MACHINE LEARNING

Modeling with NN

Intro to learning with Neural Nets

Universidad Francisco de Vitoria
UFV Madrid

- Learning from data
- Learning as parameter estimation
- Differentiation and gradient descent
- Walking through a simple learning algorithm
- PyTorch *autograd*

MODELING WITH NN

- *Learning*

Learning though data

Algorithms not engineered to solve a particular problem.

- Instead, it is capable of approximating (fitting) the data using a much wider family of functions.

Key idea (supervised, deep learning):

Forward pass

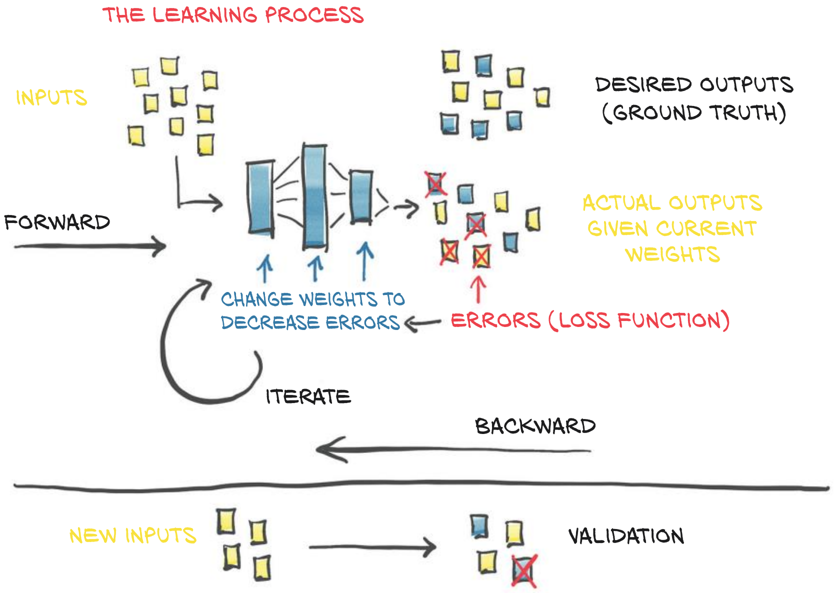
Backward pass

Optimization

Repetition Forward pass

Key idea (supervised, deep learning):

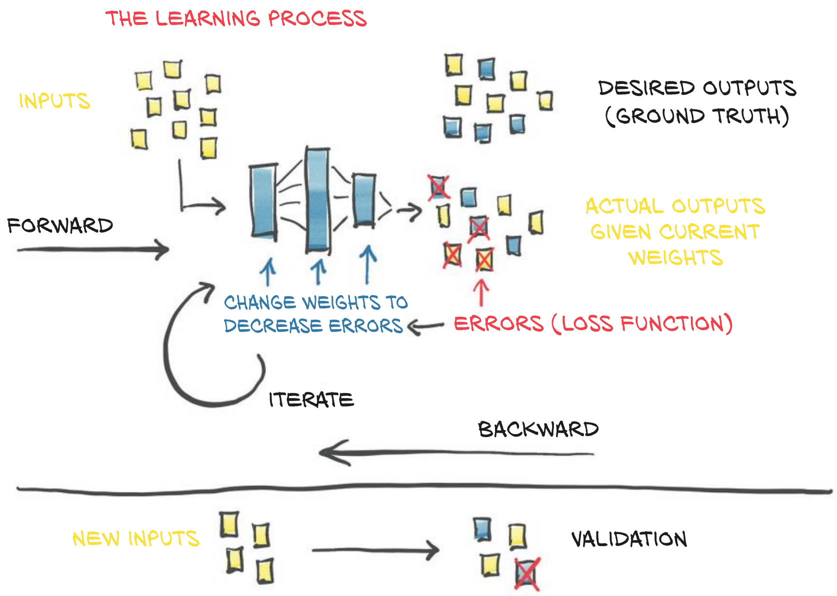
1. Forward pass
    - Given input data and the corresponding desired outputs (ground truth), as well as
    - initial values for the weights,
    - the model is fed input data (forward pass), and a measure of the error is evaluated by comparing the resulting outputs to the ground truth.
2. Backward pass
3. Optimization
4. Repetition



Learning though data

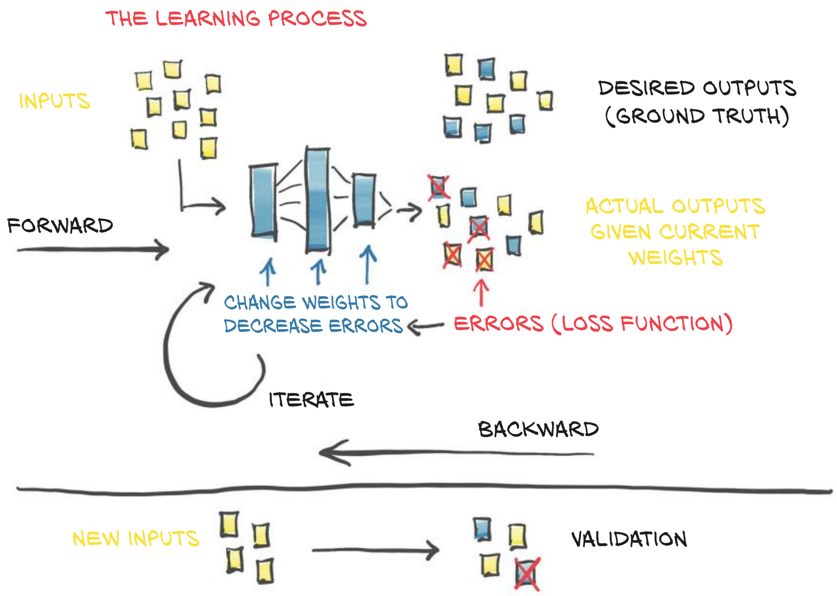
Key idea (supervised, deep learning):

1. Forward pass
2. Backward pass
    - In order to optimize the parameters of the model—its *weights and biases*—the change in the error following a unit change in weights (that is, the ***gradient*** **of the error with respect to the parameters**) is computed using the chain rule for the derivative of a composite function (backward pass).
3. Optimization
4. Repetition



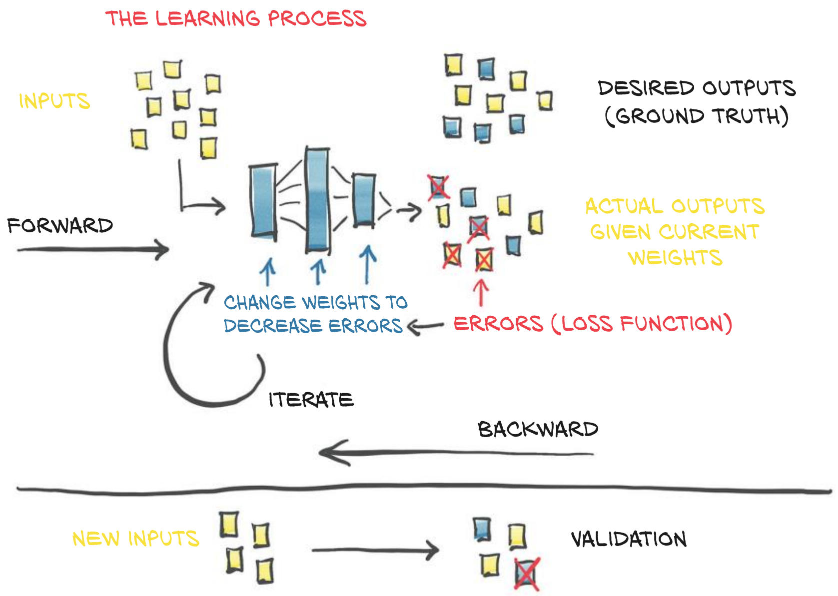
Key idea (supervised, deep learning):

1. Forward pass
2. Backward pass
3. Optimization
   - The value of the weights is then updated in the direction that leads to a decrease in the error.
4. Repetition



Key idea (supervised, deep learning):

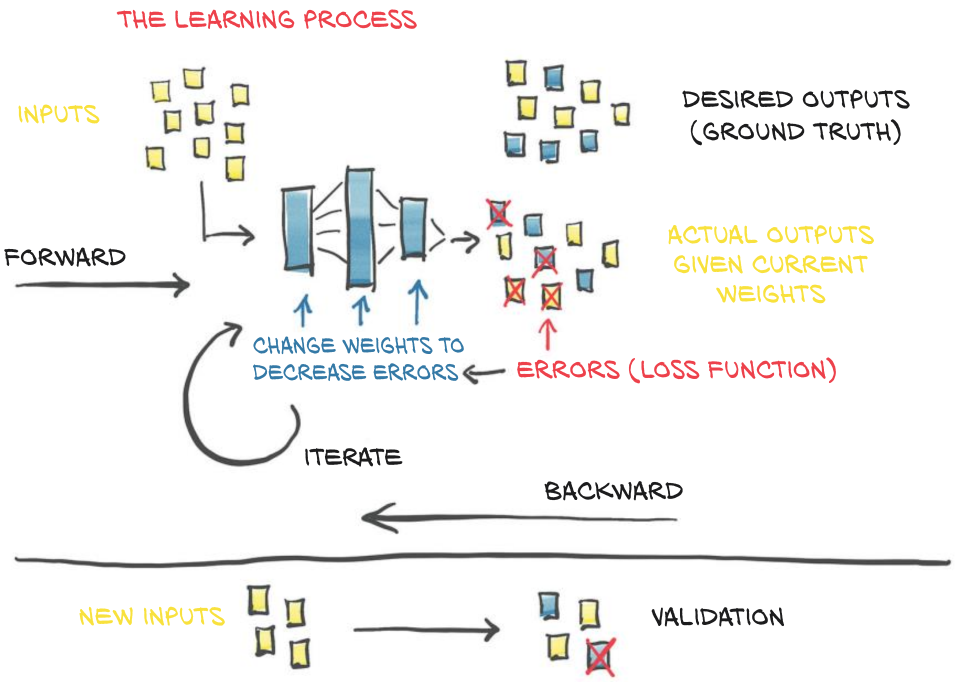
1. Forward pass
2. Backward pass
3. Optimization
4. Repetition
    - The procedure is repeated until the error, evaluated on unseen data, falls below an acceptable level.



Key idea (supervised, deep learning)

Key concepts:

- Labled data
- Forward pass
- Error
- Backward pass
- Update weights
- Iteration
- Validation

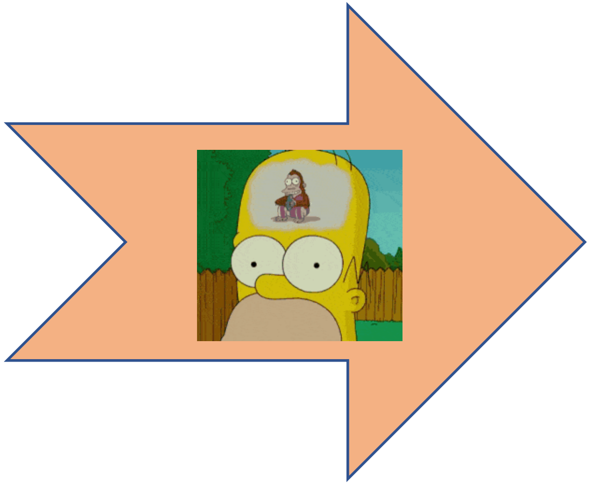


MODELING WITH NN

- *Learning*
- *Example: Simple Linear Regression*

Pretty straightforward example

Say we want to discover the equation that matches the following set of data

0.1		0.27
0.2		0.3
0.3		0.38
0.4		0.36
0.5		0.39
0.6		0.42

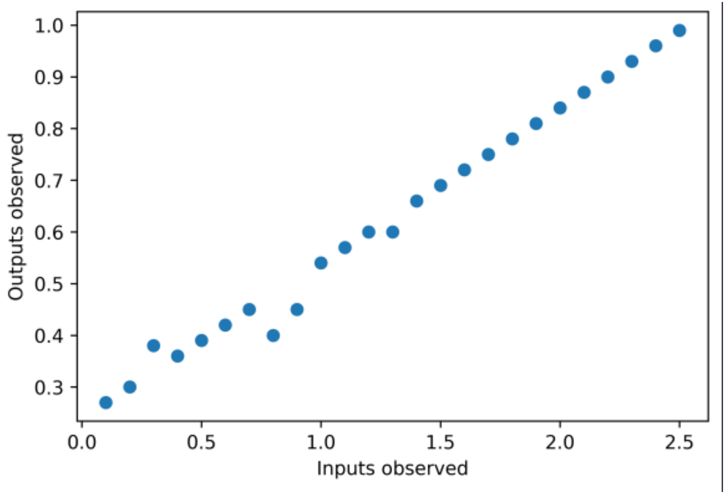
Pretty straightforward example

Inputs observed

[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2,2.1,2.2,2.3,2.4,2.5]

Outputs observed

[0.27,0.3,0.38,0.36,0.39,0.42,0.45,0.40,0.45,0.54,0.57,0.6,0.60,0.66,0.69,0.72,0.75,0.78,0.81,0.84,0.87,0.9,0.93,0.96,0.99]



It seems to me that it's a linear function

It seems that a simple linear function should fit our observations

Weight Bias

$$y = w\,x + b$$

Output Input

Figuring out a model

Simplest model, linear. We want to apply this model on our set of inputs/outputs.

$m(x, w, b)$ returns y

```
def model(x, w, b):
    y = w * x + b
    return y
```

Training a neural network will essentially involve changing the model for a slightly more elaborate one, with a few (or millions) more parameters.

We'll go through this simple example using PyTorch.

The model has a weight w and a bias b that we want to estimate.

ml/temperature_estimation/temp_model_unscaled.py

Less loss is what we want

Our optimization process aims at finding w and b so that the loss function is at a minimum.

What is a Loss Function????

The Loss Function describes de difference between the set of values predicted by our model ($\overline{y}$) or y_hat and the set o values observed as outputs (y)

Convention in ML community

$\overline{y}$ =values of our model we want to fit (labels, y_hat)

y = observed values

$$L(\overline{y}, y) = |\overline{y} - y|$$

$$L(y) = (\overline{y} - y)^2$$

MSE (mean squared error) -> widely used

Less loss is what we want

```
def loss_fn(y_hat, y):
    squared_diffs = (y_hat - y)**2
    mse = squared_diffs.mean()
    return mse
```

As less loss as possible

Computing the loss

Initialize weights and calculate predicted value y

```
w = torch.ones(())
b = torch.zeros(())

y = model(x, w, b)
```

Remember

```
def model(x, w, b):
y = w * x + b
return y
```

Calculate loss:

Note that this returns a tensor.

```
loss = loss_fn(y_hat, y)
loss
```

```
IPdb [2]: squared_diffs
Out  [2]:
tensor([0.0035, 0.0161, 0.0209, 0.0687, 0.1088, 0.1580, 0.2164, 0.3757,
        0.4365, 0.4467, 0.5418, 0.6460, 0.8126, 0.8820, 1.0138, 1.1547,
        1.3048, 1.4641, 1.6325, 1.8101, 1.9968, 2.1928, 2.3979, 2.6121,
        2.8356])

IPdb [3]: mse
Out  [3]: tensor(1.0059)

IPdb [4]:
```

MODELING WITH NN

- *Learning*
- *Example: Simple Linear Regression*
- *Gradient descent*

Minimize our loss function

Reminder!: What do we wan to figure out ?

We want to figure out the set of (w,b) that fits our linear equation $\overline{y} = w * x + b$ to our set of observations (labels, or y)

We will use our loss function to achieve this target, the less values got by our loss function, the better.

We will try to find the set of (w,b) that minimizes our loss function

Universidad Francisco de Vitoria
UFV Madrid

We tackle three calculus challenges in order to reach our target

Two parameters to change when minimizing the loss function, not jut one -> w and b	→	**Partial Derivatives**
Our loss function is not directly connected to our parameters	→	**Chain Rule**
We want to write and algorithm that manages to find the parameters that minimizes our values	→	**Gradient!!**

Calculus provide us a tool to find minimum values

Derivatives

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Compute a derivative

Universidad Francisco de Vitoria
UFV Madrid

Given: $$\boldsymbol{y = (x - 3)^2}$$

Compute it derivative: $\dfrac{dy}{dx}$

Reminder, partial derivatives

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, and an equation

$$y = f(z^1, z^2, \ldots, z^n)$$

where y and z^i for $i = 1 \ldots n$ are scalars, we will write the partial derivative of the function f with respect to the variable z^i as

$$\frac{\partial f(z^1 \ldots z^n)}{\partial z^i}$$

Definition of a partial derivative: this is the derivative of f with respect to parameter z^i, holding the other arguments to f (i.e., z^j for $j \neq i$) fixed. Or more precisely,

$$\frac{\partial f(z^1, \ldots, z^n)}{\partial z^i} = \lim_{h \to 0} \frac{f\left(z^1 \ldots, z^i + h, \ldots z^n\right) - f(z^1, \ldots, z^n)}{h}$$

which can be read as “the partial derivative of y with respect to z^i , under function f, at values $z^1, \ldots, z^n$.”

Reminder: chain rule

Universidad Francisco de Vitoria
UFV Madrid

Say we have the following function: $L(y) = (\overline{y} - \mathrm{y})^2$

But y is another function that has our parameters we want to minimize: $\overline{y} = w \cdot x + b$

Then if we want to calculate the derivative of the loss function with respect to w

$$\frac{\partial L(\overline{y})}{\partial w} = \frac{\partial L}{\partial \overline{y}} \cdot \frac{\partial \overline{y}}{\partial w}$$

$$\frac{\partial L(\overline{y})}{\partial b} = \frac{\partial L}{\partial \overline{y}} \cdot \frac{\partial \overline{y}}{\partial b}$$

Reminder, gradient

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, and an equation

$$y = f(z^1, z^2, \dots, z^n)$$

where y and z^i for $i = 1 \dots n$ are scalars, the gradient of f, ∇f, the gradient of f at a point $p = (z^1, z^2, \dots, z^n)$

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f(z^1, \dots z^n)}{\partial z^1} \\ \dots \\ \frac{\partial f(z^1, \dots z^n)}{\partial z^n} \end{bmatrix}$$

Note that $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ defined at point $p = (z^1, z^2, \dots, z^n)$ and returning an n-dimensional vector

MODELING WITH NN

- *Learning*
- *Example: RTD*
- *Gradient descent*

Gradient descent

Multi-variable function $f(\boldsymbol{z})$ is defined and differentiable in a neighborhood of a point $\boldsymbol{z}$, then $f(\boldsymbol{z})$ decreases fastest if one goes from $\boldsymbol{z}$ in the direction of the negative gradient of f at $\boldsymbol{z}$, $-\nabla f(\boldsymbol{z})$.

Given a value $\boldsymbol{z}_0$ it follows that, for a small $\gamma \in \mathbb{R}^+$, then if we perform the update:

$$z_1 = z_0 - \gamma \nabla f(z_0)$$

If we perform multiple updates:

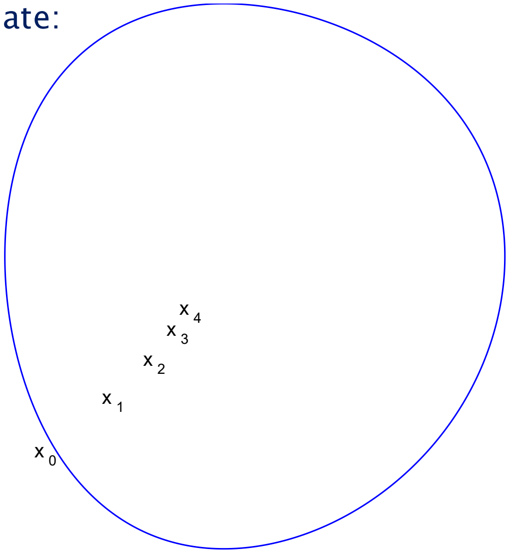
$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n - \gamma \nabla f(\boldsymbol{z}_n)$$

We update a monotonically decreasing function:

$$\boldsymbol{f}(\boldsymbol{z}_0) \geq f(\boldsymbol{z}_1) \geq f(\boldsymbol{z}_2) \geq f(\boldsymbol{z}_3) \dots$$

…hopefully, the sequence converges to the desired local minimum.

$\boldsymbol{z}_1 = \boldsymbol{z}_0 - \gamma \nabla f(\boldsymbol{z}_0)$ then $f(\boldsymbol{z}_1) \leq \boldsymbol{f}(\boldsymbol{z}_0)$.



Lines are level sets (function takes constant value)

Gradient descent algorithm is the key takeaway

We'll optimize the loss function with respect to the parameters using the *gradient descent* algorithm.

Compute the rate of change of the loss with respect to each parameter; and

Modify each parameter in the direction of decreasing loss (we can do it because we understand what a gradient represents

Guys!, We've got all the tools!

Computing our Gradient Vector (I)

Universidad Francisco de Vitoria
UFV Madrid

$$\frac{\partial L(\overline{y})}{\partial w} = \frac{\partial L}{\partial \overline{y}} \cdot \frac{\partial \overline{y}}{\partial w} \quad \left\{ \begin{array}{l} \dfrac{\partial L}{\partial \overline{y}} = \dfrac{\partial (\overline{y} - \mathrm{y})^2}{\partial \overline{y}} = \dfrac{\partial (\overline{y}^2 - 2y\overline{y} + \mathrm{y}^2)}{\partial \overline{y}} = 2\overline{y} - 2\mathrm{y} = 2(\overline{y} - \mathrm{y}) \\[2ex] \dfrac{\partial \overline{y}}{\partial w} = \dfrac{\partial (w \cdot x + b)}{\partial w} = \dfrac{\partial (w \cdot x) + \partial (b)}{\partial w} = x \end{array} \right.$$

$$\boldsymbol{\frac{\partial L(y)}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w} = 2(\overline{y} - \mathrm{y}) \cdot x}$$

Computing our Gradient Vector (II)

$$\frac{\partial \mathrm{L}(\overline{y})}{\partial b} = \frac{\partial L}{\partial \overline{y}} \cdot \frac{\partial \overline{y}}{\partial b} \quad \left\{ \begin{array}{l} \dfrac{\partial L}{\partial \overline{y}} = \dfrac{\partial (\overline{y} - \mathrm{y})^2}{\partial \overline{y}} = \dfrac{\partial (\overline{y}^2 - 2y\overline{y} + \mathrm{y}^2)}{\partial \overline{y}} = 2\overline{y} - 2\mathrm{y} = 2(\overline{y} - \mathrm{y}) \\ \dfrac{\partial y}{\partial b} = \dfrac{\partial (\mathrm{w \cdot x + b})}{\partial b} = \dfrac{\partial (\mathrm{w} \cdot x) + \partial (\mathrm{b})}{\partial b} = 1 \end{array} \right.$$

$$\frac{\partial L(\overline{y})}{\partial w} = \frac{\partial L}{\partial \overline{y}} \cdot \frac{\partial \overline{y}}{\partial w} = 2(\overline{y} - \mathrm{y})$$

Computing Gradient Vector (IV)

The gradient vector has two components (2-dimensional vector)

$$\boldsymbol{\nabla L(y) = [2(\overline{y} - y)] \cdot x, 2(\overline{y} - y)]}$$

Code your own functions

Code in Python the following functions:

- A functions that, given a weight w and a bias b, returns the función $y = ax + b$
- Write a function that, given a function $y = wx + b$, returns its gradient vector

Coding functions

Universidad Francisco de Vitoria
UFV Madrid

$$\frac{\partial L}{\partial y} = 2(\overline{y} - y)$$

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

$$\frac{\partial \overline{y}}{\partial b} = 1$$

```
def dloss_fn(y_hat, y):
    dLoss_y = 2 * (y_hat - y)
    return dLoss_y
```

```
def dmodel_dw(x, w, b):
    return x
```

```
def dmodel_db(x, w, b):
    return 1.0
```

```
def grad_fn(x, y, y_hat, w, b):
    dloss_dtp = dloss_fn(y_hat, y)
    dloss_dw = dloss_dtp * dmodel_dw(x, w, b)
    dloss_db = dloss_dtp * dmodel_db(x, w, b)
    return torch.stack([dloss_dw.sum()/ y_hat.size(0), dloss_db.sum()/ y_hat.size(0)])
```

Computing Gradient Vector (IV)

Universidad Francisco de Vitoria
UFV Madrid

$$\nabla L(y) = [2(\bar{y} - y)] \cdot x, 2(\bar{y} - y)]$$

```
def grad_fn(x, y, y_hat, w, b):
    dloss_dtp = dloss_fn(y_hat, y)
    dloss_dw = dloss_dtp * dmodel_dw(x, w, b)
    dloss_db = dloss_dtp * dmodel_db(x, w, b)
    return torch.stack([dloss_dw.sum()/ y_hat.size(0), dloss_db.sum()/ y_hat.size(0)])
```

Computing Gradient Vector (V)

Universidad Francisco de Vitoria
UFV Madrid

Ey Ey Ey Ey... Why do you include sum() and divide by my_model.size() ??

```
def grad_fn(x, y, y_hat, w, b):
    dloss_dtp = dloss_fn(y_hat, y)
    dloss_dw = dloss_dtp * dmodel_dw(x, w, b)
    dloss_db = dloss_dtp * dmodel_db(x, w, b)
    return torch.stack([dloss_dw.sum()/ y_hat.size(0), dloss_db.sum()/ y_hat.size(0)])
```

Reminder: we have this observations

Inputs observed

[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2,2.1,2.2,2.3,2.4,2.5]

Outputs observed

[0.27,0.3,0.38,0.36,0.39,0.42,0.45,0.40,0.45,0.54,0.57,0.6,0.60,0.66,0.69,0.72,0.75,0.78,0.81,0.84,0.87,0.9,0.93,0.96,0.99]

Therefore, what we are going to calculate is the mean of the gradient in the points we have observed!

$$\nabla L(y) = \left[\frac{\sum_{k=1}^{N}\left[\left(\overline{y}_k - y_k\right) \cdot x_k\right]}{N}, \frac{\sum_{k=1}^{N}\left(\overline{y}_k - y_k\right)}{N}\right]$$

N = number of observations

Computing the Gradient Descent

Given a set of values w_0, b_0 it follows that, for a small $\gamma \in \mathbb{R}^+$, then if we perform the update:

$$(w_1, b_1) = (w_0, b_0) - \gamma \cdot \nabla y(w_0, b_0)$$

```
w, b = params

y = model(x_hat, w, b)

gradient_vector = grad_fn(x_hat, y_hat, y, w, b)

params = params - learning_rate * gradient_vector

loss = loss_fn(y, y_hat)
losses.append(loss)
```

$$\gamma = learning\ rate$$

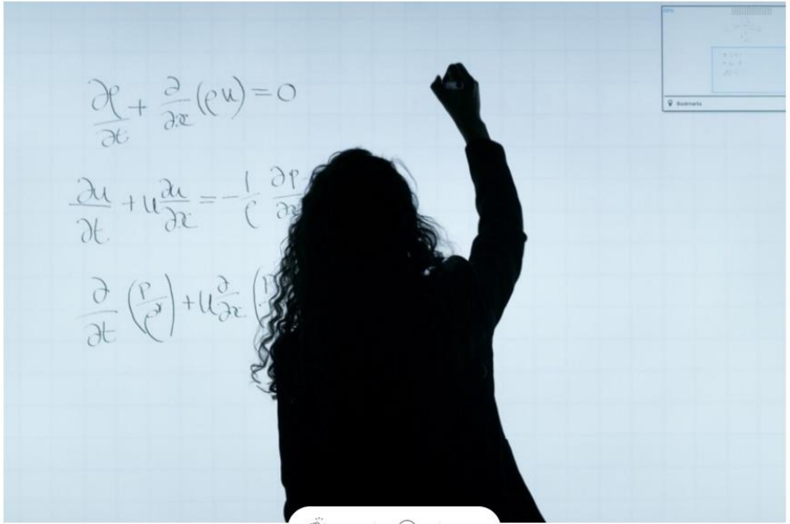
https://towardsdatascience.com/the-gradient-vector-66ad563ab55a

Roman Paolucci

Jun 5, 2020 · 4 min read · Member-only · Listen

The Gradient Vector

What is it, and how do we compute it?



98

MODELING WITH NN

- *Learning*
- *Example: RTD*
- *Gradient descent*
- *A training loop*

Training loop

All in place to start optimizing parameters.

Iterate a fixed number of iterations, or until paramters don't change (enough).

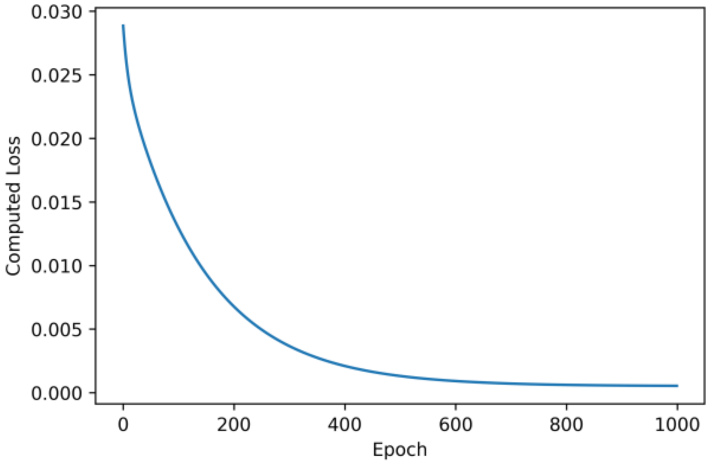
Epoch: an interation over the dataset.

```
def training_loop(n_epochs, learning_rate, params, x, y, print_params=True):

    for epoch in range(1, n_epochs + 1):

        # Params we want to fit
        w, b = params

        # Setting the model we will use (a simple linear equation)
        y_hat = model(x, w, b)

        # Setting our gradient vector
        gradient_vector = grad_fn(x, y, y_hat, w, b)

        # Setting our Gradient Descent step
        params = params - learning_rate * gradient_vector

        loss = loss_fn(y_hat, y)
        losses.append(loss)
```

1. Setting our params
2. Setting our model
3. Setting our Gradient Vector
4. Gradient Descent step

Let’s run our algorithm

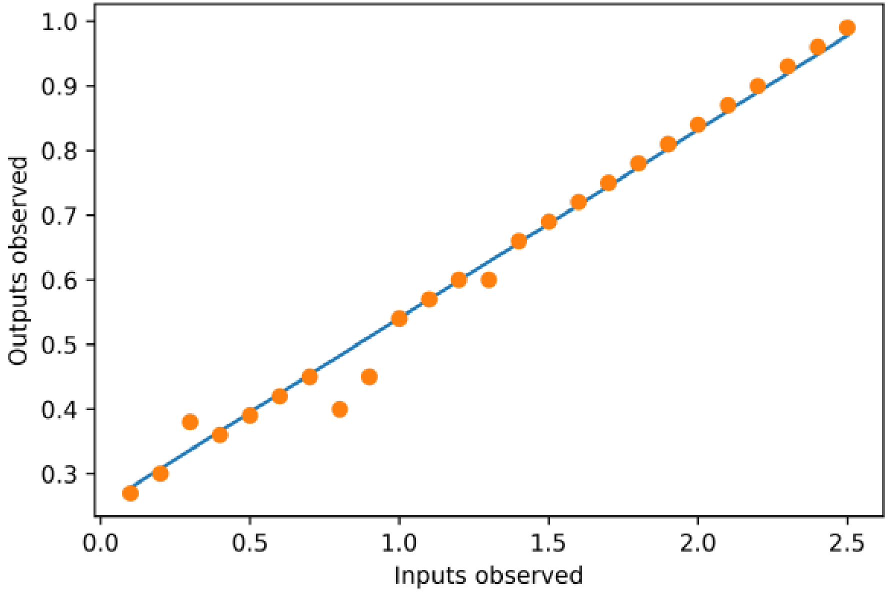

```
Epoch 1, Loss 0.020676
    Params: tensor([0.3936, 0.2293])
    Grad:   tensor([0.4222, 0.2474])
    Loss:   tensor(0.0207)
Epoch 2, Loss 0.018354
    Params: tensor([0.3896, 0.2270])
    Grad:   tensor([0.3971, 0.2315])
    Loss:   tensor(0.0184)
Epoch 3, Loss 0.016306
    Params: tensor([0.3859, 0.2248])
    Grad:   tensor([0.3735, 0.2165])
    Loss:   tensor(0.0163)
...
Epoch 10, Loss 0.007339
    Params: tensor([0.3652, 0.2131])
    Grad:   tensor([0.2441, 0.1345])
    Loss:   tensor(0.0073)
Epoch 11, Loss 0.006586
    Params: tensor([0.3629, 0.2119])
    Grad:   tensor([0.2298, 0.1254])
    Loss:   tensor(0.0066)
...
Epoch 99, Loss 0.000765
    Params: tensor([0.3218, 0.2016])
    Grad:   tensor([ 0.0079, -0.0105])
    Loss:   tensor(0.0008)
Epoch 100, Loss 0.000763
    Params: tensor([0.3217, 0.2017])
    Grad:   tensor([ 0.0079, -0.0105])
    Loss:   tensor(0.0008)
...
Epoch 1000, Loss 0.000513
    Params: tensor([0.3020, 0.2323])
    Grad:   tensor([ 0.0003, -0.0005])
    Loss:   tensor(0.0005)
    Params: tensor([0.3020, 0.2323])
```



Check out the results

```
Our model estimates a w = 0.291 and b = 0.249

Our model predicts for the input 2.800 the output 1.065
```

It seems that our algorithm has found a model that fit our observations



Trying fit other linear regressions

- Try another set of observations
- Load your observations into the model
- Did the algorithm manage to converge?

What have we acheived

Just a new way of computing parameters of a linear regression… but:

This process can be applied to any model that depends on parámeters, as long as all functions can be differentiated analytically.

The only issue is that if we change the model, we need to recompute the gradient of the loss with respect to the parametersm applying the chain rule.

… this is where ***autograd*** and ***backpropagation*** comes in.

What have we acheived

Tensors, computational graphs and automatic differentiation

Until now, tensors were just multidimensional arrays (plus divice).

But tensors objects are designed to be the building block of **computational graphs**, and to **support automatic differentiation**.

Nodes of computational graphs, connected to other nodes through functions (functional module instances)

Three key attributes:

- data
- grad
- grad_fn