IfI Summer School 2018 on Machine Learning

Deep Learning #3 – MLP, backprop, autograd

François Fleuret
http://fleuret.org/ifi/

June 26, 2018

A bit of history, the perceptron

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs:

$$f(x) = \mathbf{1}_{\{w \sum_i x_i + b \geq 0\}}.$$

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs:

$$f(x) = \mathbf{1}_{\{w \sum_i x_i + b \geq 0\}}.$$

It can in particular implement

- $or(a, b) = \mathbf{1}_{\{a+b-0.5 \geq 0\}}$
- $and(a, b) = \mathbf{1}_{\{a+b-1.5 \geq 0\}}$
- $not(a) = \mathbf{1}_{\{-a+0.5 \geq 0\}}$

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs:

$$f(x) = \mathbf{1}_{\{w \sum_i x_i + b \geq 0\}}.$$

It can in particular implement

- $or(a, b) = \mathbf{1}_{\{a+b-0.5 \geq 0\}}$
- $and(a, b) = \mathbf{1}_{\{a+b-1.5 \geq 0\}}$
- $not(a) = \mathbf{1}_{\{-a+0.5 \geq 0\}}$

Hence, **any Boolean function can be build with such units.**

(McCulloch and Pitts, 1943)

The perceptron is very similar

$$f(x) = \begin{cases} 1 & \text{if} \quad \sum_i w_i \, x_i + b \geq 0 \\ 0 & \quad\quad\quad\quad \text{otherwise} \end{cases}$$

but the inputs are real values and the weights can be different.

The perceptron is very similar

$$f(x) = \begin{cases} 1 & \text{if} \quad \sum_i w_i \, x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

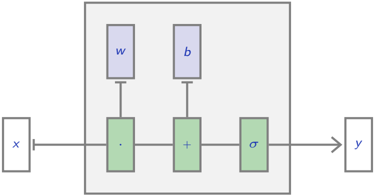
but the inputs are real values and the weights can be different.

This model was originally motivated by biology, with w_i being the *synaptic weights*, and x_i and f firing rates.

It is a (very) crude biological model.

(Rosenblatt, 1957)

We can represent this as



Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, 1\}, \quad n = 1, \dots, N,$$

a very simple scheme to train such a linear operator for classification is the **perceptron algorithm:**

Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, 1\}, \quad n = 1, \dots, N,$$

a very simple scheme to train such a linear operator for classification is the **perceptron algorithm:**

1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k}\left(w^k \cdot x_{n_k}\right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, 1\}, \quad n = 1, \dots, N,$$

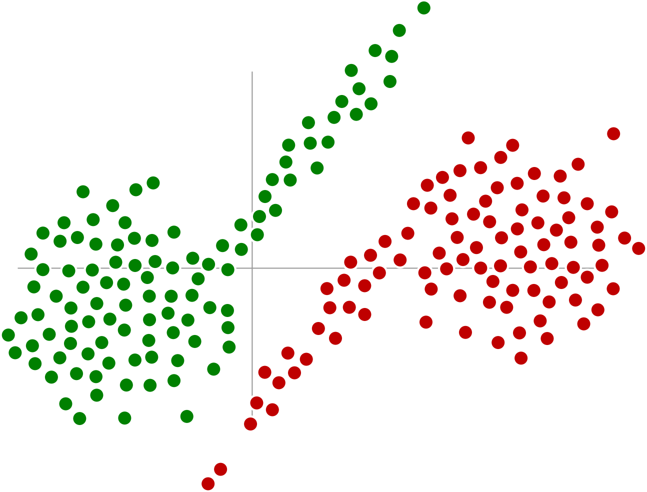
a very simple scheme to train such a linear operator for classification is the **perceptron algorithm:**

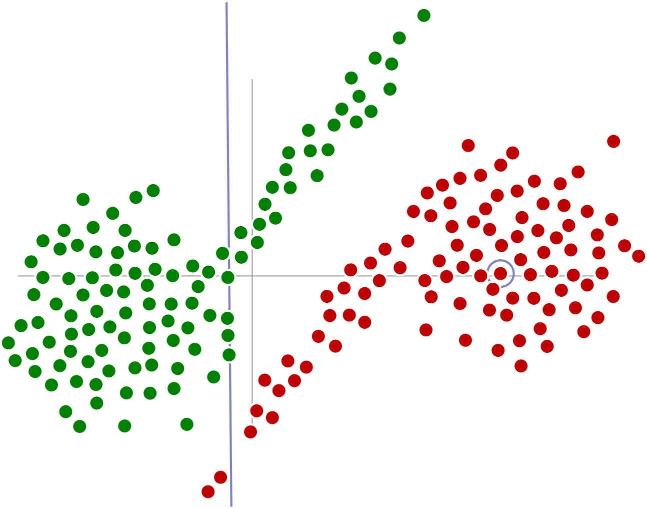
1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k}\left(w^k \cdot x_{n_k}\right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

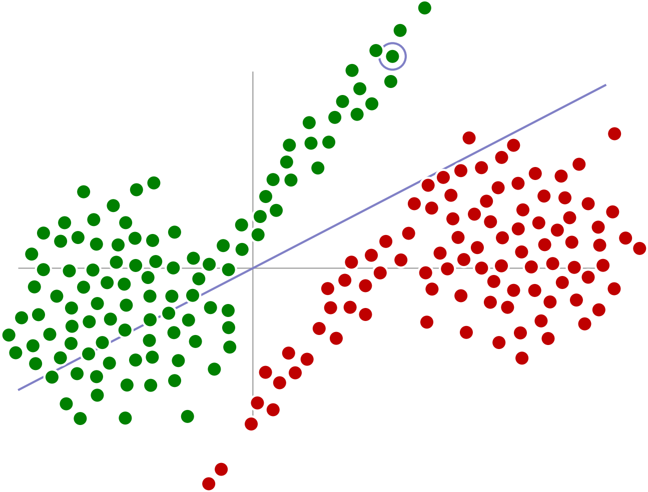
The bias b can be introduced as one of the ws by adding a constant component to x equal to 1.

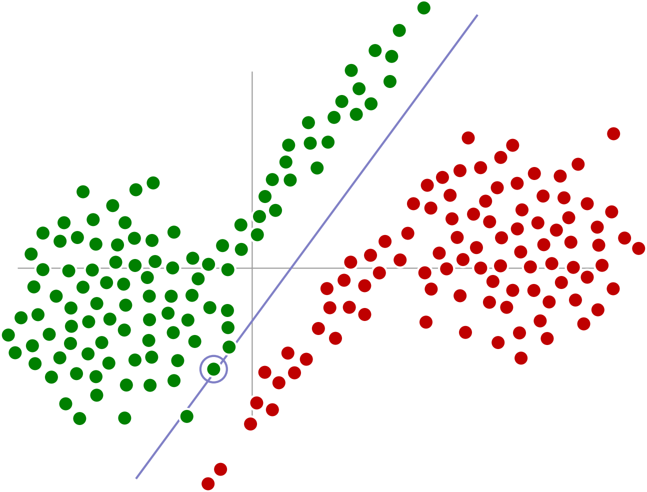
(Rosenblatt, 1957)

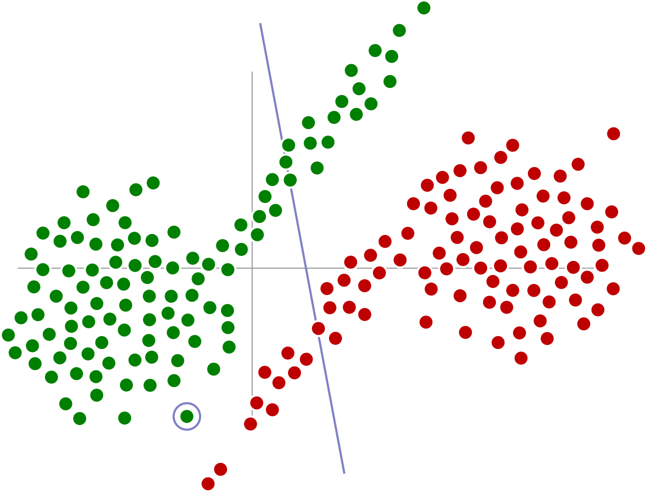
```
def train_perceptron(x, y, nb_epochs_max):
    w = torch.zeros(x.size(1))

    for e in range(nb_epochs_max):
        nb_changes = 0
        for i in range(x.size(0)):
            if x[i].dot(w) * y[i] <= 0:
                w = w + y[i] * x[i]
                nb_changes = nb_changes + 1
        if nb_changes == 0: break;

    return w
```

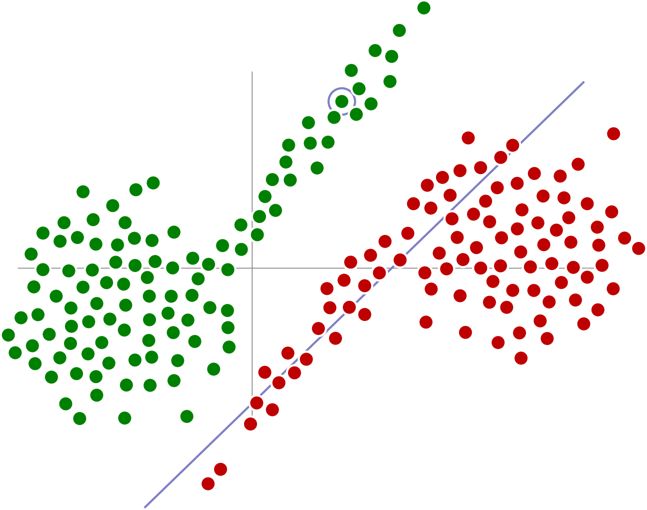


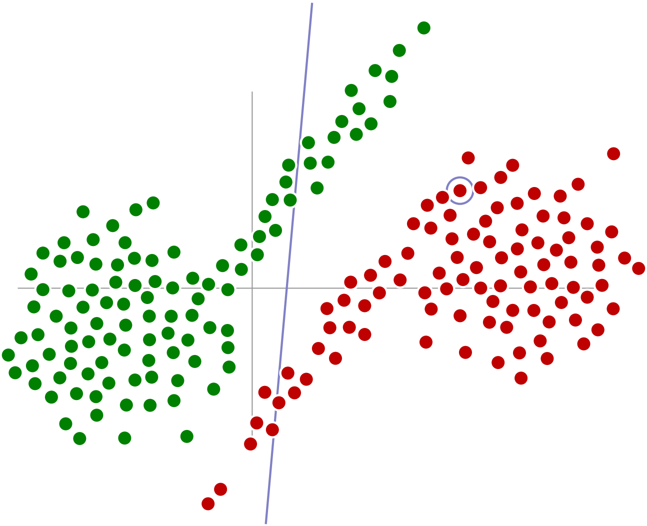


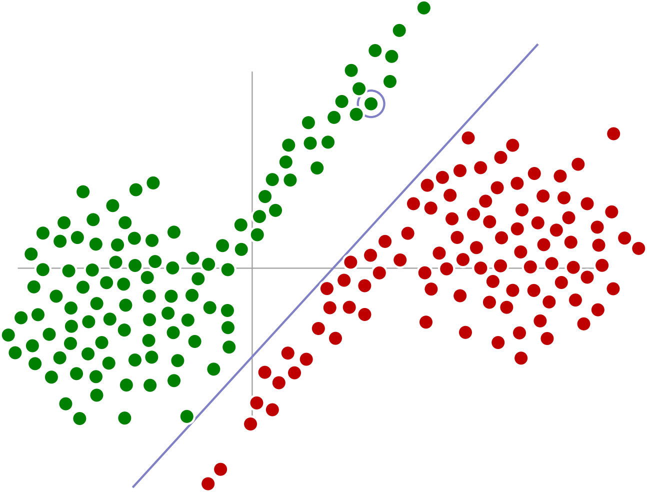




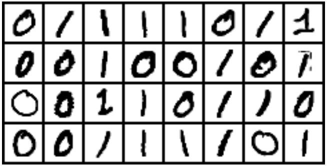




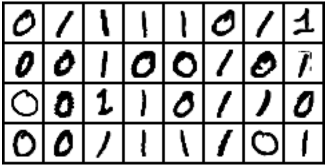




This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.

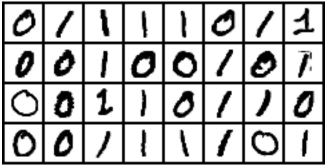


This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.

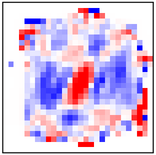


```
epoch 0 nb_changes 64 train_error 0.23%% test_error 0.19%%
epoch 1 nb_changes 24 train_error 0.07%% test_error 0.00%%
epoch 2 nb_changes 10 train_error 0.06%% test_error 0.05%%
epoch 3 nb_changes 6 train_error 0.03%% test_error 0.14%%
epoch 4 nb_changes 5 train_error 0.03%% test_error 0.09%%
epoch 5 nb_changes 4 train_error 0.02%% test_error 0.14%%
epoch 6 nb_changes 3 train_error 0.01%% test_error 0.14%%
epoch 7 nb_changes 2 train_error 0.00%% test_error 0.14%%
epoch 8 nb_changes 0 train_error 0.00%% test_error 0.14%%
```

This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.



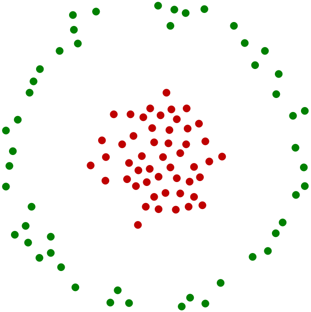
```
epoch 0 nb_changes 64 train_error 0.23%% test_error 0.19%%
epoch 1 nb_changes 24 train_error 0.07%% test_error 0.00%%
epoch 2 nb_changes 10 train_error 0.06%% test_error 0.05%%
epoch 3 nb_changes 6 train_error 0.03%% test_error 0.14%%
epoch 4 nb_changes 5 train_error 0.03%% test_error 0.09%%
epoch 5 nb_changes 4 train_error 0.02%% test_error 0.14%%
epoch 6 nb_changes 3 train_error 0.01%% test_error 0.14%%
epoch 7 nb_changes 2 train_error 0.00%% test_error 0.14%%
epoch 8 nb_changes 0 train_error 0.00%% test_error 0.14%%
```



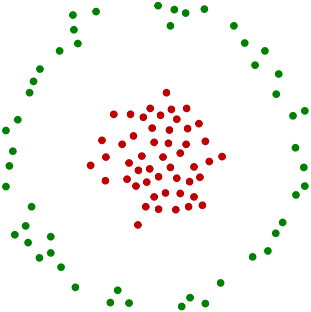
Limitation of linear classifiers

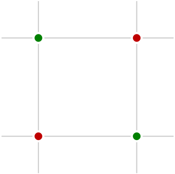
The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.

The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.



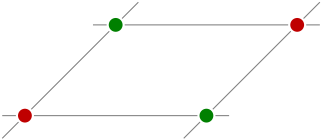
The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.





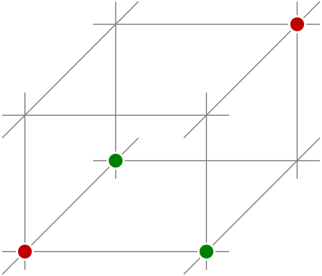
"xor"

The xor example can be solved by pre-processing the data to make the two populations linearly separable:



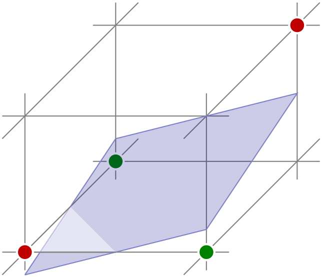
The xor example can be solved by pre-processing the data to make the two populations linearly separable:

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$



The xor example can be solved by pre-processing the data to make the two populations linearly separable:

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$



So we can model the xor with

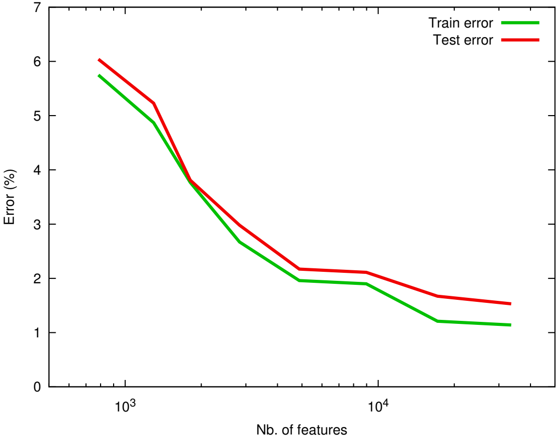
$$f(x) = \sigma(w\,\Phi(x) + b).$$

We can apply the same to a more realistic binary classification problem: MNIST's "8" vs. the other classes with a perceptron.

The original 28×28 features are supplemented with the products of pairs of features taken at random.

We can apply the same to a more realistic binary classification problem: MNIST's "8" vs. the other classes with a perceptron.

The original 28×28 features are supplemented with the products of pairs of features taken at random.



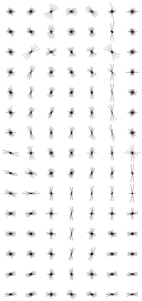
Beside increasing capacity to reduce the bias, “feature design” may also be a way of reducing capacity without hurting the bias, or with improving it.

Beside increasing capacity to reduce the bias, “feature design” may also be a way of reducing capacity without hurting the bias, or with improving it.

In particular, good features should be invariant to perturbations of the signal known to keep the value to predict unchanged.

A classical example is the “Histogram of Oriented Gradient” descriptors (HOG), initially designed for person detection.

Roughly: divide the image in 8×8 blocks, compute in each the distribution of edge orientations over 9 bins.



Dalal and Triggs (2005) combined them with a linear predictor, and Dollár et al. (2009) extended them with other modalities into the “channel features”.

Training a model composed of manually engineered features and a parametric model is now referred to as **"shallow learning"**.

The signal goes through a single processing trained from data.

A core notion of "Deep Learning" is precisely to avoid this dichotomy and to rely on ["deep"] sequences of processing with limited hand-designed structures.

Multi-Layer Perceptron

We can combine several “layers”. With $x^{(0)} = x$,

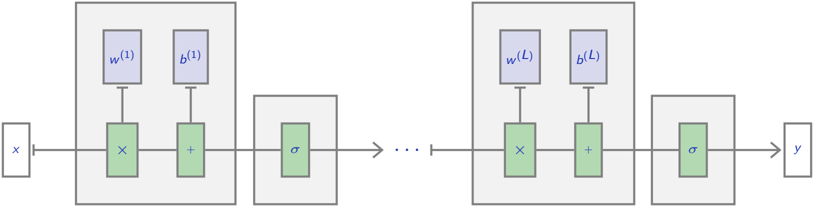
$$\forall l = 1, \dots, L,\ x^{(l)} = \sigma\left(w^{(l)} x^{(l-1)} + b^{(l)}\right)$$

and $f(x; w, b) = x^{(L)}$.

We can combine several "layers". With $x^{(0)} = x$,

$$\forall l = 1, \dots, L,\ x^{(l)} = \sigma\left(w^{(l)} x^{(l-1)} + b^{(l)}\right)$$

and $f(x; w, b) = x^{(L)}$.

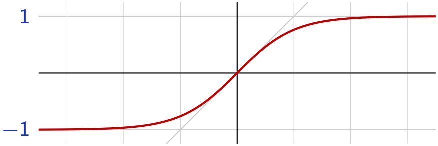


Such a model is a **Multi-Layer Perceptron (MLP).**

⚠ If σ is affine, this is an affine mapping.

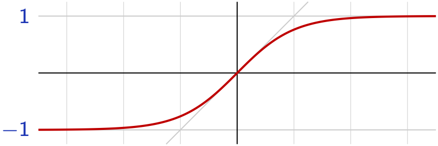
The two classical activation functions are the hyperbolic tangent

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



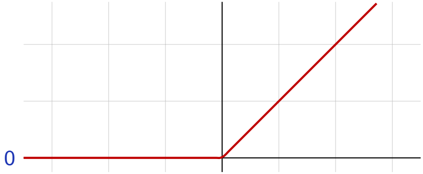
The two classical activation functions are the hyperbolic tangent

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



and the rectified linear unit (ReLU)

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



Under mild assumption on σ, we can approximate any continuous function

$$f : [0,1]^D \rightarrow \mathbb{R}$$

with a one hidden layer perceptron if it has enough units. This is the **universal approximation theorem.**

Under mild assumption on σ, we can approximate any continuous function

$$f : [0,1]^D \to \mathbb{R}$$

with a one hidden layer perceptron if it has enough units. This is the **universal approximation theorem.**

⚠ This says nothing about the number of units and the resulting mapping “complexity”.

Training and gradient descent

We saw that training consists of finding the model parameters minimizing an empirical risk or loss, for instance the mean-squared error (MSE)

$$\mathscr{L}(w,b) = \frac{1}{N}\sum_n \ell\left(f(x_n; w, b) - y_n\right)^2.$$

Other losses are more fitting for classification, certain regression problems, or density estimation. We will come back to this.

We saw that training consists of finding the model parameters minimizing an empirical risk or loss, for instance the mean-squared error (MSE)

$$\mathscr{L}(w, b) = \frac{1}{N} \sum_n \ell \left(f(x_n; w, b) - y_n\right)^2.$$

Other losses are more fitting for classification, certain regression problems, or density estimation. We will come back to this.

In what we saw, we minimized the MSE with an analytic solution and the empirical error rate with the perceptron.

The general optimization method when dealing with an arbitrary loss and model is the **gradient descent**.

Given a functional

$$\begin{aligned} f : \mathbb{R}^D &\to \mathbb{R} \\ x &\mapsto f(x_1, \dots, x_D), \end{aligned}$$

its gradient is the mapping

$$\begin{aligned} \nabla f : \mathbb{R}^D &\to \mathbb{R}^D \\ x &\mapsto \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_D}(x) \right). \end{aligned}$$

To minimize a functional

$$\mathscr{L} : \mathbb{R}^D \to \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

To minimize a functional

$$\mathscr{L} : \mathbb{R}^D \to \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

For $w_0 \in \mathbb{R}^D$, consider an approximation of $\mathscr{L}$ around w_0

$$\tilde{\mathscr{L}}_{w_0}(w) = \mathscr{L}(w_0) + \nabla \mathscr{L}(w_0)^T (w - w_0) + \frac{1}{2\eta} \|w - w_0\|^2.$$

Note that the chosen quadratic term does not depend on $\mathscr{L}$.

To minimize a functional

$$\mathscr{L} : \mathbb{R}^D \rightarrow \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

For $w_0 \in \mathbb{R}^D$, consider an approximation of $\mathscr{L}$ around w_0

$$\tilde{\mathscr{L}}_{w_0}(w) = \mathscr{L}(w_0) + \nabla \mathscr{L}(w_0)^T (w - w_0) + \frac{1}{2\eta} \|w - w_0\|^2.$$

Note that the chosen quadratic term does not depend on $\mathscr{L}$.

We have

$$\nabla \tilde{\mathscr{L}}_{w_0}(w) = \nabla \mathscr{L}(w_0) + \frac{1}{\eta}(w - w_0),$$

which leads to

$$\underset{w}{\operatorname{argmin}}\, \tilde{\mathscr{L}}_{w_0}(w) = w_0 - \eta \nabla \mathscr{L}(w_0).$$

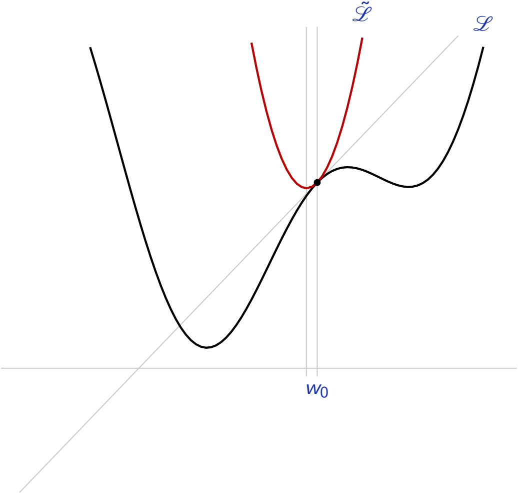
The resulting iterative rule takes the form of:

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t).$$

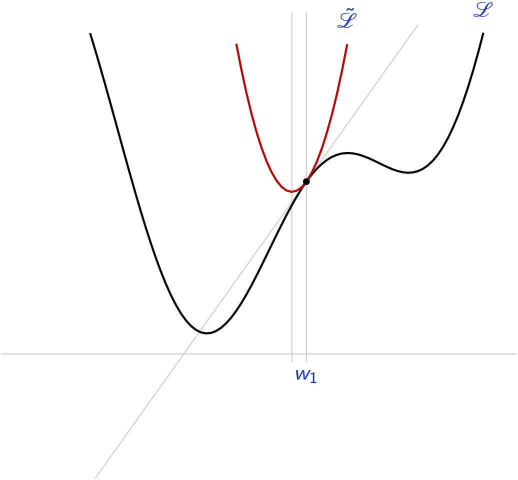
Which corresponds intuitively to "following the steepest descent".

This finds a **local** minimum, and the choices of w_0 and η are important.

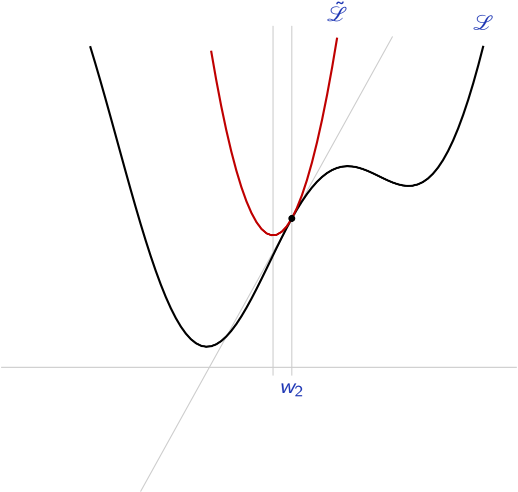
$\eta = 0.125$



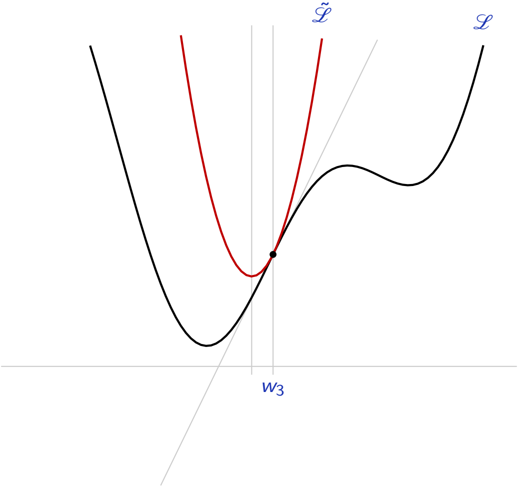
$\eta = 0.125$



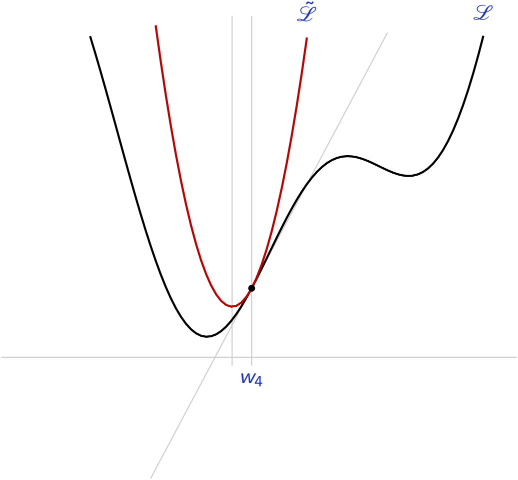
$\eta = 0.125$



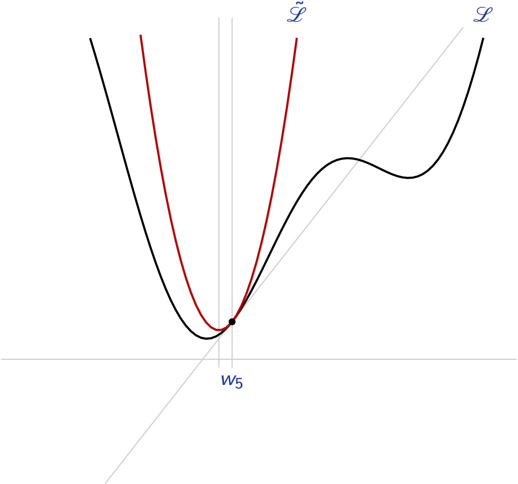
$\eta = 0.125$



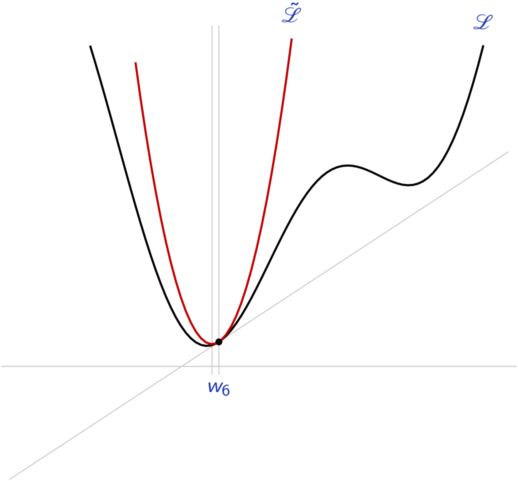
$\eta = 0.125$



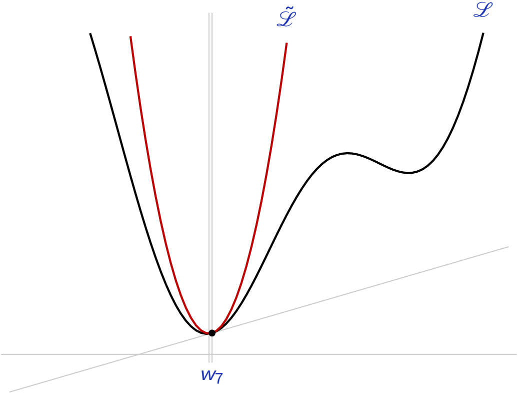
$\eta = 0.125$



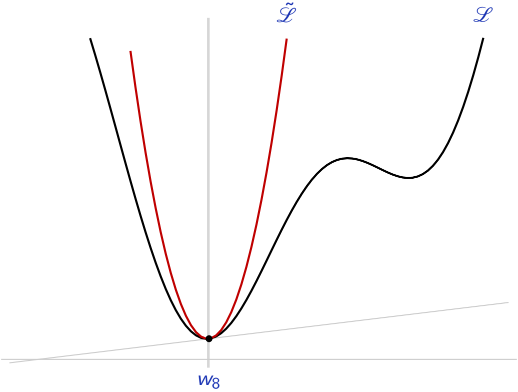
$\eta = 0.125$



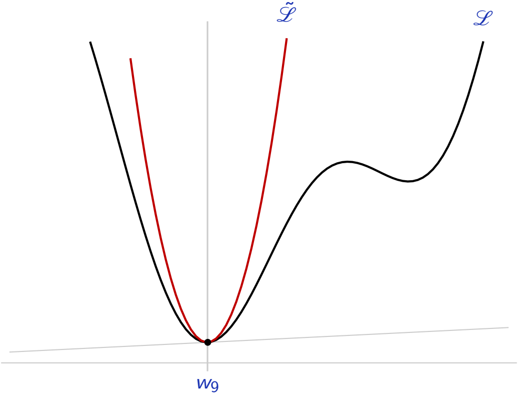
$\eta = 0.125$



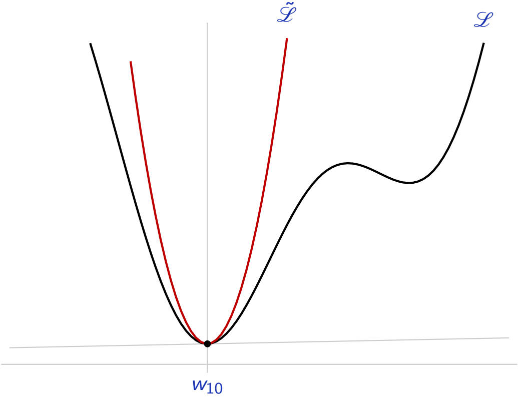
$\eta = 0.125$



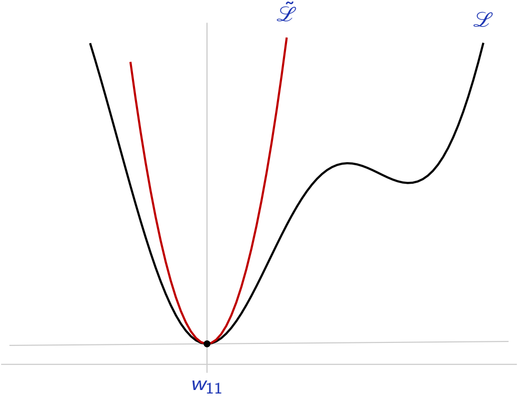
$\eta = 0.125$



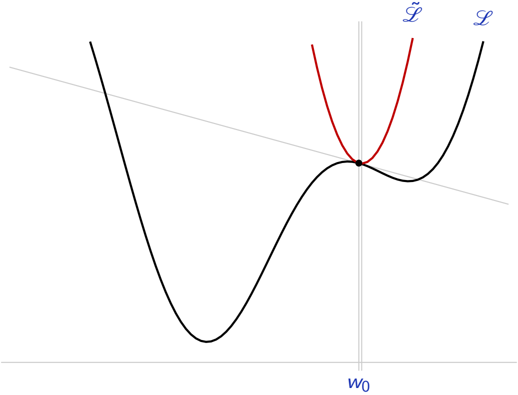
$\eta = 0.125$



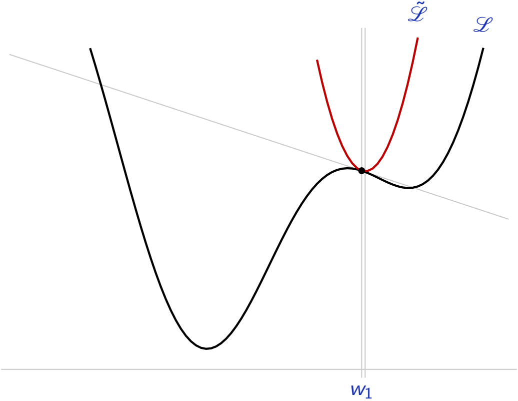
$\eta = 0.125$



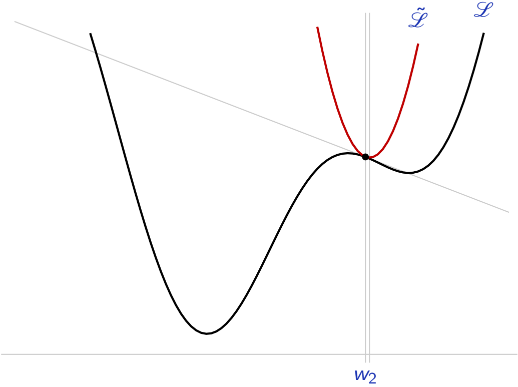
$\eta = 0.125$



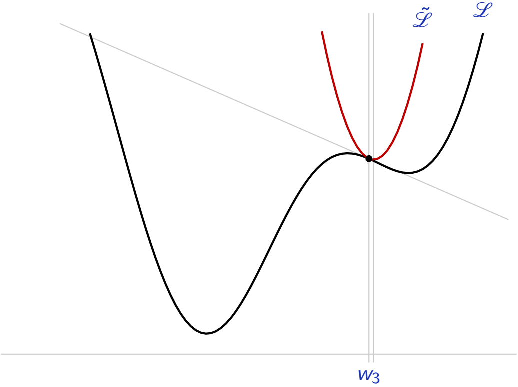
$\eta = 0.125$



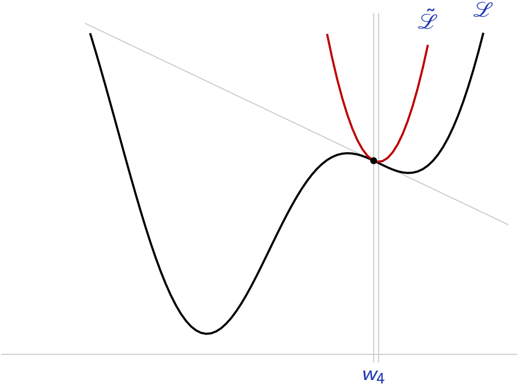
$\eta = 0.125$



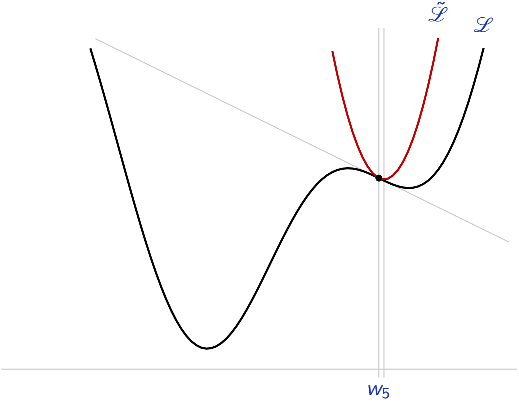
$\eta = 0.125$



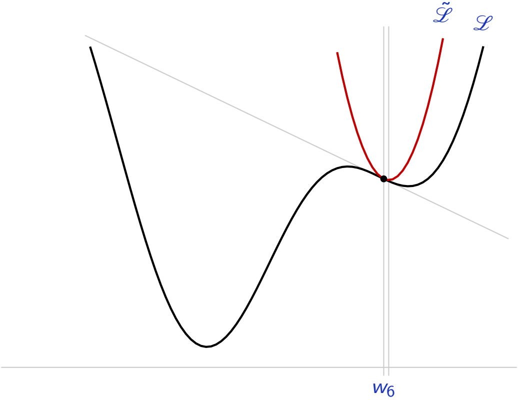
$\eta = 0.125$



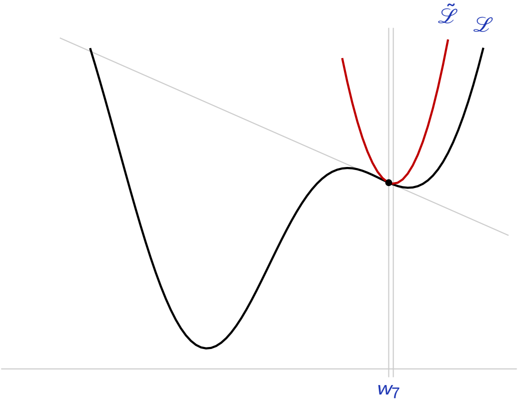
$\eta = 0.125$



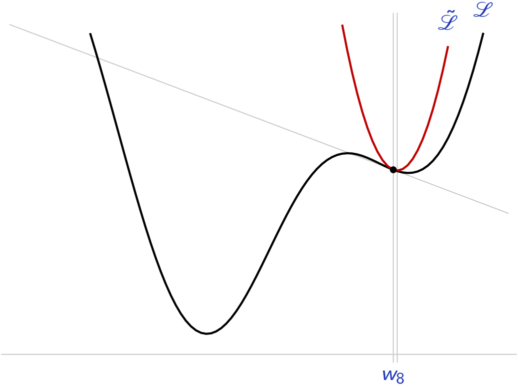
$\eta = 0.125$



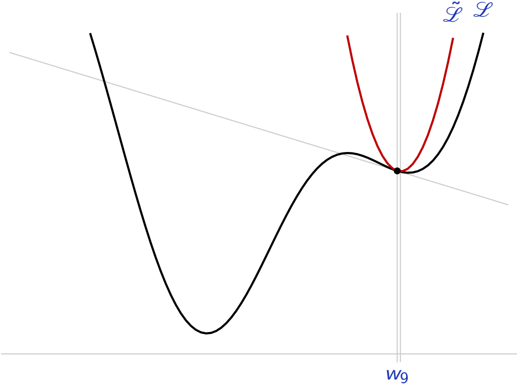
$\eta = 0.125$



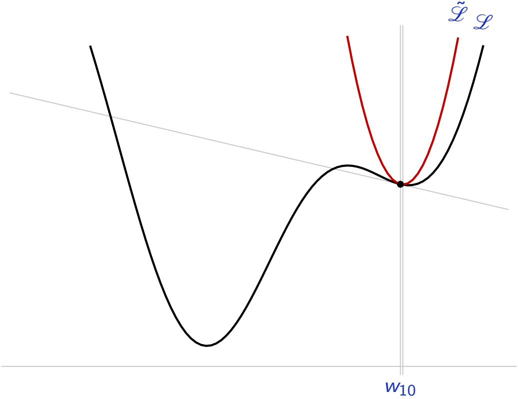
$\eta = 0.125$



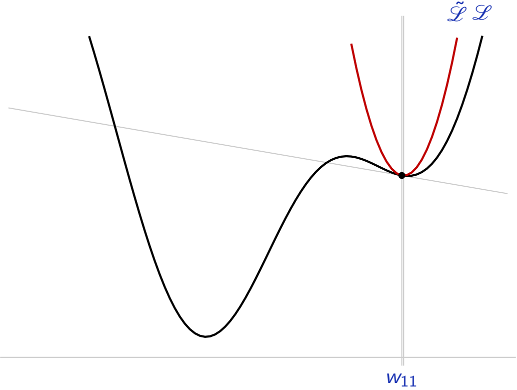
$\eta = 0.125$



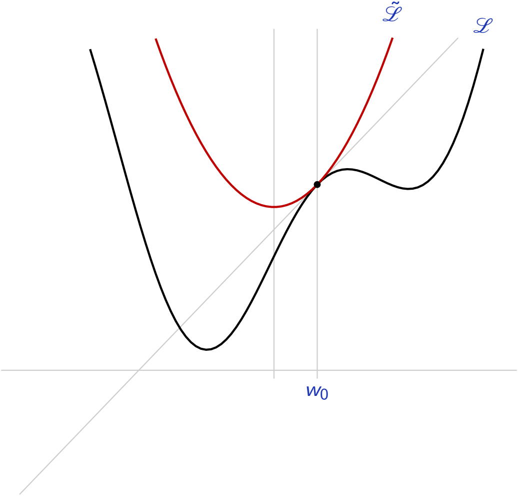
$\eta = 0.125$



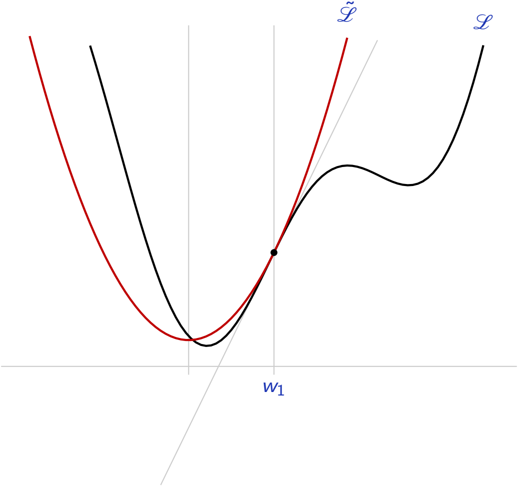
$\eta = 0.125$



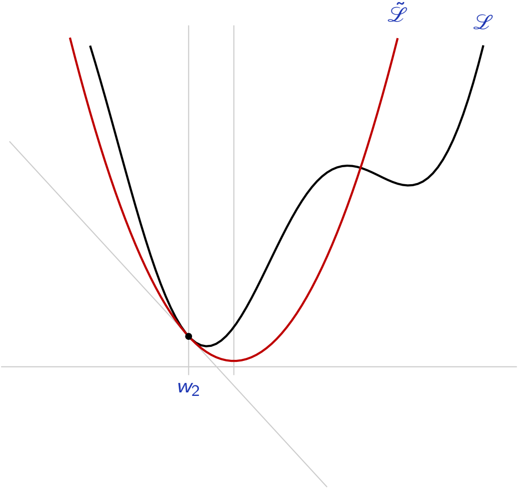
$\eta = 0.5$



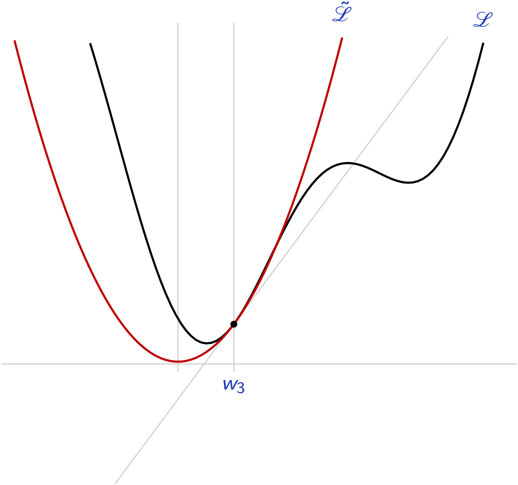
$\eta = 0.5$



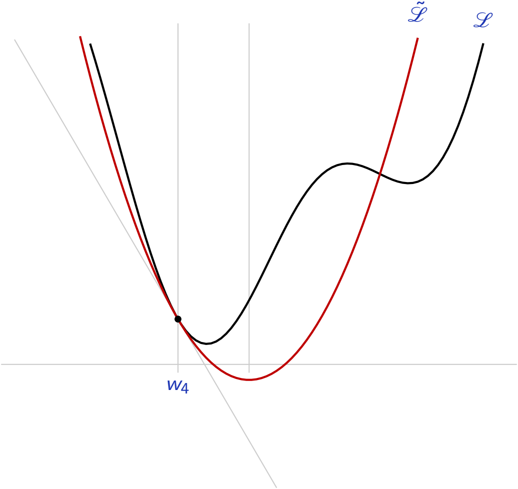
$\eta = 0.5$



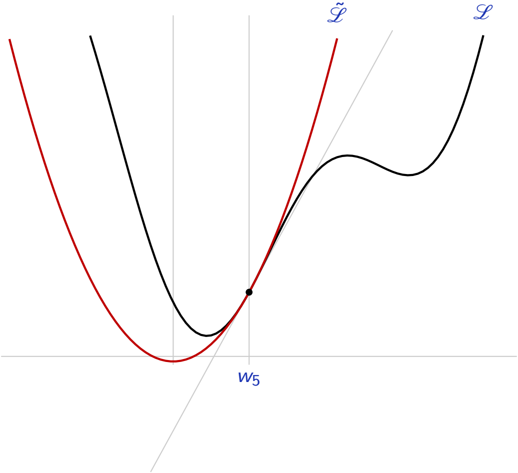
$\eta = 0.5$



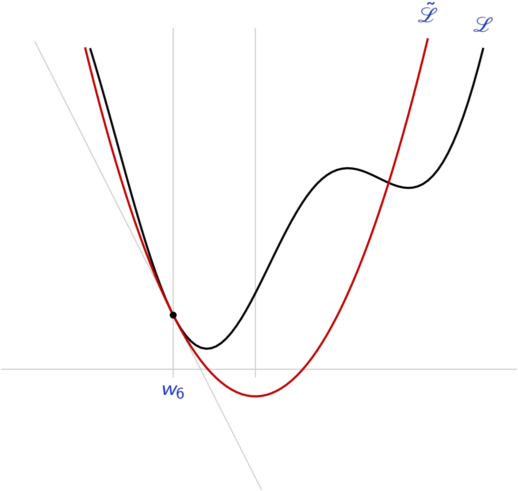
$\eta = 0.5$



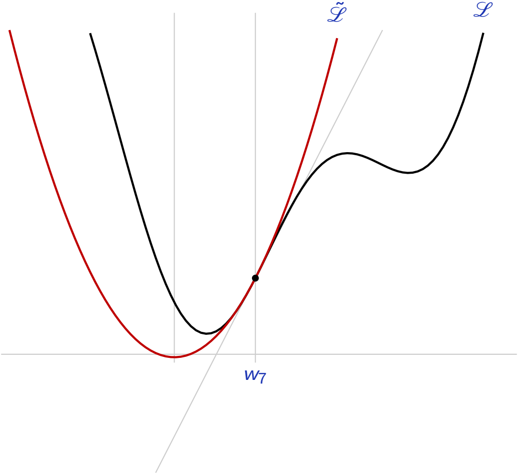
$\eta = 0.5$



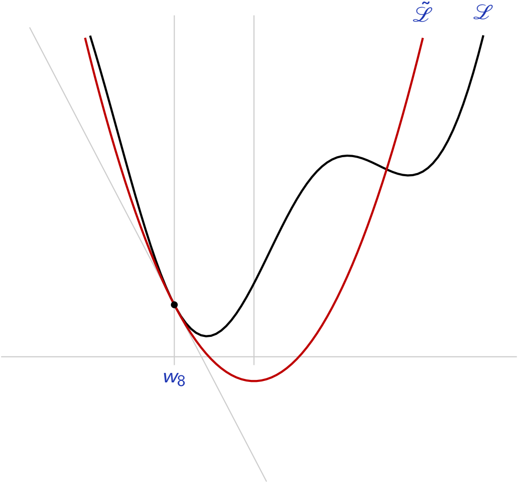
$\eta = 0.5$



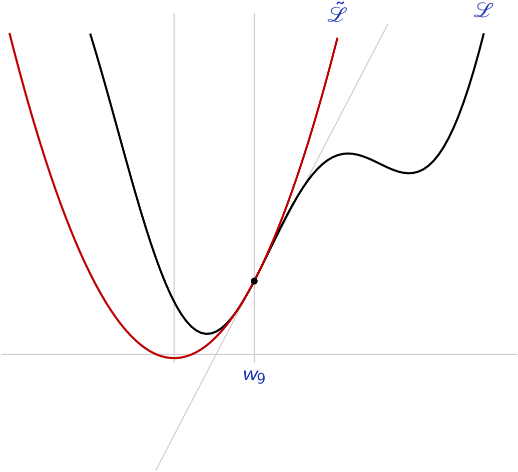
$\eta = 0.5$



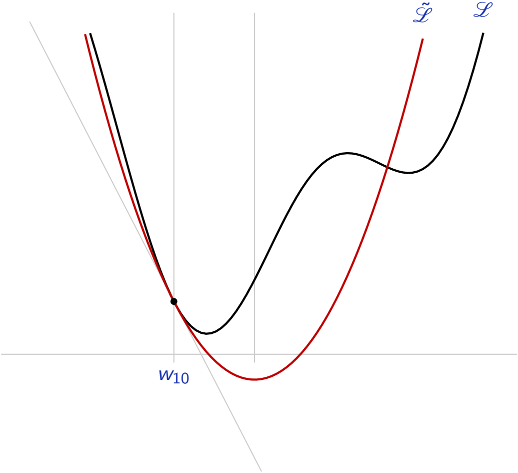
$\eta = 0.5$



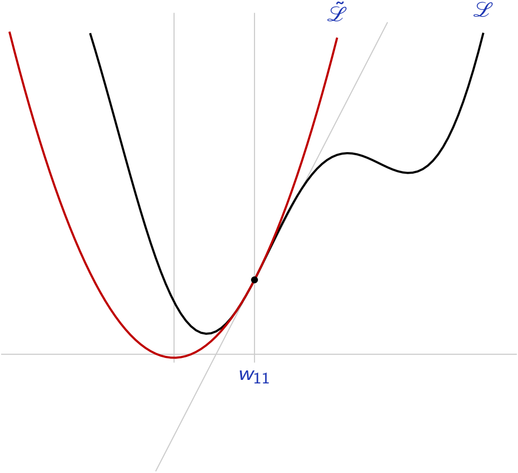
$\eta = 0.5$

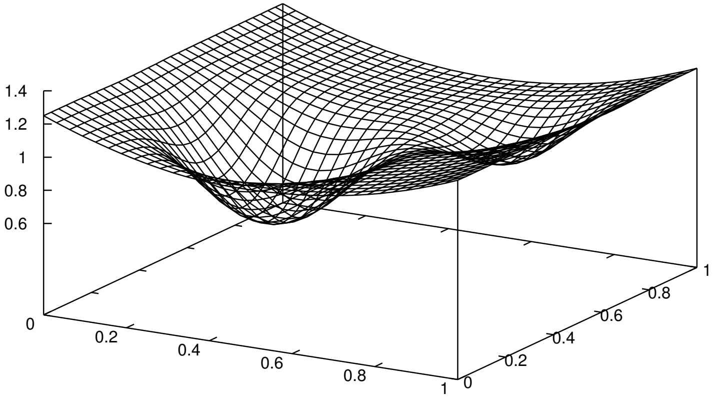


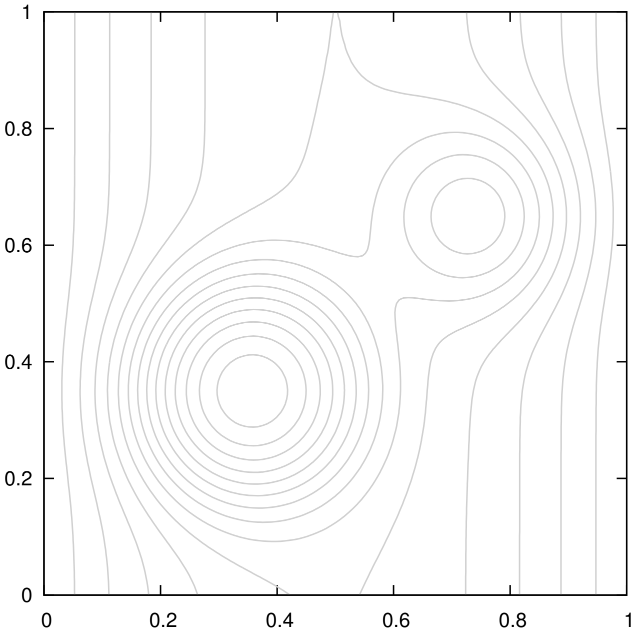
$\eta = 0.5$

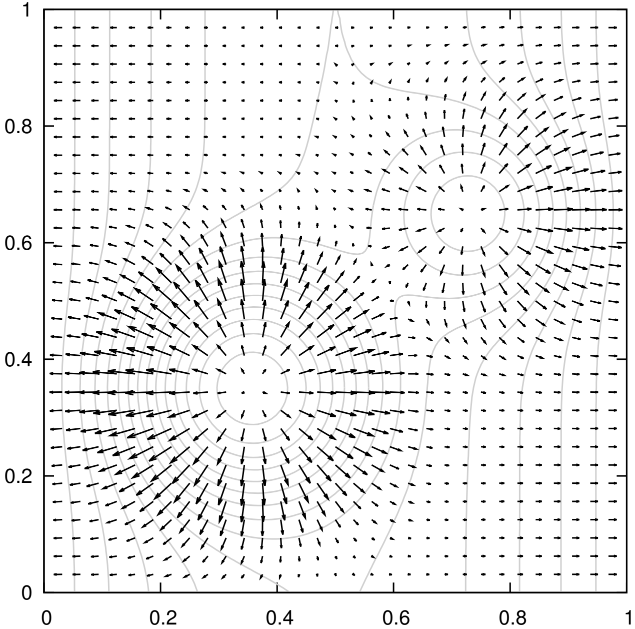


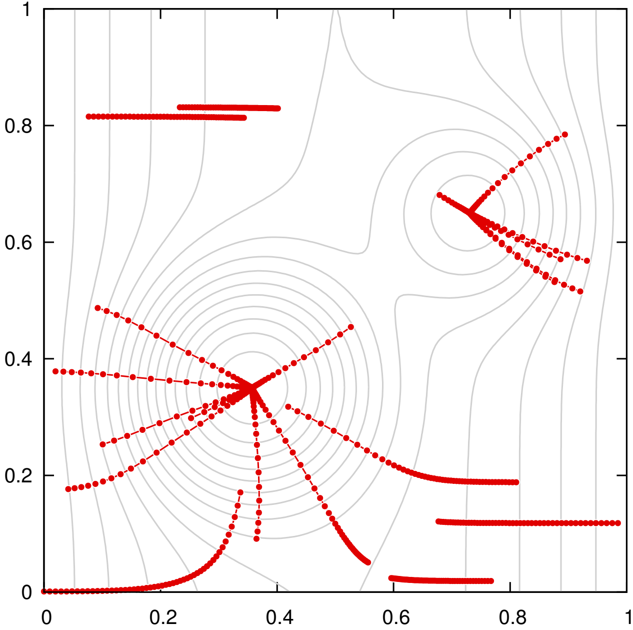
$\eta = 0.5$




1.4
1.2
1
0.8
0.6
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

```
from torch import nn
from torchvision import datasets

nb_samples, positive_class = 1000, 5

data = datasets.MNIST('./data/mnist/', train = True, download = True)
x = data.train_data.narrow(0, 0, nb_samples).view(-1, 28 * 28).float()
y = (data.train_labels.narrow(0, 0, nb_samples) == positive_class).float()

x.sub_(x.mean()).div_(x.std()) # Normalize the data
```

```
from torch import nn
from torchvision import datasets

nb_samples, positive_class = 1000, 5

data = datasets.MNIST('./data/mnist/', train = True, download = True)
x = data.train_data.narrow(0, 0, nb_samples).view(-1, 28 * 28).float()
y = (data.train_labels.narrow(0, 0, nb_samples) == positive_class).float()

x.sub_(x.mean()).div_(x.std()) # Normalize the data

model = nn.Sequential(nn.Linear(784, 200), nn.ReLU(), nn.Linear(200, 1))

for k in range(1001):
    yhat = model(x).view(-1) # Makes the vector 1d
    loss = (yhat - y).pow(2).mean()
    if k%100 == 0:
        nb_errors = ((yhat > 0.5).float() != y).sum()
        print(k, loss.item(), nb_errors.item())

    # Automagically compute the gradient
    model.zero_grad()
    loss.backward()

    for p in model.parameters(): p.detach().sub_(1e-2 * p.grad)
```

```
0 0.14268699288368225 93
100 0.03848343715071678 45
200 0.02569294534623623 18
300 0.0191640704870224 12
400 0.01507069543004036 7
500 0.012178068049252033 4
600 0.010016602464020252 4
700 0.008355352096259594 2
800 0.007042048498981516 1
900 0.006001953966915607 5 0
1000 0.005159074906259775 0
```

Note that these are the **training** loss and error.

Back-propagation

We want to train an MLP by minimizing a loss over the training set

$$\mathscr{L}(w, b) = \sum_{n} \ell(f(x_n; w, b), y_n).$$

We want to train an MLP by minimizing a loss over the training set

$$\mathscr{L}(w, b) = \sum_n \ell(f(x_n; w, b), y_n).$$

To use gradient descent, we need the expression of the gradient of the loss with respect to the parameters:

$$\frac{\partial \mathscr{L}}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \mathscr{L}}{\partial b_i^{(l)}}.$$

We want to train an MLP by minimizing a loss over the training set

$$\mathscr{L}(w, b) = \sum_n \ell(f(x_n; w, b), y_n).$$

To use gradient descent, we need the expression of the gradient of the loss with respect to the parameters:

$$\frac{\partial \mathscr{L}}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \mathscr{L}}{\partial b_i^{(l)}}.$$

So, with $\ell_n = \ell(f(x_n; w, b), y_n)$, what we need is

$$\frac{\partial \ell_n}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \ell_n}{\partial b_i^{(l)}}.$$

The core principle of the back-propagation algorithm is the “chain rule” from differential calculus:

$$(g \circ f)' = (g' \circ f) f'$$

which generalizes to longer compositions and higher dimensions

$$J_{f_N \circ f_{N-1} \circ \dots \circ f_1}(x) = \prod_{n=1}^{N} J_{f_n}(f_{n-1} \circ \dots \circ f_1(x)),$$

where $J_f(x)$ is the Jacobian of f at x, that is the matrix of the linear approximation of f in the neighborhood of x.

The core principle of the back-propagation algorithm is the “chain rule” from differential calculus:

$$(g \circ f)' = (g' \circ f) f'$$

which generalizes to longer compositions and higher dimensions

$$J_{f_N \circ f_{N-1} \circ \dots \circ f_1}(x) = \prod_{n=1}^{N} J_{f_n}(f_{n-1} \circ \dots \circ f_1(x)),$$

where $J_f(x)$ is the Jacobian of f at x, that is the matrix of the linear approximation of f in the neighborhood of x.

The linear approximation of a composition of mappings is the product of their individual linear approximations.

The core principle of the back-propagation algorithm is the “chain rule” from differential calculus:

$$(g \circ f)' = (g' \circ f)f'$$

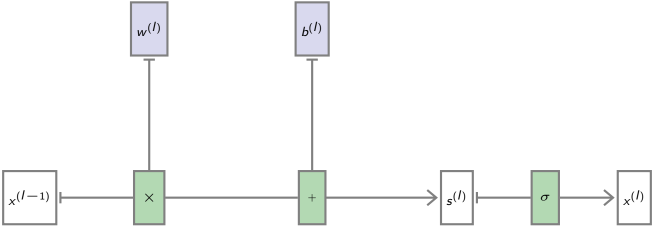
which generalizes to longer compositions and higher dimensions

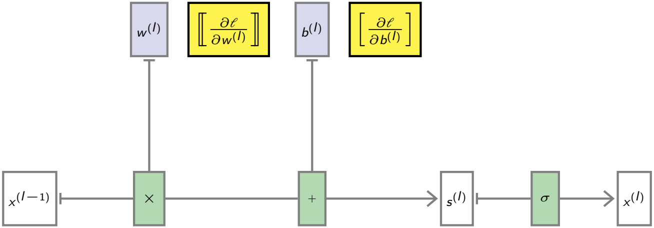
$$J_{f_N \circ f_{N-1} \circ \dots \circ f_1}(x) = \prod_{n=1}^{N} J_{f_n}(f_{n-1} \circ \dots \circ f_1(x)),$$

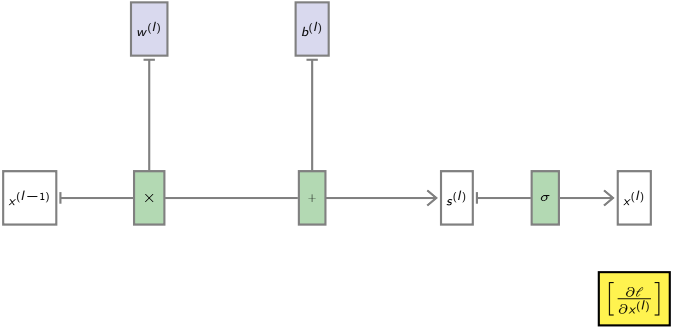
where $J_f(x)$ is the Jacobian of f at x, that is the matrix of the linear approximation of f in the neighborhood of x.

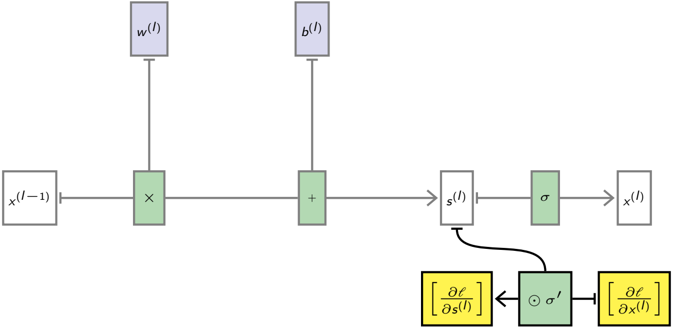
The linear approximation of a composition of mappings is the product of their individual linear approximations.

What follows is exactly this principle applied to a MLP.

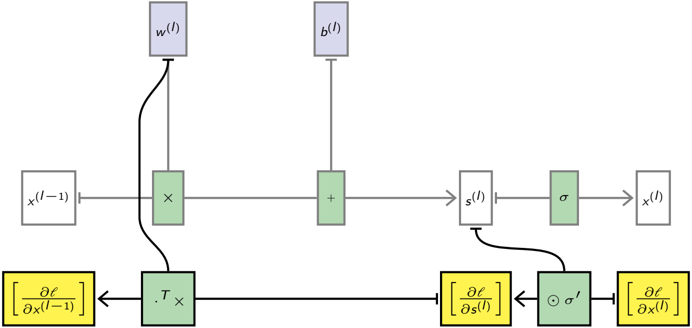

$w^{(l)}$
$b^{(l)}$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$


$w^{(l)}$
$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right]$
$b^{(l)}$
$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$

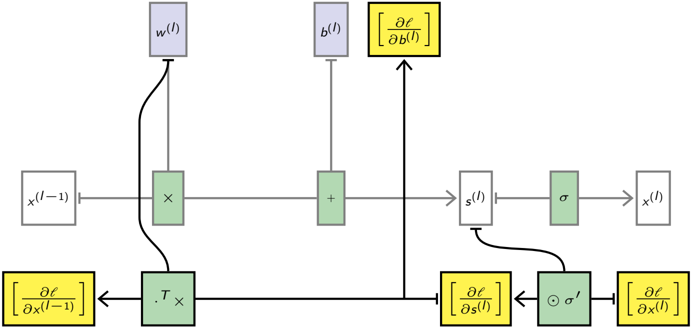

$w^{(l)}$
$b^{(l)}$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$


$w^{(l)}$
$b^{(l)}$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot \sigma'$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

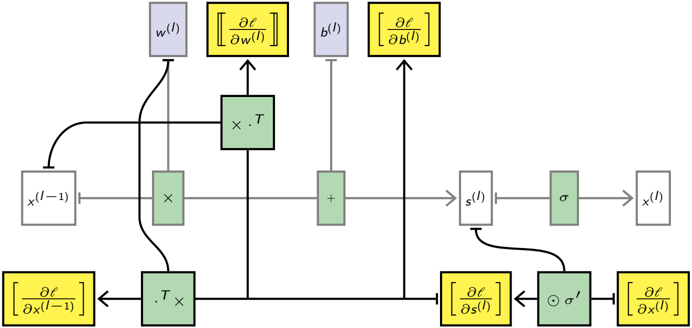
$$\frac{\partial \ell}{\partial s_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l)}} \sigma'\left(s_i^{(l)}\right)$$


$w^{(l)}$
$b^{(l)}$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
$\left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]$
$.^T \times$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot\, \sigma'$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

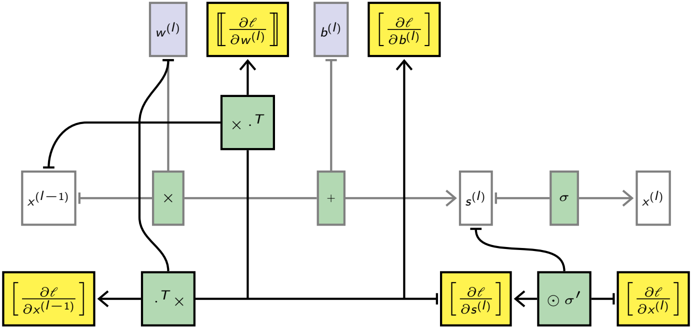
$$\frac{\partial \ell}{\partial x_j^{(l-1)}} = \sum_i w_{i,j}^{(l)} \frac{\partial \ell}{\partial s_i^{(l)}}$$


$w^{(l)}$
$b^{(l)}$
$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
$\left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]$
$.^T \times$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot \sigma'$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

$$\frac{\partial \ell}{\partial b_i^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}}$$


$w^{(l)}$
$\left[\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\right]$
$b^{(l)}$
$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$
$\times .^T$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
$\left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]$
$.^T \times$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot \sigma'$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

$$\frac{\partial \ell}{\partial w_{i,j}^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}} x_j^{(l-1)}$$

$w^{(l)}$
$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right]$
$b^{(l)}$
$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$
$\times\ .^T$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
$\left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]$
$.^T\times$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot\ \sigma'$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

Forward pass

$$\forall n,\ x_n^{(0)} = x_n, \quad \forall l = 1, \dots, L, \ \begin{cases} s_n^{(l)} = w^{(l)} x_n^{(l-1)} + b^{(l)} \\ x_n^{(l)} = \sigma\left(s_n^{(l)}\right) \end{cases}$$

Forward pass

$$\forall n,\ x_n^{(0)} = x_n,\quad \forall l = 1, \dots, L,\ \begin{cases} s_n^{(l)} = w^{(l)} x_n^{(l-1)} + b^{(l)} \\ x_n^{(l)} = \sigma\left(s_n^{(l)}\right) \end{cases}$$

Backward pass

$$\begin{cases} \left[\frac{\partial \ell_n}{\partial x_n^{(L)}}\right] = \nabla_1 \ell_n \left(x_n^{(L)}\right) \\ \text{if } l < L, \left[\frac{\partial \ell_n}{\partial x_n^{(l)}}\right] = \left(w^{l+1}\right)^T \left[\frac{\partial \ell_n}{\partial s^{l+1}}\right] \end{cases}$$

$$\left[\frac{\partial \ell_n}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell_n}{\partial x_n^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$$

$$\left[\!\left[\frac{\partial \ell_n}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell_n}{\partial s^{(l)}}\right] \left(x_n^{(l-1)}\right)^T$$

$$\left[\frac{\partial \ell_n}{\partial b^{(l)}}\right] = \left[\frac{\partial \ell_n}{\partial s^{(l)}}\right].$$

Forward pass

$$\forall n,\ x_n^{(0)} = x_n, \quad \forall l = 1, \dots, L, \begin{cases} s_n^{(l)} = w^{(l)} x_n^{(l-1)} + b^{(l)} \\ x_n^{(l)} = \sigma\left(s_n^{(l)}\right) \end{cases}$$

Backward pass

$$\begin{cases} \left[\frac{\partial \ell_n}{\partial x_n^{(L)}}\right] = \nabla_1 \ell_n \left(x_n^{(L)}\right) \\ \text{if } l < L, \left[\frac{\partial \ell_n}{\partial x_n^{(l)}}\right] = \left(w^{l+1}\right)^T \left[\frac{\partial \ell_n}{\partial s^{l+1}}\right] \end{cases} \qquad \left[\frac{\partial \ell_n}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell_n}{\partial x_n^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$$

$$\left[\!\left[\frac{\partial \ell_n}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell_n}{\partial s^{(l)}}\right] \left(x_n^{(l-1)}\right)^T \qquad \left[\frac{\partial \ell_n}{\partial b^{(l)}}\right] = \left[\frac{\partial \ell_n}{\partial s^{(l)}}\right].$$

Gradient step

$$w^{(l)} \leftarrow w^{(l)} - \eta \sum_n \left[\!\left[\frac{\partial \ell_n}{\partial w^{(l)}}\right]\!\right] \qquad b^{(l)} \leftarrow b^{(l)} - \eta \sum_n \left[\frac{\partial \ell_n}{\partial b^{(l)}}\right]$$

In spite of its hairy formalization, the backward pass is conceptually trivial: **Apply the chain rule again and again.**

In spite of its hairy formalization, the backward pass is conceptually trivial: **Apply the chain rule again and again.**

As for the forward pass, it can be expressed in tensorial form. Heavy computation is concentrated in linear operations, and all the non-linearities go into component-wise operations.

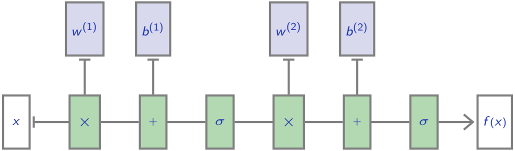
In spite of its hairy formalization, the backward pass is conceptually trivial: **Apply the chain rule again and again.**

As for the forward pass, it can be expressed in tensorial form. Heavy computation is concentrated in linear operations, and all the non-linearities go into component-wise operations.

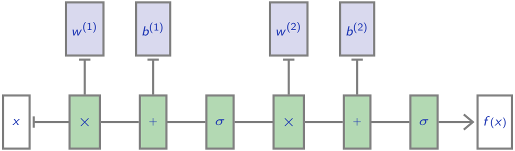
Regarding computation, the rule of thumb is that the backward pass is twice more expensive than the forward one.

DAG networks

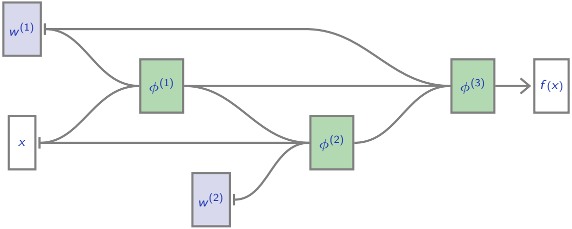
Everything we have seen for an MLP



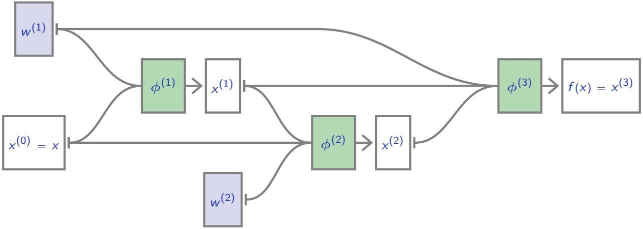
Everything we have seen for an MLP



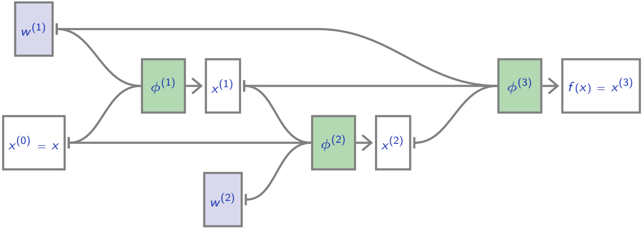
can be generalized to an arbitrary “Directed Acyclic Graph” (DAG) of operators



Forward pass

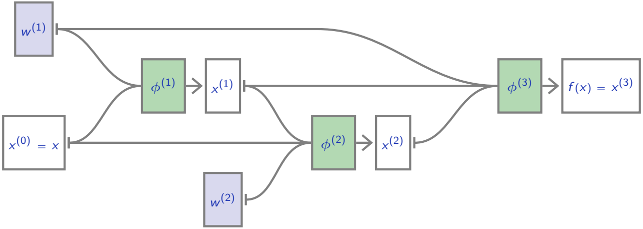


Forward pass



$$x^{(0)} = x$$

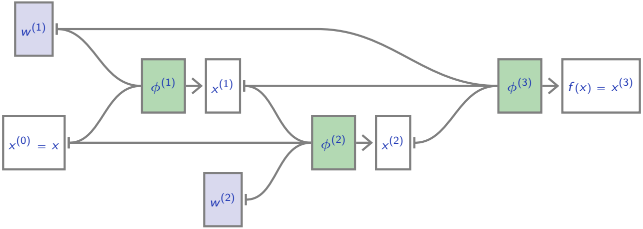
Forward pass



$$x^{(0)} = x$$

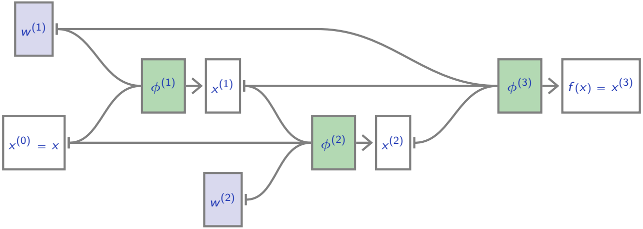
$$x^{(1)} = \phi^{(1)}(x^{(0)}; w^{(1)})$$

Forward pass



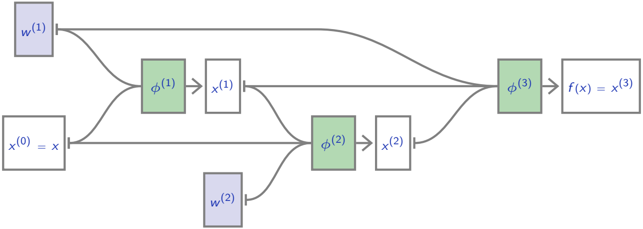
$$x^{(0)} = x$$
$$x^{(1)} = \phi^{(1)}(x^{(0)}; w^{(1)})$$
$$x^{(2)} = \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)})$$

Forward pass

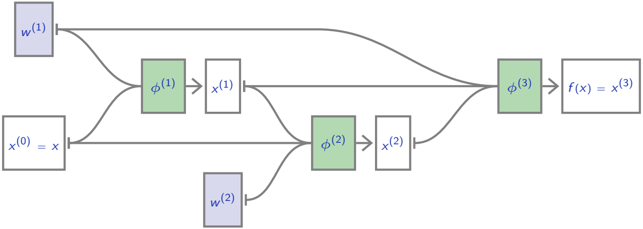


$$
\begin{aligned}
x^{(0)} &= x \\
x^{(1)} &= \phi^{(1)}(x^{(0)}; w^{(1)}) \\
x^{(2)} &= \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \\
f(x) = x^{(3)} &= \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)})
\end{aligned}
$$

Backward pass, derivatives w.r.t activations

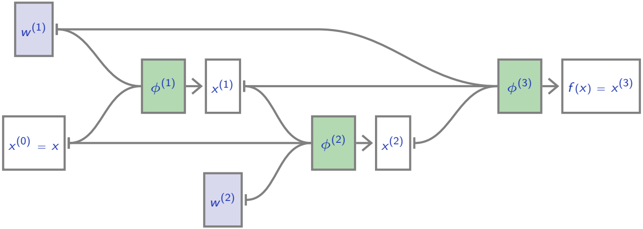


Backward pass, derivatives w.r.t activations



$$\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(3)}|x^{(2)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

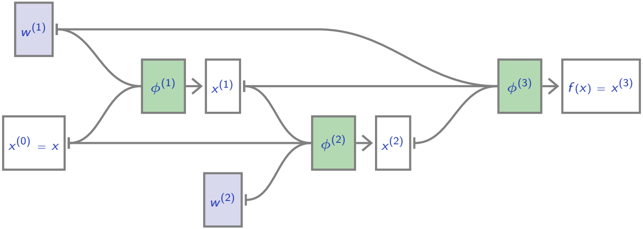
Backward pass, derivatives w.r.t activations



$$\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(3)}|x^{(2)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial x^{(1)}}\right] = \left[\frac{\partial x^{(2)}}{\partial x^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(2)}|x^{(1)}}\left[\frac{\partial \ell}{\partial x^{(2)}}\right] + J_{\phi^{(3)}|x^{(1)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

Backward pass, derivatives w.r.t activations

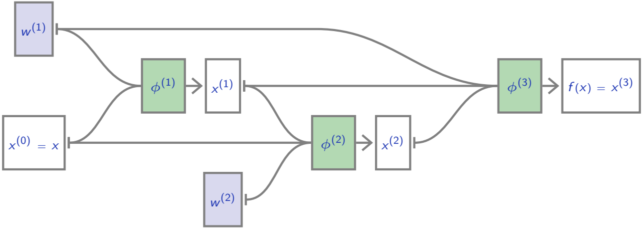


$$\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(3)}|x^{(2)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

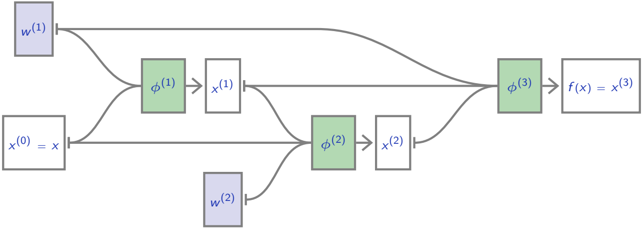
$$\left[\frac{\partial \ell}{\partial x^{(1)}}\right] = \left[\frac{\partial x^{(2)}}{\partial x^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(2)}|x^{(1)}}\left[\frac{\partial \ell}{\partial x^{(2)}}\right] + J_{\phi^{(3)}|x^{(1)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial x^{(0)}}\right] = \left[\frac{\partial x^{(1)}}{\partial x^{(0)}}\right]\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(2)}}{\partial x^{(0)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = J_{\phi^{(1)}|x^{(0)}}\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(2)}|x^{(0)}}\left[\frac{\partial \ell}{\partial x^{(2)}}\right]$$

Backward pass, derivatives w.r.t parameters

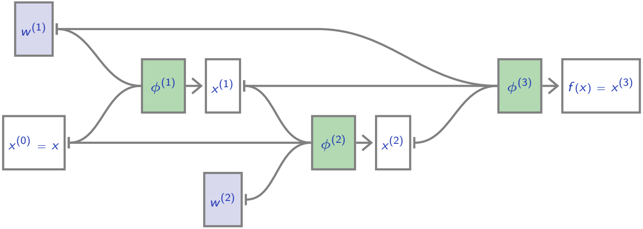


Backward pass, derivatives w.r.t parameters



$$\left[\frac{\partial \ell}{\partial w^{(1)}}\right] = \left[\frac{\partial x^{(1)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(3)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(1)}|w^{(1)}}\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(3)}|w^{(1)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

Backward pass, derivatives w.r.t parameters



$$\left[\frac{\partial \ell}{\partial w^{(1)}}\right] = \left[\frac{\partial x^{(1)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(3)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(1)}|w^{(1)}}\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(3)}|w^{(1)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial w^{(2)}}\right] = \left[\frac{\partial x^{(2)}}{\partial w^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = J_{\phi^{(2)}|w^{(2)}}\left[\frac{\partial \ell}{\partial x^{(2)}}\right]$$

So if we have a library of “tensor operators”, and implementations of

$$
\begin{aligned}
& (x_1, \dots, x_d, w) \mapsto \phi(x_1, \dots, x_d; w) \\
\forall c,\ & (x_1, \dots, x_d, w) \mapsto J_{\phi | x_c}(x_1, \dots, x_d; w) \\
& (x_1, \dots, x_d, w) \mapsto J_{\phi | w}(x_1, \dots, x_d; w),
\end{aligned}
$$

So if we have a library of “tensor operators”, and implementations of

$$\begin{aligned} & (x_1, \dots, x_d, w) \mapsto \phi(x_1, \dots, x_d; w) \\ \forall c, \; & (x_1, \dots, x_d, w) \mapsto J_{\phi|x_c}(x_1, \dots, x_d; w) \\ & (x_1, \dots, x_d, w) \mapsto J_{\phi|w}(x_1, \dots, x_d; w), \end{aligned}$$

we can build an arbitrary directed acyclic graph with these operators at the nodes, compute the response of the resulting mapping, and compute its gradient with back-prop.

Writing from scratch a large neural network is complex and error-prone.

Writing from scratch a large neural network is complex and error-prone.

Multiple frameworks provide libraries of tensor operators and mechanisms to combine them into DAGs and automatically differentiate them.

	Language(s)	**License**	**Main backer**
PyTorch	**Python**	BSD	Facebook
Caffe2	C++, Python	Apache	Facebook
TensorFlow	Python, C++	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
CNTK	Python, C++	MIT	Microsoft
Torch	Lua	BSD	Facebook
Theano	Python	BSD	U. of Montreal
Caffe	C++	BSD 2 clauses	U. of CA, Berkeley

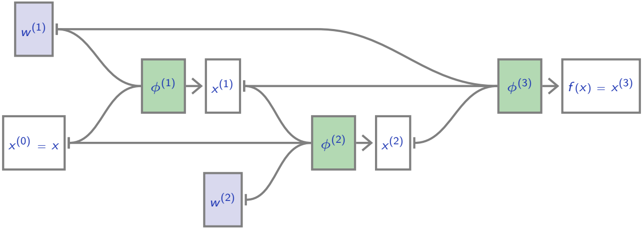
Writing from scratch a large neural network is complex and error-prone.

Multiple frameworks provide libraries of tensor operators and mechanisms to combine them into DAGs and automatically differentiate them.

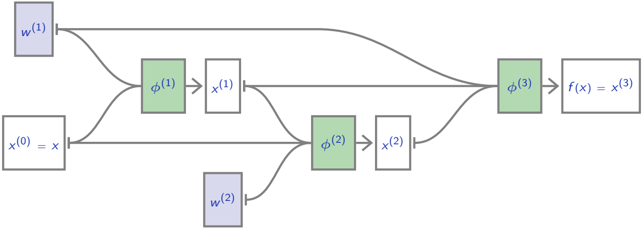
	Language(s)	**License**	**Main backer**
PyTorch	**Python**	BSD	Facebook
Caffe2	C++, Python	Apache	Facebook
TensorFlow	Python, C++	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
CNTK	Python, C++	MIT	Microsoft
Torch	Lua	BSD	Facebook
Theano	Python	BSD	U. of Montreal
Caffe	C++	BSD 2 clauses	U. of CA, Berkeley

One approach is to define the nodes and edges of such a DAG statically (Torch, TensorFlow, Caffe, Theano, etc.)

For instance, in TensorFlow, to run a forward/backward pass on



For instance, in TensorFlow, to run a forward/backward pass on



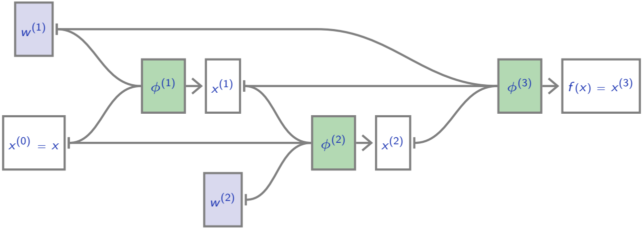
with

$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)} x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)} x^{(1)}$$

$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

For instance, in TensorFlow, to run a forward/backward pass on



with

$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$

$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

we can do

```
w1 = tf.Variable(tf.random_normal([5, 5]))
w2 = tf.Variable(tf.random_normal([5, 5]))
x = tf.Variable(tf.random_normal([5, 1]))
x0 = x
x1 = tf.matmul(w1, x0)
x2 = x0 + tf.matmul(w2, x1)
x3 = tf.matmul(w1, x1 + x2)
q = tf.norm(x3)

gw1, gw2 = tf.gradients(q, [w1, w2])

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    _grads = sess.run(grads)
```

Autograd

The forward pass is “just” a computation as usual. The graph structure is needed only to apply the chain rule during the backward pass.

The forward pass is “just” a computation as usual. The graph structure is needed only to apply the chain rule during the backward pass.

PyTorch can record tensorial operations, and compute the gradient of any quantity with respect to any tensor involved.

The forward pass is “just” a computation as usual. The graph structure is needed only to apply the chain rule during the backward pass.

PyTorch can record tensorial operations, and compute the gradient of any quantity with respect to any tensor involved.

This “autograd” mechanism has two main benefits:

- Simpler syntax: one just need 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.

A `Tensor` has a Boolean field `requires_grad`, set to `False` by default, which states if PyTorch should record operations involving it so that gradients wrt 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`.

A `Tensor` has a Boolean field `requires_grad`, set to `False` by default, which states if PyTorch should record operations involving it so that gradients wrt 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`.

A `Tensor` also has a field `grad`, itself a tensor of same size, type, and device (or `None`) used to accumulate gradients.

A `Tensor` has a Boolean field `requires_grad`, set to `False` by default, which states if PyTorch should record operations involving it so that gradients wrt 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`.

A `Tensor` also has a field `grad`, itself a tensor of same size, type, and device (or `None`) used to accumulate gradients.

A `Parameter` is a `Tensor` with `requires_grad` to `True` by default, and known to be a model parameter by various utility functions.

`torch.autograd.grad(outputs, inputs)` computes and returns the sum of gradients of outputs wrt the specified inputs. This is always a `tuple`.

`torch.autograd.grad(outputs, inputs)` computes and returns the sum of gradients of outputs wrt the specified inputs. This is always a `tuple`.

An alternative is `torch.autograd.backward(tensors)` or `Tensor.backward()`, which accumulates the gradients in the `grad` fields of the "leaf" tensors, those which are not results of an operation.

Using the latter is standard for training models, as it automatically updates gradients for all parameters influencing the loss.

Consider a simple example $(x_1, x_2, x_3) = (1, 2, 2)$, and

$$\ell = \|x\| = \sqrt{x_1^2 + x_2^2 + x_3^2}.$$

We have $\ell = 3$ and

$$\frac{\partial \ell}{\partial x_i} = \frac{x_i}{\|x\|}.$$

Consider a simple example $(x_1, x_2, x_3) = (1, 2, 2)$, and

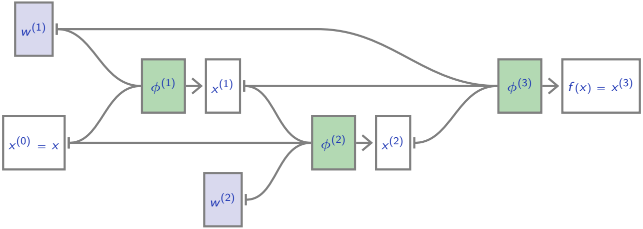
$$\ell = \|x\| = \sqrt{x_1^2 + x_2^2 + x_3^2}.$$

We have $\ell = 3$ and

$$\frac{\partial \ell}{\partial x_i} = \frac{x_i}{\|x\|}.$$

```
>>> import torch
>>> x = torch.tensor([1., 2., 2.]).requires_grad_()
>>> l = x.norm()
>>> g = torch.autograd.grad(l, (x,))
>>> g
(tensor([ 0.3333,  0.6667,  0.6667]),)
```

For instance, in PyTorch, to run a forward/backward pass on



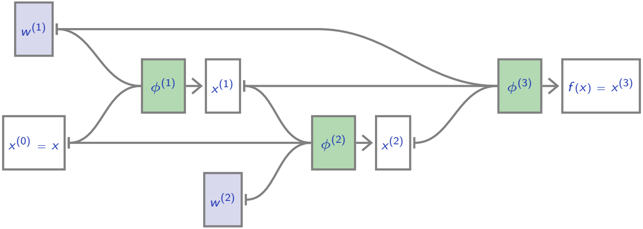
with

$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)} x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)} x^{(1)}$$

$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

For instance, in PyTorch, to run a forward/backward pass on



with

$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$

$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

we can do

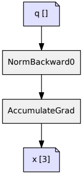
```
w1 = Parameter(torch.empty(5, 5).normal_())
w2 = Parameter(torch.empty(5, 5).normal_())
x = torch.empty(5).normal_()

x0 = x
x1 = w1.mv(x0)
x2 = x0 + w2.mv(x1)
x3 = w1.mv(x1 + x2)

q = x3.norm()

q.backward()
```

We can look precisely at the graph built during a computation.

```
x = torch.tensor([1, 2, 2]).requires_grad_()
q = x.norm()
```

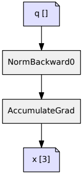
We can look precisely at the graph built during a computation.

```
x = torch.tensor([1, 2, 2]).requires_grad_()
q = x.norm()
```



We can look precisely at the graph built during a computation.

```
x = torch.tensor([1, 2, 2]).requires_grad_()
q = x.norm()
```



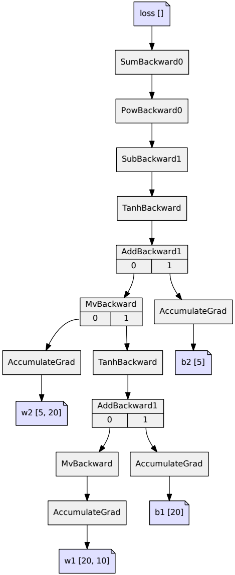
This graph was generated with

https://fleuret.org/git/agtree2dot

and Graphviz.

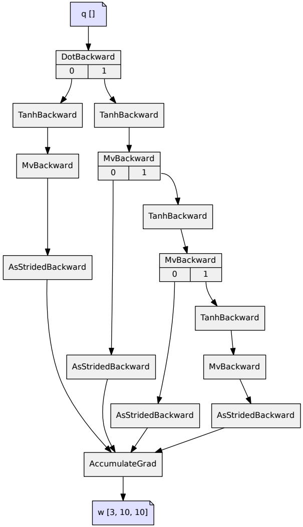
```
w1 = Parameter(torch.rand(20, 10))
b1 = Parameter(torch.rand(20))
w2 = Parameter(torch.rand(5, 20))
b2 = Parameter(torch.rand(5))

x = torch.rand(10)
h = torch.tanh(w1.mv(x) + b1)
y = torch.tanh(w2.mv(h) + b2)

target = torch.rand(5)

loss = (y - target).pow(2).sum()
```

```
w1 = Parameter(torch.rand(20, 10))
b1 = Parameter(torch.rand(20))
w2 = Parameter(torch.rand(5, 20))
b2 = Parameter(torch.rand(5))

x = torch.rand(10)
h = torch.tanh(w1.mv(x) + b1)
y = torch.tanh(w2.mv(h) + b2)

target = torch.rand(5)

loss = (y - target).pow(2).sum()
```



```
w = Parameter(torch.rand(3, 10, 10))

def blah(k, x):
    for i in range(0, k):
        x = torch.tanh(w[i].mv(x))
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```

```
w = Parameter(torch.rand(3, 10, 10))

def blah(k, x):
    for i in range(0, k):
        x = torch.tanh(w[i].mv(x))
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```



⚠ `Tensor.backward()` **accumulates** the gradients in the different `Tensor`s, so one may have to zero them before.

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.

⚠ `Tensor.backward()` **accumulates** the gradients in the different `Tensor`s, so one may have to zero them before.

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.

⚠ Although they are related, **the autograd graph is not the network’s structure**, but the graph of operations to compute the gradient. It can be data-dependent and miss or replicate sub-parts of the network.

In-place operations may corrupt values required to compute the gradient, and this is tracked down by autograd.

```
>>> x = torch.tensor([1., 2., 3.]).requires_grad_()
>>> y = x.sin()
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y += 1
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y *= y
>>> l = y.sum()
>>> l.backward()
Traceback (most recent call last):
/.../
RuntimeError: one of the variables needed for gradient computation has been modified by
        an inplace operation
```

They are also prohibited on leaf tensors.

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.

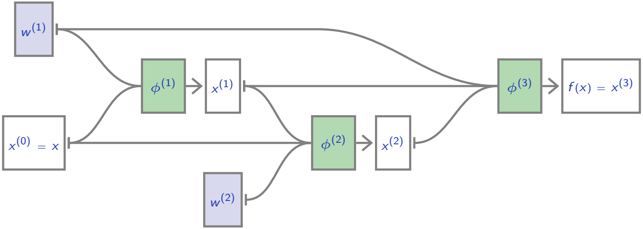
```
model = nn.Sequential(nn.Linear(784, 200), nn.ReLU(), nn.Linear(200, 1))

for k in range(1001):
    yhat = model(x).view(-1) # Makes the vector 1d
    loss = (yhat - y).pow(2).mean()
    if k%100 == 0:
        nb_errors = ((yhat > 0.5).float() != y).sum()
        print(k, loss.item(), nb_errors.item())

    # Automagically compute the gradient
    model.zero_grad()
    loss.backward()

    for p in model.parameters(): p.detach().sub_(1e-2 * p.grad)
```

Weight sharing

In our generalized DAG formulation, we have in particular implicitly allowed the same parameters to modulate different parts of the processing.

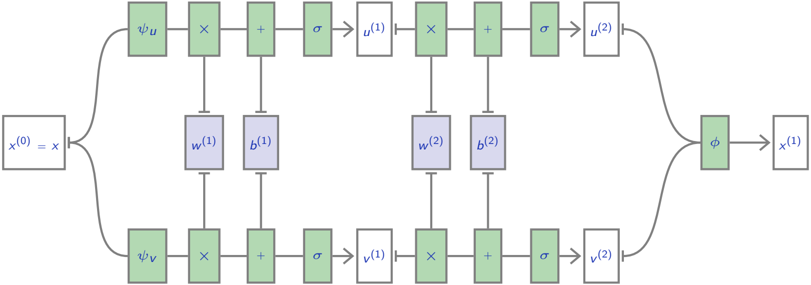
In our generalized DAG formulation, we have in particular implicitly allowed the same parameters to modulate different parts of the processing.

For instance $w^{(1)}$ in our example parametrizes both $\phi^{(1)}$ and $\phi^{(3)}$.

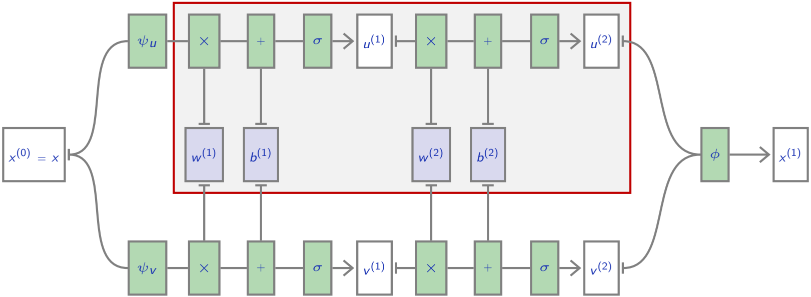


This is called **weight sharing**.

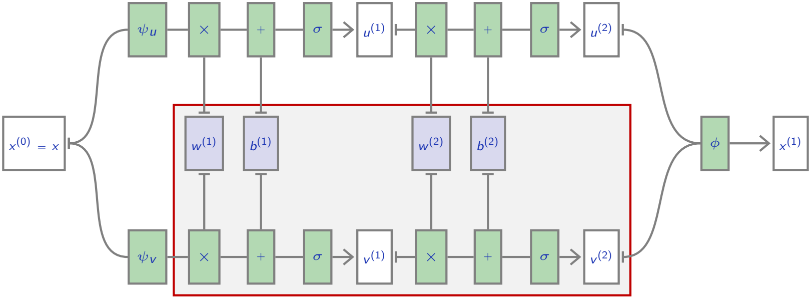
Weight sharing allows in particular to build **siamese networks** where a full sub-network is replicated several times.



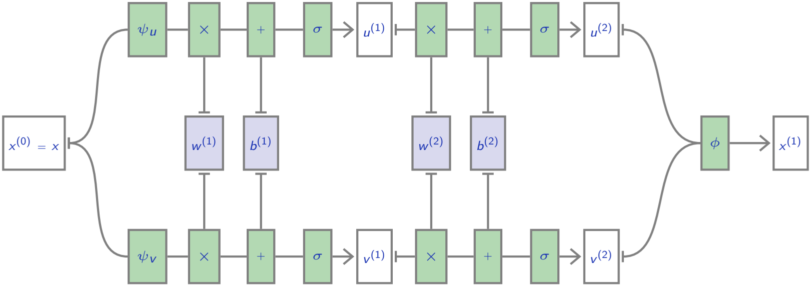
Weight sharing allows in particular to build **siamese networks** where a full sub-network is replicated several times.



Weight sharing allows in particular to build **siamese networks** where a full sub-network is replicated several times.



Weight sharing allows in particular to build **siamese networks** where a full sub-network is replicated several times.



The end

References

N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 886–893, 2005.

P. Dollár, Z. Tu, P. Perona, and S. Belongie. Integral channel features. In *British Machine Vision Conference*, pages 91.1–91.11, 2009.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

F. Rosenblatt. The perceptron–A perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.