DEEP LEARNING

Machine Learning for Autonomous Robots

Dr. Matias Valdenegro[1] & Alexander Fabisch[2]
[1]Assistant Professor at University of Groningen
[2]DFKI, Robotics Innovation Center

January 09, 2023 – Bremen, Deutschland

University of Bremen

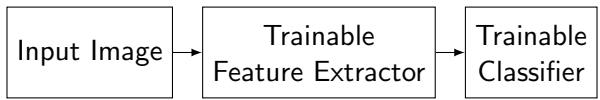
German Research Center for Artificial Intelligence

Introduction

Typical Machine Learning Pipeline

- Handcrafted features can be: SIFT, SURF, HoG, LBP, or any kind of feature engineering, or the original data itself.
- Trainable classifiers are typically SVMs, Random Forests, Logistic regression, etc.

Deep Learning Pipeline



- The model is trained in a *end-to-end* fashion, meaning no feature engineering is performed, and all features are learned using gradient descent.
- In general, learning features from data beats feature engineering all the time.
- The only issue is that now we have to *design* the learnable feature extractor module.

What is Deep Learning?

Definition 1 from ?

A class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification.

Alex's definition

Machine learning techniques to train neural nets with many layers and many neurons.

What is Deep Learning?

My Definition

Hierarchical models that entirely learn features and tasks from data.

Incorporates some important concepts:

- Feature hierarchies.
- Learning from data.
- End to End Learning.
- Tasks $\rightarrow$ Classification, regression or a combination (Multi-task Learning) of both.

Feature Hierarchies

Visual Information

Pixels $\rightarrow$ Edges $\rightarrow$ Textons $\rightarrow$ Parts $\rightarrow$ Objects.

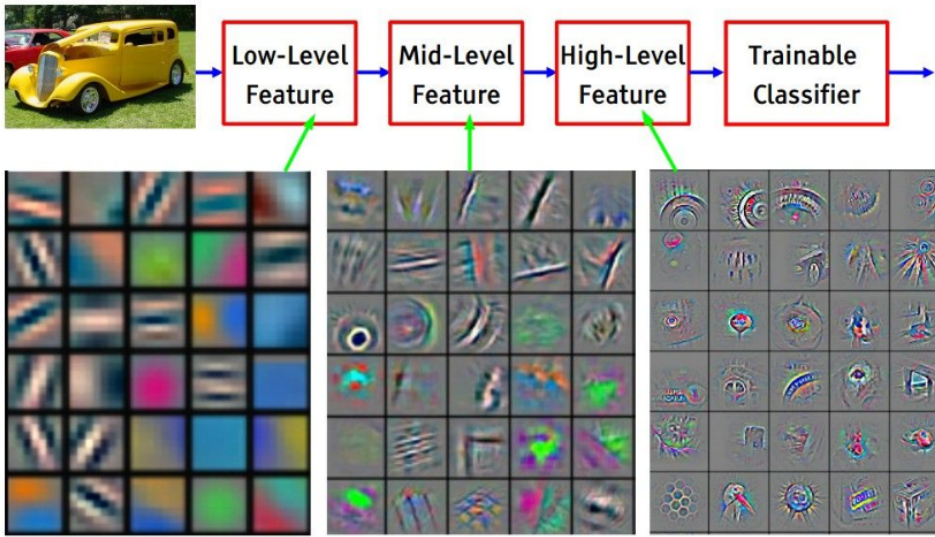
Textual Information

Character $\rightarrow$ Word $\rightarrow$ Word Groups $\rightarrow$ Sentence $\rightarrow$ Story.

Acoustic Information

Sample $\rightarrow$ Spectral Band $\rightarrow$ Sound $\rightarrow$ Phoneme $\rightarrow$ Word.

Feature Hierarchies



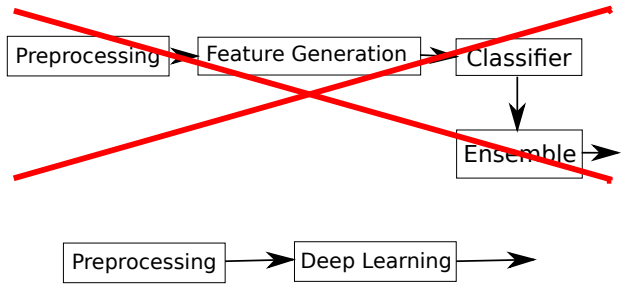
Why Depth/Deep?

- Easier to train than "Wide" models.
- Requires less data, a Wide model might require an exponential amount of data, while a Deep model require only a linear amount (which can still be too much data).
- Learns the feature hierarchy. This cannot be done in Shallow or Wide models.
- Trade-off between Parallel vs Sequential Computation. Wide models are easily parallelizable, while Deep models are not.

Why Learning Features is good?

- Domain adaptation. Same architecture learns different features depending on data and labels.
- Exploits structures that might not be intuitive but still be present in data.
- Exploits the most relevant structures first.
- Relevant features are different for each problem.
- Practice moves faster than theory.

Advantages



- Deep learning replaces the ML pipeline
- One method for all modalities (no domain-specific features)
- We can reuse nets as feature generators for other problems (transfer learning)
- Knowledge can be learned and then used for other tasks (This is called Transfer Learning).

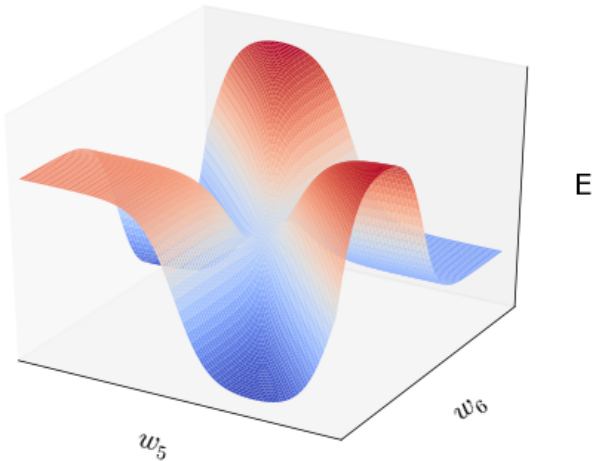
Disadvantages

- Large quantities of data *might* be required, depending on task.
- Large datasets and networks require large computational resources (# of CPUs, # of GPUs).
- Large networks usually use lots of RAM. This is limited in GPUs which is a problem.
- Vanishing gradient problems and issues during training.

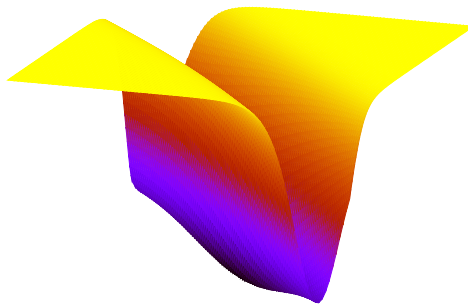
Problems

- There are many **local minima**.
- There is **no guarantee** to reach the global minimum.
- Neural networks are considered to be the **blackest box** of all learning algorithms.
- Training neural networks has been regarded as **black magic** by many researchers (the grimoire: *Neural Networks: Tricks of the Trade*; 1st edition: 1998, 2nd edition: 2012).
- There are sooo many **hyperparameters** (number of layers/nodes, activation function, connectivity, ...).
- The optimization problem is **ill-conditioned**.
- There are many **flat regions** (e.g. saddle points).
- Deep architectures suffer from the **vanishing gradient**.

Multiple Local Minima



Ill-conditioning



Saddle Points

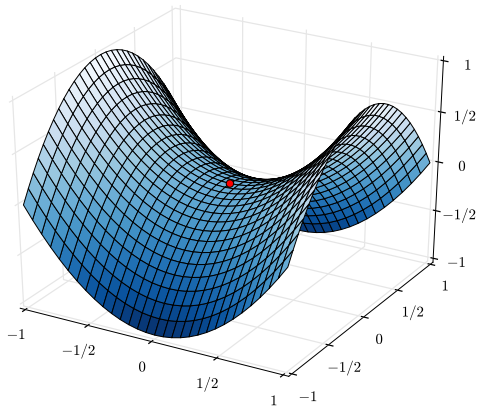


Figure: https://en.wikipedia.org/wiki/Saddle_point#/media/File:Saddle_point.svg

Vanishing (or Exploding) Gradient Problem

Deltas will become smaller in first layers (**with sigmoid activation functions**), which results in an **ill-conditioned** optimization problem.

$$\nabla_{\mathbf{x}} E = \mathbf{W}^T \left(g'(\mathbf{a}) \circ \nabla_{\mathbf{y}} E \right), \quad g' = 1 - g^2 \in [0, 1]$$

There are two problems:

- Active neurons **saturate** and prevent error backpropagation:

$$g(\mathbf{a}) \approx 1 \rightarrow \nabla_{\mathbf{x}} E \approx 0$$

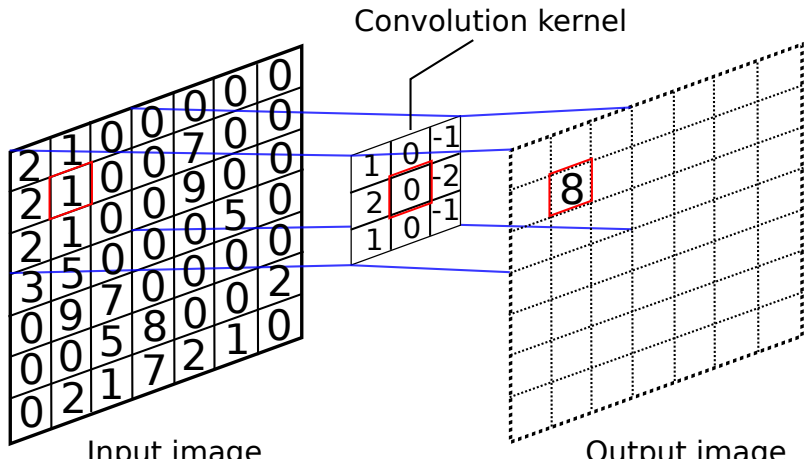
- **W** is usually initialized with random values $|w_{ji}| \ll 1$, hence, the gradient magnitude decays exponentially with each layer, due to the maximum eigenvalue of the weight matrix.

Solutions

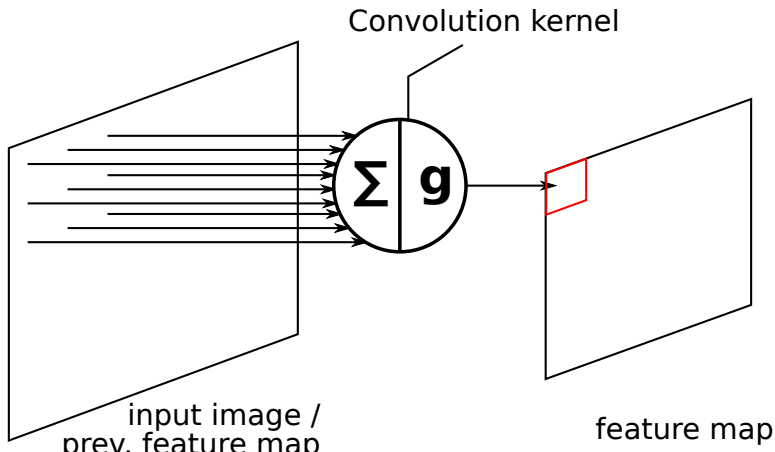
- Convolutional Neural Networks (CNNs; 1989)
- Unsupervised *pretraining* of lower layers (*Deep Learning breakthrough*; 2006, not really used anymore)
- Rectified Linear Unit (ReLU) activation function ($\max(a, 0)$; 2009)
- Better optimization and regularization algorithms (with low complexity for large datasets and many weights; 2010)

CNNs

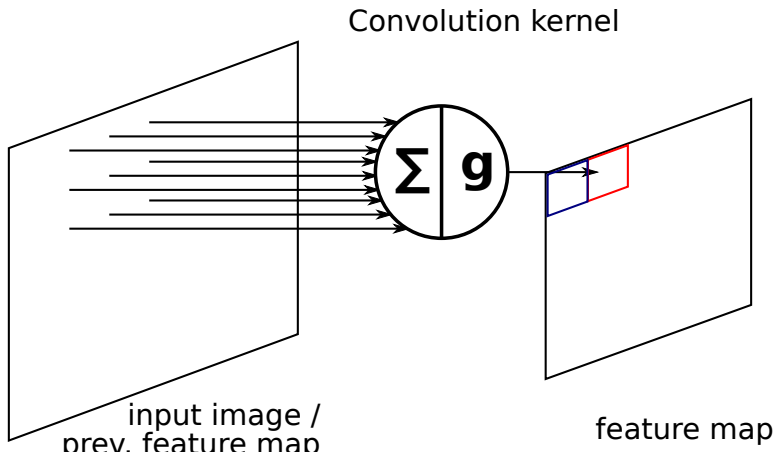
Image convolution



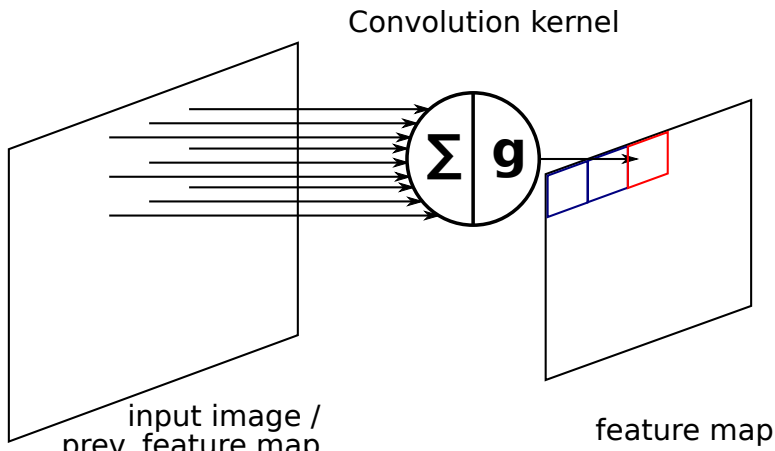
Convolutional Layer



Convolutional Layer



Convolutional Layer



Convolutional Layer

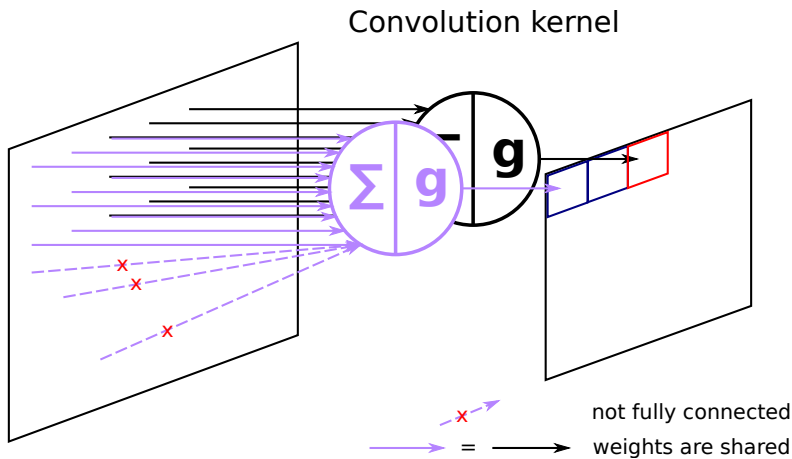


Image convolution

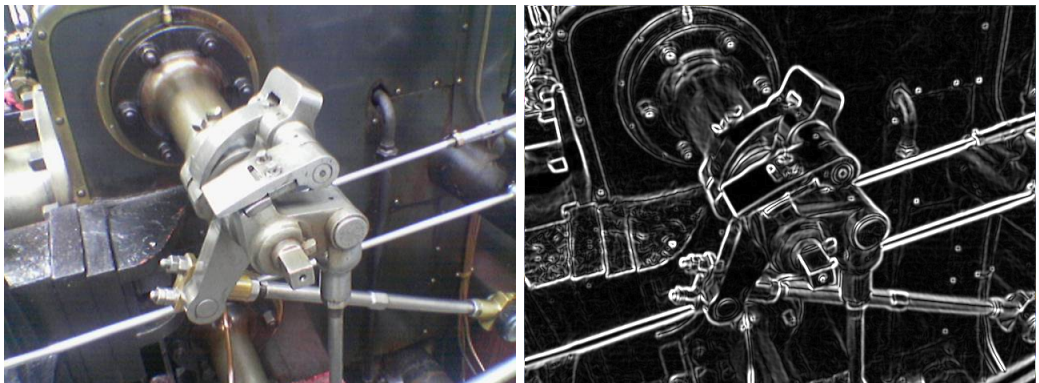


Figure: Edge detection with Sobel filter (source: `http://en.wikipedia.org/wiki/Sobel_operator`)

Idea of Convolutional Neural Nets

- Use learnable filter kernels
- Build multiple 2D **feature maps**
- Each feature map in one layer is connected to each feature map from the previous layer and ...
- we use different filter kernels for each pair of feature maps.
- **We exploit the structure of images: in comparison to a fully connected layer, we make use of the spatial order of the pixels**

→ We reduce the number of connections significantly and the number of weights even more.

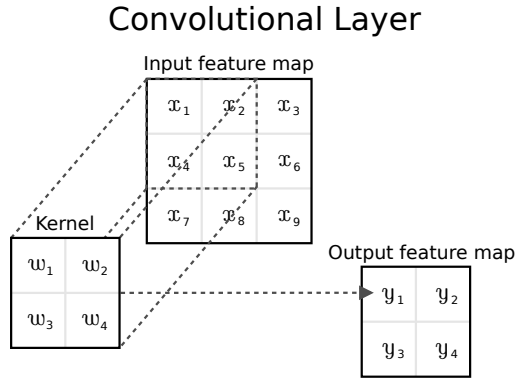
→ We gain a little bit rotation and illumination invariance.

Convolutional layer: a special form of fully connected layer

The input feature map is convolved with a parametrizable kernel to compute the output. For example,

$$y_1 = w_1 x_1 + w_2 x_2 + w_3 x_4 + w_4 x_5.$$

We can construct a fully connected layer, that computes the same output, where $\boldsymbol{W}$ is **sparse** and weights are **shared** among neurons:



$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_9 \end{pmatrix}$$

Does it “fix” the problem?

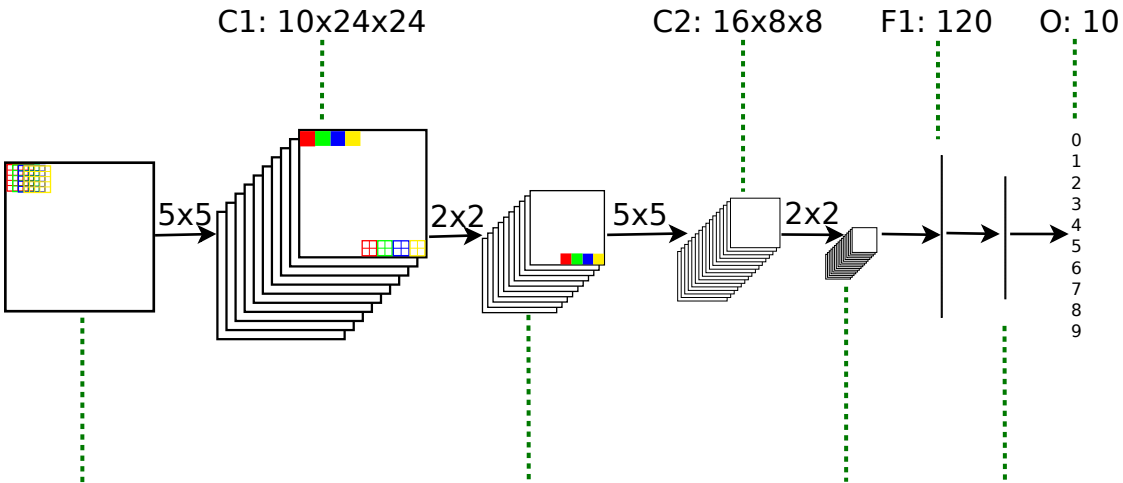
- Because weights are shared among neurons, their derivatives will be very big even in lower layers.
- Constraining most weights to be zero alleviates some optimization caveats (local minima, flat regions).
- Convergence is usually very fast with stochastic gradient descent.
- Using the ReLU activation function and pooling layers helps.

Notation

Matrix multiplication	$\mathbf{A} \cdot \mathbf{B}$
Hadamard product (coefficient-wise)	$\mathbf{A} \circ \mathbf{B}$
Convolution	$\mathbf{A} * \mathbf{B}$
Gradient of function f with respect to $\mathbf{A}$	$\nabla_{\mathbf{A}} f$

Convolution / Pooling

Notation: feature maps x rows x columns



Fully Connected

Purpose: learn weighting of the inputs

Forward pass:

$$\mathbf{a} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}, \quad \mathbf{y} = g(\mathbf{a})$$

Backward pass:

$$\boldsymbol{\delta} = g'(\mathbf{a}) \circ \nabla_{\mathbf{y}} E$$
$$\nabla_{\mathbf{W}} E = \boldsymbol{\delta} \cdot \mathbf{x}^{\mathbf{T}}, \quad \nabla_{\mathbf{b}} E = \boldsymbol{\delta}$$
$$\nabla_{\mathbf{x}} E = \mathbf{W}^T \cdot \boldsymbol{\delta}$$

Convolutional

Purpose: dimensionality reduction, preprocessing, feature learning
Forward pass:

$$\mathbf{A}^j = \sum_{i=0}^{I-1} \mathbf{W}^{ji} * \mathbf{X}^i + b^{ji} \cdot \mathbf{1}, \quad \mathbf{Y}^j = g(\mathbf{A}^j),$$

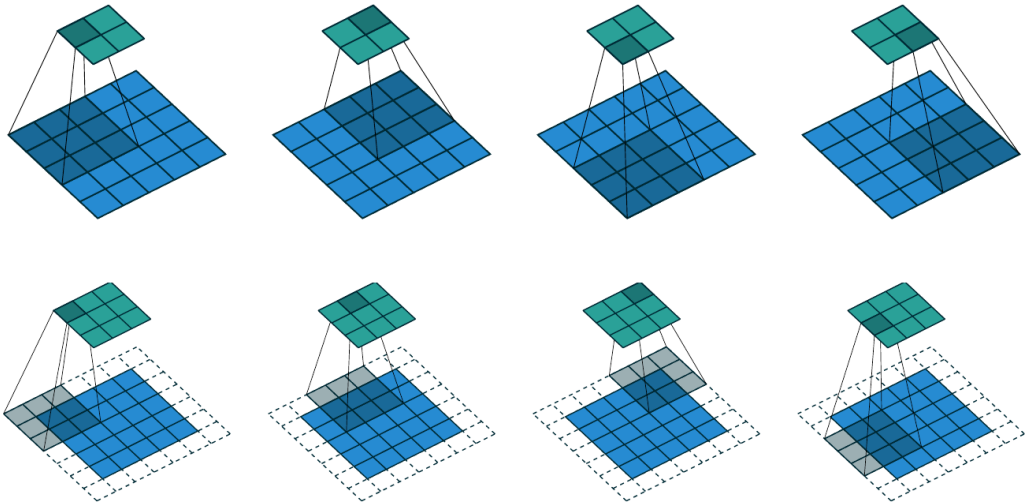
where

- $\boldsymbol{X}^i$ is the i-th feature map in the input
- $\boldsymbol{Y}^j$ is the j-th feature map in the output
- $\mathbf{W}^{ji}$ is the filter kernel and b^{ji} the bias between feature maps i and j
- $\mathbf{1}$ is a matrix of ones and has the same size as the output feature map
- Note that the filter kernel will usually be moved by only one pixel in each step (they have a stride of 1), i.e. they overlap

Strided Convolutions

- The stride S is the amount of pixels that the sliding window moves. It is typically set to $S = 1$.
- A stride $S > 1$ additionally performs down-sampling of the feature map. This also reduces the amount of required computation.

Strided Convolutions



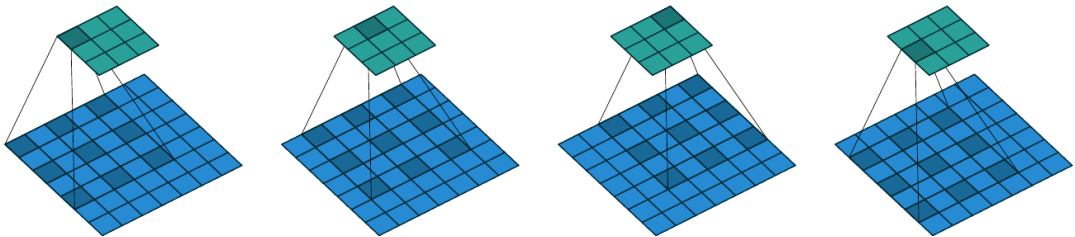
(Green is the output feature map, blue is the input, and dark blue/grey is the filter kernel)

Dilated Convolutions

- A dilated convolution is the one where there are *spaces* in the filter kernel. These spaces are fixed to a zero value.
- The dilation rate D controls how many spaces are inserted between each element in the filter kernel. $D = 1$ corresponds to standard convolution, implying that $D - 1$ spaces are actually inserted.
- The effective filter size $\hat{F}$ is given by:

$$\hat{F} = F + (F - 1)(D - 1)$$

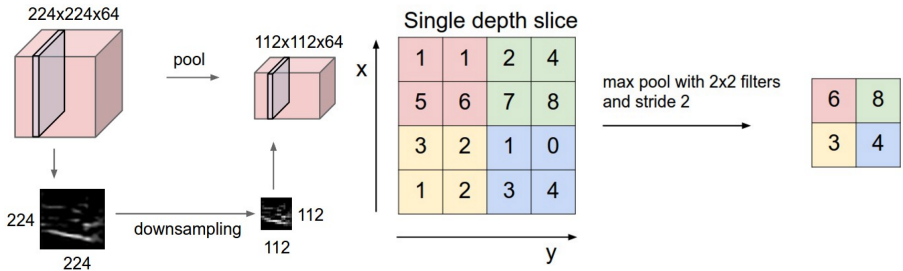
Dilated Convolutions



(Green is the output feature map, blue is the input, and dark blue/grey is the filter kernel)

Pooling

Purpose: dimensionality reduction and local translation invariance

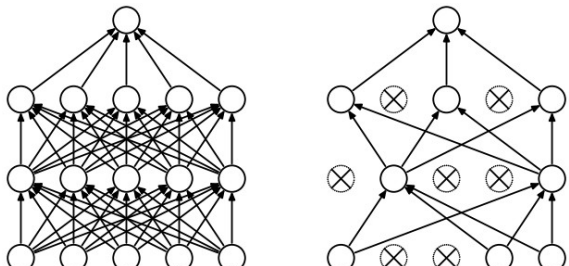


Max-pooling example (source: Andrej Karpathy, Stanford CS class CS231n, `http://cs231n.github.io/`)

- Summarize low-level features in the input region
- Reduce the feature dimension and number of parameters to be learned
- Pooling function can be max or average.

Dropout

Purpose: generalize better to new data (regularization method)



(a) Standard Neural Net

(b) After applying dropout.

- During training, each neurons is present with probability p
- Reduces overfitting due to co-activations
- Features need to be useful by themselves (robustness)
- Applied primarily in fully connected layers

Batch Normalization

This layer normalizes the activation of the previous layer with the mean μ_z and standard deviation σ_z across batches:

$$\hat{z}_k = \frac{z_k - \mu_k}{\sigma_k}$$

The idea is to reduce the co-variate shift during training. To restore the network's original representation power, the normalized activation is denormalized using learned factors:

$$z'_k = \gamma_k \hat{z} + \beta_k$$

The index k indicates that normalization and scaling is done for each feature dimension separately. The effect of BN is to make convergence faster, reducing training time, and stabilizing training.

Input-Output Shapes

Layer	Input Shape	Output Shape	Remarks
Fully Connected	$(f,)$	$(n,)$	Only operates on the last dimension of input
Convolutional	(w_i, h_i, c_i)	(w_o, h_o, c_o)	$()_o = \frac{()_i - f + 2p}{s}$
Pooling	(w_i, h_i, c)	$(\frac{w_i}{p}, \frac{h_i}{p}, c)$	p is pooling factor (usually $p = 2$)
Global Pooling	(w_i, h_i, c)	$(1, 1, c)$	Consumes spatial dimensions

Parameters are: f for input feature dimensionality or filter size, c for channels, n for number of neurons, p for padding, and s for strides. Convolutional dimension can be maintained if you set $p = \frac{f-1}{2}$ with $s = 1$.

Other Layers

- Subsampling, average pooling
- Split, merge, reshape, crop, interpolate, pad, add, ...
- Recurrent, Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), ...
- Activation functions: Leaky ReLU, PReLU, ELU, Softmax, ...
- ...

Activation Functions

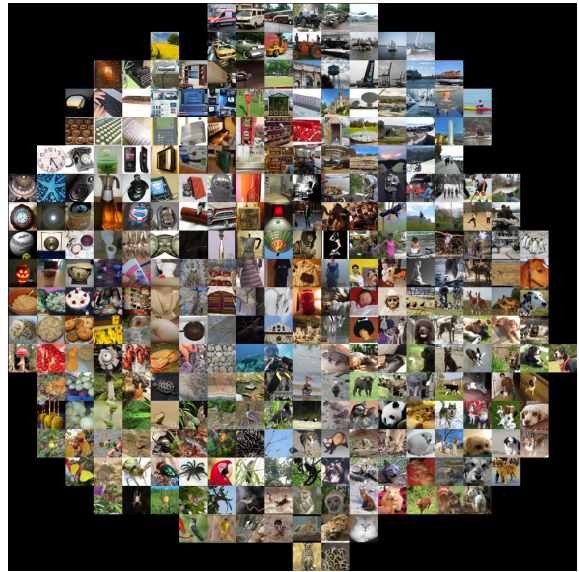
Name	Range	Function
Linear	$[-\infty, \infty]$	$g(x) = x$
Sigmoid	$[0, 1]$	$g(x) = (1 + e^{-x})^{-1}$
Hyperbolic Tangent	$[-1, 1]$	$g(x) = (e^{2x} - 1)(e^{-2x} + 1)^{-1}$
ReLU	$[0, \infty]$	$g(x) = \max(0, x)$
SoftPlus	$[0, \infty]$	$g(x) = \ln(1 + e^{x})$
SoftMax	$[0, 1]^n$	$g(\mathbf{x}) = (e^{x_i})(\sum_k e^{x_k})^{-1}$
LeakyReLU	$[0, \infty]$	$g(x) = \max(0.01x, x)$
Parametric ReLU	$[0, \infty]$	$g(x) = \max(\alpha x, x)$
Swish	$[-1, \infty]$	$g(x) = x \times \text{sigmoid}(x)$

Comparison of Current Architectures

Publication	Ciresan et al. [2010]	**?**
Dataset	MNIST (with distortions)	1.2M images, 1k classes
Model	MLNN (12.11M weights)	CNN (60M weights)
Training time	5 days on GPU	1 week on 2 GPUs

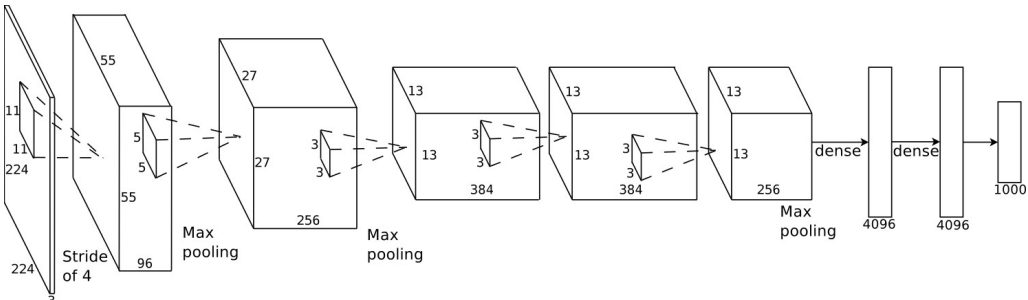
Publication	Le et al. [2012]	Nature et al. [-70000]
Dataset	10M images, 22k classes	unknown
Model	10^9 connections	10^{11} neurons, 10^{15} synapses
Training time	1 week on 2k machines	18 years on brain

ImageNet



- Data: 1.2M+ images, 1k-22k classes, depends on the subset
- `http://www.image-net.org/`
- **Link** → How does a human compare to ConvNets on this dataset?

SuperVision aka AlexNet (**?**)



- Trained on ImageNet (1.2M images, 1k classes)
- Model: 11 layers (5 conv., 3 max-pooling, 3 fully connected), 650k neurons, 60M weights, 630M connections, 4096 features in 10th layer
- Training: one week on 2 GPUs (1 GPU would not have enough memory)

Validation


koala
wombat
Norwegian elkhound
wild boar
wallaby
koala
tiger
tiger
tiger cat
jaguar
lynx
leopard
European fire salamander
European fire salamander
spotted salamander
common newt
long-horned beetle
box turtle
loggerhead
African crocodile
Gila monster
loggerhead
mud turtle
leatherback turtle
seat belt
seat belt
ice lolly
hotdog
burrito
Band Aid
television
television
microwave
monitor
screen
car mirror
sliding door
sliding door
shoji
window shade
window screen
four-poster
wallaby
hare
wallaby
wood rabbit
Lakeland terrier
kit fox

Inception [Szegedy et al. 2014]

- Google Researchers proposed this network for the ImageNet competition, it is also called Inception.
- The name Inception comes from convolutional modules that contain smaller convolution operations inside them.
- The idea of these inception modules is to perform convolution at different filter scales, which extract different kinds of features. Then the designer stack several kinds of these modules to make a full network.

Inception [Szegedy et al. 2014]

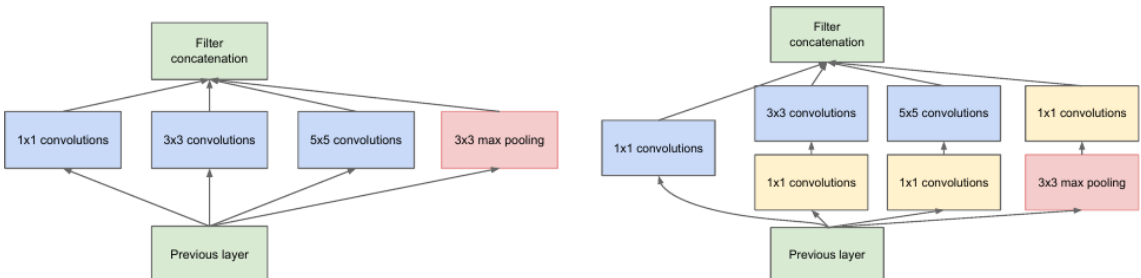


Figure 2: Inception module

Inception [Szegedy et al. 2014]

Convolution
Pooling
Softmax
Other

Inception [Szegedy et al. 2014]

- 22 layers, 9 inception modules stacked.
- Inception modules represent more functions with less parameters and computation.
- 7 Network ensemble trained with 144 crops.
- 5 Million parameters [1].
- Top-5 error of 6.67% on ImageNet.

[1] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Rabinovich, A. "Going deeper with convolutions". 2015

VGG [Simonyan and Zisserman 2015]

- VGG stands for the Visual Geometry Group at Uni of Oxford.
- It got the 2nd place at the 2014 ImageNet competition, but researchers have experimentally found that the features produced by this network are considerably better.
- Typically used for Object Detection (SSD) and Image Captioning. It is basically a deeper version of AlexNet.

VGG [Simonyan and Zisserman 2015]

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

VGG [Simonyan and Zisserman 2015]

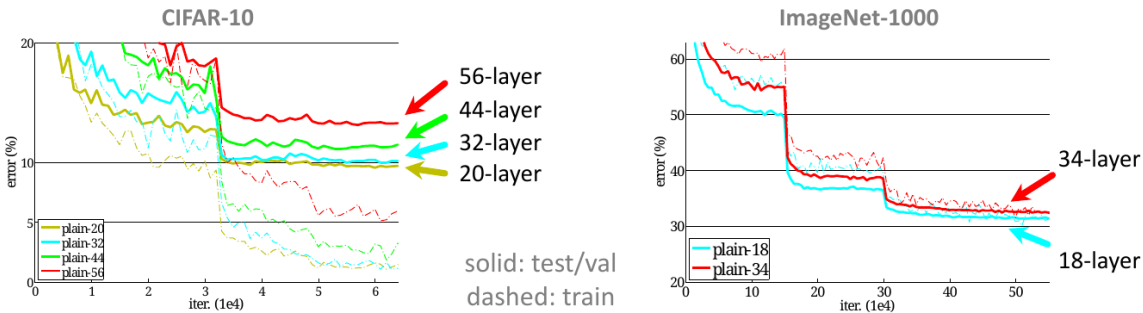
- Uses two 3×3 filters to emulate a 5×5 filter.
- 16-19 layers, only 3×3 filters are used.
- Configuration E has 144 Million parameters, Configuration A has 133 Million parameters.
- Top-5 error of 6.8% on ImageNet over a ensemble of two networks and multiple crops [2].

[2] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." 2014

ResNet [He et al. 2015]

- Residual Networks was developed as a way to make networks deeper.
- The authors at MS Research Asia found that if you make current networks (GoogleNet, VGG) deeper, performance actually decreases instead of increasing as normally it does.
- This is because of vanishing gradient issues. They proposed a way to propagate the gradient forward as the network becomes deeper.

ResNet [He et al. 2015]



ResNet [He et al. 2015]

- The residual block contains a skip connection that can propagate the gradient forward through the layer. If a layer computes $F(x)$, then adding this skip connection will compute $F(x) + x$
- There is an additional constraint that the output of $F(x)$ must have the same dimensionality as x, in order for the addition to be possible.
- If only spatial dimensions match (and not channels), then instead of addition, concatenation can be used.

ResNet [He et al. 2015]

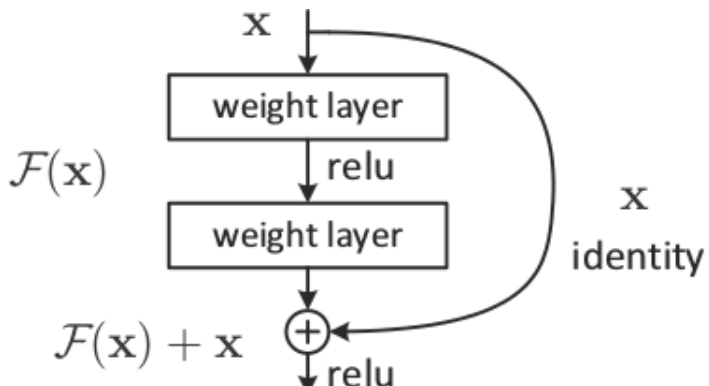


Figure 2. Residual learning: a building block.

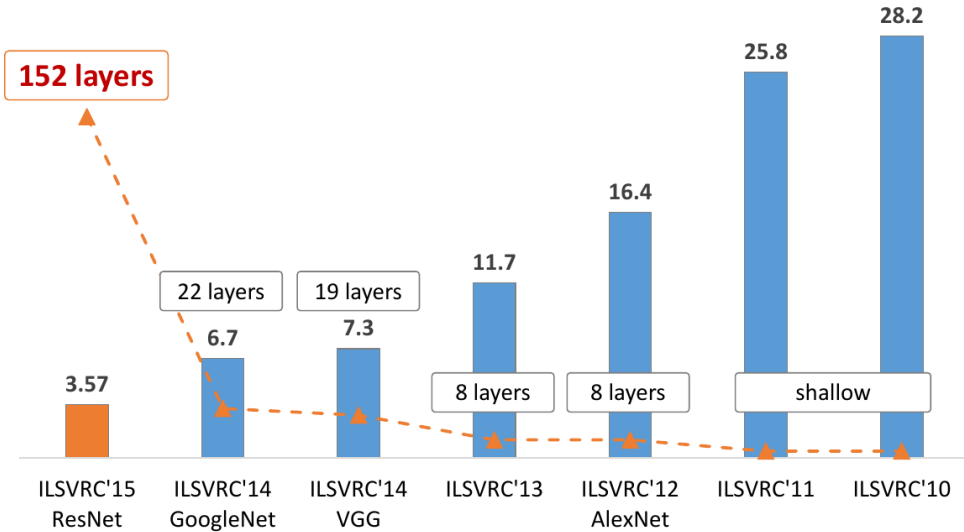
ResNet [He et al. 2015]

ResNet [He et al. 2015]

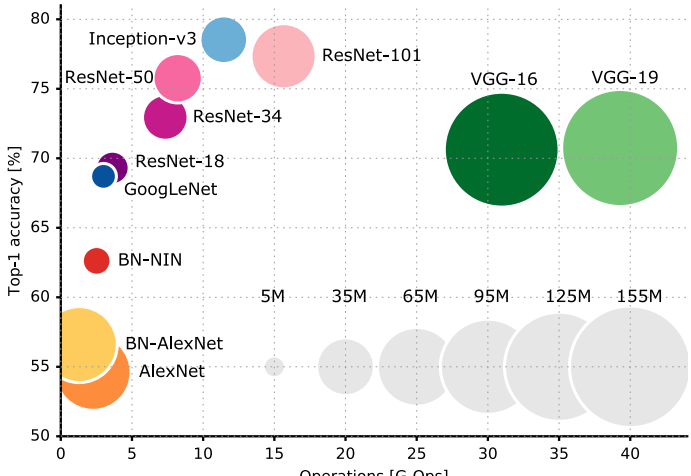
- ResNet for ILSVRC has 152 layers, approx 2.5 Million parameters.
- 3.57% Top-5 error on ImageNet.
- Authors tested a 1202-layer network for other purposes.
- The limit is now GPUs memory [3].

[3] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." 2015.

Summary of progress on ImageNet



Architecture Overview



Prediction performance (computational effort vs. accuracy) and complexity of current deep neural network architectures (2015).

Some Architectural Patterns

- One pattern in AlexNet/VGG is that the number of filter is increasing according to depth.
- AlexNet has filters: 96, 256, 384, 56.
- VGG has filters: 64, 128, 256, 512, 512.
- ResNet has filters: 64, 128, 256, 512.
- Note the use of powers of two for the number of filters, while filter sizes are usually odd numbers.

Optimization

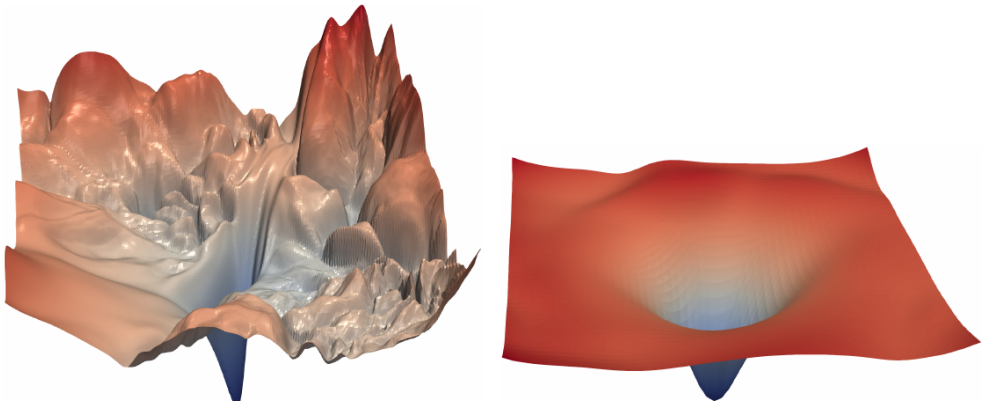
Improving the Generalization

- Do we really only want to minimize the error on the **training set**?

$$E = \sum_n ||\boldsymbol{y}_n - \boldsymbol{t}_n||_2^2$$

- We can add a penalty term for large weights:
  - L1-Norm: $E \leftarrow E + \gamma \sum_k |\boldsymbol{w}_k|$
  - Squared L2-Norm: $E \leftarrow E + \gamma \boldsymbol{w}^T \boldsymbol{w}$
- We can constrain the weights of each neuron j: $\sum_i w_{ji}^2 \leq c$
- **Dropout**: We can randomly suppress the output of neurons (prevents co-adaption of neurons, similar to ensemble methods)
- Early stopping: observe the error on a validation set during optimization

ResNet Loss Landscape [Li et al. 2017]



(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.

Optimization for Deep Networks: What does **not** work?

An optimization algorithm that uses second-order information would be desirable (e.g. Newton's method), however,

- they usually require too much space (e.g. $O(MN)$ or $O(M^2)$)
- they usually require too much time (e.g. $O(M^3)$)
- they usually do batch updates, i.e. they need to see the whole training set for an update (which is often very large in deep learning problems)

M: number of parameters (weights, biases); N: size of training set

What would work?

We need an algorithm that can cope with

- large datasets (e.g. >1M)
- many parameters (e.g. >10M)
- ill-conditioned objective functions
- local minima

Many solutions have been proposed for this problem, e.g.

- variations of Levenberg-Marquardt, Newton's method, ...
- mini-batch updates with L-BFGS, conjugate gradient
- **variations of stochastic gradient descent**

Gradient Descent

Update rule:

$$\Delta \boldsymbol{w}_t = \frac{\alpha}{|T|} \sum_{n \in T} \boldsymbol{g}_n, \quad \boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \Delta \boldsymbol{w}_t$$

T: training set $\{(\boldsymbol{x}_0, \boldsymbol{y}_0), \ldots, (\boldsymbol{x}_n, \boldsymbol{y}_n), \ldots\}$
$\boldsymbol{g}_n$: gradient of the error function with respect to a specific training instance

Stochastic Gradient Descent

Update rule: $\Delta \boldsymbol{w}_t = \alpha \boldsymbol{g}_n$

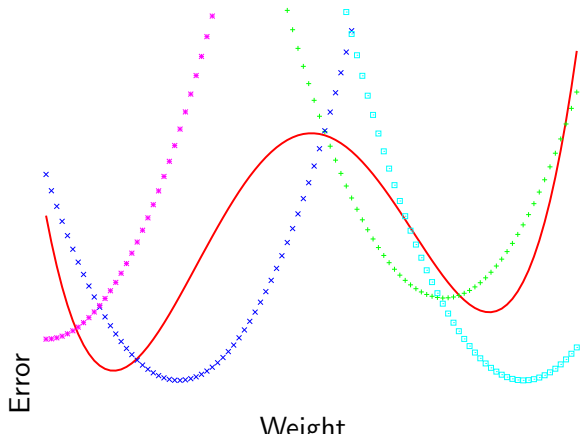


Figure: Gradients of single instances can be used to escape local minima.

Mini-Batch Stochastic Gradient Descent

Update rule:

$$\Delta \boldsymbol{w}_t = \frac{\alpha}{|B_t|} \sum_{n \in B_t} \boldsymbol{g}_n,$$

B_t is a **mini-batch**

- Gradients of mini-batches can be used to escape local minima.
- **Gradient will be smoother than SGD gradient.**
- **Gradients of mini-batches can be computed in parallel (GPU, SIMD, multicore).**

MBSGD and Learning Rate Schedule

Update rule:

$$\Delta \boldsymbol{w}_t = \frac{\alpha_t}{|B_t|} \sum_{n \in B_t} \boldsymbol{g}_n, \quad \alpha_t = \max(\alpha_0 \alpha_{decay}^t, \alpha_{min})$$

- Gradients of mini-batches can be used to escape local minima.
- Gradient will be smoother than SGD gradient.
- Gradients of mini-batches can be computed in parallel.
- $\boldsymbol{w}^t$ **will converge at the end of the optimization.**

MBSGD and Momentum

Update rule:

$$\Delta \boldsymbol{w}_t = \eta_t \Delta \boldsymbol{w}_{t-1} - \frac{\alpha_t}{|B_t|} \sum_{n \in B_t} \boldsymbol{g}_n, \quad \boldsymbol{w}_t = \boldsymbol{w}_{t-1} + \Delta \boldsymbol{w}_t,$$

- Gradients of mini-batches can be used to escape local minima.
- Gradient will be smoother than SGD gradient.
- Gradients of mini-batches can be computed in parallel.
- $\boldsymbol{w}^t$ will converge at the end of the optimization.
- **Updates will be smoother. It is easier to escape flat regions.**

MBSGD and Momentum

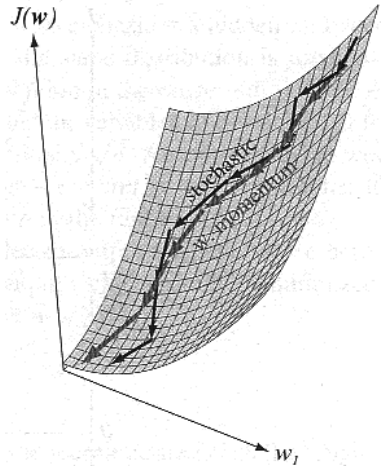
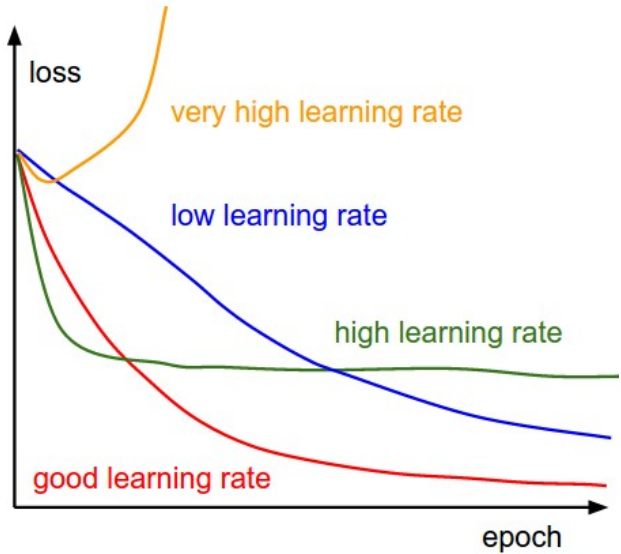


Figure: Source: http://www.cse.unsw.edu.au/ cs9417ml/MLP1/tutorial/advanced-tutorial2.htm

Tuning the Learning Rate

- The most important parameter to have successful training is the learning rate, and the number of training epochs.
- The idea is to set the learning rate to a value that produces consistent decrease of the loss value across batches.
- The number of epochs should be set to a value makes sure that the loss has converged to a small value. Not enough epochs will produce underfitting.

Tuning the Learning Rate



Tuning the Learning Rate

- **High Learning Rate**. The loss diverges to infinity or NaN (not a number). In some cases the loss does not decrease instead, and oscillates.
- **Low Learning Rate**. The loss decreases but slowly, making training unnecessarily long.
- **Good Learning Rate**. The loss decreases quickly, converging into a reasonable number of epochs.

Tuning the Learning Rate

- **High Learning Rate**. Divide the current learning rate by 10, until the loss decreases consistently.
- **Low Learning Rate**. Carefully increase the learning rate, by factors of 2-5.
- **Good Learning Rate**. Keep this learning rate.

Optimizers

- An extension of the previous idea is to have different learning rates for each parameter.
- This is usually done by gathering information about the loss landscape through the gradient, and making adjustments to the learning rate based on this information.
- For example, if the gradient of the loss varies considerably, then the learning rate should be decreased. If the gradient is quite constant, then the learning rate can be increased.

Adagrad

- Adagrad takes this idea by to using the square root of the sum of past squared gradients r_n as a scaling factor for the learning rate:

$$
\begin{aligned}
g_n &= \nabla_\theta L(\theta) \\
r_n &= r_{n-1} + g_n \odot g_n \\
\Theta_{n+1} &= \Theta_n - \alpha \frac{g_n}{\epsilon + \sqrt{r_n}}
\end{aligned}
$$

- Where ϵ is a small constant for numerical stability, and $\odot$ is component-wise multiplication.
- Works but doesn't forget past gradients.

RMSProp

- RMSProp uses a moving exponential average of past gradients, meaning that older gradients are given less weight:

$$r_n = \rho r_{n-1} + (1-\rho) g_n \odot g_n$$
$$\Theta_{n+1} = \Theta_n - \alpha \frac{g_n}{\sqrt{\epsilon + r_n}}$$

- The decay parameter ρ controls the weight of past gradients, it is usually set to 0.9 or 0.99.
- More stable than Adagrad due to better control of gradient history.

Adaptive Moments - Adam

- Combines RMSProp with momentum. The exponentially weighted averages can be biased, and Adam corrects these biases:

$$
\begin{aligned}
s_n &= \rho_1 s_{n-1} + (1-\rho_1) g_n \\
r_n &= \rho_2 r_{n-1} + (1-\rho_2) g_n \odot g_n \\
\Theta_{n+1} &= \Theta_n - \alpha \frac{s_n}{\epsilon + \sqrt{r_n}} \frac{1-\rho_2^n}{1-\rho_1^n}
\end{aligned}
$$

- Adam uses both the first and second moments of the gradients, in general it is the best optimizer.

Adaptive Moments - Adam

- Adam works quite well for most problems, but notably it struggles with some networks with large number of parameters, namely AlexNet and VGG.
- Using Adam with VGG has the effect that the loss does not decrease, most likely because the parameter vector is so large (over 100 million dimensions) that it over-smoothes the gradient.
- Optimizers are no magic or free lunch, one has to be carefully selected. To train VGG, plain SGD with a LR schedule is typically used.

Wrap-Up

Current State of Deep Learning

- Deep learning (almost) replaces the whole ML pipeline in computer vision and speech recognition (preprocessing, feature generation, classification, ensemble)
- Pretrained nets trained on very large datasets (e.g. ImageNet) can be used for transfer learning
- Unsupervised pretraining is important in applications where labels are sparse
- Recurrent neural nets are important as well, e.g. for language translation, handwritten character recognition, image labeling, computer games, etc. (where memory is important to discover temporal dependencies)

Who Uses Deep Learning and Why?

State of the art in computer vision (**?**), speech recognition (**?**), . . . , **huge potential** in computer games, robotics, . . .

Successful Applications

- Google's speech recognition (e.g. for Android phones, Chrome)
- Baidu's search for similar images
- DeepMind is better than human experts at playing some games for the Atari 2600 (Breakout, Enduro and Pong)
- Applications like style transfer of art and new art as dreams of ANNs
- Not so successful: Missing robustness against intelligent image disturbances **?**

Thank You!
Please feel free to ask questions in the forums.