Deep learning

6.1. Benefits of depth

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Using deeper architectures has been key in improving performance in many applications. For instance image classification:

model	top-1 err.	top-5 err.
VGG-16 [41]	28.07	9.33
GoogLeNet [44]	-	9.15
PReLU-net [13]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	**21.43**	**5.71**

Table 3. Error rates (%, **10-crop** testing) on ImageNet validation. VGG-16 is based on our test. ResNet-50/101/152 are of option B that only uses projections for increasing dimensions.

(He et al., 2015)

"Notably, we did not depart from the classical ConvNet architecture of LeCun et al. (1989), but improved it by substantially increasing the depth."

(Simonyan and Zisserman, 2014)

A theoretical analysis provides an intuition of how a network's output "irregularity" grows:

- linearly with its width and
- exponentially with its depth.

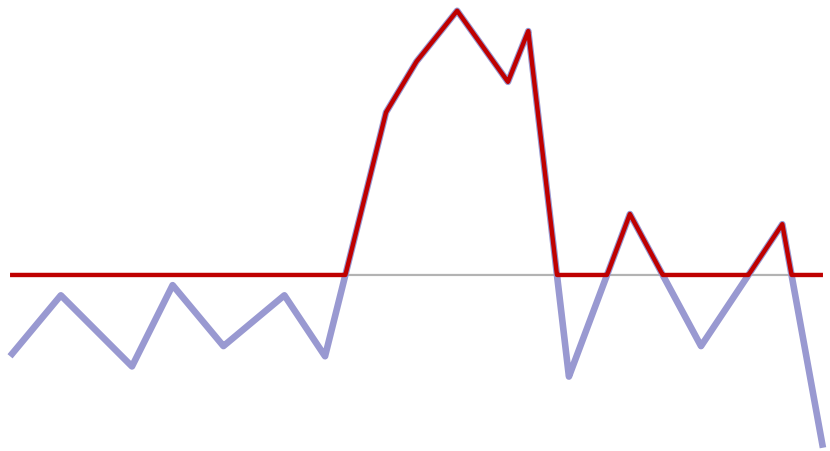
Let $\mathscr{F}$ be the set of piece-wise linear mappings on $[0,1]$, and $\forall f \in \mathscr{F}$, let $\kappa(f)$ be the minimum number of linear pieces in f.

Let $\mathscr{F}$ be the set of piece-wise linear mappings on $[0,1]$, and $\forall f \in \mathscr{F}$, let $\kappa(f)$ be the minimum number of linear pieces in f.

Let σ be the ReLU function

$$\begin{aligned}\sigma : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \max(0, x).\end{aligned}$$

Let $\mathscr{F}$ be the set of piece-wise linear mappings on $[0,1]$, and $\forall f \in \mathscr{F}$, let $\kappa(f)$ be the minimum number of linear pieces in f.



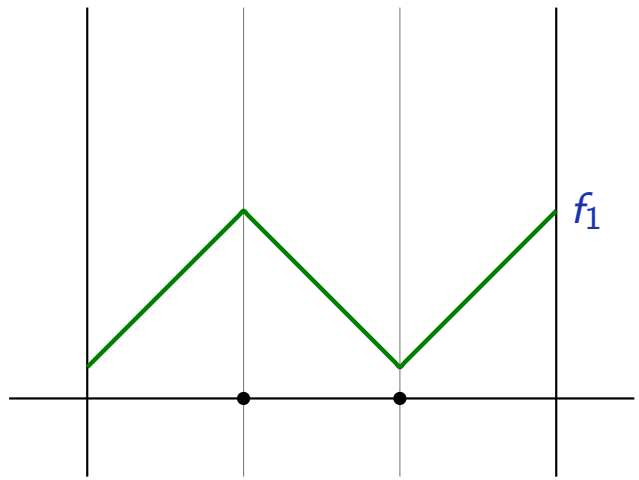
Let σ be the ReLU function

$$\begin{aligned}\sigma : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \max(0, x).\end{aligned}$$

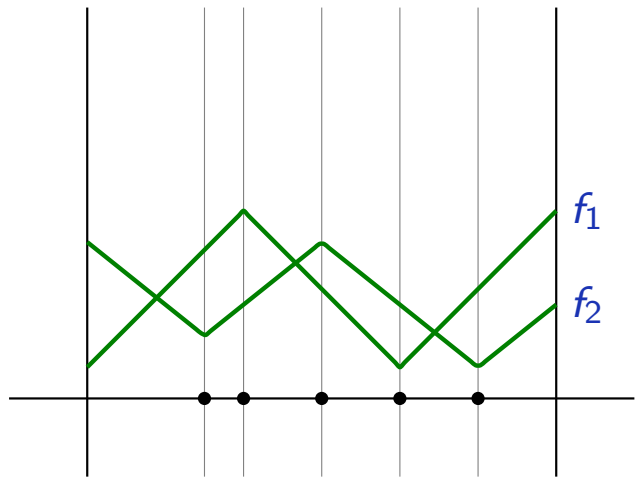
If we compose σ and $f \in \mathscr{F}$, any linear piece that does not cross 0 remains a single piece or disappears, and one that does cross 0 breaks into two, hence

$$\forall f \in \mathscr{F},\ \kappa(\sigma(f)) \leq 2\kappa(f)\,.$$

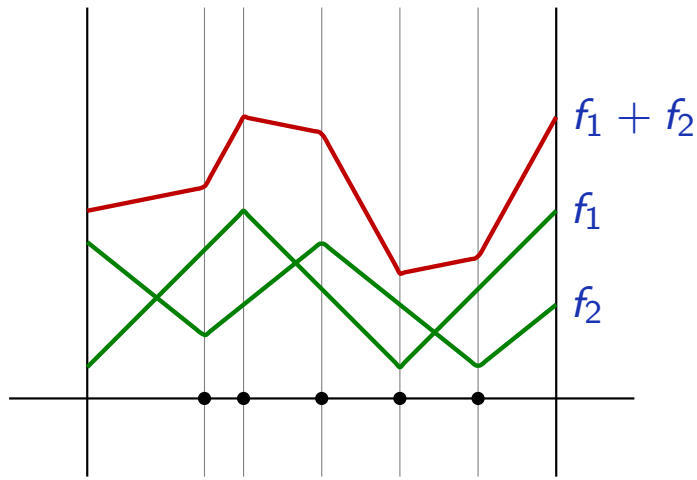
Also, when summing functions, a change of slope in the sum happens only if there was a change of slope in one of the operands.



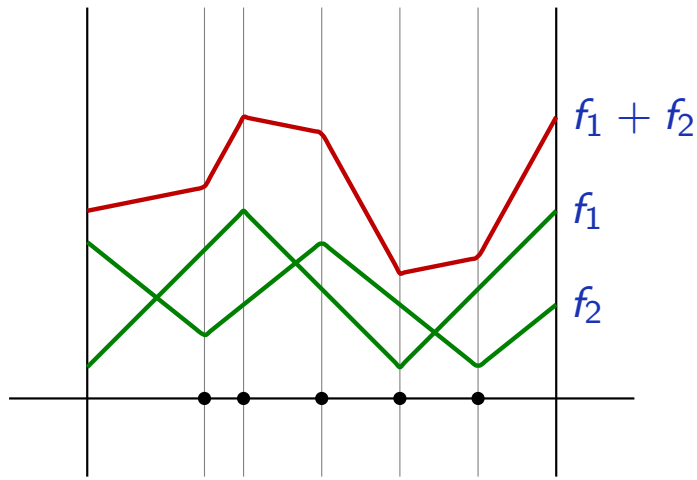
Also, when summing functions, a change of slope in the sum happens only if there was a change of slope in one of the operands.



Also, when summing functions, a change of slope in the sum happens only if there was a change of slope in one of the operands.



Also, when summing functions, a change of slope in the sum happens only if there was a change of slope in one of the operands.



Hence

$$\forall f_n \in \mathscr{F}, n = 1, \ldots, N, \ \kappa\left(\sum_n f_n\right) \leq \sum_n \kappa(f_n)\,.$$

Consider a MLP with ReLU, D layers, a single input unit, and a single output unit.

$$x_1^0 = x,$$

$$\forall d = 1, \ldots, D, \forall i, \quad \begin{cases} s_i^d & = \sum_{j=1}^{W^{(d-1)}} w_{i,j}^d x_j^{d-1} + b_i^d \\ x_i^d & = \sigma(s_i^d) \end{cases}$$

$$y = x_1^D.$$

All the s_i^ds and x_i^ds are piece-wise linear functions of x with $\forall i, \kappa\left(s_i^1\right) = 1$, and

$$\forall d, i, \kappa\left(x_i^d\right) = \kappa\left(\sigma(s_i^d)\right) \leq 2\kappa\left(s_i^d\right) \leq 2 \sum_{j=1}^{W^{(d-1)}} \kappa\left(w_{i,j}^d x_j^{d-1} + b_i^d\right) = 2 \sum_{j=1}^{W^{(d-1)}} \kappa\left(x_j^{d-1}\right)$$

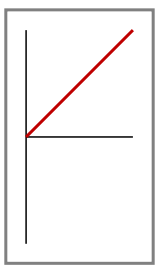
from which

$$\forall d, \max_i \kappa\left(x_i^d\right) \leq 2W^{(d-1)} \max_j \kappa\left(x_j^{d-1}\right)$$

and we get the following bound for any ReLU MLP

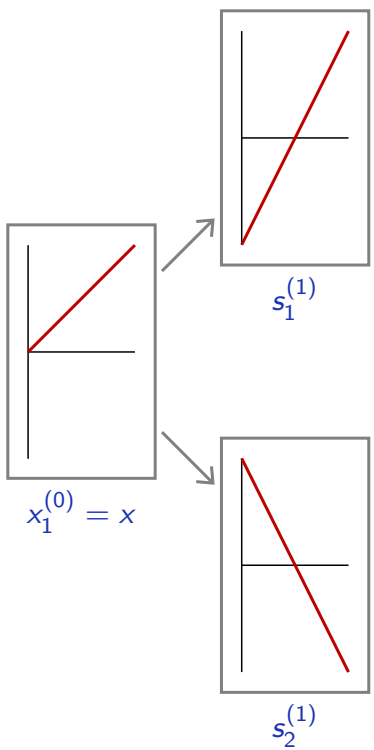
$$\kappa(y) \leq 2^D \prod_{d=1}^{D} W^{(d)}.$$

Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:

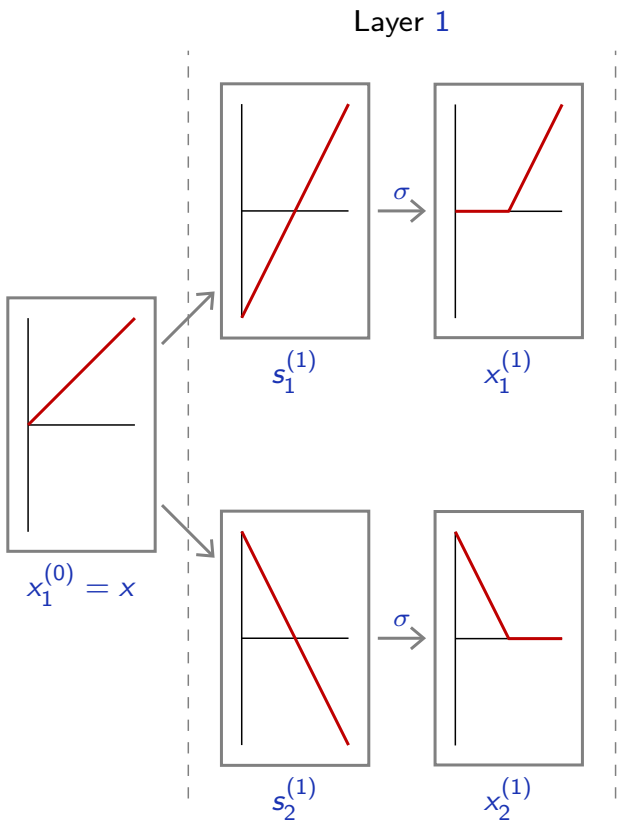


$x_1^{(0)} = x$

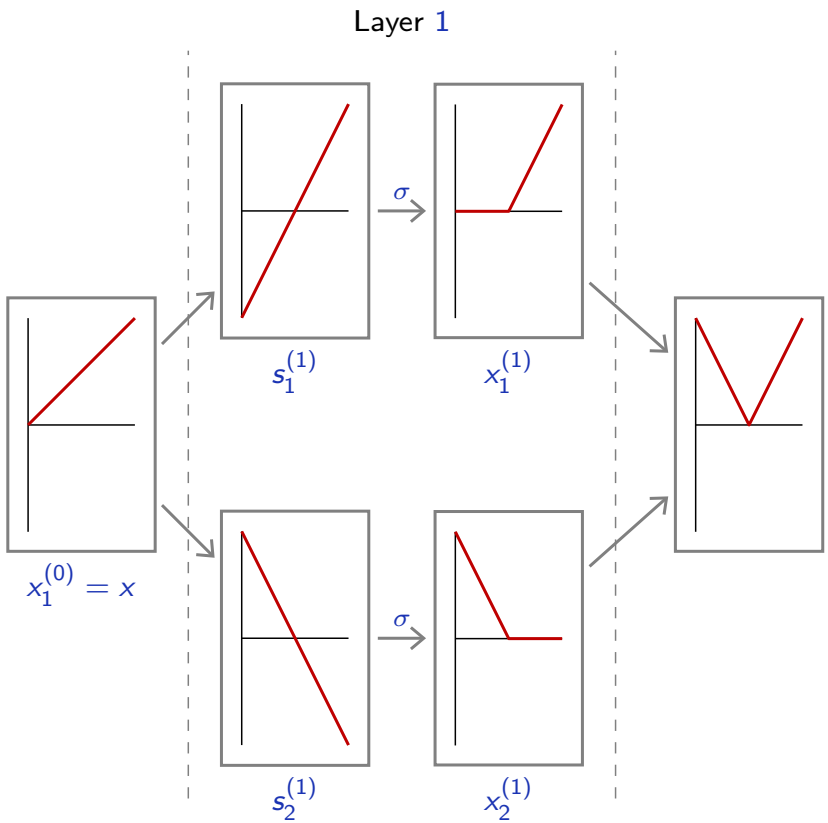
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



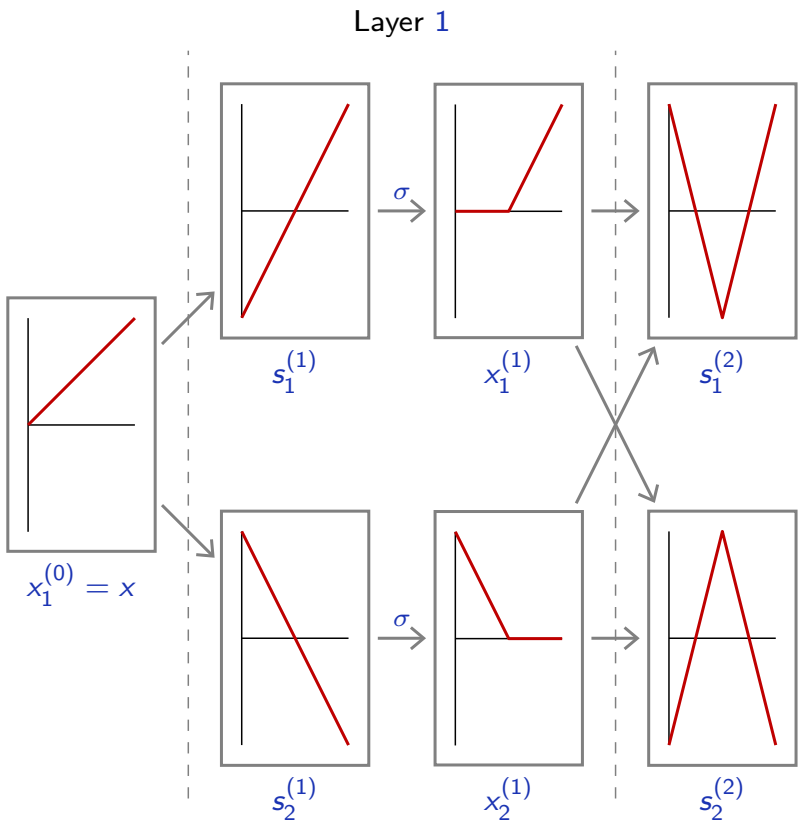
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



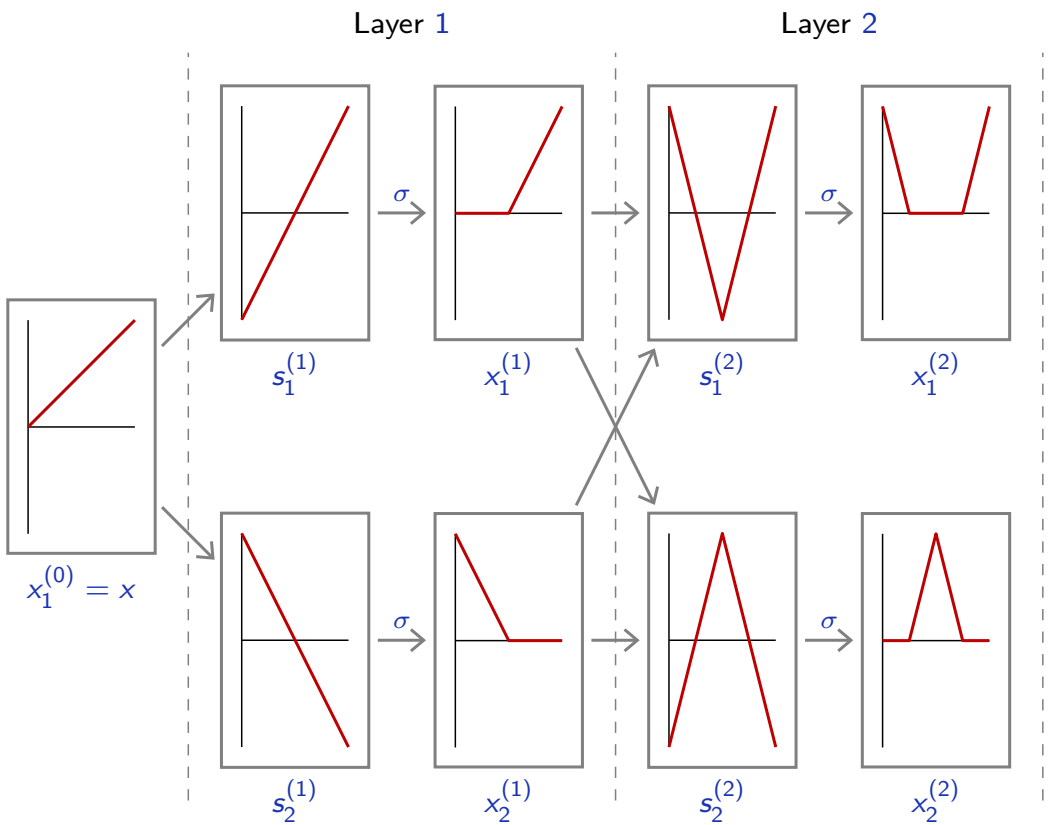
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



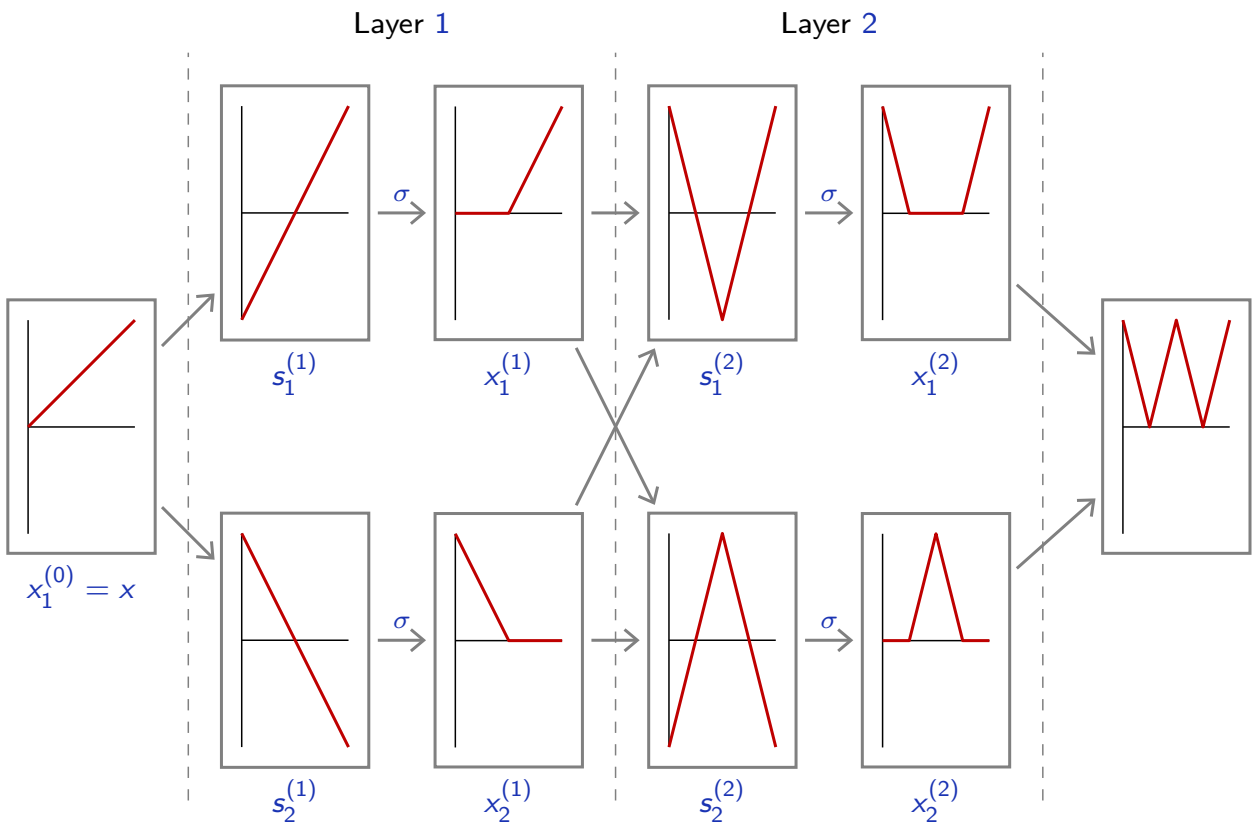
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



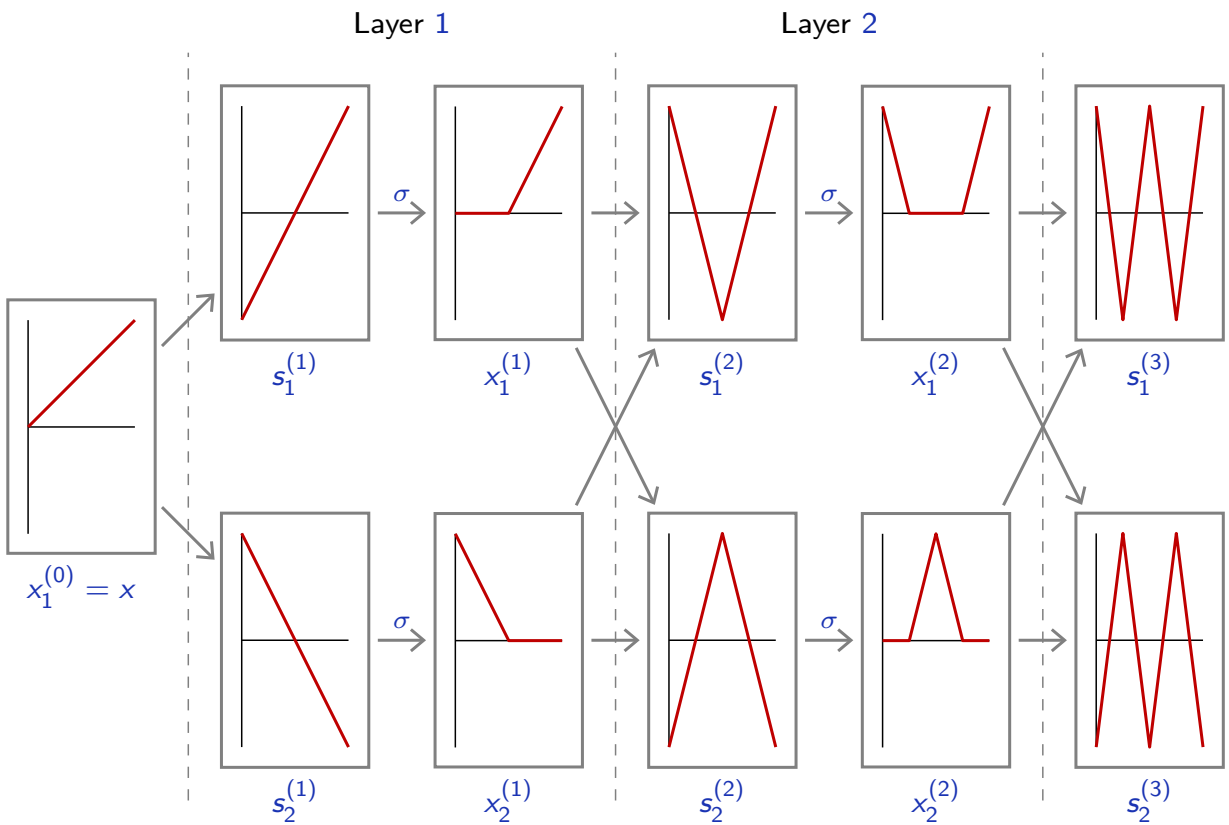
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



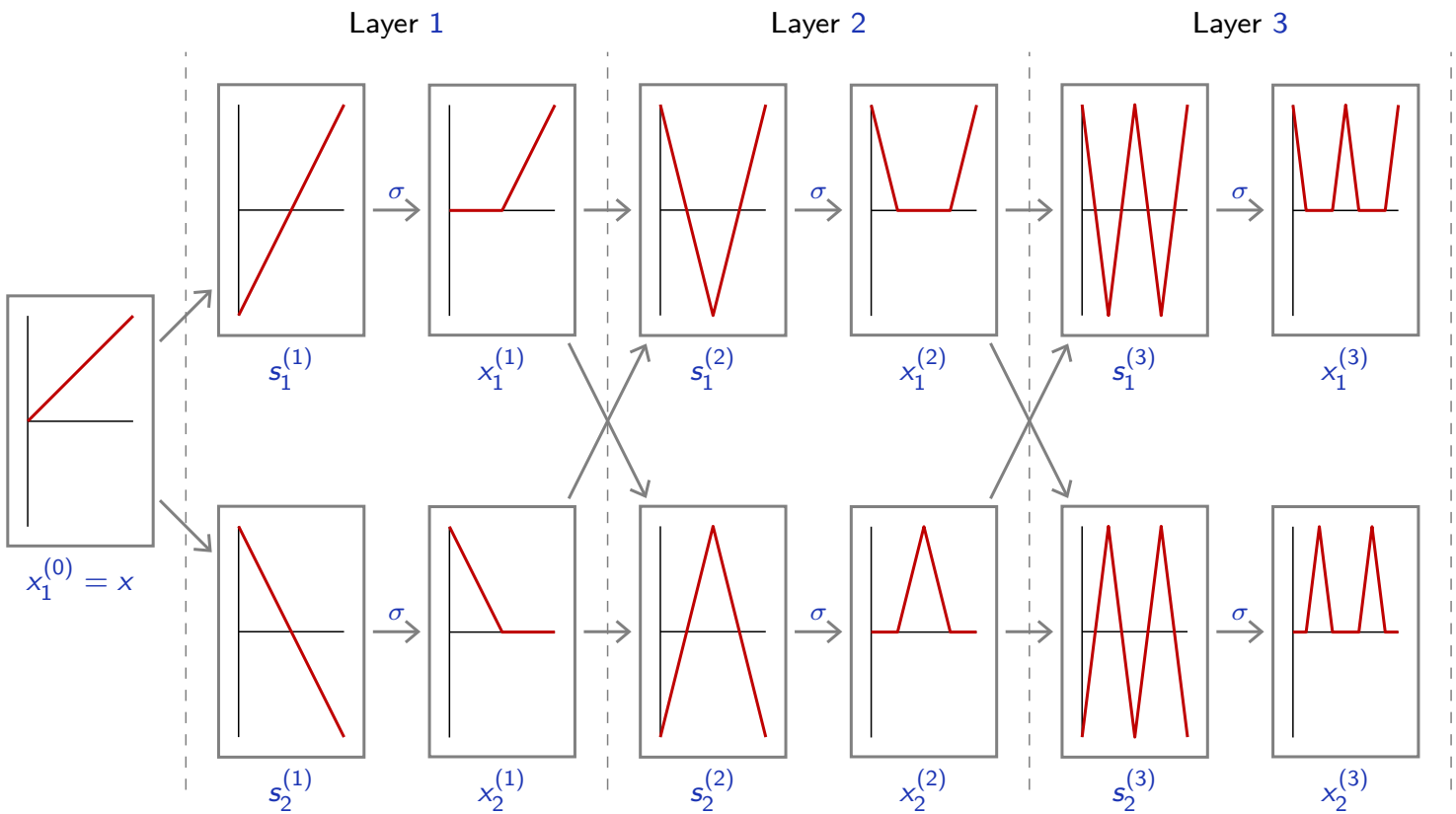
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



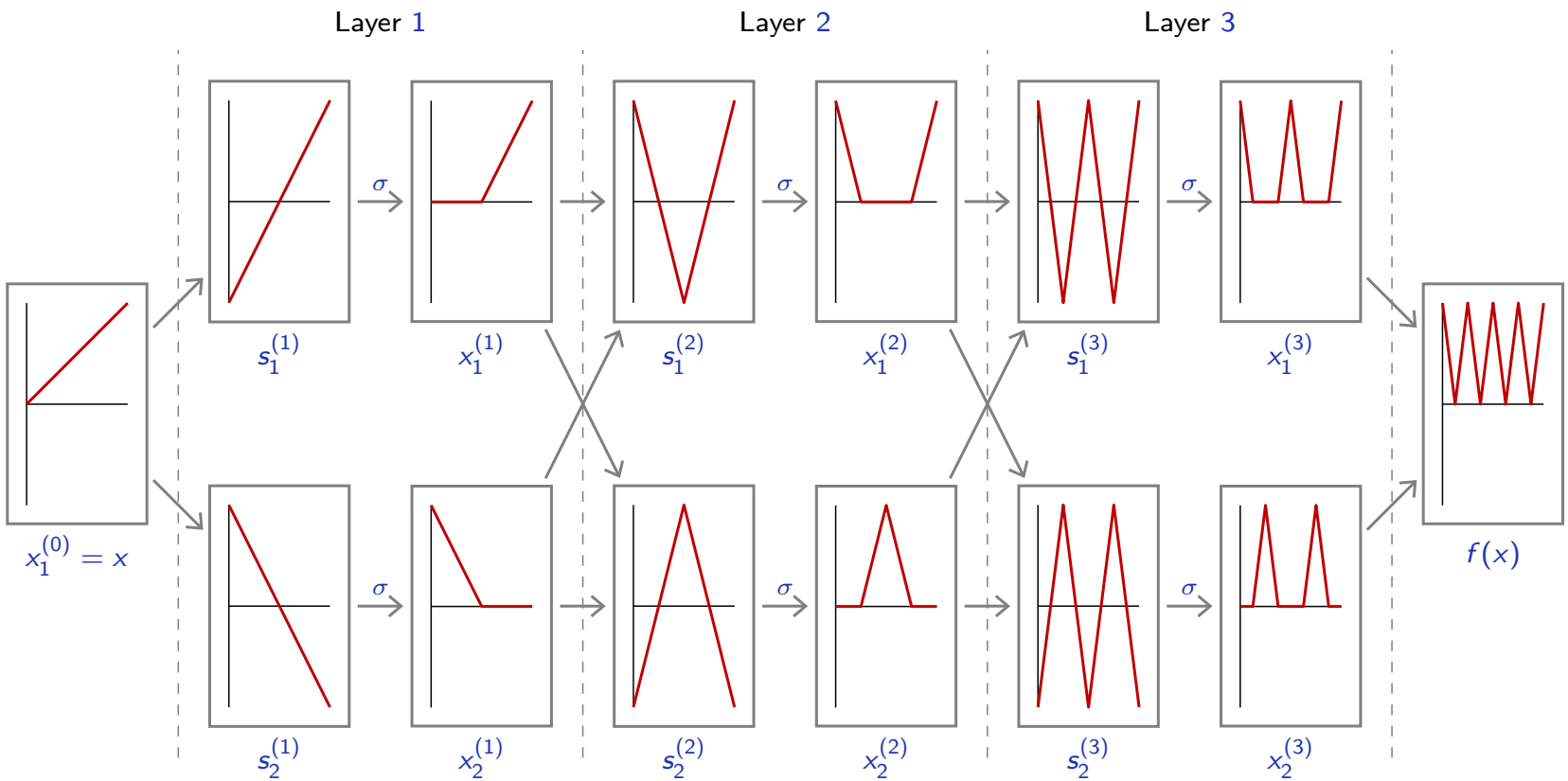
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:



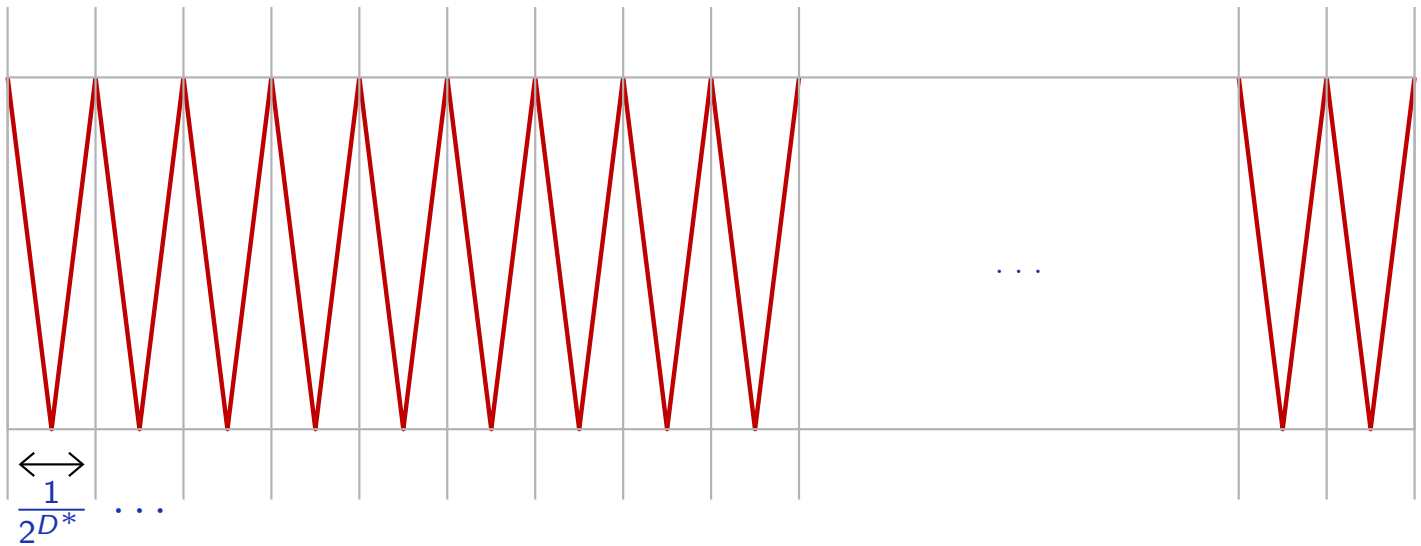
Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:

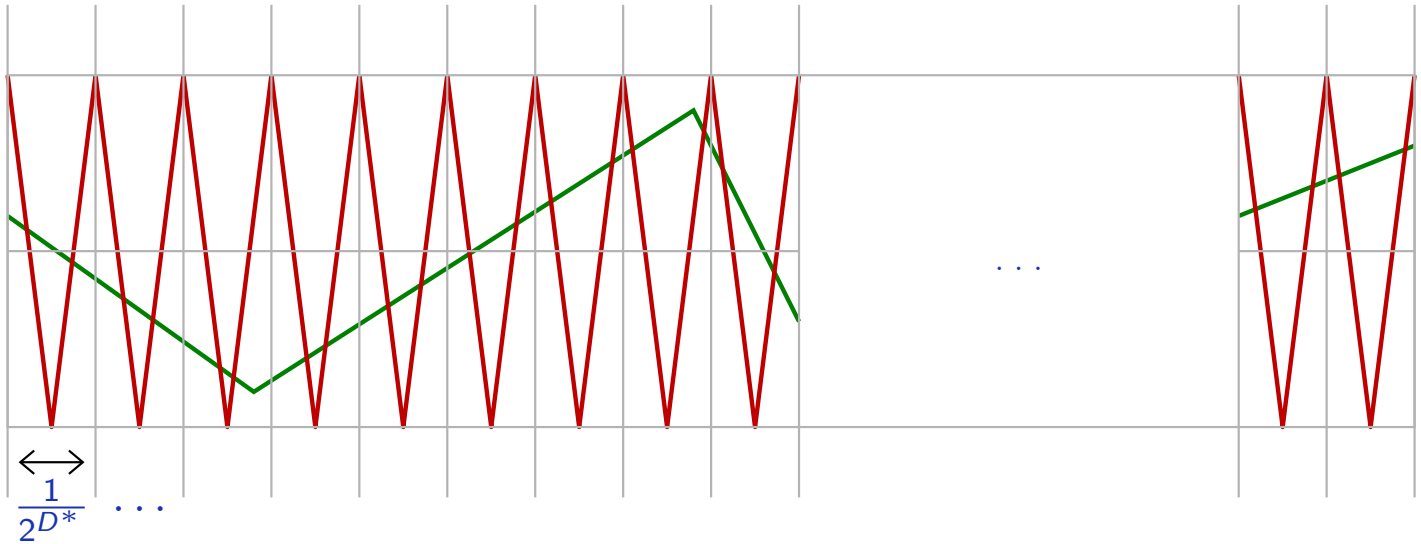


Although this seems quite a pessimistic bound, we can hand-design a network that [almost] reaches it:

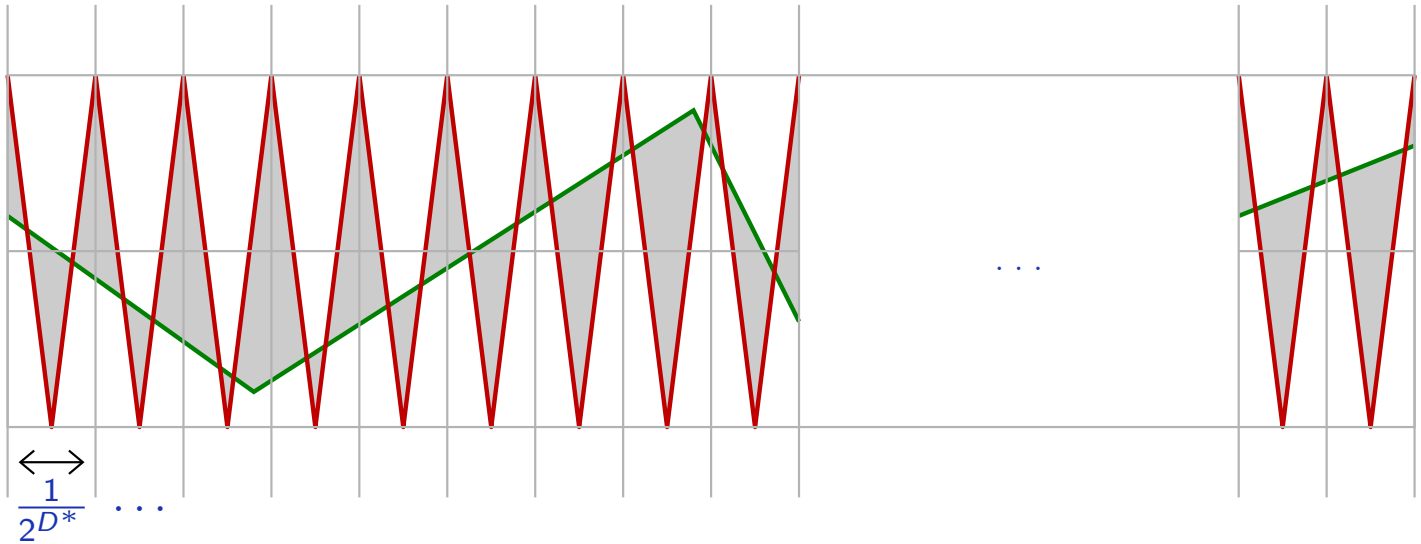


So for any D^*, there is a network with D^* hidden layers and $2D^*$ hidden units which computes an $f : [0,1] \to [0,1]$ of period $1/2^{D^*}$



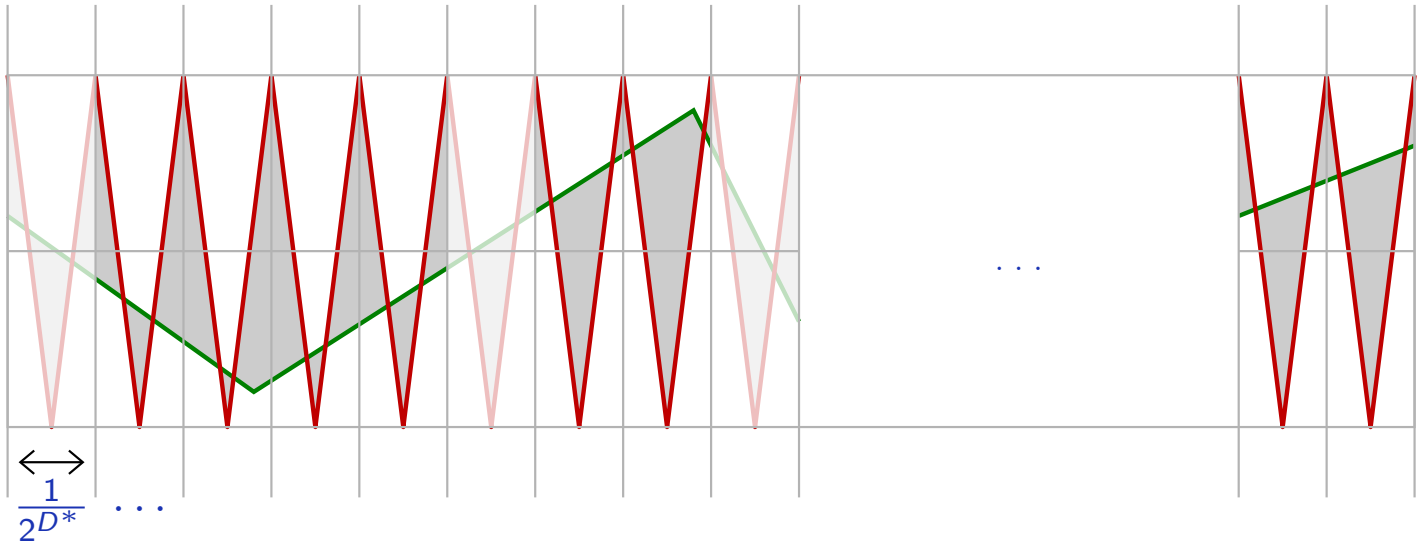


Given $g \in \mathscr{F}$



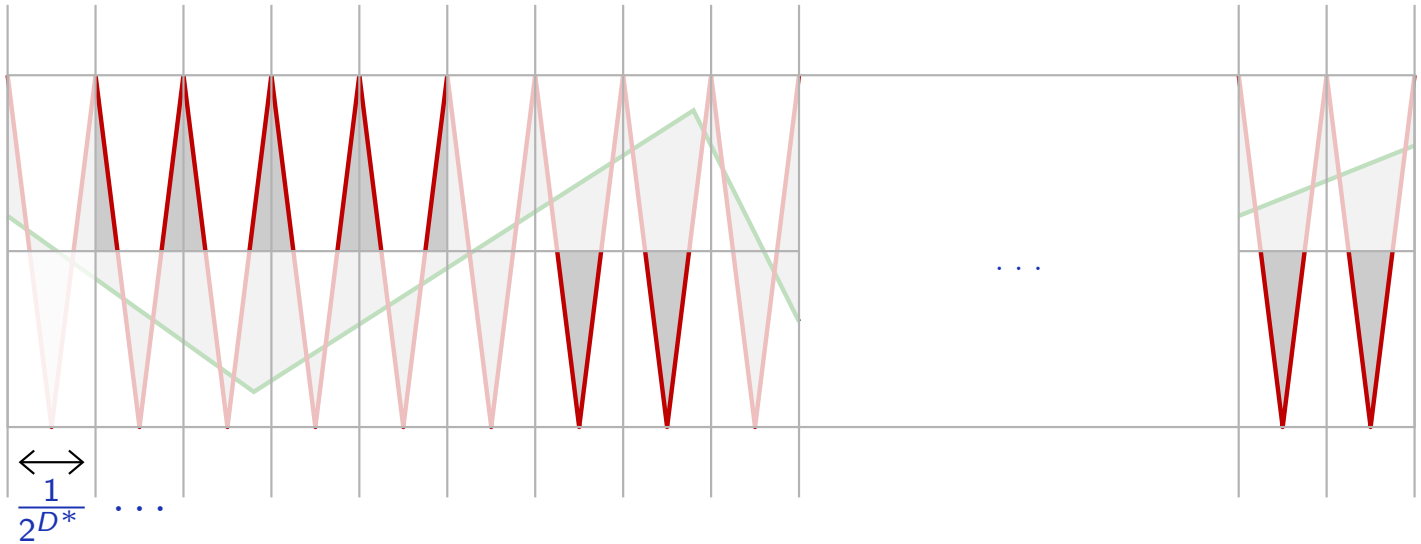
Given $g \in \mathscr{F}$

$$\|f - g\|_1 = \int_0^1 |f(x) - g(x)|$$



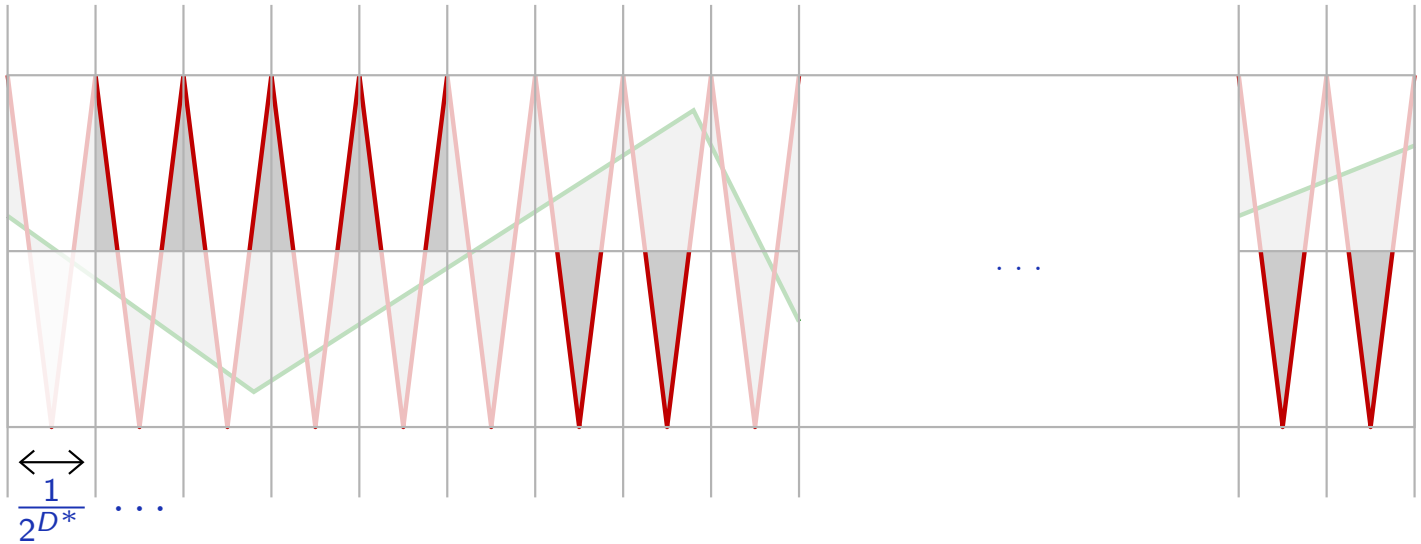
Given $g \in \mathscr{F}$, it crosses $\frac{1}{2}$ at most $\kappa(g)$ times

$$\|f - g\|_1 = \int_0^1 |f(x) - g(x)|$$



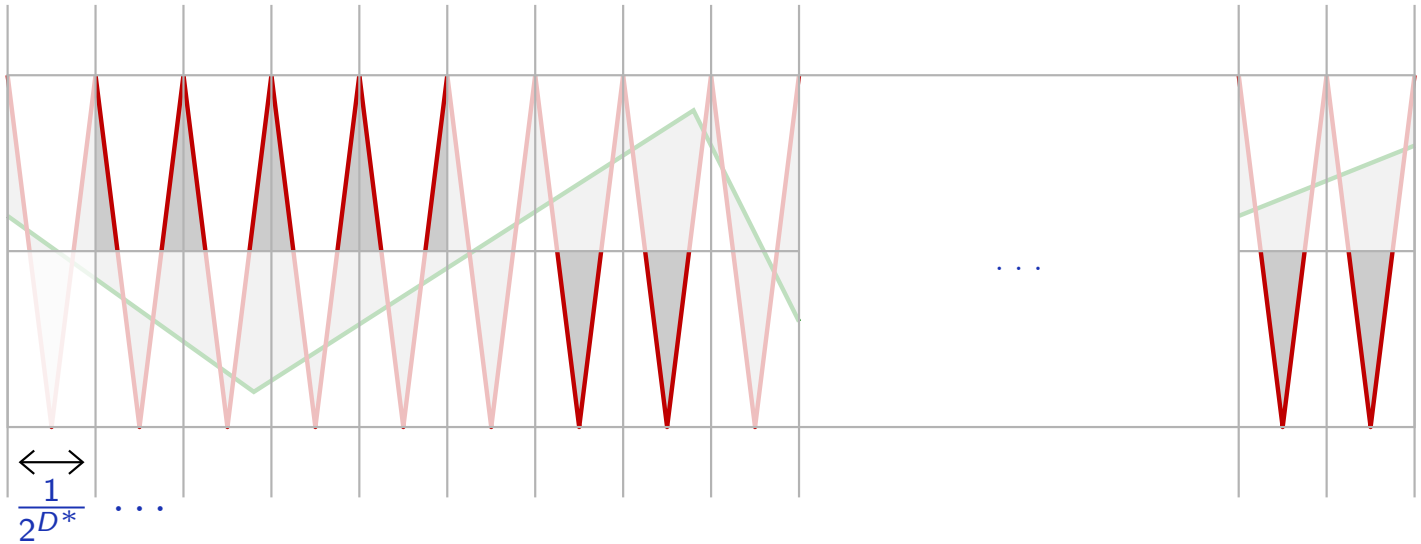
Given $g \in \mathscr{F}$, it crosses $\frac{1}{2}$ at most $\kappa(g)$ times, which means that on at least $2^{D^*} - \kappa(g)$ segments of length $1/2^{D^*}$, it is on one side of $\frac{1}{2}$, and

$$\|f - g\|_1 = \int_0^1 |f(x) - g(x)|$$



Given $g \in \mathscr{F}$, it crosses $\frac{1}{2}$ at most $\kappa(g)$ times, which means that on at least $2^{D^*} - \kappa(g)$ segments of length $1/2^{D^*}$, it is on one side of $\frac{1}{2}$, and

$$\begin{aligned}
\|f - g\|_1 &= \int_0^1 |f(x) - g(x)| \\
&\geq \left(2^{D^*} - \kappa(g)\right) \frac{1}{2} \frac{1}{2^{D^*}} \int_0^1 \left| f(x) - \frac{1}{2} \right| \\
&= \frac{1}{16} \left(1 - \frac{\kappa(g)}{2^{D^*}}\right).
\end{aligned}$$



Given $g \in \mathcal{F}$, it crosses $\frac{1}{2}$ at most $\kappa(g)$ times, which means that on at least $2^{D^*} - \kappa(g)$ segments of length $1/2^{D^*}$, it is on one side of $\frac{1}{2}$, and

$$\begin{aligned}\|f-g\|_1 &= \int_0^1 |f(x)-g(x)| \\ &\geq \left(2^{D^*} - \kappa(g)\right) \frac{1}{2} \frac{1}{2^{D^*}} \int_0^1 \left|f(x) - \frac{1}{2}\right| \\ &= \frac{1}{16}\left(1 - \frac{\kappa(g)}{2^{D^*}}\right).\end{aligned}$$

And we multiply f by 16 to get rid of the $\frac{1}{16}$.

So, considering ReLU MLPs with a single input/output, there exists a network f with D^* layers, and $2D^*$ internal units, such that, for any network g with D layers of sizes $\{W^{(1)}, \dots, W^{(D)}\}$, since $\kappa(g) \leq 2^D \prod_{d=1}^{D} W^{(d)}$:

$$\|f - g\|_1 \geq 1 - \frac{2^D}{2^{D^*}} \prod_{d=1}^{D} W^{(d)}.$$

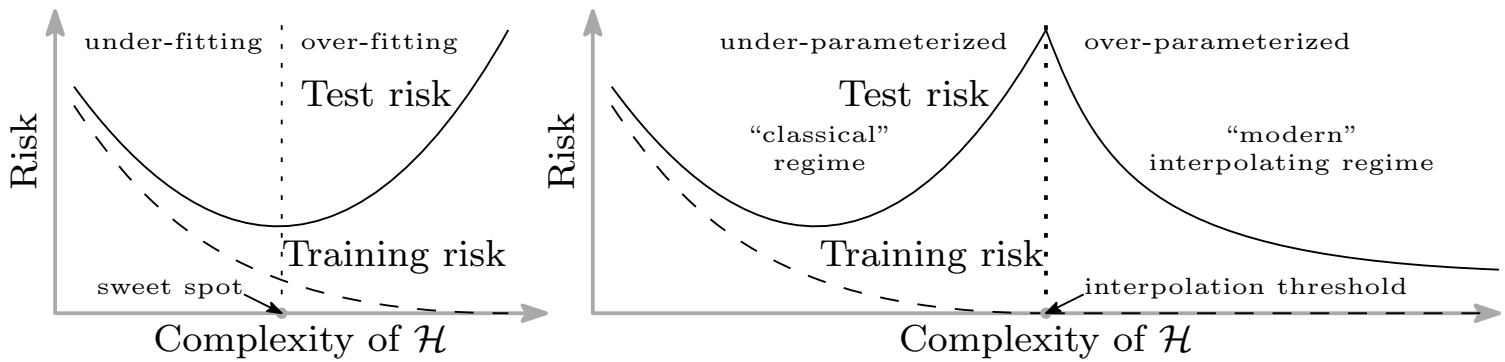
In particular, with g a single hidden layer network

$$\|f - g\|_1 \geq 1 - 2\frac{W^{(1)}}{2^{D^*}}.$$

To approximate f properly, the width $W^{(1)}$ of g's hidden layer has to increase exponentially with f's depth D^*.

This is a simplified variant of results by Telgarsky (2015, 2016).

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



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

(b) "double descent" risk curve

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

(Belkin et al., 2018)

So we have good reasons to increase depth, but we saw that an important issue then is to control the amplitude of the gradient, which is tightly related to controlling activations.

In particular we have to ensure that

- the gradient does not "vanish" (Bengio et al., 1994; Hochreiter et al., 2001),
- gradient amplitude is homogeneous so that all parts of the network train at the same rate (Glorot and Bengio, 2010),
- the gradient does not vary too unpredictably when the weights change (Balduzzi et al., 2017).

Modern techniques change the functional itself instead of trying to improve training "from the outside" through penalty terms or better optimizers.

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

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

Deep learning

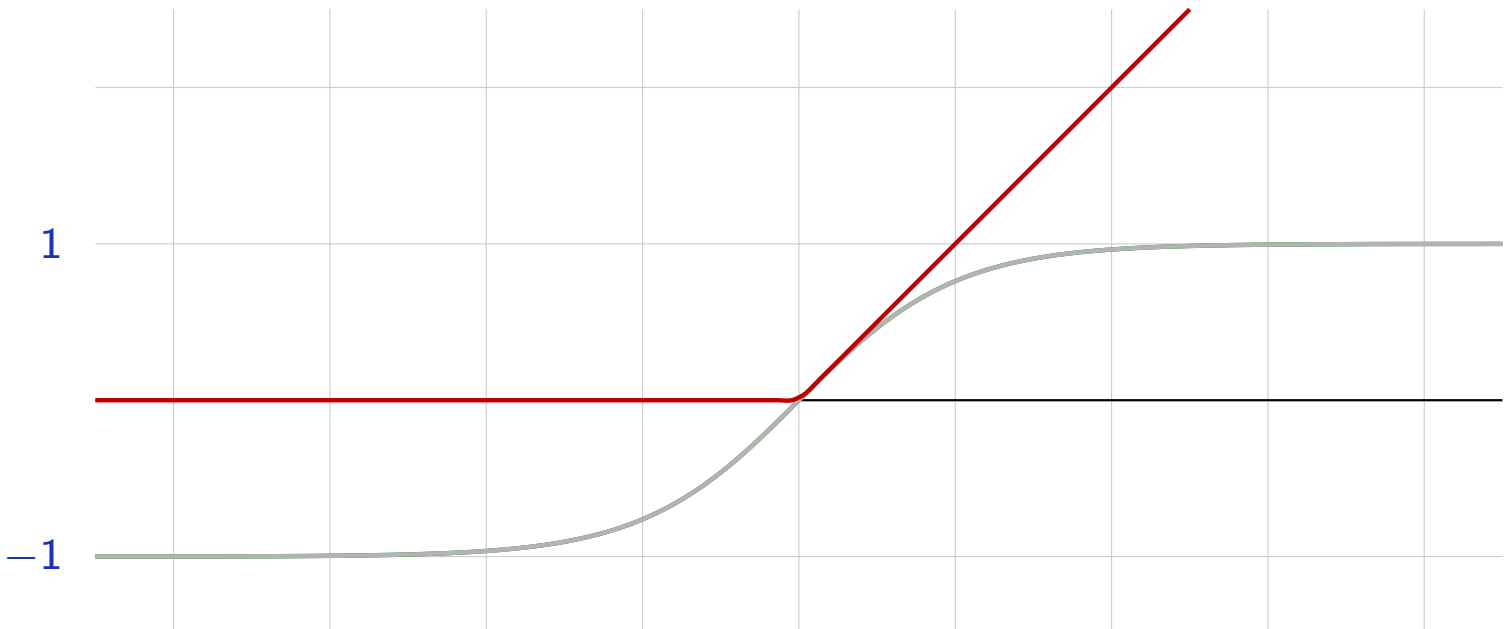
6.2. Rectifiers

François Fleuret

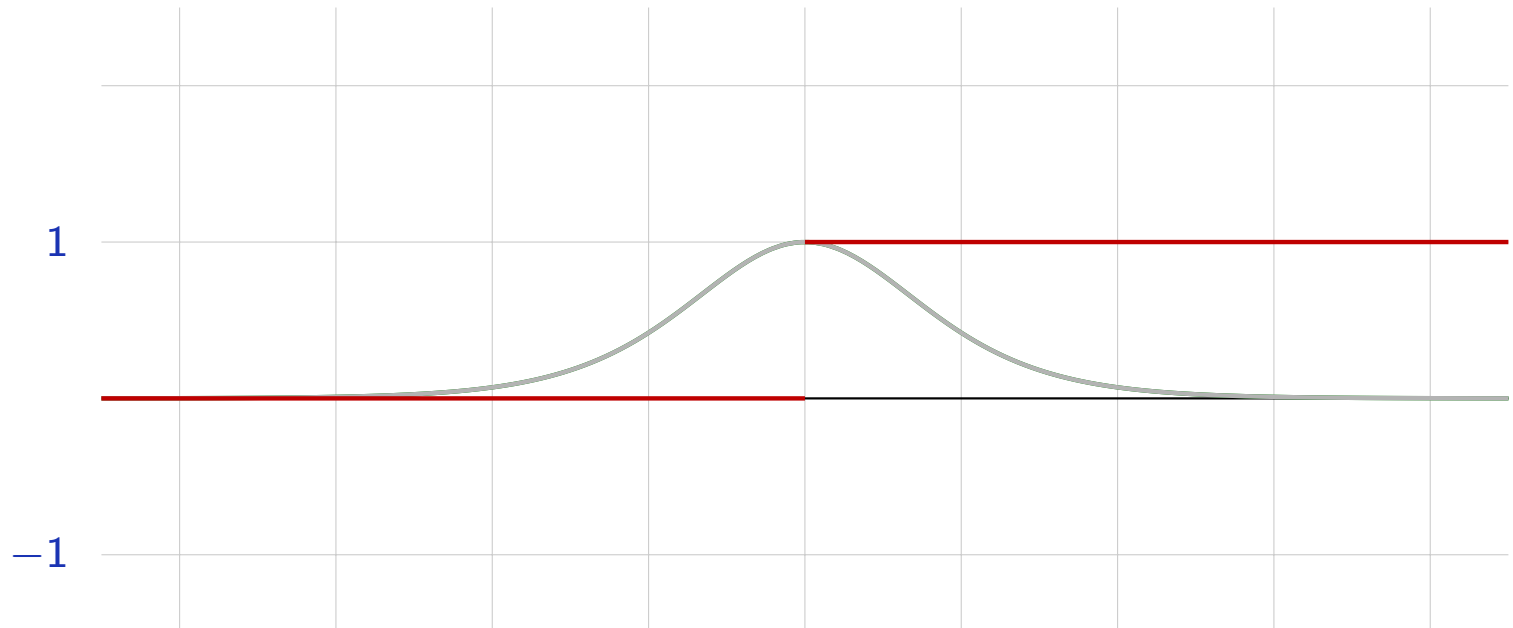
https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

The use of the ReLU activation function was a great improvement compared to the historical tanh (Glorot et al., 2011; Krizhevsky et al., 2012).



This can be explained by the derivative of ReLU itself not vanishing, and by the resulting coding being sparse (Glorot et al., 2011).



The steeper slope in the loss surface speeds up the training.

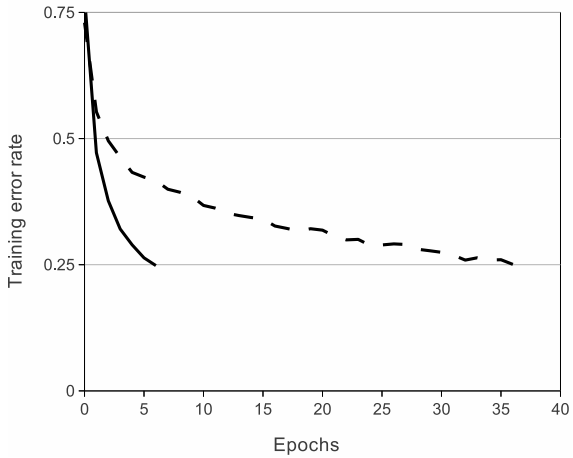


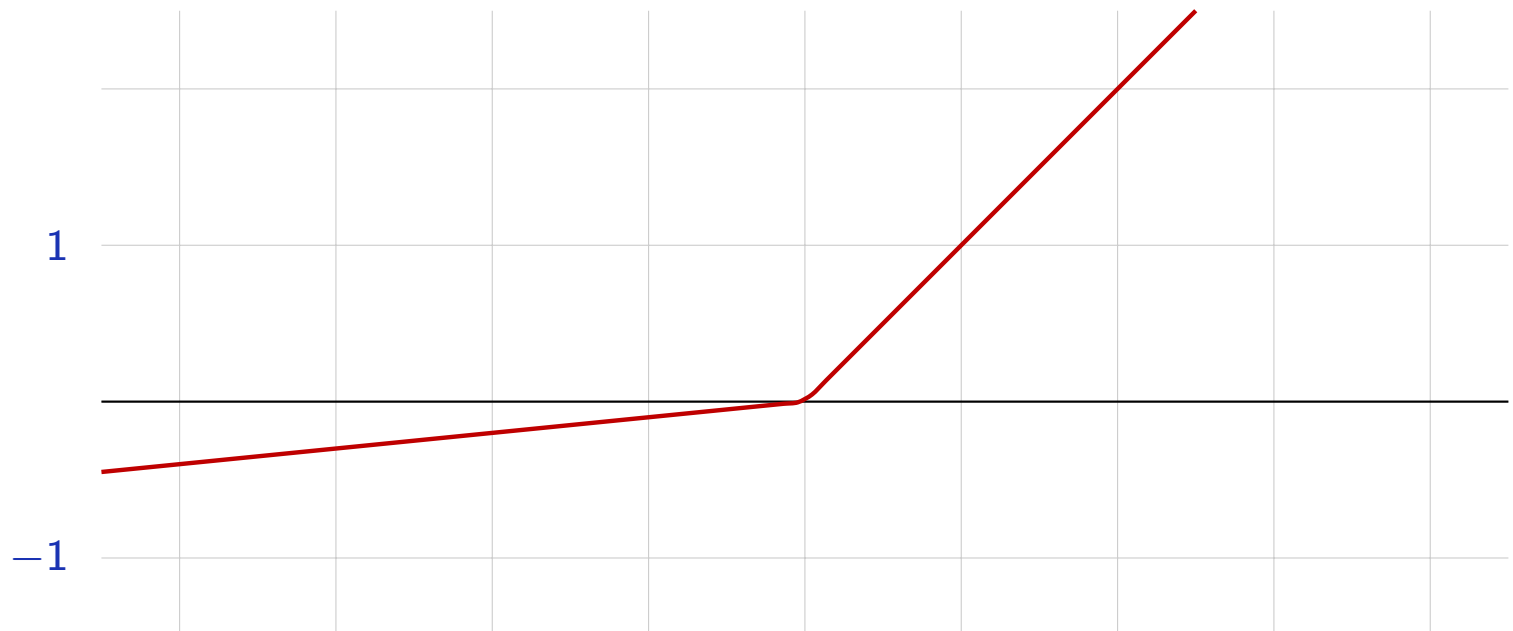
Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with $\tanh$ neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

(Krizhevsky et al., 2012)

A first variant of ReLU is Leaky-ReLU (Maas et al., 2013)

$$\mathbb{R} \to \mathbb{R}$$
$$x \mapsto \max(ax, x)$$

with $0 \leq a < 1$ usually small.



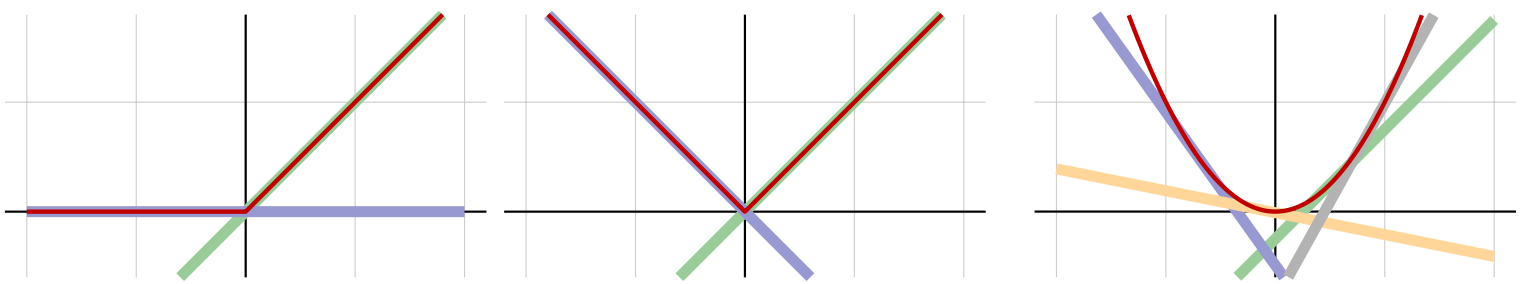
The parameter a can be optimized during training (PReLU, He et al., 2015), or randomized for every sample (RReLU, Xu et al., 2015).

The "maxout" layer proposed by Goodfellow et al. (2013) takes the max of several linear units. This is not an activation function in the usual sense, since it has trainable parameters.

$$h : \mathbb{R}^D \to \mathbb{R}^M$$

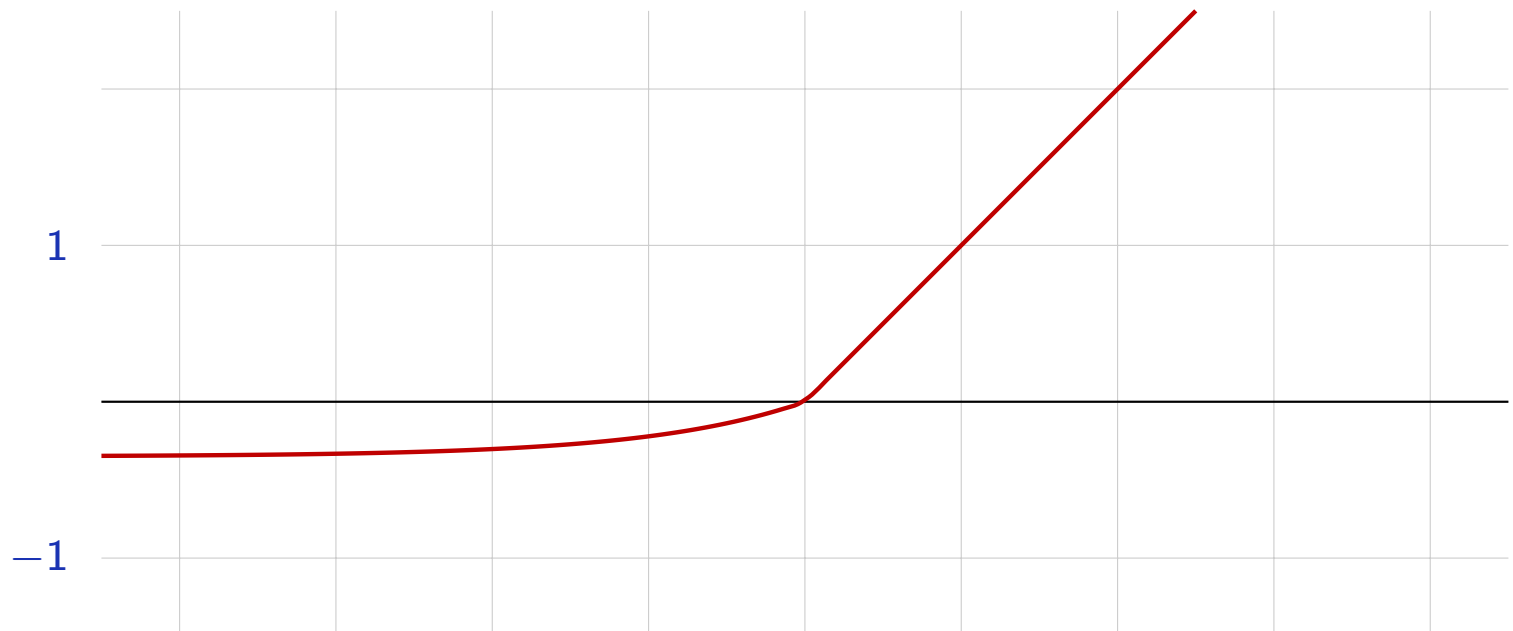
$$x \mapsto \left(\max_{j=1}^{K} x^\top W_{1,j} + b_{1,j}, \ldots, \max_{j=1}^{K} x^\top W_{M,j} + b_{M,j} \right)$$

It can in particular encode ReLU and absolute value, but can also approximate any convex function.



Clevert et al. (2015) proposed the exponential linear unit (ELU), with an exponential saturation

$$x \mapsto \begin{cases} x & \text{if } x \geq 0 \\ \alpha\left(e^x - 1\right) & \text{otherwise.} \end{cases}$$



Another variant is the "Concatenated Rectified Linear Unit" (CReLU) proposed by Shang et al. (2016):

$$\begin{aligned} \mathbb{R} &\to \mathbb{R}^2 \\ x &\mapsto (\max(0, x), \max(0, -x)), \end{aligned}$$

which doubles the number of activations but keeps the norm of the signal intact during both the forward and the backward passes.

Deep learning

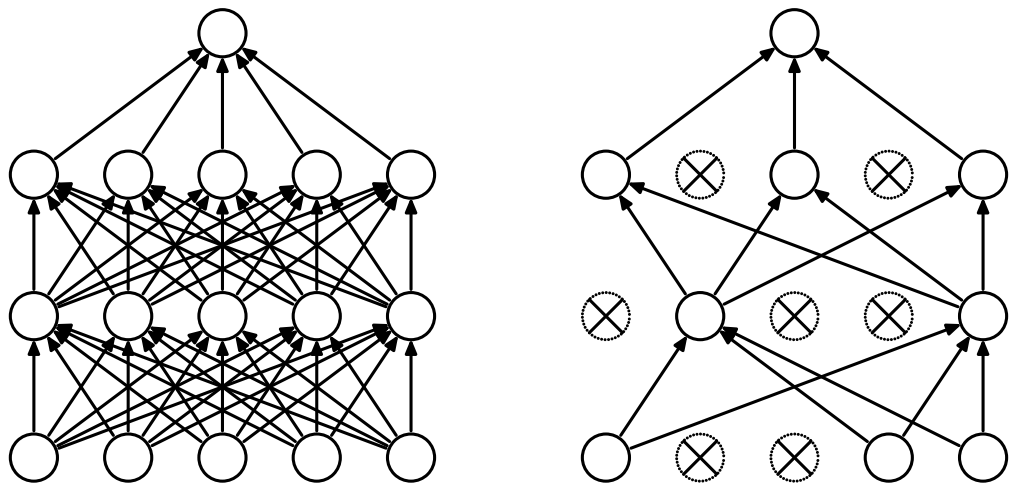
6.3. Dropout

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

A first "deep" regularization technique is **dropout** (Srivastava et al., 2014). It consists of removing units at random during the forward pass on each sample, and putting them all back during test.



(a) Standard Neural Net (b) After applying dropout.

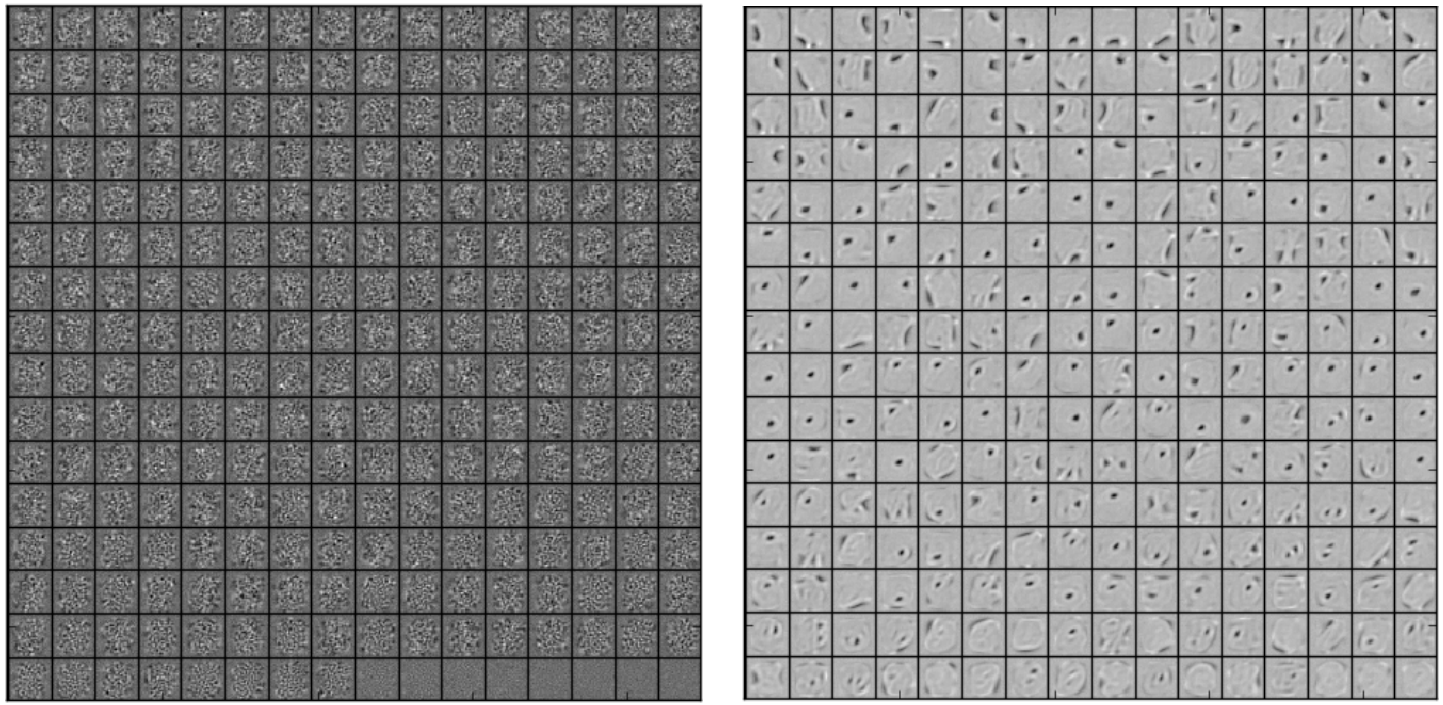
Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

(Srivastava et al., 2014)

This method increases independence between units, and distributes the representation. It generally improves performance.

> "In a standard neural network, the derivative received by each parameter tells it how it should change so the final loss function is reduced, given what all other units are doing. Therefore, units may change in a way that they fix up the mistakes of the other units. This may lead to complex co-adaptations. This in turn leads to overfitting because these co-adaptations do not generalize to unseen data. **We hypothesize that for each hidden unit, dropout prevents co-adaptation by making the presence of other hidden units unreliable.** Therefore, a hidden unit cannot rely on other specific units to correct its mistakes. It must perform well in a wide variety of different contexts provided by the other hidden units."

(Srivastava et al., 2014)



(a) Without dropout

(b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

(Srivastava et al., 2014)

A network with dropout can be interpreted as an ensemble of 2^N models with heavy weight sharing (Goodfellow et al., 2013).

One has to decide on which units/layers to use dropout, and with what probability p units are dropped.

During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.

Let X be a unit activation, and D be an independent Boolean random variable of probability $1 - p$. We have

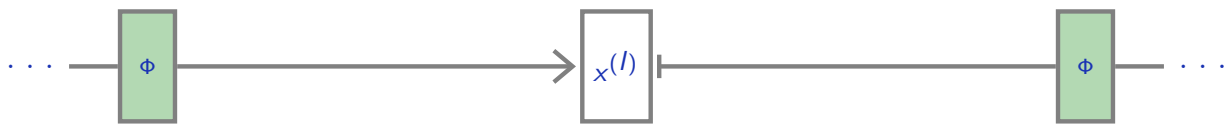
$$\mathbb{E}(D\,X) = \mathbb{E}(D)\,\mathbb{E}(X) = (1-p)\mathbb{E}(X)$$

To keep the means of the inputs to layers unchanged, the initial version of dropout was multiplying activations by $1 - p$ during test.

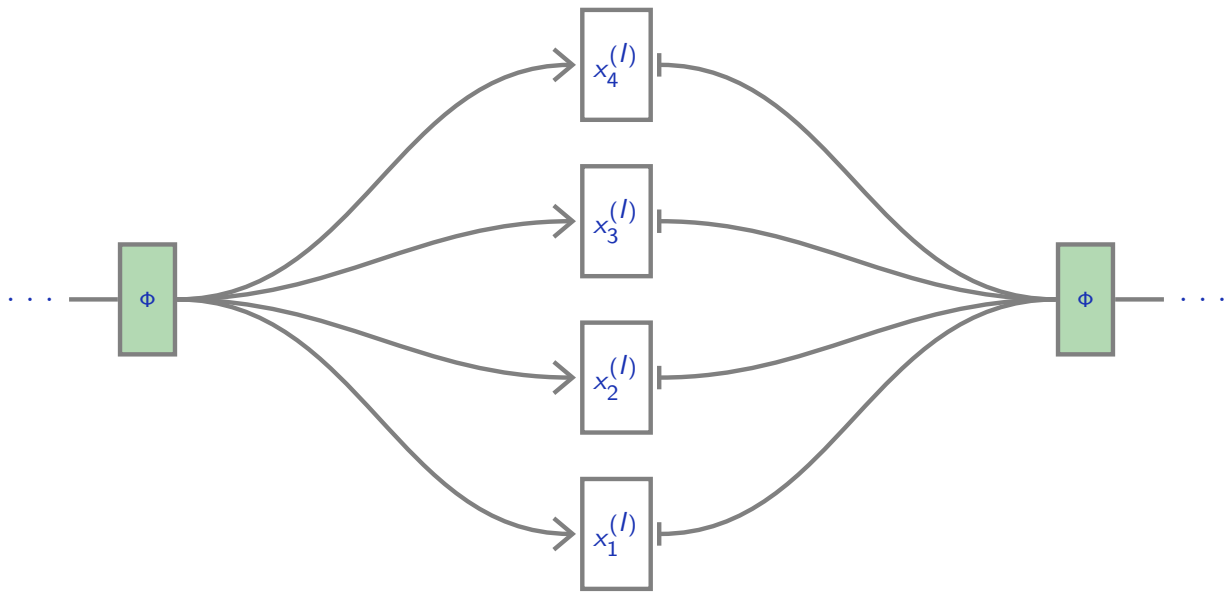
The standard variant in use is the "inverted dropout". It multiplies activations by $\frac{1}{1-p}$ during train and keeps the network untouched during test.

Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.

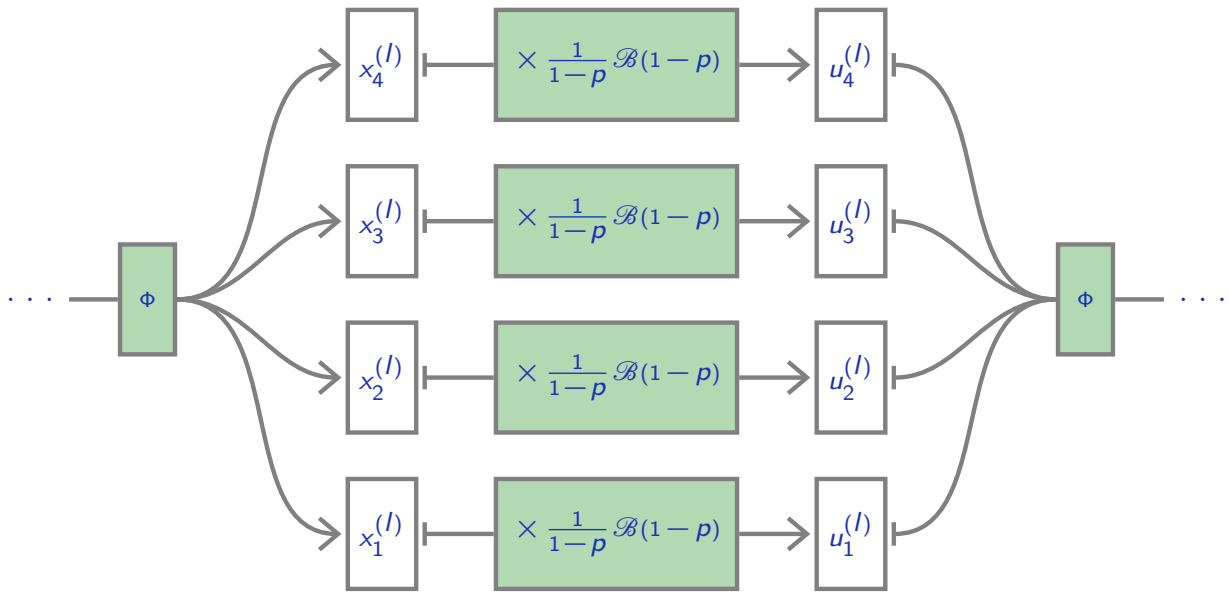
Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



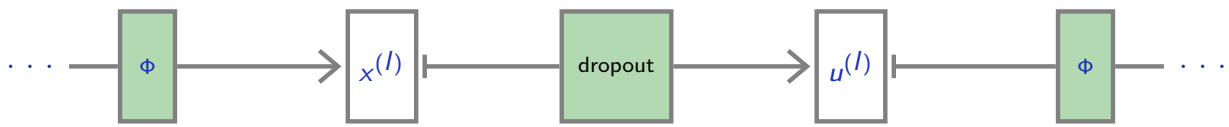
Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



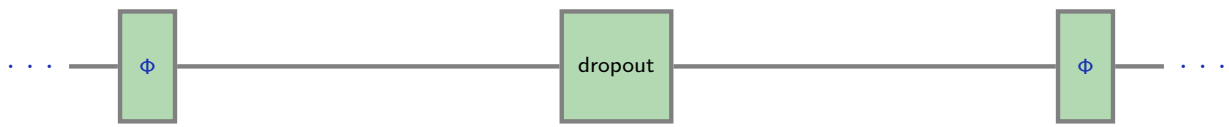
Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



Dropout is implemented in PyTorch as `nn.Dropout`, which is a `torch.Module`.

In the forward pass, it samples a Boolean variable for each component of the tensor it gets as input, and zeroes entries accordingly.

Default probability to drop is $p = 0.5$, but other values can be specified.

```
>>> x = torch.full((3, 5), 1.0).requires_grad_()
>>> x
tensor([[ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.]])
>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y
tensor([[ 0.,  0.,  4.,  0.,  4.],
        [ 0.,  4.,  4.,  4.,  0.],
        [ 0.,  0.,  4.,  0.,  0.]])
>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad
tensor([[ 0.0000,  0.0000,  2.8284,  0.0000,  2.8284],
        [ 0.0000,  2.3094,  2.3094,  2.3094,  0.0000],
        [ 0.0000,  0.0000,  4.0000,  0.0000,  0.0000]])
```

If we have a network

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```

we can simply add dropout layers

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(50, 2));
```

A model using dropout has to be set in "train" or "test" mode.

The method `nn.Module.train(mode)` recursively sets the flag `training` to all sub-modules.

```
>>> dropout = nn.Dropout()
>>> model = nn.Sequential(nn.Linear(3, 10), dropout, nn.Linear(10, 3))
>>> dropout.training
True
>>> model.train(False)
Sequential (
  (0): Linear (3 -> 10)
  (1): Dropout (p = 0.5)
  (2): Linear (10 -> 3)
)
>>> dropout.training
False
```

As pointed out by Tompson et al. (2015), units in a 2d activation map are generally locally correlated, and dropout has virtually no effect. They proposed SpatialDropout, which drops channels instead of individual units.

```
>>> dropout2d = nn.Dropout2d()
>>> x = torch.full((2, 3, 2, 4), 1.)
>>> dropout2d(x)
tensor([[[[ 2.,  2.,  2.,  2.],
          [ 2.,  2.,  2.,  2.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]],

         [[ 2.,  2.,  2.,  2.],
          [ 2.,  2.,  2.,  2.]]],


        [[[ 2.,  2.,  2.,  2.],
          [ 2.,  2.,  2.,  2.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  0.,  0.,  0.]]]])
```

Another variant is dropconnect, which drops connections instead of units.

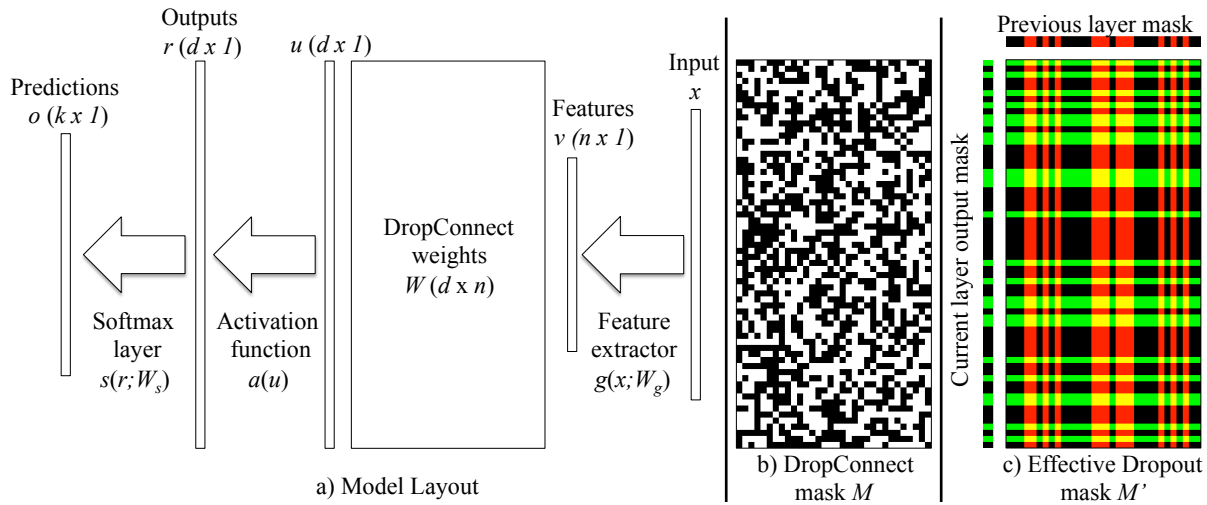


Figure 1. (a): An example model layout for a single DropConnect layer. After running feature extractor $g()$ on input x, a random instantiation of the mask M (e.g. (b)), masks out the weight matrix W. The masked weights are multiplied with this feature vector to produce u which is the input to an activation function a and a softmax layer s. For comparison, (c) shows an effective weight mask for elements that Dropout uses when applied to the previous layer's output (red columns) and this layer's output (green rows). Note the lack of structure in (b) compared to (c).

(Wan et al., 2013)

It cannot be implemented as a separate layer and is computationally intensive.

Deep learning

6.4. Batch normalization

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It was the main motivation behind Xavier's weight initialization rule.

A different approach consists of explicitly forcing the activation statistics during the forward pass by re-normalizing them.

Batch normalization proposed by Ioffe and Szegedy (2015) was the first method introducing this idea.

"Training Deep Neural Networks is complicated by the fact that **the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.** This slows down the training by requiring lower learning rates and careful parameter initialization /.../"

(Ioffe and Szegedy, 2015)

Batch normalization can be done anywhere in a deep architecture, and forces the activations' first and second order moments, so that the following layers do not need to adapt to their drift.

During training batch normalization **shifts and rescales according to the mean and variance estimated on the batch.**

Processing a batch jointly is unusual. Operations used in deep models can virtually always be formalized per-sample.

During test, it simply shifts and rescales according to the empirical moments estimated during training.

If $x_b \in \mathbb{R}^D, b = 1, \ldots, B$ are the samples in the batch, we first compute the empirical per-component mean and variance **on the batch**

$$\hat{m}_{batch} = \frac{1}{B} \sum_{b=1}^{B} x_b$$

$$\hat{v}_{batch} = \frac{1}{B} \sum_{b=1}^{B} (x_b - \hat{m}_{batch})^2$$

from which we compute normalized $z_b \in \mathbb{R}^D$, and outputs $y_b \in \mathbb{R}^D$

$$\forall b = 1, \ldots, B, \ z_b = \frac{x_b - \hat{m}_{batch}}{\sqrt{\hat{v}_{batch} + \epsilon}}$$

$$y_b = \gamma \odot z_b + \beta.$$

where $\odot$ is the Hadamard component-wise product, and $\gamma \in \mathbb{R}^D$ and $\beta \in \mathbb{R}^D$ are parameters to optimize.

During inference, batch normalization shifts and rescales independently each component of the input x according to statistics estimated during training:

$$y = \gamma \odot \frac{x - \hat{m}}{\sqrt{\hat{v} + \epsilon}} + \beta.$$

Hence, during inference, batch normalization performs **a component-wise affine transformation**, and it can process samples individually.

As for dropout, the model behaves differently during train and test.

As dropout, batch normalization is implemented as separate modules that process input components independently.

```
>>> bn = nn.BatchNorm1d(3)
>>> with torch.no_grad():
...     bn.bias.copy_(torch.tensor([2., 4., 8.]))
...     bn.weight.copy_(torch.tensor([1., 2., 3.]))
...
Parameter containing:
tensor([2., 4., 8.], requires_grad=True)
Parameter containing:
tensor([1., 2., 3.], requires_grad=True)
>>> x = torch.randn(1000, 3)
>>> x = x * torch.tensor([2., 5., 10.]) + torch.tensor([-10., 25., 3.])
>>> x.mean(0)
tensor([-9.9669, 25.0213,  2.4361])
>>> x.std(0)
tensor([1.9063, 5.0764, 9.7474])
>>> y = bn(x)
>>> y.mean(0)
tensor([2.0000, 4.0000, 8.0000], grad_fn=<MeanBackward2>)
>>> y.std(0)
tensor([1.0005, 2.0010, 3.0015], grad_fn=<StdBackward1>)
```

As for any other module, we have to compute the derivatives of the loss $\mathscr{L}$ with respect to the inputs values and the parameters.

For clarity, since components are processed independently, in what follows we consider a single dimension and do not index it.

We have

$$\hat{m}_{batch} = \frac{1}{B}\sum_{b=1}^{B} x_b$$

$$\hat{v}_{batch} = \frac{1}{B}\sum_{b=1}^{B} (x_b - \hat{m}_{batch})^2$$

$$\forall b = 1, \ldots, B, \; z_b = \frac{x_b - \hat{m}_{batch}}{\sqrt{\hat{v}_{batch} + \epsilon}}$$

$$y_b = \gamma z_b + \beta.$$

From which

$$\forall b = 1, \ldots, B, \; \frac{\partial \mathscr{L}}{\partial z_b} = \gamma \frac{\partial \mathscr{L}}{\partial y_b}$$

$$\frac{\partial \mathscr{L}}{\partial \gamma} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b}\frac{\partial y_b}{\partial \gamma} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b} z_b$$

$$\frac{\partial \mathscr{L}}{\partial \beta} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b}\frac{\partial y_b}{\partial \beta} = \sum_b \frac{\partial \mathscr{L}}{\partial y_b}.$$

Every sample in the batch impacts the moment estimates, hence all the outputs, which makes the derivative with respect to an input complicated.

$$\frac{\partial \mathscr{L}}{\partial \hat{v}_{batch}} = -\frac{1}{2}\left(\hat{v}_{batch} + \epsilon\right)^{-3/2} \sum_{b=1}^{B} \frac{\partial \mathscr{L}}{\partial z_b}(x_b - \hat{m}_{batch})$$

$$\frac{\partial \mathscr{L}}{\partial \hat{m}_{batch}} = -\frac{1}{\sqrt{\hat{v}_{batch} + \epsilon}} \sum_{b=1}^{B} \frac{\partial \mathscr{L}}{\partial z_b}$$

$$\forall b = 1, \ldots, B, \quad \frac{\partial \mathscr{L}}{\partial x_b} = \frac{\partial \mathscr{L}}{\partial z_b}\frac{1}{\sqrt{\hat{v}_{batch} + \epsilon}} + \frac{2}{B}\frac{\partial \mathscr{L}}{\partial \hat{v}_{batch}}(x_b - \hat{m}_{batch}) + \frac{1}{B}\frac{\partial \mathscr{L}}{\partial \hat{m}_{batch}}$$

In standard implementations, test $\hat{m}$ and $\hat{v}$ are estimated with a moving average during train, to avoid the need for an additional pass through the samples.

Results on ImageNet's LSVRC2012:

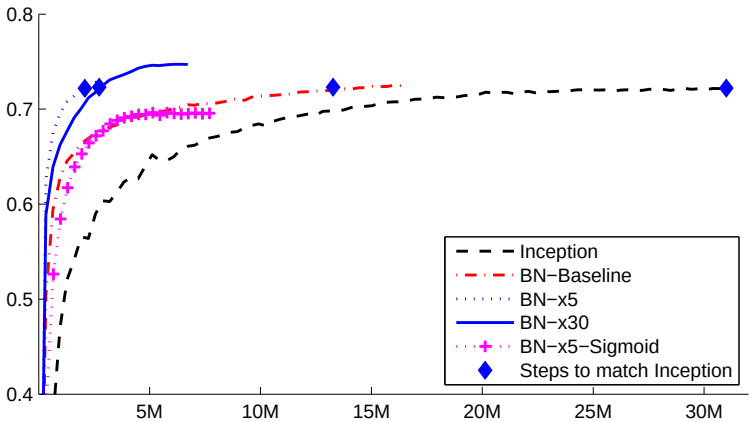


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

(Ioffe and Szegedy, 2015)

The authors state that with batch normalization

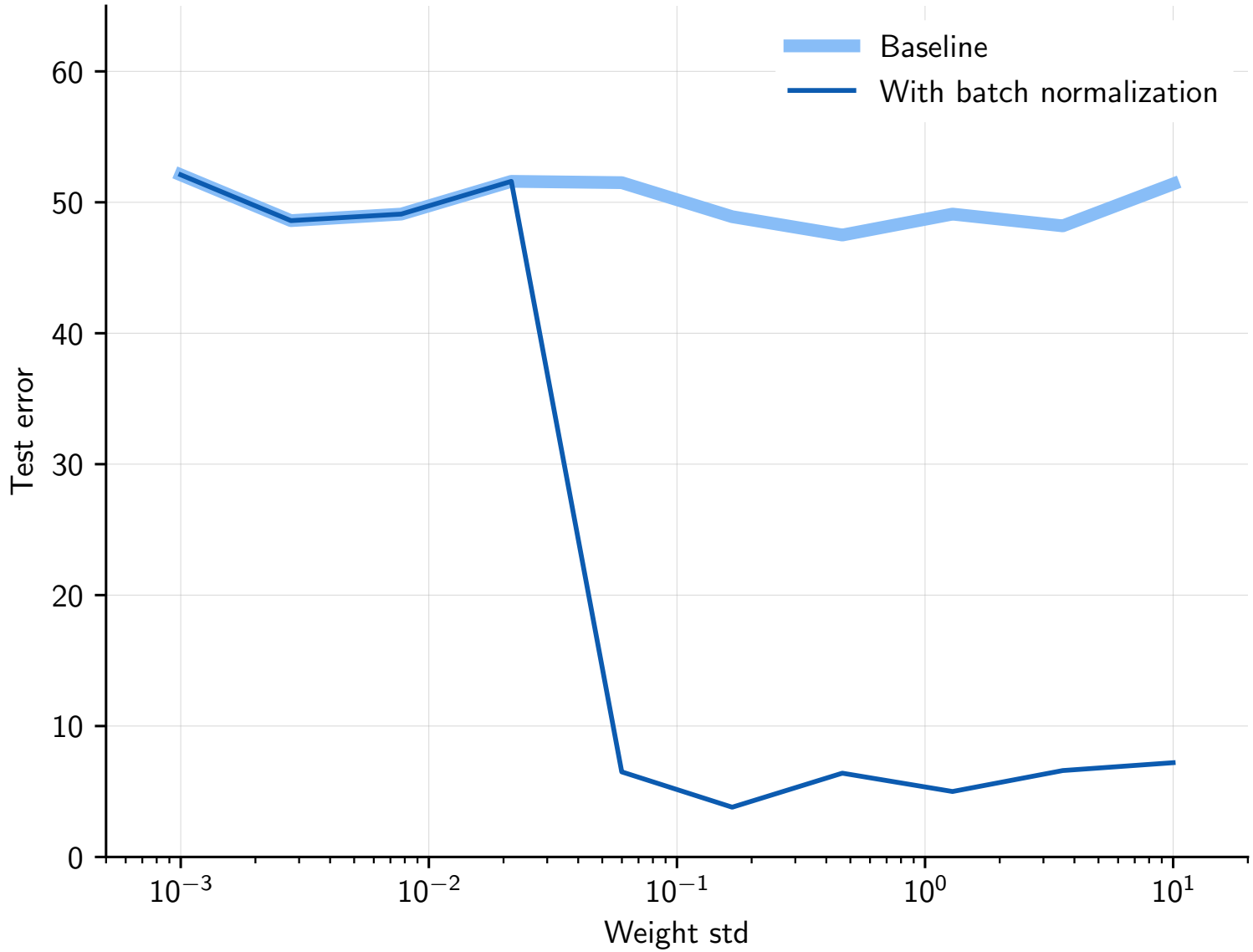
- samples have to be shuffled carefully,
- the learning rate can be greater,
- dropout and local normalization are not necessary,
- L^2 regularization influence should be reduced.

Deep MLP on a 2d "disc" toy example, with naive Gaussian weight initialization, cross-entropy, standard SGD, $\eta = 0.1$.

```
def create_model(with_batchnorm, dimh = 32, nb_layers = 16):
    modules = []

    modules.append(nn.Linear(2, dimh))
    if with_batchnorm: modules.append(nn.BatchNorm1d(dimh))
    modules.append(nn.ReLU())

    for d in range(nb_layers):
        modules.append(nn.Linear(dimh, dimh))
        if with_batchnorm: modules.append(nn.BatchNorm1d(dimh))
        modules.append(nn.ReLU())

    modules.append(nn.Linear(dimh, 2))

    return nn.Sequential(*modules)
```

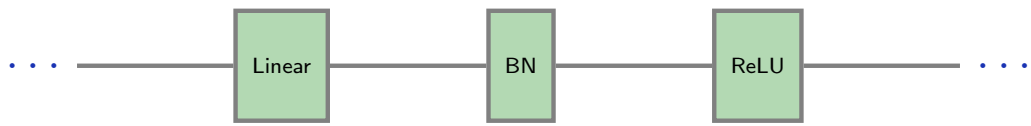
We try different standard deviations for the weights

```
with torch.no_grad():
    for p in model.parameters(): p.normal_(0, std)
```

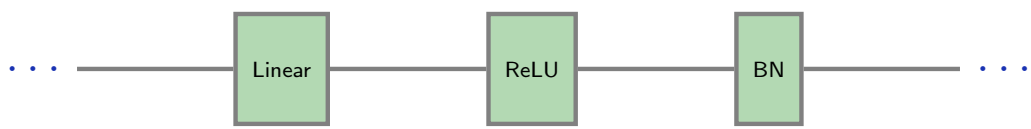

Baseline
With batch normalization
Test error
60
50
40
30
20
10
0
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Weight std

The position of batch normalization relative to the non-linearity is not clear.

> "We add the BN transform immediately before the nonlinearity, by normalizing $x = Wu + b$. We could have also normalized the layer inputs u, but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, $Wu + b$ is more likely to have a symmetric, non-sparse distribution, that is 'more Gaussian' (Hyvärinen and Oja, 2000); normalizing it is likely to produce activations with a stable distribution. "
>
> (Ioffe and Szegedy, 2015)



However, this argument goes both ways: activations after the non-linearity are less "naturally normalized" and benefit more from batch normalization.
Experiments are generally in favor of this solution, which is the current default.



As for dropout, using properly batch normalization on a convolutional map requires parameter-sharing.

The module `torch.BatchNorm2d` (respectively `torch.BatchNorm3d`) processes samples as multi-channels 2d maps (respectively multi-channels 3d maps) and normalizes each channel separately, with a γ and a β for each.

Another normalization in the same spirit is the **layer normalization** proposed by Ba et al. (2016).

Given a single sample $x \in \mathbb{R}^D$, it normalizes the components of x, hence normalizing activations across the layer instead of doing it across the batch

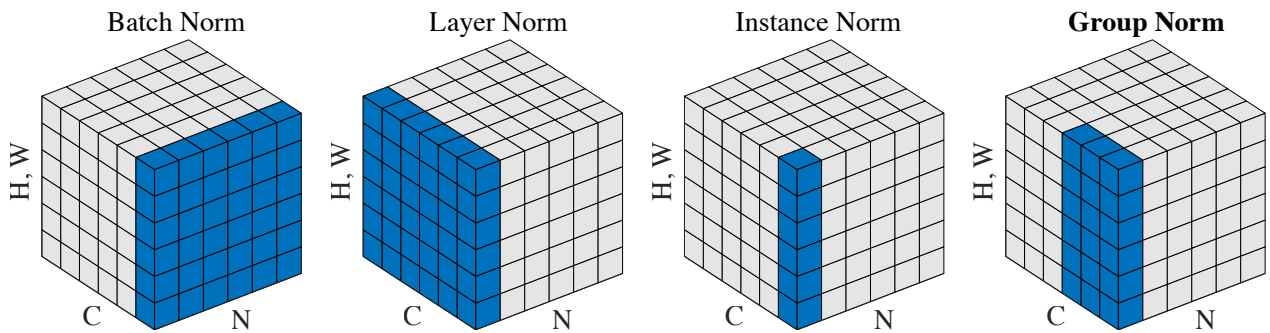
$$\mu = \frac{1}{D} \sum_{d=1}^{D} x_d$$

$$\sigma = \sqrt{\frac{1}{D} \sum_{d=1}^{D} (x_d - \mu)^2}$$

$$\forall d,\ y_d = \frac{x_d - \mu}{\sigma}$$

Although it gives slightly worst improvements than BN it has the advantage of behaving similarly in train and test, and processing samples individually.

These normalization schemes are examples of a larger class of methods.



(Wu and He, 2018)

Deep learning

6.5. Residual networks

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

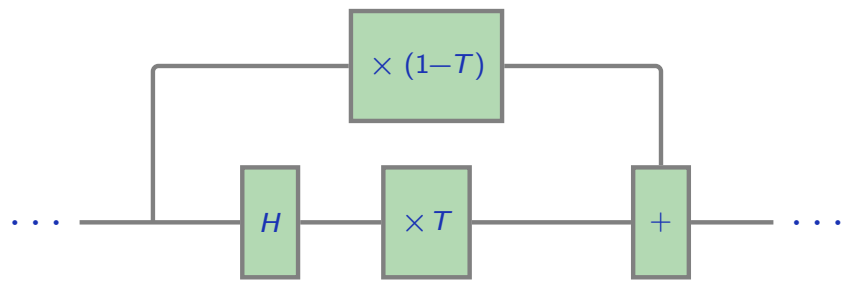
The "Highway networks" by Srivastava et al. (2015) use the idea of gating developed for recurrent units. It replaces a standard non-linear layer

$$y = H(x; W_H)$$

with a layer that includes a "gated" pass-through

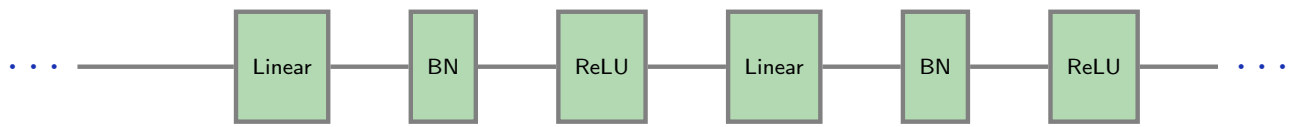
$$y = T(x; W_T)H(x; W_H) + (1 - T(x; W_T))x$$

where $T(x; W_T) \in [0, 1]$ modulates how much the signal should be transformed.

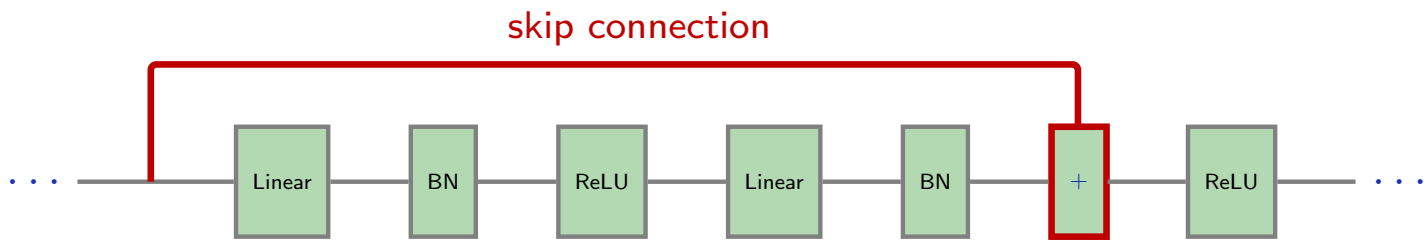


Initializing T's parameters so that $T \simeq 0$ at first, assures that gradients will pass through, and allows to train networks with up to 100 layers.

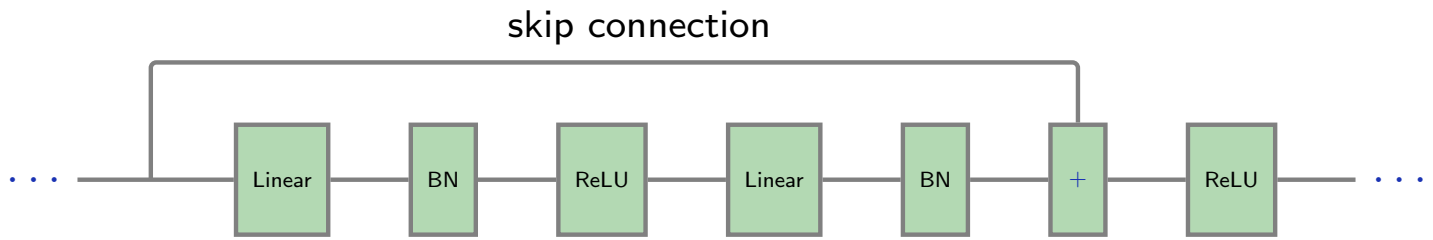
The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a **skip connection**.



The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a **skip connection**.



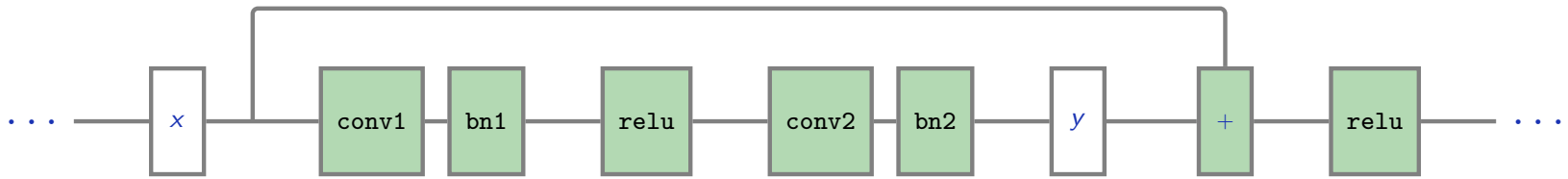
The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a **skip connection**.



Thanks to this structure, the parameters are optimized to learn a **residual**, that is the difference between the value before the block and the one needed after.

We can implement such a network for MNIST, composed of:

- A first convolution layer `conv0` with kernels 1×1 to convert the tensor from $1 \times 28 \times 28$ to `nb_channels` $\times 28 \times 28$,
- a series of `ResBlock`s, each composed of two convolution layers and two batch normalization layers, that maintains the tensor size unchanged,
- an average poling layer `avg` that produces an output of size `nb_channels` $\times 1 \times 1$,
- a fully connected layer `fc` to make the final prediction.




```
class ResBlock(nn.Module):
    def __init__(self, nb_channels, kernel_size):
        super().__init__()

        self.conv1 = nn.Conv2d(nb_channels, nb_channels, kernel_size,
                               padding = (kernel_size-1)//2)
        self.bn1 = nn.BatchNorm2d(nb_channels)

        self.conv2 = nn.Conv2d(nb_channels, nb_channels, kernel_size,
                               padding = (kernel_size-1)//2)
        self.bn2 = nn.BatchNorm2d(nb_channels)

    def forward(self, x):
        y = self.bn1(self.conv1(x))
        y = F.relu(y)
        y = self.bn2(self.conv2(y))
        y += x
        y = F.relu(y)
        return y
```

```
class ResNet(nn.Module):
    def __init__(self, nb_channels, kernel_size, nb_blocks):
        super().__init__()

        self.conv0 = nn.Conv2d(1, nb_channels, kernel_size = 1)

        self.resblocks = nn.Sequential(
            # A bit of fancy Python
            *(ResBlock(nb_channels, kernel_size) for _ in range(nb_blocks))
        )

        self.avg = nn.AvgPool2d(kernel_size = 28)
        self.fc = nn.Linear(nb_channels, 10)

    def forward(self, x):
        x = F.relu(self.conv0(x))
        x = self.resblocks(x)
        x = F.relu(self.avg(x))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

With 25 residual blocks, 16 channels, and convolution kernels of size 3×3, we get the following structure, with $117,802$ parameters.

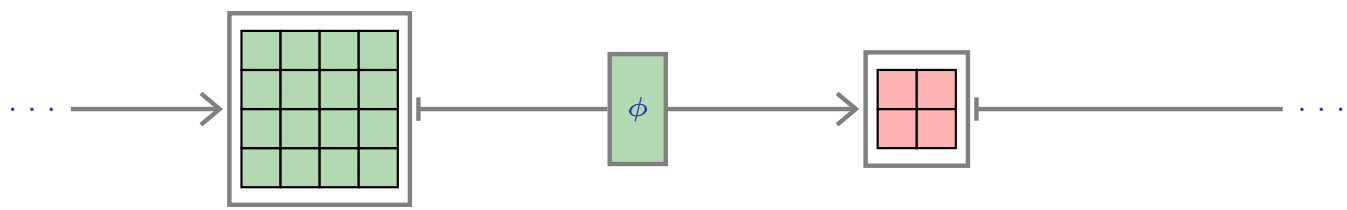
```
ResNet(
  (conv0): Conv2d(1, 16, kernel_size=(1, 1), stride=(1, 1))
  (resblocks): Sequential(
    (0): ResBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    /.../
    (24): ResBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avg): AvgPool2d(kernel_size=28, stride=28, padding=0)
  (fc): Linear(in_features=16, out_features=10, bias=True)
)
```

A technical point for a more general use of a residual architecture is to deal with convolution layers that change the activation map sizes or numbers of channels.

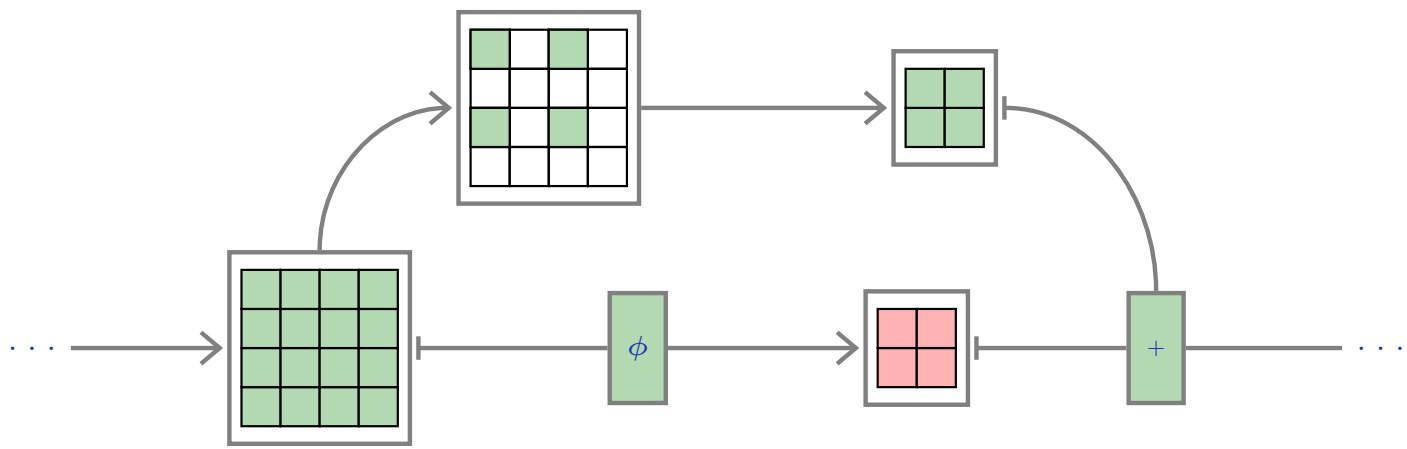
He et al. (2015) only consider:

- reducing the activation map size by a factor 2,
- increasing the number of channels.

To reduce the activation map size by a factor 2, the identity pass-trough extracts 1/4 of the activations over a regular grid (i.e. with a stride of 2),



To reduce the activation map size by a factor 2, the identity pass-trough extracts 1/4 of the activations over a regular grid (i.e. with a stride of 2),



To increase the number of channels from C to C', they propose to either:

- pad the original value with $C' - C$ zeros, which amounts to adding as many zeroed channels, or
- use C' convolutions with a $1 \times 1 \times C$ filter, which corresponds to applying the same fully-connected linear model $\mathbb{R}^C \to \mathbb{R}^{C'}$ at every location.

Finally, He et al.'s residual networks are fully convolutional, which means they have no fully connected layers. We will come back to this.

Their one-before last layer is a per-channel global average pooling that outputs a $1d$ tensor, fed into a single fully-connected layer.

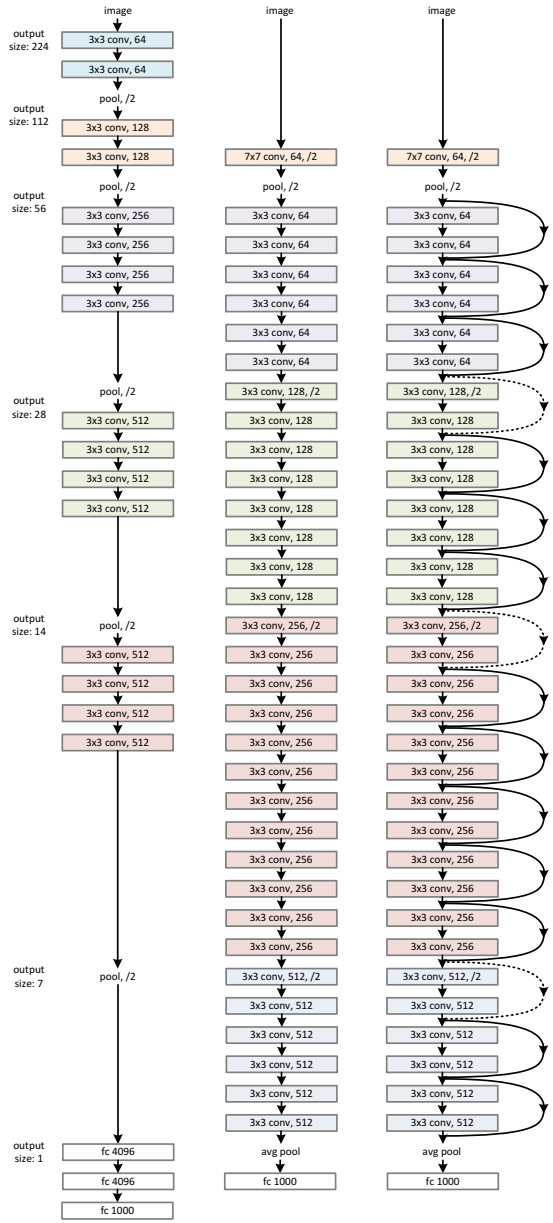


image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
fc 1000
output size: 1
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000

(He et al., 2015)

Performance on ImageNet.

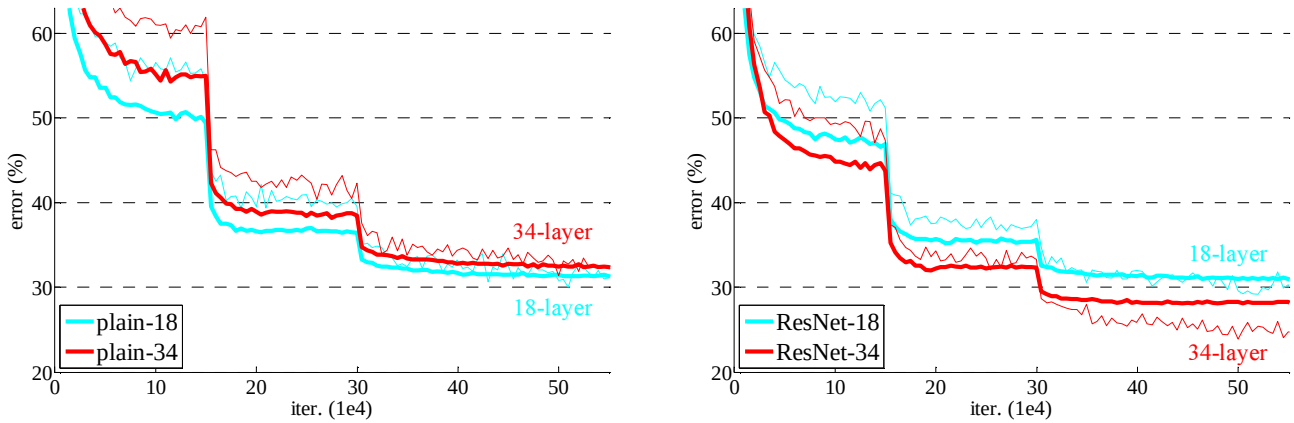
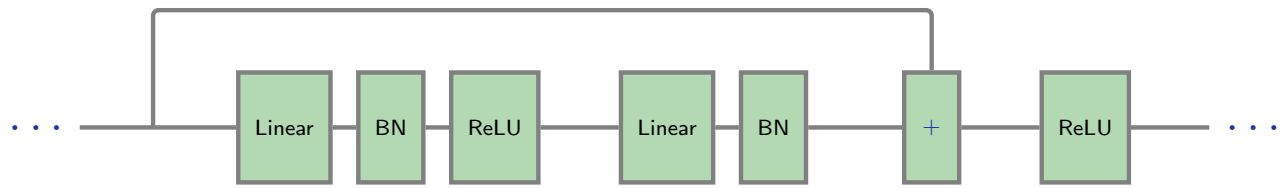


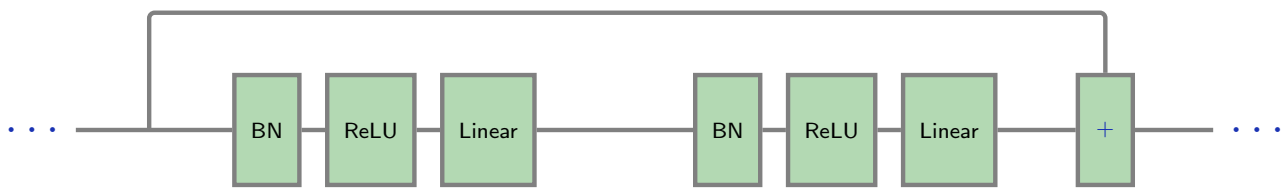
Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

(He et al., 2015)

He et al. (2016) proposed to sequence operations in a residual block so that the main "pathway" has no non-linearity. This results in substantial improvements.



Original (He et al., 2015)



Identity residual (He et al., 2016)

Veit et al. (2016) interpret a residual network as an ensemble, which explains in part its stability.

E.g., with three blocks we have

$$\begin{aligned} x_1 &= x_0 + f_1(x_0) \\ x_2 &= x_1 + f_2(x_1) \\ x_3 &= x_2 + f_3(x_2) \end{aligned}$$

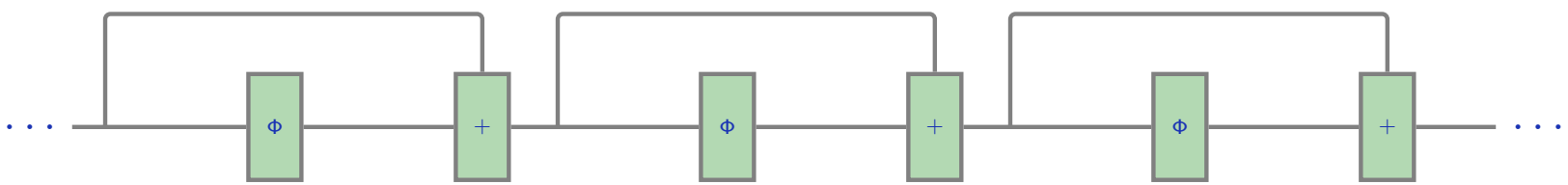
hence there are four "paths":

$$\begin{aligned} x_3 &= x_2 + f_3(x_2) \\ &= x_1 + f_2(x_1) + f_3(x_1 + f_2(x_1)) \\ &= \underbrace{x_0} + \underbrace{f_1(x_0)} + \underbrace{f_2(x_0 + f_1(x_0))} + \underbrace{f_3(x_0 + f_1(x_0) + f_2(x_0 + f_1(x_0)))}. \end{aligned}$$

Veit et al. show that (1) performance reduction correlates with the number of paths removed from the ensemble, not with the number of blocks removed, (2) only gradients through shallow paths matter during train.

An extension of the residual network, is the **stochastic depth** network.

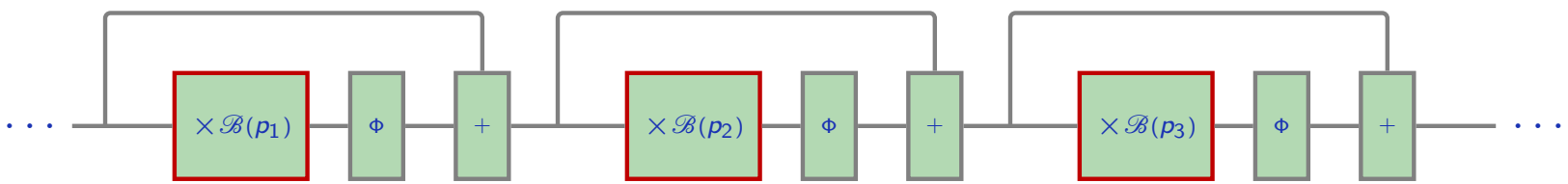
> "Stochastic depth aims to shrink the depth of a network during training, while keeping it unchanged during testing. We can achieve this goal by randomly dropping entire ResBlocks during training and bypassing their transformations through skip connections."
>
> (Huang et al., 2016)



An extension of the residual network, is the **stochastic depth** network.

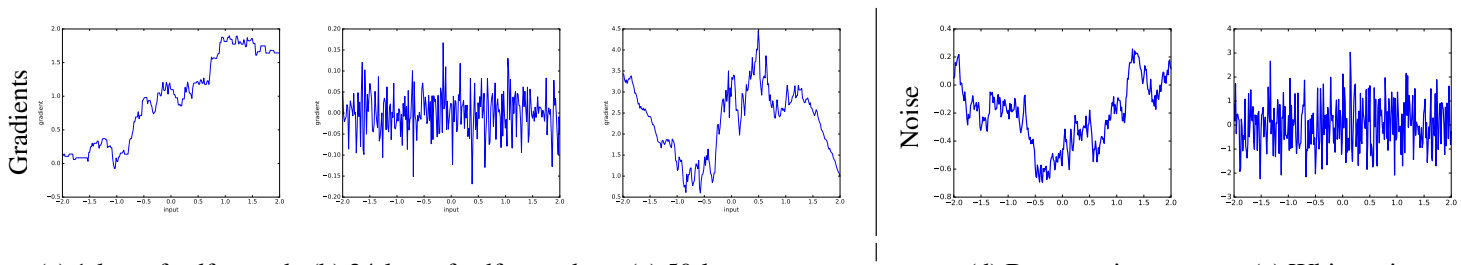
"Stochastic depth aims to shrink the depth of a network during training, while keeping it unchanged during testing. We can achieve this goal by randomly dropping entire ResBlocks during training and bypassing their transformations through skip connections."

(Huang et al., 2016)



Shattered Gradient

Balduzzi et al. (2017) points out that depth "shatters" the relation between the input and the gradient w.r.t. the input, and that Resnets mitigate this effect.



(a) 1-layer feedforward. (b) 24-layer feedforward. (c) 50-layer resnet. (d) Brown noise. (e) White noise.

(Balduzzi et al., 2017)

Since linear networks avoid this problem, they suggest to combine CReLU (see lecture 6.2. "Rectifiers") with a **Looks Linear initialization** that makes the network linear initially.

Let $\sigma(x) = \max(0, x)$, and

$$\Phi : \mathbb{R}^D \to \mathbb{R}^{2D}$$

the CReLU non-linearity, i.e.

$$\forall x \in \mathbb{R}^D,\ q = 1, \ldots, D, \begin{cases} \Phi(x)_{2q-1} & = \sigma(x_q), \\ \Phi(x)_{2q} & = \sigma(-x_q) \end{cases}$$

and a weight matrix $\tilde{W} \in \mathbb{R}^{D' \times 2D}$ such that

$$\forall j = 1, \ldots, D', q = 1, \ldots, D,\ \tilde{W}_{j,2q-1} = -\tilde{W}_{j,2q} = W_{j,q}.$$

So two neighboring columns of $\Phi(x)$ are the $\sigma(\cdot)$ and $\sigma(-\cdot)$ of a column of x, and two neighboring columns of $\tilde{W}$ are a column of W and its opposite.

From this we get, $\forall i = 1, \ldots, B,\ j = 1, \ldots, D'$:

$$\begin{aligned}
\left(\tilde{W}\Phi(x)\right)_j &= \sum_{k=1}^{2D} \tilde{W}_{j,k}\Phi(x)_k \\
&= \sum_{q=1}^{D} \tilde{W}_{j,2q-1}\Phi(x)_{2q-1} + \tilde{W}_{j,2q}\Phi(x)_{2q} \\
&= \sum_{q=1}^{D} W_{j,q}\sigma(x_q) - W_{j,q}\sigma(-x_q) \\
&= \sum_{q=1}^{D} W_{j,q}x_q \\
&= (Wx)_j .
\end{aligned}$$

Hence

$$\forall x,\ \tilde{W}\Phi(x) = Wx$$

and doing this in every layer results in a linear network.

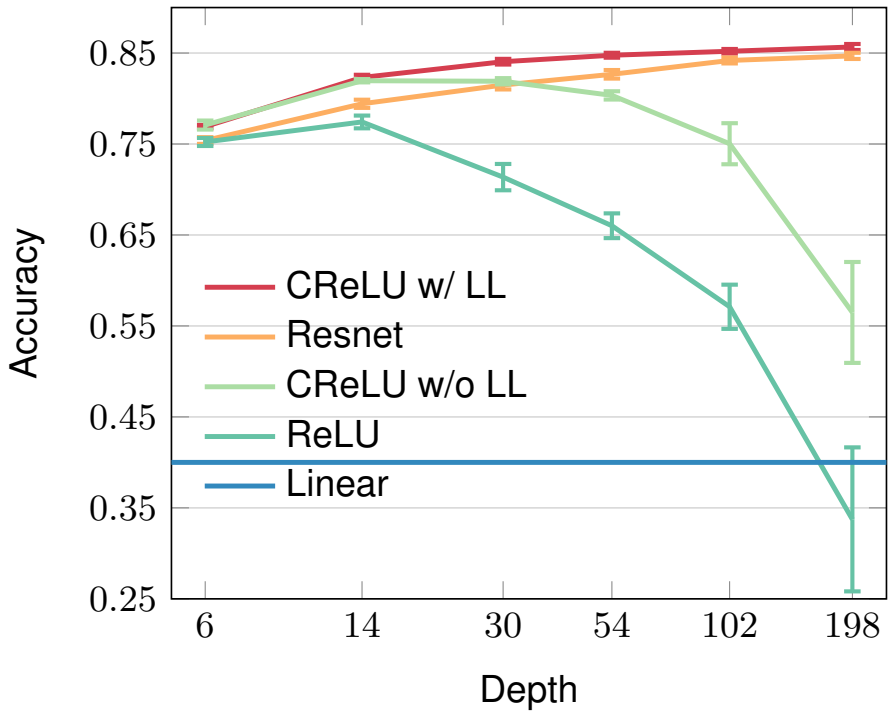


Figure 6: **CIFAR-10 test accuracy**. Comparison of test accuracy between networks of different depths with and without LL initialization.

(Balduzzi et al., 2017)

We can summarize the techniques which have enabled the training of very deep architectures:

- rectifiers to prevent the gradient from vanishing during the backward pass,
- dropout to force a distributed representation,
- batch normalization to dynamically maintain the statistics of activations,
- identity pass-through to keep a structured gradient and distribute representation,
- smart initialization to put the gradient in a good regime.

Deep learning

6.6. Using GPUs

François Fleuret

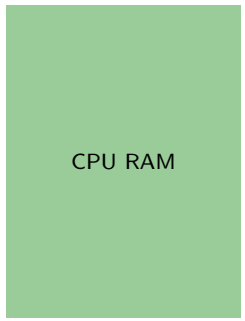
https://fleuret.org/dlc/

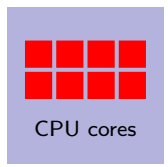
UNIVERSITÉ
DE GENÈVE

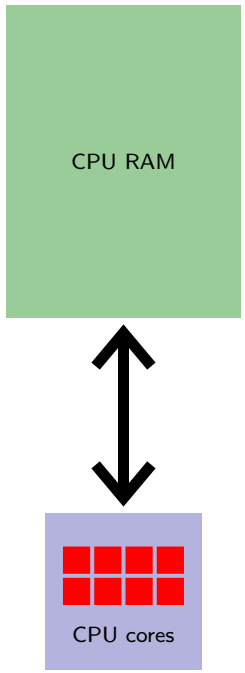
The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.

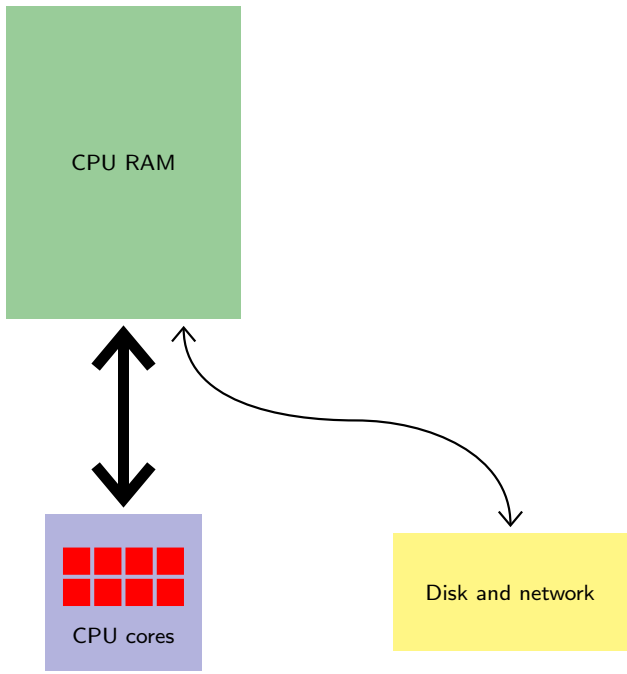
Although they were historically developed for mass-market real-time CGI, the highly parallel architecture of GPUs is extremely fitting to signal processing and high dimension linear algebra.

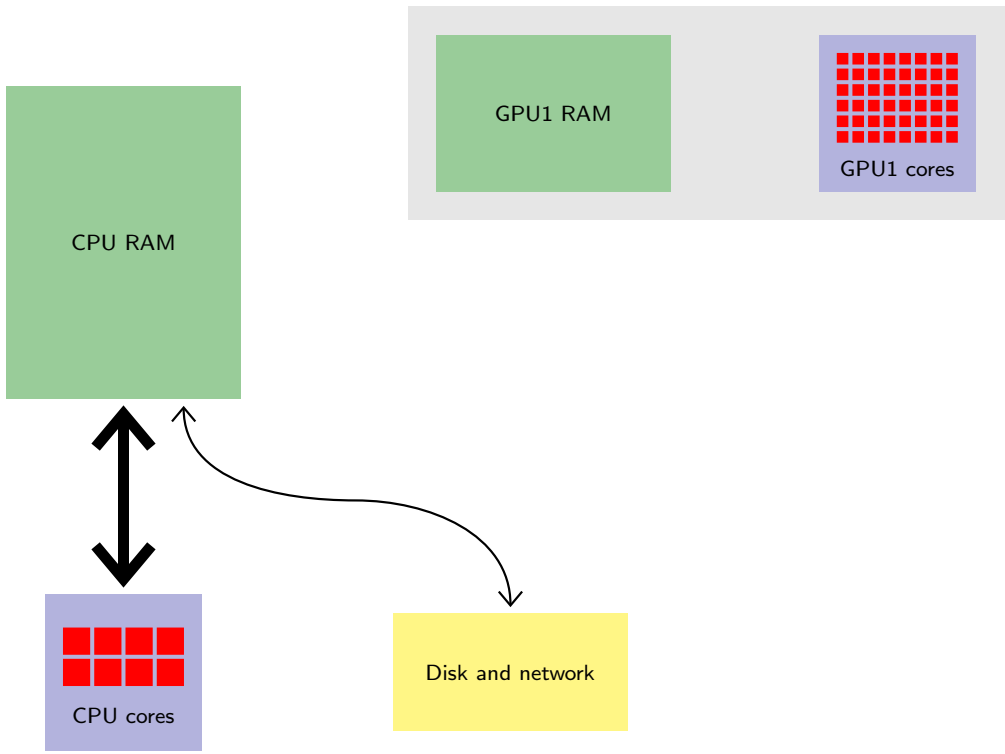
Their use is instrumental in the success of deep-learning (Raina et al., 2009; Ciresan et al., 2010; Krizhevsky et al., 2012; Shi et al., 2016).


CPU RAM

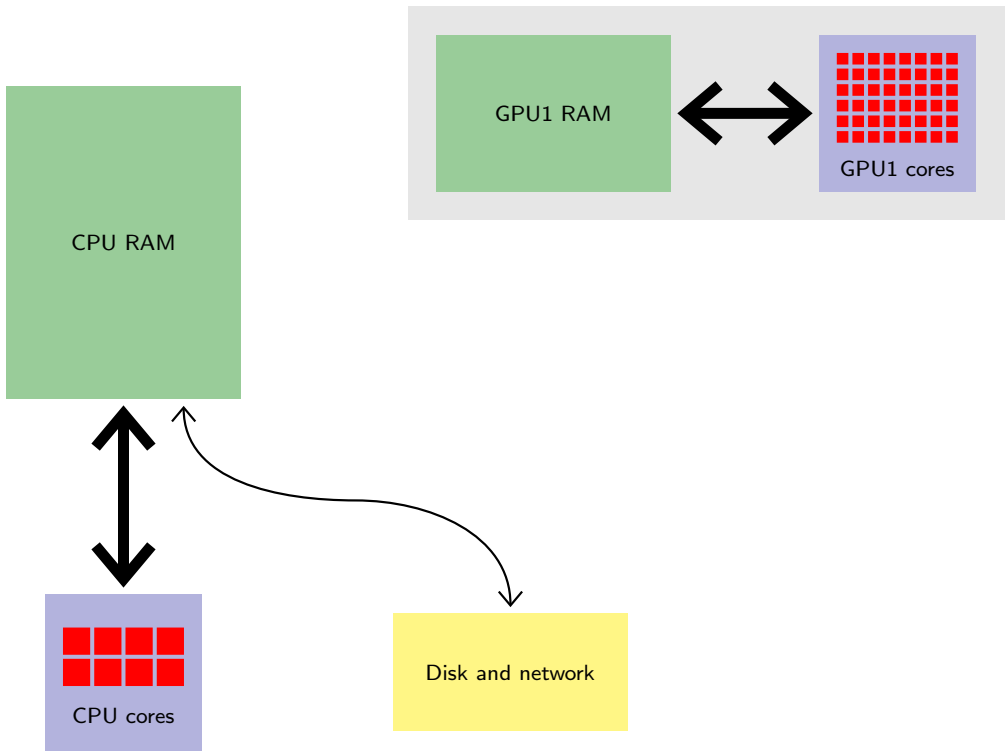

CPU cores


CPU RAM
CPU cores

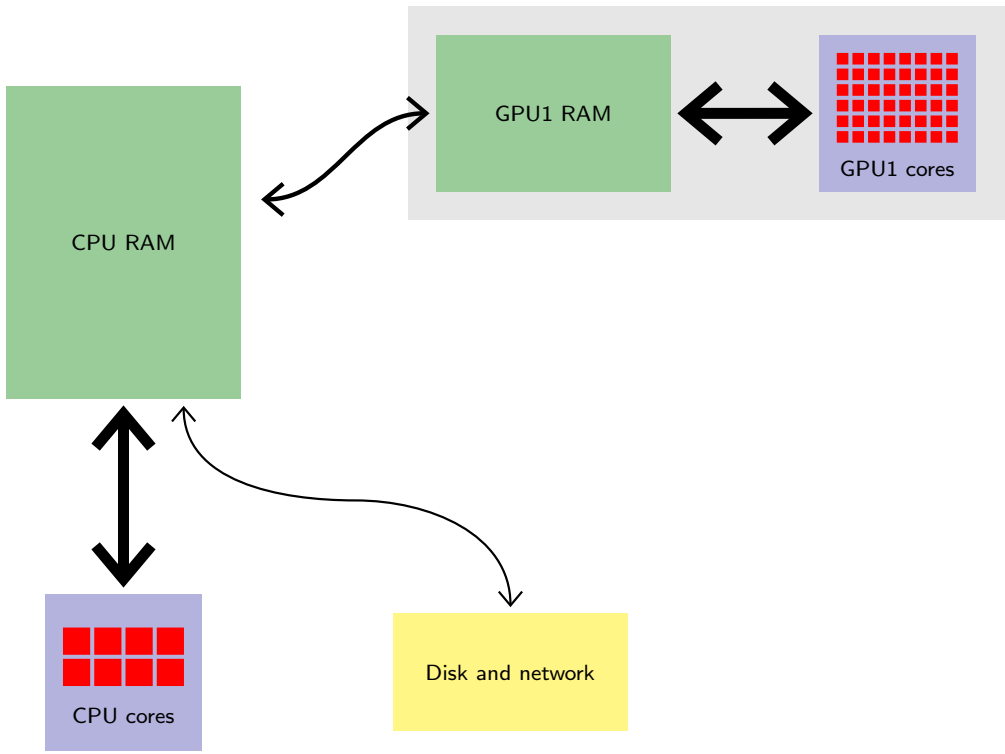

CPU RAM
CPU cores
Disk and network



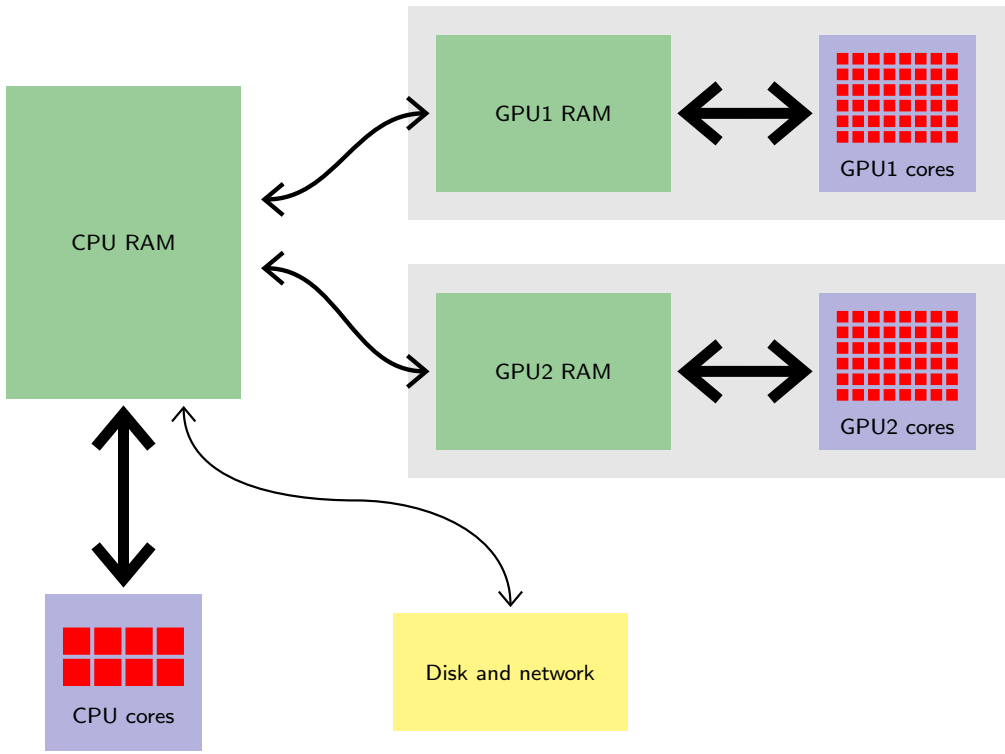
A standard NVIDIA GTX 3090 has $10,500$ computing cores clocked at 1.5GHz, and delivers a peak performance of $\simeq 35$ TFlops.



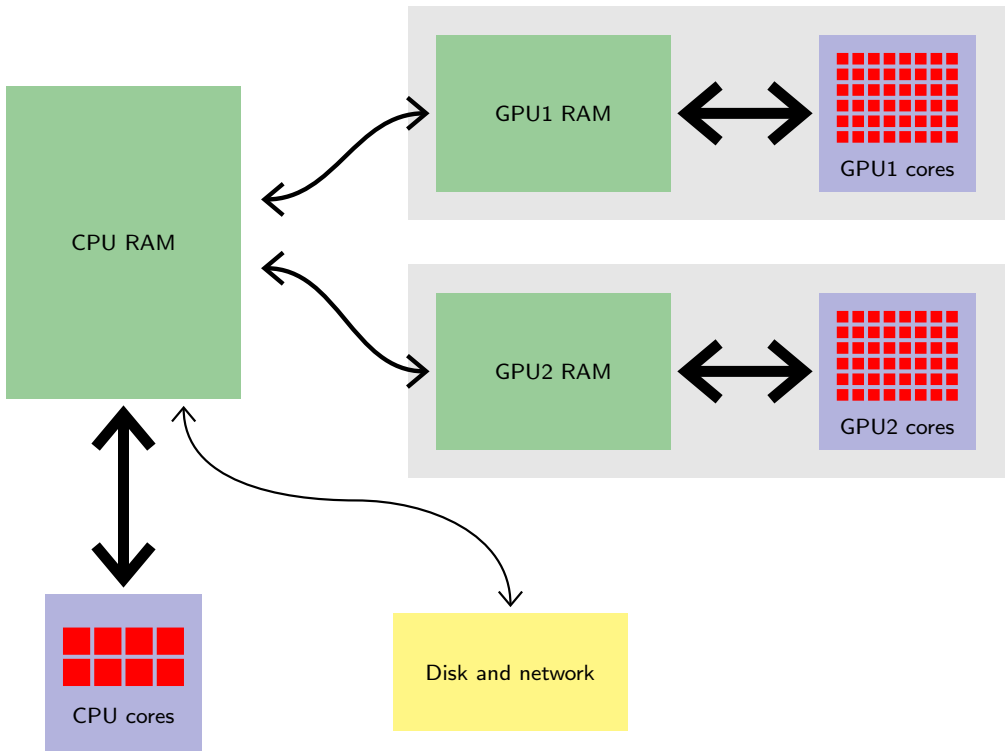
A standard NVIDIA GTX 3090 has $10,500$ computing cores clocked at 1.5GHz, and delivers a peak performance of $\simeq 35$ TFlops.



A standard NVIDIA GTX 3090 has $10,500$ computing cores clocked at 1.5GHz, and delivers a peak performance of $\simeq 35$ TFlops.



A standard NVIDIA GTX 3090 has $10,500$ computing cores clocked at 1.5GHz, and delivers a peak performance of $\simeq 35$ TFlops.



A standard NVIDIA GTX 3090 has $10,500$ computing cores clocked at 1.5GHz, and delivers a peak performance of $\simeq 35$ TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.

TABLE 7. COMPARATIVE EXPERIMENT RESULTS (TIME PER MINI-BATCH IN SECOND)

		Desktop CPU (Threads used)				Server CPU (Threads used)						Single GPU		
		1	2	4	8	1	2	4	8	16	32	G980	G1080	K80
FCN-S	Caffe	1.324	0.790	**0.578**	15.444	1.355	0.997	0.745	**0.573**	0.608	1.130	0.041	**0.030**	0.071
	CNTK	1.227	0.660	**0.435**	-	1.340	0.909	0.634	0.488	**0.441**	1.000	0.045	**0.033**	0.074
	TF	7.062	4.789	2.648	**1.938**	9.571	6.569	3.399	1.710	0.946	**0.630**	0.060	**0.048**	0.109
	MXNet	4.621	2.607	2.162	**1.831**	5.824	3.356	2.395	2.040	**1.945**	2.670	-	**0.106**	0.216
	Torch	1.329	0.710	**0.423**	-	1.279	1.131	0.595	0.433	**0.382**	1.034	0.040	**0.031**	0.070
AlexNet-S	Caffe	1.606	0.999	**0.719**	-	1.533	1.045	**0.797**	0.850	0.903	1.124	0.034	**0.021**	0.073
	CNTK	3.761	1.974	**1.276**	-	3.852	2.600	1.567	1.347	**1.168**	1.579	0.045	**0.032**	0.091
	TF	6.525	2.936	1.749	**1.535**	5.741	4.216	2.202	1.160	**0.701**	0.962	0.059	**0.042**	0.130
	MXNet	2.977	2.340	2.250	**2.163**	3.518	3.203	2.926	2.828	**2.827**	2.887	0.020	**0.014**	0.042
	Torch	4.645	2.429	**1.424**	-	4.336	2.468	1.543	1.248	**1.090**	1.214	0.033	**0.023**	0.070
RenNet-50	Caffe	11.554	7.671	**5.652**	-	10.643	8.600	6.723	**6.019**	6.654	8.220	-	**0.254**	0.766
	CNTK	-	-	-	-	-	-	-	-	-	-	0.240	**0.168**	0.638
	TF	23.905	16.435	10.206	**7.816**	29.960	21.846	11.512	6.294	**4.130**	4.351	0.327	**0.227**	0.702
	MXNet	48.000	46.154	44.444	**43.243**	57.831	57.143	54.545	54.545	**53.333**	55.172	0.207	**0.136**	0.449
	Torch	13.178	7.500	**4.736**	4.948	12.807	8.391	5.471	4.164	**3.683**	4.422	0.208	**0.144**	0.523
FCN-R	Caffe	2.476	1.499	**1.149**	-	2.282	1.748	1.403	1.211	1.127	**1.127**	0.025	**0.017**	0.055
	CNTK	1.845	0.970	0.661	**0.571**	1.592	0.857	0.501	0.323	**0.252**	0.280	0.025	**0.017**	0.053
	TF	2.647	1.913	1.157	**0.919**	3.410	2.541	1.297	0.661	0.361	**0.325**	0.033	**0.020**	0.063
	MXNet	1.914	1.072	0.719	**0.702**	1.609	1.065	0.731	0.534	0.451	**0.447**	0.029	**0.019**	0.060
	Torch	1.670	0.926	**0.565**	0.611	1.379	0.915	0.662	0.440	0.402	**0.366**	0.025	**0.016**	0.051
AlexNet-R	Caffe	3.558	2.587	**2.157**	2.963	4.270	3.514	3.381	**3.364**	4.139	4.930	0.041	**0.027**	0.137
	CNTK	9.956	7.263	**5.519**	6.015	9.381	6.078	4.984	**4.765**	6.256	6.199	0.045	**0.031**	0.108
	TF	4.535	3.225	1.911	**1.565**	6.124	4.229	2.200	1.396	1.036	**0.971**	**0.227**	0.317	0.385
	MXNet	13.401	12.305	12.278	**11.950**	17.994	17.128	16.764	**16.471**	17.471	17.770	0.060	**0.032**	0.122
	Torch	5.352	3.866	**3.162**	3.259	6.554	5.288	4.365	**3.940**	4.157	4.165	0.069	**0.043**	0.141
RenNet-56	Caffe	6.741	5.451	**4.989**	6.691	7.513	**6.119**	6.232	6.689	7.313	9.302	-	**0.116**	0.378
	CNTK	-	-	-	-	-	-	-	-	-	-	0.206	**0.138**	0.562
	TF	-	-	-	-	-	-	-	-	-	-	0.225	**0.152**	0.523
	MXNet	34.409	31.255	**30.069**	31.388	44.878	43.775	**42.299**	42.965	43.854	44.367	0.105	**0.074**	0.270
	Torch	5.758	3.222	**2.368**	2.475	8.691	4.965	3.040	**2.560**	2.575	2.811	0.150	**0.101**	0.301
LSTM	Caffe	-	-	-	-	-	-	-	-	-	-	-	-	-
	CNTK	0.186	0.120	**0.090**	0.118	0.211	0.139	0.117	0.114	**0.114**	0.198	0.018	**0.017**	0.043
	TF	4.662	3.385	1.935	**1.532**	6.449	4.351	2.238	1.183	0.702	**0.598**	0.133	**0.065**	0.140
	MXNet	-	-	-	-	-	-	-	-	-	-	0.089	**0.079**	0.149
	Torch	6.921	3.831	**2.682**	3.127	7.471	4.641	3.580	**3.260**	5.148	5.851	0.399	**0.324**	0.560

Note: The mini-batch sizes for FCN-S, AlexNet-S, ResNet-50, FCN-R, AlexNet-R, ResNet-56 and LSTM are 64, 16, 16, 1024, 1024, 128 and 128 respectively.

(Shi et al., 2016)

The current standard to program a GPU is through the CUDA ("Compute Unified Device Architecture") model, defined by NVIDIA.

Alternatives are OpenCL, backed by several CPU/DSP manufacturers, and more recently AMD's HIP/ROCm.

Google developed its own line of processors for deep learning dubbed TPU ("Tensor Processing Unit") which offer excellent flops/watt performance.

In practice, as of today (29.03.2022), NVIDIA hardware remains the default choice for deep learning, and CUDA is the reference framework in use.

From a practical perspective, libraries interface the framework (e.g. PyTorch) with the "computational backend" (e.g. CPU or GPU)

- BLAS ("Basic Linear Algebra Subprograms"): vector/matrix products, and the cuBLAS implementation for NVIDIA GPUs,
- LAPACK ("Linear Algebra Package"): linear system solving, Eigen-decomposition, etc.
- cuDNN ("NVIDIA CUDA Deep Neural Network library") computations specific to deep-learning on NVIDIA GPUs.

Using GPUs in PyTorch

The use of the GPUs in PyTorch is done by creating or copying tensors into their memory.

Operations on tensors in a device's memory are done by the said device.

As for the type, the device can be specified to the creation operations as a device, or as a string that will implicitly be converted to a device.

```
>>> x = torch.zeros(10, 10)
>>> x.device
device(type='cpu')
>>> x = torch.zeros(10, 10, device = torch.device('cuda'))
>>> x.device
device(type='cuda', index=0)
>>> x = torch.zeros(10, 10, device = torch.device('cuda:1'))
>>> x.device
device(type='cuda', index=1)
>>> x = torch.zeros(10, 10, device = 'cuda:0')
>>> x.device
device(type='cuda', index=0)
```

The `torch.Tensor.to(device)` returns a clone on the specified device **if the tensor is not already there** or returns the tensor itself if it was already there.

The argument `device` can be either a string, or a device.

Alternatives are `torch.Tensor.cuda([gpu_id])` and `torch.Tensor.cpu()`.

Moving data between the CPU and the GPU memories is far slower than moving it inside the GPU memory.

```
>>> u = torch.tensor([1, 2, 3])
>>> u.device
device(type='cpu')
>>> v = u.to('cuda') # copy of u
>>> v
tensor([1, 2, 3], device='cuda:0')
>>> v[0] = 5
>>> u
tensor([1, 2, 3])
>>> w = u.to('cpu') # this is u itself
>>> w
tensor([1, 2, 3])
>>> w[0] = 5
>>> u
tensor([5, 2, 3])
```

```
>>> m = torch.randn(10, 10)
>>> m.device
device(type='cpu')
>>> x = torch.randn(10, 100)
>>> q = m@x
>>> q.device
device(type='cpu')
```

```
>>> m = m.to('cuda')
>>> x = x.to('cuda')
>>> q = m@x # This is done on GPU (#0)
>>> q.device
device(type='cuda', index=0)
```

Since operations maintain the types and devices of the tensors, you generally do not need to worry about making your code generic regarding these aspects.

To explicitly create new tensors you can use a tensor's `new_*()` methods.

```
>>> u = torch.randn(3, 5, dtype = torch.float64)
>>> v = u.new_zeros(1, 2)
>>> v
tensor([[0., 0.]], dtype=torch.float64)
>>> w = torch.empty(3, 5, dtype = torch.float16,
...                 device = 'cuda:1').fill_(1.0)
>>> w.new_full((2, 3), 1.4)
tensor([[1.4004, 1.4004, 1.4004],
        [1.4004, 1.4004, 1.4004]], device='cuda:1', dtype=torch.float16)
```

Apart from `copy_()`, operations cannot mix different tensor types or devices:

```
>>> import torch
>>> x = torch.randn(3, 5)
>>> y = torch.randn(3, 5).to('cuda')
>>> x.copy_(y)
tensor([[ 0.4071,  0.7589, -0.5321,  0.9103, -1.4985],
        [-0.1059,  2.1554, -0.0774, -0.4520,  1.5123],
        [ 0.1322,  0.1002, -0.4071,  1.8927, -0.5800]])

>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Expected object of type torch.FloatTensor but found type
torch.cuda.FloatTensor for argument #3 'other'
```

Similarly if multiple GPUs are available, cross-GPUs operations are not allowed by default, with the exception of `copy_()`.

Another exception to this rule are 0d tensors, which act as scalars and can be combined without device constraint.

The method `torch.Module.to(device)` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the specified device.

Although they do not have a "_" in their names, these `Module` operations make changes in-place.

The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available, so a typical GPU-friendly code would start with

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

and then have some `device = device` in some places, and/or

```
model.to(device)
criterion.to(device)
train_input, train_target = train_input.to(device), train_target.to(device)
test_input, test_target = test_input.to(device), test_target.to(device)
```

Multiple GPUs with `nn.DataParallel`

A very simple way to leverage multiple GPUs is to wrap the model in a `nn.DataParallel`.

The `forward` of `nn.DataParallel(my_module)` will

1. split the input mini-batch along the first dimension in as many mini-batches as there are GPUs,
2. send them to the `forward`s of clones of `my_module` located on each GPU,
3. concatenate the results.

And it is (of course!) autograd-compliant.

If we define a simple module to printout the calls to `forward`.

```
class Dummy(nn.Module):
    def __init__(self, m):
        super().__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), x.device)
        return self.m(x)
```

```
x = torch.randn(50, 10)
model = Dummy(nn.Linear(10, 5))

print('On CPU')
y = model(x)

x = x.to('cuda')
model.to('cuda')

print('On GPU w/o nn.DataParallel')
y = model(x)

print('On GPU w/ nn.DataParallel')
parallel_model = nn.DataParallel(model)
y = parallel_model(x)
```

will print, on a machine with two GPUs:

```
On CPU
Dummy.forward torch.Size([50, 10]) cpu
On GPU w/o nn.DataParallel
Dummy.forward torch.Size([50, 10]) cuda:0
On GPU w/ nn.DataParallel
Dummy.forward torch.Size([25, 10]) cuda:0
Dummy.forward torch.Size([25, 10]) cuda:1
```

The end