Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 2: Supervised learning (for vectors)

Lecturer: S. Scardapane

Sapienza Università di Roma

Supervised learning

Setup and examples

A (supervised) dataset is a set of n **examples**:

$$\mathcal{S} = \{(x_1, y_1), \ldots, (x_n, y_n)\} \ . \tag{1}$$

Informally, given a 'new' pair (x, y) not contained in $\mathcal{S}$, we want a function $f(\cdot)$ such that:

$$f(x) \approx y \, . \tag{2}$$

More generally, we test the model on a separate dataset $\mathcal{T}$ never seen during training, i.e., $\mathcal{S} \bigcap \mathcal{T} = \emptyset$.

We always assume implicitly that the elements in $\mathcal{S}$ and the elements in $\mathcal{T}$ are taken from *the same* i.i.d, *unknown* distribution $p(x, y)$.

- **Identically distributed**: the data-generating process is stable (e.g., cats in an apartments).
- **Independently distributed**: there is no bias in the data collection (e.g., only siamese cats).

If the distribution between $\mathcal{S}$ and $\mathcal{T}$ varies, we talk about **domain shift**.

1. **Spam identification**: x_i is an email, and y_i describes its probability of being spam.
2. **Robot navigation**: x_i is a sensory representation of the environment, and y_i is a motor command.
3. **Text translation**: x_i is a text and y_i its corresponding translation.
4. **Product recommendation**: x_i is a user, and y_i its affinity w.r.t. a certain catalogue of products.

Note: ensuring the i.i.d. property sometimes is far from trivial!

Supervised learning

Loss functions

What is a good approximation?

"Il gatto è sul tavolo."

f(x)

The cat is on the table.

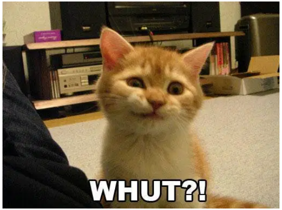
Good translation

A cat sitted in table.

Still understandable

Table sitting cat cats.

Uh?



Given a point x, a desired value y, and a prediction $\hat{y} = f(x)$, we formalize its quality with a **loss function** $l(y, \hat{y})$, such that:

1. Low value of loss: good approximation;
2. High value of loss: poor approximation.

In this way, learning becomes a problem of minimizing a certain loss quantity that we designed.

The **expected loss** (risk) of a function f is:

$$f^*(x) = \arg\min \left\{ \mathbb{E}_{p(x,y)} \left[l(y, f(x)) \right] \right\} . \tag{3}$$

The expected risk is uncomputable, but can be approximated via **empirical risk minimization**:

$$f^*(x) = \arg\min_f \left\{ \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)) \right\} . \tag{4}$$

The gap between the two approaches is called **generalization gap**.

To begin our exploration of supervised learning, we will make a few simplifying assumptions:

- The input $\underset{(d)}{\mathbf{x}}$ is a vector of shape d.
- The output $y \in \mathbb{R}$ is a single real number.

In this case, a common loss is the squared norm:

$$l(y, \hat{y}) = (y - \hat{y})^2 \ . \tag{5}$$

Alternative losses are the absolute value $|y - \hat{y}|$ and the Huber loss.

Linear models

Linear models for regression

A linear model f is defined as:

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \mathbf{w}^\top \mathbf{x} = \sum_j w_j x_j \,, \tag{6}$$

where $\mathbf{w}$ is a vector of adaptable parameters.

This model is fundamental in many disciplines, ranging from econometrics to statistics.

A more general formulation considers the inclusion of an **offset** (bias) $b \in \mathbb{R}$:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b\,. \tag{7}$$

Because we can always rewrite this as $\mathbf{w}^\top \bar{\mathbf{x}}$, with $\bar{\mathbf{x}} = [\mathbf{x};\ 1]$, we can avoid writing the bias explicitly to simplify the notation.

Hint: Everytime we write a linear model, mentally add an offset term whenever needed.

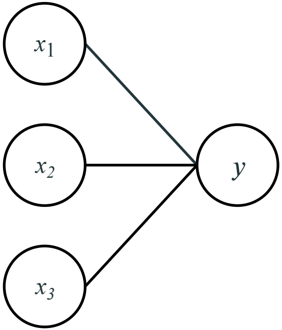


Figure 1: Each arrow represents a *linear* influence on the destination, which sums the results.

Combining the squared loss with a linear model results in the **least-squares** optimization problem:

$$\mathrm{LS}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 . \tag{8}$$

We can vectorize LS as:

$$\mathrm{LS}(\mathbf{w}) = \frac{1}{n}\Big\| \underset{(n)}{\mathbf{y}} - \underset{(n,d)}{\mathbf{X}}\underset{(d)}{\mathbf{w}} \Big\|^2 , \tag{9}$$

where $[\mathbf{X}]_i = \mathbf{x}_i$ and $[\mathbf{y}]_i = y_i$.

LS is a convex problem, with a simple gradient (**normal equations**):

$$\nabla \mathrm{LS}(\mathbf{w}) = -\frac{2}{n}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w})\,. \tag{10}$$

However, LS is special in the sense that $\nabla \mathrm{LS}(\mathbf{w}) = 0$ is a linear equation that can be solved explicitly:

$$\mathbf{w}^* = \underbrace{(\mathbf{X}^\top\mathbf{X})}_{(d,d)}^{-1}\underbrace{\mathbf{X}^\top}_{(d,n)}\underbrace{\mathbf{y}}_{(n)}\,. \tag{11}$$

Numerical problems in the inversion of $(\mathbf{X}^\top\mathbf{X})$ can be solved by adding a small amount of ℓ_2 **regularization** (**ridge regression**):

$$\text{LS-REG}(\mathbf{w}) = \text{LS}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2 \,, \tag{12}$$

for some $\lambda > 0$. This makes the problem *strictly* convex and forces the solution to be contained in a ball of given radius, modifying the gradient and the explicit solution as:

$$\nabla\text{LS-REG}(\mathbf{w}) = \nabla\text{LS}(\mathbf{w}) + \lambda\mathbf{w} \,. \tag{13}$$

$$\mathbf{w}^* = \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^\top\mathbf{y} \,. \tag{14}$$

where $\mathbf{I}$ is the identity matrix of appropriate shape.

Generating some data:

```
# Linear model with unknown coefficients
X = tf.random.normal((10, 5))
y = X @ tf.random.normal((5, 1))
```

Computing a linear model:

```
w = tf.random.normal((5, 1))
yhat = X @ w # (10, 1)
```

Computing the objective function:

```
mse = tf.reduce_sum((y - yhat)**2)
```

Explicit solution (numerically unstable):

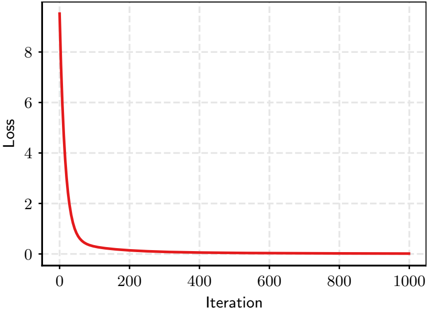
```
wopt = tf.linalg.inv(tf.transpose(X) @ X) @ tf.transpose(X) @ y
```

Explicit solution (better numerical conditioning):

```
wopt = tf.linalg.solve(tf.transpose(X) @ X, tf.transpose(X) @ y)
```

Simple implementation of gradient descent:

```
for i in range(15000):
  # Note the sign: the derivative has a minus!
  w = w + 0.001 * tf.transpose(X) @ (y - X @ w)
```



Linear models

Linear models for classification

Another important class of supervised learning problems is **classification**, where y is an integer $\{1, \dots, c\}$, such that $y_i = j$ means that $\mathbf{x}_i$ is of class j.

For example, with $c = 3$ we might have:

- $y = 1$: the email is spam;
- $y = 2$: the email is legit;
- $y = 3$: the email is dubious.

Solving these as regression tasks is generally not an optimal choice: among other things, it is not guaranteed that classes have a definite ordering.

A common solution is to predict a **probability distribution** over the classes.

A vector $\mathbf{a}_{(c)}$ belongs to the **probability simplex** Δ_c if:

$$\sum_i [\mathbf{a}]_i = 1\,, \quad [\mathbf{a}]_i \geq 0\,. \tag{15}$$

If $f(\mathbf{x}) = \hat{\mathbf{y}} \in \Delta_c$, we can interpret it as a probability distribution, e.g., we can select the class with highest probability as:

$$\text{class} = \arg\max_i \, [\hat{\mathbf{y}}]_i\,. \tag{16}$$

Note that we *cannot* directly predict an integer with our models, because it would require some form of threshold operation which is not compatible with gradient descent (gradient zero almost everywhere).

Predicting a probability distribution can be seen as a *soft* approximation to this problem.

The **softmax** function maps any vector to the probability simplex:

$$[\text{softmax}(\mathbf{a})]_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)} \tag{17}$$

The numerator ensures that all outputs are positive, while the denominator ensures that the final vector sums to 1.

Our linear model for classification becomes:

$$\underset{(c)}{f(\mathbf{x})} = \text{softmax}(\underset{(c,d)}{\mathbf{W}} \cdot \underset{(d)}{\mathbf{x}}) \tag{18}$$

The pre-softmax values $\mathbf{Wx}$ are called the **logits** of the model.

In order to compare the predictions with the ground truth, we encode our targets using a **one-hot encoding**. Given a pair $(\mathbf{x}, \mathbf{y})$:

$$y_i = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is of class } i\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{19}$$

For example, with 3 classes $\{\text{cat}, \text{dog}, \text{other}\}$:

$$\text{cat} = [1, 0, 0] \quad \text{dog} = [0, 1, 0] \quad \text{other} = [0, 0, 1]\,. \tag{20}$$

This is a probability distribution putting all the **mass** on a single class.

Finally, we need a loss function l to compare two probability distributions.

The **cross-entropy** loss is defined for two vectors $\mathbf{y}, \hat{\mathbf{y}} \in \Delta_c$ as:

$$\mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log\left(\hat{y}_i\right) \,. \tag{21}$$

The cross-entropy can be interpreted as the Kullback–Leibler divergence (a common distance measure between probability distributions) between $\mathbf{y}$ and $\hat{\mathbf{y}}$.

A **logistic regression** is a linear model $f(\mathbf{x}) = \text{softmax}(\mathbf{W}\mathbf{x})$ trained by optimizing the cross-entropy:

$$\text{LR}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \text{CE}\left(\mathbf{y}_i, f(\mathbf{x}_i)\right) \ . \tag{22}$$

It is not possible to solve the logistic regression problem explicitly. A linear model for classification has dc parameters.

A special case is **binary classification**, where $c = 2$. In this case, we can predict a single scalar value $f(\mathbf{x}) \in [0, 1]$ since:

$$
\begin{aligned}
f(\mathbf{x}) &\quad \text{probability of class 1} \;, && (23)\\
1 - f(\mathbf{x}) &\quad \text{probability of class 2} \;. && (24)
\end{aligned}
$$

In this case, the softmax simplifies to the **sigmoid function**:

The sigmoid $\sigma(s) \in [0, 1]$ is defined as:

$$
\sigma(s) = \frac{1}{1 + \exp(-s)} \,. \tag{25}
$$

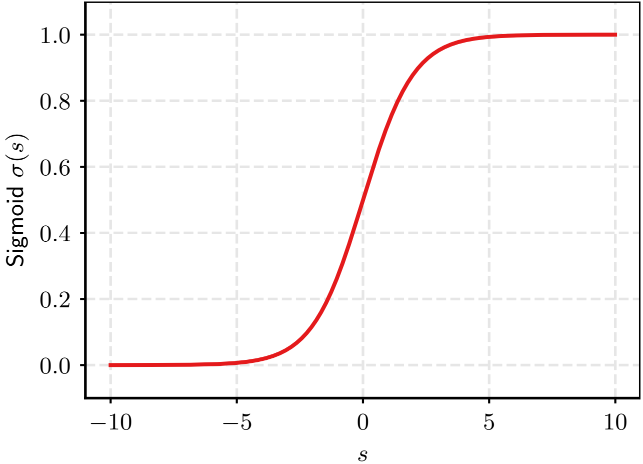


Figure 2: A visualization of the sigmoid function. Note that 0 and 1 are only approached asymptotically.

Combining everything, we obtain a binary version of the logistic regression algorithm:

$$\text{BIN-LR}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\Big[-\underbrace{y_i \log\left(\sigma(\mathbf{w}^\top \mathbf{x})\right)}_{\text{Class 1}} - \underbrace{(1-y_i)\log\left(1-\sigma(\mathbf{w}^\top \mathbf{x})\right)}_{\text{Class 2}}\Big] \tag{26}$$

In this case, we can obtain the most probable class from the model as:

$$\text{class} = \begin{cases} 1 & \text{if } \sigma(\mathbf{w}^\top \mathbf{x}) > 0.5\,, \\ 2 & \text{otherwise}\quad. \end{cases} \tag{27}$$

By manually differentiating we obtain:

$$\sigma'(s) = \sigma(s)(1 - \sigma(s)) \,. \tag{28}$$

Plugging this into the gradient computation we obtain:

$$\nabla \text{BIN-LR}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} (\sigma(\mathbf{w}^\top \mathbf{x}_i) - y_i)\mathbf{x}_i \,, \tag{29}$$

showing its similarity to the regression case.

```python
from tensorflow.keras.metrics import *

# The one we have described up to now.
categorical_crossentropy(ytrue, yhat)

# ytrue should contain the indexes of the classes instead of the
# one-hot encodings.
sparse_categorical_crossentropy(ytrue, yhat)

# Numerically-stable versions requiring the logits as inputs
# (see the LogSumExp trick).
categorical_crossentropy(ytrue, yhat, from_logits=True)
sparse_categorical_crossentropy(ytrue, yhat, from_logits=True)
```

Linear models

Calibration and a probabilistic formulation

More formally, for the classification setting we assumed $f(x)$ parameterizes a *probability distribution* $\hat{p}(y \,|\, f(x))$ over the output y. We can always do this, e.g., for regression:

$$\hat{p}(y \,|\, f(x)) = \mathcal{N}(y \,|\, f(x), \sigma^2)\,, \tag{30}$$

where the model predicts the center of a Gaussian distribution with fixed variance (hyper-parameter). This probabilistic formulation can be more flexible or useful in many contexts.

The probabilistic formulation also provides a principled way to interpret training by maximizing the likelihood of the model (assuming the elements of the dataset are i.i.d.):

$$f^*(x) = \arg\max \prod_{(x_i, y_i)} \hat{p}(y_i \mid f(x_i))\,. \tag{31}$$

For (30), this is **equivalent** to training with a squared loss. Similarly, training with cross-entropy is **equivalent** to assuming a categorical distribution over the output (can you prove it?).

A common misconception when doing classification is that $[f(x)]_i$ can be immediately interpreted as *the probability of pattern x being of class i*.

However, this is only true whenever the trained model satisfies:

$$p(y = i \mid x) = [f(x)]_i \,. \tag{32}$$

We say the model is well **calibrated**, but this must be checked manually.

Guo, C., et al.. **On calibration of modern neural networks**. ICML 2017.

To measure the calibration of a model, we keep a separate validation set, and we split the interval $[0, 1]$ into m equispaced bins (each of size $1/m$). Define:

- B_m the number of samples from the validation set, whose predicted confidence falls in bin m.
- p_m the average confidence of the network for that bin.
- a_m the average accuracy of the network for these elements.

Then, the **expected calibration error** (ECE) is given by:

$$\text{ECE} = \sum_m \frac{B_m}{n} |a_m - p_m| \,. \tag{33}$$

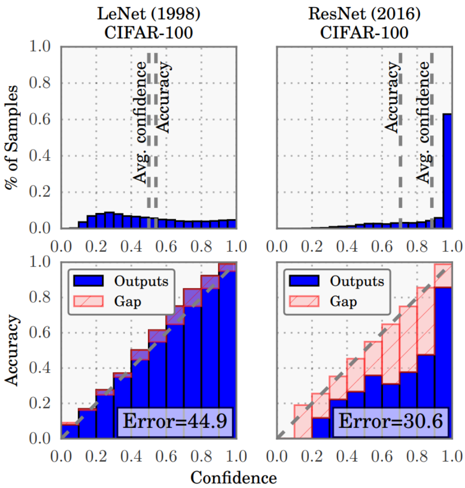


Figure 3: Plotting a_m against p_m for every bin gives us a **reliability plot** (from Guo et al., 2017).

This topic is important because more complex networks may be highly over (or under) confident, with many methods to improve it (temperature scaling, logit normalization, ...).

A simple (and popular) option is to decrease the weight given to 'easy' samples using a variant of cross-entropy call the **focal loss**:

$$\text{FL}_{\alpha}(\mathbf{y}, \hat{\mathbf{y}}) = -(1 - \hat{\mathbf{y}}_c)^{\alpha} \log \hat{\mathbf{y}}_c \,, \tag{34}$$

where $c = \arg\max \mathbf{y}$.

Mukhoti, J., et al., 2020. **Calibrating deep neural networks using focal loss**. *Advances in Neural Information Processing Systems*, 33, pp. 15288-15299.

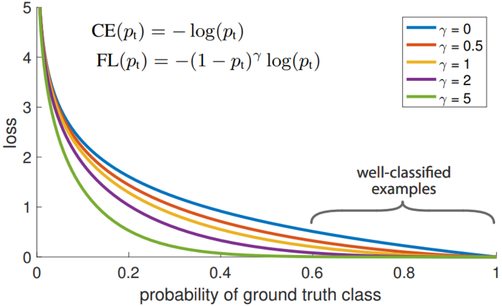


Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As

Lin, T.Y., et al., 2017. **Focal loss for dense object detection**. In *IEEE ICCV* (pp. 2980-2988).

- (**Required**) Chapter 3 and 4 from the book.