IfI Summer School 2018 on Machine Learning

Deep Learning #5 – Going deeper

François Fleuret
http://fleuret.org/ifi/
June 26, 2018

Batch processing

PyTorch's `Module`s take as input a batch of samples, that is a tensor whose first index is the sample's index.

We get with row vectors, for the full batch of a fully connected layer

$$x^{(l)} = x^{(l-1)} \left(w^{(l)}\right)^T,$$

and for the backward pass

$$\left[\!\left[\frac{\partial \mathscr{L}}{\partial w^{(l)}} \right]\!\right] = \left[\!\left[\frac{\partial \mathscr{L}}{\partial x^{(l)}} \right]\!\right]^T x^{(l-1)},$$

and

$$\left[\!\left[\frac{\partial \mathscr{L}}{\partial x^{(l)}} \right]\!\right] = \left[\!\left[\frac{\partial \ell}{\partial x^{(l+1)}} \right]\!\right] w^{(l+1)}.$$

Batch processing allows to use efficient highly parallel matrix product implementations, which in particular deal properly with cache memory.

```
import torch, time

def timing(x, w, nb = 51):
    t = torch.zeros(nb)

    for u in range(0, t.size(0)):
        t0 = time.perf_counter()
        y = x.mm(w.t())
        y.is_cuda and torch.cuda.synchronize()
        t[u] = time.perf_counter() - t0
    tb = t.median().item()

    for u in range(0, t.size(0)):
        t0 = time.perf_counter()
        for k in range(0, y.size(0)): y[k] = w.mv(x[k])
        y.is_cuda and torch.cuda.synchronize()
        t[u] = time.perf_counter() - t0
    tl = t.median().item()

    print('{:s} batch vs. loop speed ratio {:.01f}'
          .format((y.is_cuda and 'GPU') or 'CPU', tl / tb))

x = torch.empty(2500, 1000).normal_()
w = torch.empty(1500, 1000).normal_()
timing(x, w)

x = torch.empty(2500, 1000, device = torch.device('cuda:0')).normal_()
w = torch.empty(1500, 1000, device = torch.device('cuda:0')).normal_()
timing(x, w)
```

Prints:

```
CPU batch vs. loop speed ratio 5.1
GPU batch vs. loop speed ratio 76.0
```

Stochastic gradient descent

So far, to minimize a loss of the form

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

we have considered the gradient-descent algorithm

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

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

- it takes time to compute (more exactly **all our time!**),
- it is computed incrementally

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

- it is an empirical estimation of an hidden quantity, and any partial sum would similarly be an unbiased empirical estimate, although more noisy.

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

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

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

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

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

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

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

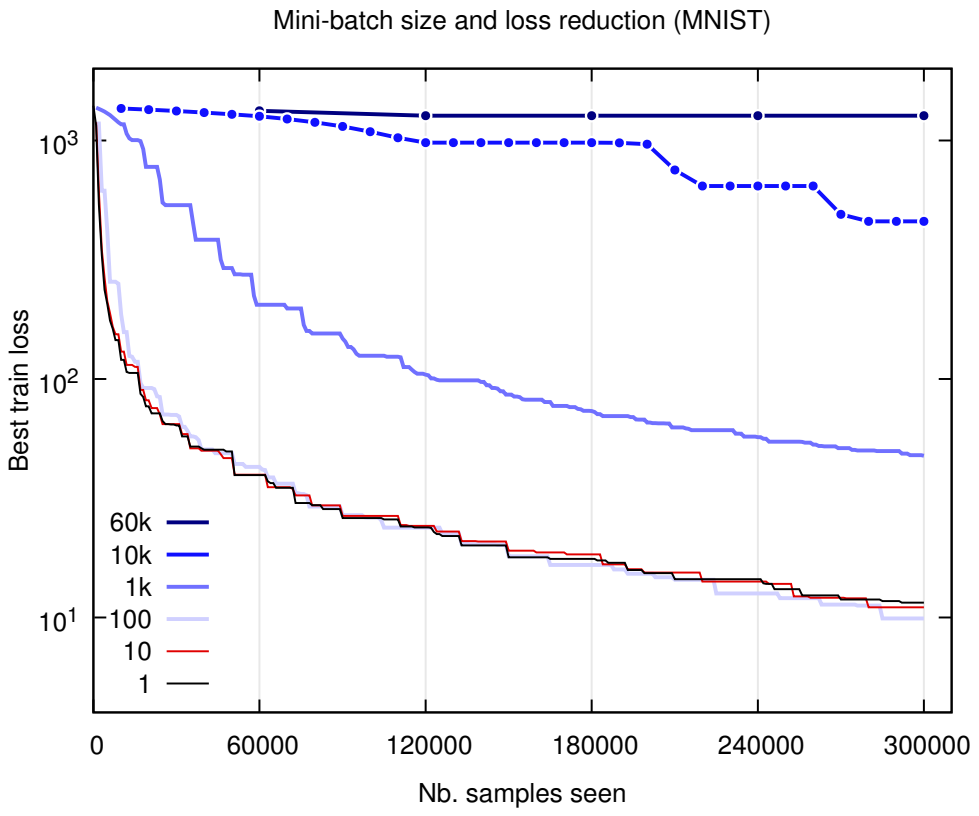
However this does not benefit from the speed-up of batch-processing.

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

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

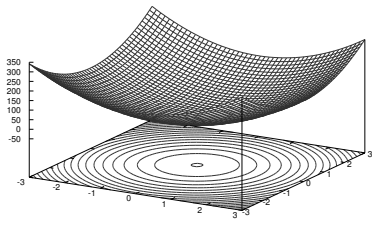
The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

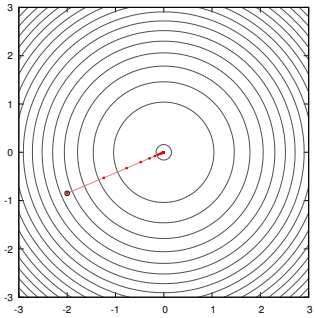
The stochastic behavior of this procedure helps evade local minima.



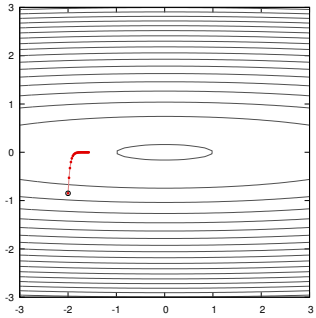
Limitation of the gradient descent

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

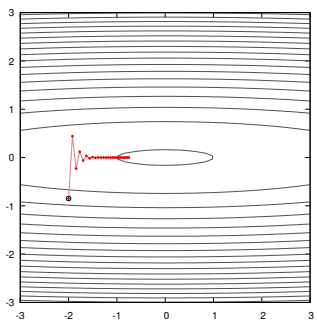




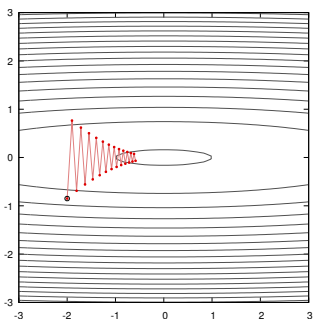
$\eta = 1.0e-2$



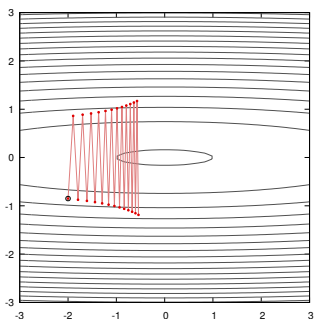
$\eta = 1.0e-2$



$\eta = 4.0e-2$



$\eta = 5.3e-2$



$\eta = 5.3e-2$

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

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

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

Momentum and moment estimation

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

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

where

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

is the gradient summed over a mini-batch.

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

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

(Rumelhart et al., 1986)

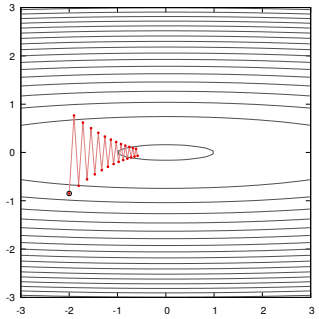
With $\gamma = 0$, this is the same as vanilla SGD.

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

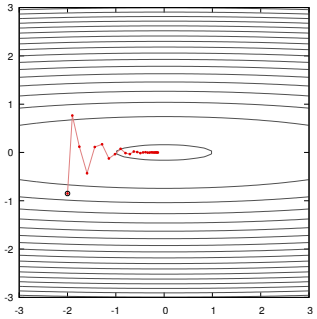
- it can "go through" local barriers,
- it accelerates if the gradient does not change much:

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

- it dampens oscillations in narrow valleys.



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



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

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

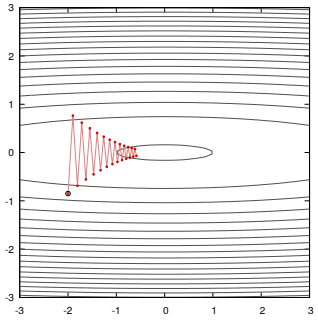
The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

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

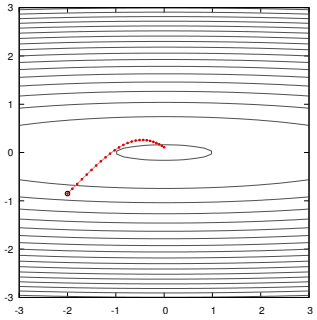
$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

(Kingma and Ba, 2014)

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



$\eta = 5.0e-2$



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

These two core strategies have been used in multiple incarnations:

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

`torch.optim`

PyTorch includes the standard variants of the stochastic gradient descent.

We had the following *ad hoc* implementation of SGD

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

It can be re-written as follows with the `torch.optim` package

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

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

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

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

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

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

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

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

into

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

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

Full example

We now have the tools to define a deep network:

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

And we have the tools to optimize it:

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

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

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

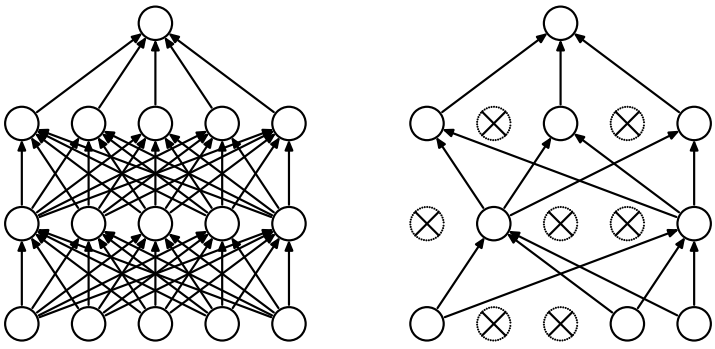
```
from torch import cuda, nn, optim
import torch, torchvision, time
from torch import nn
from torch.nn import functional as F

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

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



```
train_set = torchvision.datasets.MNIST('./data/mnist/', train = True, download = True)
train_input = train_set.train_data.view(-1, 1, 28, 28).float()
train_target = train_set.train_labels
nb_train_samples = train_input.size(0)

model = Net()

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

optimizer = torch.optim.SGD(model.parameters(), lr = lr)
criterion, bs = nn.CrossEntropyLoss(), 100

model.cuda()
criterion.cuda()
train_input, train_target = train_input.cuda(), train_target.cuda()

for e in range(10):
    for b in range(0, nb_train_samples, bs):
        output = model(train_input.narrow(0, b, bs))
        loss = criterion(output, train_target.narrow(0, b, bs))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

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.

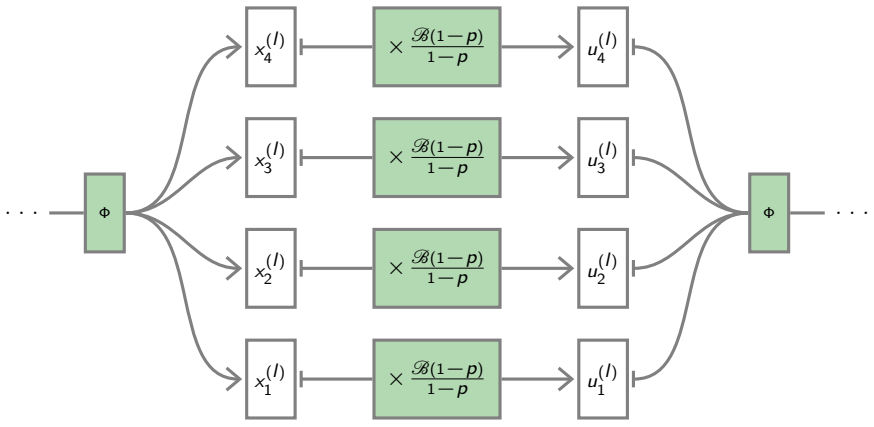
(Srivastava et al., 2014)

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 objective of dropout is to reduce the joint dependence between units and induce a "ensemble" effect. The training procedure can be seen as training a very large set of trees with weight sharing.

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 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 = torch.full((3, 6), 1.0).requires_grad_()
>>> x
tensor([[ 1.,  1.,  1.,  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.,  4.,  0.,  0.,  4.,  4.],
        [ 4.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  4.,  4.,  0.,  0.,  0.]])
>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad
tensor([[ 0.0000,  2.3094,  0.0000,  0.0000,  2.3094,  2.3094],
        [ 4.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  2.8284,  2.8284,  0.0000,  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
```

Batch normalization

Maintaining proper statistics of the activations and derivatives is a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization methods.

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.

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 $x_1, \ldots, x_B$ are the samples in the batch

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

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.

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

```
>>> x = torch.empty(1000, 3).normal_()
>>> x = x * torch.tensor([2., 5., 10.]) + torch.tensor([-10., 25., 3.])
>>> x.mean(0)
tensor([ -9.9108,  25.1234,   3.4366])
>>> x.std(0)
tensor([  2.0033,   4.9148,  10.0001])
>>> bn = nn.BatchNorm1d(3)
>>> bn.bias = Parameter(Tensor([2, 4, 8]))
>>> bn.weight = Parameter(Tensor([1, 2, 3]))
>>> y = bn(x)
>>> y.mean(0)
tensor([ 2.0000,  4.0000,  8.0000])
>>> y.std(0)
tensor([ 1.0005,  2.0010,  3.0015])
```


Results on ImageNet's LSVRC2012:

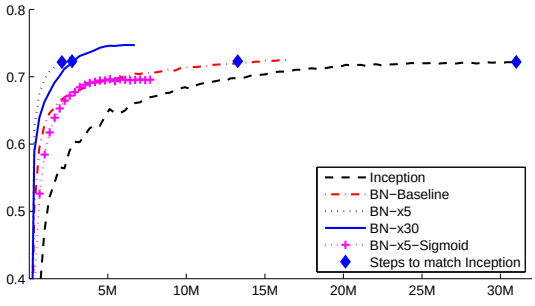


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

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.*

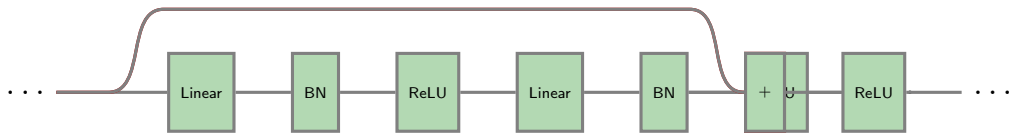
(Ioffe and Szegedy, 2015)

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.

Residual networks

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



⚠ This is possible only if the activation dimension(s) is unchanged.

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.

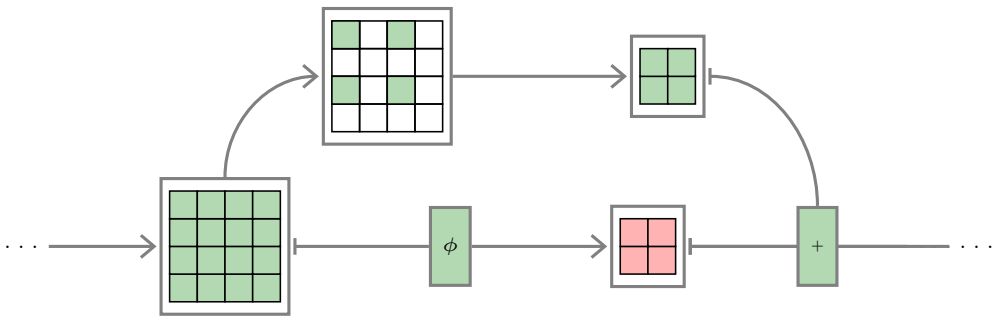
Also, the network initialization is around the identity.

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

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),

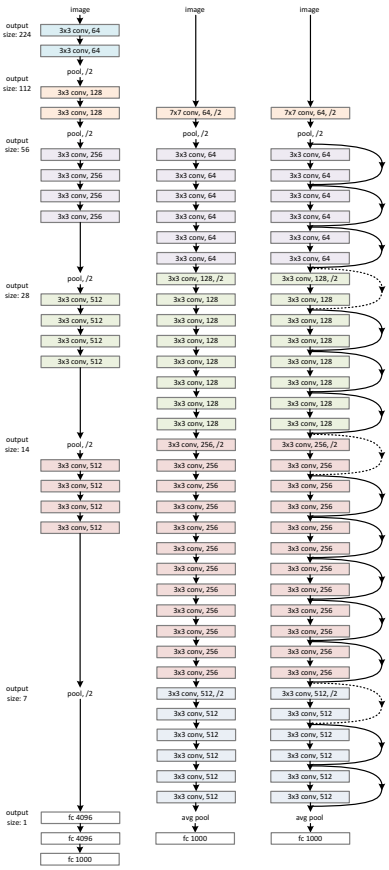


To increase the number of channels from C to C', they propose to either:

- 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.



(He et al., 2015)

Performance on ImageNet.

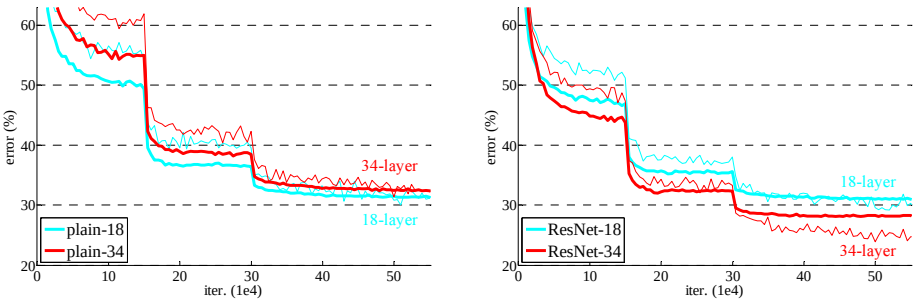


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

(He et al., 2015)

torch.utils.data.DataLoader

Until now, we have dealt with image sets that could fit in memory, and we manipulated them as regular tensors:

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

Large sets do not fit in memory, and samples have to be constantly loaded during training.

This require a [sophisticated] machinery to parallelize the loading itself, but also the normalization, and data-augmentation operations.

PyTorch offers the `torch.utils.data.DataLoader` object which combines a data-set and a sampling policy to create an iterator over mini-batches.

Standard data-sets are available in `torchvision.datasets`, and they allow to apply transformations over the images or the labels transparently.


```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, utils

train_transforms = transforms.Compose(
    [
        transforms.RandomCrop(28, padding = 3),
        transforms.ToTensor(),
        transforms.Normalize(mean = (33.32, ), std = (78.56, ))
    ]
)

train_loader = DataLoader(
    datasets.MNIST(root = './data/mnist', train = True, download = True,
                   transform = train_transforms),
    batch_size = 100,
    num_workers = 4,
    shuffle = True,
    pin_memory = torch.cuda.is_available()
)
```

Given this `train_loader`, we can now re-write our training procedure with a loop over the mini-batches

```
for e in range(nb_epochs):
    for input, target in iter(train_loader):

        if torch.cuda.is_available():
            input, target = input.cuda(), target.cuda()

        output = model(input)
        loss = criterion(output, target)
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

Note that for data-sets that can fit in memory this is quite inefficient, as they are constantly moved from the CPU to the GPU memory.

Example of neuro-surgery and fine-tuning in PyTorch

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,
- several resnet blocks, stored in a `nn.ModuleList` and each combining `nn.Conv2d`, `nn.BatchNorm2d`, and `nn.ReLU`,
- a final channel-wise averaging, using `nn.AvgPool2d`, and
- a final fully connected linear layer `nn.Linear`.

During training, we keep the AlexNet features frozen for a few epochs. This is done by setting `requires_grad` of the related `Parameter`s to `False`.


```
data_dir = os.environ.get('PYTORCH_DATA_DIR') or '.'

num_workers = 4
batch_size = 64

transform = torchvision.transforms.ToTensor()

train_set = torchvision.datasets.CIFAR10(root = data_dir, train = True,
                                         download = False, transform = transform)

train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size,
                                           shuffle = True, num_workers = num_workers)

test_set = torchvision.datasets.CIFAR10(root = data_dir, train = False,
                                        download = False, transform = transform)

test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size,
                                          shuffle = False, num_workers = num_workers)
```

```
def make_resnet_block(nb_channels, kernel_size = 3):

    return nn.Sequential(

        nn.Conv2d(nb_channels, nb_channels,
                  kernel_size = kernel_size,
                  padding = (kernel_size - 1) // 2),

        nn.BatchNorm2d(nb_channels),

        nn.ReLU(inplace = True),

        nn.Conv2d(nb_channels, nb_channels,
                  kernel_size = kernel_size,
                  padding = (kernel_size - 1) // 2),

        nn.BatchNorm2d(nb_channels),

    )
```



```
class Monster(nn.Module):
    def __init__(self, nb_residual_blocks, nb_channels):
        super(Monster, self).__init__()

        nb_alexnet_channels = 64
        alexnet_feature_map_size = 7 # For 32x32 (e.g. CIFAR)

        alexnet = torchvision.models.alexnet(pretrained = True)

        # Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
        self.features = nn.Sequential(
            alexnet.features[0],
            nn.ReLU(inplace = True)
        )

        self.converter = nn.Sequential(
            nn.Conv2d(nb_alexnet_channels, nb_channels,
                      kernel_size = 3, padding = 1),
            nn.ReLU(inplace = True)
        )

        self.resnet_blocks = nn.ModuleList()
        for k in range(nb_residual_blocks):
            self.resnet_blocks.append(make_resnet_block(nb_channels, 3))

        self.final_average = nn.AvgPool2d(alexnet_feature_map_size)
        self.fc = nn.Linear(nb_channels, 10)
```

```
    def freeze_features(self, q):
        for p in self.features.parameters():
            # If frozen (q == True) we do NOT need the gradient
            p.requires_grad = not q
```

```
    def forward(self, x):
        x = self.features(x)
        x = self.converter(x)
        for b in self.resnet_blocks:
            x = x + b(x)
        x = self.final_average(x).view(x.size(0), -1)
        x = self.fc(x)
        return x
```



```
nb_epochs = 100
nb_epochs_frozen_features = nb_epochs // 2
nb_residual_blocks = 16
nb_channels = 64

model, criterion = Monster(nb_residual_blocks, nb_channels), nn.CrossEntropyLoss()

if torch.cuda.is_available():
    model.cuda()
    criterion.cuda()

optimizer = optim.SGD(model.parameters(), lr = 1e-2)

model.train(True)

for e in range(nb_epochs):
    model.freeze_features(e < nb_epochs_frozen_features)

    acc_loss = 0.0

    for input, target in iter(train_loader):

        if torch.cuda.is_available():
            input, target = input.cuda(), target.cuda()

        output = model(input)
        loss = criterion(output, target)
        acc_loss += loss.item()

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

    print(e, acc_loss)
```

```
nb_test_errors, nb_test_samples = 0, 0

model.train(False)

for input, target in iter(test_loader):

    if torch.cuda.is_available():
        input = input.cuda()
        target = target.cuda()

    output = model(input)
    wta = torch.max(output, 1)[1].view(-1)

    for i in range(target.size(0)):
        nb_test_samples += 1
        if wta[i] != target[i]: nb_test_errors += 1

print('test_error {:.02f}% ({:d}/{:d})'.format(
    100 * nb_test_errors / nb_test_samples,
    nb_test_errors,
    nb_test_samples)
)
```


References

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

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

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

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

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

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.