2,1 do
    g ← ∇_{a^(k)} J = g ⊙ φ^(k)'(a^(k))
    ∇_{b^(k)} J = g
    ∇_{W^(k)} J = g h^(k-1)T
    g ← ∇_{h^(k-1)} J = W^(k)T g          /* Propagate gradient */
end
return ∇_{W^(k)} J, ∇_{b^(k)} J, ∀k = 1, ..., L
```

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

for k=L,..,2,1 **do**

$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot \phi^{(k)\prime}(\boldsymbol{a}^{(k)})$

$\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$

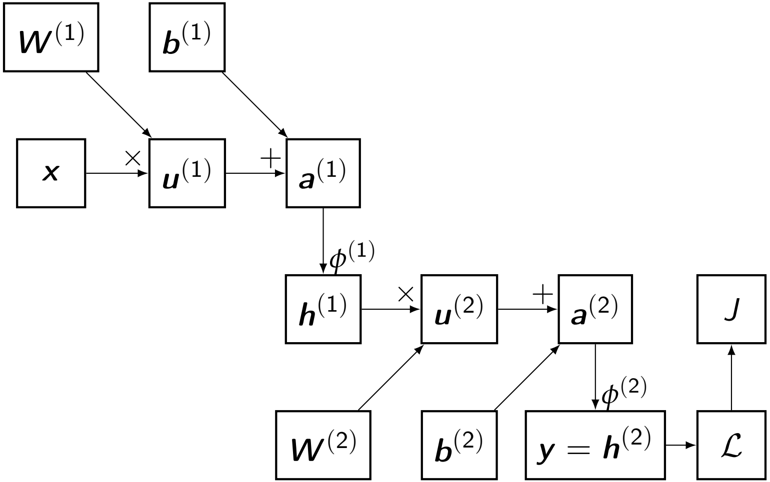
$\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$

$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T}\boldsymbol{g}$ /* Propagate gradient */

end

return $\nabla_{\boldsymbol{W}^{(k)}} J, \nabla_{\boldsymbol{b}^{(k)}} J, \forall k = 1, \ldots, L$

Example

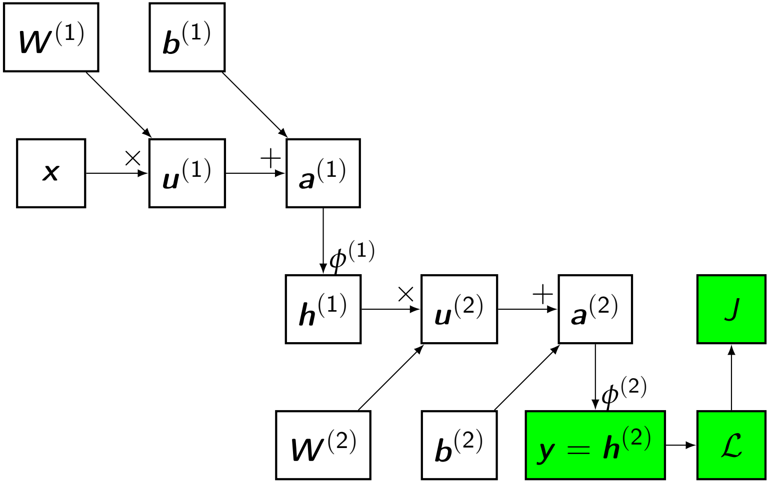


Backward Pass (2)

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

- $\boldsymbol{g}$ is initialized as the gradient of the cost function with respect to the output vector $\boldsymbol{y}$ (easy to compute)
- 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 $\nabla_{\boldsymbol{y}} \mathcal{L} = (\boldsymbol{y} - \boldsymbol{t})$
- We will see that J may include a regularization term, which does not depend on the output though

Example



Backward Pass (3)

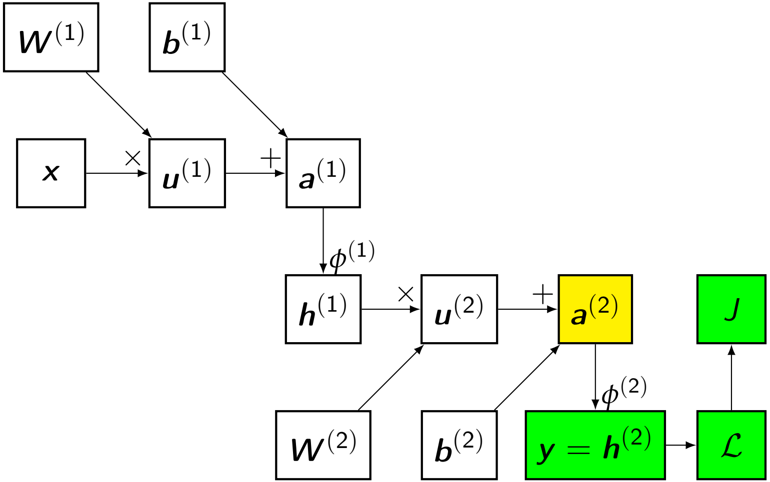
for k=L,..,2,1 **do**

$\quad g \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = g \odot f^{(k)\prime}(\boldsymbol{a}^{(k)})$

$\quad$...

- We iterate backwards 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

Example



Backward Pass (4)

for k=L,..,2,1 **do**

  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot \phi^{(k)\prime}(\boldsymbol{a}^{(k)})$

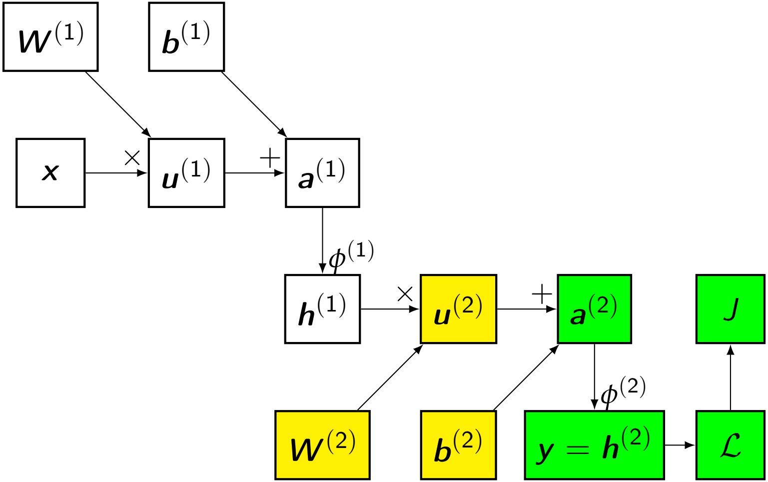
- We compute the gradient w.r.t. the pre-activation value
- Being $\phi^{(k)}$ the activation function of the k-th layer:

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

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

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

Example



Backward Pass (5)

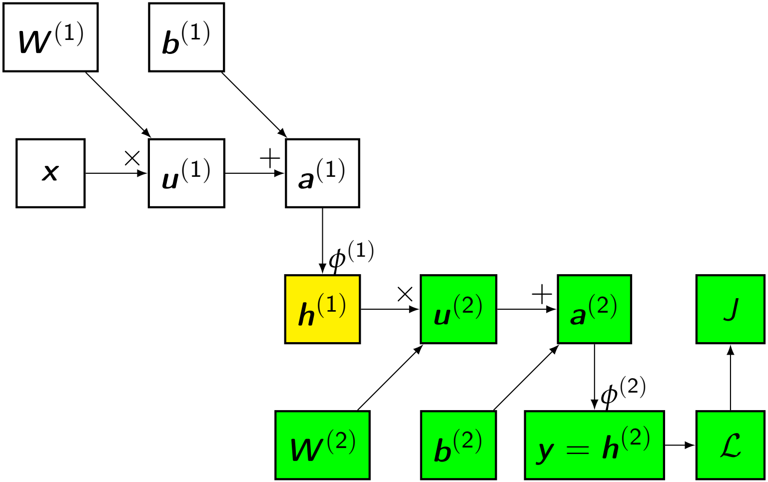
4 $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
5 $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$

- Recall: $\boldsymbol{g}$ is the gradient w.r.t. $\boldsymbol{a}^{(k)} = \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)}$

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

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

Example



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

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

Remark: Softmax

2 **for** *k=L,..,2,1* **do**

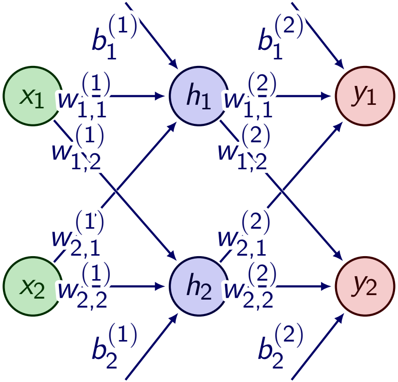
3 $\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot \phi^{(k)\prime}(\boldsymbol{a}^{(k)})$

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

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

Exercise

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



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

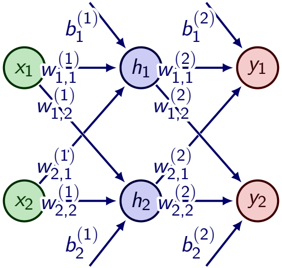
$$\boldsymbol{W}^{(2)} = \begin{bmatrix} 0.4 & 0.5 \\ 0.5 & 0.55 \end{bmatrix}$$

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

All the units use the logistic activation.

ex_backprop.ipynb (solution)

Exercise: Solution (partial)

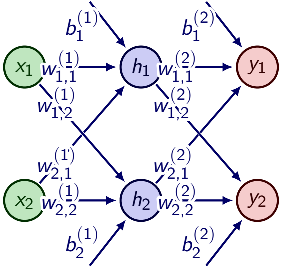


$$
\begin{aligned}
\boldsymbol{a}^{(1)} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} = \\
&= \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \\
&= \begin{bmatrix} 0.095 \\ 0.155 \end{bmatrix}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

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

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

Exercise: Solution (partial)

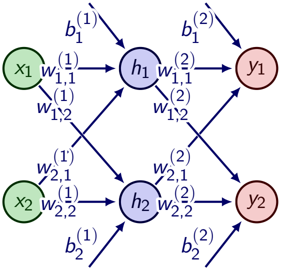


$$
\begin{aligned}
\boldsymbol{a}^{(2)} &= \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)} = \\
&= \begin{bmatrix} 0.4 & 0.5 \\ 0.5 & 0.55 \end{bmatrix} \cdot \begin{bmatrix} 0.6094 \\ 0.6236 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \\
&= \begin{bmatrix} 0.5244 \\ 0.6478 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 1.1244 \\ 1.2478 \end{bmatrix}
\end{aligned}
$$

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

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

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

- Consider a NN with L hidden layers, each with m neurons
- Forward pass: an add-and-multiply operation for every weight: $\mathcal{O}(Lm^2)$
- Backward pass: more operations to perform, but still: $\mathcal{O}(Lm^2)$
- As desired, the cost of gradient evaluation is the same of the forward pass
- But, keep in mind that this is just a tiny step in the whole training process!

Generalizing Backpropagation: Minibatch

- 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

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

Introduction to Keras and TensorFlow

TensorFlow

- Library for ML focused on DNNs
- Developed by Google Brain team
  - Many scientists and developers involved, including (in the past) 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

Installing TensorFlow

- Main version available for Linux; Windows; macOS
- Additional versions (for inference):
  - tensorflow.js for browsers
  - TensorFlow Lite for embedded/mobile devices
- To install:

```
conda install tensorflow
```

Alternatively: to explicitly include/exclude GPU support:

```
conda install tensorflow-cpu
conda install tensorflow-gpu
```

- If you use Google Colab, it is already installed

Keras

- 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

TensorFlow APIs

- 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

tf_basics.ipynb

Keras

- 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)

tf_keras.ipynb

NNs for Regression: Hints

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
Output activation	None; ReLU for positive outputs; tanh for bounded outputs
Loss	MSE or MAE

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

Remark: Cross-Entropy Loss

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

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

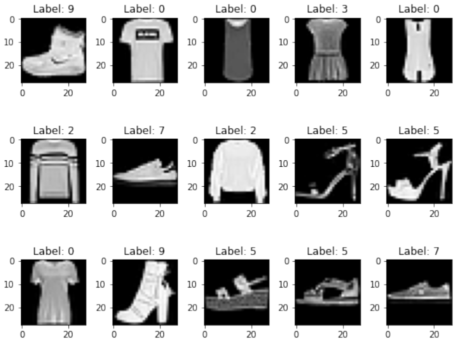
- With $C > 2$ clssses and, hence, $\mathrm{y} = (y_1, \ldots, y_C)$, the loss is:

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

- `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

Hyperparameter Optimization (HPO)

- Flexibility of NNs is also one of their drawbacks: many hyperparameters to tweak!
  - Number of layers, units, activation functions, ...
- General approach: try many configurations and pick the best one (evaluated on **validation data**!)
- Exhaustive exploration (i.e., grid search) usually not admissible
- We have seen "randomized search" in action in scikit-learn
- Specialized libraries for hyperparameter tuning
  - Hyperopt, Ray Tune, Optuna, ...
  - They integrate with the most popular DNN frameworks (e.g., Tensorflow)

KerasTuner

- Keras provides KerasTuner for HPO
- Not advanced as other libraries (e.g., Optuna), but still a good option
- Different algorithms available, including:
  - Random Search
  - Bayesian Optimization
  - Hyperband
- `conda install keras-tuner`
- Docs: `https://keras.io/guides/keras_tuner/getting_started/`

tf_fashion_hyper.ipynb

References

- "Understanding Deep Learning", 6.1, 6.2, 7.1–7.4
- D2L: 5.3–5.6, 8.5, 12
- Goodfellow et al.: 6.5, 7.1, 7.4, 7.8, 7.11, 7.12, 8.1–8.5, 8.7.1
- Hands-on ML: Chapters 10-11