

## EE-559 – Deep learning

### 5.4. $L_2$ and $L_1$ penalties

François Fleuret

<https://fleuret.org/ee559/>

Sat Nov 10 11:27:56 UTC 2018



We have motivated the use of a loss with a Bayesian formulation combining the probability of the data given the model and the probability of the model

$$\log \mu_W(w \mid \mathcal{D} = \mathbf{d}) = \log \mu_{\mathcal{D}}(\mathbf{d} \mid W = w) + \log \mu_W(w) - \log Z.$$

If  $\mu_W$  is a Gaussian density with a covariance matrix proportional to the identity, the log-prior  $\log \mu_W(w)$  results in a quadratic penalty

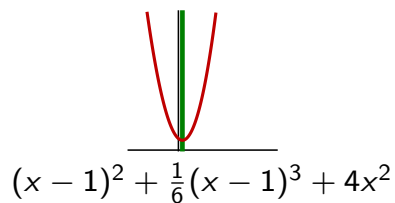
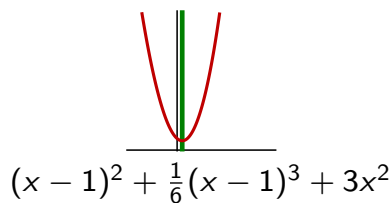
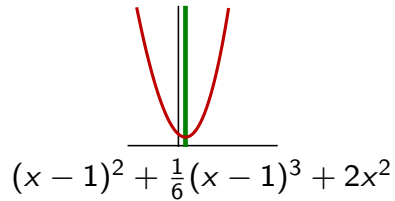
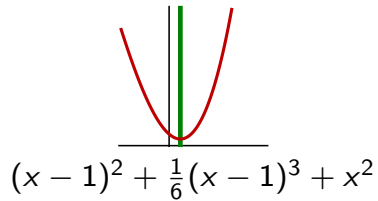
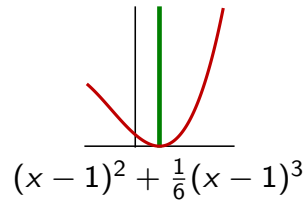
$$\lambda \|w\|_2^2.$$

Since this penalty is convex, its sum with a convex functional is convex.

This is called the  $L_2$  regularization, or “weight decay” in the artificial neural network community.

Increasing the  $\lambda$  parameter moves the optimal closer to 0, and away from the optimal for the loss alone.

Since the derivative of  $\|x\|_2^2$  is zero at zero, the optimal will never move there if it was not already there.



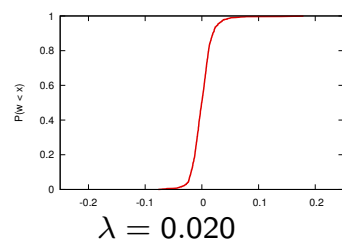
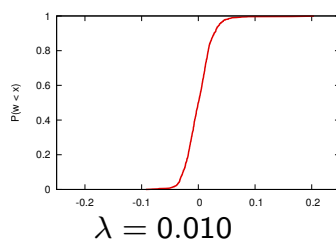
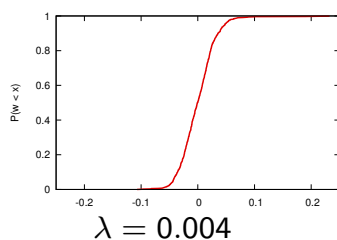
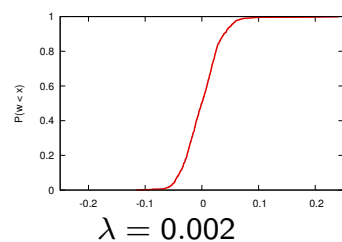
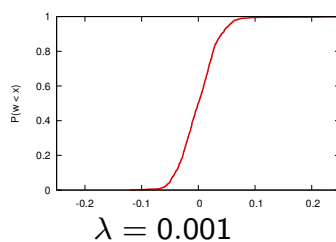
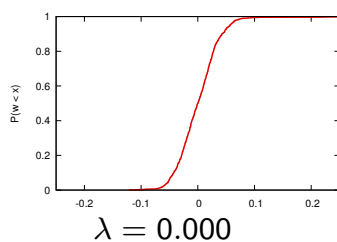
Convnet trained on MNIST with 1,000 samples and a  $L_2$  penalty.

$\lambda$	Error	
	Train	Test
0.000	0.000	0.064
0.001	0.000	0.063
0.002	0.000	0.064
0.004	0.005	0.065
0.010	0.022	0.075
0.020	0.048	0.101

```
output = model(train_input[b:b+batch_size])
loss = criterion(output, train_target[b:b+batch_size])
```

```
for p in model.parameters():
    loss += lambda_l2 * p.pow(2).sum()
```

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```



We can apply the exact same scheme with a Laplace prior

$$\begin{aligned}\mu(w) &= \frac{1}{(2b)^D} \exp\left(-\frac{\|w\|_1}{b}\right) \\ &= \frac{1}{(2b)^D} \exp\left(-\frac{1}{b} \sum_{d=1}^D |w_d|\right),\end{aligned}$$

which results in a penalty term of the form

$$\lambda \|w\|_1.$$

This is the  $L_1$  regularization. As for the  $L_2$ , this penalty is convex, and its sum with a convex functional is convex.

An important property of the  $L_1$  penalty is that, if  $\mathcal{L}$  is convex, and

$$w^* = \underset{w}{\operatorname{argmin}} \mathcal{L}(w) + \lambda \|w\|_1$$

then

$$\forall d, \left| \frac{\partial \mathcal{L}}{\partial w_d}(w^*) \right| < \lambda \Rightarrow w_d^* = 0.$$

In practice it means that this penalty pushes some of the variables to zero, but contrary to the  $L_2$  penalty they actually move and remain there.

The  $\lambda$  parameter controls the sparsity of the solution.

With the  $L_1$  penalty, the update rule becomes

$$w_{t+1} = w_t - \eta g_t - \lambda \text{sign}(w_t),$$

where  $\text{sign}$  is applied per-component. This is almost identical to

$$\begin{aligned} w'_t &= w_t - \eta g_t \\ w_{t+1} &= w'_t - \lambda \text{sign}(w'_t). \end{aligned}$$

This update may overshoot, and result in a component of  $w'_t$  strictly on one side of 0, while the same component in  $w_{t+1}$  is strictly on the other.

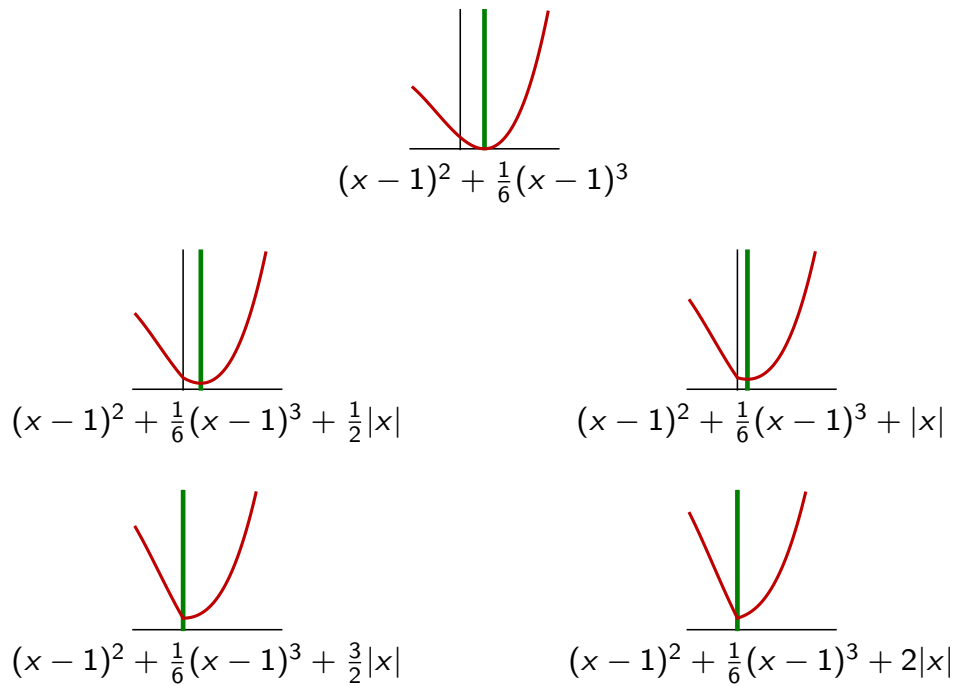
While this is not a problem in principle, since  $w_t$  will fluctuate around zero, it can be an issue if the zeroed weights are handled in a specific manner (e.g. sparse coding to reduce memory footprint or computation).

The **proximal operator** takes care of preventing parameters from “crossing zero”, by adapting  $\lambda$  when it is too large

$$\begin{aligned} w'_t &= w_t - \eta g_t \\ w_{t+1} &= w'_t - \min(\lambda, |w'_t|) \odot \text{sign}(w'_t). \end{aligned}$$

where  $\min$  is component-wise, and  $\odot$  is the Hadamard component-wise product.

Increasing the  $\lambda$  parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



Convnet trained on MNIST with 1,000 samples and a  $L_1$  penalty.

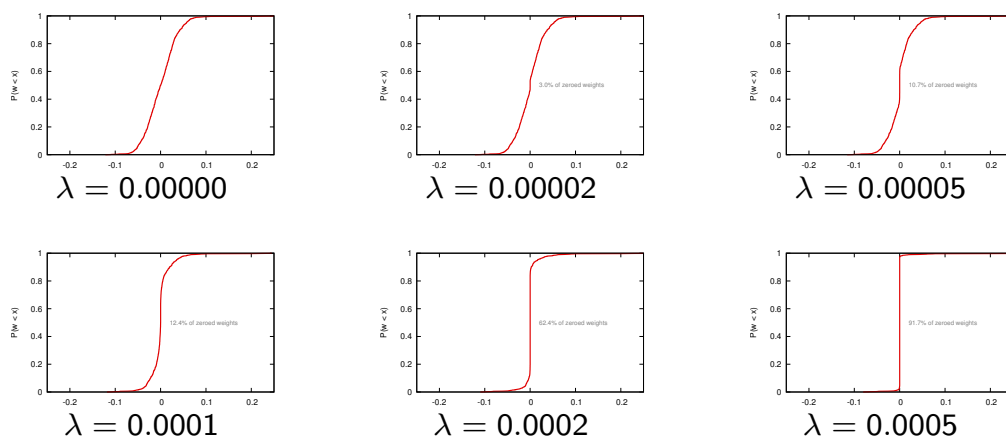
$\lambda$	Error	
	Train	Test
0.00000	0.000	0.064
0.00001	0.000	0.063
0.00002	0.000	0.067
0.00005	0.004	0.068
0.00010	0.087	0.128
0.00020	0.057	0.101
0.00050	0.496	0.516

```

output = model(train_input[b:b+batch_size])
loss = criterion(output, train_target[b:b+batch_size])

optimizer.zero_grad()
loss.backward()
optimizer.step()

with torch.no_grad():
    for p in model.parameters():
        p.sub_(p.sign() * p.abs().clamp(max = lambda_l1))
    
```



Penalties on the weights may be useful when dealing with small models and small data-sets and are still standard when data is scarce.

While they have a limited impact for large-scale deep learning, they may still provide the little push needed to beat baselines.