Deep Learning

Training pipeline, optimization and image analysis (CNNs)

Lessons: **Kevin Scaman**
TPs: Paul Lerner

Class overview

Lessons

1. Introduction, simple architectures (MLPs) and autodiff 09/02
2. **Training pipeline, optimization and image analysis (CNNs)** 16/02
3. Sequence regression (RNNs), stability and robustness 08/03
4. Generative models in vision and text (Transformers, GANs) 15/03

Practicals

- **TP1:** MLPs and CNNs in Pytorch 01/03
- **TP2:** RNNs and generative models 22/03

Projects

Overview

- Teams of **up to 4 students**
- DL task with **Pytorch implementation**
- You can chose your topics from:
  `https://kscaman.github.io/teaching/2023_ENSAE_DL.html`
- Can also propose one, but better to start with an existing library

Projects

Overview

- Teams of **up to 4 students**
- DL task with **Pytorch implementation**
- You can chose your topics from:
  `https://kscaman.github.io/teaching/2023_ENSAE_DL.html`
- Can also propose one, but better to start with an existing library

Deadlines

- **(01/03) Team formation and topics:** Send email (`kevin.scaman@inria.fr`) with team and topic (link to repo + short description)
- **(29/03) Deliverables:** Report (pdf) + code (link to a colab/git repo)

Projects

Guidelines and tips

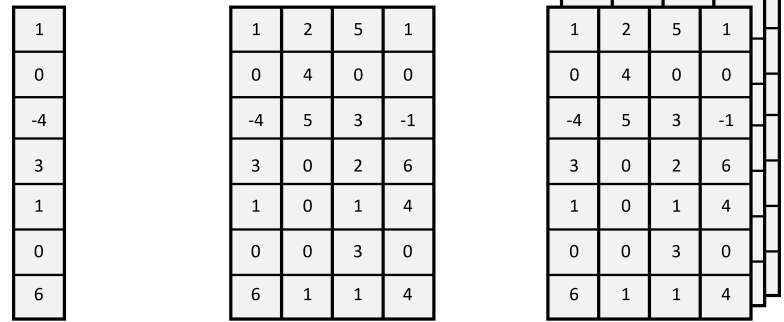
- Usually better to start with an **existing library**.
- Re-obtaining the results and implementing **one** alternative method / adaptation to another setting / use on an application is sufficient.
- Usually, the first try doesn't work... **Investigate why!**
- We're not looking for SOTA performance... **hard work is more valued**. :)
- Don't start too late... **debugging takes time**.

Pytorch tensors

The building blocks of DL implementations

Pytorch tensors

- A tensor is a $d-$**dimensional array** in Pytorch.
- Can store **real values**, **vectors**, **matrices**...
- Made to mimic **Numpy arrays**.



Some remarks on tensors (1)

1. **Tensor creation:**
   - We can create a tensor with "`x = torch.Tensor([[1,0,2],[3,2,3]])`".
   - We can clone a tensor with "`x.clone()`".
   - Tensors have a data type, e.g. "`x = torch.Tensor(..., dtype=torch.int64)`".

Some remarks on tensors (1)

1. **Tensor creation:**
   - We can create a tensor with "`x = torch.Tensor([[1,0,2],[3,2,3]])`".
   - We can clone a tensor with "`x.clone()`".
   - Tensors have a data type, e.g. "`x = torch.Tensor(..., dtype=torch.int64)`".
2. **Coordinate-wise operations:** "`x * y`" (needs matching sizes), "`torch.exp(x)`", "`x**2`",...

Some remarks on tensors (1)

1. **Tensor creation:**
   - We can create a tensor with "`x = torch.Tensor([[1,0,2],[3,2,3]])`".
   - We can clone a tensor with "`x.clone()`".
   - Tensors have a data type, e.g. "`x = torch.Tensor(..., dtype=torch.int64)`".
2. **Coordinate-wise operations:** "`x * y`" (needs matching sizes), "`torch.exp(x)`", "`x**2`",...
3. **Matrix multiplication:** "`x @ y`".

Some remarks on tensors (1)

1. **Tensor creation:**
   - We can create a tensor with "`x = torch.Tensor([[1,0,2],[3,2,3]])`".
   - We can clone a tensor with "`x.clone()`".
   - Tensors have a data type, e.g. "`x = torch.Tensor(..., dtype=torch.int64)`".
2. **Coordinate-wise operations:** "`x * y`" (needs matching sizes), "`torch.exp(x)`", "`x**2`",...
3. **Matrix multiplication:** "`x @ y`".
4. **Reshaping:** "`x.view(1,3,-1)`" or "`x.unsqueeze(0)`" to add a dimension of size 1.

Some remarks on tensors (1)

1. **Tensor creation:**
   - We can create a tensor with "`x = torch.Tensor([[1,0,2],[3,2,3]])`".
   - We can clone a tensor with "`x.clone()`".
   - Tensors have a data type, e.g. "`x = torch.Tensor(..., dtype=torch.int64)`".
2. **Coordinate-wise operations:** "`x * y`" (needs matching sizes), "`torch.exp(x)`", "`x**2`",...
3. **Matrix multiplication:** "`x @ y`".
4. **Reshaping:** "`x.view(1,3,-1)`" or "`x.unsqueeze(0)`" to add a dimension of size 1.
5. **Other operations:** See doc ☺. "`torch.sum(x)`", "`torch.mean(x)`"...

Some remarks on tensors (2)

1. The first dimension is usually the samples ("`x.shape[0]`" is the batch size)

Some remarks on tensors (2)

1. The first dimension is usually the samples ("`x.shape[0]`" is the batch size)
2. **Gradients:**
   - Tensors can have a gradient in "`x.grad`".
   - We can remove (and clone) this tensor from the computation graph by using "`y = x.detach()`".

Some remarks on tensors (2)

1. The first dimension is usually the samples ("`x.shape[0]`" is the batch size)
2. **Gradients:**
   - Tensors can have a gradient in "`x.grad`".
   - We can remove (and clone) this tensor from the computation graph by using "`y = x.detach()`".
3. **To NumPy:** "`x.numpy()`" or "`x.detach().numpy()`".

Some remarks on tensors (2)

1. The first dimension is usually the samples ("`x.shape[0]`" is the batch size)
2. **Gradients:**
   - Tensors can have a gradient in "`x.grad`".
   - We can remove (and clone) this tensor from the computation graph by using "`y = x.detach()`".
3. **To NumPy:** "`x.numpy()`" or "`x.detach().numpy()`".
4. **Debug:**
   - Many errors can be unnoticed due to wrong tensor sizes and Python's dynamic typing...
   - Always verify your intermediate computations with e.g. "`print(x[:5])`".
   - Always verify your tensor shapes with "`print(x.shape)`"!

Some remarks on tensors (2)

1. The first dimension is usually the samples ("`x.shape[0]`" is the batch size)
2. **Gradients:**
   - Tensors can have a gradient in "`x.grad`".
   - We can remove (and clone) this tensor from the computation graph by using "`y = x.detach()`".
3. **To NumPy:** "`x.numpy()`" or "`x.detach().numpy()`".
4. **Debug:**
   - Many errors can be unnoticed due to wrong tensor sizes and Python's dynamic typing...
   - Always verify your intermediate computations with e.g. "`print(x[:5])`".
   - Always verify your tensor shapes with "`print(x.shape)`"!

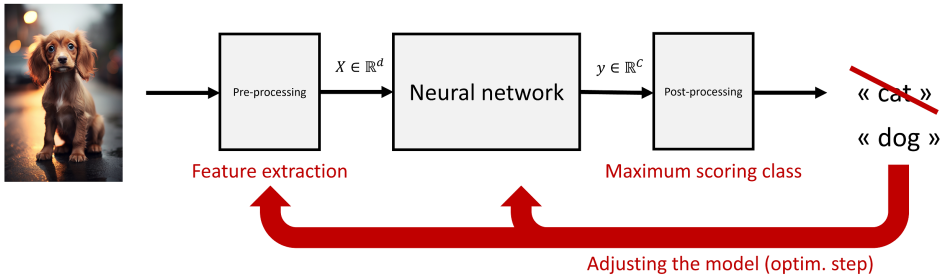
A tensor of shape (5,1) is not the same a tensor of shape (5)!

Deep learning training pipeline

Training and testing neural networks (in Pytorch)

Back to cats and dogs

Typical binary classification task. Objective is to distinguish **cat images from dog images**.



Pytorch training pipeline

Data loader → **model creation** → **loss function** → **optimization loop**

Dataloading

Dataset class

`torch.utils.data.Dataset` is an abstract class representing a dataset. Your custom dataset should inherit `Dataset` and override the following methods:

- `__len__` so that `len(dataset)` returns the size of the dataset.
- `__getitem__` to support the indexing such that `dataset[i]` gives the ith sample.

Iterating through the dataset with Dataloader

By using a simple for loop to iterate over the data, we are missing out on:

- Batching the data,
- Shuffling the data,
- Load the data in parallel using multiprocessing workers.

`torch.utils.data.DataLoader` is an iterator which provides all these features.

Dataloading (advanced)

Transformations

- `torchvision.transforms` allows to easily compose data transformations to the data.
- `img_transform = transforms.Compose([transforms.CenterCrop(224), transforms.ToTensor(), transforms.Normalize(mean, std)])`
- On most vision dataset, the `transform` field is applied before accessing the data.

Dataloading (advanced)

Transformations

- `torchvision.transforms` allows to easily compose data transformations to the data.
- `img_transform = transforms.Compose([transforms.CenterCrop(224), transforms.ToTensor(), transforms.Normalize(mean, std)])`
- On most vision dataset, the `transform` field is applied before accessing the data.

Dataloader

- To create a dataloader for the training set, use `torch.utils.data.DataLoader`:

  `loader = DataLoader(dataset, batch_size=64, shuffle=True, num_workers=6)`
- Includes a **random mini-batch** selection mechanism and **parallelization**.
- Used as an **iterator**: `for inputs, targets in train_loader: ...`

Model creation

Sequential neural networks and MLPs

- One liner for MLPs: `model = nn.Sequential(nn.Linear(2,4), nn.ReLU(),...)`
- More generally any **sequence** of already existing layers.
- How to create **new layers** or entirely new architectures?

The Module class

- All models are exentions of the `nn.Module` class.
- Need to implement a `model.forward(x)` function.
- Can be called as a function `model(x)`.

Neural networks in Pytorch

The Module class

- All models are exentions of the `nn.Module` class.
- Need to implement a `model.forward(x)` function.
- Can be called as a function `model(x)`.

Neural networks in Pytorch

The Module class

- All models are exentions of the `nn.Module` class.
- Need to implement a `model.forward(x)` function.
- Can be called as a function `model(x)`.

Example (function of two variables)

```
class YourModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = nn.Linear(10, 100)

    def forward(self, x, y)
        x = self.lin(x)
        return y + torch.exp(torch.mean(x, dim=1))
```

Neural networks in Pytorch (2)

- A function `backward` is automatically implemented to perform **backpropagation**.

Neural networks in Pytorch (2)

- A function `backward` is automatically implemented to perform **backpropagation**.
- By default, the parameters `model.parameters()` are **randomly initialized**.

Neural networks in Pytorch (2)

- A function `backward` is automatically implemented to perform **backpropagation**.
- By default, the parameters `model.parameters()` are **randomly initialized**.
- **Hierarchical structure:** All layers also extend `nn.Module`, and any module can be used in another module. All modules used by a model are accessible via `model.children()`.

Loss functions (recap)

Empirical risk minimization

Let $(x_i, y_i)_{i \in [\![1,n]\!]}$ be a collection of n observations drawn independently according to $\mathcal{D}$. Then, the objective of *empirical risk minimization* (ERM) is to find a minimizer $\hat{\theta}_n \in \mathbb{R}^p$ of

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(g_\theta(x_i), y_i)$$

Loss functions (recap)

Empirical risk minimization

Let $(x_i, y_i)_{i \in [\![1,n]\!]}$ be a collection of n observations drawn independently according to $\mathcal{D}$. Then, the objective of *empirical risk minimization* (ERM) is to find a minimizer $\hat{\theta}_n \in \mathbb{R}^p$ of

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(g_\theta(x_i), y_i)$$

Losses used for training

- **Regression:** Mean sqaure error (MSE) $\ell(y, y') = \|y - y'\|_2^2 = \sum_i (y_i - y'_i)^2$
- **Classification:** Cross entropy (CE) $\ell(y, y') = -\sum_i y'_i \ln\left(\exp(y_i)/\sum_j \exp(y_j)\right)$

Cross entropy

- **Intuition:** The model outputs a score for each class $y_i = g_\theta(x)$. We create a probability on the classes $p_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)}$. We then take the negative logarithm of the true class's probability $\ell(y, y') = -\log(p_k)$, where $k \in [\![1, C]\!]$ s.t. $y'_i = \mathbb{1}\{i = k\}$.

Cross entropy

- **Intuition:** The model outputs a score for each class $y_i = g_\theta(x)$. We create a probability on the classes $p_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)}$. We then take the negative logarithm of the true class's probability $\ell(y, y') = -\log(p_k)$, where $k \in [\![1, C]\!]$ s.t. $y'_i = \mathbb{1}\{i = k\}$.
- **Interpretation #1:** Minimizing cross entropy is equivalent to **maximum likelihood estimation** for the probabilistic model of the data samples (X_i, Y_i) such that $\log \mathbb{P}(Y_i = k \mid X_i) \propto g_\theta(X_i)_k$ where X_i are i.i.d. and independent of θ, as

$$\mathbb{P}_\theta((X_i, Y_i)) \;=\; \prod_i \mathbb{P}(X_i)\mathbb{P}_\theta(Y_i \mid X_i) \propto \prod_i \frac{\exp(g_\theta(X_i)_{Y_i})}{\sum_k \exp(g_\theta(X_i)_k)}$$

Cross entropy

- **Intuition:** The model outputs a score for each class $y_i = g_\theta(x)$. We create a probability on the classes $p_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)}$. We then take the negative logarithm of the true class's probability $\ell(y, y') = -\log(p_k)$, where $k \in [\![1, C]\!]$ s.t. $y'_i = \mathbb{1}\{i = k\}$.
- **Interpretation #1:** Minimizing cross entropy is equivalent to **maximum likelihood estimation** for the probabilistic model of the data samples (X_i, Y_i) such that $\log \mathbb{P}(Y_i = k \mid X_i) \propto g_\theta(X_i)_k$ where X_i are i.i.d. and independent of θ, as

$$\mathbb{P}_\theta((X_i, Y_i)) \;=\; \prod_i \mathbb{P}(X_i)\mathbb{P}_\theta(Y_i \mid X_i) \propto \prod_i \frac{\exp(g_\theta(X_i)_{Y_i})}{\sum_k \exp(g_\theta(X_i)_k)}$$

- **Interpretation #2:** Difference between the scores of the predicted and true classes.

$$\ell(y, y') = \log\left(\sum_i \exp(y_i)\right) - y_k \approx \max_i y_i - y_k$$

Cross entropy: in practice

- **Definition:** $\ell(x, y) = -\log\left(\frac{\exp(x_y)}{\sum_i \exp(x_i)}\right)$.
- **PyTorch:** `criterion = nn.CrossEntropyLoss()`
- Several parameters (`reduction='sum'` or `reduction='mean'`, see the doc)
- `criterion` takes as input the scores (a tensor of shape $[b, d]$), and either a class index per sample, or class probabilities for each sample.
- Composition of `nn.LogSoftmax()` and `nn.NLLLoss()`.

Gradient through a softmax can explode due to numerical errors (taken care of by the Pytorch implementation of `nn.CrossEntropyLoss()` and `nn.LogSoftmax()`).

First-order optimization

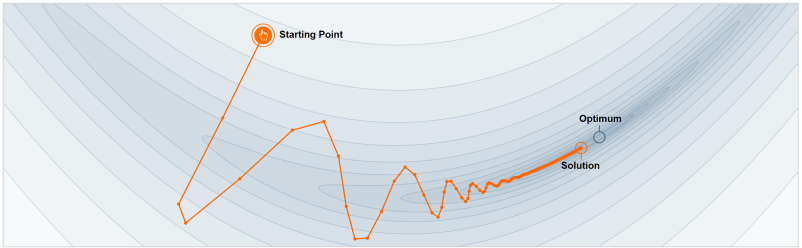
Gradient descent and co.

First-order optimization

- Find a **minimizer** $\theta^\star \in \mathbb{R}^d$ of a given objective function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$,

$$\theta^\star \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \mathcal{L}(\theta)$$

- Using an iterative algorithm relying on the **gradient** $\nabla \mathcal{L}(\theta_t)$ at each iteration $t \geqslant 0$.



source: https://distill.pub/2017/momentum/

First-order optimization

Iterative optimization algorithms

- **Initialization:** $\theta_0 \in \mathbb{R}^d$ (important in practice!).
- **Iteration:** Usually $\theta_{t+1} = \varphi_t\left(\theta_t, \nabla\mathcal{L}(\theta_t), s_t\right)$ where s_t is a hidden variable that is also updated at each iteration.
- **Stopping time:** $T > 0$ (also important in practice!).

First-order optimization

Iterative optimization algorithms

- **Initialization:** $\theta_0 \in \mathbb{R}^d$ (important in practice!).
- **Iteration:** Usually $\theta_{t+1} = \varphi_t\left(\theta_t, \nabla\mathcal{L}(\theta_t), s_t\right)$ where s_t is a hidden variable that is also updated at each iteration.
- **Stopping time:** $T > 0$ (also important in practice!).

Main difficulties in neural network training

First-order optimization

Iterative optimization algorithms

- **Initialization:** $\theta_0 \in \mathbb{R}^d$ (important in practice!).
- **Iteration:** Usually $\theta_{t+1} = \varphi_t\left(\theta_t, \nabla\mathcal{L}(\theta_t), s_t\right)$ where s_t is a hidden variable that is also updated at each iteration.
- **Stopping time:** $T > 0$ (also important in practice!).

Main difficulties in neural network training

- **Non-convexity:** If $\mathcal{L}$ is **convex**, i.e. $\forall\theta, \theta', \mathcal{L}(\frac{\theta+\theta'}{2}) \leqslant \frac{\mathcal{L}(\theta)+\mathcal{L}(\theta')}{2}$, the optimization problem is **simple**. Most theoretical results use this assumption to prove convergence.

First-order optimization

Iterative optimization algorithms

- **Initialization:** $\theta_0 \in \mathbb{R}^d$ (important in practice!).
- **Iteration:** Usually $\theta_{t+1} = \varphi_t\left(\theta_t, \nabla\mathcal{L}(\theta_t), s_t\right)$ where s_t is a hidden variable that is also updated at each iteration.
- **Stopping time:** $T > 0$ (also important in practice!).

Main difficulties in neural network training

- **Non-convexity:** If $\mathcal{L}$ is **convex**, i.e. $\forall\theta, \theta', \mathcal{L}(\frac{\theta+\theta'}{2}) \leqslant \frac{\mathcal{L}(\theta)+\mathcal{L}(\theta')}{2}$, the optimization problem is **simple**. Most theoretical results use this assumption to prove convergence.
- **High dimensionality:** number of parameters $d \gg 1000$.

First-order optimization

Iterative optimization algorithms

- **Initialization:** $\theta_0 \in \mathbb{R}^d$ (important in practice!).
- **Iteration:** Usually $\theta_{t+1} = \varphi_t\left(\theta_t, \nabla\mathcal{L}(\theta_t), s_t\right)$ where s_t is a hidden variable that is also updated at each iteration.
- **Stopping time:** $T > 0$ (also important in practice!).

Main difficulties in neural network training

- **Non-convexity:** If $\mathcal{L}$ is **convex**, i.e. $\forall\theta, \theta', \mathcal{L}(\frac{\theta+\theta'}{2}) \leqslant \frac{\mathcal{L}(\theta)+\mathcal{L}(\theta')}{2}$, the optimization problem is **simple**. Most theoretical results use this assumption to prove convergence.
- **High dimensionality:** number of parameters $d \gg 1000$.
- **Access to the gradient:** the gradient of $\mathcal{L}$ is too expensive to compute! In practice, $\nabla\mathcal{L}(\theta_t)$ is replaced by a **stochastic** or **mini-batch** approximation $\widetilde{\nabla}_t$.

Gradient descent variants

- Let $\mathcal{L}_i(\theta) = \ell(g_\theta(x_i), y_i)$. Recall **empirical risk minimization**, aka **training error**:

$$\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(\theta)$$

Gradient descent variants

- Let $\mathcal{L}_i(\theta) = \ell(g_\theta(x_i), y_i)$. Recall **empirical risk minimization**, aka **training error**:

$$\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(\theta)$$

- **Batch gradient descent:** uses the true gradient, learning rate (or step-size) $\eta > 0$,

$$\theta_{t+1} = \theta_t - \frac{\eta}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(\theta)$$

Gradient descent variants

- Let $\mathcal{L}_i(\theta) = \ell(g_\theta(x_i), y_i)$. Recall **empirical risk minimization**, aka **training error**:

$$\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(\theta)$$

- **Batch gradient descent:** uses the true gradient, learning rate (or step-size) $\eta > 0$,

$$\theta_{t+1} = \theta_t - \frac{\eta}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(\theta)$$

- **Stochastic gradient descent:** gradient approximated with one random sample.

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}_{i_t}(\theta)$$

Gradient descent variants

- Let $\mathcal{L}_i(\theta) = \ell(g_\theta(x_i), y_i)$. Recall **empirical risk minimization**, aka **training error**:

$$\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(\theta)$$

- **Batch gradient descent:** uses the true gradient, learning rate (or step-size) $\eta > 0$,

$$\theta_{t+1} = \theta_t - \frac{\eta}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(\theta)$$

- **Stochastic gradient descent:** gradient approximated with one random sample.

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}_{i_t}(\theta)$$

- **Mini-batch gradient descent:** gradient approximated with multiple random samples.

$$\theta_{t+1} = \theta_t - \frac{\eta}{b} \sum_{i=1}^{b} \nabla \mathcal{L}_{i_{b,t}}(\theta)$$

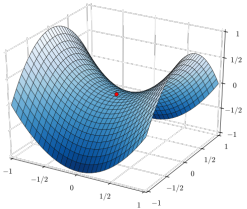
Some warnings about optimization in deep learning

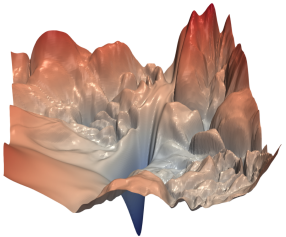
Our final goal is to reduce the **population risk**, i.e. $\mathbb{E}(\ell(g_\theta(X), Y))$!

- We need to pay attention to **overfitting** in addition to using the optimization algorithm to reduce the training error.
- In this class, we focus specifically on the **performance** of the optimization algorithm in minimizing the objective function, rather than the model's generalization error.
- In the next lessons, we will see techniques to avoid **overfitting**.

Challenges

- **Mini-batch gradient descent** is the algorithm of choice when training a neural network. The term SGD is usually employed also when mini-batches are used!
  - Choosing a learning rate can be difficult. How to **adapt the learning rate** during training?
  - Why applying the **same learning rate to all parameter updates**?
  - How to escape **saddle points** where the gradient is close to zero in all dimension?
- In the rest of the lecture, we will introduce modifications to (S)GD.
- Nice survey by Sebastian Ruder: `http://ruder.io/optimizing-gradient-descent/`





source: Visualizing the Loss Landscape of Neural Nets, Li et.al., 2018

Momentum

- Accelerating SGD by dampening oscillations, i.e. by averaging the last values of the latest gradients.

$$v_{t+1} = \gamma v_t + \eta \nabla \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

Momentum

- Accelerating SGD by dampening oscillations, i.e. by averaging the last values of the latest gradients.

$$v_{t+1} = \gamma v_t + \eta \nabla \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

- Why does it work? With $g_t = \nabla \mathcal{L}(\theta_t)$, we have for any $k \geqslant 0$:

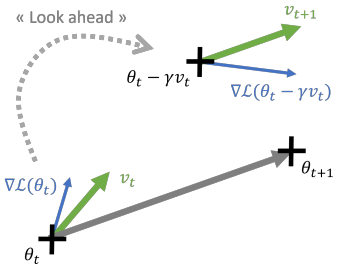
$$v_{t+1} = \gamma^k v_{t-k} + \eta \underbrace{\sum_{i=0}^{k} \gamma^i g_{t-i}}_{\text{average of last gradients}}$$

- Typical value for $\gamma = 0.9$.

Nesterov accelerated gradient

- **Idea:** A variant of momentum, in which we **look ahead** before computing the gradient.

$$v_{t+1} = \gamma v_t + \eta \nabla \mathcal{L}(\theta_t - \gamma v_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$



source: Nesterov, A method for solving the convex programming problem with convergence rate $O(1/k^2)$, Dokl. Akad. Nauk SSSR 1983

Adagrad

- We would like to adapt our updates to each individual parameter, i.e. have a different decreasing learning rate for each parameter.

$$s_{t+1,i} = s_{t,i} + \nabla \mathcal{L}(\theta_t)_i^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t+1,i} + \epsilon}} \nabla \mathcal{L}(\theta_t)_i$$

Adagrad

- We would like to adapt our updates to each individual parameter, i.e. have a different decreasing learning rate for each parameter.

$$s_{t+1,i} = s_{t,i} + \nabla \mathcal{L}(\theta_t)_i^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t+1,i} + \epsilon}} \nabla \mathcal{L}(\theta_t)_i$$

- No manual tuning of the learning rate.
- Typical default values: $\eta = 0.01$ and $\epsilon = 10^{-8}$.

source: Duchi et al., Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, JMLR 2011

RMSProp

- Problem with Adagrad, learning rate goes to zero and never forgets about the past.
- Idea proposed by G. Hinton in his Coursera class: use exponential average.

$$s_{t+1,i} = \gamma s_{t,i} + (1-\gamma)\nabla\mathcal{L}(\theta_t)_i^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t+1,i}+\epsilon}}\nabla\mathcal{L}(\theta_t)_i$$

RMSProp

- Problem with Adagrad, learning rate goes to zero and never forgets about the past.
- Idea proposed by G. Hinton in his Coursera class: use exponential average.

$$s_{t+1,i} = \gamma s_{t,i} + (1-\gamma)\nabla\mathcal{L}(\theta_t)_i^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t+1,i}+\epsilon}}\nabla\mathcal{L}(\theta_t)_i$$

- With a slight abuse of notation, we re-write the update as follows:

$$s_{t+1} = \gamma s_t + (1-\gamma)\nabla\mathcal{L}(\theta_t)^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_{t+1}+\epsilon}}\nabla\mathcal{L}(\theta_t)$$

- Typical values: $\gamma = 0.9$ and $\eta = 0.001$.

source: Hinton Coursera lecture 6

Adam

- Mixing RMSProp and momentum, we get Adam = Adaptive moment Estimation.

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\nabla\mathcal{L}(\theta_t) \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)\nabla\mathcal{L}(\theta_t)^2 \\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}} \\
\hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}}+\epsilon}\hat{m}_{t+1}
\end{aligned}
$$

- $\hat{m}_t$ and $\hat{v}_t$ are estimates for the first and second moments of the gradients. Because $m_0 = v_0 = 0$, these estimates are biased towards 0, the factors $(1-\beta^{t+1})^{-1}$ are here to counteract these biases.
- Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

source: Kingma et al. , Adam: a Method for Stochastic Optimization, ICLR 2015

AMSGrad

- Sometimes, Adam forgets too fast. To fix it, we replace the moving average by a $\max$:

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\nabla\mathcal{L}(\theta_t) \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)\nabla\mathcal{L}(\theta_t)^2 \\
\hat{v}_{t+1} &= \max\left(\hat{v}_t, v_{t+1}\right) \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} m_{t+1}
\end{aligned}
$$

source: Reddi et al.,On the Convergence of Adam and Beyond ICLR 2018

PyTorch optimizers

- All have similar constructor `torch.optim.*(params, lr=..., momentum=...)`. Default values are different for all optimizers, check the doc.
- `params` should be an iterable (like a list) containing the parameters to optimize over. It can be obtained from any module with `module.parameters()`.
- The `step` method updates the internal state of the optimizer according to the `grad` attributes of the `params`, and updates the latter according to the internal state.

Pytorch training loop

Training over one epoch becomes:

```
model = YourModel()
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for inputs, targets in train_loader:
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Last details

Faster parallel computations with GPUs

- To know if you have **access to GPUs**:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Using gpu: %s ' % torch.cuda.is_available())
```

- Tensors are allocated on a device using: `x.to(device)`.

Testing, metrics and more

- We need to create a **test set** separate from the training set to **evaluate** the model.
- We need to **store all loss values and accuracies** after each epoch.
- Set the model to `model.train()` or `model.eval()`.
- Perform multiple epochs.

Updated Pytorch training loop

The training pipeline becomes: (dataloader → model → loss → optimizer → visualization)

```
model.to(device)
model.train()
for epoch in range(num_epochs):
    running_loss = 0.
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch}: Loss: {running_loss/n_data:.2f}")
```

Pytorch test function

Test should not compute gradients, hence `torch.no_grad()`.

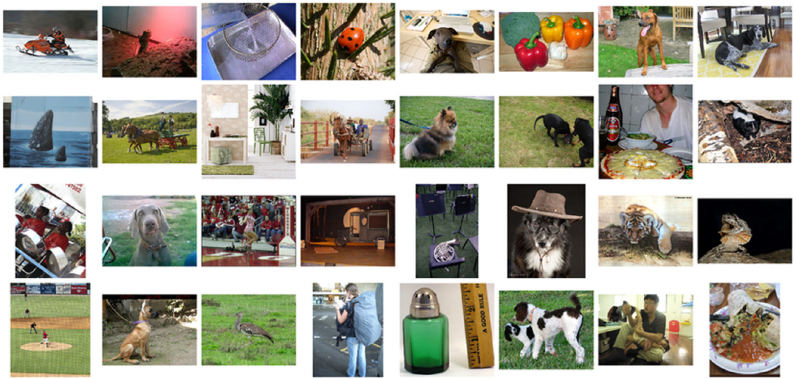
```
model.to(device)
model.eval()
running_loss, running_acc = 0., 0
with torch.no_grad():
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        output = model(inputs)
        loss = criterion(outputs, targets)
        preds = torch.argmax(outputs,1)
        running_loss += loss.item()
        running_acc += torch.sum(preds == targets)
    print(f"Loss: {running_loss/n_data:.2f} Acc: {running_acc/n_data:.2f}")
```

Image analysis

Introduction to convolutional neural networks

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- Object recognition challenge, from 2010 to 2017.
- 1.2 million images (avg. 469x387), 1000 object classes.



source: ImageNet Large Scale Visual Recognition Challenge. Russakovsky et.al., 2015.

Classification accuracy on Imagenet

Performance of classification methods (top-5 accuracy)

- **Random strategy:** 0.5%.
- **Human performance:** Expert 1: 94.9%. Expert 2: 88%.

Classification accuracy on Imagenet

Performance of classification methods (top-5 accuracy)

- **Random strategy:** 0.5%.
- **Human performance:** Expert 1: 94.9%. Expert 2: 88%.
- **Before 2012:** Feature extraction + SVMs, 74.2%.

Classification accuracy on Imagenet

Performance of classification methods (top-5 accuracy)

- **Random strategy:** 0.5%.
- **Human performance:** Expert 1: 94.9%. Expert 2: 88%.
- **Before 2012:** Feature extraction + SVMs, 74.2%.
- **Winners of 2012:** CNN (AlexNet) 84.7%.

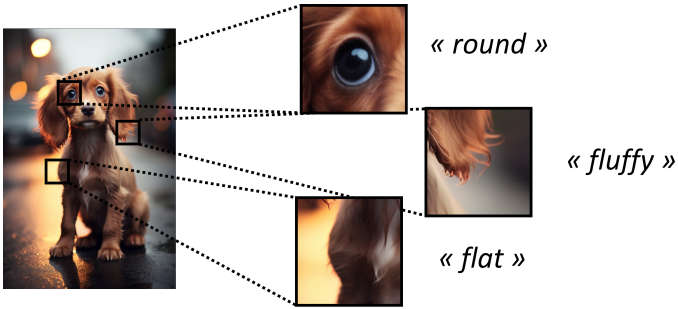
Classification accuracy on Imagenet

Performance of classification methods (top-5 accuracy)

- **Random strategy:** 0.5%.
- **Human performance:** Expert 1: 94.9%. Expert 2: 88%.
- **Before 2012:** Feature extraction + SVMs, 74.2%.
- **Winners of 2012:** CNN (AlexNet) 84.7%.
- **After 2012:** Always DL architectures, current best $\approx$ 99%.

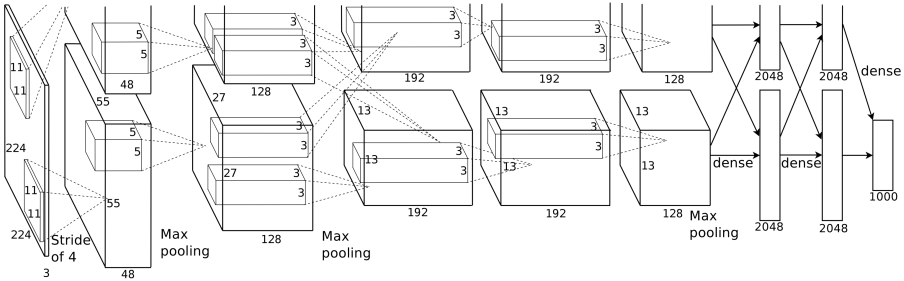
Current leaderboard: `https://paperswithcode.com/sota/image-classification-on-imagenet`

Encoding local information



- We want to find sharp edges, round eyes, fur-like textures...
- How can we encode these **local characteristics**?
- How can we ensure **translation invariance**?

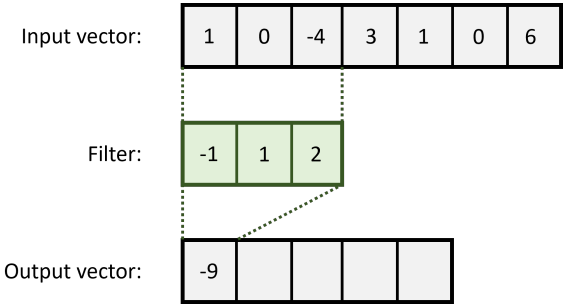
Convolutional Neural Networks



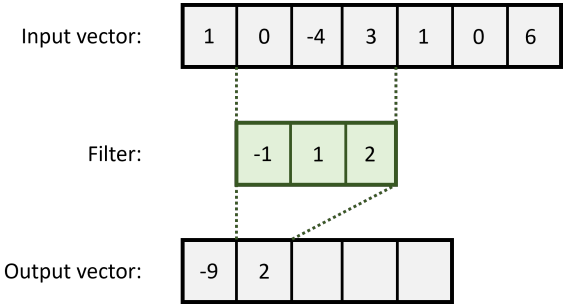
source: ImageNet Classification with Deep Convolutional Neural Networks. Krizhevsky et.al., 2012.

- First idea introduced by **Fukushima in 1980**.
- Linear layers in MLPs are replaced by **convolution** and **pooling** layers.
- Higher-level structures are extracted via a **hierarchical information processing**.

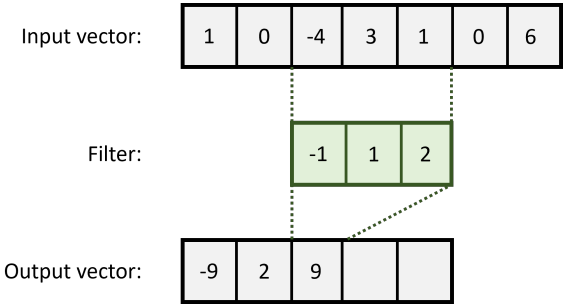
Convolutions (1D)



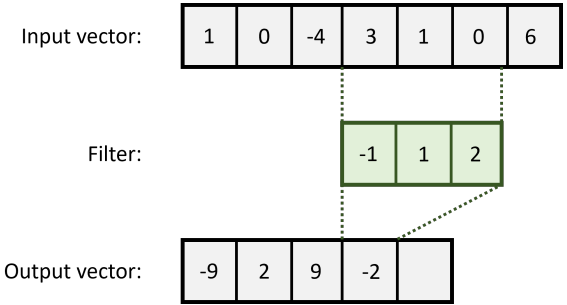
Convolutions (1D)



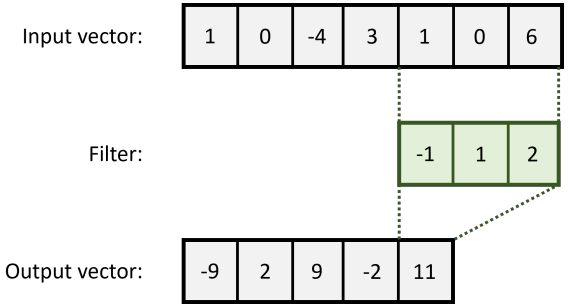
Convolutions (1D)



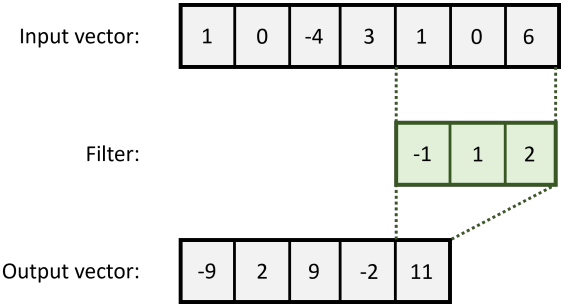
Convolutions (1D)



Convolutions (1D)

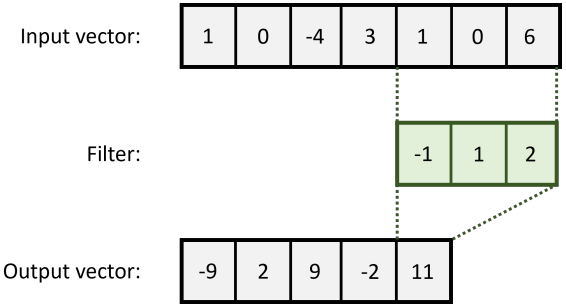


Convolutions (1D)



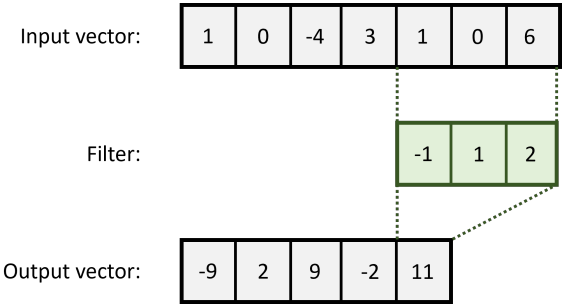
- **Continuous setting:** $(f * g)(u) = \int_{v=-\infty}^{+\infty} f(v)\, g(u-v)\, dv$

Convolutions (1D)



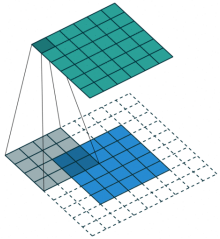
- **Continuous setting:** $(f * g)(u) = \int_{v=-\infty}^{+\infty} f(v)\, g(u-v)\, dv$
- **Discrete version:** $(x * y)_i = \sum_{j=1}^{m} x_j\, y_{i-j[n]}$

Convolutions (1D)



- **Continuous setting:** $(f * g)(u) = \int_{v=-\infty}^{+\infty} f(v)\, g(u-v)\, dv$
- **Discrete version:** $(x * y)_i = \sum_{j=1}^{m} x_j\, y_{i-j[n]}$
- **Pytorch implementation:** $(x * y)_i = \sum_j x_j\, y_{i+j}$ (technically, a cross-correlation)
- **Key properties:** Local operation, limited receptive field, translation equivariant.

Convolutions (2D)

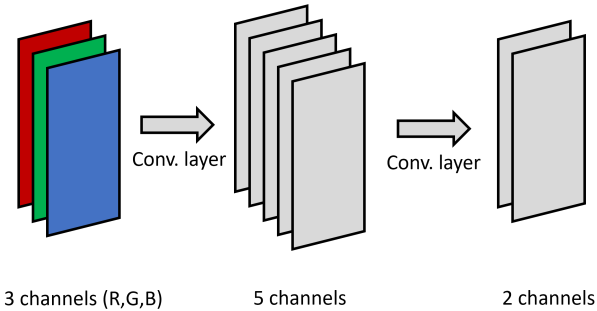


source: https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

Technical details

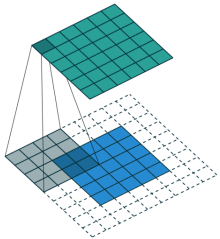
- **Receptive field:** shape of the filter (typically 3x3).
- **Padding:** Adding a boundary of $K > 0$ layers of **zeros** (increases output image size).
- **Stride:** do the computation for one pixel every $K > 0$ (decreases output image size).
- **See in action:** `https://setosa.io/ev/image-kernels/`

Convolution channels



- **Idea:** Allows to store multiple local information (e.g. vertical/horizontal edges, corners,...)
- **Definition:** Dense connections between channels, i.e. for each output channel k, $y_k = \sum_l W_{k,l} * x_l + b_k$ where $W_{k,l}$ is the filter for input channel l and output channel k.
- **Rule of thumb:** number of channels increases while image size decreases.

Pooling



source: https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

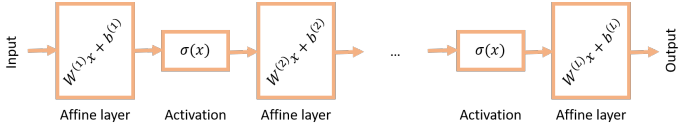
- **Main idea: Aggregate** local information to **reduce complexity**.
- **Example:** *Is there an edge in this region of the image?*
- **No parameters:** Applies a simple function to local image patches.
- **Two major variants: AvgPool** (mean over values) or **MaxPool** (max over values).

Example of a real-world CNN: VGG-16

source: https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

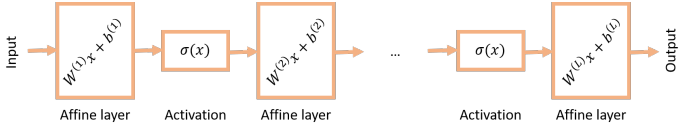
- **Features:** 13 layers of convolution and 5 layers of padding
- **Classifier:** Last layers are an MLP with 3 linear layers.
- First layers encode **low-level** information (e.g. edges or circles).
- Last layers encode **high-level** information. (e.g. "fluffiness" or "eye-shaped elements")
- **See in action:** `https://distill.pub/2017/feature-visualization/`

But... why convolutions?



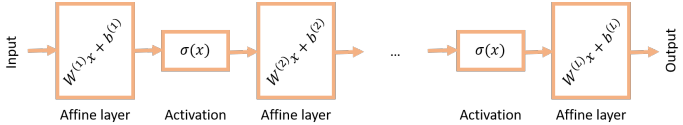
- **Idea:** why not take an MLP and make it **translation invariant**?

But... why convolutions?



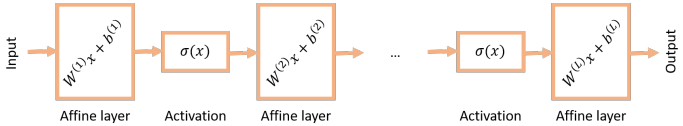
- **Idea:** why not take an MLP and make it **translation invariant**?
- **Equivariance:** a function f is equivariant w.r.t. to a transformation τ iff $f \circ \tau = \tau \circ f$.

But... why convolutions?



- **Idea:** why not take an MLP and make it **translation invariant**?
- **Equivariance:** a function f is equivariant w.r.t. to a transformation τ iff $f \circ \tau = \tau \circ f$.
- **Translations (circular):** For any $u \in [\![1, N]\!]$ and input $x \in \mathbb{R}^N$, let $\tau_u(x)_i = x_{i+u[N]}$.

But... why convolutions?

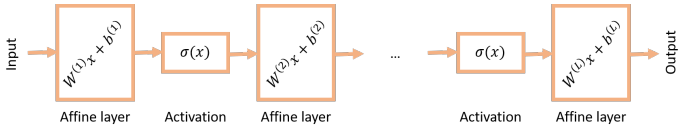


- **Idea:** why not take an MLP and make it **translation invariant**?
- **Equivariance:** a function f is equivariant w.r.t. to a transformation τ iff $f \circ \tau = \tau \circ f$.
- **Translations (circular):** For any $u \in [\![1, N]\!]$ and input $x \in \mathbb{R}^N$, let $\tau_u(x)_i = x_{i+u[N]}$.

Lemma (convolutions)

The only linear functions that are **translation equivariant** are the **convolutions**.

But... why convolutions?



- **Idea:** why not take an MLP and make it **translation invariant**?
- **Equivariance:** a function f is equivariant w.r.t. to a transformation τ iff $f \circ \tau = \tau \circ f$.
- **Translations (circular):** For any $u \in [\![1, N]\!]$ and input $x \in \mathbb{R}^N$, let $\tau_u(x)_i = x_{i+u[N]}$.

Lemma (convolutions)

The only linear functions that are **translation equivariant** are the **convolutions**.

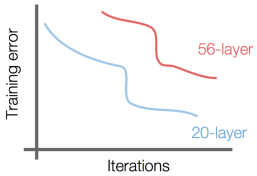
Proof.

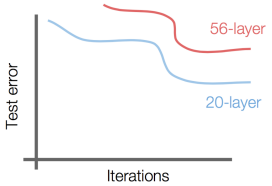
- By linearity, we have $f(x)_i = \sum_j M_{i,j} x_j$.
- Then, we have $\sum_j M_{i,j} x_{j+u[N]} = \sum_j M_{i+u[N],j} x_j$ and $\forall i, j, u,\ M_{i,j} = M_{i+u[N],j+u[N]}$.

The ResNet architecture

Creating deeper neural networks

How deep can we go?





- Some properties require a large number of simple operations.
- **Limitations of VGG:** can't add too many layers (due to **vanishing gradients**).

Residuals

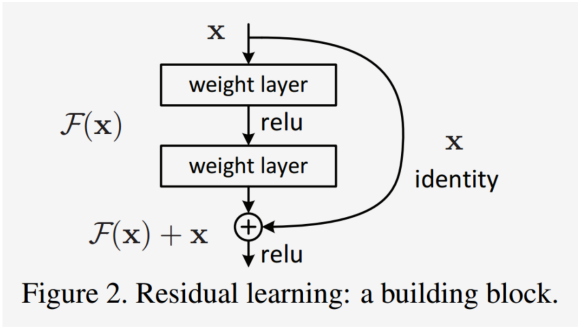
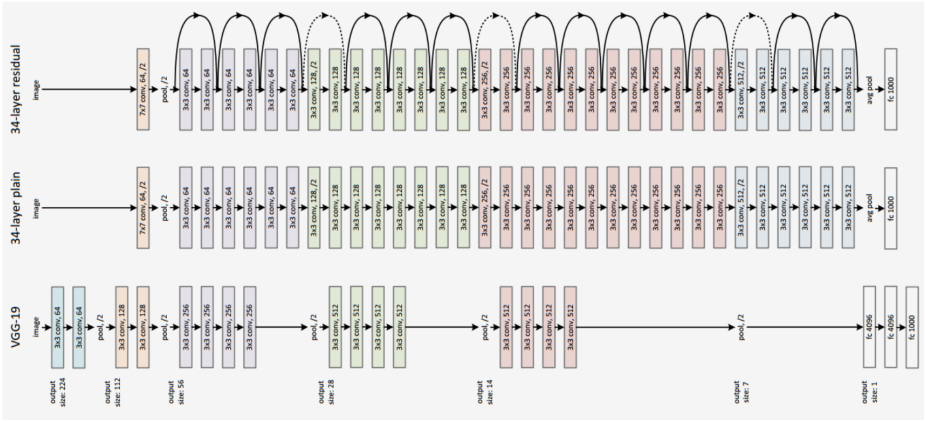


Figure 2. Residual learning: a building block.

- **Idea:** only encode the **residual**: $x^{(l+1)} = x^{(l)} + g_\theta(x^{(l)})$ where g_θ is a computation block.
- **Impact:** Increases **stability** (gradients closer to 1, mapping closer to identity).

source: K. He et al., Deep residual learning for image recognition, CVPR 2016.

The ResNet architecture



source: K. He et al., Deep residual learning for image recognition, CVPR 2016.

The ResNet architecture

- Even **deeper ResNet models** are possible: 34, 50, 101, and 152 layers!

The ResNet architecture

- Even **deeper ResNet models** are possible: 34, 50, 101, and 152 layers!

ResNet50 compared to VGG

- **Accuracy:** Superior in all vision tasks: **5.25%** top-5 error vs 7.1%
- **Less parameters: 25M** vs 138M
- **Computational complexity: 3.8B Flops** vs 15.3B Flops
- **Fully Convolutional** until the last layer

Performance of ResNet architectures

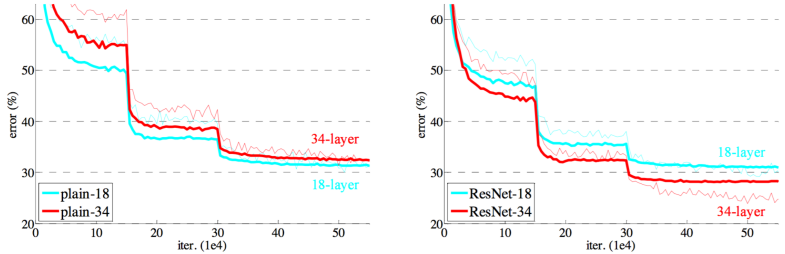
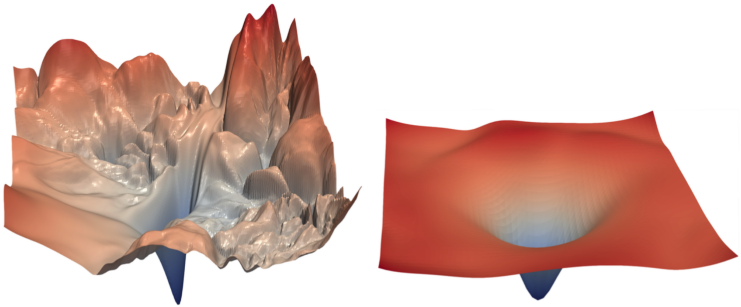


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

source: K. He et al., Deep residual learning for image recognition, CVPR 2016.

Impact on the loss landscape



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The vertical axis is logarithmic to show dynamic range. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

source: K. He et al., Deep residual learning for image recognition, CVPR 2016.

Performance of ResNet architectures

method	top-5 err. (**test**)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

source: K. He et al., Deep residual learning for image recognition, CVPR 2016.

Recap

- **CNN** = **convolutions** + **pooling** (+ activations + BatchNorm)
- Convolutions are (the only) **local**, **translation equivariant** linear mappings.
- First layers extract low-level **local** features of the image.
- Last layers extract high-level **global** features of the image.
- Receptive field of neurons increases as we move towards the output.
- Residuals improve **stability** and **performance** for very deep CNNs.