EE-559 – Deep learning

6. Going deeper

François Fleuret
https://fleuret.org/dlc/

[version of: March 14, 2018]

RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Benefits and challenges of greater depth

For image classification for instance, there has been a trend toward deeper architectures to improve performance.

Network	Nb. layers
LeNet5 (leCun et al., 1998)	5
AlexNet (Krizhevsky et al., 2012)	8
VGG (Simonyan and Zisserman, 2014)	11–19
GoogleLeNet (Szegedy et al., 2015)	22
Inception v4 (Szegedy et al., 2016)	76
Resnet (He et al., 2015)	34–152
Resnet (He et al., 2016)	1001
Resnet (Huang et al., 2016)	1202

For image classification for instance, there has been a trend toward deeper architectures to improve performance.

Network	Nb. layers
LeNet5 (leCun et al., 1998)	5
AlexNet (Krizhevsky et al., 2012)	8
VGG (Simonyan and Zisserman, 2014)	11–19
GoogleLeNet (Szegedy et al., 2015)	22
Inception v4 (Szegedy et al., 2016)	76
Resnet (He et al., 2015)	34–152
Resnet (He et al., 2016)	1001
Resnet (Huang et al., 2016)	1202

"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)

For image classification for instance, there has been a trend toward deeper architectures to improve performance.

Network	Nb. layers
LeNet5 (leCun et al., 1998)	5
AlexNet (Krizhevsky et al., 2012)	8
VGG (Simonyan and Zisserman, 2014)	11–19
GoogleLeNet (Szegedy et al., 2015)	22
Inception v4 (Szegedy et al., 2016)	76
Resnet (He et al., 2015)	34–152
Resnet (He et al., 2016)	1001
Resnet (Huang et al., 2016)	1202

> "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)

Regarding the representation capacity, a theoretical analysis provides an intuition of how the "irregularity" of the output of a network grows linearly with its width and exponentially with its depth.

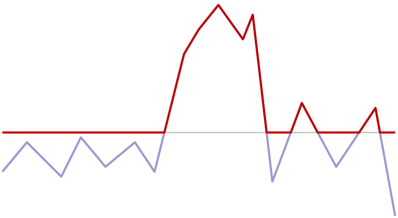
Let $\mathscr{F}$ be the set of piece-wise linear mappings on $[0,1]$, and $\forall f \in \mathscr{F}$, let $\kappa(f)$ be the minimum number of linear pieces needed to represent f.

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 needed to represent 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 needed to represent f.



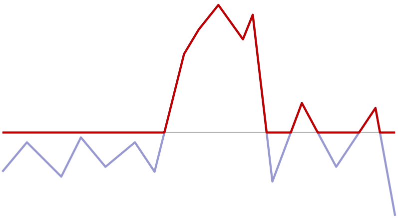
Let σ be the ReLU function

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

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

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

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 needed to represent f.



Let σ be the ReLU function

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

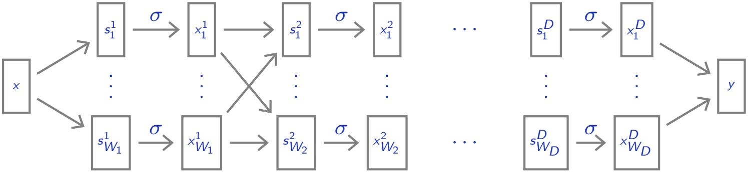
If we compose σ and $f \in \mathscr{F}$, any linear piece that does not cross 0 remains a single piece or disappears, and one that does cross 0 breaks into two, hence

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

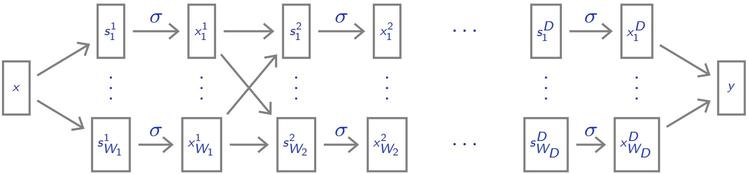
and we also have

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

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

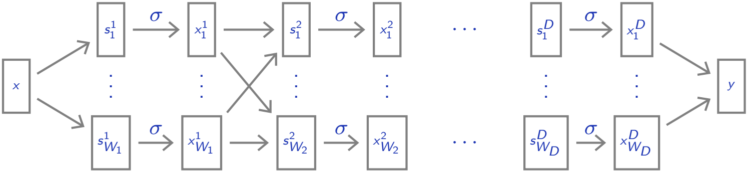


Consider a MLP with ReLU, a single input unit, and a single output unit.



All the s_i^ds and x_i^ds are piece-wise linear functions of x

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



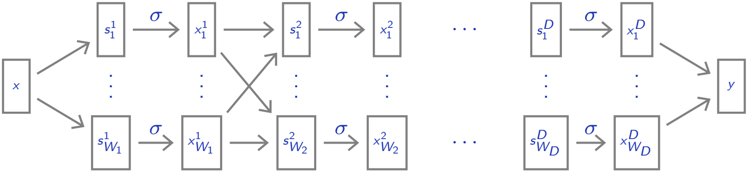
All the s_i^ds and x_i^ds are piece-wise linear functions of x with $\forall i, \kappa(s_i^1) = 1$, and

$$\forall l, i, \kappa\left(x_i^l\right) = \kappa\left(\sigma(s_i^l)\right) \leq 2\kappa\left(s_i^l\right) \leq 2 \sum_{j=1}^{W_{l-1}} \kappa\left(x_j^{l-1}\right)$$

from which

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

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



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

$$\forall l, i, \kappa\left(x_i^l\right) = \kappa\left(\sigma(s_i^l)\right) \leq 2\kappa\left(s_i^l\right) \leq 2 \sum_{j=1}^{W_{l-1}} \kappa\left(x_j^{l-1}\right)$$

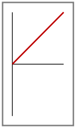
from which

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

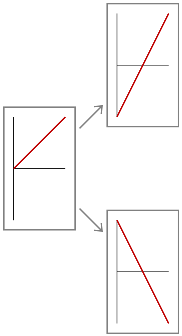
and we get the following bound on the κ of any ReLU MLP

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

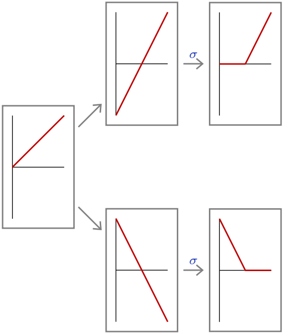
We can hand-design a network that [almost] reaches the bound:



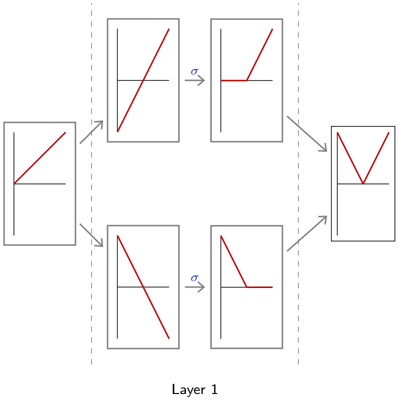
We can hand-design a network that [almost] reaches the bound:



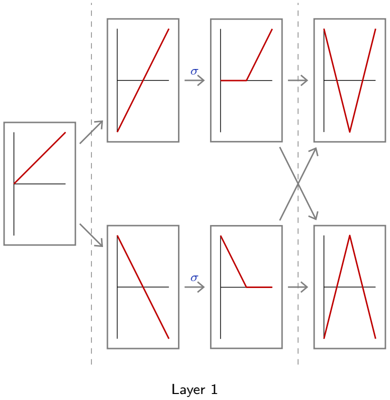
We can hand-design a network that [almost] reaches the bound:



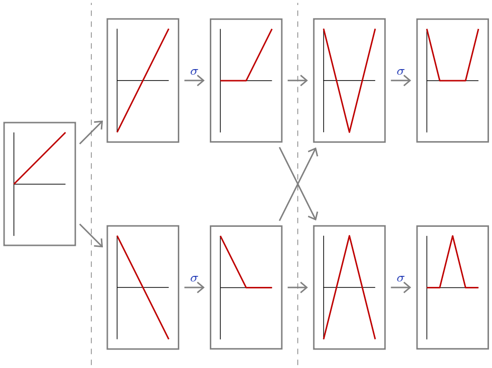
We can hand-design a network that [almost] reaches the bound:



We can hand-design a network that [almost] reaches the bound:

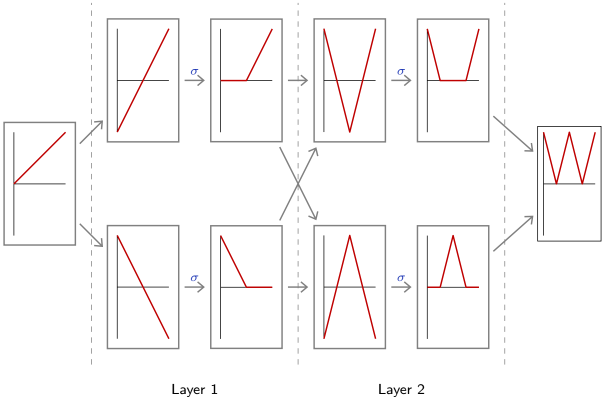


We can hand-design a network that [almost] reaches the bound:

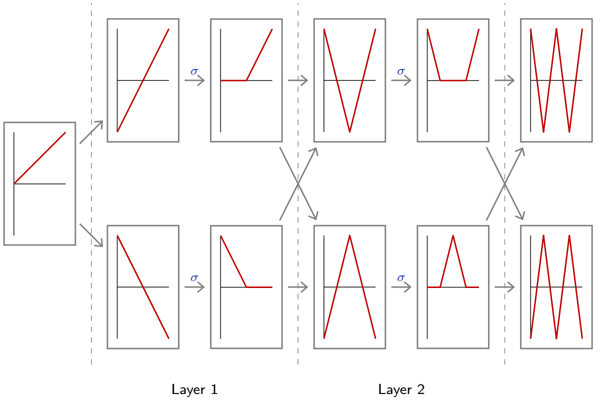


Layer 1

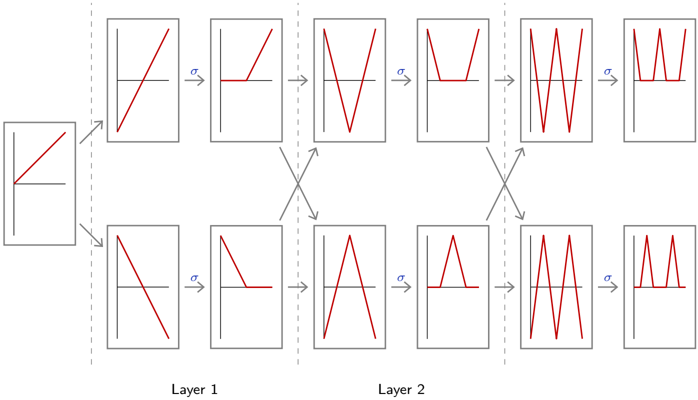
We can hand-design a network that [almost] reaches the bound:



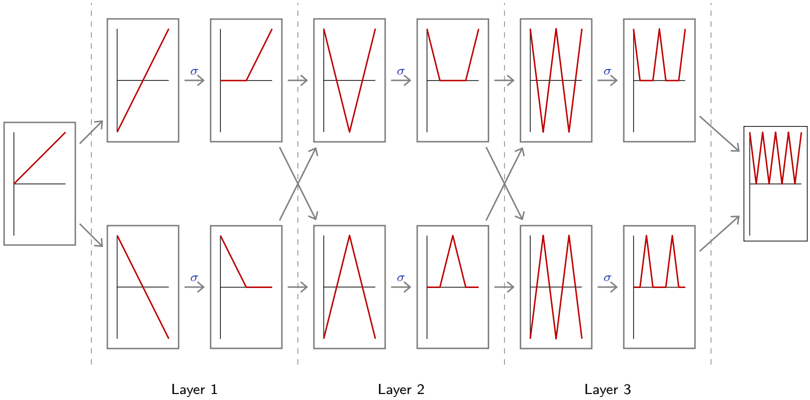
We can hand-design a network that [almost] reaches the bound:



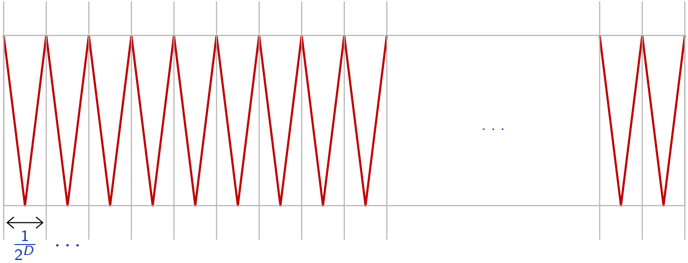
We can hand-design a network that [almost] reaches the bound:

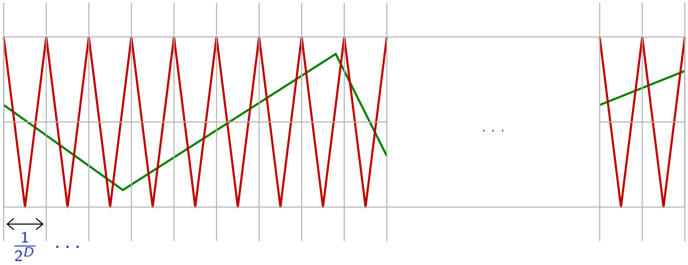


We can hand-design a network that [almost] reaches the bound:

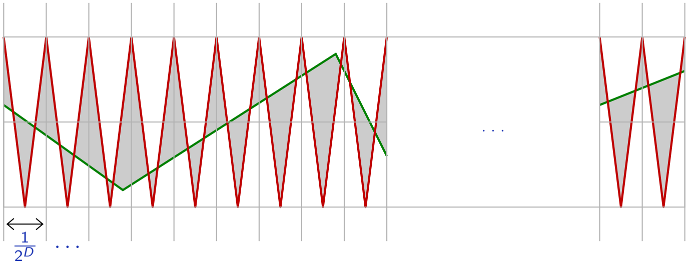


So there is a network with D hidden layers and $2D$ hidden units which computes an $f : [0,1] \to [0,1]$ of period $1/2^D$



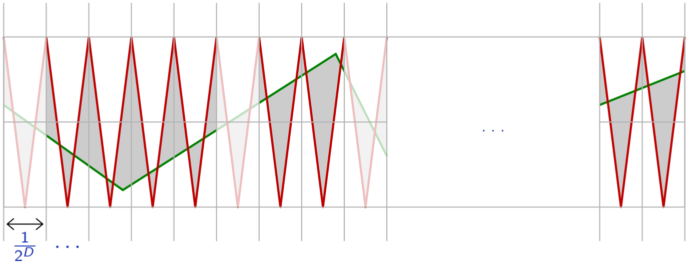


Given $g \in \mathscr{F}$



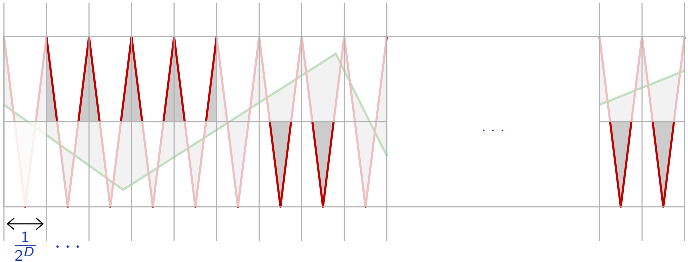
Given $g \in \mathscr{F}$

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



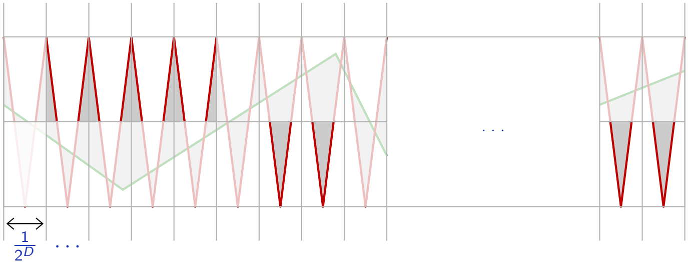
Given $g \in \mathscr{F}$, it crosses $\frac{1}{2}$ at most $\kappa(g)$ times

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



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

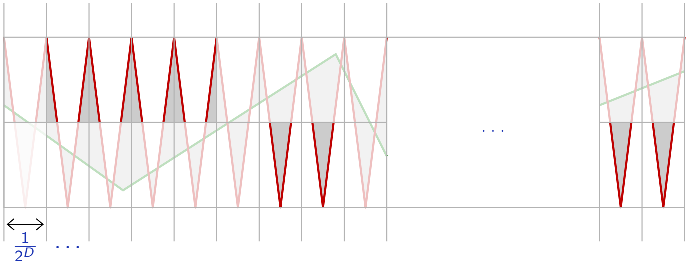
$$\int_0^1 |f(x) - g(x)|$$



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

$$\int_0^1 |f(x) - g(x)| \geq \left(2^D - \kappa(g)\right) \frac{1}{2} \int_0^{1/2^D} |f(x)|$$

$$= \frac{1}{8}\left(1 - \frac{\kappa(g)}{2^D}\right)$$



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

$$\int_0^1 |f(x) - g(x)| \geq \left(2^D - \kappa(g)\right) \frac{1}{2} \int_0^{1/2^D} |f(x)|$$
$$= \frac{1}{8}\left(1 - \frac{\kappa(g)}{2^D}\right)$$

And we multiply f by 8 to get our final result.

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 $\{W_1, \dots, W_D\}$, we have

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

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 $\{W_1, \dots, W_D\}$, we have

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

For instance, with g a single hidden layer network

$$\|f - g\|_1 \geq 1 - 2\frac{W_1}{2^{D^*}}.$$

To approximate f properly, the width of g's hidden layer has to increase exponentially with f's depth.

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 $\{W_1, \dots, W_D\}$, we have

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

For instance, with g a single hidden layer network

$$\|f - g\|_1 \geq 1 - 2\frac{W_1}{2^{D^*}}.$$

To approximate f properly, the width of g's hidden layer has to increase exponentially with f's depth.

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

We saw that an important issue to train deep architectures is to control the amplitude of the gradient, which is tightly related to controlling activations.

We saw that an important issue to train deep architectures 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.

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.

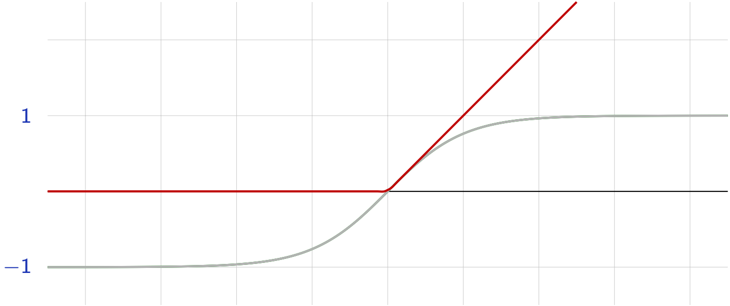
Modern techniques change the functional itself instead of trying to improve training "from the outside" through penalty terms or better optimizers.

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.

Rectifiers

The use of the ReLU activation function was a great improvement compared to the historical tanh.



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.

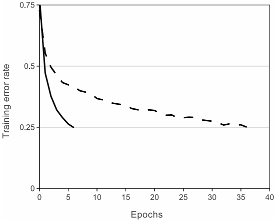


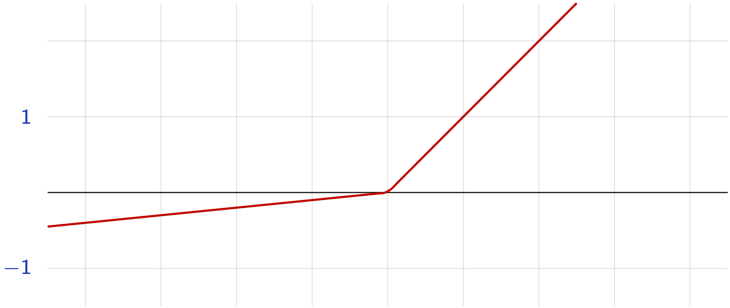
Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. 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

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

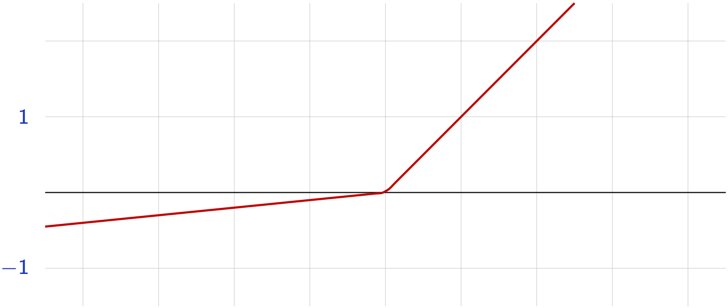
with $0 \leq a < 1$ usually small.



A first variant of ReLU is Leaky-ReLU

$$\begin{aligned} \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto \max(ax, x) \end{aligned}$$

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



The parameter a can be either fixed or optimized during training.

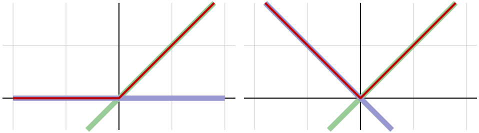
The "maxout" layer proposed by Goodfellow et al. (2013) takes the max of several linear units. This is not an activation function in the usual sense, since it has trainable parameters.

$$
\begin{aligned}
h : \mathbb{R}^D &\rightarrow \mathbb{R}^M \\
x &\mapsto \left(\max_{j=1}^{K} x^T W_{1,j} + b_{1,j}, \dots, \max_{j=1}^{K} x^T W_{M,j} + b_{M,j} \right)
\end{aligned}
$$

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.

$$
\begin{aligned}
h : \mathbb{R}^D &\to \mathbb{R}^M \\
x &\mapsto \left(\max_{j=1}^{K} x^T W_{1,j} + b_{1,j}, \dots, \max_{j=1}^{K} x^T W_{M,j} + b_{M,j} \right)
\end{aligned}
$$

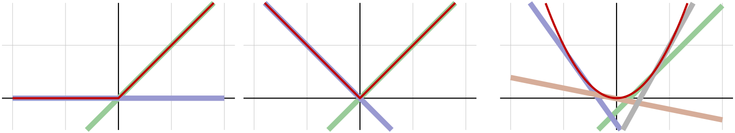
It can in particular encode ReLU and absolute value



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.

$$h : \mathbb{R}^D \to \mathbb{R}^M$$
$$x \mapsto \left(\max_{j=1}^{K} x^T W_{1,j} + b_{1,j}, \dots, \max_{j=1}^{K} x^T W_{M,j} + b_{M,j} \right)$$

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



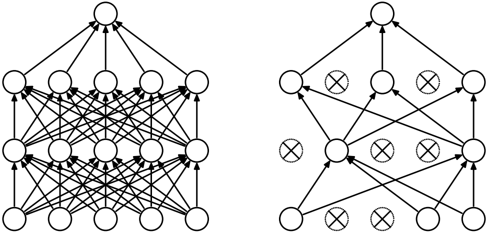
A more recent proposal is the “Concatenated Rectified Linear Unit” (CReLU) proposed by Shang et al. (2016):

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

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

Dropout

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.

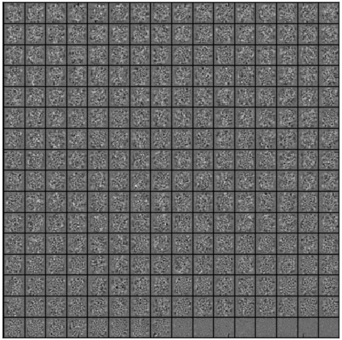
(Srivastava et al., 2014)

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

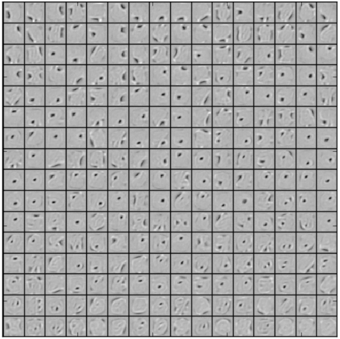
> "In a standard neural network, the derivative received by each parameter tells it how it should change so the final loss function is reduced, given what all other units are doing. Therefore, units may change in a way that they fix up the mistakes of the other units. This may lead to complex co-adaptations. 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 network with dropout can be interpreted as an ensemble of 2^N models with heavy weight sharing (Goodfellow et al., 2013).



(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)

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

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.

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.

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

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

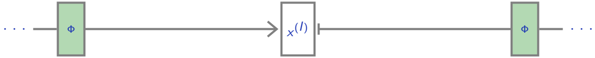
During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.

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

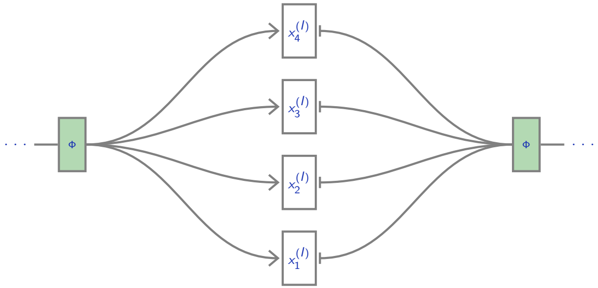
The standard variant in use is the "inverted dropout". It multiplies activations by $\frac{1}{1-p}$ during train and keeps the network untouched during test.

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

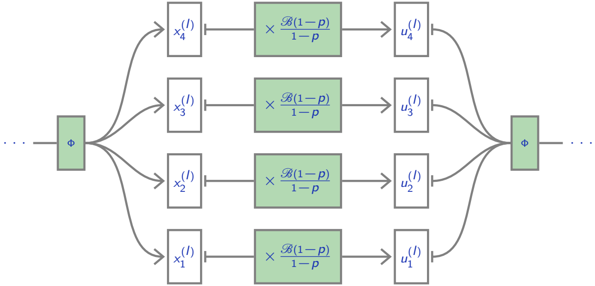
Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



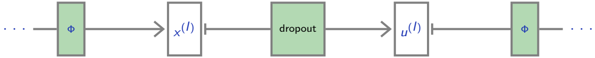
Dropout is not implemented by actually switching off units, but equivalently as a module that drops activations at random on each sample.



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 `torch.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.

dropout is implemented in PyTorch as `torch.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 = Variable(Tensor(3, 9).fill_ (1.0), requires_grad = True)
>>> x.data

    1     1     1     1     1     1     1     1     1
    1     1     1     1     1     1     1     1     1
    1     1     1     1     1     1     1     1     1
[torch.FloatTensor of size 3x9]

>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y.data

    4     0     4     4     4     0     4     0     0
    4     0     0     0     0     0     0     0     0
    0     0     0     0     4     0     4     0     4
[torch.FloatTensor of size 3x9]

>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad.data

 1.7889  0.0000  1.7889  1.7889  1.7889  0.0000  1.7889  0.0000  0.0000
 4.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000  2.3094  0.0000  2.3094  0.0000  2.3094
[torch.FloatTensor of size 3x9]
```

If we have a network

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

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.

⚠ 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.

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 = Variable(Tensor(2, 3, 2, 2).fill_(1.0))
>>> dropout2d(x)
Variable containing:
(0 ,0 ,.,.) =
  0  0
  0  0

(0 ,1 ,.,.) =
  0  0
  0  0

(0 ,2 ,.,.) =
  2  2
  2  2

(1 ,0 ,.,.) =
  2  2
  2  2

(1 ,1 ,.,.) =
  0  0
  0  0

(1 ,2 ,.,.) =
  2  2
  2  2
[torch.FloatTensor of size 2x3x2x2]
```

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

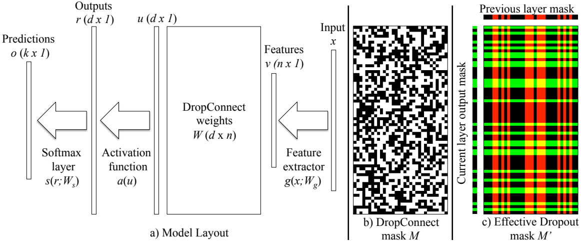


Figure 1. (a): An example model layout for a single DropConnect layer. After running feature extractor $g()$ on input x, a random instantiation of the mask M (e.g. (b)), masks out the weight matrix W. The masked weights are multiplied with this feature vector to produce u which is the input to an activation 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)

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

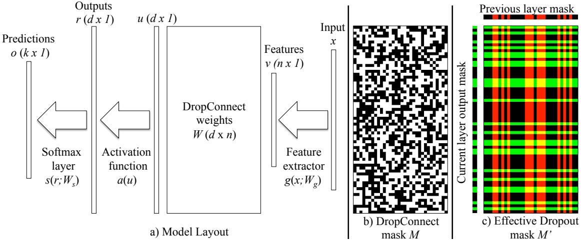


Figure 1. (a): An example model layout for a single DropConnect layer. After running feature extractor $g()$ on input x, a random instantiation of the mask M (e.g. (b)), masks out the weight matrix W. The masked weights are multiplied with this feature vector to produce u which is the input to an activation 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.

crop	rotation scaling	model	error(%) 5 network	voting error(%)
no	no	No-Drop	0.77±0.051	0.67
		Dropout	0.59±0.039	**0.52**
		DropConnect	0.63±0.035	0.57
yes	no	No-Drop	0.50±0.098	0.38
		Dropout	0.39±0.039	0.35
		DropConnect	0.39±0.047	**0.32**
yes	yes	No-Drop	0.30±0.035	**0.21**
		Dropout	0.28±0.016	0.27
		DropConnect	0.28±0.032	**0.21**

Table 3. MNIST classification error. Previous state of the art is *0.47*% (Zeiler and Fergus, 2013) for a single model without elastic distortions and *0.23*% with elastic distortions and voting (Ciresan et al., 2012).

(Wan et al., 2013)

Activation normalization

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

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.

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, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as internal covariate shift, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch."

(Ioffe and Szegedy, 2015)

"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, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as internal covariate shift, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch."

(Ioffe and Szegedy, 2015)

Batch normalization forces the activation first and second order moments, so that the following layers do not need to adapt to their drift.

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

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

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

The quantities $\hat{m}$ and $\hat{v}$ are respectively the component-wise data mean and variance estimated during training. The parameters γ and β are the desired moments, which are either fixed, or optimized during training.

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

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

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

The quantities $\hat{m}$ and $\hat{v}$ are respectively the component-wise data mean and variance estimated during training. The parameters γ and β are the desired moments, which are either fixed, or optimized during training.

During test, batch normalization is a simple component-wise linear transformation.

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

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

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

The quantities $\hat{m}$ and $\hat{v}$ are respectively the component-wise data mean and variance estimated during training. The parameters γ and β are the desired moments, which are either fixed, or optimized during training.

During test, batch normalization is a simple component-wise linear transformation.

If it is applied just before or after a fully connected layer, it can be integrated in it by changing its weights and biases appropriately.

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

If $x_1, \dots, x_B$ are the samples in the batch

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

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

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

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

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

If $x_1, \dots, x_B$ are the samples in the batch

$$
\begin{aligned}
\hat{m}_{batch} &= \frac{1}{B} \sum_{b=1}^{B} x_b \\
\hat{v}_{batch} &= \frac{1}{B} \sum_{b=1}^{B} (x_b - \hat{m}_{batch})^2 \\
\forall b = 1, \dots, B,\ z_b &= \frac{x_b - \hat{m}_{batch}}{\sqrt{\hat{v}_{batch} + \epsilon}} \\
y_b &= \gamma \odot z_b + \beta.
\end{aligned}
$$

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

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

If $x_1, \dots, x_B$ are the samples in the batch

$$
\begin{aligned}
\hat{m}_{batch} &= \frac{1}{B} \sum_{b=1}^{B} x_b \\
\hat{v}_{batch} &= \frac{1}{B} \sum_{b=1}^{B} (x_b - \hat{m}_{batch})^2 \\
\forall b = 1, \dots, B,\ z_b &= \frac{x_b - \hat{m}_{batch}}{\sqrt{\hat{v}_{batch} + \epsilon}} \\
y_b &= \gamma \odot z_b + \beta.
\end{aligned}
$$

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

⚠ Processing a batch jointly is unusual, as operations used in deep models can usually be formalized per-sample.

Dinh et al. (2016, appendix E) propose a variant with moving averages to deal with small batches.

As dropout, batch normalization is implemented as a separate module `torch.BatchNorm1d` that processes the input components separately.

```
>>> x = Tensor(10000, 3).normal_()
>>> x = x * Tensor([2, 5, 10]) + Tensor([-10, 25, 3])
>>> x = Variable(x)
>>> x.data.mean(0)

 -9.9952
 25.0467
  2.9453
[torch.FloatTensor of size 3]

>>> x.data.std(0)

  1.9780
  5.0530
 10.0587
[torch.FloatTensor of size 3]
```

Since the module has internal variables to keep statistics, it mush be provided with the sample dimension at creation.

```
>>> bn = nn.BatchNorm1d(3)
>>> bn.bias.data = Tensor([2, 4, 8])
>>> bn.weight.data = Tensor([1, 2, 3])
>>> y = bn(x)
>>> y.data.mean(0)

 2.0000
 4.0000
 8.0000
[torch.FloatTensor of size 3]

>>> y.data.std(0)

 1.0000
 2.0001
 3.0001
[torch.FloatTensor of size 3]
```

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

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

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

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

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

From which

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

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite hairy.

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite hairy.

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite hairy.

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

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

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite hairy.

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

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

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

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

Since **each input in the batch impacts all the outputs of the batch**, the derivative of the loss with respect to an input is quite hairy.

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

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

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

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

In standard implementation, $\hat{m}$ and $\hat{v}$ for test are estimated with a moving average during train, so that it can be implemented as a module which does not need an additional pass through the training samples.

Results on ImageNet's LSVRC2012:

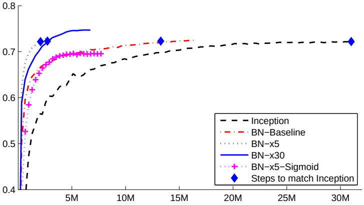


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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)

Results on ImageNet's LSVRC2012:

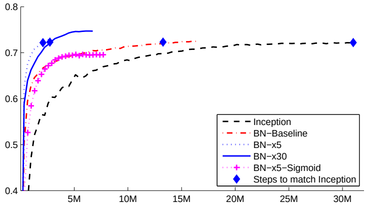


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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.*

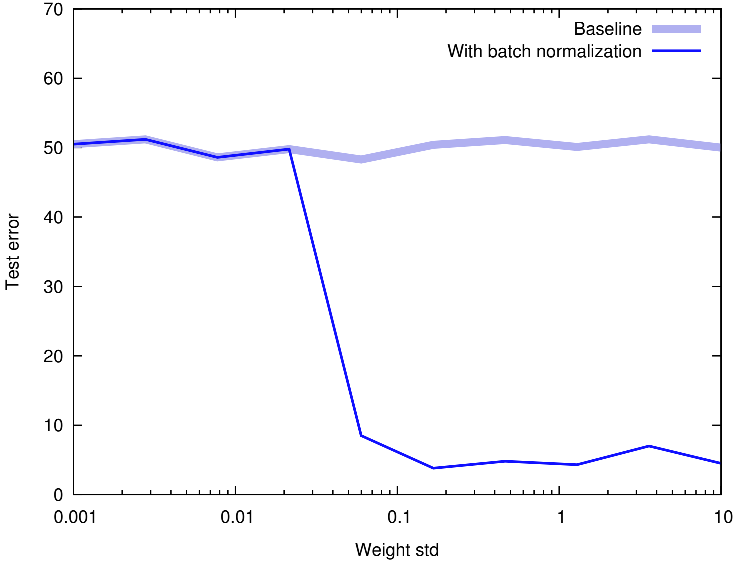
(Ioffe and Szegedy, 2015)

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, $\eta = 0.1$.

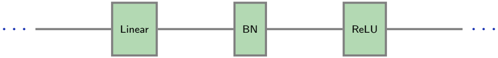
```
def create_model(with_batchnorm, nc = 32, depth = 16):
    modules = []

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

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

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

    return nn.Sequential(*modules)
```


Baseline
With batch normalization
Test error
Weight std
0
10
20
30
40
50
60
70
0.001
0.01
0.1
1
10

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."

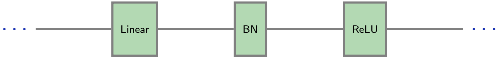
(Ioffe and Szegedy, 2015)



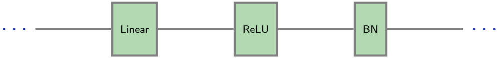
The position of batch normalization relative to the non-linearity is not clear.

"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.

A more recent variant 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

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

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

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

A more recent variant 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

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

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

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

Experiments show that it gives similar or better improvements than BN, while behaving similarly in train and test, and not processing batches jointly.

Residual networks

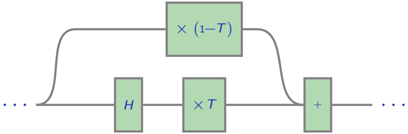
The “Highway networks” proposed by Srivastava et al. (2015) take advantage of the idea of gating developed for recurrent units. It replaces a standard non-linear layer

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

with a layer that includes a “gated” pass-through

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

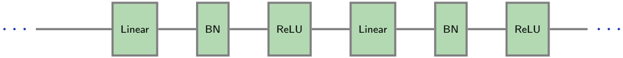
where $T(x; W_T) \in [0, 1]$ modulates how much the signal should be transformed.



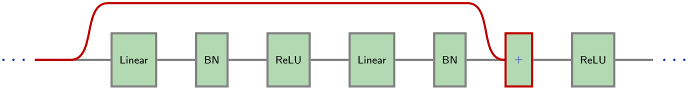
This technique allowed them to train networks with up to 100 layers.

The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a pass-through identity mapping.

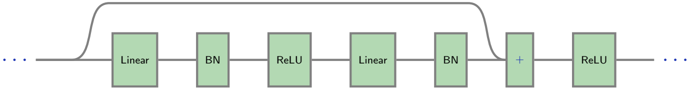
The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a pass-through identity mapping.



The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a pass-through identity mapping.

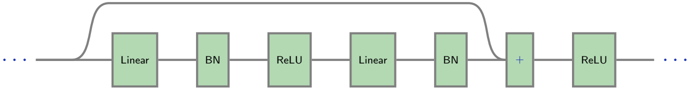


The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a pass-through identity mapping.



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.

The residual networks proposed by He et al. (2015) simplify the idea and use a building block with a pass-through identity mapping.



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.

Also, the network initialization is around the identity.

A technical point is to deal with convolution layers that change the activation map sizes or numbers of channels.

A technical point is to deal with convolution layers that change the activation map sizes or numbers of channels.

He et al. (2015) only consider:

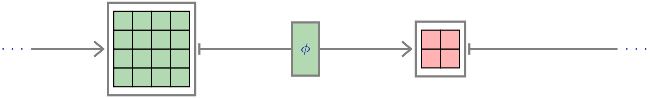
- reducing the activation map size by a factor 2,

A technical point is to deal with convolution layers that change the activation map sizes or numbers of channels.

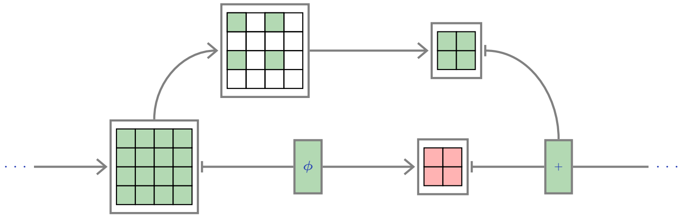
He et al. (2015) only consider:

- 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**.

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

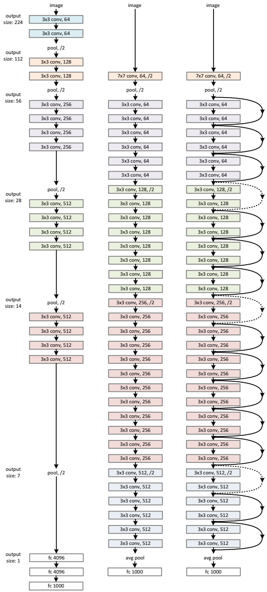


image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
output size: 14
pool, /2
3x3 conv, 512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

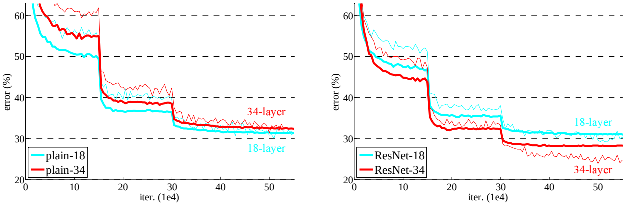
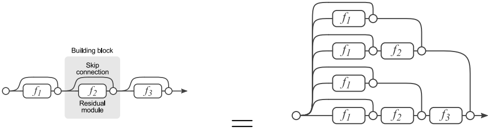
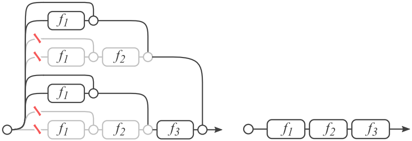


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

(He et al., 2015)

The output of a residual network can be understood as an ensemble, which explains in part its stability.



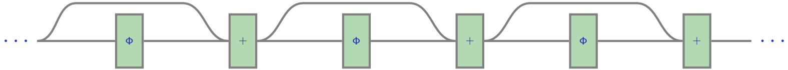


(Veit et al., 2016)

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."

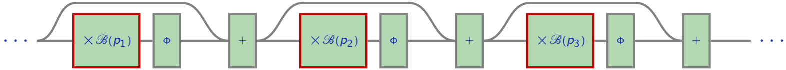
(Huang et al., 2016)



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)



The current state of the art on CIFAR10 and CIFAR100 (respectively 2.86% and 15.85% as of 22.08.2017) was obtained with a quite standard residual network using the "Shake-shake regularization".

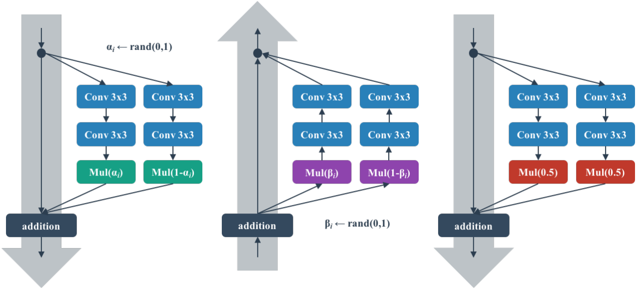


Figure 1: **Left:** Forward training pass. **Center:** Backward training pass. **Right:** At test time.

(Gastaldi, 2017)

Smart initialization

We saw that proper initialization is key, and taking into account the structure of the network help normalizing the weights adequately.

We saw that proper initialization is key, and taking into account the structure of the network help normalizing the weights adequately.

To go one step further, some techniques initialize the weights explicitly so that the empirical moments of the activations are as desired.

As such, they take into account the statistics of the network activation induced by the statistics of the data.

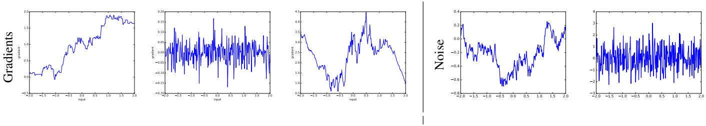
An example of such class of techniques is the **Layer-Sequential Unit-Variance** (LSUV) initialization (Mishkin and Matas, 2015).

It consists of

1. Initialize the weights of all layers with orthonormal matrices,
2. re-scale layers one after another in a forward direction, so that the empirical activation variance is 1.0.

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

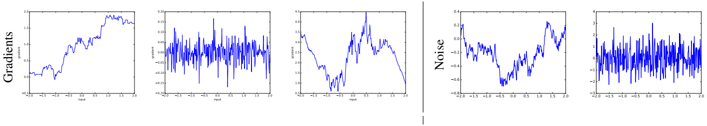
Balduzzi et al. (2017) points out that depth "shatters" the relation between the input and the gradient wrt the input, and that Resnets mitigate this effect.



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

(Balduzzi et al., 2017)

Balduzzi et al. (2017) points out that depth “shatters” the relation between the input and the gradient wrt 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 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.*

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

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

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

the CReLU non-linearity, *i.e.*

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

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

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

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

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

the CReLU non-linearity, *i.e.*

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

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

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

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, $\forall i = 1, \dots, B,\ j = 1, \dots, D'$:

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

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

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

Hence

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

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

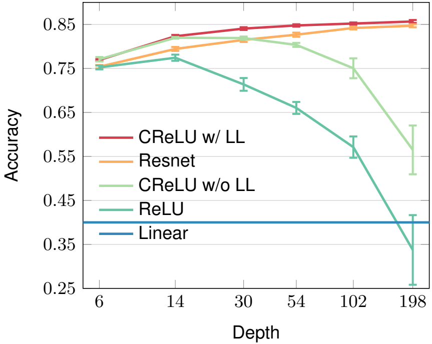


Figure 6: **CIFAR-10 test accuracy**. Comparison of test accuracy between networks of different depths with and without LL initialization.

(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,
- drop-out 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.

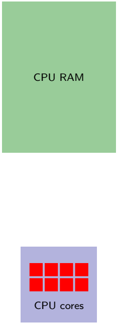
Using GPUs

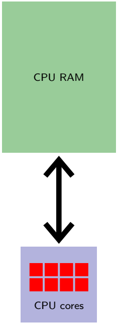
The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.

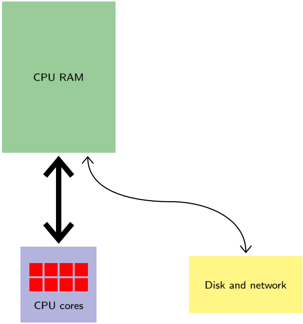
The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.

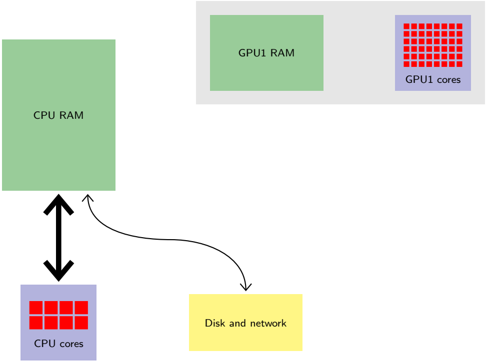
Although they were historically developed for mass-market real-time CGI, their massively parallel architecture is extremely fitting to signal processing and high dimension linear algebra.

Their use is instrumental in the success of deep-learning.

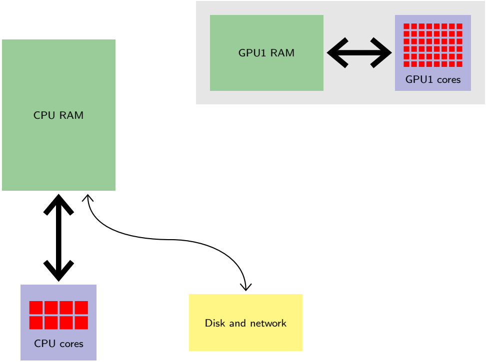

CPU RAM
CPU cores


CPU RAM
CPU cores

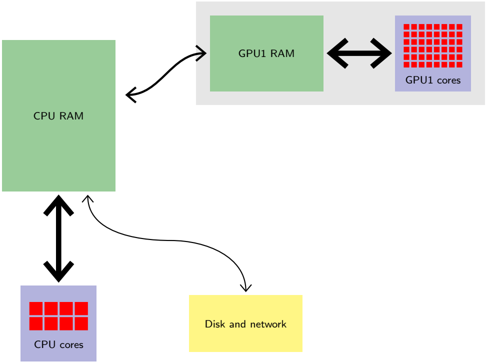

CPU RAM
CPU cores
Disk and network



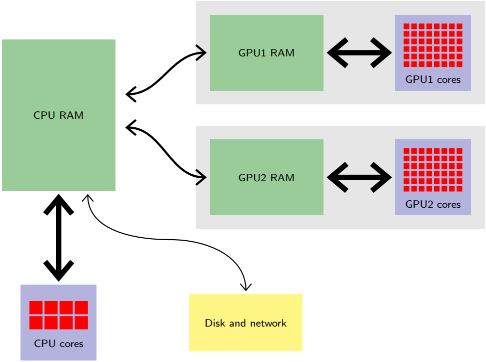
A standard NVIDIA GTX 1080 has $2,560$ single-precision computing cores clocked at 1.6GHz, and deliver a peak performance of $\simeq 9$ TFlops.



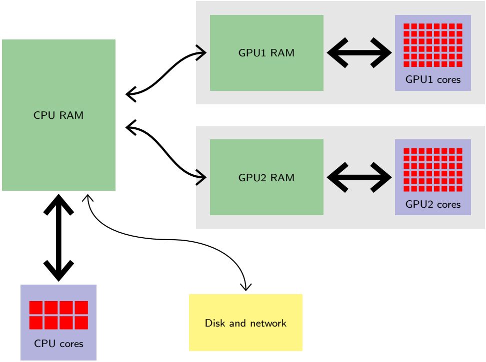
A standard NVIDIA GTX 1080 has $2,560$ single-precision computing cores clocked at 1.6GHz, and deliver a peak performance of $\simeq 9$ TFlops.



A standard NVIDIA GTX 1080 has $2,560$ single-precision computing cores clocked at 1.6GHz, and deliver a peak performance of $\simeq 9$ TFlops.



A standard NVIDIA GTX 1080 has $2,560$ single-precision computing cores clocked at 1.6GHz, and deliver a peak performance of $\simeq 9$ TFlops.



A standard NVIDIA GTX 1080 has $2,560$ single-precision computing cores clocked at 1.6GHz, and deliver a peak performance of $\simeq 9$ 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.

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

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD's HIP ("Heterogeneous-compute Interface for Portability").

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

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD's HIP ("Heterogeneous-compute Interface for Portability").

Google developed its own processor for deep learning dubbed TPU ("Tensor Processing Unit") for in-house use. It is targeted at TensorFlow and offers excellent flops/watt performance.

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

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD's HIP ("Heterogeneous-compute Interface for Portability").

Google developed its own processor for deep learning dubbed TPU ("Tensor Processing Unit") for in-house use. It is targeted at TensorFlow and offers excellent flops/watt performance.

In practice, as of today (16.08.2017), 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)

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.

The use of the GPUs in PyTorch is done by using the relevant tensor types.

Tensors of `torch.cuda` types are in the GPU memory. Operations on them are done by the GPU and resulting tensors are stored in its memory.

The use of the GPUs in PyTorch is done by using the relevant tensor types.

Tensors of `torch.cuda` types are in the GPU memory. Operations on them are done by the GPU and resulting tensors are stored in its memory.

Data type	CPU tensor	GPU tensor
8-bit int (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
64-bit int (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`
16-bit float	`torch.HalfTensor`	`torch.cuda.HalfTensor`
32-bit float	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit float	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`

Apart from `copy_()`, operations cannot mix different tensor types (CPU vs. GPU, or different numerical types):

```
>>> x = torch.FloatTensor(3, 5).normal_()
>>> y = torch.cuda.FloatTensor(3, 5).normal_()
>>> x.copy_(y)

-0.6817 -0.1927 -0.9117 -0.9456 -0.1488
-0.2441  0.5881  0.3959  0.7421 -0.5713
 0.8148 -0.7252  0.3839 -0.9684 -0.3364
[torch.FloatTensor of size 3x5]
```

Apart from `copy_()`, operations cannot mix different tensor types (CPU vs. GPU, or different numerical types):

```
>>> x = torch.FloatTensor(3, 5).normal_()
>>> y = torch.cuda.FloatTensor(3, 5).normal_()
>>> x.copy_(y)

-0.6817 -0.1927 -0.9117 -0.9456 -0.1488
-0.2441  0.5881  0.3959  0.7421 -0.5713
 0.8148 -0.7252  0.3839 -0.9684 -0.3364
[torch.FloatTensor of size 3x5]
>>> x+y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/fleuret/misc/anaconda3/lib/python3.5/site-packages/torch/tensor.py", line
        293, in __add__
    return self.add(other)
TypeError: add received an invalid combination of arguments - got (torch.cuda.
      FloatTensor), but expected one of:
 * (float value)
      didn't match because some of the arguments have invalid types: (torch.cuda.
            FloatTensor)
 * (torch.FloatTensor other)
      didn't match because some of the arguments have invalid types: (torch.cuda.
            FloatTensor)
 * (torch.SparseFloatTensor other)
      didn't match because some of the arguments have invalid types: (torch.cuda.
            FloatTensor)
 * (float value, torch.FloatTensor other)
 * (float value, torch.SparseFloatTensor other)
```

Operations maintain the type of the tensors, so you generally do not need to worry about making your code generic regarding the tensor types.

However, if you have to explicitly create a new tensor, the best is to use variables' `new()` method.

```
>>> def the_same_full_of_zeros_please(x):
...     return x.new(x.size()).zero_()
...
>>> u = torch.FloatTensor(3, 5).normal_()
>>> the_same_full_of_zeros_please(u)

 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
[torch.FloatTensor of size 3x5]

>>> v = torch.cuda.DoubleTensor(5,2).fill_(1.0)
>>> the_same_full_of_zeros_please(v)

 0  0
 0  0
 0  0
 0  0
 0  0
[torch.cuda.DoubleTensor of size 5x2 (GPU 0)]
```

Operations maintain the type of the tensors, so you generally do not need to worry about making your code generic regarding the tensor types.

However, if you have to explicitly create a new tensor, the best is to use variables' `new()` method.

```
>>> def the_same_full_of_zeros_please(x):
...     return x.new(x.size()).zero_()
...
>>> u = torch.FloatTensor(3, 5).normal_()
>>> the_same_full_of_zeros_please(u)

 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
[torch.FloatTensor of size 3x5]

>>> v = torch.cuda.DoubleTensor(5,2).fill_(1.0)
>>> the_same_full_of_zeros_please(v)

 0  0
 0  0
 0  0
 0  0
 0  0
[torch.cuda.DoubleTensor of size 5x2 (GPU 0)]
```

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

The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available.

The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available.

The `Tensor`s' method `cuda()` returns a clone on the GPU **if the tensor is not already there** or returns the tensor itself if it was already there, keeping the bit precision. Conversely the method `cpu()` makes a clone on the CPU if needed.

They both keep the original tensor unchanged.

The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

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

The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

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

A typical snippet of code to use the GPU would be

```
if torch.cuda.is_available():
    model.cuda()
    criterion.cuda()
    train_input, train_target = train_input.cuda(), train_target.cuda()
    test_input, test_target = test_input.cuda(), test_target.cuda()
```

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

An operation between tensors in the same GPU produces a results in the same GPU also.

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

An operation between tensors in the same GPU produces a results in the same GPU also.

Each GPU has a numerical id, and `torch.cuda.set_device(id)` allows to specify where GPU tensors should be moved by `cuda()`. An explicit id can also be provided to the latter.

`torch.cuda.device_of(obj)` selects the device to that of the specified tensor or storage.

A very simple way to leverage multiple GPUs is to use

```
torch.nn.DataParallel(module)
```

A very simple way to leverage multiple GPUs is to use

`torch.nn.DataParallel(module)`

The `forward` of the resulting 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 `module` located on each GPU,
3. concatenate the results.

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

For instance, on a machine with two GPUs

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

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

For instance, on a machine with two GPUs

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

    def forward(self, x):
        print('Dummy.forward', x.size(), torch.cuda.current_device())
        return self.m(x)

x = Variable(Tensor(50, 10).normal_())
m = Dummy(nn.Linear(10, 5))
x = x.cuda()
m = m.cuda()

print('Without data_parallel')
y = m(x)
print()

mp = nn.DataParallel(m)

print('With data_parallel')
y = mp(x)
```

prints

```
Without data_parallel
Dummy.forward torch.Size([50, 10]) 0

With data_parallel
Dummy.forward torch.Size([25, 10]) 0
Dummy.forward torch.Size([25, 10]) 1
```

The end

References

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

D. Balduzzi, M. Frean, L. Leary, J. Lewis, K. Wan-Duo Ma, and B. McWilliams. The shattered gradients problem: If resnets are the answer, then what is the question? *CoRR*, abs/1702.08591, 2017.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, Mar. 1994.

L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real NVP. *CoRR*, abs/1605.08803, 2016.

X. Gastaldi. Shake-shake regularization. *CoRR*, abs/1705.07485, 2017.

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

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *International Conference on Machine Learning (ICML)*, 2013.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*, pages 237–243. IEEE Press, 2001.

G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. *CoRR*, abs/1603.09382, 2016.

A. Hyvärinen and E. Oja. Independent component analysis: Algorithms and applications. *Neural Networks*, 13(4-5):411–430, 2000.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015.

A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 2012.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

Y. leCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

D. Mishkin and J. Matas. All you need is a good init. *CoRR*, abs/1511.06422, 2015.

W. Shang, K. Sohn, D. Almeida, and H. Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. *CoRR*, abs/1603.05201, 2016.

S. Shi, Q. Wang, P. Xu, and X. Chu. Benchmarking state-of-the-art deep learning software tools. *CoRR*, abs/1608.07249, 2016.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958, 2014.

R. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

C. Szegedy, S. Ioffe, and V. Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

M. Telgarsky. Representation benefits of deep feedforward networks. *CoRR*, abs/1509.08101, 2015.

M. Telgarsky. Benefits of depth in neural networks. *CoRR*, abs/1602.04485, 2016.

J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler. Efficient object localization using convolutional networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

A. Veit, M. Wilber, and S. Belongie. Residual networks behave like ensembles of relatively shallow networks. *CoRR*, abs/1605.06431, 2016.

L. Wan, M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus. Regularization of neural network using dropconnect. In *International Conference on Machine Learning (ICML)*, 2013.