Introduction to Deep Learning

Copernicus Master on Digital Earth

Lecture 4: MultiLayer Perceptron

Dr. Charlotte Pelletier
charlotte.pelletier@univ-ubs.fr

Today

Summary on the backpropagation in PyTorch

MultiLayer Perceptron

- How it works?
- The role of the non-linearity, and activation functions
- Computing the number of trainable parameters
- Loss functions
- Going deeper

Autograd

Autograd

- Conceptually, the forward pass is a standard tensor computation, and the computational graph of tensor operations is required only to compute derivatives.
- When executing tensor operations, PyTorch can automatically construct on-the-fly the graph of operations to compute the gradient of any quantity with respect to any tensor involved.
- This "autograd" mechanism (Paszke et al., 2017) has two main benefits:
  - Simpler syntax: one just needs to write the forward pass as a standard sequence of Python operations,
  - greater flexibility: since the graph is not static, the forward pass can be dynamically modulated.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... & Lerer, A. (2017). Automatic differentiation in pytorch.

A `Tensor` has a Boolean field `requires_grad`, set to `False` by default, which states if PyTorch should build the graph of operations so that gradients with respect to it can be computed.

The result of a tensorial operation has this flag to `True` if any of its operand has it to `True`.

```
>>> x = torch.tensor([ 1., 2. ])
>>> y = torch.tensor([ 4., 5. ])
>>> z = torch.tensor([ 7., 3. ])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```

Only floating point type tensors can have their gradient computed:

```
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
/.../
RuntimeError: only Tensors of floating point dtype can require gradients
```

The method `torch.autograd.grad(outputs, inputs)` computes and returns the gradient of outputs with respect to inputs.

```
>>> t = torch.tensor([1., 2., 4.]).requires_grad_()
>>> u = torch.tensor([10., 20.]).requires_grad_()
>>> a = t.pow(2).sum() + u.log().sum()
>>> torch.autograd.grad(a, (t, u))
(tensor([2., 4., 8.]), tensor([0.1000, 0.0500]))
```

If `outputs` is a tuple, the result is the sum of the gradients of its elements.

The method `requires_grad_(value = True)` set `requires_grad` to value, which is `True` by default.

The function `Tensor.backward()` accumulates gradients in the grad fields of tensors which are not results of operations, the "leaves" in the autograd graph.

```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.])
```

This function is an alternative to `torch.autograd.grad(...)` and standard for training models.

`Tensor.backward()` accumulates the gradients in the grad fields of tensors, so one may have to set them to zero before calling it.

This accumulating behavior is desirable in particular to compute the gradient of a loss summed over several "mini-batches", or the gradient of a sum of losses.

The `detach()` method creates a tensor, which shares the data, but does not require gradient computation, and is not connected to the current graph.

This method should be used when the gradient should not be propagated beyond a variable, or to update leaf tensors.

MultiLayer Perceptron

MultiLayer Perceptron (MLP)

A linear classifier has the form

$$\mathbb{R}^d \to \mathbb{R}$$
$$x = g(w^T \cdot x + b),$$

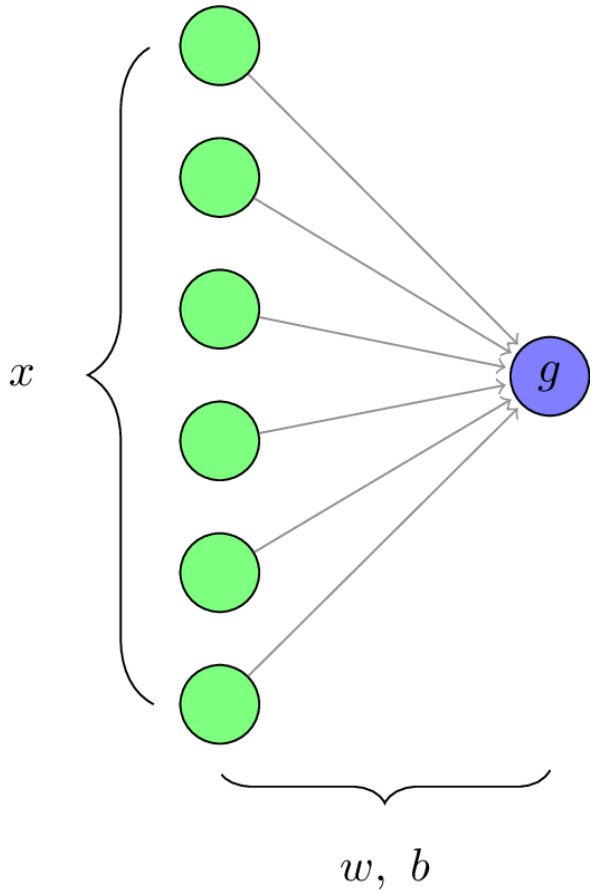
where $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ are the trainable parameters, and $g : \mathbb{R} \to \mathbb{R}$ the activation function.

It can be easily extended to a multi-dimensional output by applying a similar transformation to each output

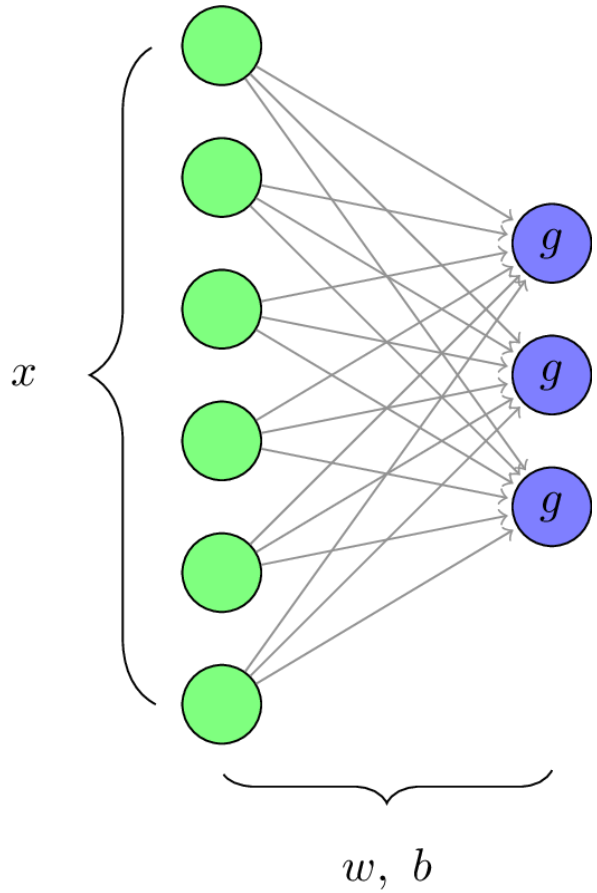
$$\mathbb{R}^d \to \mathbb{R}$$
$$x = g(w \cdot x + b),$$

where $w \in \mathbb{R}^{m \times d}, b \in \mathbb{R}^m$ (m is the number of inputs) and g is applied element-wise.

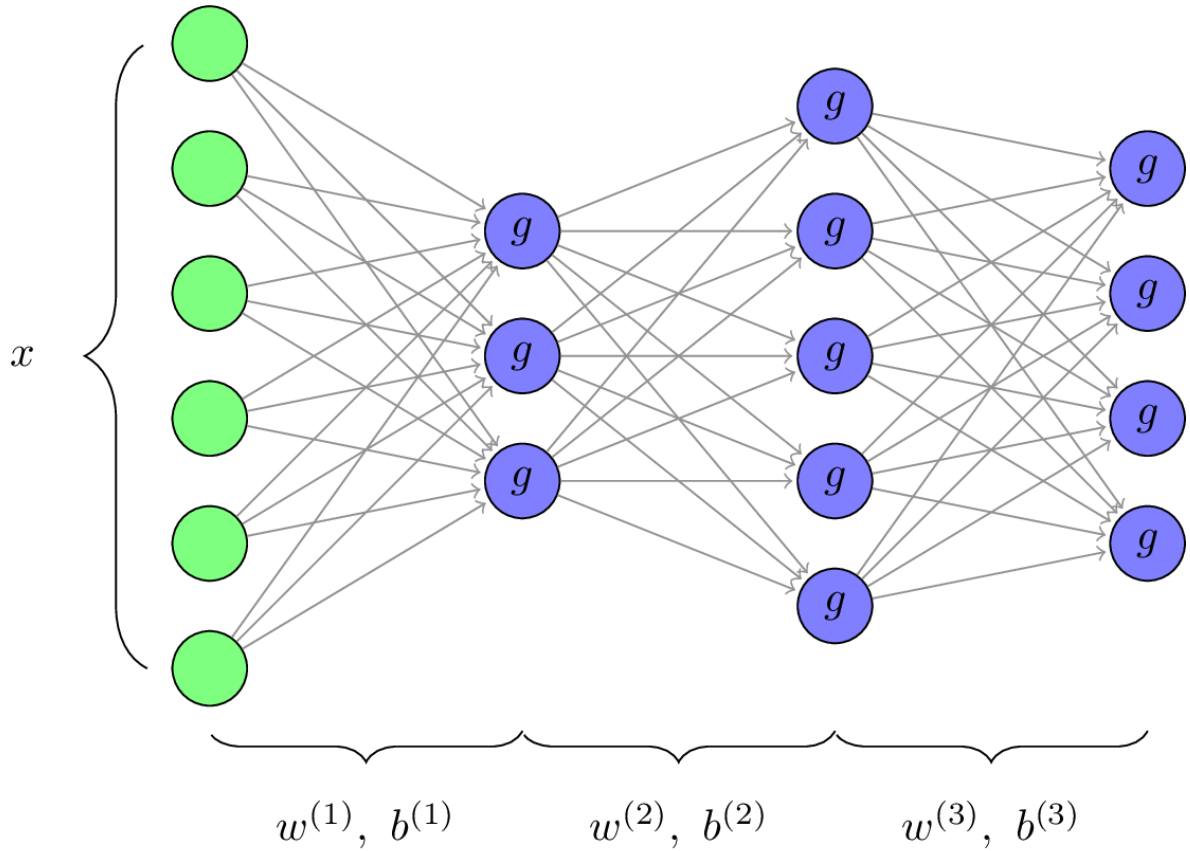
Such a model can be seen as a combination of units



.. and it can also be extended



.. and it can also be extended



This latter structure can be formally defined by

$$x^{(l)} = g(w^{(l)} x^{(l-1)} + b^{(l)}), \forall l \in \{1, ..., L\}$$

where L is the number of layers in the network.
We have also $x^{(0)} = x$ and $x^{(L)} = f(x; w, b)$.

Such a model is a MultiLayer Perceptron (MLP).

Activation functions

The role of the non-linearity

If g is an affine transformation ($g : x \mapsto A \cdot x + B$), the full MLP is a composition of affine mappings, and itself an affine mapping.

For example, if $g : x \mapsto x$, then

$$\begin{aligned} x^{(1)} &= g(w^{(1)}x^{(0)} + b^{(1)}) = w^{(1)}x + b^{(1)} \\ x^{(2)} &= g(w^{(2)}x^{(1)} + b^{(2)}) = w^{(2)}(w^{(1)}x + b^{(1)}) + b^{(2)} \\ \Rightarrow x^{(2)} &= w^{(1)}w^{(2)}x + w^{(2)}b^{(1)} + b^{(2)} \end{aligned}$$

It can be demonstrated for an infinite number of hidden layers. In other words, the output is a linear function of the inputs when the activation function is linear.

The role of the non-linearity

If g is an affine transformation ($g : x \mapsto A \cdot x + B$), the full MLP is a composition of affine mappings, and itself an affine mapping.

For example, if $g : x \mapsto x$, then

$$\begin{aligned} x^{(1)} &= g(w^{(1)}x^{(0)} + b^{(1)}) = w^{(1)}x + b^{(1)} \\ x^{(2)} &= g(w^{(2)}x^{(1)} + b^{(2)}) = w^{(2)}(w^{(1)}x + b^{(1)}) + b^{(2)} \\ \Rightarrow x^{(2)} &= w^{(1)}w^{(2)}x + w^{(2)}b^{(1)} + b^{(2)} \end{aligned}$$

It can be demonstrated for an infinite number of hidden layers. In other words, the output is a linear function of the inputs when the activation function is linear.

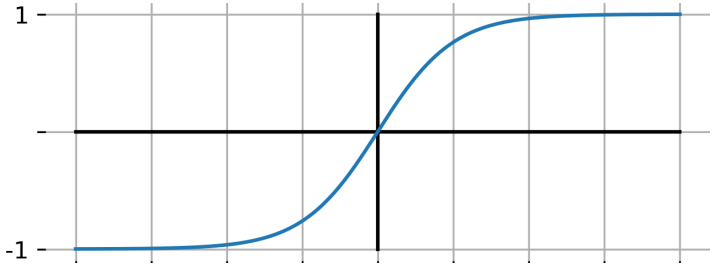
Activation functions

The activation functions should thus be non-linear to learn a non-linear transformation of the data.

Classical activation functions are sigmoidal functions such as tangent hyperbolic and logistic function.

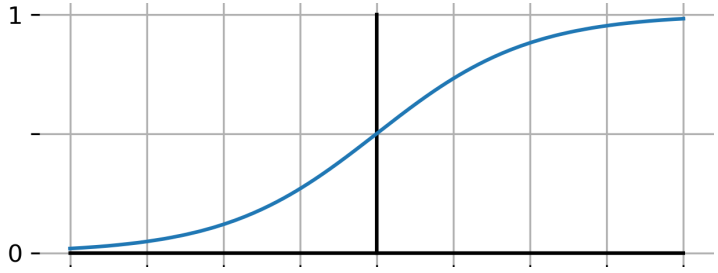
Hyperbolic tangent

$$x \mapsto \frac{2}{1 + e^{-2x}} - 1$$



Logistic

$$x \mapsto \frac{1}{1 + e^{-x}}$$



However, the most used activation functions is Rectified Linear Units (ReLU).

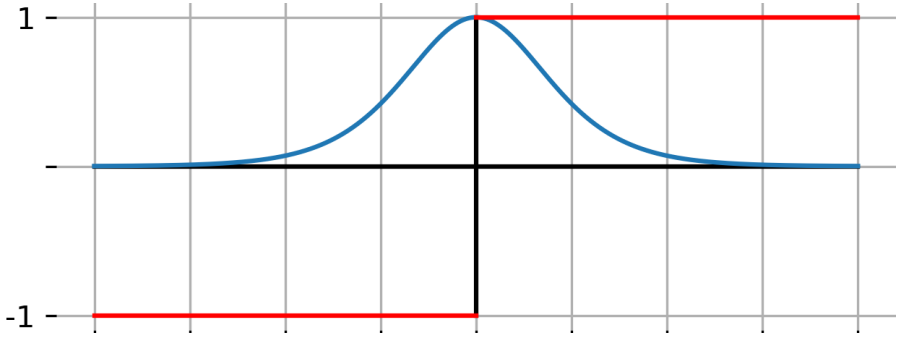
$$x \mapsto max(0, x)$$

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.

However, the most used activation functions is Rectified Linear Units (ReLU).

$$x \mapsto max(0, x)$$

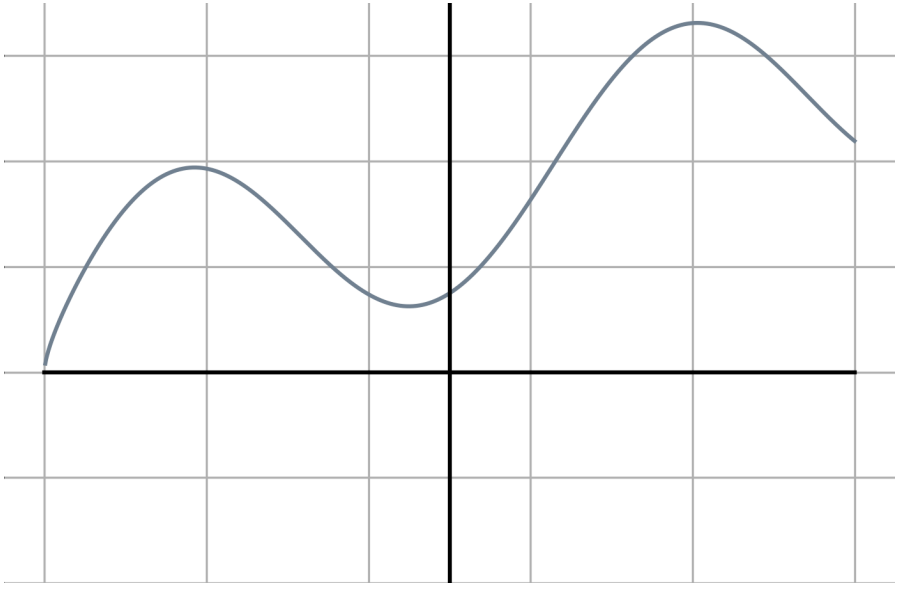
Derivatives of the hyperbolic tangent (in blue) and the ReLU (in red)



Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.

Universal approximation

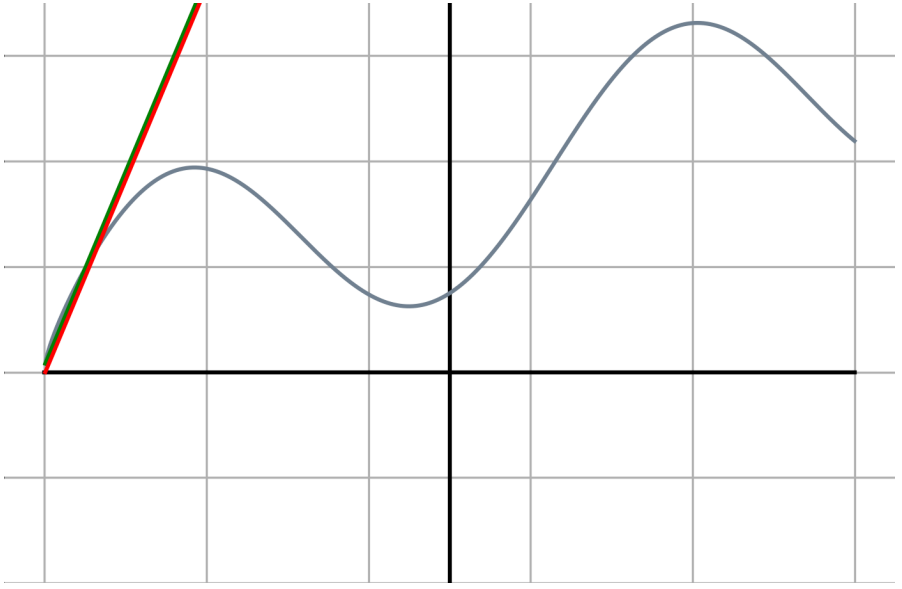
It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

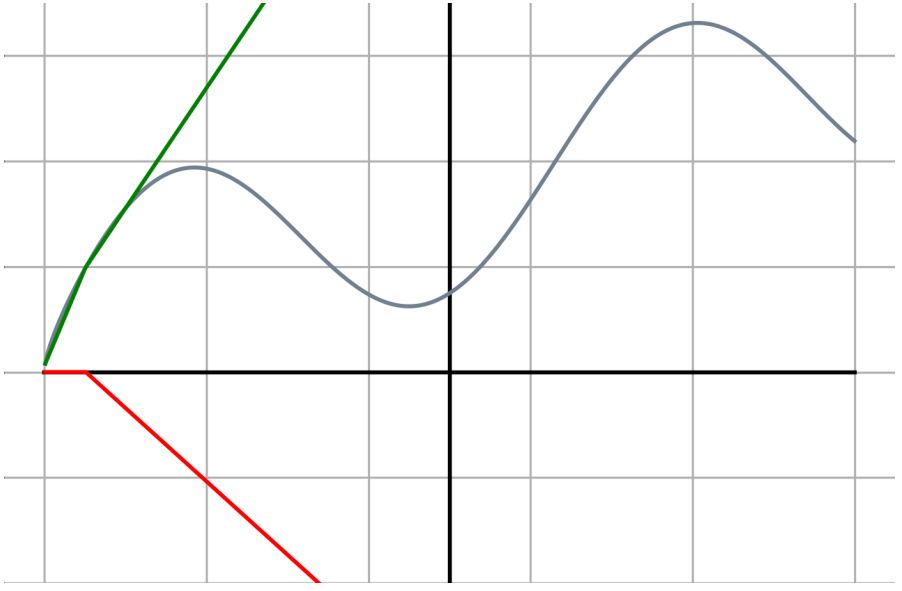
$$f(x) = \sigma(w_1 x + b_1)$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

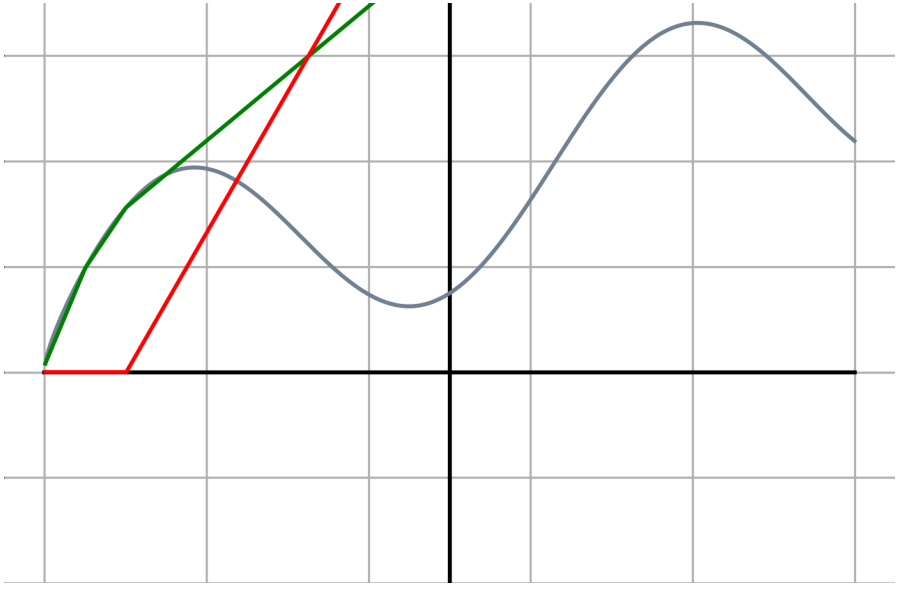
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

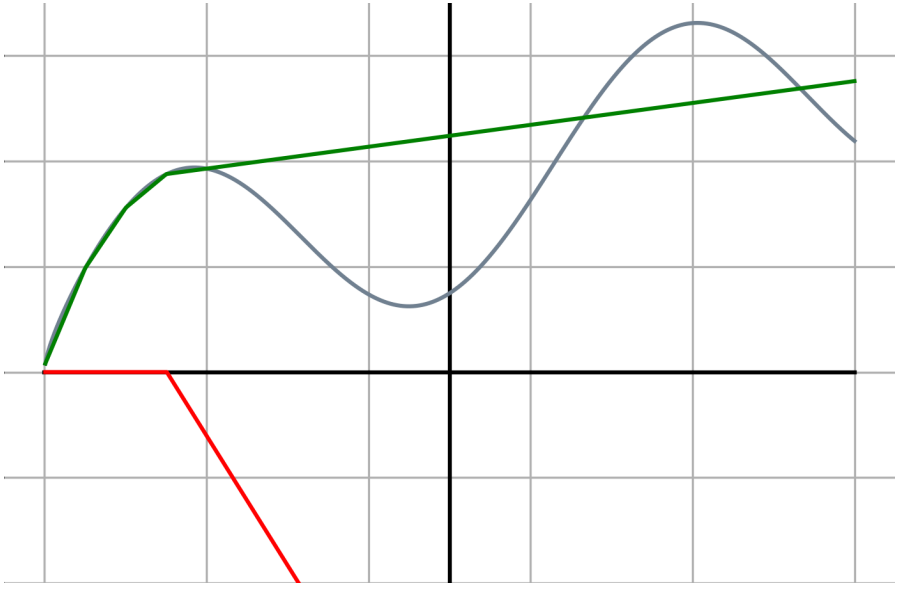
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3)$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

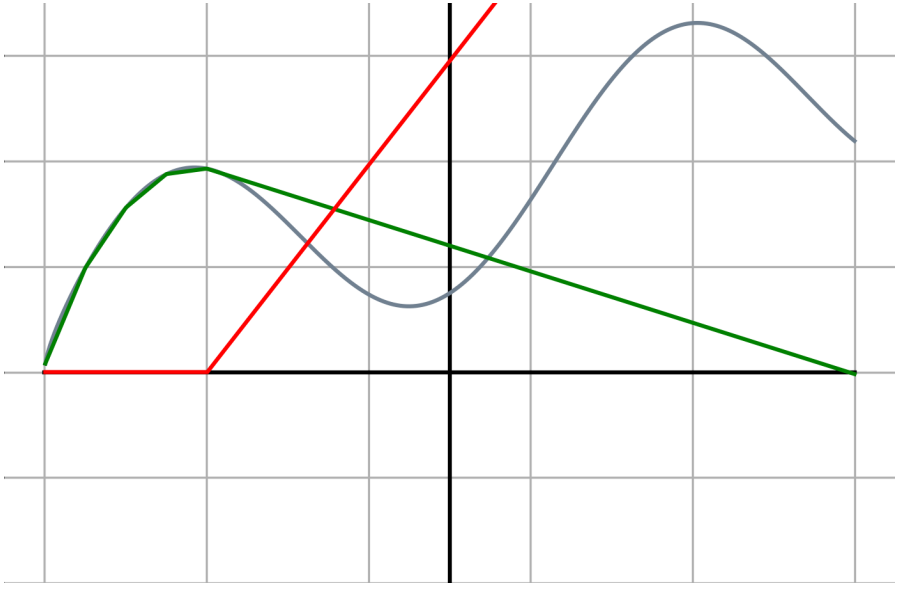
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4)$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

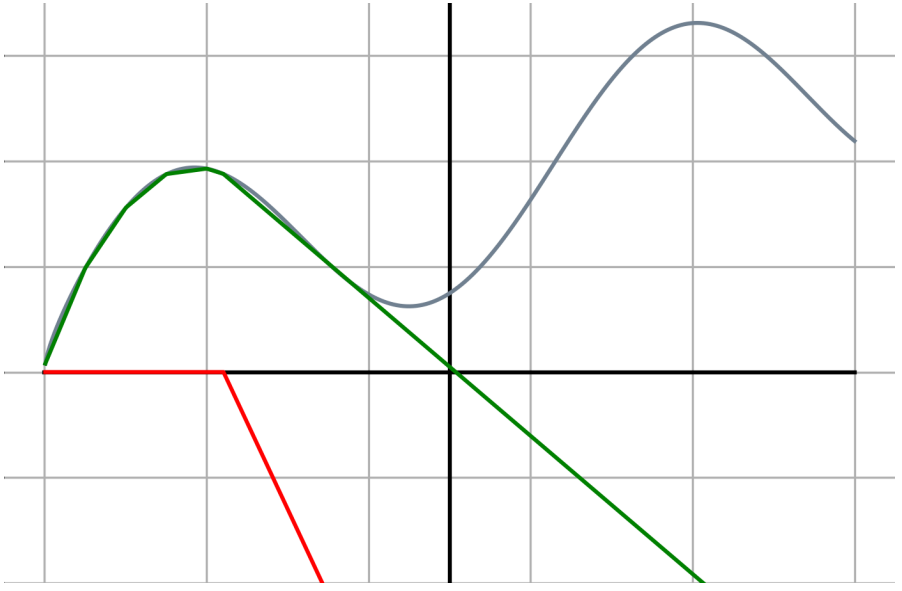
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

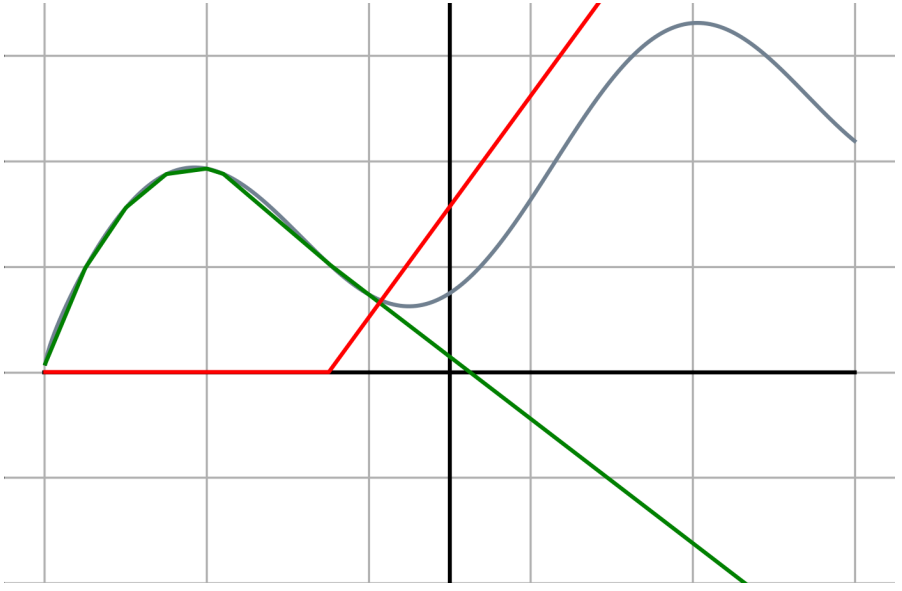
$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$



Universal approximation

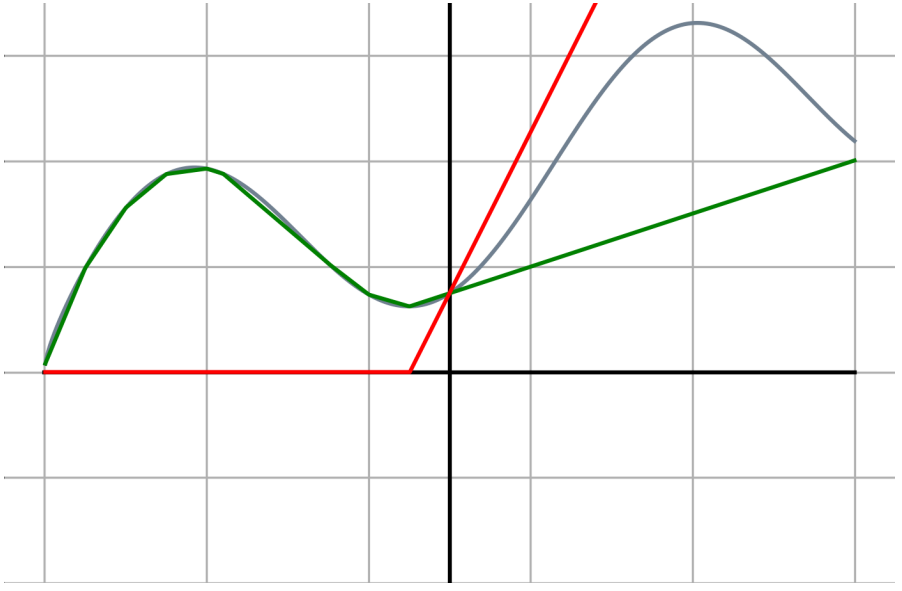
It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$

Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$



Universal approximation

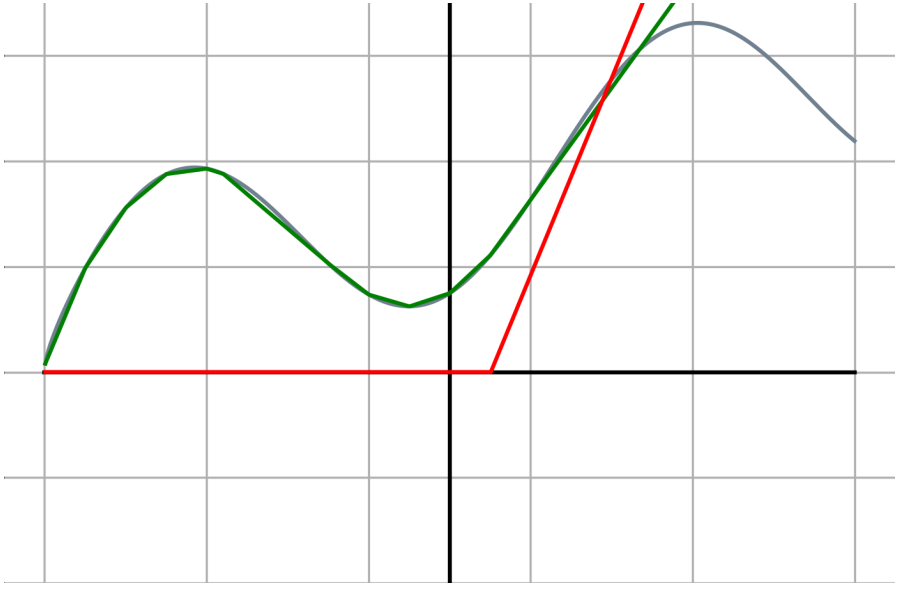
It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$

Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$



Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$

Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$

Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$

Universal approximation

It is possible to approximate any ψ function defined in the interval $[a, b]$ and that takes values in $\mathbb{R}$ with a linear combination of translated/scaled ReLU functions:

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \sigma(w_4 x + b_4) + \ldots$$

Universal approximation

This results can be extended to show that any continuous function

$$\psi : [0,1]^d \to \mathbb{R}$$

can be approximated with a one hidden layer perceptron

$$x \mapsto \beta \cdot \sigma(wx + b)$$

where $b \in \mathbb{R}^{K \times d}$, $b \in \mathbb{R}^K$, and $\beta \in \mathbb{R}^K$.

K is the number of neurons in the hidden layer: it represents the width of the network.

This is the universal approximation theorem.

A better approximation requires a larger hidden layer (larger K), and this theorem says nothing about the relation between the two.

This theorem states that we can make the training error as low as we want by using a larger hidden layer. It indicates nothing about the test error.

Deploying MLP in practice is often a balancing act between underfitting and overfitting.

Some notations

Let us consider the training set $\mathcal{D}$:

$$\mathcal{D} = (x_i, y_i) \in \mathbb{R}^d \times \mathcal{Y}, i = 1, \cdots, m$$

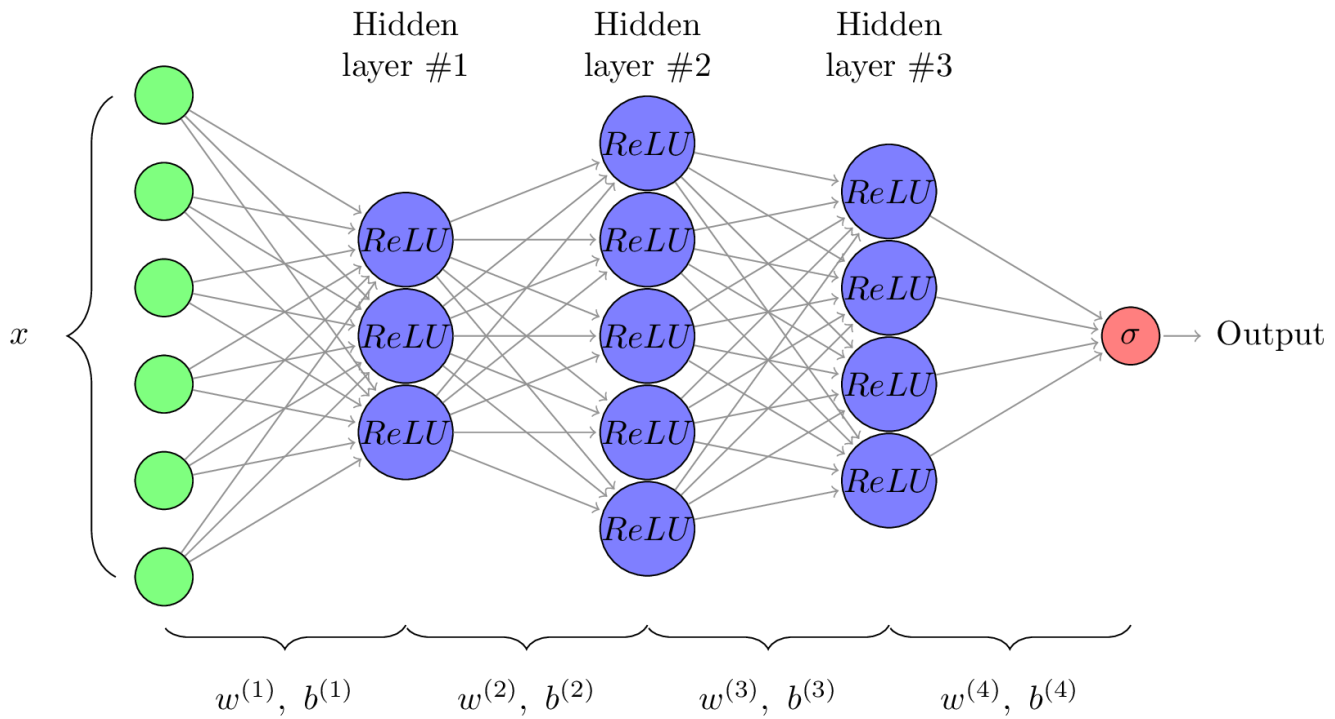
where m is the number of training instances (observations), and d the number of (explanatory) variable. y, the target variable, is also known as the label.

Recall:

- in a regression task $\mathcal{Y} = \mathbb{R}$
- in a classification task $\mathcal{Y} = \{0, \cdots, \mathcal{C} - 1\}$ where $\mathcal{C}$ is the number of class (for example, $\mathcal{C} = 2$ in a binary classification problem)

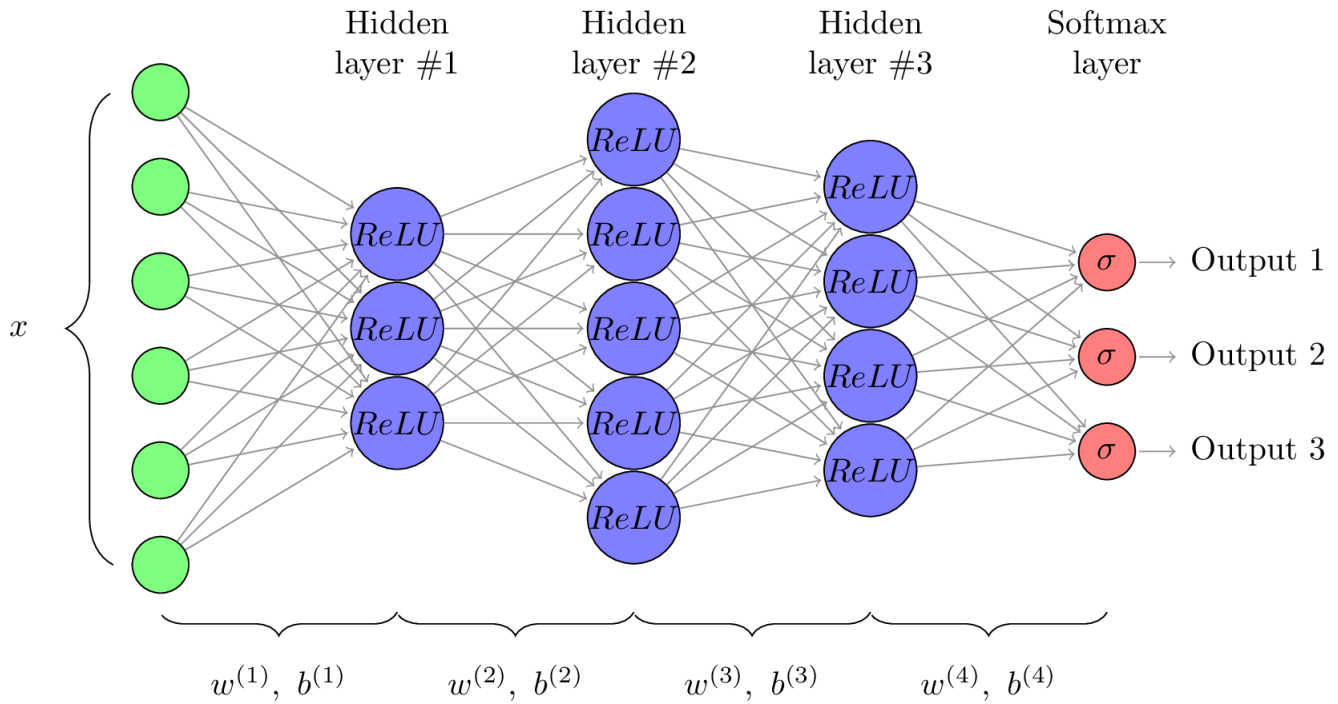
Output layer

- For a linear regression problem, a linear activation function is used.
- For a logistic regression problem, a logistic activation function is used, where $g = \sigma$ is a sigmoid function.



Output layer

- For a binary classification problem, a logistic activation function is also used.
- For a multiclass classification problem, a multi-class logistic activation function is used. It is known as the softmax regression.



where the number of outputs corresponds to the number of class $\mathcal{C}$.

Softmax regression

The softmax regression (also known as the multi-class logistic regression, multinomial logisitic or maximum entropy classifier) is the generalization of logistic regression, used for binary classification ($\mathcal{C} = 2$), for multi-class classification ($\mathcal{C} > 2$).

In softmax regression, the sigmoid function is replaced by the softmax function:

$$P(y = i|z^{(L)}(i)) = \frac{e^{z^{(L)}(i)}}{\sum_{j=0}^{\mathcal{C}-1} e^{z^{(L)}(j)}}$$

where $z^{(L)} = w^{(L)} x^{(L-1)} + b^{(L)}$.

It ouptus a vector that can be interpreted as class probability distribution vector:

$$\sum_{j=0}^{\mathcal{C}-1} P(y = j|z^{(L)}(i)) = 1$$

Soft-label versus hard-label

In a multiclass classification problem, the last layer L (namely the softmax layer) outputs a soft-label, i.e., a prediction $\hat{y}$ is a probabiltiy distribution vector ($\mathcal{C}$):

$$\hat{\mathbf{y}} = [0.2, 0.7, 0.1]$$

It can be easily converted into a hard-label by taking the $\arg\max$ of this vector. For the previous example,

$$\hat{y} = \arg\max \hat{\mathbf{y}} = 1$$

One-hot encoding

The use of the softmax regression also means that labels $Y \in \mathcal{Y}^n$ first need to be converted into a tensor $Z \in \mathbb{R}^{n \times c}$:

$$\forall i, z_{ij} = \begin{cases} 1 & \text{if } j = y_i \\ 0 & \text{otherwise} \end{cases}$$

Let us use our previous example

$$\begin{matrix} Y & & Z \\ & & \overset{\longleftarrow c \longrightarrow}{} \end{matrix}$$

$$\begin{pmatrix} 2 \\ 0 \\ 2 \\ 1 \\ 3 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Loss function

Loss function

The loss function (also known as cost function, objective function or error) measures the quality of a particuliar set of parameters based on how well a model performs on a given task.

- It is to guide the training process.
- It needs to be minimised.
- We denote it by $\mathcal{L}(y, \hat{y})$ where $\hat{y}$ is a prediction

Regression task

Most common loss:

- Mean Squarred Error (MSE), the most classical

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- Mean Absolute Error (MAE), penalises less errors on outliers (extremely small or large values far from the mean value)

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} |(y_i - \hat{y}_i)|$$

Classification task

Can we use the MSE loss for a classification problem?

Let us present the following example, where $n = 5$ and $\mathcal{C} = 4$:

$$y \qquad \hat{y}$$

$$\begin{pmatrix} 2 \\ 0 \\ 2 \\ 1 \\ 3 \end{pmatrix} \qquad \begin{pmatrix} 1 \\ 3 \\ 2 \\ 1 \\ 0 \end{pmatrix}$$

Classification task

Can we use the MSE loss for a classification problem?

Let us present the following example, where $n = 5$ and $\mathcal{C} = 4$:

$$\begin{array}{ccc} y & \hat{y} & (y - \hat{y})^2 \\ \begin{pmatrix} 2 \\ 0 \\ 2 \\ 1 \\ 3 \end{pmatrix} & \begin{pmatrix} 1 \\ 3 \\ 2 \\ 1 \\ 2 \end{pmatrix} & \begin{pmatrix} 1 \\ 9 \\ 0 \\ 0 \\ 1 \end{pmatrix} \end{array}$$

Does that make sense?

The MSE (or MAE) loss is conceptually wrong for classification: it penalises the difference between class number. MSE is justified with a Gaussian noise around a target value that makes sense geometrically.

$$y \qquad \hat{y} \qquad (y-\hat{y})^2$$

$$\begin{pmatrix} 2 \\ 0 \\ 2 \\ 1 \\ 3 \end{pmatrix} \qquad \begin{pmatrix} 1 \\ 3 \\ 2 \\ 1 \\ 2 \end{pmatrix} \qquad \begin{pmatrix} 1 \\ 9 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

In the example, we penalize more strongly, for no reason, an error between class 0 and class 3 than between class 2 and class 1.

Cross-entropy loss

In practice, the loss function of choice is the cross-entropy.

Given two distributions p and q, their cross-entropy loss is defined as:

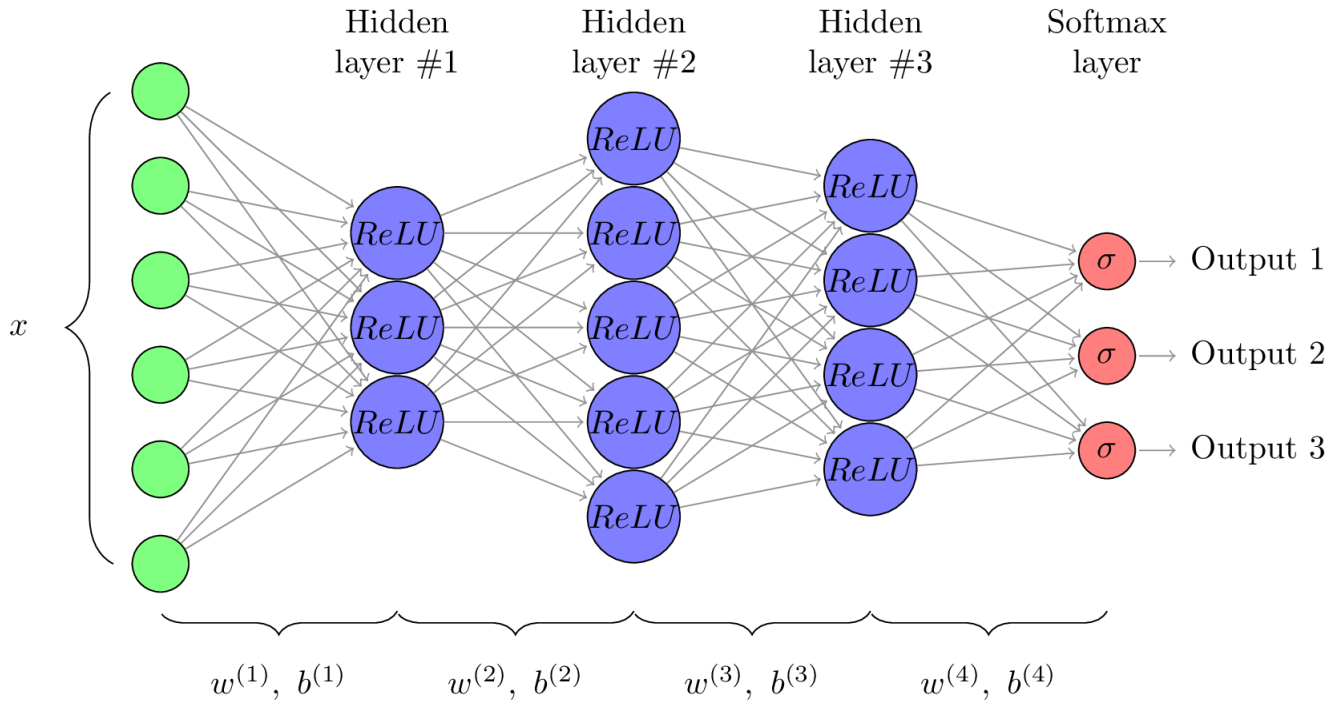
$$\mathcal{L}(p, q) = -\sum_{k} p(k) \cdot log\big(q(k)\big)$$

As is standard, $0 \cdot log(0) = 0$.

Number of trainable parameters

How to compute the number of trainable parameters?

Let us consider the following MultiLayer Perceptron network that can be used for solving a multiclass classification problem:



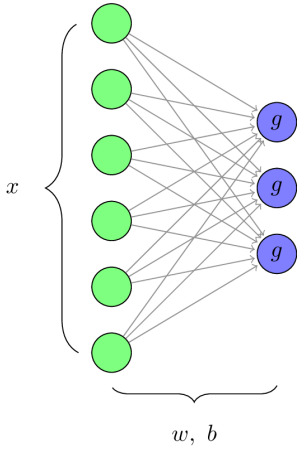
where $x \in \mathbb{R}^d$.

How to compute the number of trainable parameters?

Trainable parameters are: $(w^{(i)}, b^{(i)}), \forall i \in 1, ..., L$, where L is the number of layers in the network.

Let us denote by $K^{(i)}$ the number of neurons (also known as units) in layer i.

For the first hidden layer:



One neuron performs the following computation:

$$x^{(1)} = g(w^{(1)} \cdot x^{(0)} + b^{(1)})$$

where $x^{(0)} = x \in \mathbb{R}^d, w^{(0)} \in \mathbb{R}^d$ and $b^{(0)} \in \mathbb{R}$.
For $K^{(1)}$ neurons, operations are repeated $K^{(1)}$ times.
Consequently, $w^{(1)} \in \mathbb{R}^{K^{(1)} \times d}$ and $b^{(1)} \in \mathbb{R}^{K^{(1)}}$.

For the second hidden layer, the following operation is performed :

$$x^{(2)} = g(w^{(2)} \cdot x^{(1)} + b^{(2)})$$

where $w^{(2)} \in \mathbb{R}^{K^{(2)} \times K^{(1)}}$ and $b^{(2)} \in \mathbb{R}^{K^{(2)}}$.

For the second hidden layer, the following operation is performed :

$$x^{(2)} = g(w^{(2)} \cdot x^{(1)} + b^{(2)})$$

where $w^{(2)} \in \mathbb{R}^{K^{(2)} \times K^{(1)}}$ and $b^{(2)} \in \mathbb{R}^{K^{(2)}}$.

It can be generalized for any hidden layer l

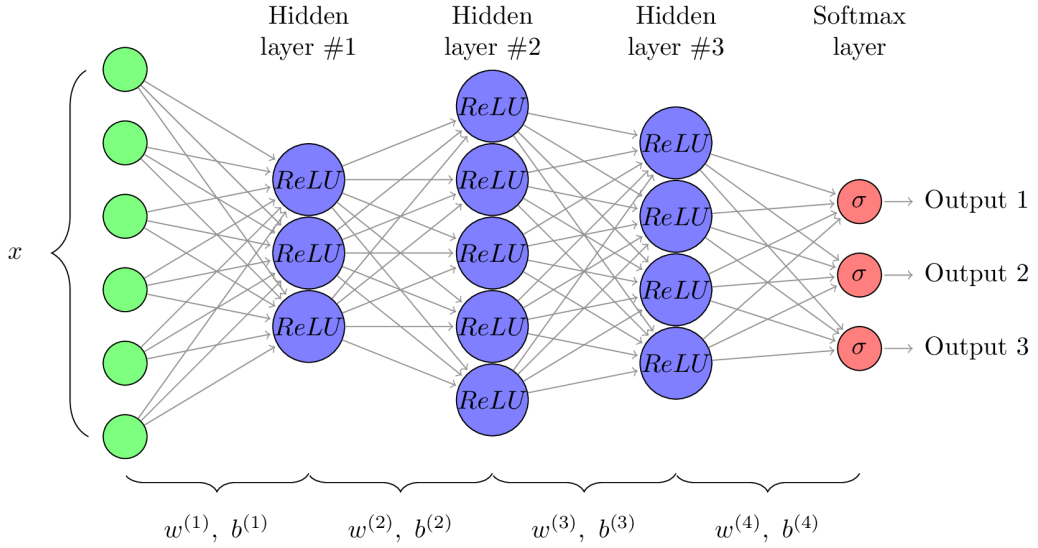
$$x^{(l)} = g(w^{(l)} \cdot x^{(l-1)} + b^{(l)})$$

where $w^{(l)} \in \mathbb{R}^{K^{(l)} \times K^{(l-1)}}$ and $b^{(l)} \in \mathbb{R}^{K^{(l)}}$.

There are two particular cases:

- $K^{(0)} = d$ the number of variables
- $K^{(L)} = \mathcal{C}$ the number of classes

With the initial example



where $x \in \mathbb{R}^6$.

The number of trainable parameters P is

$$P = K^{(1)} \cdot (d+1) + K^{(2)} \cdot (K^{(1)}+1) + K^{(3)} \cdot (K^{(2)}+1) + C \cdot (K^{(3)}+1)$$
$$P = 3 \cdot (6+1) + 5 \cdot (3+1) + 4 \cdot (5+1) + 3 \cdot (4+1)$$
$$P = 80$$

Going deeper

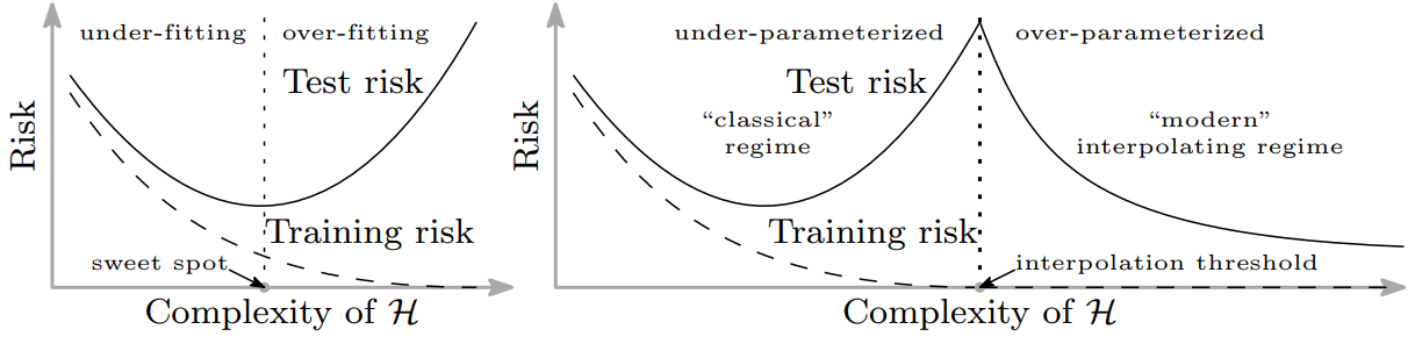
Going deeper

Using deeper architectures has been key in improving performance in many applications.

Going deeper

Using deeper architectures has been key in improving performance in many applications.

Regarding overfitting, over-parametrizing a deep model often improves test performance, contrary to what the bias-variance decomposition predicts (Belkin et al., 2018).



(a) U-shaped "bias-variance" risk curve

(b) "double descent" risk curve

Figure 1: Curves for training risk (dashed line) and test risk (solid line). (a) The classical *U-shaped risk curve* arising from the bias-variance trade-off. (b) The *double descent risk curve*, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high complexity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

Belkin, M., Hsu, D., Ma, S., & Mandal, S. (2018). Reconciling modern machine learning and the bias-variance trade-off. stat, 1050, 28.

There is thus good reasons to increase the network's depth, but an important issue then is to control the amplitude of gradient, which is tightly related to controlling activations.

In particular we have to ensure that

- the gradient does not "vanish"
- the gradient amplitude is homogeneous in order to ensure that all sections of the network are trained at the same rate
- the gradient does not vary too unpredictably when the weights change

Modern techniques change the functional itself instead of trying to improve training "from the outside" through penalty terms or better optimizers. For example, we move from the use of hyperbolic tangent to the use of ReLu as an activation function.

Our main concern is to make the gradient descent work, even at the cost of engineering substantially the class of functions.

An additional issue for training very large architectures is the computational cost, which often turns out to be the main practical problem.

Torch

Torch modules

- `torch.nn`, usually imported as nn, includes losses and network components, which embed parameters to be optimized during training.

An example with the cross entropy loss:

```
>>> yhat = torch.tensorr([[-1., -3., 4.], [-3., 3., -1.]])
>>> target = torch.tensor([0, 1])
>>> loss = nn.CrossEntropyLoss()
>>> loss(yhat, target)
tensor(2.5141)
```

Similarly, the mean squarred error loss can be computed with `nn.MSELoss()`.

```
>>> yhat = torch.tensor([[ 3., 0., 0., 0. ]])
>>> target = torch.tensor([[ 0., 0., 0., 0. ]])
>>> loss = nn.MSELoss()
>>> loss(yhat, target)
tensor(2.2500)
```

The first parameter of a loss is traditionally called the input annd the second the target. These two quantities may be of different dimensions or even types for some losses (see `CrossEntropyLoss`).

- `torch.nn.functional`, usually imported as `F`, includes autograd-compliant functions, which compute a result from provided arguments.

References

References

- François Fleuret's lectures: https://fleuret.org/ee559/