Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 3: Linear regression and classification

Lecturer: S. Scardapane

Sapienza Università di Roma

Supervised learning

Setup and examples

(Informal) definition of supervised learning

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

Constraints on the dataset

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., in recognizing cats, the distribution of species do not change).
- **Independently distributed**: there is no bias in the data collection (e.g., we mostly collect siamese cats).

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

Some motivating examples

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."

$$\begin{tikzcd} \text{"Il gatto è sul tavolo."} \arrow[d] \\ f(x) \arrow[r] & \text{The cat is on the table.} \\ & \text{A cat sitted in table.} \\ & \text{Table sitting cat cats.} \end{tikzcd}$$

The cat is on the table.

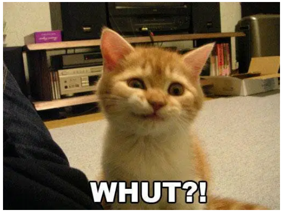
Good translation

A cat sitted in table.

Still understandable

Table sitting cat cats.

Uh?



Introducing loss functions

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. Importantly, the loss function should be a scalar, vary gradually, and (as we will see) be **differentiable**.

Expected risk and empirical risk

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**.

Overfitting

Consider the following algorithm (a variation of 1-NN):

$$f(x) = \begin{cases} y & \text{if } (x, y) \in \mathcal{S}\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{5}$$

This has 0 training error by construction, but a very large test error: this is an example of **overfitting**. We will see that (large) neural networks have many counter-intuitive properties when it comes to generalization.

Some simplifications

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 is a single real number.

There are three interesting cases: when $y \in \mathbb{R}$, it is a **regression** task; when $y \in \{0, \ldots, c-1\}$, it is a **classification** task; when $c = 2$, it is a **binary classification** task.

Supervised learning

Building a loss function

How to build a loss function?

Building a loss function can be done empirically: considering regression, for example, the prediction error $(y - \hat{y})$ is a reasonable quantity to penalize.

Since most times we do not care about the sign of the error, these are all valid choices:

$$\underbrace{(y - \hat{y})^2}_{\textbf{squared loss}} \quad , \quad \underbrace{|y - \hat{y}|}_{\textbf{absolute loss}} \quad , \quad \underbrace{(\log(1 + y) - \log(1 + \hat{y}))^2}_{\textbf{squared log loss}} \tag{6}$$

Is there a principled way to make the choice?

Probabilistic formulation

Let us assume that $f(x)$ parameterizes a *probability distribution* $\hat{p}(y \,|\, f(x))$ over the possible outputs y. We are now choosing $\hat{p}(\cdot \,|\, \cdot)$ instead of $l(\cdot, \cdot)$, which can feel more natural.

For example, for regression we can assume a Gaussian shape:

$$\hat{p}(y \,|\, f(x)) = \mathcal{N}(y \,|\, f(x), \sigma^2)\,, \tag{7}$$

where the model predicts the center of a Gaussian distribution with fixed variance (hyper-parameter).

Maximum likelihood

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

$$f^*(x) = \arg\max \prod_{(x_i,y_i)} \hat{p}(y_i \,|\, f(x_i)) = \arg\min \sum_{(x_i,y_i)} -\log \hat{p}(y_i \,|\, f(x_i))\,. \tag{8}$$

Under the Gaussian assumption in (7), we obtain $-\log \hat{p}(y_i \,|\, f(x_i)) \propto (y_i - f(x_i))^2$, i.e., we should optimize the squared loss!

Linear models

Linear models for regression

What is a linear model?

A linear model f is defined as:

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \mathbf{w}^\top \mathbf{x} = \sum_j w_j x_j \,, \tag{9}$$

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

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

Linear models with bias

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

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b\,. \tag{10}$$

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.

Graphical representation

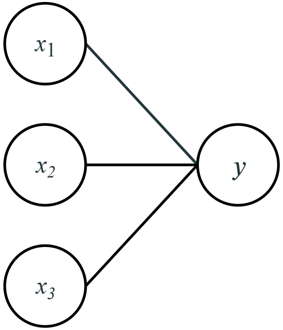


Figure 1: Each arrow represents a *linear* influence on the destination, which sums the results.

Least-squares cost function

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

$$\mathrm{LS}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 . \tag{11}$$

We can vectorize LS as:

$$\mathrm{LS}(\mathbf{w}) = \frac{1}{n}\Big\| \underset{(n)}{\mathbf{y}} - \underset{(n,d)}{\mathbf{X}}\underset{(d)}{\mathbf{w}}\Big\|^2 , \tag{12}$$

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

Linear models

Solving the LS problem

Solving the LS problem

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

$$\nabla \text{LS}(\mathbf{w}) = \frac{2}{n}\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})\,. \tag{13}$$

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

$$\mathbf{w}^* = \underbrace{(\mathbf{X}^\top\mathbf{X})}_{(d,d)}{}^{-1}\underbrace{\mathbf{X}^\top}_{(d,n)}\underbrace{\mathbf{y}}_{(n)} = \underbrace{\mathbf{X}^\top}_{(d,n)}\underbrace{(\mathbf{X}\mathbf{X}^\top)}_{(n,n)}{}^{-1}\underbrace{\mathbf{y}}_{(n)}\,. \tag{14}$$

The terms $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ and $\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}$ are called the **pseudoinverses** of $\mathbf{X}$, and they require the corresponding matrices to be invertible (full rank).

Regularizing the LS problem

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**):

$$\text{LS-REG}(\mathbf{w}) = \text{LS}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2\,, \tag{15}$$

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:

$$\nabla\text{LS-REG}(\mathbf{w}) = \nabla\text{LS}(\mathbf{w}) + \lambda\mathbf{w}\,. \tag{16}$$

$$\mathbf{w}^* = \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^\top\mathbf{y}\,. \tag{17}$$

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

Show me some code!

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_mean((y - yhat)**2)
```

Show me some code (2)!

Explicit solution (numerically unstable):

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

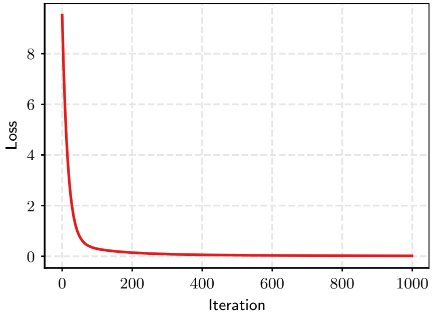
Explicit solution (better numerical conditioning):

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

Show me some code (3)!

Simple implementation of gradient descent:

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



Computational complexity of LS

Matrix multiplication $\underset{(a,b)}{\mathbf{A}}\underset{(b,c)}{\mathbf{B}}$ has complexity $\mathcal{O}(abc)$, while matrix inversion has complexity (roughly) cubic.

For large models and datasets, we want algorithms that can scale linearly in both n and d: we will see that gradient descent can satisfy this property.

Linear models

Overparameterized models and convergence (*optional*)

Analyzing the convergence

Rewriting the GD step for the LS problem (ignoring the constant factor $2/n$):

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \mathbf{X}^\top (\mathbf{X}\mathbf{w}_{t-1} - \mathbf{y}) \,. \tag{18}$$

We can use this to write out how the *predictions* over the training set evolve:

$$\hat{\mathbf{y}}_t = \hat{\mathbf{y}}_{t-1} - \eta \mathbf{X}\mathbf{X}^\top \left(\hat{\mathbf{y}}_{t-1} - \mathbf{y}\right) \,, \tag{19}$$

or, after some manipulation:

$$\left(\hat{\mathbf{y}}_t - \mathbf{y}\right) = \left(\mathbf{I} - \eta \mathbf{X}\mathbf{X}^\top\right)\left(\hat{\mathbf{y}}_{t-1} - \mathbf{y}\right) = \left(\mathbf{I} - \eta \mathbf{X}\mathbf{X}^\top\right)^t \left(\hat{\mathbf{y}}_0 - \mathbf{y}\right) \,. \tag{20}$$

See PPA, Chapter 5 for the full analysis.

Overparameterized models

Eq. (20) is an interesting dynamical system: it will diverge in general, unless (a) all eigenvalues $\lambda_0, \ldots, \lambda_n$ of $\mathbf{X}\mathbf{X}^\top$ are non-negative, and (b) $\eta \in [0, 1/\lambda_0]$. In this case, it will decrease to 0. If all eigenvalues are positive, it will do so *exponentially*.

For the latter condition, we need $d > n$, otherwise the matrix $\mathbf{X}\mathbf{X}^\top$ will have low-rank (hence, some zero eigenvalues). We call this the **over-parameterized** regime: it is possible in this case for LS to perfectly interpolate all training data exponentially fast!

Linear models

Linear models for classification

Multi-class classification

In **classification**, y is an integer $\{0, \ldots, c-1\}$, such that $y_i = j$ means that $\mathbf{x}_i$ is of class j.

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

- $y = 0$: the email is spam;
- $y = 1$: the email is legit;
- $y = 2$: 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 $\underset{(c)}{\mathbf{a}}$ belongs to the **probability simplex** Δ_c if:

$$\sum_i [\mathbf{a}]_i = 1 \,, \quad [\mathbf{a}]_i \geq 0 \,. \tag{21}$$

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

$$\text{class} = \arg\max_i \, [\hat{\mathbf{y}}]_i \,. \tag{22}$$

A comment on differentiability

Note that we *cannot* easily 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

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

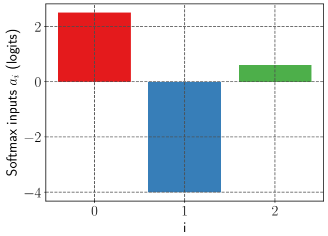
$$[\text{softmax}(\mathbf{a})]_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)} \tag{23}$$

The numerator ensures that all outputs are positive, while the denominator ensures that the final vector sums to 1. It can be seen as a soft approximation to the argmax (a better name is in fact *softargmax*).

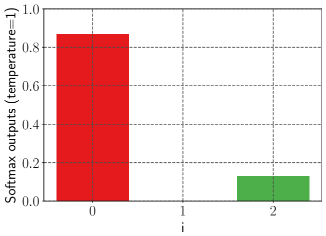
If desired, we can control the approximation with an additional hyper-parameter τ called **temperature**:

$$[\text{softmax}(\mathbf{a})]_i = \frac{\exp(a_i/\tau)}{\sum_j \exp(a_j/\tau)} \tag{24}$$

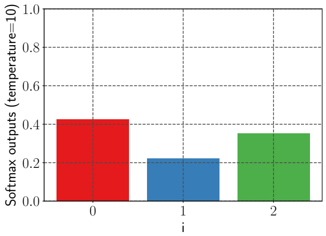
Visualizing the softmax function



(a) Softmax inputs (logits)



(b) Softmax outputs (temperature = 1)



(c) Softmax outputs (temperature = 10)

Final model

Our linear model for classification becomes:

$$\underset{(c)}{f(\mathbf{x})} = \text{softmax}(\underset{(c,d)}{\mathbf{W}} \underset{(d)}{\mathbf{x}}) \tag{25}$$

The pre-softmax values $\mathbf{Wx}$ are called the **logits** of the model. By explicitly writing the biases $\underset{(c)}{\mathbf{b}}$:

$$f(\mathbf{x}) = \text{softmax}(\mathbf{Wx} + \mathbf{b})\,. \tag{26}$$

One-hot encoding

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

$$y_i = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is of class } i\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{27}$$

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

$$\text{cat} = [1, 0, 0] \quad \text{dog} = [0, 1, 0] \quad \text{other} = [0, 0, 1]\,. \tag{28}$$

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

Cross-entropy loss

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:

$$\text{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log\left(\hat{y}_i\right) \,. \tag{29}$$

The CE loss can be derived from the maximum likelihood principle under the assumption that $f(\mathbf{x})$ encodes a **categorical** distribution (try it!).[1]

[1]The equation for the categorical distribution in this case would be $p(\mathbf{y}|f(\mathbf{x})) = \prod_i [f(\mathbf{x})]_i^{y_i}$.

Log-likelihood loss

The CE loss is generic for *any* pair of vectors belonging to the probability simplex. However, in our case $\mathbf{y}$ is always a one-hot encoded vector.

Denote by t the index of the class, $t = \arg\max \mathbf{y}$, the CE loss simplifies since only one term in the summation will be different from 0:

$$\text{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\log(\hat{y}_t)\,. \tag{30}$$

Hence, CE can be interpreted as maximizing the probability of the true class at the expense of all other outputs.

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

$$\text{LR}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \text{CE}\left(\mathbf{y}_i, f(\mathbf{x}_i)\right) \; . \tag{31}$$

It is not possible to solve the logistic regression problem explicitly. A linear model for classification has dc parameters (or $dc + c$ with biases).

Binary classification

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:

$$\begin{aligned} f(\mathbf{x}) &\quad \text{probability of class 1} \ , \qquad (32) \\ 1 - f(\mathbf{x}) &\quad \text{probability of class 2} \ . \qquad (33) \end{aligned}$$

The softmax function simplifies to the **sigmoid** function:

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

$$\sigma(s) = \frac{1}{1 + \exp(-s)} \ . \tag{34}$$

Visualizing the sigmoid function

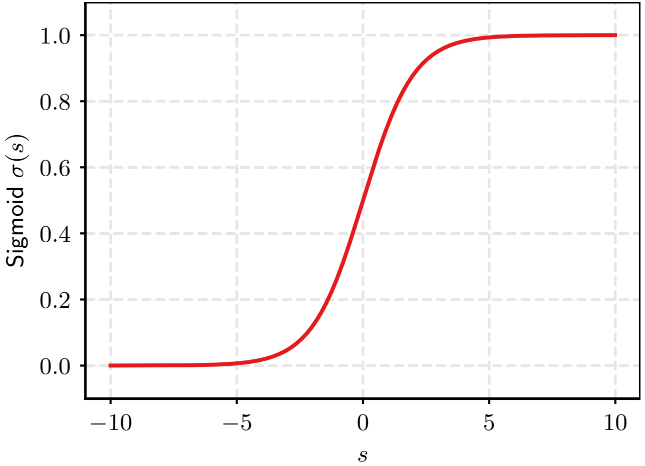


Figure 2: A visualization of the sigmoid function. Note that 0 and 1 are only approached asymptotically.

Binary logistic regression model

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

$$\text{BIN-LR}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\Big[-\underbrace{y_i \log\left(\sigma(\mathbf{w}^\top\mathbf{x})\right)}_{\text{Class 1}} - \underbrace{(1-y_i)\log\left(1-\sigma(\mathbf{w}^\top\mathbf{x})\right)}_{\text{Class 2}}\Big] \tag{35}$$

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

$$\text{class} = \begin{cases} 1 & \text{if } \sigma(\mathbf{w}^\top\mathbf{x}) > 0.5\,, \\ 0 & \text{otherwise}\ . \end{cases} \tag{36}$$

Gradient of the logistic regression

By manually differentiating we obtain:

$$\sigma'(s) = \sigma(s)(1 - \sigma(s))\,. \tag{37}$$

Plugging this into the gradient computation we obtain:

$$\nabla \text{BIN-LR}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(\sigma(\mathbf{w}^\top \mathbf{x}_i) - y_i)\mathbf{x}_i\,, \tag{38}$$

showing its similarity to the regression case.

Implementation

```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
categorical_crossentropy(ytrue, yhat, from_logits=True)
sparse_categorical_crossentropy(ytrue, yhat, from_logits=True)
```

The logsumexp function

Why a variant with logits in input? Note that the *i*th term of the cross-entropy wrt the logits $\mathbf{p}$ is given by:

$$-\log\left(\frac{\exp p_i}{\sum_j \exp p_j}\right) . \tag{39}$$

In practice, this can be *highly* unstable. However, it can be rewritten as:

$$-p_i + \operatorname{logsumexp}(\mathbf{p}) , \tag{40}$$

where logsumexp is defined as $\operatorname{logsumexp}(\mathbf{p}) = \log\left(\sum_i \exp p_i\right)$.

`https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/`

The logsumexp trick

The reason this is important is that the logsumexp function is invariant in the following sense:

$$\text{logsumexp}(\mathbf{p}) = \log\left(\sum_i \exp(p_i - c)\right) + c\,, \tag{41}$$

where c is an arbitrary constant. By setting $c = \max(\mathbf{p})$, we can ensure numerical instabilities never occur.

In this sense, softmax can be interpreted as part of the model *or* as part of the loss; this is not an issue, since $\arg\max$ only cares about the relative ranking of the values, which is not changed by the softmax.

Linear models

Calibration and a probabilistic formulation

Calibration

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:

$$p(y = i \mid x) = [f(x)]_i \,. \tag{42}$$

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.

Measuring calibration

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:

$$\text{ECE} = \sum_m \frac{B_m}{n} |a_m - p_m| \,. \tag{43}$$

Calibration plots

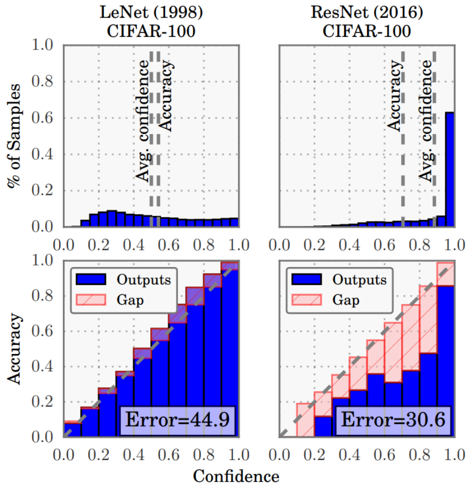


Figure 3: Plotting a_m against p_m for every bin gives us a **reliability plot** (from Guo et al., 2017).

Focal loss

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:

$$\text{FL}_\alpha(\mathbf{y}, \hat{\mathbf{y}}) = -(1 - \hat{y}_c)^\alpha \log \hat{y}_c \,, \tag{44}$$

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.

Visualizing the focal loss

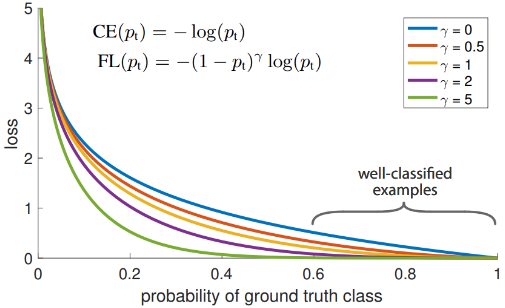


Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As

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

Reading material

- D2L: Chapters 3 and 4; UDL: Chapter 2; PPA: parts of Chapters 3-5.