EE-559 – Deep learning

5. Losses, optimization, and initialization

François Fleuret
https://fleuret.org/dlc/
[version of: May 17, 2018]

Cross-entropy

We have used until now the Mean-Squared Error loss

$$\mathscr{L}(w) = \frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n; w))^2$$

even for classification, by converting a discrete class value into a one-hot vector with as many dimensions as there are classes.

This loss is justified with a Gaussian noise around a target value. Beside being conceptually wrong for classification, is has the undesirable property of penalizing a response "too strongly on the right side".

As we will see, the criterion of choice for classification is the cross-entropy.

We can generalize the logistic regression to a multi-class setup with $f_1, \ldots, f_C$ functionals that we interpret as

$$P(Y = y \mid X = x, W = w) = \frac{1}{Z} \exp f_y(x; w) = \frac{\exp f_y(x; w)}{\sum_k \exp f_k(x; w)},$$

from which

$$\begin{aligned}
&\log \mu_W(w \mid \mathscr{D} = \mathbf{d}) \\
&\qquad = \log \frac{\mu_{\mathscr{D}}(\mathbf{d} \mid W = w)\, \mu_W(w)}{\mu_{\mathscr{D}}(\mathbf{d})} \\
&\qquad = \log \mu_{\mathscr{D}}(\mathbf{d} \mid W = w) + \log \mu_W(w) - \log Z \\
&\qquad = \sum_n \log \mu_{\mathscr{D}}(x_n, y_n \mid W = w) + \log \mu_W(w) - \log Z \\
&\qquad = \sum_n \log P(Y = y_n \mid X = x_n, W = w) + \log \mu_W(w) - \log Z' \\
&\qquad = \underbrace{\sum_n \log \left(\frac{\exp f_{y_n}(x; w)}{\sum_k \exp f_k(x; w)} \right)}_{\text{Depends on the outputs}} + \underbrace{\log \mu_W(w)}_{\text{Depends on } w} - \log Z'.
\end{aligned}$$

If we ignore the penalty on w, it makes sense to minimize the average

$$\mathscr{L}(w) = -\frac{1}{N}\sum_{n=1}^{N} \log \underbrace{\left(\frac{\exp f_{y_n}(x_n; w)}{\sum_k \exp f_k(x_n; w)}\right)}_{P_w(Y=y_n|X=x_n)}.$$

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

$$\mathbb{H}(p, q) = -\sum_k p(k) \log q(k),$$

with the convention that $0 \log 0 = 0$. So we can re-write

$$\begin{aligned} -\log\left(\frac{\exp f_{y_n}(x_n; w)}{\sum_k \exp f_k(x_n; w)}\right) &= -\log P_w(Y = y_n \mid X = x_n) \\ &= -\sum_k \delta_{y_n}(k) \log P_w(Y = k \mid X = x_n) \\ &= \mathbb{H}\left(\delta_{y_n}, P_w(Y = \cdot \mid X = x_n)\right). \end{aligned}$$

So $\mathscr{L}$ above is the average of the cross-entropy between the deterministic "true" posterior δ_{y_n} and the estimated $P_w(Y = \cdot \mid X = x_n)$.

This is precisely the value of `torch.nn.CrossEntropyLoss`.

```
>>> f = Variable(Tensor([[-1, -3, 4], [-3, 3, -1]]))
>>> target = Variable(torch.LongTensor([0, 1]))
>>> criterion = torch.nn.CrossEntropyLoss()
>>> criterion(f, target)
```

prints

```
Variable containing:
 2.5141
[torch.FloatTensor of size 1]
```

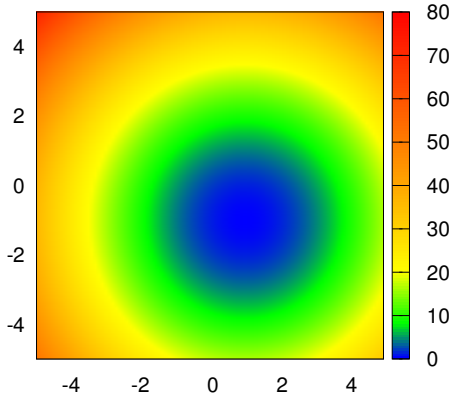
and indeed

$$-\frac{1}{2}\left(\log\frac{e^{-1}}{e^{-1}+e^{-3}+e^{4}} + \log\frac{e^{3}}{e^{-3}+e^{3}+e^{-1}}\right) \simeq 2.5141.$$

The range of values is 0 for perfectly classified samples, $\log(C)$ if the posterior is uniform, and up to $+\infty$ if the posterior distribution is "worst" than uniform.

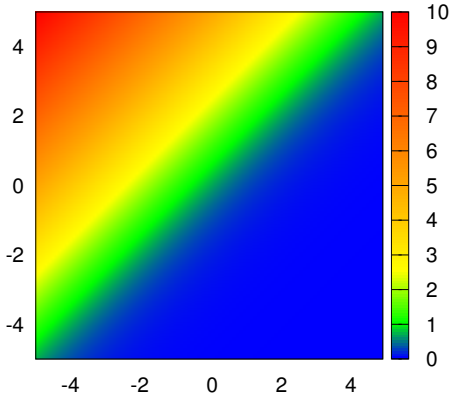
Let's consider the loss for a single sample in a two-class problem, with a predictor with two output values. The x axis here is the activation of the correct output unit, and the y axis is the activation of the other one.

MSE



$$\mathscr{L} = (x-1)^2 + (y+1)^2$$

Cross-entropy



$$\mathscr{L} = -\log\left(\frac{e^x}{e^x+e^y}\right)$$

MSE incorrectly penalizes outputs which are perfectly valid for prediction, contrary to cross-entropy.

The `torch.nn.NLLLoss` criterion implements

$$\mathscr{L}(w) = -\frac{1}{N}\sum_{n=1}^{N} f_{y_n}(x;w).$$

So combining it with the "log soft-max" function

$$(\alpha_1,\ldots,\alpha_C) \mapsto \left(\log\frac{\exp\alpha_1}{\sum_k \exp\alpha_k},\ldots,\log\frac{\exp\alpha_C}{\sum_k \exp\alpha_k}\right)$$

gives the `torch.nn.CrossEntropyLoss` criterion.

Hence, if a network should compute log-probabilities, it may have a `torch.nn.LogSoftmax` final layer, and be trained with `torch.nn.NLLLoss`.

The mapping

$$(\alpha_1, \ldots, \alpha_C) \mapsto \left(\frac{\exp \alpha_1}{\sum_k \exp \alpha_k}, \ldots, \frac{\exp \alpha_C}{\sum_k \exp \alpha_k} \right)$$

is called soft-max since it computes a "soft arg-max Boolean label."

```
>>> y = Variable(Tensor([[-10, -10, 10, -5 ],
                         [  3,   0,  0,  0 ],
                         [  1,   2,  3,  4 ]]))
>>> f = torch.nn.Softmax(1)
>>> f(y)
```

prints

```
Variable containing:
 2.0612e-09  2.0612e-09  1.0000e+00  3.0590e-07
 8.7005e-01  4.3317e-02  4.3317e-02  4.3317e-02
 3.2059e-02  8.7144e-02  2.3688e-01  6.4391e-01
[torch.FloatTensor of size 3x4]
```


PyTorch provides many other criteria, among which

- `torch.nn.MSELoss`
- `torch.nn.CrossEntropyLoss`
- `torch.nn.NLLLoss`
- `torch.nn.L1Loss`
- `torch.nn.NLLLoss2d`
- `torch.nn.MultiMarginLoss`

Stochastic gradient descent

So far, to minimize a loss of the form

$$\mathscr{L}(w) = \sum_{n=1}^{N} \underbrace{\ell(f(x_n; w), y_n)}_{\ell_n(w)}$$

we have considered the gradient-descent algorithm

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

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).
- It is an empirical estimation of an hidden quantity, and any partial sum would similarly be an unbiased empirical estimate, although more noisy.
- It is computed incrementally

$$\nabla \mathscr{L}(w_t) = \sum_{n=1}^{N} \nabla \ell_n(w_t),$$

and when we compute ℓ_n, we have already computed $\ell_1, \ldots, \ell_{n-1}$, and we could have a better estimate of w^* than w_t.

Also, consider that our training set is actually the same set of $M \ll N$ samples replicated K times. In that case

$$\begin{aligned}
\mathscr{L}(w) &= \sum_{n=1}^{N} \ell(f(x_n; w), y_n) \\
&= \sum_{k=1}^{K} \sum_{m=1}^{M} \ell(f(x_m; w), y_m) \\
&= K \sum_{m=1}^{M} \ell(f(x_m; w), y_m).
\end{aligned}$$

So instead of summing over all the samples and moving by η, we can visit only M samples and move by $K\eta$, which would cut the computation by K.

Although this is an ideal case, there is redundancy in practice that results in similar behaviors.

The **stochastic gradient descent** consists of updating the parameters w_t after every sample

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

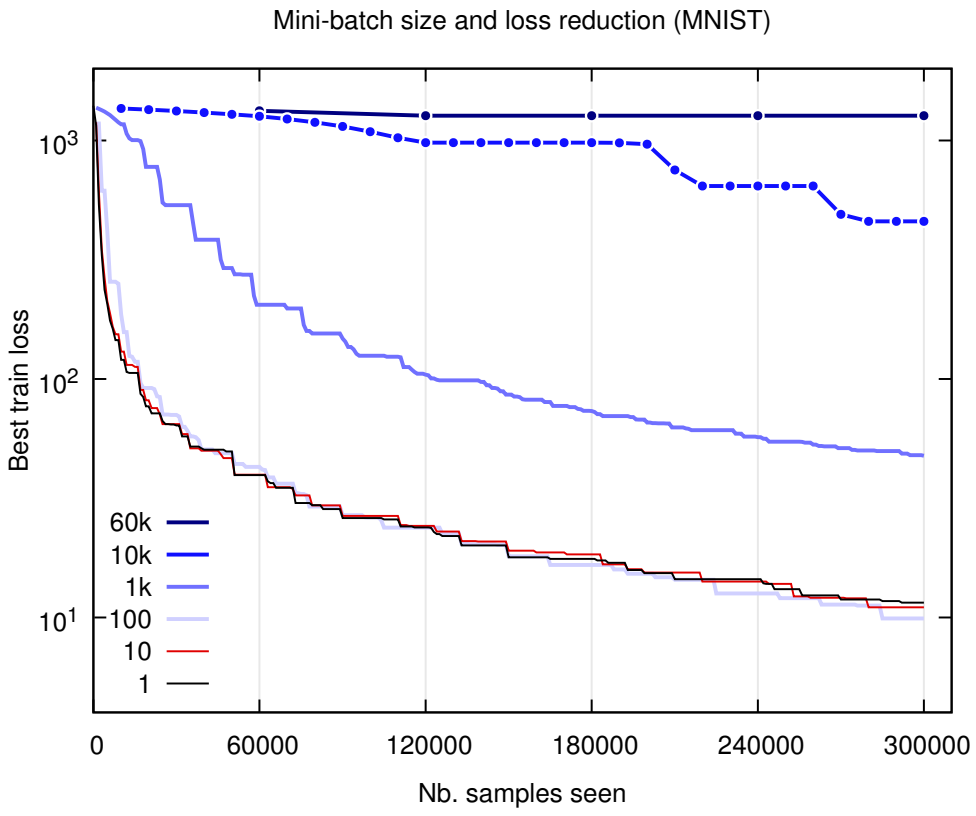
However this does not benefit from the speed-up of batch-processing.

The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time.

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t).$$

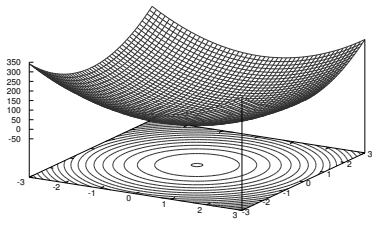
The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

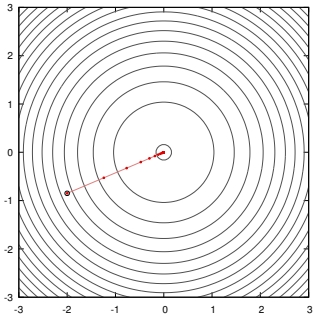
The stochastic behavior of this procedure helps evade local minima.



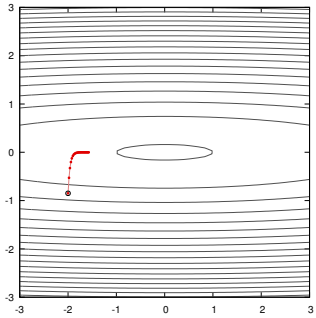
Limitation of the gradient descent

The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

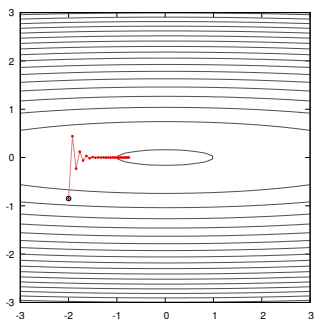




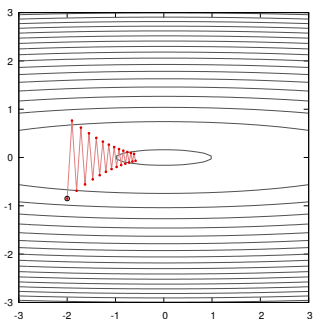
$\eta = 1.0e-2$



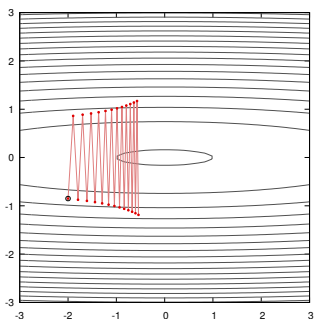
$\eta = 1.0e-2$



$\eta = 4.0e-2$



$\eta = 5.0e-2$



$\eta = 5.3e-2$

Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.

However the resulting computational overhead reduces the number of iterations for a fixed budget, and it is rarely at the advantage of these methods.

Deep-learning generally relies on a smarter use of the gradient, using statistics over its past values to make a "smarter step" with the current one.

Momentum and moment estimation

The "vanilla" mini-batch stochastic gradient descent (SGD) consists of

$$w_{t+1} = w_t - \eta g_t,$$

where

$$g_t = \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t)$$

is the gradient summed over a mini-batch.

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

$$\begin{aligned} u_t &= \gamma u_{t-1} + \eta g_t \\ w_{t+1} &= w_t - u_t. \end{aligned}$$

(Rumelhart et al., 1986)

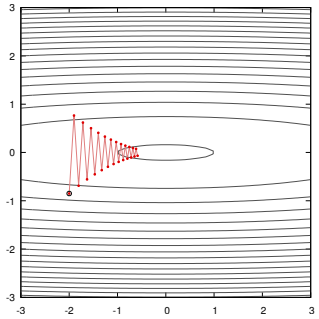
With $\gamma = 0$, this is the same as vanilla SGD.

With $\gamma > 0$, this update has three nice properties:

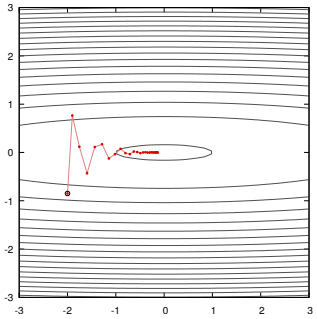
- it can "go through" local barriers,
- it accelerates if the gradient does not change much:

$$(u = \gamma u + \eta g) \Rightarrow \left(u = \frac{\eta}{1-\gamma} g\right),$$

- it dampens oscillations in narrow valleys.



$\eta = 5.0e-2, \gamma = 0$



$\eta = 5.0e-2, \gamma = 0.5$

Another class of methods exploits the statistics over the previous steps to compensate for the anisotropy of the mapping.

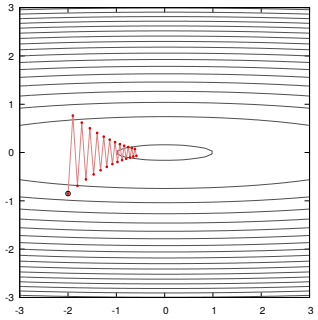
The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

The update rule is, **on each coordinate separately**

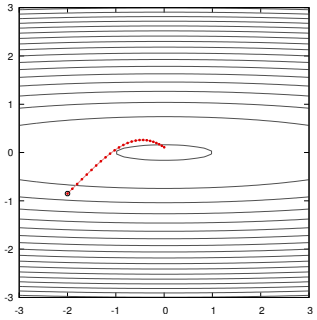
$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \frac{v_t}{1-\beta_2} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

(Kingma and Ba, 2014)

This can be seen as a combination of momentum, with $\hat{m}_t$, and a per-coordinate re-scaling with $\hat{v}_t$.



$\eta = 5.0e-2$



Adam,
$\beta_1 = 0.9, \beta_2 = 0.999,$
$\epsilon = 1e-8, \eta = 1.0e-1$

These two core strategies have been used in multiple incarnations:

- Nesterov's accelerated gradient,
- Adagrad,
- Adadelta,
- RMSprop,
- AdaMax,
- Nadam ...

`torch.optim`

PyTorch includes the many variants of the stochastic gradient descent.

We can implement the standard SGD as follows

```
for e in range(25):
    for b in range(0, train_input.size(0), mini_batch_size):
        output = model(train_input.narrow(0, b, mini_batch_size))
        loss = criterion(output, train_target.narrow(0, b, mini_batch_size))
        model.zero_grad()
        loss.backward()
        for p in model.parameters(): p.data -= eta * p.grad.data
```

which can be re-written with the `torch.optim` package

```
optimizer = torch.optim.SGD(model.parameters(), lr = eta)

for e in range(25):
    for b in range(0, train_input.size(0), mini_batch_size):
        output = model(train_input.narrow(0, b, mini_batch_size))
        loss = criterion(output, train_target.narrow(0, b, mini_batch_size))
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

An optimizer has an internal state to keep quantities such as moving averages, and operates on an iterator over `Parameter`s.

Values specific to the optimizer can be specified to its constructor, and the `step` method updates the internal state according to the `grad` attributes of the `Parameter`s, and updates the latter according to the internal state.

- `torch.optim.SGD` (momentum, and Nesterov's algorithm),
- `torch.optim.Adam`
- `torch.optim.Adadelta`
- `torch.optim.Adagrad`
- `torch.optim.RMSprop`
- `torch.optim.LBFGS`
- ...

An optimizer can also operate on several iterators, each corresponding to a group of `Parameter`s that should be handled similarly. For instance, different layers may have different learning rates or momentums.

So to use Adam with its default setting instead of vanilla SGD, we just have to change

```
optimizer = optim.SGD(model.parameters(), lr = eta)
```

into

```
optimizer = optim.Adam(model.parameters(), lr = eta)
```

The learning rate may have to be different if the functional was not properly scaled.

An example putting all this together

We now have the tools to define a deep network:

- fully connected layers,
- convolutional layers,
- pooling layers,
- ReLU.

And we have the tools to optimize it:

- Loss,
- back-propagation,
- stochastic gradient descent.

The only piece missing is the policy to initialize the parameters.

PyTorch initializes parameters with default rules when modules are created. They normalize weights according to the layer sizes (Glorot and Bengio, 2010) and behave usually very well. We will come back to this.

```
from torch import cuda, nn, optim
from torch.nn import functional as F
from torch.autograd import Variable
from torchvision import datasets

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = F.relu(self.fc1(x.view(-1, 256)))
        x = self.fc2(x)
        return x
```



```
train_set = datasets.MNIST('./data/mnist/', train = True, download = True)
train_input = Variable(train_set.train_data.view(-1, 1, 28, 28).float())
train_target = Variable(train_set.train_labels)

model, criterion = Net(), nn.CrossEntropyLoss()

if cuda.is_available():
    model.cuda()
    criterion.cuda()
    train_input, train_target = train_input.cuda(), train_target.cuda()

mu, std = train_input.data.mean(), train_input.data.std()
train_input.data.sub_(mu).div_(std)

lr, nb_epochs, batch_size = 1e-1, 10, 100

optimizer = optim.SGD(model.parameters(), lr = lr)

for k in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input.narrow(0, b, batch_size))
        loss = criterion(output, train_target.narrow(0, b, batch_size))
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

L_2 and L_1 penalties

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

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

If μ_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

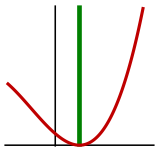
$$\lambda \|w\|_2^2.$$

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

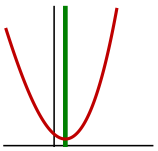
This is called the L_2 regularization, or "weight decay" in the artificial neural network community.

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

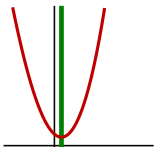
Since the derivative of $\|x\|_2^2$ is zero at zero, the optimal will never move there if it was not already there.



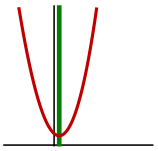
$$(x-1)^2 + \tfrac{1}{6}(x-1)^3$$



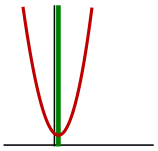
$$(x-1)^2 + \tfrac{1}{6}(x-1)^3 + x^2$$



$$(x-1)^2 + \tfrac{1}{6}(x-1)^3 + 2x^2$$



$$(x-1)^2 + \tfrac{1}{6}(x-1)^3 + 3x^2$$

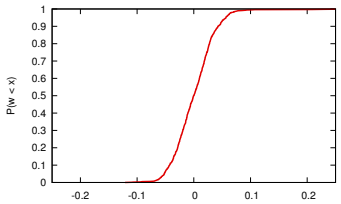


$$(x-1)^2 + \tfrac{1}{6}(x-1)^3 + 4x^2$$

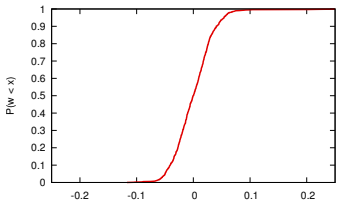
Convnet trained on MNIST with $1,000$ samples and a L_2 penalty.

	Error	
λ	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

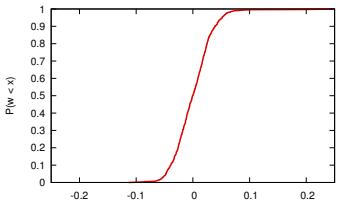
```
output = model(train_input.narrow(0, b, bs))
loss = criterion(output, train_target.narrow(0, b, bs))

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

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



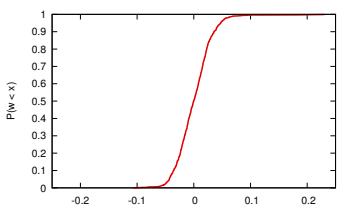
$\lambda = 0.000$



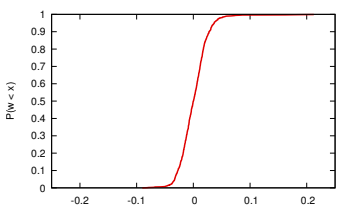
$\lambda = 0.001$



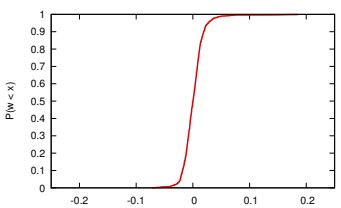
$\lambda = 0.002$



$\lambda = 0.004$



$\lambda = 0.010$



$\lambda = 0.020$

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 $\mathscr{L}$ is convex, and

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

then

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

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 λ parameter controls the sparsity of the solution.

With the L_1 penalty, the update rule becomes

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

where sign is applied per-component. This is almost identical to

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

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.

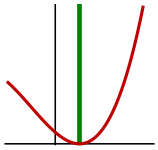
While this is not a problem in principle, since w_t will fluctuate around zero, it can be an issue if the zeroed weights are handled in a specific manner (*e.g.* sparse coding to reduce memory footprint or computation).

The **proximal operator** takes care of preventing parameters from "crossing zero", by adapting λ when it is too large

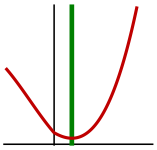
$$\begin{aligned} w'_t &= w_t - \eta g_t \\ w_{t+1} &= w'_t - \min(\lambda, |w'_t|) \odot \operatorname{sign}(w'_t). \end{aligned}$$

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

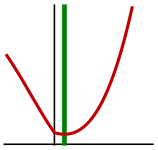
Increasing the λ parameter moves the optimal closer to 0, and away from the optimal for the loss without penalty.



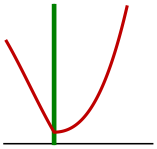
$$(x-1)^2 + \frac{1}{6}(x-1)^3$$



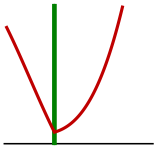
$$(x-1)^2 + \frac{1}{6}(x-1)^3 + \frac{1}{2}|x|$$



$$(x-1)^2 + \frac{1}{6}(x-1)^3 + |x|$$



$$(x-1)^2 + \frac{1}{6}(x-1)^3 + \frac{3}{2}|x|$$

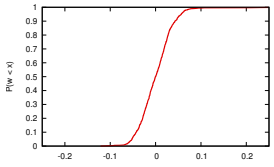


$$(x-1)^2 + \frac{1}{6}(x-1)^3 + 2|x|$$

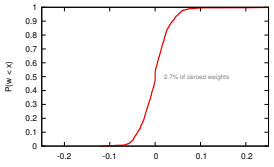
Convnet trained on MNIST with $1,000$ samples and a L_1 penalty.

	Error	
λ	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

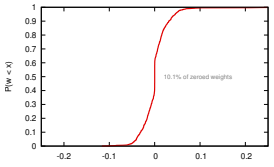
```
output = model(train_input.narrow(0, b, bs))
loss = criterion(output, train_target.narrow(0, b, bs))

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

for p in model.parameters():
    p.data -= p.data.sign() * p.data.abs().clamp(max = lambda_1)
```



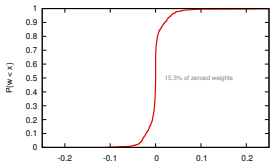
$\lambda = 0.00000$



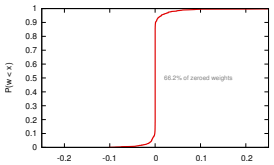
$\lambda = 0.00002$



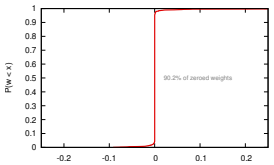
$\lambda = 0.00005$



$\lambda = 0.0001$



$\lambda = 0.0002$



$\lambda = 0.0005$

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.

Vanishing gradient

Consider the gradient estimation for a standard MLP:

Forward pass

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

Backward pass

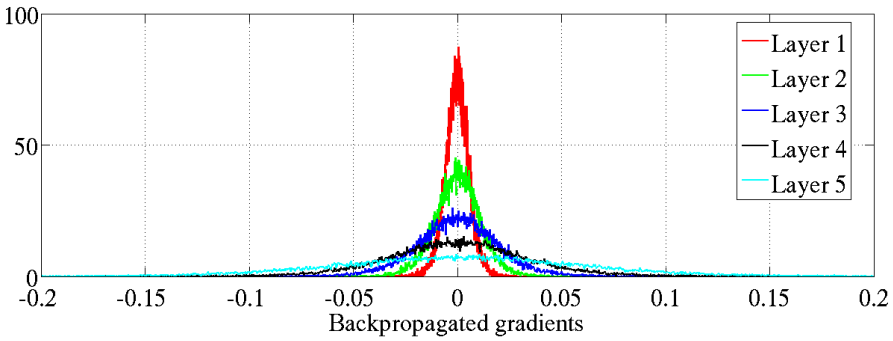
$$\begin{cases} \left[\frac{\partial \ell}{\partial x^{(L)}}\right] = \nabla_1 \ell\left(x^{(L)}\right) \\ \text{if } l < L, \left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \end{cases} \qquad \left[\frac{\partial \ell}{\partial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

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

We have

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

so the gradient "vanishes" exponentially with the depth if the ws are ill-conditioned or the activations are in the saturating domain of σ.



(Glorot and Bengio, 2010)

Weight initialization

The analysis for the weight initialization relies on controlling

$$\mathbb{V}\left(\frac{\partial \ell}{\partial w_{i,j}^{(l)}}\right) \text{ and } \mathbb{V}\left(\frac{\partial \ell}{\partial b_{i}^{(l)}}\right)$$

so that

- the gradient does not vanish, and
- weights evolve at the same rate across layers during training, and no layer reaches a saturation behavior before others.

We will use that, if A and B are independent

$$\mathbb{V}(AB) = \mathbb{V}(A)\,\mathbb{V}(B) + \mathbb{V}(A)\,\mathbb{E}(B)^2 + \mathbb{V}(B)\,\mathbb{E}(A)^2\,.$$

Notation in the coming slides will drop indexes when variances are identical for all activations or parameters in a layer.

In a standard layer

$$x_i^{(l)} = \sigma\left(\sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)}\right)$$

where N_l is the number of units in layer l, and σ is the activation function.

Assuming $\sigma'(0) = 1$, and we are in the linear regime

$$x_i^{(l)} \simeq \sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)}.$$

From which, if both the $w^{(l)}$s and $x^{(l-1)}$s are centered

$$\begin{aligned}\mathbb{V}\left(x_i^{(l)}\right) &\simeq \mathbb{V}\left(\sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} x_j^{(l-1)}\right)\\ &= \sum_{j=1}^{N_{l-1}} \mathbb{V}\left(w_{i,j}^{(l)}\right)\mathbb{V}\left(x_j^{(l-1)}\right)\end{aligned}$$

and if the $b^{(l)}$ are centered, so are the $x^{(l)}s$.

So if the $w_{i,j}^{(l)}$ are sampled i.i.d in each layer, and the $x_i^{(l)}$ have all the same variance for l fixed

$$\begin{aligned}\mathbb{V}\left(x^{(l)}\right) &\simeq \sum_{j=1}^{N_{l-1}} \mathbb{V}\left(w^{(l)}\right)\mathbb{V}\left(x^{(l-1)}\right)\\ &= N_{l-1}\mathbb{V}\left(w^{(l)}\right)\mathbb{V}\left(x^{(l-1)}\right).\end{aligned}$$

So **we have for the variance of the activations:**

$$\mathbb{V}\left(x^{(l)}\right) \simeq \mathbb{V}\left(x^{(0)}\right)\prod_{q=1}^{l} N_{q-1}\mathbb{V}\left(w^{(q)}\right).$$

This leads to a first type of initialization

$$\mathbb{V}\left(w^{(l)}\right) = \frac{1}{N_{l-1}}.$$

In `torch/nn/_functions/linear.py`

```
def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    self.weight.data.uniform_(-stdv, stdv)
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)
```

There is a slight mistake here: the standard deviation of $\mathcal{U}[-\delta,\delta] = \sqrt{3}\delta$, hence a $\sqrt{3}$ is missing.

We can look at the variance of the gradient wrt the activations. Since

$$\begin{aligned}\frac{\partial \ell}{\partial x_i^{(l)}} &= \sum_{h=1}^{N_{l+1}} \frac{\partial \ell}{\partial x_h^{(l+1)}} \frac{\partial x_h^{(l+1)}}{\partial x_i^{(l)}} \\ &\simeq \sum_{h=1}^{N_{l+1}} \frac{\partial \ell}{\partial x_h^{(l+1)}} w_{h,i}^{(l+1)}\end{aligned}$$

we get

$$\mathbb{V}\left(\frac{\partial \ell}{\partial x^{(l)}}\right) \simeq N_{l+1} \mathbb{V}\left(\frac{\partial \ell}{\partial x^{(l+1)}}\right) \mathbb{V}\left(w^{(l+1)}\right).$$

So **we have for the variance of the gradient wrt the activations:**

$$\mathbb{V}\left(\frac{\partial \ell}{\partial x^{(l)}}\right) \simeq \mathbb{V}\left(\frac{\partial \ell}{\partial x^{(L)}}\right) \prod_{q=l+1}^{L} N_q \mathbb{V}\left(w^{(q)}\right).$$

Since

$$x_i^{(l)} \simeq \sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)}$$

we have

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}^{(l)}} &= \frac{\partial \ell}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial w_{i,j}^{(l)}} \\ &\simeq \frac{\partial \ell}{\partial x_i^{(l)}} x_j^{(l-1)}\end{aligned}$$

and **we get the variance of the gradient wrt the weights**

$$\begin{aligned}\mathbb{V}\left(\frac{\partial \ell}{\partial w^{(l)}}\right) &\simeq \mathbb{V}\left(\frac{\partial \ell}{\partial x^{(l)}}\right) \mathbb{V}\left(x^{(l-1)}\right) \\ &= \mathbb{V}\left(\frac{\partial \ell}{\partial x^{(L)}}\right) \left(\prod_{q=l+1}^{L} N_q \mathbb{V}\left(w^{(q)}\right)\right) \mathbb{V}\left(x^{(0)}\right) \left(\prod_{q=1}^{l} N_{q-1} \mathbb{V}\left(w^{(q)}\right)\right) \\ &= \frac{N_0}{N_l} \left(\prod_{q=1}^{L} N_q \mathbb{V}\left(w^{(q)}\right)\right) \mathbb{V}\left(x^{(0)}\right) \mathbb{V}\left(\frac{\partial \ell}{\partial x^{(L)}}\right).\end{aligned}$$

Similarly, since

$$x_i^{(l)} \simeq \sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)}$$

we have

$$\begin{aligned}\frac{\partial \ell}{\partial b_i^{(l)}} &= \frac{\partial \ell}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial b_i^{(l)}} \\ &\simeq \frac{\partial \ell}{\partial x_i^{(l)}}\end{aligned}$$

so **we get the variance of the gradient wrt the biases**

$$\mathbb{V}\left(\frac{\partial \ell}{\partial b^{(l)}}\right) \simeq \mathbb{V}\left(\frac{\partial \ell}{\partial x^{(l)}}\right).$$

So finally, there is nothing we can do to control the variance of the gradient wrt the weights.

To control the variance of activations, we need

$$\mathbb{V}\left(w^{(l)}\right) = \frac{1}{N_{l-1}},$$

and to control the variance of the gradient wrt activations, and through it the variance of the gradient wrt the biases

$$\mathbb{V}\left(w^{(l)}\right) = \frac{1}{N_l}.$$

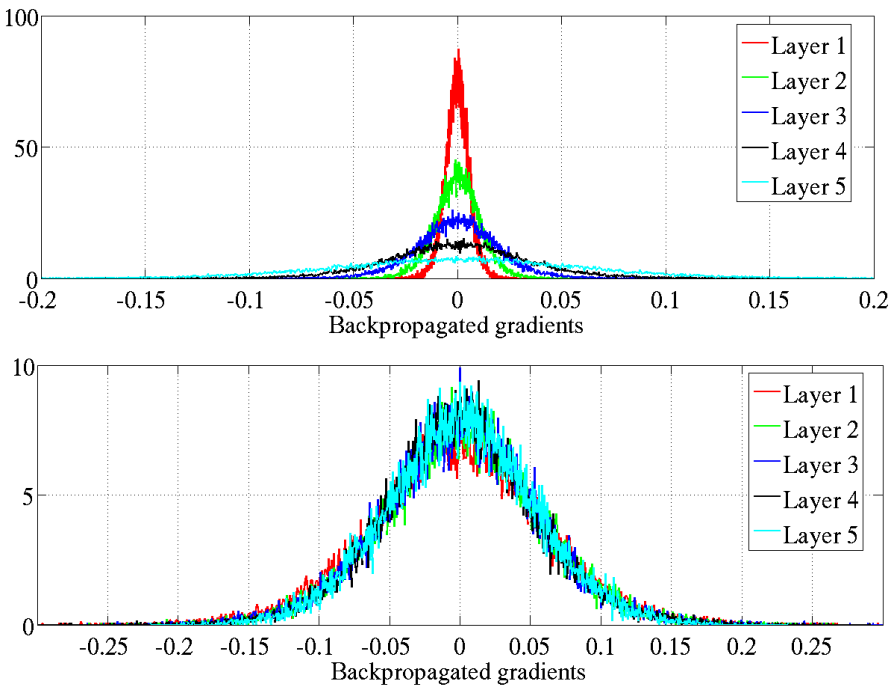
From which we get as a compromise the "Xavier initialization"

$$\mathbb{V}\left(w^{(l)}\right) = \frac{1}{\frac{N_{l-1}+N_l}{2}} = \frac{2}{N_{l-1}+N_l}.$$

(Glorot and Bengio, 2010)

In `torch/nn/init.py`

```
def xavier_normal(tensor, gain=1):
    if isinstance(tensor, Variable):
        xavier_normal(tensor.data, gain=gain)
        return tensor

    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / (fan_in + fan_out))
    return tensor.normal_(0, std)
```

(Glorot and Bengio, 2010)

The weights can also be scaled to account for the activation functions.

Remember that we have

$$\begin{aligned}\mathbb{V}(AB) &= \mathbb{V}(A)\,\mathbb{V}(B) + \mathbb{V}(A)\,\mathbb{E}(B)^2 + \mathbb{V}(B)\,\mathbb{E}(A)^2 \\ &= \mathbb{V}(A)\,\mathbb{E}\left(B^2\right) + \mathbb{V}(B)\,\mathbb{E}(A)^2 .\end{aligned}$$

For the forward pass, if

$$\begin{aligned}s_i^{(l)} &= \sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} \sigma\left(s_j^{(l-1)}\right) + b_i^{(l)} \\ x_i^{(l)} &= \sigma\left(s_i^{(l)}\right),\end{aligned}$$

and $\mathbb{E}\left(w^{(l)}\right) = 0$, $s^{(l-1)}$ is symmetric, and σ is ReLU, we have

$$\begin{aligned}\mathbb{V}\left(s_i^{(l)}\right) &= N_{l-1}\mathbb{V}\left(w^{(l)}\sigma\left(s^{(l-1)}\right)\right) \\ &= N_{l-1}\mathbb{V}\left(w^{(l)}\right)\mathbb{E}\left(\sigma\left(s^{(l-1)}\right)^2\right) \\ &= N_{l-1}\mathbb{V}\left(w^{(l)}\right)\frac{1}{2}\mathbb{E}\left(\left(s^{(l-1)}\right)^2\right) \\ &= \frac{1}{2}N_{l-1}\mathbb{V}\left(w^{(l)}\right)\mathbb{V}\left(s^{(l-1)}\right).\end{aligned}$$

For the backward

$$\begin{aligned}
\mathbb{V}\left(\frac{\partial \ell}{\partial x_i^{(l)}}\right) &= \sum_{h=1}^{N_{l+1}} \mathbb{V}\Big(\underbrace{\sigma'\left(s_h^{(l+1)}\right)}_{0/1} \underbrace{\frac{\partial \ell}{\partial x_h^{(l+1)}} w_{h,i}^{(l+1)}}_{\mathbb{E}(.)=0,\text{ symmetric}}\Big) \\
&= \sum_{h=1}^{N_{l+1}} \mathbb{E}\left(\sigma'\left(s_h^{(l+1)}\right)\left(\frac{\partial \ell}{\partial x_h^{(l+1)}} w_{h,i}^{(l+1)}\right)^2\right) \\
&= \sum_{h=1}^{N_{l+1}} \frac{1}{2}\mathbb{E}\left(\left(\frac{\partial \ell}{\partial x_h^{(l+1)}} w_{h,i}^{(l+1)}\right)^2\right) \\
&= \frac{1}{2}\sum_{h=1}^{N_{l+1}} \mathbb{V}\left(\frac{\partial \ell}{\partial x_h^{(l+1)}}\right)\mathbb{V}\left(w_{h,i}^{(l+1)}\right).
\end{aligned}$$

So ReLU impacts the forward and backward pass as if the weights had half their variances, which motivates multiplying them by a corrective gain of $\sqrt{2}$.

(He et al., 2015)

The same type of reasoning can be applied to other activation functions.

In `torch/nn/init.py`

```
if nonlinearity in linear_fns or nonlinearity == 'sigmoid':
    return 1
elif nonlinearity == 'tanh':
    return 5.0 / 3
elif nonlinearity == 'relu':
    return math.sqrt(2.0)
```

Data normalization

The analysis for the weight initialization relies on keeping the activation variance constant.

For this to be true, not only the variance has to remained unchanged through layers, but it has to be correct for the input too.

$$\mathbb{V}\left(x^{(0)}\right) = 1.$$

This can be done in several ways. Under the assumption that all the input components share the same statistics, we can do

```
mu, std = train_input.mean(), train_input.std()
train_input.sub_(mu).div_(std)
test_input.sub_(mu).div_(std)
```

Thanks to the magic of broadcasting we can normalize component-wise with

```
mu, std = train_input.mean(0), train_input.std(0)
train_input.sub_(mu).div_(std)
test_input.sub_(mu).div_(std)
```

Choice of the architecture and step size

Choosing the network structure is a difficult exercise. **There is no silver bullet.**

- Re-use something "well known, that works", or at least start from there,
- split feature extraction / inference (although this is debatable),
- modulate the capacity until it overfits a small subset, but does not overfit / underfit the full set,
- capacity increases with more layers, more channels, larger receptive fields, or more units,
- regularization to reduce the capacity or induce sparsity,
- identify common paths for siamese-like,
- identify what path(s) or sub-parts need more/less capacity,
- use prior knowledge about the "scale of meaningful context" to size filters / combinations of filters (e.g. knowing the size of objects in a scene, the max duration of a sound snippet that matters),
- grid-search all the variations that come to mind (and hopefully have farms of GPUs to do so).

We will re-visit this list with additional regularization / normalization methods.

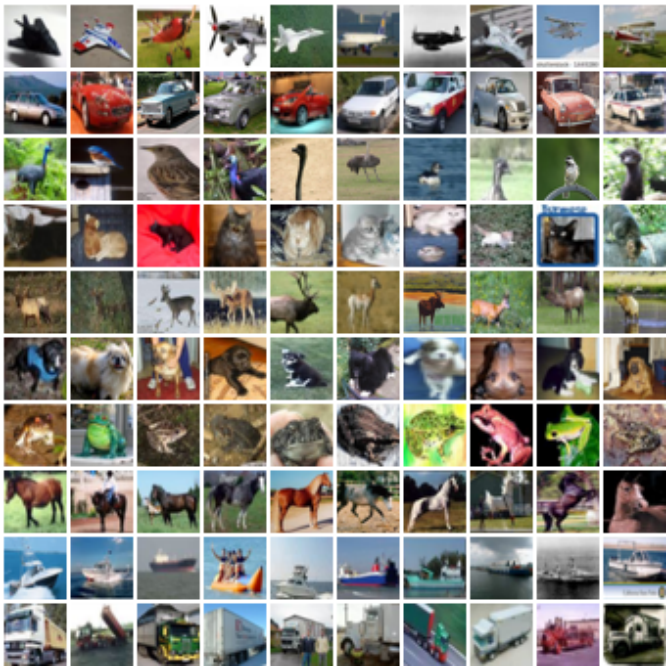
Regarding the learning rate, for training to succeed it has to

- reduce the loss quickly $\Rightarrow$ large learning rate,
- not be trapped in a bad minimum $\Rightarrow$ large learning rate,
- not bounce around in narrow valleys $\Rightarrow$ small learning rate, and
- not oscillate around a minimum $\Rightarrow$ small learning rate.

These constraints lead to a general policy of using a larger step size first, and a smaller one in the end.

The practical strategy is to look at the losses and error rates across epochs and pick a learning rate and learning rate adaptation. For instance by reducing it at discrete pre-defined steps, or with a geometric decay.

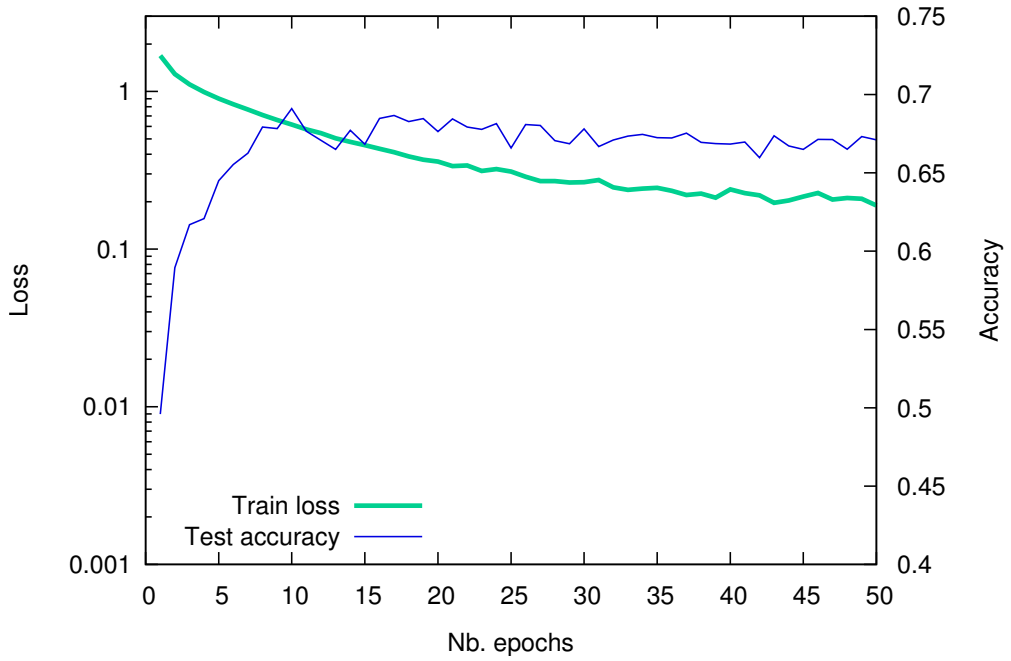
CIFAR10 data-set



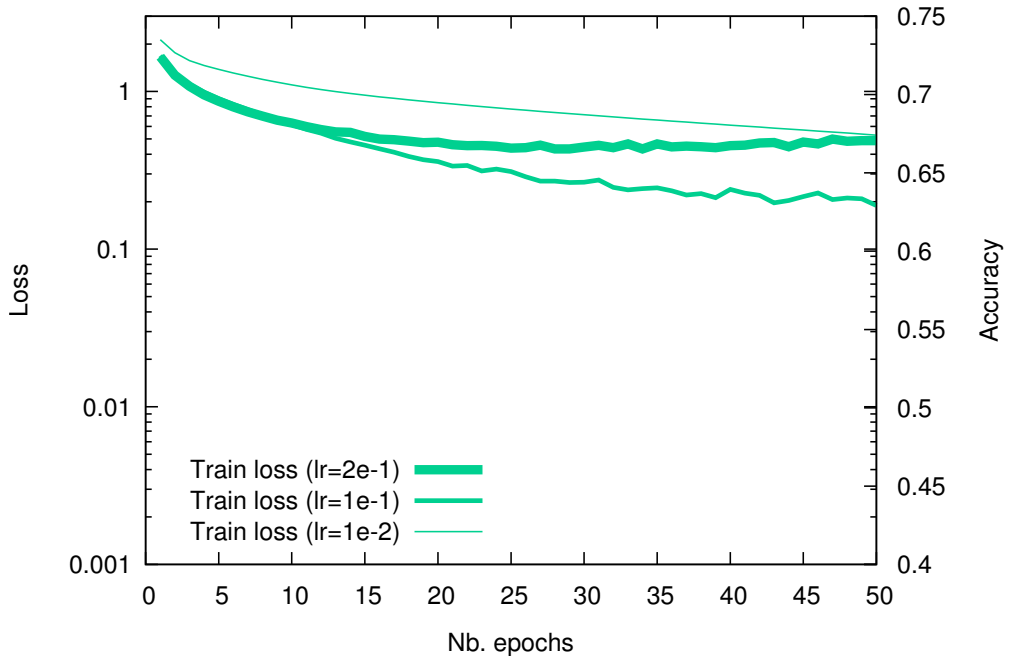
32×32 color images, $50,000$ train samples, $10,000$ test samples.

(Krizhevsky, 2009, chap. 3)

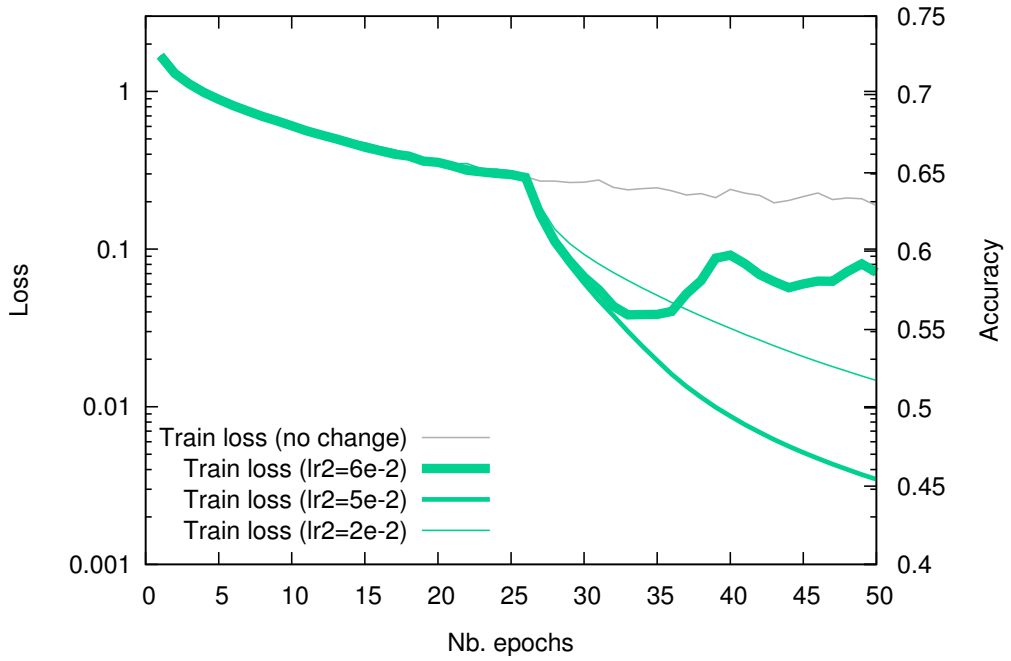
Small convnet on CIFAR10, cross-entropy, batch size 100, $\eta = 1e-1$.



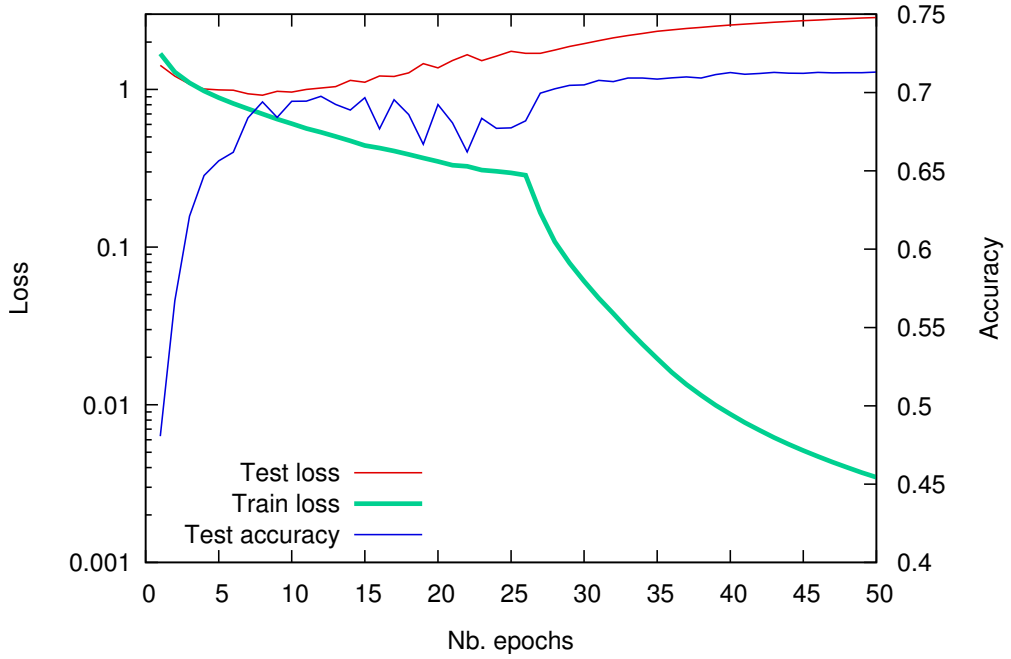
Small convnet on CIFAR10, cross-entropy, batch size 100



Using $\eta = 1e-1$ for 25 epochs, then reducing it.



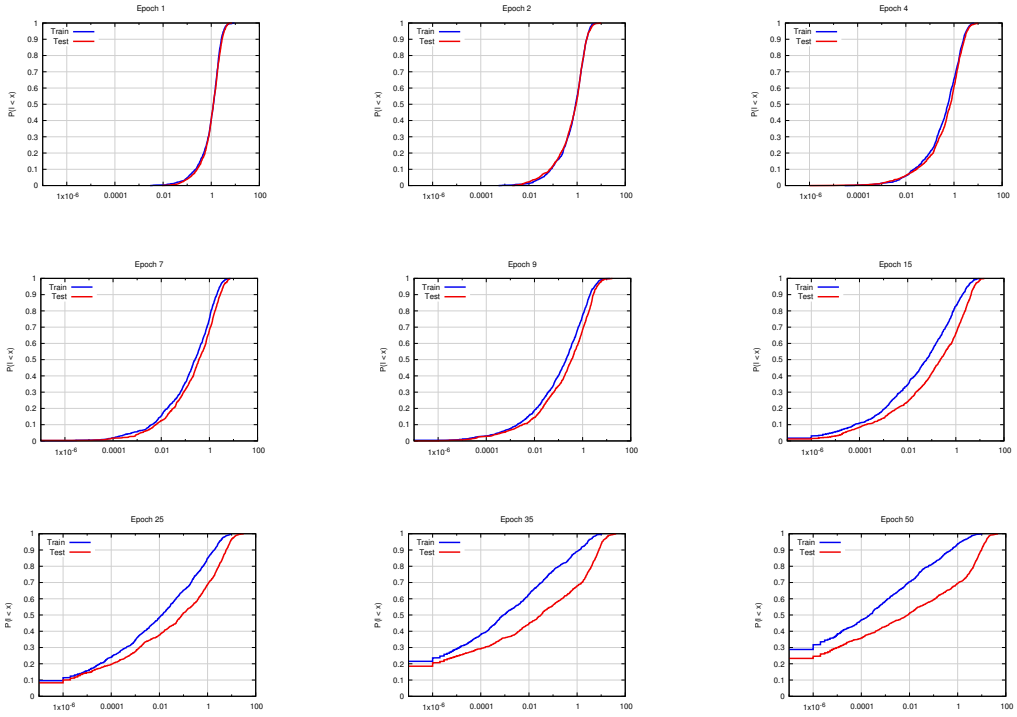
The test loss is a poor performance indicator, as it may increase even more on misclassified examples, and decrease less on the ones getting fixed.



We can plot the cdf of

$$\ell = -\log\left(\frac{\exp(f_Y(X;w))}{\sum_k \exp(f_k(X;w))}\right)$$

through epochs to visualize the over-fitting.



Writing a `torch.autograd.Function`

We have seen how to write new `torch.nn.Module`s. We may have to implement new functions usable with autograd, so that `Module`s remain defined through their forward pass alone.

This is achieved by writing sub-classes of `torch.autograd.Function`, which have to implement two static methods:

- `forward(...)` takes as argument a context to store information needed for the backward pass, and the quantities it should process, which are `Tensor`s for the differentiable ones, but can also be any other types. It should return one or several `Tensor`s.
- `backward(...)` takes as argument the context and as many `Variable`s as `forward` returns `Tensor`s, and it should return as many values as `forward` takes argument, `Variable`s for the tensors and `None` for the others.

Evaluating such a `Function` is done through its `apply(...)` method, which takes as many arguments as `forward(...)`, context excluded.

If you create a new `Function` named `Dummy`, when `Dummy.apply(...)` is called, autograd first adds a new node of type `DummyBackward` in its graph, and then calls `Dummy.forward(...)`.

To compute the gradient, autograd evaluates the graph and calls `Dummy.backward(...)` when it reaches the corresponding node, with the same context as the one given to `Dummy.forward(...)`.

This machinery is hidden to you and this level of details should not be required for normal operations.

Consider a function to set to zero the first n components of a tensor.

```
class KillHead(Function):
    @staticmethod
    def forward(ctx, input, n):
        ctx.n = n
        result = input.clone()
        result[:, 0:ctx.n] = 0
        return result

    @staticmethod
    def backward(ctx, grad_output):
        result = grad_output.clone()
        result[:, 0:ctx.n] = 0
        return result, None

killhead = KillHead.apply
```


It can be used for instance

```
y = Variable(Tensor(3, 8).normal_())
x = Variable(y.data.new(y.size()).normal_(), requires_grad = True)

criterion, eta = nn.MSELoss(), 1e-0

for k in range(10):
    r = killhead(x, 2)
    loss = criterion(r, y)
    if k > 0: x.grad.data.zero_()
    loss.backward()
    print(k, loss.data[0])
    x.data -= eta * x.grad.data
```

prints

```
0 0.9134871959686279
1 0.7928655743598938
2 0.6915099620819092
3 0.6063430905342102
4 0.5347792506217957
5 0.474645733833313
6 0.424116849899292
7 0.3816585838794708
8 0.3459818661212921
9 0.3160035014152527
```

The `torch.autograd.gradcheck(...)` function checks numerically that the backward function is correct, *i.e.*

$$\forall i, j, \left| \frac{f_i(x_1, \ldots, x_j + \epsilon, \ldots, x_D) - f_i(x_1, \ldots, x_j - \epsilon, \ldots, x_D)}{2\epsilon} - (J_f(x))_{i,j} \right| \leq \alpha$$

```
from torch.autograd import gradcheck

input = (Variable(torch.DoubleTensor(10, 20).uniform_(-1, 1), requires_grad=True), 4)

if gradcheck(KillHead.apply, input, eps = 1e-6, atol = 1e-4):
    print('All good captain.')
else:
    print('Ouch')
```

It is advisable to use `DoubleTensor`s for such a check.

Consider a function that takes two similar sized `Variable` and apply component-wise

$$(u, v) \mapsto |uv|.$$

The backward has to compute two tensors, and the forward must keep track of the input to compute the derivatives in the backward.

```
class Something(Function):
    @staticmethod
    def forward(ctx, input1, input2):
        ctx.save_for_backward(input1, input2)
        return (input1 * input2).abs()

    @staticmethod
    def backward(ctx, grad_output):
        input1, input2 = ctx.saved_variables
        return grad_output * input1.sign() * input2.abs(), \
               grad_output * input1.abs() * input2.sign()

something = Something.apply
```

The method `save_for_backward` is used to save `Tensor`s but the fields `saved_variables` are `Variable`s.

Practical session:

`https://fleuret.org/dlc/dlc-practical-5.pdf`

References

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto, 2009.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(9):533–536, 1986.