INF8225

Leçon 4

Christopher Pal
École Polytechnique de Montréal

Batch normalization

- A way of accelerating training for which many studies have found to be important to obtain state-of-the-art results
- Each element of a layer is normalized to zero mean and unit variance based on its statistics within a mini-batch
  - This can change the network's representational power
- Each activation has learned scaling and shifting parameter
- Mini-batch based SGD is modified by calculating the mean μ_j and variance σ_j^2 over the batch for each hidden unit h_j in each layer, then normalize the units, scale them using the learned scaling parameter γ_j and shift them by the learned shifting parameter β_j such that

$$\hat{h}_j \leftarrow \gamma_j \frac{h_j - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} + \beta_j.$$

- To update the γ_j and β_j one needs to backpropagate the gradient of the loss through these additional parameters

Dropout

- A form of regularization that randomly deletes units and their connections during training
- Intention: reducing hidden unit co-adaptation & combat over-fitting
- Has been argued it corresponds to sampling from an exponential number of networks with shared parameters & missing connections
- One averages over models at test time by using original network without dropped-out connections, but with scaled-down weights
- If a unit is retained with probability p during training, its outgoing weights are rescaled or multiplied by a factor of p at test time
- By performing dropout a neural network with n units can be made to behave like an ensemble of 2^n smaller networks
- One way to implement it is with a binary mask vector $\mathbf{m}^{(l)}$ for each hidden layer l in the network: the dropped out version of $\mathbf{h}^{(l)}$ masks out units from the original version using element-wise multiplication, $\mathbf{h}_d^{(l)} = \mathbf{h}^{(l)} \odot \mathbf{m}^{(l)}$
- If the activation functions lead to diagonal gradient matrices, the backpropagation update is $\boldsymbol{\Delta}^{(l)} = \mathbf{d}^{(l)} \odot \mathbf{m}^{(l)} \odot (\mathbf{W}^{(l+1)} \boldsymbol{\Delta}^{(l+1)})$.

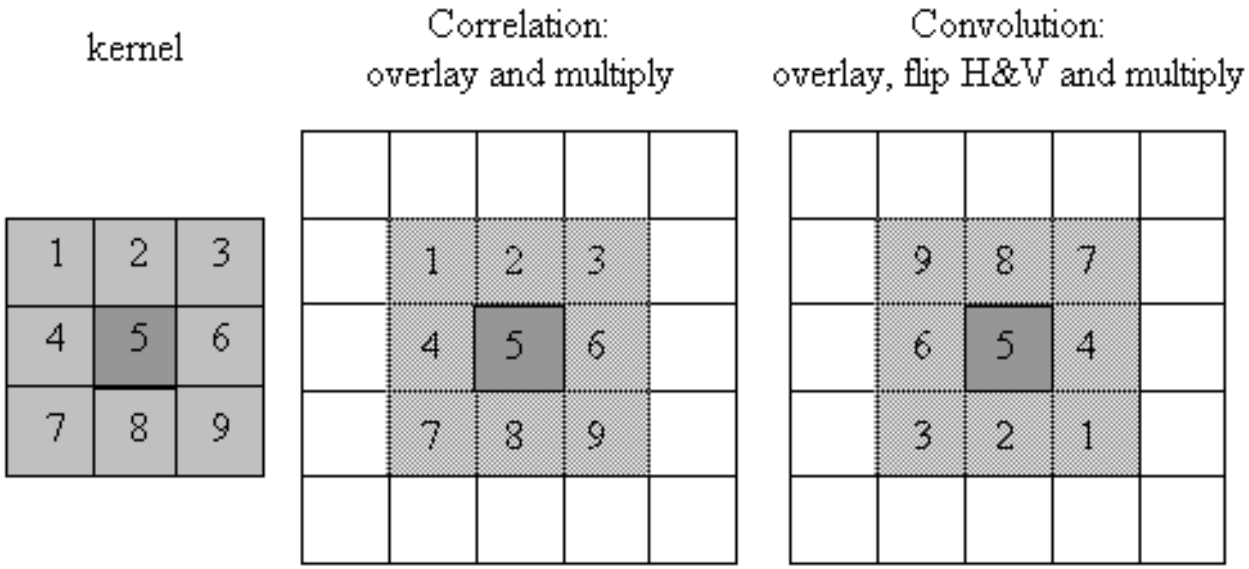
Au menu

- Réseaux de neurones convolutifs
- Notation d'assiette « plate notation », la théorie de maximisation d'espérance, les mélanges gaussiens, et ACP «PCA»
- Autoencodeurs
- Champs aléatoires de Markov et machines Boltzmann

réseaux de neurones convolutifs (préchauffage)

Le filtrage de l'image

Intuition: la corrélation croisée versus le produit de convolution



Merci à U. Rochester

Convolution vs corrélation

- Convolution - de l'intuition à l'équation

$$Conv[i][j] = P_{ij} \otimes K = \sum_{s=-\frac{k_w}{2}}^{\frac{k_w}{2}} \sum_{t=-\frac{k_h}{2}}^{\frac{k_h}{2}} P[i+s][j+t] * K[\frac{k_w}{2} - s][\frac{k_h}{2} - t]$$

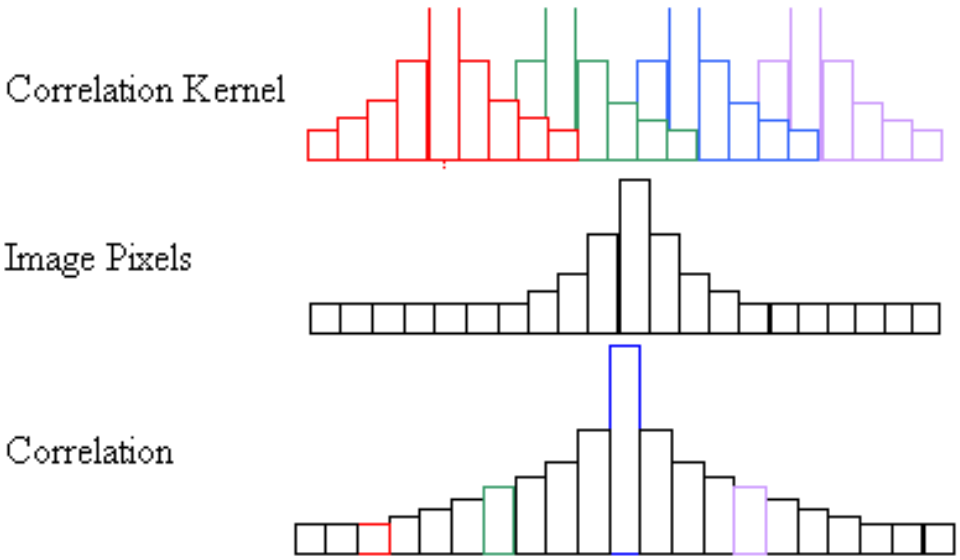
- P - une matrice de valeurs de pixels
- K - le noyau de filtrage

$$Corr[i][j] = \sum_{s=-\frac{k_w}{2}}^{\frac{k_w}{2}} \sum_{t=-\frac{k_h}{2}}^{\frac{k_h}{2}} P[i+s][j+t] * K[\frac{k_w}{2} + s][\frac{k_h}{2} + t]$$

- Corrélation - de l'intuition à l'équation

Quel est le résultat?

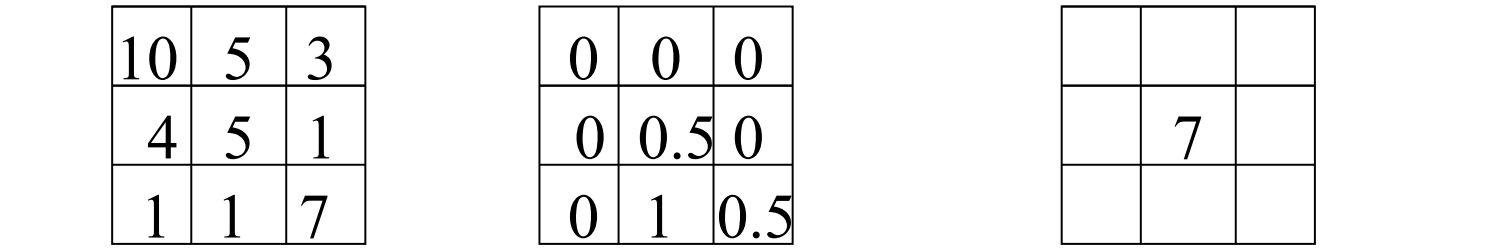
- Lorsque l'image correspond au noyau, il existe une corrélation positive élevée



- Mais, il est aussi possible de penser de l'opération comme une forme de filtrage linéaire

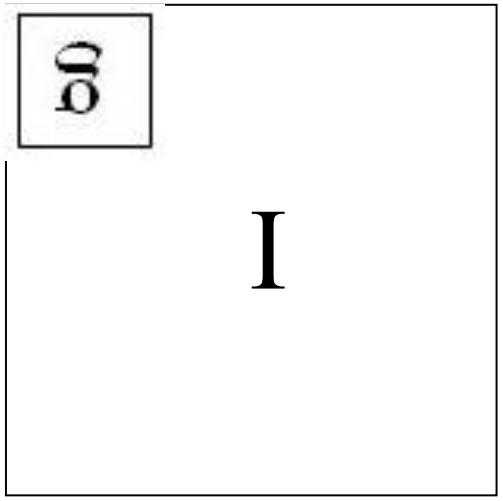
Linear functions

- Simplest: linear filtering.
  - Replace each pixel by a linear combination of its neighbors.
- The prescription for the linear combination is called the "convolution kernel".



Convolution

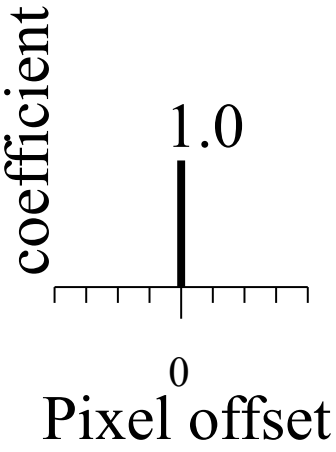
$$f[m,n] = I \otimes g = \sum_{k,l} I[m-k, n-l] g[k,l]$$



Slides from Friedman @ MIT

Linear filtering (warm-up slide)

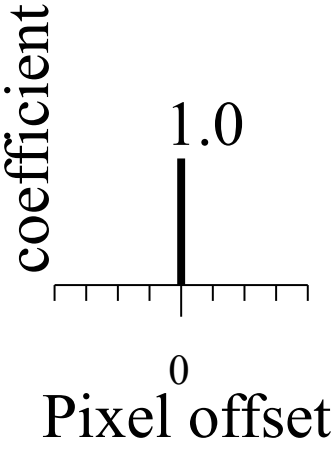
original



?

Linear filtering (warm-up slide)

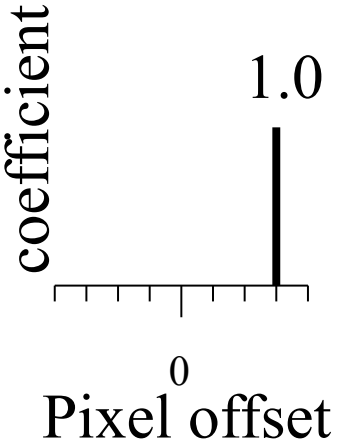
original



Filtered
(no change)

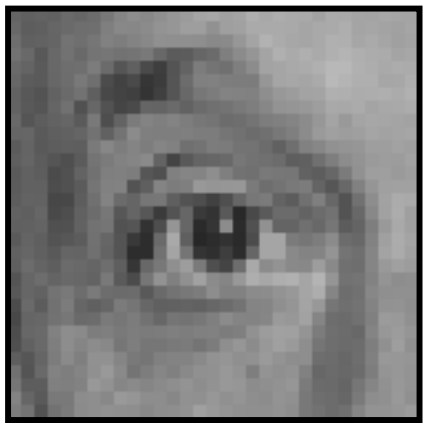
Linear filtering

original

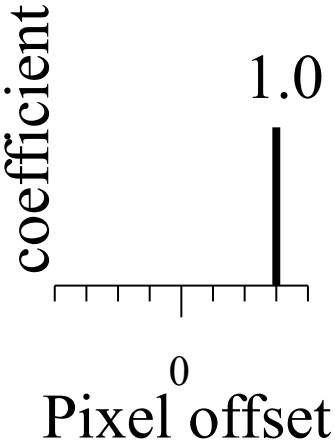


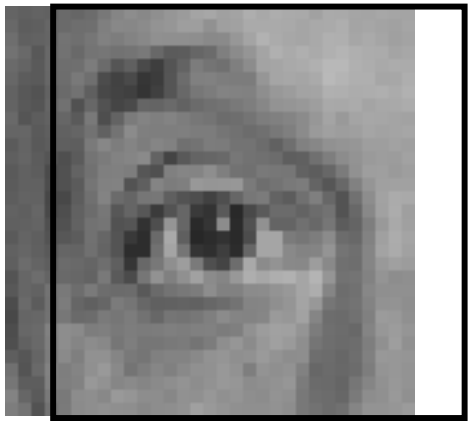
?

shift



original

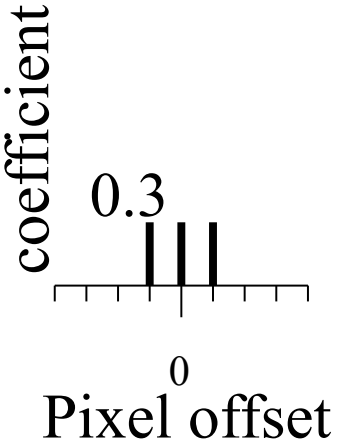




shifted

Linear filtering

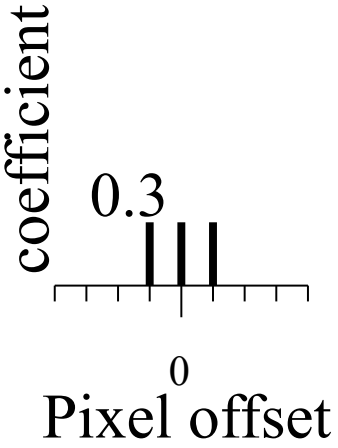
original



?

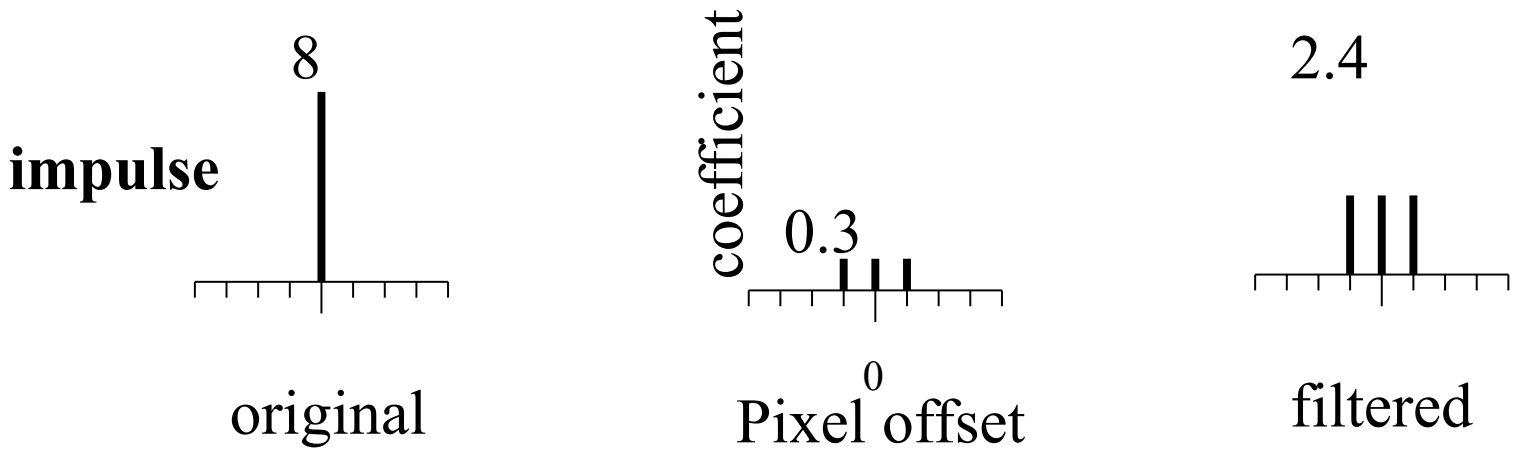
Blurring

original

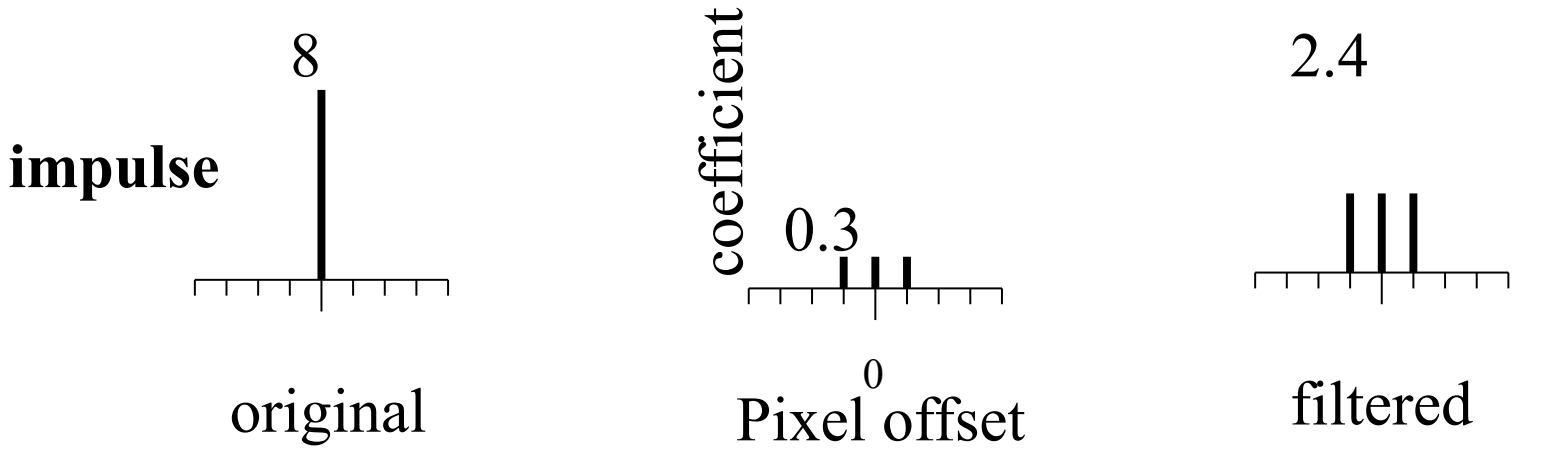


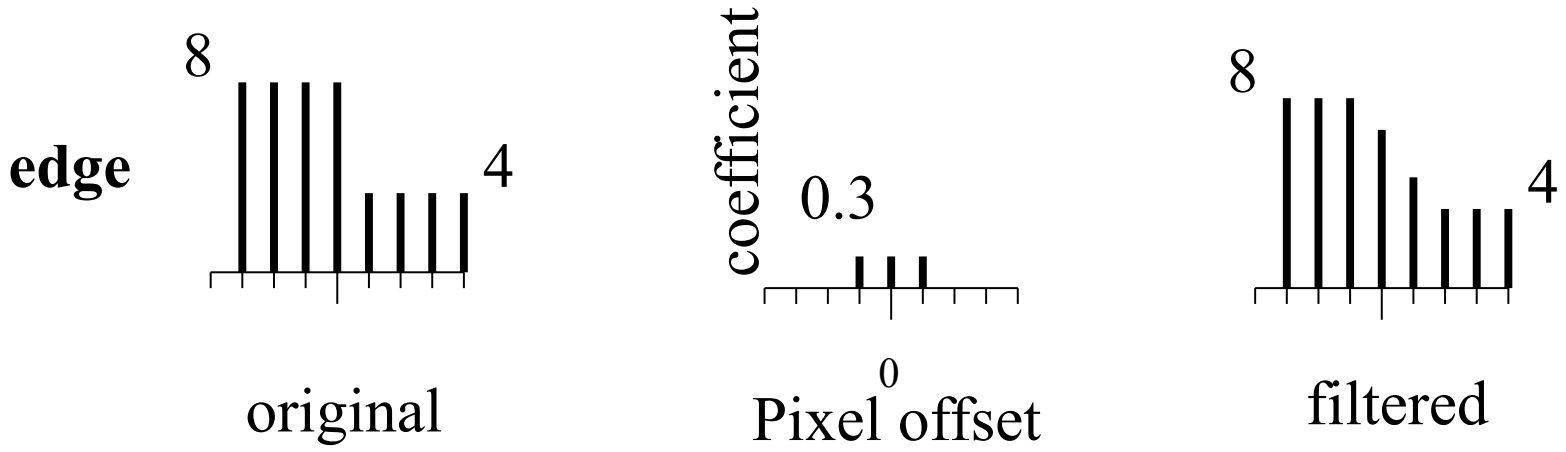
Blurred (filter applied in both dimensions).

Blur examples



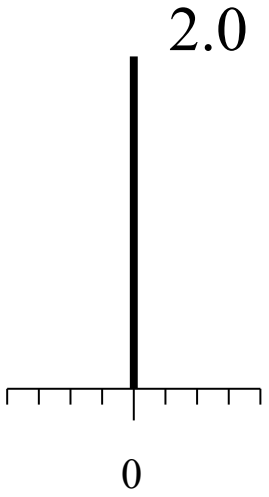
Blur examples

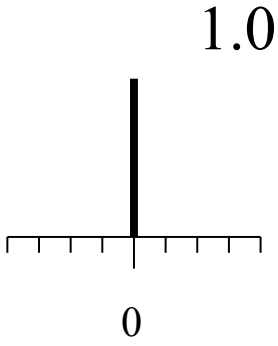




Linear filtering (warm-up slide)

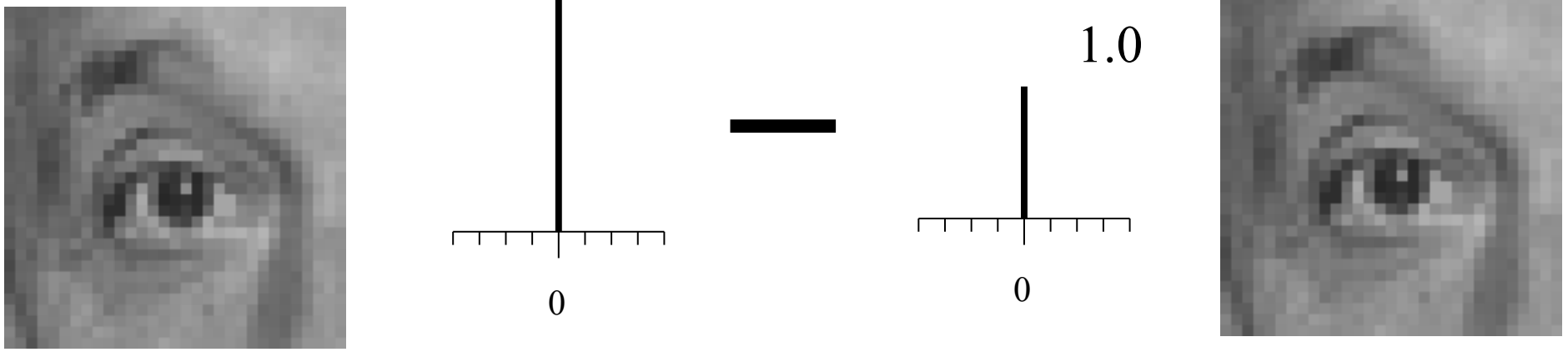
original





?

Linear filtering (no change)

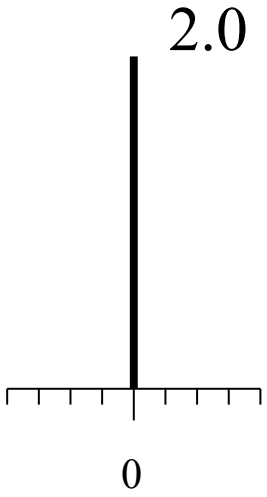


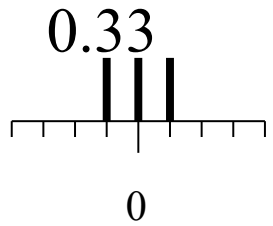
original

Filtered
(no change)

Linear filtering

original

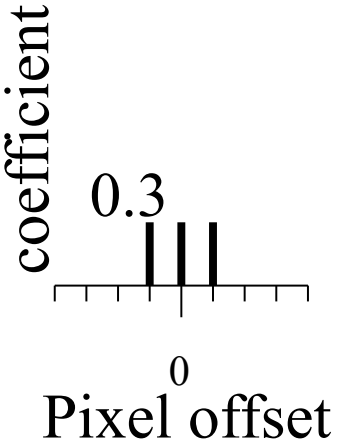




?

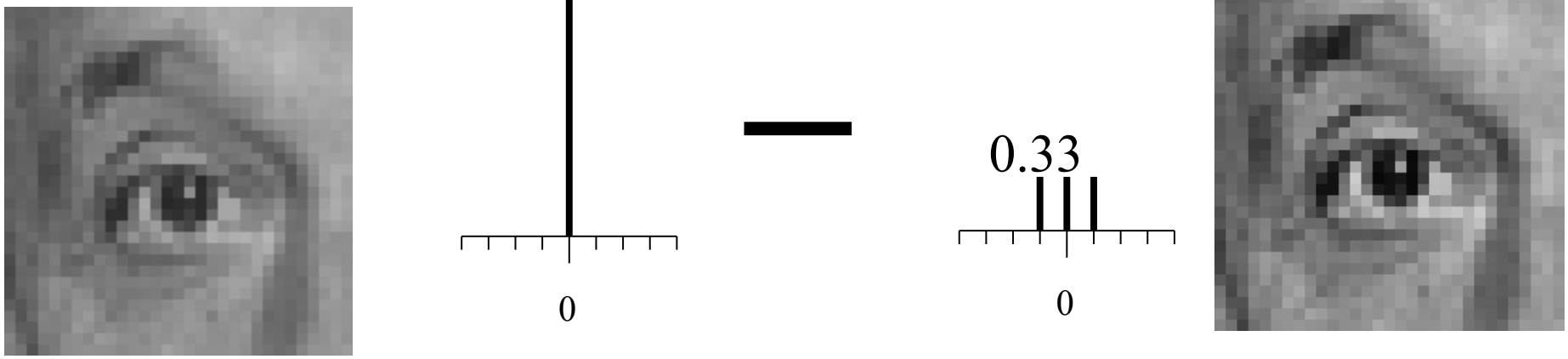
(remember blurring)

original



Blurred (filter applied in both dimensions).

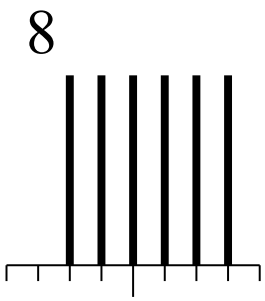
Sharpening



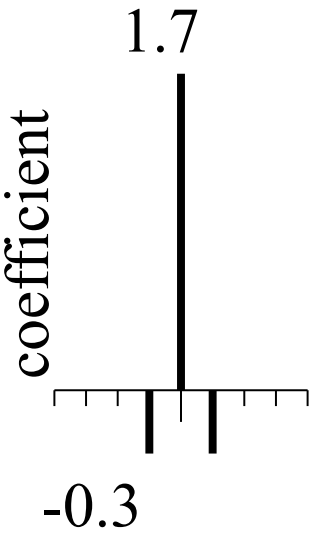
original

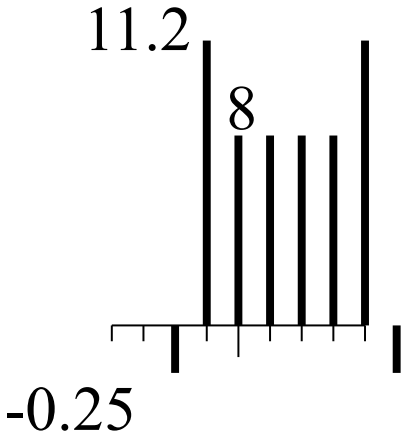
Sharpened original

Sharpening example



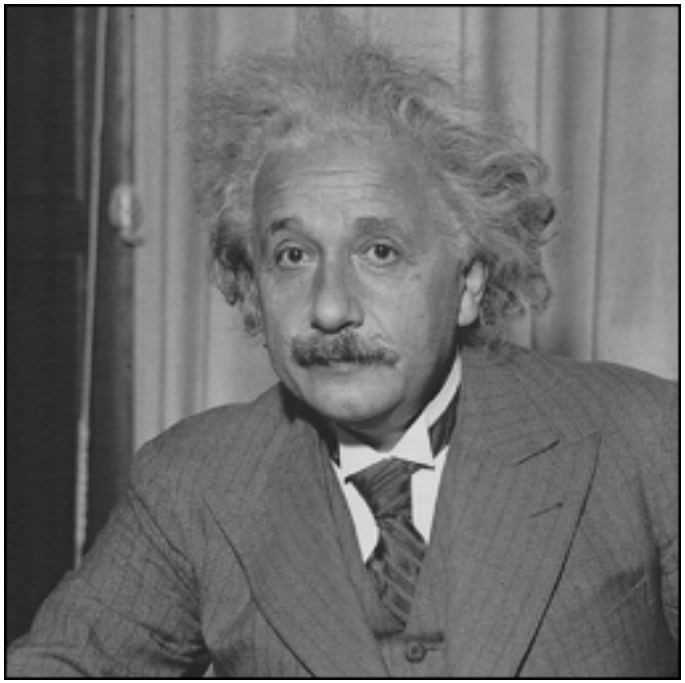
original



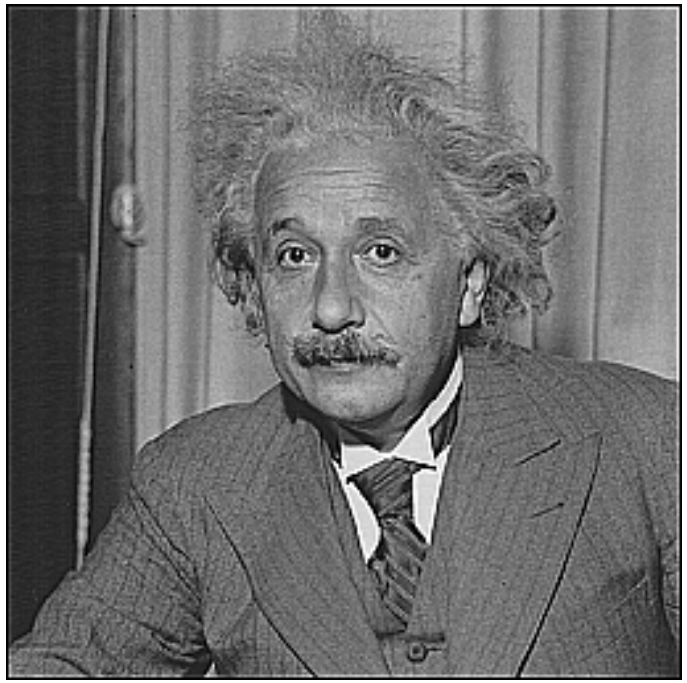


Sharpened (differences are accentuated; constant areas are left untouched).

Sharpening



before



after

Consider now the following filters

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

From left to right: a) A photograph, b) the photograph filtered with the Sobel operator $\mathbf{G}_x$, which emphasizes vertical edges, c) the photograph filtered with the Sobel operator $\mathbf{G}_y$, which emphasizes horizontal edges, d) the magnitue of the response of the pairs of filters at each spatial location, (but with the intensity flipped so that larger values are darker).

Réseaux de neurones convolutifs

Voir aussi la documentation de MatConvNet sur le sujet disponible sur moodle

Convolutional neural networks

Convolutional neural networks (CNNs)

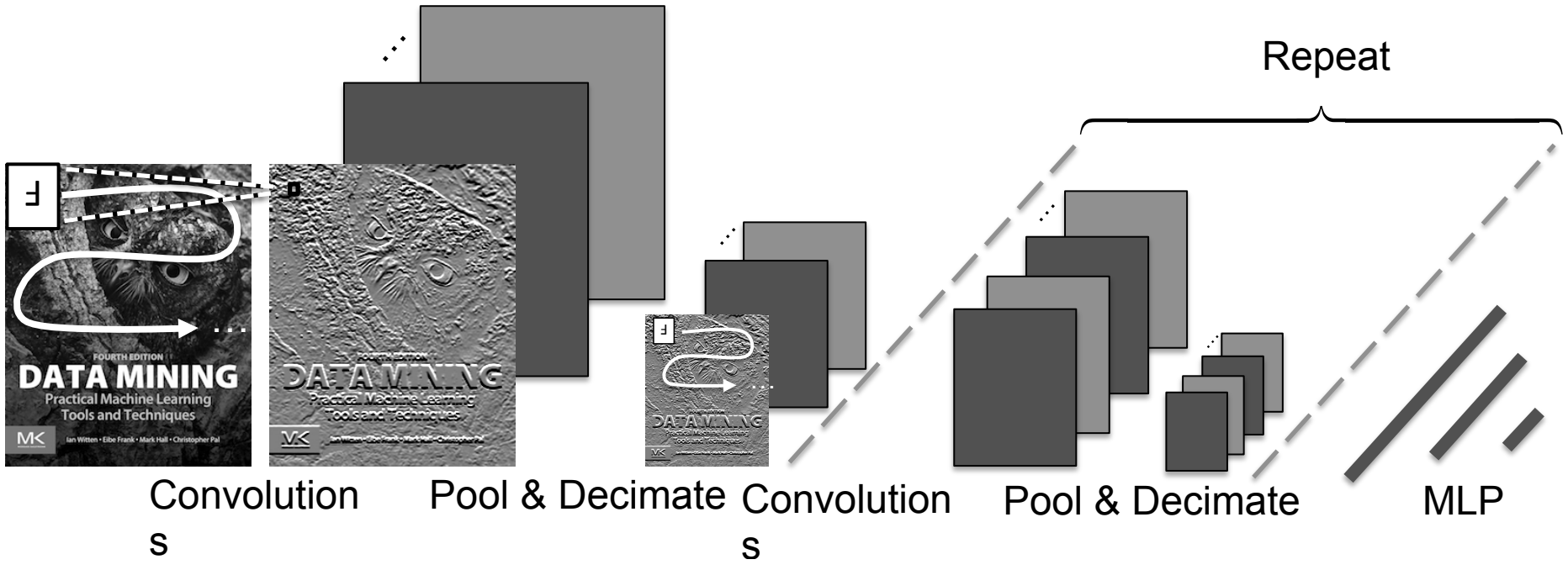
- Are a special kind of feedforward network that has proven *extremely* successful for image analysis
- Imagine filtering an image to detect edges, one could think of edges as a useful set of spatially organized 'features'
- Imagine now if one could learn many such filters jointly along with other parameters of a neural network on top
- Each filter can be implemented by multiplying a relatively small spatial zone of the image by a set of weights and feeding the result to an activation function – just like those discussed above for vanilla feedforward networks
- Because this filtering operation is simply repeated around the image using the same weights, it can be implemented using convolution operations
- The result is a CNN for which it is possible to learn both the filters and the classifier using SGD and the backpropagation algorithm

Deep CNNs

- In a convolutional neural network, once an image has been filtered by several learnable filters, each filter bank's output is often aggregated across a small spatial region, using the average or maximum value.
- Aggregation can be performed within non-overlapping regions, or using subsampling, yielding a lower-resolution layer of spatially organized features—a process that is sometimes referred to as "decimation"
- This gives the model a degree of invariance to small differences as to exactly where a feature has been detected.
- If aggregation uses the max operation, a feature is activated if it is detected anywhere in the pooling zone
- The result can be filtered and aggregated again

A typical CNN architecture

- Many feature maps are obtained from convolving learnable filters across an image
- Results are aggregated or pooled & decimated
- Process repeats until last set of feature maps are given to an MLP for final prediction



CNNs in practice

- LeNet and AlexNet architectures are canonical models
- While CNNs are designed to have a certain degree of translational invariance, augmenting data through global synthetic transformations like the cropping trick can increase performance significantly
- CNNs are usually optimized using mini-batch-based stochastic gradient descent, so practical discussions above about learning deep networks apply
- The use of GPU computing is typically *essential* to accelerate convolution operations significantly
- Resource issues related to the amount of CPU vs. GPU memory available are often important to consider

The ImageNet challenge

- Crucial in demonstrating the effectiveness of deep CNNs
- Problem: recognize object categories in Internet imagery
- The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) classification task - classify image from Flickr and other search engines into 1 of 1000 possible object categories
- Serves as a standard benchmark for deep learning
- The imagery was hand-labeled based on the presence or absence of an object belonging to these categories
- There are 1.2 million images in the training set with 732-1300 training images available per class
- A random subset of 50,000 images was used as the validation set, and 100,000 images were used for the test set where there are 50 and 100 images per class respectively

A plateau, then rapid advances

- "Top-5 error" is the % of times that the target label does not appear among the 5 highest-probability predictions
- Visual recognition methods not based on deep CNNs hit a plateau in performance at 25%

Name	Layers	Top-5 Error (%)	References
AlexNet	8	15.3	Krizhevsky et al. (2012)
VGG Net	19	7.3	Simonyan and Zisserman (2014)
ResNet	152	3.6	He et al. (2016)

- Note: the performance for human agreement has been measured at 5.1% top-5 error
- Smaller filters have been found to lead to superior results in deep networks: the methods with 19 and 152 layers use filters of size 3×3

Starting simply: image filtering

- When an image is filtered, the output can be thought of as another image that contains the filter's response at each spatial location
- Consider filtering a 1D vector **x** by multiplication with a matrix **W** that has a special structure, such as

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \begin{bmatrix} w_1 & w_2 & w_3 & & & & \\ & w_1 & w_2 & w_3 & & & \\ & & & \ddots & & & \\ & & & w_1 & w_2 & w_3 & \end{bmatrix}$$

where the elements left blank in the matrix above are zero and we have used a simple filter having only three non-zero coefficients and a "stride" of one

Correlation and convolution

- Suppose our filter is centered, giving the first vector element an index of –1, or an index of –*K*, where *K* is the "radius" of the filter, then 1D filtering can be written

$$\mathbf{y}[n] = \sum_{k=-K}^{K} \mathbf{w}[k]\mathbf{x}[n+k]$$

- Directly generalizing this filtering to a 2D image **X** and filter **W** gives the *cross-correlation*, **Y**=**W**★**X**, for which the result for row *r* and column *c* is

$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[j,k]\mathbf{X}[r+j,c+k]$$

- The *convolution* of an image with a filter, **Y**=**W***__X__ , is obtained by simply flipping the sense of the filter

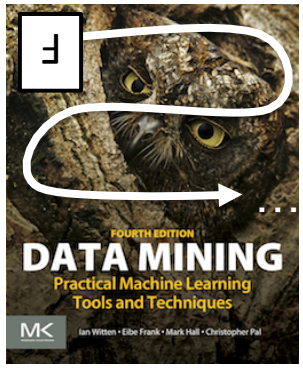
$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[-j,-k]\mathbf{X}[r+j,c+k]$$

Simple filtering example

- Ex. consider the task of detecting edges in an image
- A well known technique is to filter an image with so-called "Sobel" filters, which involves convolving it with

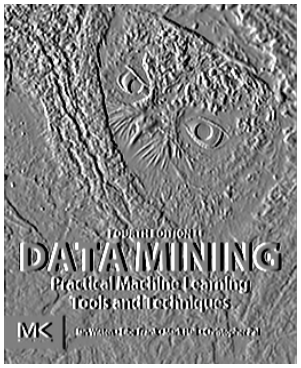
$$\mathbf{W}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \mathbf{W}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- Applied to the image X below, we have:

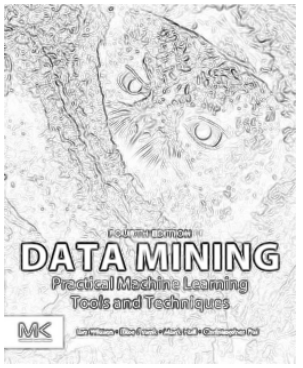


$\mathbf{X}$

$\mathbf{G}_x = \mathbf{W}_x * \mathbf{X}$



$\mathbf{G}_y = \mathbf{W}_y * \mathbf{X}$



$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$

Visualizing the filters learned by a CNN

- Learned edge-like filters and texture-like filters are frequently observed in the early layers of CNNs trained using natural images
- Since each layer in a CNN involves filtering the feature map below, so as one moves up the receptive fields become larger
- Higher- level layers learn to detect larger features, which often correspond to textures, then small pieces of objects

First Layer

Second Layer

Third Layer

(Imagery kindly provided by Matthew Zeiler)

- Above are the strongest activations of random neurons projecting the activation back into image space using the deconvolution approach of Zeiler and Fergus

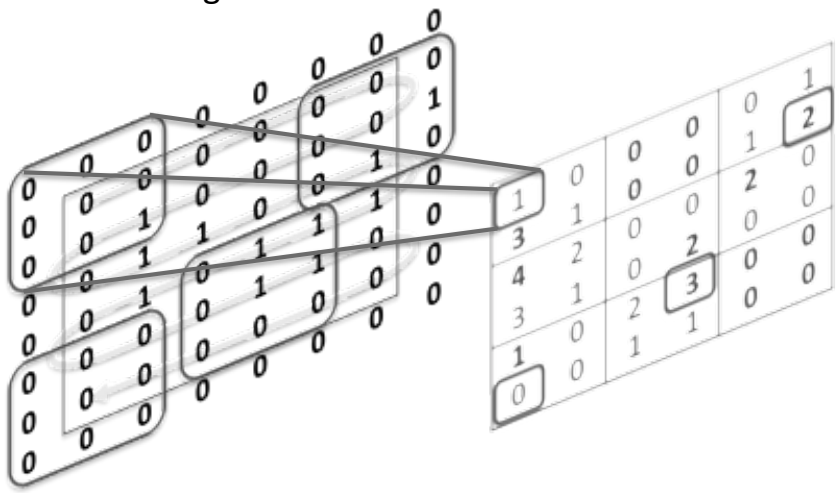
Simple example of: convolution, pooling, and decimation operations

Filter

-1	0	1
-2	0	2
-1	0	1

Image

Convolution



Max Pooling

3	3	0	0	2	2
3	3	0	0	2	2
4	4	2	2	2	2
4	4	2	2	2	2
1	1	3	3	0	0
1	1	3	3	0	0

Decimation

3	0	2
4	2	2
1	3	0

- An image is convolved with a filter; curved rectangular regions in the first large matrix depict a random set of image locations
- Maximum values within small 2×2 regions are indicated in bold in the central matrix
- The results are pooled, using max-pooling then decimated by a factor of two, to yield the final matrix

Understanding Convolution versus Deconvolution

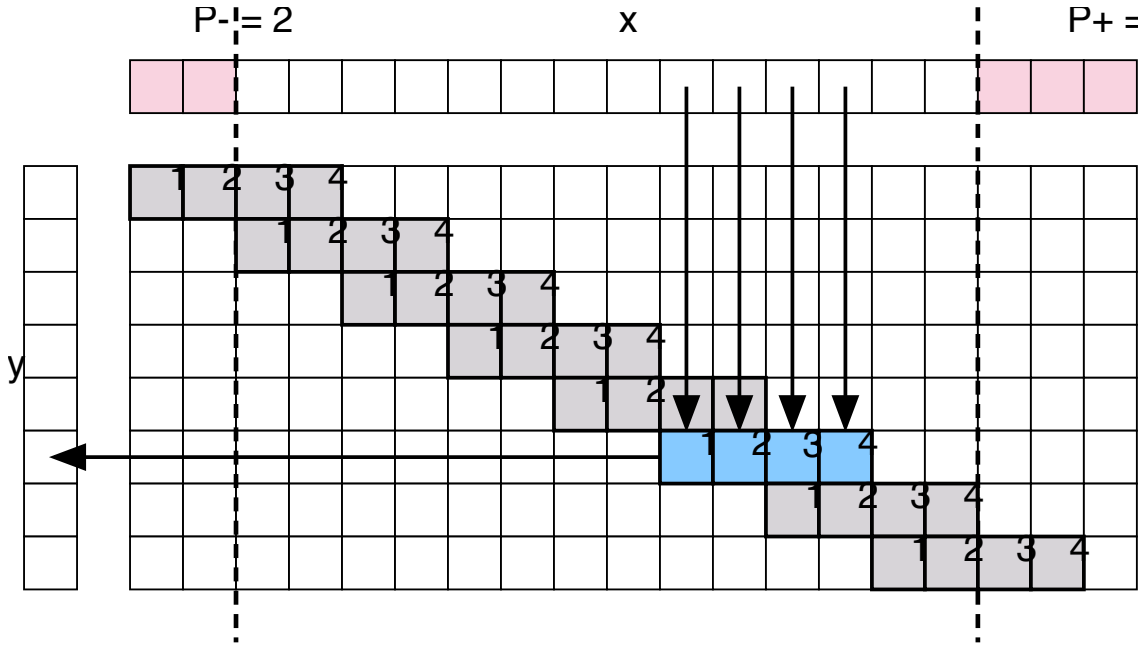


Figure 4.1: **Convolution.** The figure illustrates the process of filtering a 1D signal $\mathbf{x}$ by a filter f to obtain a signal $\mathbf{y}$. The filter has $H' = 4$ elements and is applied with a stride of $S_h = 2$ samples. The purple areas represented padding $P_- = 2$ and $P_+ = 3$ which is zero-filled. Filters are applied in a sliding-window manner across the input signal. The samples of $\mathbf{x}$ involved in the calculation of a sample of $\mathbf{y}$ are shown with arrow. Note that the rightmost sample of $\mathbf{x}$ is never processed by any filter application due to the sampling step. While in this case the sample is in the padded region, this can happen also without padding.

Figure from the MatConvNet documentation.

Convolution Transpose

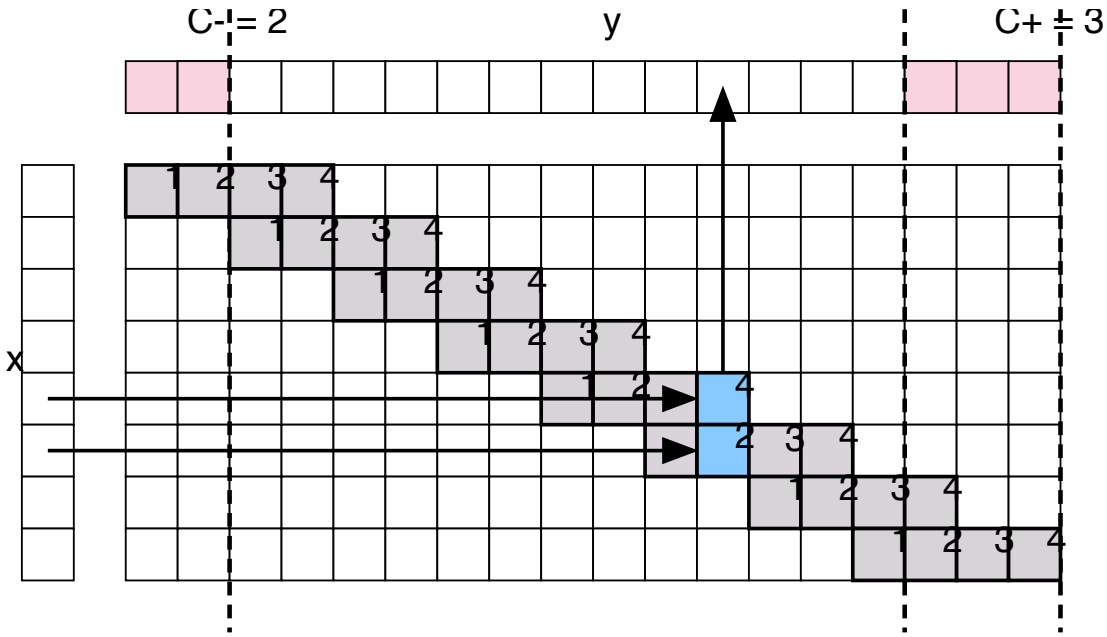


Figure 4.2: **Convolution transpose.** The figure illustrates the process of filtering a 1D signal x by a filter f to obtain a signal y. The filter is applied as a sliding-window, forming a pattern which is the transpose of the one of fig. 4.1. The filter has $H' = 4$ samples in total, although each filter application uses two of them (blue squares) in a circulant manner. The purple areas represent crops with $C_- = 2$ and $C_+ = 3$ which are discarded. The arrows exemplify which samples of x are involved in the calculation of a particular sample of y. Note that, differently from the forward convolution fig. 4.1, there is no need to add padding to the input array; instead, the convolution transpose filters can be seen as being applied with maximum input padding (more would result in zero output values), and the latter can be reduced by cropping the output instead.

Figure from the MatConvNet documentation.

MatConvNet Documentation

2.3.1 Derivatives of tensor functions

In a CNN, a layer is a function $\mathbf{y} = f(\mathbf{x})$ where both input $\mathbf{x} \in \mathbb{R}^{H\times W\times C}$ and output $\mathbf{y} \in \mathbb{R}^{H'\times W'\times C'}$ are tensors. The derivative of the function f contains the derivative of each output component $y_{i'j'k'}$ with respect to each input component x_{ijk}, for a total of $H' \times W' \times C' \times H \times W \times C$ elements naturally arranged in a 6D tensor. Instead of expressing derivatives as tensors, it is often useful to switch to a matrix notation by *stacking* the input and output tensors into vectors. This is done by the vec operator, which visits each element of a tensor in lexicographical order and produces a vector:

$$\operatorname{vec}\mathbf{x} = \begin{bmatrix} x_{111} \\ x_{211} \\ \vdots \\ x_{H11} \\ x_{121} \\ \vdots \\ x_{HWC} \end{bmatrix}.$$

MatConvNet Documentation

By stacking both input and output, each layer f can be seen reinterpreted as vector function $\operatorname{vec} f$, whose derivative is the conventional Jacobian matrix:

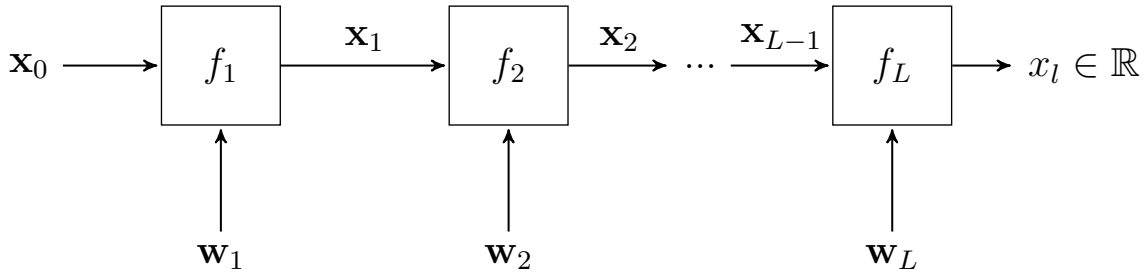
$$\frac{d \operatorname{vec} f}{d(\operatorname{vec} \mathbf{x})^\top} = \begin{bmatrix} \frac{\partial y_{111}}{\partial x_{111}} & \frac{\partial y_{111}}{\partial x_{211}} & \cdots & \frac{\partial y_{111}}{\partial x_{H11}} & \frac{\partial y_{111}}{\partial x_{121}} & \cdots & \frac{\partial y_{111}}{\partial x_{HWC}} \\ \frac{\partial y_{211}}{\partial x_{111}} & \frac{\partial y_{211}}{\partial x_{211}} & \cdots & \frac{\partial y_{211}}{\partial x_{H11}} & \frac{\partial y_{211}}{\partial x_{121}} & \cdots & \frac{\partial y_{211}}{\partial x_{HWC}} \\ \vdots & \vdots & \ldots & \vdots & \vdots & \ldots & \vdots \\ \frac{\partial y_{H'11}}{\partial x_{111}} & \frac{\partial y_{H'11}}{\partial x_{211}} & \cdots & \frac{\partial y_{H'11}}{\partial x_{H11}} & \frac{\partial y_{H'11}}{\partial x_{121}} & \cdots & \frac{\partial y_{H'11}}{\partial x_{HWC}} \\ \frac{\partial y_{121}}{\partial x_{111}} & \frac{\partial y_{121}}{\partial x_{211}} & \cdots & \frac{\partial y_{121}}{\partial x_{H11}} & \frac{\partial y_{121}}{\partial x_{121}} & \cdots & \frac{\partial y_{121}}{\partial x_{HWC}} \\ \vdots & \vdots & \ldots & \vdots & \vdots & \ldots & \vdots \\ \frac{\partial y_{H'W'C'}}{\partial x_{111}} & \frac{\partial y_{H'W'C'}}{\partial x_{211}} & \cdots & \frac{\partial y_{H'W'C'}}{\partial x_{H11}} & \frac{\partial y_{H'W'C'}}{\partial x_{121}} & \cdots & \frac{\partial y_{H'W'C'}}{\partial x_{HWC}} \end{bmatrix}.$$

This notation for the derivatives of tensor functions is taken from [7] and is used throughout this document.

MatConvNet Documentation

2.3.2 Derivatives of function compositions

In order to understand backpropagation, consider first a simple CNN terminating in a loss function $f_L = \ell_{\mathbf{y}}$:



The goal is to compute the gradient of the loss value x_L (output) with respect to each network parameter $\mathbf{w}_l$:

$$\frac{df}{d(\operatorname{vec}\mathbf{w}_l)^\top} = \frac{d}{d(\operatorname{vec}\mathbf{w}_l)^\top}\left[f_L(\cdot;\mathbf{w}_L)\circ ... \circ f_2(\cdot;\mathbf{w}_2)\circ f_1(\mathbf{x}_0;\mathbf{w}_1)\right].$$

By applying the chain rule and by using the matrix notation introduced above, the derivative can be written as

$$\frac{df}{d(\operatorname{vec}\mathbf{w}_l)^\top} = \frac{d\operatorname{vec} f_L(\mathbf{x}_{L-1};\mathbf{w}_L)}{d(\operatorname{vec}\mathbf{x}_{L-1})^\top} \times \cdots \times \frac{d\operatorname{vec} f_{l+1}(\mathbf{x}_l;\mathbf{w}_{l+1})}{d(\operatorname{vec}\mathbf{x}_l)^\top} \times \frac{d\operatorname{vec} f_l(\mathbf{x}_{l-1};\mathbf{w}_l)}{d(\operatorname{vec}\mathbf{w}_l^\top)} \tag{2.1}$$

where the derivatives are computed at the working point determined by the input $\mathbf{x}_0$ and the current value of the parameters.

Convolutional layers and gradients

- Let's consider how to compute the gradients needed to optimize a convolutional network
- At a given layer we have i=1…$N^{(l)}$ feature filters and corresponding feature maps
- The convolutional kernel matrices $\mathbf{K}_i$ contain flipped weights with respect to kernel weight matrices $\mathbf{W}_i$
- With activation function act(), and for each feature type i, a scaling factor g_i and bias matrix $\mathbf{B}_i$, the feature maps are matrices $\mathbf{H}_i(\mathbf{A}_i(\mathbf{X}))$ and can be visualized as a set of images given by

$$\mathbf{H}_i = g_i \,\mathrm{act}\left[\mathbf{K}_i * \mathbf{X} + \mathbf{B}_i\right] = g_i \,\mathrm{act}\left[\mathbf{A}_i(\mathbf{X})\right]$$

Convolutional layers and gradients

- The loss is a function of the $N^{(l)}$ feature maps for a given layer, $L = L(\mathbf{H}_1^{(l)},\ldots,\mathbf{H}_{N^{(l)}}^{(l)})$
- Define **h**=vec(**H**), **x**=vec(**X**), **a**=vec(**A**), where the vec() function returns a vector with stacked columns of the given matrix argument,
- Choose an act() function that operates elementwise on an input matrix of pre-activations and has scale parameters of 1 and biases of 0.
- Partial derivatives of hidden layer output with respect to input **X** of the convolutional units are

$$\frac{\partial L}{\partial \mathbf{X}} = \sum_i \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{X}} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} = \sum_i \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}} \frac{\partial \mathbf{h}_i}{\partial \mathbf{a}_i} \frac{\partial L}{\partial \mathbf{h}_i} = \sum_i [\mathbf{W}_i * \mathbf{D}_i], \mathbf{D}_i = dL/\partial \mathbf{A}_i$$

Convolutional layers and gradients

- Matrix $\mathbf{D}_i$ in previous side is a matrix containing the partial derivative of the elementwise act() function's input with respect to its pre-activation value for the i^{th} feature type, organized according to spatial positions given by row *j* and column *k*.
- Intuitively, the result is a sum of the convolution of each of the (zero padded) filters $\mathbf{W}_i$ with an image-like matrix of derivatives $\mathbf{D}_i$.
- The partial derivatives of the hidden layer output are

$$\frac{\partial L}{\partial \mathbf{W}_i} = \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{W}_i} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} = \left[\mathbf{X}^{\dagger} * \mathbf{D}_i \right],$$

- Where $\mathbf{X}^{\dagger}$ is the row & column-flipped version of **X**

Pooling and subsampling layers

- What are the consequences of backpropagating gradients through max or average pooling layers?
- In the former case, the units that are responsible for the maximum within each zone *j*, *k* —the "winning units"— get the backpropagated gradient
- For average pooling, the averaging is simply a special type of convolution with a fixed kernel that computes the (possibly weighted) average of pixels in a zone
  - the required gradients are therefore like std conv. layers
- The subsampling step either samples every n^{th} output, or avoids needless computation by only evaluating every n^{th} pooling computation

Implementing CNNs

- Convolutions are very well suited for acceleration using GPUs
- Since graphics hardware can accelerate convolutions by an *order of magnitude* or more over CPU implementations, they play an important often *critical* role in training CNNs
- An experimental turn-around time of days rather than weeks makes a huge difference to model development times!
- Can also be challenging to construct software for learning a convolutional neural network in such a way that alternative architectures can be explored
- Early GPU implementations were hard to extend, newer tools allow for both fast computation and flexible high-level programming primitives
- Many software tools allow gradient computations and the backpropagation algorithm for large networks to be almost completely automated.

Bibliographic Notes & Further Reading

Convolutional Networks

- Modern convolutional neural networks are widely acknowledged as having their roots with the "neocognitron" proposed by Fukushima (1980); however
- The work of LeCun et al. (1998) on the LeNet convolutional network architecture has been extremely influential.
- The MNIST dataset containing 28×28 pixel images of handwritten digits has been popular in deep learning research community since 1998
- However, it was the ImageNet challenge (Russakovsky et al., 2015), with a variety of much higher resolutions, that catapulted deep learning into the spotlight in 2012.
  - The winning entry from the University of Toronto (Krizhevsky et al. , 2012) processed the images at a resolution of 256×256 pixels.
  - Up till then, CNNs were simply incapable of processing such large volumes of imagery at such high resolutions in a reasonable amount of time.

Bibliographic Notes & Further Reading

Convolutional Networks

- Krizhevsky et al. (2012)'s dramatic ImageNet win used a GPU accelerated convolutional neural networks.
  - This spurred a great deal of development, reflected in rapid subsequent advances in visual recognition performance and on the ImageNet benchmark.
- In the 2014 challenge, the Oxford Visual Geometry Group and a team from Google pushed performance even further using much deeper architectures: 16-19 weight layers for the Oxford group, using tiny 3×3 convolutional filters (Simonyan and Zisserman, 2014); 22 layers, with filters up to 5×5 for the Google team (Szegedy et al., 2015).
- The 2015 ImageNet challenge was won by a team from Microsoft Research Asia (MSRA) using an architecture with 152 layers (He et al., 2015), using tiny 3×3 filters combined with "shortcut" connections that skip over layers, and pooling and decimating the result of multiple layers of small convolution operations

Bibliographic Notes & Further Reading

Convolutional Networks

- Good parameter initialization can be critical for the success of neural networks, as discussed in LeCun et al. (1998)'s classic work and the more recent work of Glorot and Bengio (2010).
- Krizhevsky et al. (2012)'s convolutional network of rectified linear units (ReLUs) initialized weights using 0-mean isotropic Gaussian distributions with a standard deviation of 0.01, and initialized the biases to 1 for most hidden convolutional layers as well as their model's hidden fully connected layers.
- They observed that this initialization accelerated the early phase of learning by providing ReLUs with positive inputs.

A plateau, then rapid advances

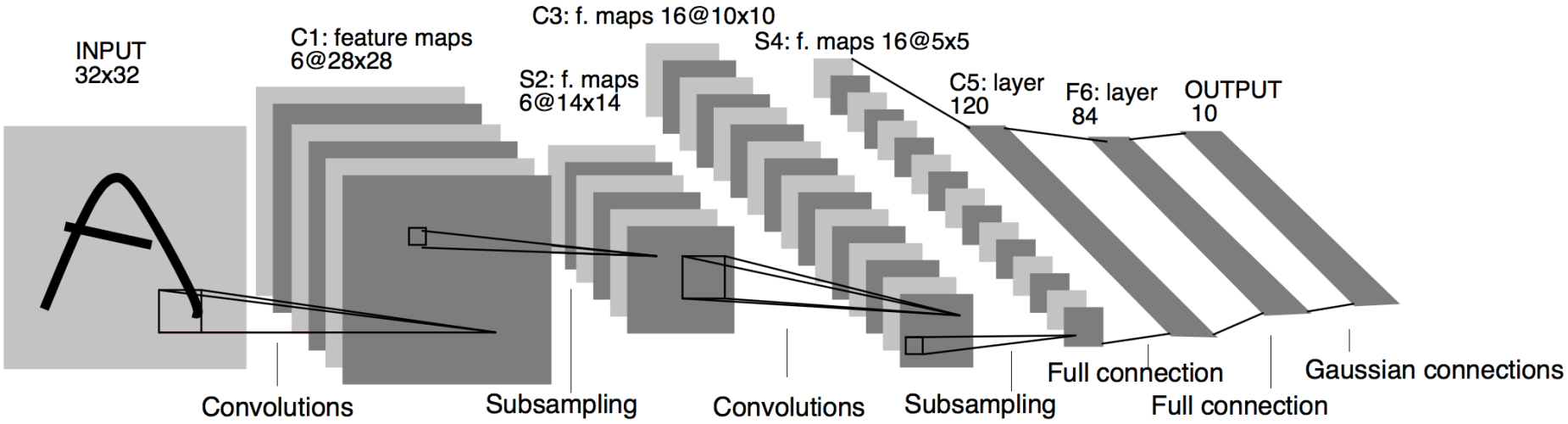
- "Top-5 error" is the % of times that the target label does not appear among the 5 highest-probability predictions
- Visual recognition methods not based on deep CNNs hit a plateau in performance at 25%

Name	Layers	Top-5 Error (%)	References
AlexNet	8	15.3	Krizhevsky et al. (2012)
VGG Net	19	7.3	Simonyan and Zisserman (2014)
ResNet	152	3.6	He et al. (2016)

- Note: the performance for human agreement has been measured at 5.1% top-5 error
- Smaller filters have been found to lead to superior results in deep networks: the methods with 19 and 152 layers use filters of size 3×3

ConvNet Canonical Model Zoo

LeNet-5



LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86, no. 11 (1998): 2278-2324. (Note: 18k citations)

Alex Net

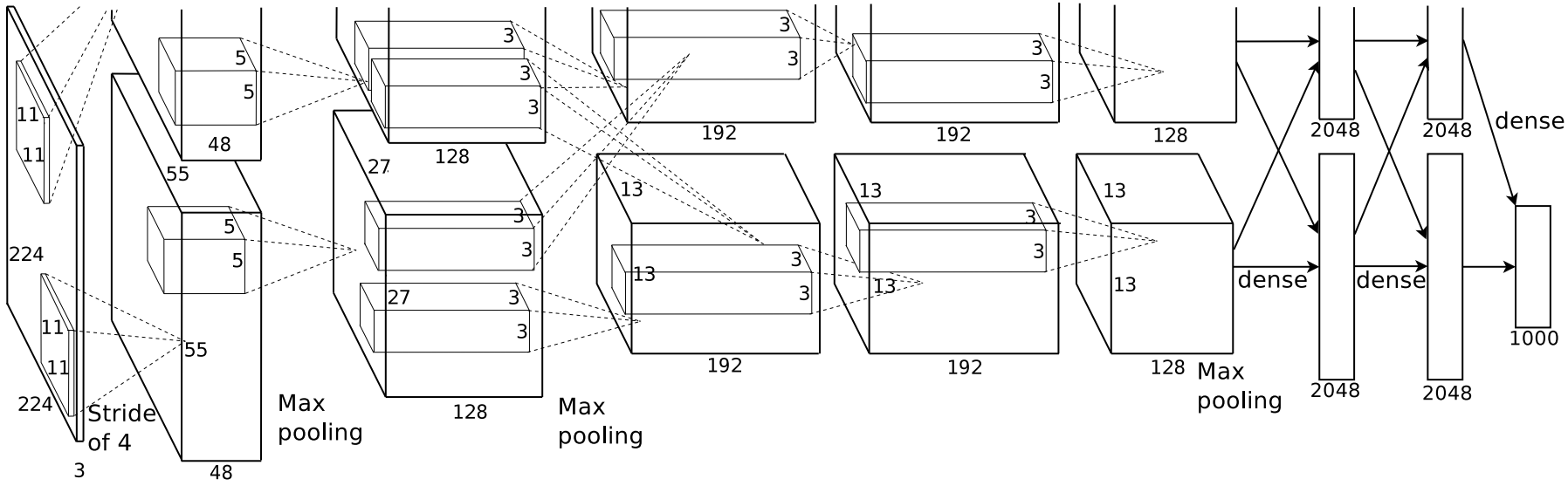
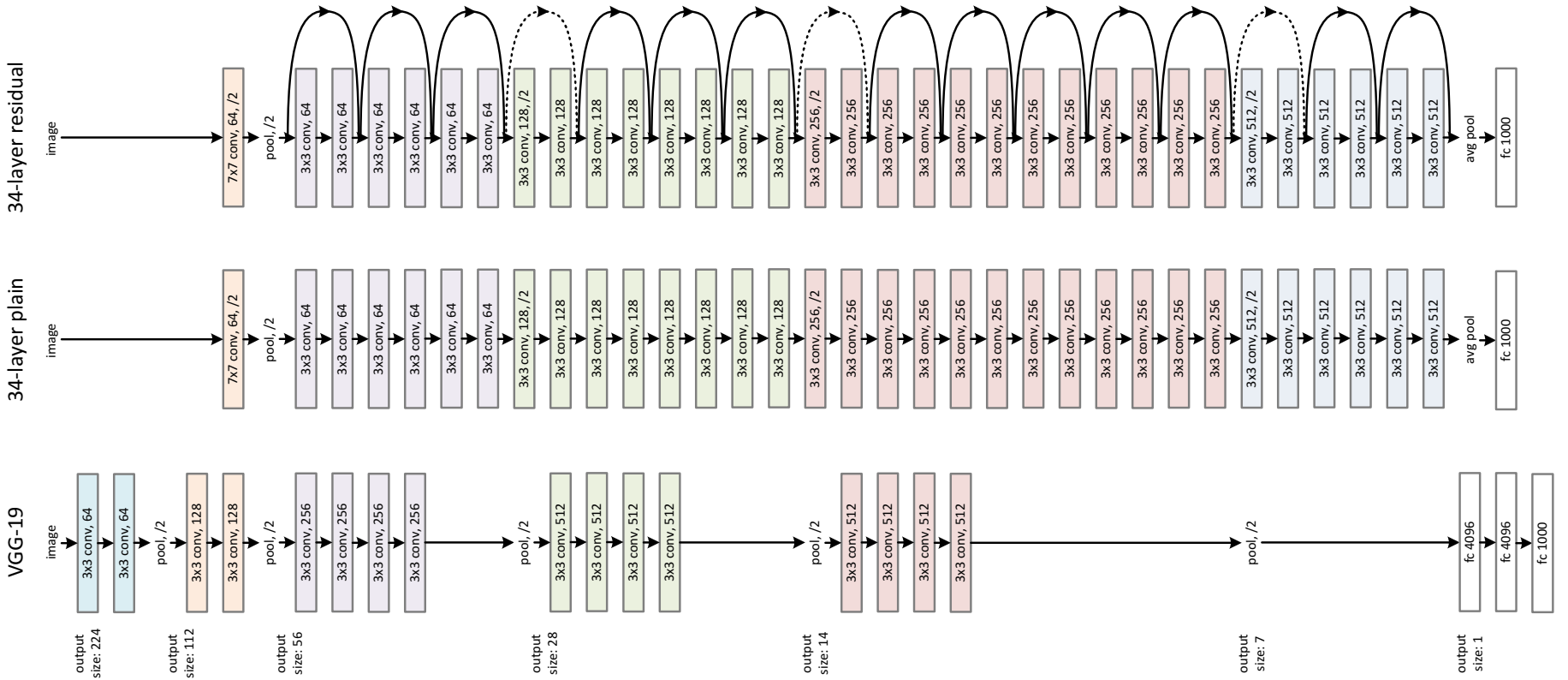


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

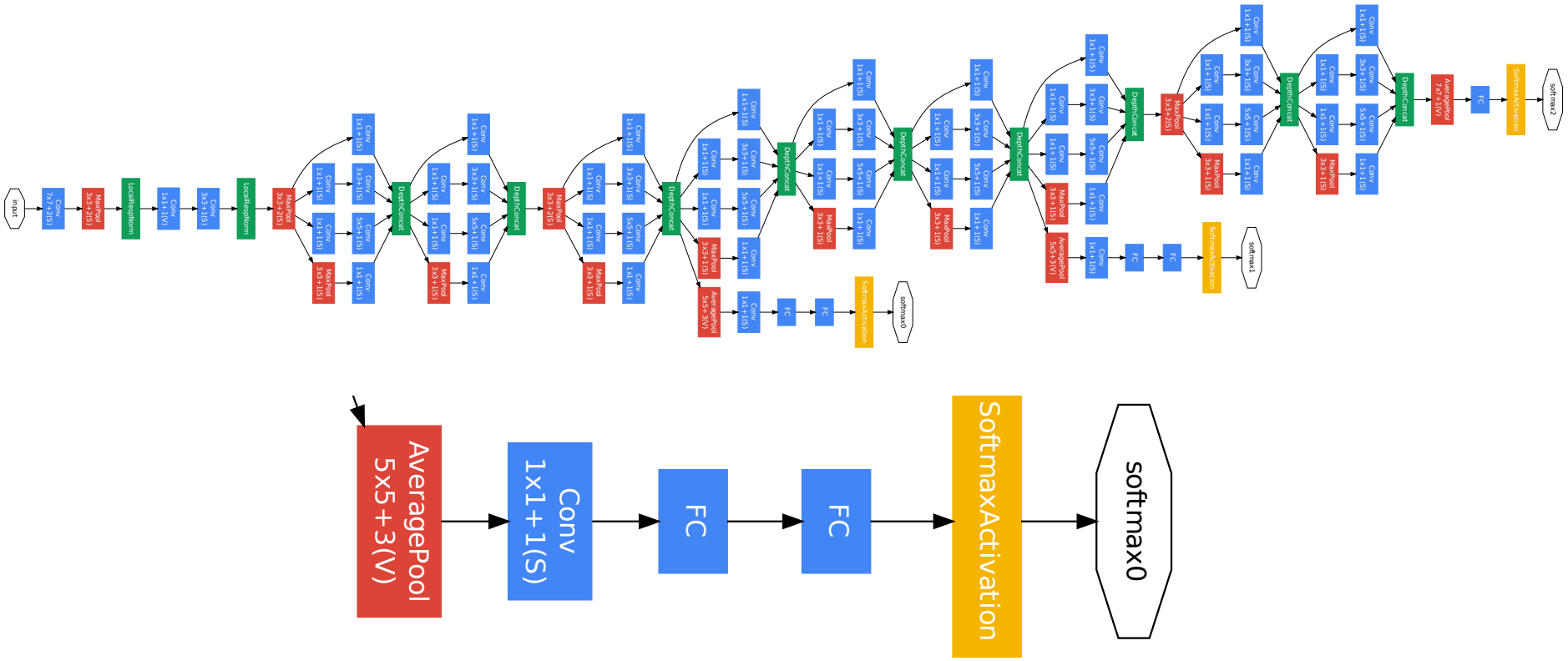
Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012. (Note: 39k citations)

From a VGG-19 to Residual Net 34



- VGG Net: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition

The Inception Architecture



Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1-9. 2015.

ResNets

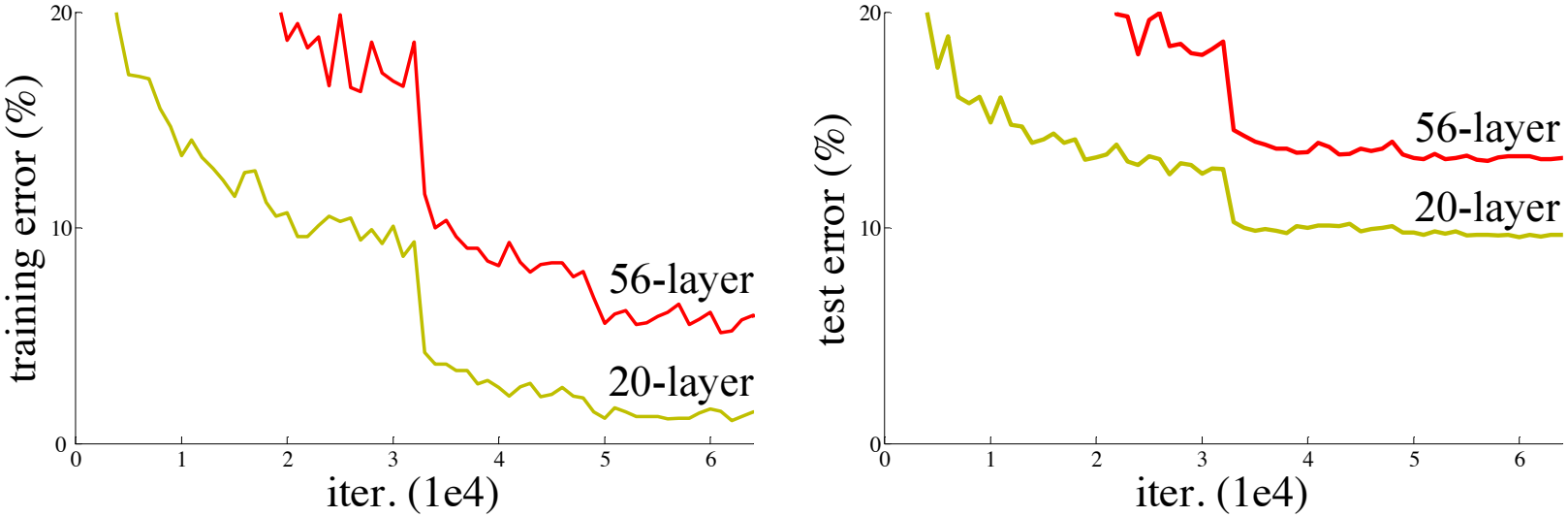


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Key Idea

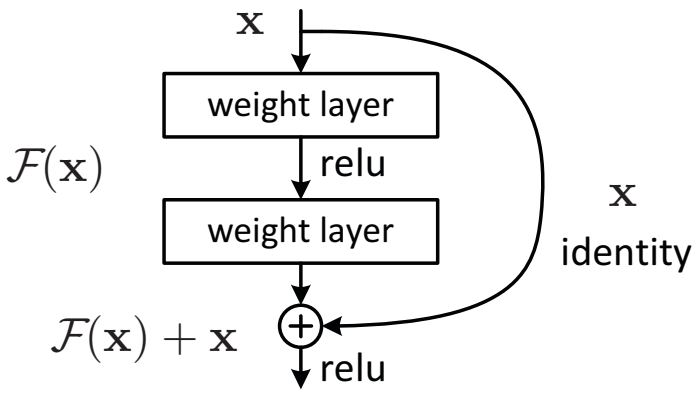


Figure 2. Residual learning: a building block.

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}_i, \mathcal{W}_i),$$

For any deeper unit *L* and any shallower unit *l*

$$\frac{\partial \mathcal{E}}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L}\left(1 + \frac{\partial}{\partial \mathbf{x}_l}\sum_{i=l}^{L-1}\mathcal{F}(\mathbf{x}_i, \mathcal{W}_i)\right).$$

Information directly propagated to layer l from L

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Identity mappings in deep residual networks." In European conference on computer vision, pp. 630-645. Springer, Cham, 2016.

He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European conference on computer vision (pp. 630-645).

ResNets Refined

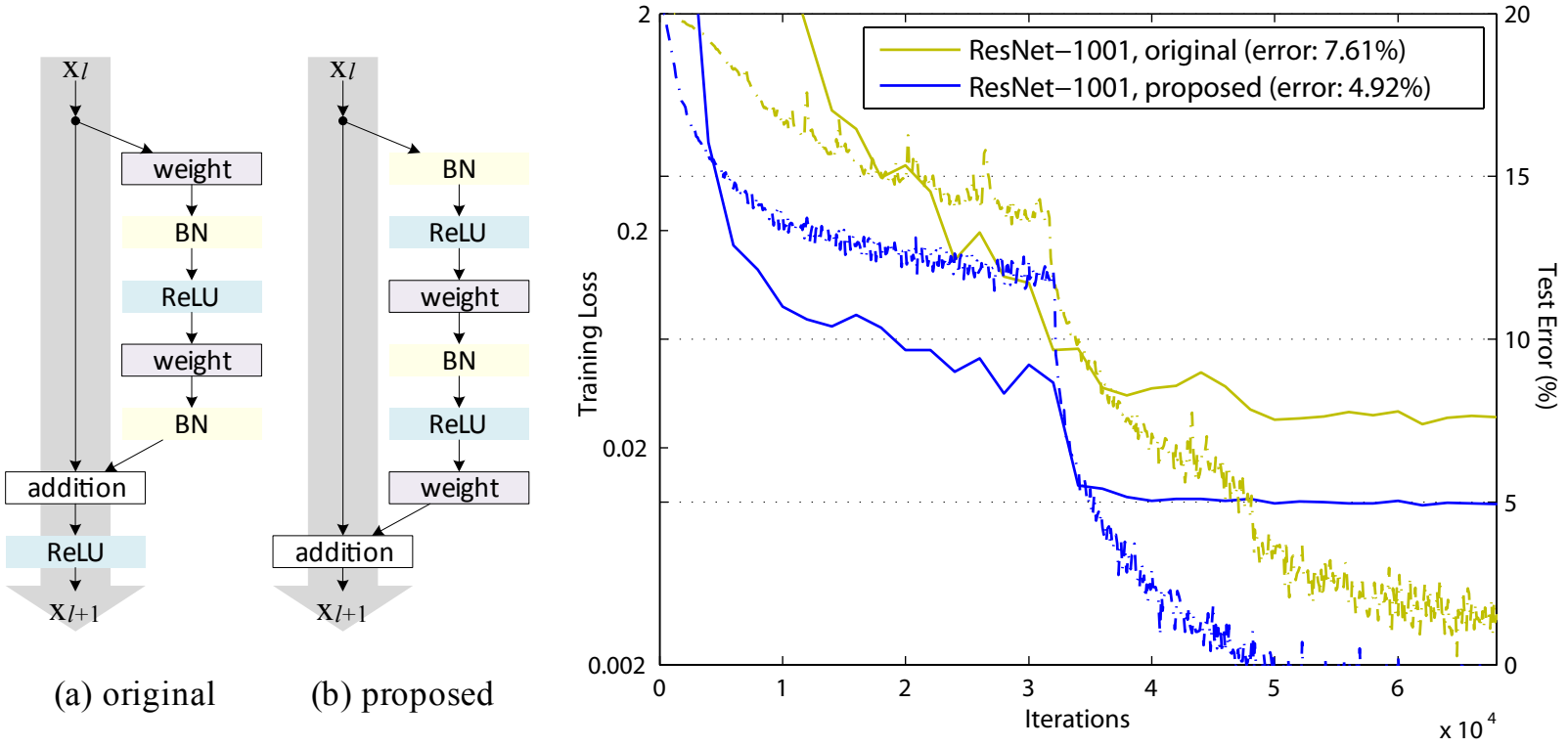


Figure 1. Left: (a) original Residual Unit in [1]; (b) proposed Residual Unit. The grey arrows indicate the easiest paths for the information to propagate, corresponding to the additive term "$\mathbf{x}_l$" in Eqn.(4) (forward propagation) and the additive term "1" in Eqn.(5) (backward propagation). **Right**: training curves on CIFAR-10 of **1001-layer** ResNets. Solid lines denote test error (y-axis on the right), and dashed lines denote training loss (y-axis on the left). The proposed unit makes ResNet-1001 easier to train.

DenseNets

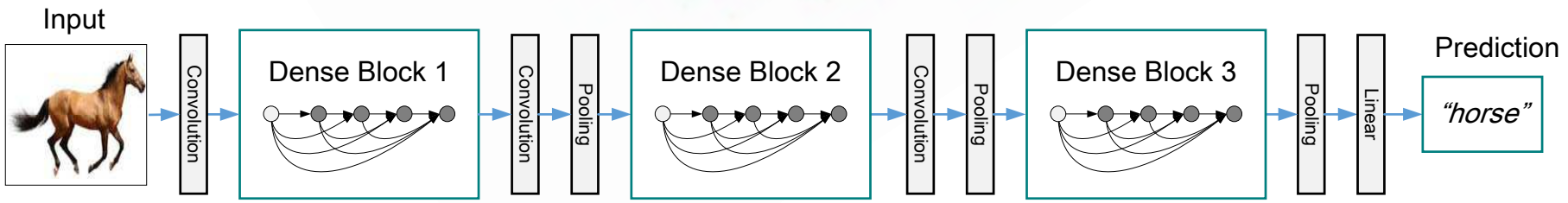
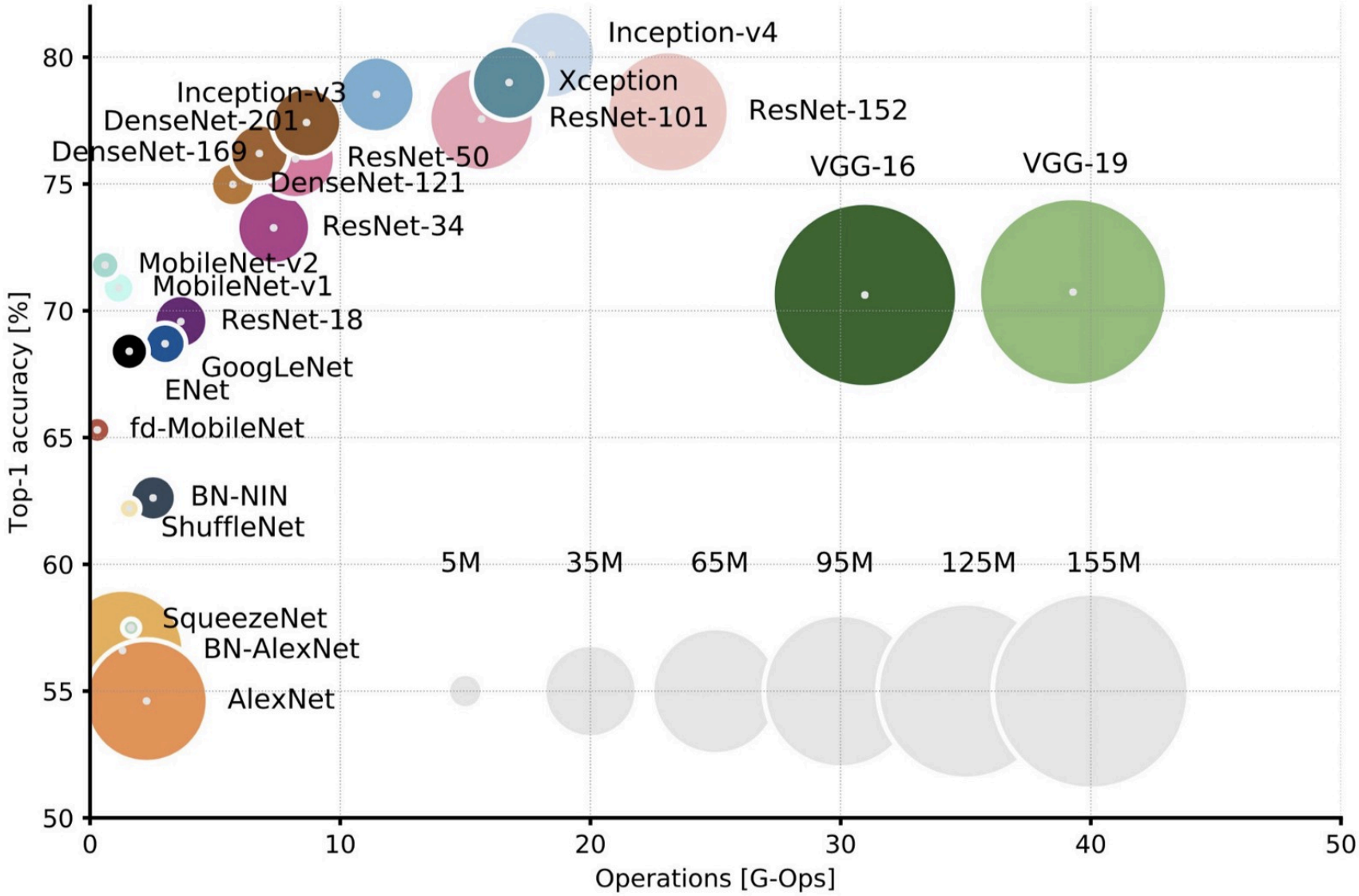


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

Comparing Models



https://towardsdatascience.com/neural-network-architectures-156e5bad51ba

ConvNet Detection Network Model Zoo

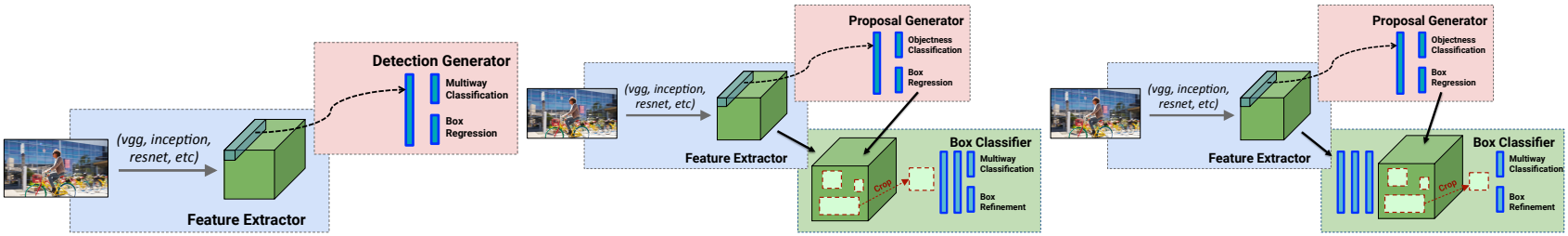
Tensorflow Object Detection API

Creating accurate machine learning models capable of localizing and identifying multiple objects in a single image remains a core challenge in computer vision. The TensorFlow Object Detection API is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. At Google we've certainly found this codebase to be useful for our computer vision needs, and we hope that you will as well.

Comparing bounding box models

Paper	Meta-architecture	Feature Extractor	Matching	Box Encoding $\phi(b_a, a)$	Location Loss functions
Szegedy et al. [40]	SSD	InceptionV3	Bipartite	$[x_0, y_0, x_1, y_1]$	L_2
Redmon et al. [29]	SSD	Custom (GoogLeNet inspired)	Box Center	$[x_c, y_c, \sqrt{w}, \sqrt{h}]$	L_2
Ren et al. [31]	Faster R-CNN	VGG	Argmax	$[\frac{x_c}{w_a}, \frac{y_c}{h_a}, \log w, \log h]$	SmoothL_1
He et al. [13]	Faster R-CNN	ResNet-101	Argmax	$[\frac{x_c}{w_a}, \frac{y_c}{h_a}, \log w, \log h]$	SmoothL_1
Liu et al. [26] (v1)	SSD	InceptionV3	Argmax	$[x_0, y_0, x_1, y_1]$	L_2
Liu et al. [26] (v2, v3)	SSD	VGG	Argmax	$[\frac{x_c}{w_a}, \frac{y_c}{h_a}, \log w, \log h]$	SmoothL_1
Dai et al [6]	R-FCN	ResNet-101	Argmax	$[\frac{x_c}{w_a}, \frac{y_c}{h_a}, \log w, \log h]$	SmoothL_1

Table 1: Convolutional detection models that use one of the meta-architectures described in Section 2. Boxes are encoded with respect to a matching anchor a via a function ϕ (Equation 1), where $[x_0, y_0, x_1, y_1]$ are min/max coordinates of a box, x_c, y_c are its center coordinates, and w, h its width and height. In some cases, w_a, h_a, width and height of the matching anchor are also used. **Notes**: (1) We include an early arXiv version of [26], which used a different configuration from that published at ECCV 2016; (2) [29] uses a fast feature extractor described as being inspired by GoogLeNet [39], which we do not compare to; (3) YOLO matches a groundtruth box to an anchor if its center falls inside the anchor (we refer to this as *BoxCenter*).



(a) SSD. (b) Faster RCNN. (c) R-FCN.

Figure 1: High level diagrams of the detection meta-architectures compared in this paper.

"Speed/accuracy trade-offs for modern convolutional object detectors."
Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, Murphy K, CVPR 2017

Comparing architectures

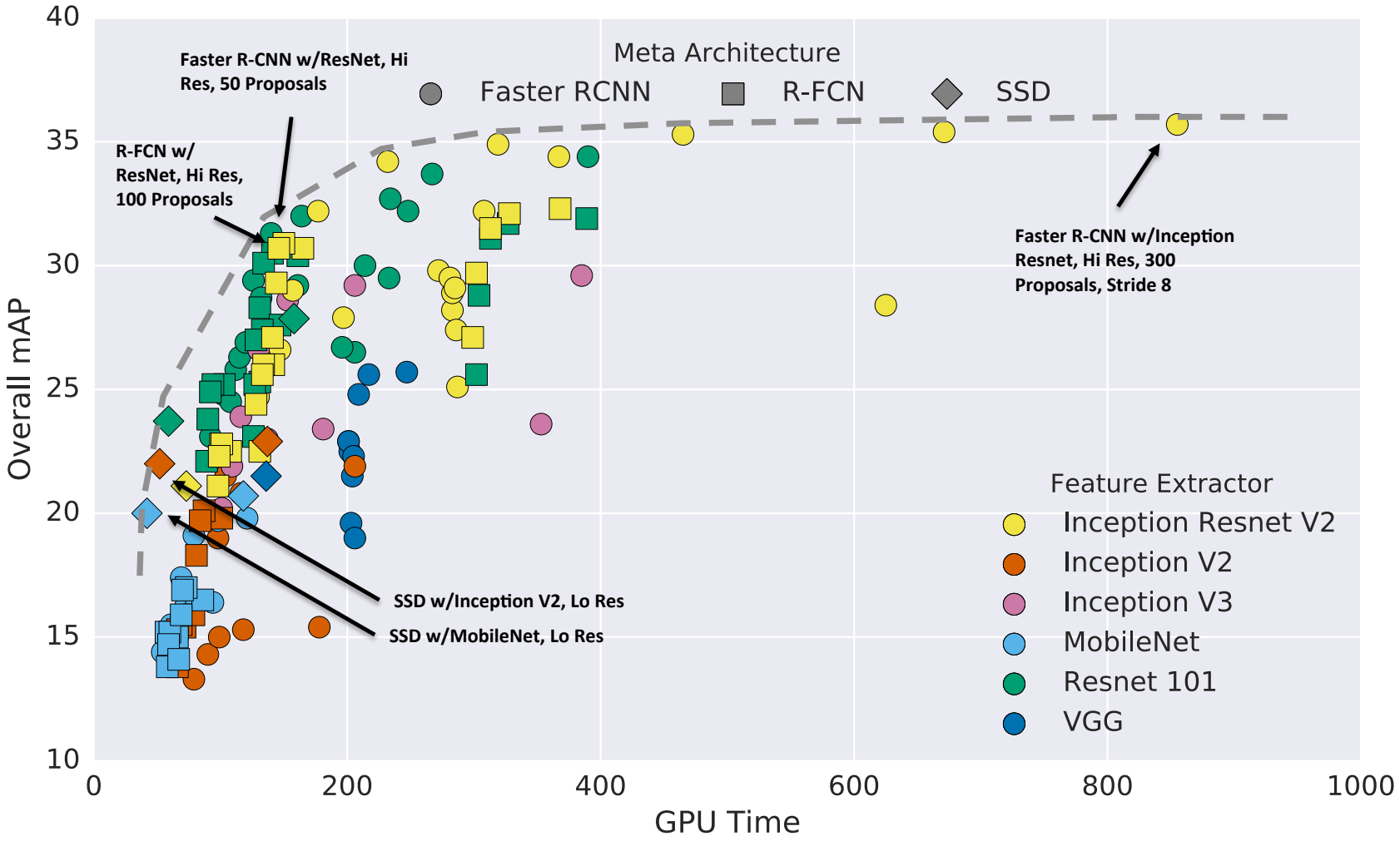
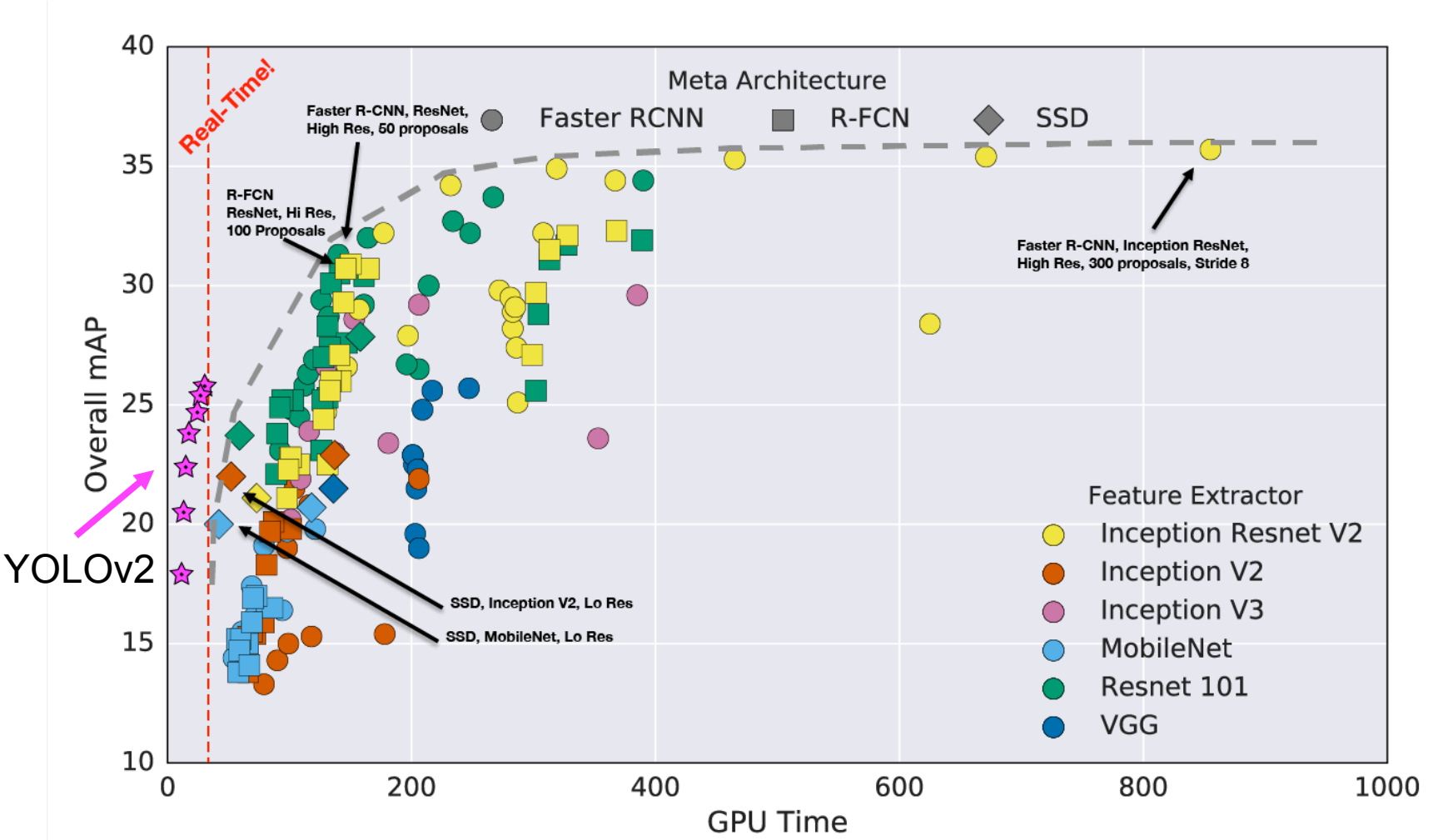


Figure 2: Accuracy vs time, with marker shapes indicating meta-architecture and colors indicating feature extractor. Each (meta-architecture, feature extractor) pair can correspond to multiple points on this plot due to changing input sizes, stride, etc.


JOSEPH REDMON
ALI FARHADI
RETURN IN.....
YOLO9000
Better, Faster,
Stronger
NOW PLAYING IN A DEMO NEAR YOU
@DARKNETFOREVER #YOLO9000
pjreddie.com/yolo
XNOR.AI
YOLO: You Only Look Once
musher
malamute

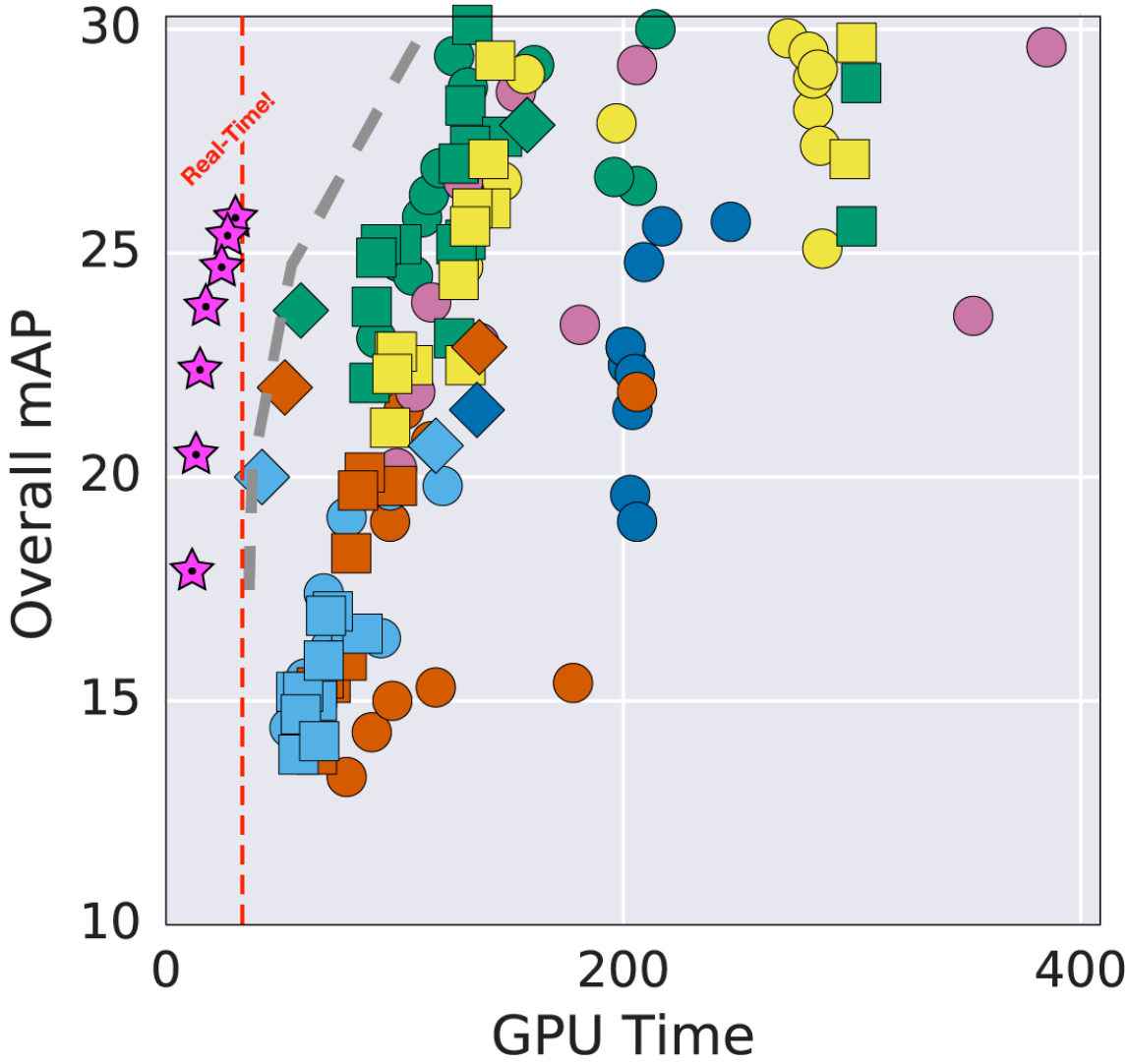
How does YOLO Compare?



Huang, Jonathan, et al. "Speed/accuracy trade-offs for modern convolutional object detectors." *arXiv preprint arXiv:1611.10012* (2016).

Slide from Joseph Redmon's CVPR 2017 talk.

How does YOLO Compare?



Slide from Joseph Redmon's CVPR 2017 talk.

Hidden Variable Models
Theoretical Foundations

Hidden variable models

- Given a model with observations given by $\tilde{x}_i$, a per observation hidden discrete random variable h_i and hidden continuous random variable z_i the marginal likelihood of such a model is given by

$$\mathrm{L}(\theta; \tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n) = \prod_{i=1}^{n} p(\tilde{x}_i; \theta) = \prod_{i=1}^{n} \int_{z_i} \sum_{h_i} p(\tilde{x}_i, z_i, h_i; \theta) dz_i$$

  where the sum is taken over all possible discrete values of h_i & the integral is taken over the entire domain of z_i

- To be extra clear about different quantities here we use $p(x_i = \tilde{x}_i) = p(\tilde{x}_i)$ to denote the probability of the random variable x_i associated with instance *i* taking the value represented by the observation $\tilde{x}_i$

Learning hidden variable models

- If a model has hidden variables we perform learning by integrating out the uncertainty of the hidden variables, i.e. we work with the *marginal likelihood*
- Taking the derivative of the marginal likelihood we see

$$\frac{\partial}{\partial\theta}\log p(\tilde{x}_i;\theta) = \frac{1}{p(\tilde{x}_i;\theta)}\frac{\partial}{\partial\theta}\int_{z_i}\sum_{h_i} p(\tilde{x}_i, z_i, h_i;\theta)dz_i$$

$$= \int_{z_i}\sum_{h_i}\frac{p(\tilde{x}_i, z_i, h_i;\theta)}{p(\tilde{x}_i;\theta)}\frac{\partial}{\partial\theta}\log p(\tilde{x}_i, z_i, h_i;\theta)dz_i$$

$$= \int_{z_i}\sum_{h_i} p(z_i, h_i \mid \tilde{x}_i;\theta)\frac{\partial}{\partial\theta}\log p(\tilde{x}_i, z_i, h_i;\theta)dz_i$$

$$= \mathrm{E}\left[\frac{\partial}{\partial\theta}\log p(\tilde{x}_i, z_i, h_i;\theta)\right]_{p(z_i, h_i \mid \tilde{x}_i;\theta)}$$

Learning with expected gradient descent (EGD)

- The marginal likelihood can be optimized using gradient ascent by computing and following the expected gradient of the log joint likelihood
- This can be decomposed into three steps, a

  P-step where we compute the posterior over hidden variables; then an

  E-step where we compute the expectation of the gradient given the posterior; then a

  G-step, where we use gradient-based optimization to maximize the (*marginal likelihood*) objective function with respect to the model parameters.

The Expectation Maximization (EM) Algorithm

- A more well known method, similar to EGD
- The EM algorithm consists of two steps, an **E-step** that computes the expectations used in the expected log-likelihood; and an **M-step** in which the objective is maximized – typically using a closed-form parameter update.
- The expected log joint likelihood is related to the marginal likelihood in the following way

$$\log P(\tilde{x}_i;\theta) = \mathrm{E}_{p(z_i,h_i \mid \tilde{x}_i;\theta)}\left[\log p(\tilde{x}_i,z_i,h_i;\theta)\right] + \mathrm{H}[p(z_i,h_i \mid \tilde{x}_i;\theta)],$$

The EM algorithm in more detail

- In a joint probability model with discrete hidden variables **H** the probability of the observed data can be maximized by initializing θ^{old} and repeating the steps

1. **E-step**: Compute expectations using $P(H \mid X;\theta^{old})$
2. **M-step**: Find $\theta^{new} = \arg\max_{\theta} \left[\sum_{H} P(H \mid X;\theta^{old}) \log P(X,H;\theta) \right]$
3. If the algorithm has not converged, set $\theta^{old} = \theta^{new}$ and return to step 1

- The M step corresponds to maximizing the expected log-likelihood, the overall procedure maximizes the marginal likelihood of the data
- Although discrete hidden variables are used above, the approach generalizes to continuous ones

EM for Bayesian networks

- Unconditional probability distributions are estimated in the same way in which it would be computed if the variables A_i had been observed, but with each observation replaced by its posterior marginal probability (i.e. marginalizing over the other variables)

$$\theta_{A=a}^{new} = P(A=a) = \frac{1}{N}\sum_{i=1}^{N} P(A_i = a \mid \{\tilde{X}\}_i; \theta^{current})$$

- Conditional distributions are updated using ratios of posterior marginals, ex.

$$P(B=b \mid A=a) = \frac{\sum_{i=1}^{N} P(A_i = a, B_i = b \mid \{\tilde{X}\}_i; \theta^{current})}{\sum_{i=1}^{N} P(A_i = a \mid \{\tilde{X}\}_i; \theta^{current})}.$$

EM in practice

- For many latent variable models – Gaussian mixture models, probabilistic principal component analysis, and hidden Markov models – the required posterior distributions can be computed exactly, which accounts for their popularity.
- However, for many probabilistic models it is simply not possible to compute an exact posterior distribution.
- This can easily happen with multiple hidden random variables, because the posterior needed in the E-step is the joint posterior of the hidden variables.
- There is a vast literature on the subject of how to compute approximations to the true posterior distribution over hidden variables in more complex models.
- Sampling methods and variational methods are popular in machine learning

Probability Models and PCA

Latent semantic analysis (LSA) and the singular value decomposition (SVD)

- LSA factorizes a data matrix (ex. of word counts) using SVD

$$\underset{t \times d}{\left[\tilde{\mathbf{X}}\right]} = \underset{t \times k}{\left[\mathbf{U}_k\right]} \underset{k \times k}{\begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_k \end{bmatrix}} \underset{k \times d}{\left[\mathbf{V}_k^{\mathrm{T}}\right]}$$

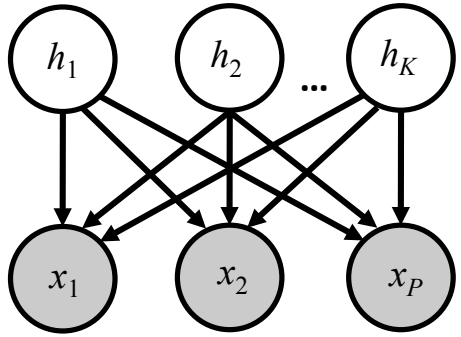
where **U** and **V** have orthogonal columns and **S** is a diagonal matrix containing the singular values, usually sorted in decreasing order (t=#terms, d=#docs, k=#topics)

- For every $k < d$, $\tilde{\mathbf{X}} \approx \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^{\mathrm{T}}$ and if all but the k largest singular values are discarded the data matrix can be reconstructed in an optimal in a least squares sense

Probabilistic PCA

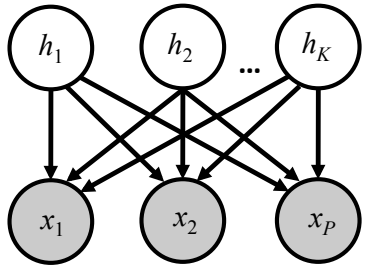
- Probabilistic PCA, (PPCA) can be written as a Bayes net
- Let **x** be a *d*-dimensional random vector of observed data, and **h** be a *k*-dimensional vector of hidden, $k < d$ typically
- The joint probability model has this linear Gaussian form

$$p(\mathbf{x},\mathbf{h}) = p(\mathbf{x}\,|\,\mathbf{h})p(\mathbf{h})$$
$$= N(\mathbf{x};\mathbf{W}\mathbf{h}+\boldsymbol{\mu},\mathbf{D})N(\mathbf{h};\mathbf{0},\mathbf{I}),$$



- where $p(\mathbf{h})$ is a Gaussian distributed random variable having a zero mean and identity covariance, while $p(\mathbf{x}|\mathbf{h})$ is Gaussian with mean $\mathbf{W}\mathbf{h}+\boldsymbol{\mu}$, and a diagonal covariance matrix $\mathbf{D}=\sigma^2\mathbf{I}$
- The mean $\boldsymbol{\mu}$ is included as a parameter, but it would be zero if we first mean-centered the data, mixtures of PPCA use these means to model more complex distributions

PPCA and Factor Analysis



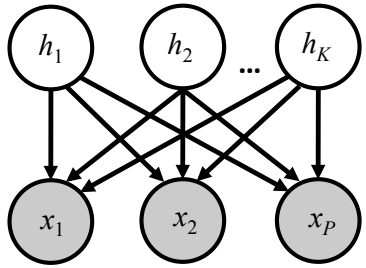
- Restricting the covariance matrix **D** to be diagonal produces a model known as *factor analysis*
- Because of the nice properties of Gaussian distributions, *the marginal distribution* of **x** in these models are also Gaussian

$$p(\mathbf{x}) = N(\mathbf{x};\boldsymbol{\mu},\mathbf{M} = \mathbf{W}\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I})$$

- The *posterior distribution* for **h** can be obtained from Bayes' rule, and some Gaussian identities

$$p(\mathbf{h} \mid \mathbf{x}) = N(\mathbf{h};\mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu}),\sigma^2\mathbf{M}^{-1})$$

PPCA, learning and the marginal likelihood



- The *log marginal likelihood* of the parameters given all the observed data can be maximized using this objective function

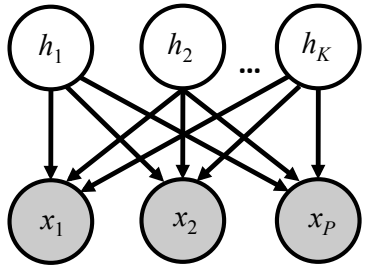
$$L(\tilde{\mathbf{X}};\theta) = \log\left[\prod_{i=1}^{N} P(\tilde{\mathbf{x}}_i;\theta)\right] = \sum_{i=1}^{N} \log\left[N(\tilde{\mathbf{x}}_i;\boldsymbol{\mu}, \mathbf{W}\mathbf{W}^{\mathbf{T}} + \sigma^2\mathbf{I})\right]$$

where the model parameters are $\theta = \{\mathbf{W}, \boldsymbol{\mu}, \sigma^2\}$

- Compare this to the joint probability

$$L(\tilde{\mathbf{X}}, \mathbf{H};\theta) = \log\left[\prod_{i=1}^{N} p(\tilde{\mathbf{x}}_i, \mathbf{h}_i;\theta)\right] = \sum_{i=1}^{N} \log\left[p(\tilde{\mathbf{x}}_i \mid \mathbf{h}_i; \mathbf{W}) p(\mathbf{h}_i;\sigma^2)\right].$$

Gradient ascent for PPCA



- The expected log-likelihood (ELL) of the data is

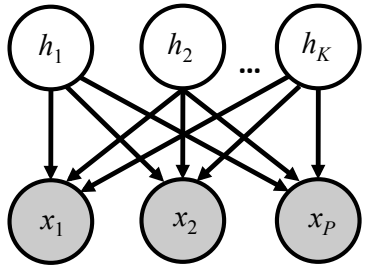
$$\mathrm{E}[L(\tilde{\mathbf{X}},\mathbf{H};\theta)]_{p(\mathbf{H}|\tilde{\mathbf{X}})} = \sum_{i=1}^{N} \mathrm{E}\Big[\log\big[p(\tilde{\mathbf{x}}_i,\mathbf{h}_i;\theta)\big]\Big]_{p(\mathbf{h}_i|\tilde{\mathbf{x}}_i)}$$

- The gradient of the ELL under $p(\mathbf{h} \mid \tilde{\mathbf{x}})$ is

$$\begin{aligned}\frac{\partial}{\partial \mathbf{W}} \mathrm{E}[L(\tilde{\mathbf{x}},\mathbf{h})] &= \mathrm{E}[\mathbf{W}\mathbf{h}\mathbf{h}^{\mathrm{T}}] - \mathrm{E}[\tilde{\mathbf{x}}\mathbf{h}^{\mathrm{T}}] \\ &= \mathbf{W}\,\mathrm{E}[\mathbf{h}\mathbf{h}^{\mathrm{T}}] - \tilde{\mathbf{x}}\,\mathrm{E}[\mathbf{h}]^{\mathrm{T}}\end{aligned}$$

- This has a natural interpretation as a difference between two expectations, the first is akin to the models prediction or reconstruction of the input forming a matrix with the models explanation of the input

EM for PPCA



- The E and M steps of the PPCA EM algorithm can be written as

E-Step:
$$\mathrm{E}[\mathbf{h}_i] = \mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}\tilde{\mathbf{x}}_i, \quad \mathrm{E}[\mathbf{h}_i\mathbf{h}_i^{\mathrm{T}}] = \sigma^2\mathbf{M}^{-1} + \mathrm{E}[\mathbf{h}_i]\,\mathrm{E}[\mathbf{h}_i^{\mathrm{T}}],$$

M-Step:
$$\mathbf{W}^{New} = \left[\sum_{i=1}^{N} \tilde{\mathbf{x}}_i\, \mathrm{E}[\mathbf{h}_i]^{\mathbf{T}}\right]\left[\sum_{i=1}^{N} \mathrm{E}[\mathbf{h}_i\mathbf{h}_i{}^{\mathrm{T}}]\right]^{-1},$$

where all expectations are taken with respect to each example's posterior distribution

- In the zero input noise limit these equations can be written even more compactly

Bibliographic Notes & Further Reading

Expectation Maximization (EM) and Variational Methods

- The expectation maximization algorithm originates in the influential work of Dempster, Laird, and Rubin (1977)
- The modern variational view provides a solid theoretical justification for the use of approximate posterior distributions, see Bishop (2006) for more details
- The variational perspective on EM originated in the 1990s with work by Neal and Hinton (1998), Jordan et al. (1998), and others
- Salakhutdinov, Roweis and Ghahramani (2003) explore the EM approach and compare it with the expected gradient, including the more sophisticated expected conjugate gradient based optimization

Bibliographic Notes & Further Reading

Principal Component Analysis and Related Probabilistic Models

- Roweis (1998) gives an early EM formulation for probabilistic principal component analysis: he examines the zero input noise case and provides the elegant mathematics for the simplified EM algorithm
- Tipping and Bishop (1999) give further analysis, and show that after optimizing the model and learning the variance of the observation noise the columns of the matrix **W** are scaled and rotated principal eigenvectors of the covariance matrix of the data.
- Further generalizations, such as mixtures of principal component analyzers are in (Dony and Haykin, 1995; Tipping and Bishop, 1999)
- See Ghahramani, and Hinton, (1996) for mixtures of factor analyzers
- Ilin and Raiko (2010) present PPCA with data that is missing at random
- Edwards (2012) provides a nice introduction to graphical modeling, including mixed models with discrete and continuous components and graphical Gaussian or inverse covariance models

Bayesian estimation and prediction

- In a more Bayesian modelling setting the joint distribution of data *and* parameters under a model can be written as

$$p(x_1, x_2, \ldots, x_n, \theta; \alpha) = \prod_{i=1}^{n} p(x_i \mid \theta) p(\theta; \alpha),$$

- Where α is a hyperparameter
- Bayesian predictions use a quantity known as the *posterior predictive distribution* - the probability model for a new observation marginalized over the posterior probability inferred for the parameters given the observations so far.

$$p(x_{new} \mid \tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n; \alpha) = \int_{\theta} p(x_{new} \mid \theta) p(\theta \mid \tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n; \alpha) d\theta,$$

Empirical Bayesian methods

- One approach is obtained by maximizing the marginal likelihood with respect to the model's hyperparameters, ex.

$$\alpha_{MML} = \underset{\alpha}{\operatorname{argmax}} \left[\log \int \prod_{i=1}^{n} p(x_i \mid \theta) p(\theta; \alpha) d\theta \right]$$

Probabilistic inference

- With complex probability models – and even with some seemingly simple ones – computing quantities such as posterior distributions, marginal distributions, and the maximum probability configuration, often require specialized methods to achieve results efficiently – even approximate ones.
- This is the field of *probabilistic inference*

Sampling

- Sampling methods are popular in both statistics and machine learning
- Useful for implementing fully Bayesian methods that use distributions on parameters, or for inference in graphical models with cyclic structures
- *Gibbs sampling* is a popular special case of the more general Metropolis-Hastings algorithm
- Allows one to generate samples from a joint distribution even when the true distribution is a complex continuous function

Gibbs sampling

- Gibbs sampling is conceptually very simple.
- Assign an initial set of states to the random variables of interest.
- With *n* random variables, the initial assignments or samples can be written as $x_1 = x_1^{(0)}, \ldots, x_n = x_n^{(0)}$
- Then iteratively cycling through updates to each variable by *sampling* from its conditional distribution given the others:

$$x_1^{(i+1)} \sim p(x_1 \mid x_2 = x_2^{(i)}, \ldots, x_n = x_n^{(i)}),$$
$$\vdots$$
$$x_n^{(i+1)} \sim p(x_n \mid x_1 = x_1^{i}, \ldots, x_{n-1} = x_{n-1}^{i}).$$

Simulated annealing

- A procedure that seeks an approximate most probable explanation (MPE) or configuration
- We can obtain it by adapting Gibbs sampling to include an iteration-dependent "temperature" term t_i, that decreases (usually slowly) over time
- Starting with an initial assignment, $x_1 = x_1^{(0)}, \ldots, x_n = x_n^{(0)}$ subsequent samples take the form

$$x_1^{(i+1)} \sim p(x_1 \mid x_2 = x_2^{(i)}, \ldots, x_n = x_n^{(i)})^{\frac{1}{t_i}},$$
$$\vdots$$
$$x_n^{(i+1)} \sim p(x_n \mid x_1 = x_1^{i}, \ldots, x_{n-1} = x_{n-1}^{i})^{\frac{1}{t_i}},$$

Iterated conditional modes

- Consists of iterations of the form

$$x_1^{(i+1)} \sim \underset{x_1}{\operatorname{argmax}}\, p(x_1 \mid x_2 = x_2^{(i)}, \ldots, x_n = x_n^{(i)}),$$

$$\ldots$$

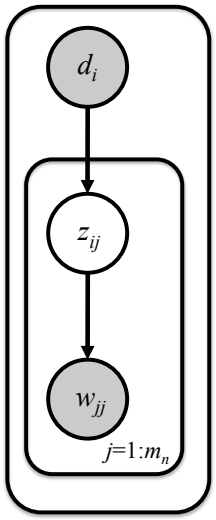
$$x_n^{(i+1)} \sim \underset{x_n}{\operatorname{argmax}}\, p(x_n \mid x_1 = x_1^{i}, \ldots, x_{n-1} = x_{n-1}^{i}).$$

- Often converges quickly to an approximate MPE, but is prone to local minima
- Zero temperature limit of a simulated annealer
- Gibbs sampling is a popular special case of the more general Metropolis Hastings algorithm
- Gibbs sampling and simulated annealing are special cases of the broader class of Markov Chain Monte Carlo (MCMC) methods

Variational methods

- Rather than sampling from a complex distribution the distribution can be approximated by a simpler, more tractable, function
- Suppose we have a probability model with a set of hidden variables *H* and observed variables *X*
- Let $p = p(H \mid \tilde{X}; \theta)$ be the exact posterior
- Define $q = q(H \mid \tilde{X}; \Phi)$ as the variational approx. having its own variational parameters, Φ
- Variational methods make q close to p, yielding principled EM algorithms with approx. posteriors

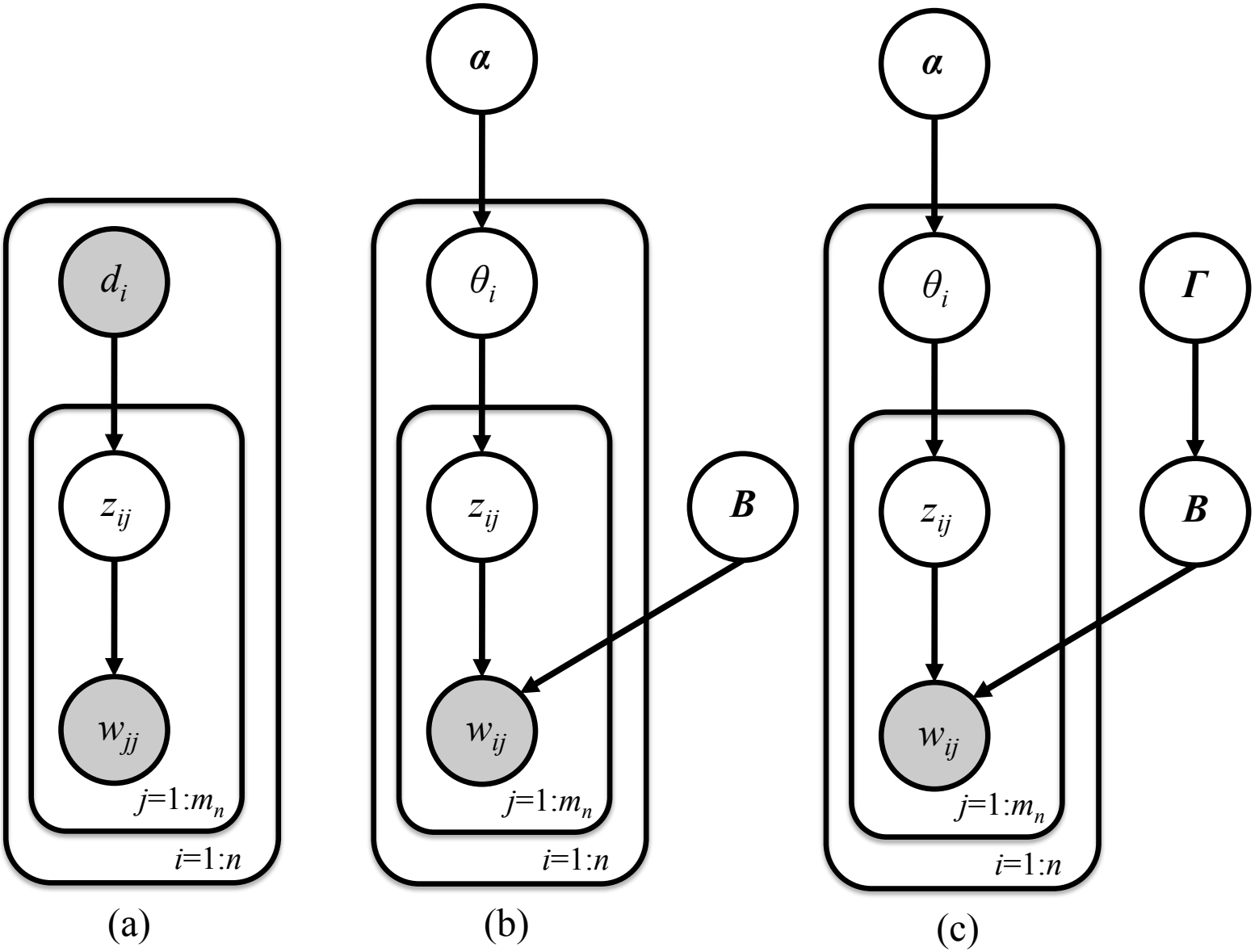
Probabilistic Latent Semantic Analysis (pLSA)



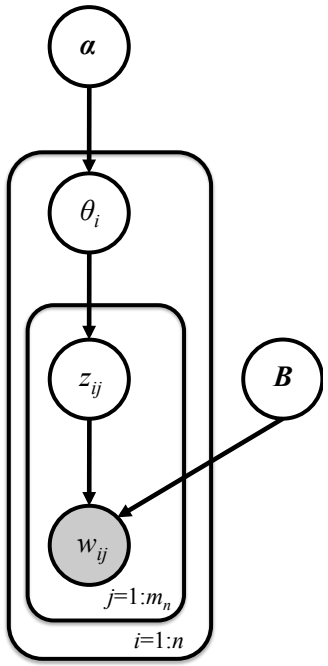
- In the pLSA framework one considers the *index* of each document as encoded using observations of discrete random variables d_i for i=1,…,n documents
- Each variable d_i has n states, and over the document corpus there is one observation of the variable for each state.
- Topics represented with discrete variables z_{ij}, while words are represented with random variables w_{ij}, where m_i words are associated with each document and each word is associated with a topic.

$$P(W,D) = \prod_{i=1}^{n} P(d_i) \prod_{j=1}^{m_n} \sum_{z_{ij}} P(z_{ij} \mid d_i) P(w_{ij} \mid z_{ij}).$$

pLSA, LDA and smoothed LDA



Latent Dirichlet Allocation (LDA)



- Reformulates pLSA replacing document index variables d_i with the random parameter θ
- The distribution of θ is influenced by a Dirichlet prior with hyperparameter α
- The relationship between discrete topic variables z_{ij} and the words w_{ij} is also given an explicit dependence on the hyperparameter, matrix **B** .
- The probability model for all observed words W is

$$P(\mathrm{W}|\boldsymbol{\alpha},\mathbf{B}) = \prod_{i=1}^{n} \int P(\boldsymbol{\theta}_i \mid \boldsymbol{\alpha}) \left[\prod_{j=1}^{m_n} \sum_{z_{ij}} P(z_{ij} \mid \boldsymbol{\theta}_i) P(w_{ij} \mid z_{ij}, \mathbf{B}) \right] d\boldsymbol{\theta}_i$$

$$= \prod_{i=1}^{n} \int P(\boldsymbol{\theta}_i \mid \boldsymbol{\alpha}) \left[\prod_{j=1}^{m_n} P(w_{ij} \mid \boldsymbol{\theta}_i, \mathbf{B}) \right] d\boldsymbol{\theta}_i,$$

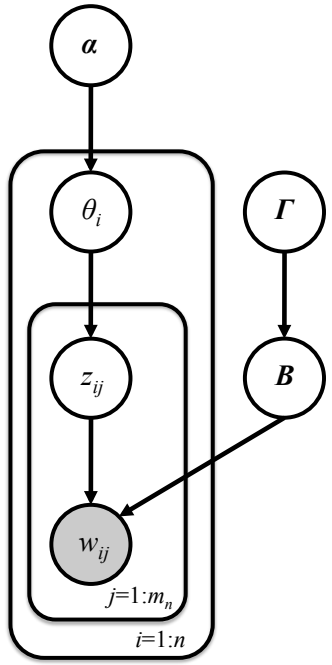
Inference for LDA

- The marginal log-likelihood of the model can be optimized using an empirical Bayesian method by adjusting the hyperparameter α and parameter B
- To perform the E-step of EM, the posterior distribution over the unobserved random quantities is used

$$P(\boldsymbol{\theta},\mathbf{z}|\mathbf{w},\boldsymbol{\alpha},\mathbf{B}) = \frac{P(\boldsymbol{\theta},\mathbf{z},\mathbf{w}|\boldsymbol{\alpha},\mathbf{B})}{P(\mathbf{w}|\boldsymbol{\alpha},\mathbf{B})}$$

- This posterior is intractable and is typically computed using a variational approximation or by sampling

Smoothed LDA



- Reduces the effects of overfitting
- Adds another Dirichlet prior with hyperparameters given by **Γ** on the topic parameters **B**
- *Collapsed Gibbs sampling* can be performed by integrating out the θs and **B** analytically, which deals with these distributions *exactly*
- The Gibbs sampler proceeds by simply iteratively updating each z_{ij} conditioned on **Γ** and α to compute the required approximate posterior

Topics from PNAS

- Example: Griffiths and Steyvers (2004) applied smoothed LDA to 28,154 abstracts of papers published in the *Proceedings of the National Academy of Science (PNAS)* from 1991 to 2001

Topic 2	**Topic 39**	**Topic 102**	**Topic 201**	**Topic 210**
Species	Theory	Tumor	Resistance	Synaptic
Global	Time	Cancer	Resistant	Neurons
Climate	Space	Tumors	Drug	Postsynaptic
Co_2	Given	Human	Drugs	Hippocampal
Water	Problem	Cells	Sensitive	Synapses
Geophysics, geology, ecology	Physics, math, applied math	Medical sciences	Pharmacology	Neurobiology

- User tags shown at the bottom were not used to create the topics, but correlate very well with the inferred topics

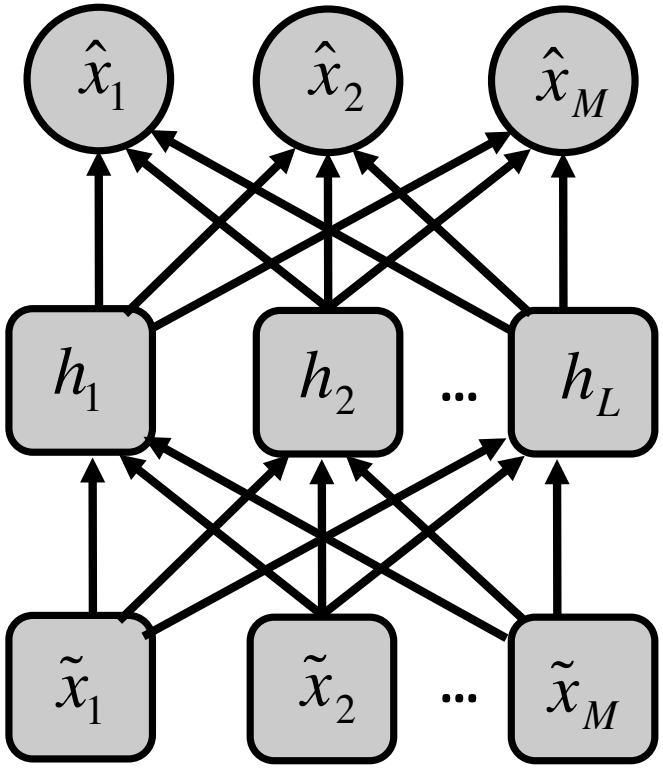
Autoencoders

Autoencoders

- Used for unsupervised learning
- It is a network that learns an efficient coding of its input.
- The objective is simply to reconstruct the input, but through the intermediary of a compressed or reduced-dimensional representation.
- If the output is formulated using probability, the objective function is to optimize $p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}})$, that is, the probability that the model gives a random variable **x** the value $\hat{\mathbf{x}}$ given the observation $\tilde{\mathbf{x}}$, where $\hat{\mathbf{x}} = \tilde{\mathbf{x}}$.
- In other words, the model is trained to predict its own input—but it must map it through a representation created by the hidden units of a network.

A simple autoencoder

- Predicts its own input, ex. $p(\hat{\mathbf{x}} \mid \tilde{\mathbf{x}}) = p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}}; \mathbf{f}(\tilde{\mathbf{x}}))$
- Going through an encoding, $\mathbf{e} = \mathbf{h} = \text{act}(\mathbf{a}^{(1)})$



where

$$\mathbf{f}(\tilde{\mathbf{x}}) = \mathbf{f}\left(\mathbf{d}\left(\mathbf{e}(\tilde{\mathbf{x}})\right)\right),$$

$$\mathbf{d} = \text{out}(\mathbf{a}^{(2)}),$$

$$\mathbf{a}^{(2)} = \mathbf{W}^{\mathrm{T}}\mathbf{h} + \mathbf{b}^{(2)},$$

$$\mathbf{h} = \text{act}(\mathbf{a}^{(1)}),$$

$$\mathbf{a}^{(1)} = \mathbf{W}\tilde{\mathbf{x}} + \mathbf{b}^{(1)}$$

Autoencoders

- Since the idea of an autoencoder is to compress the data into a lower-dimensional representation, the number L of hidden units used for encoding is less than the number M in the input and output layers
- Optimizing the autoencoder using the negative log probability over a data set as the objective function leads to the usual forms
- Like other neural networks it is typical to optimize autoencoders using backpropagation with mini-batch based SGD

Linear autoencoders and PCA

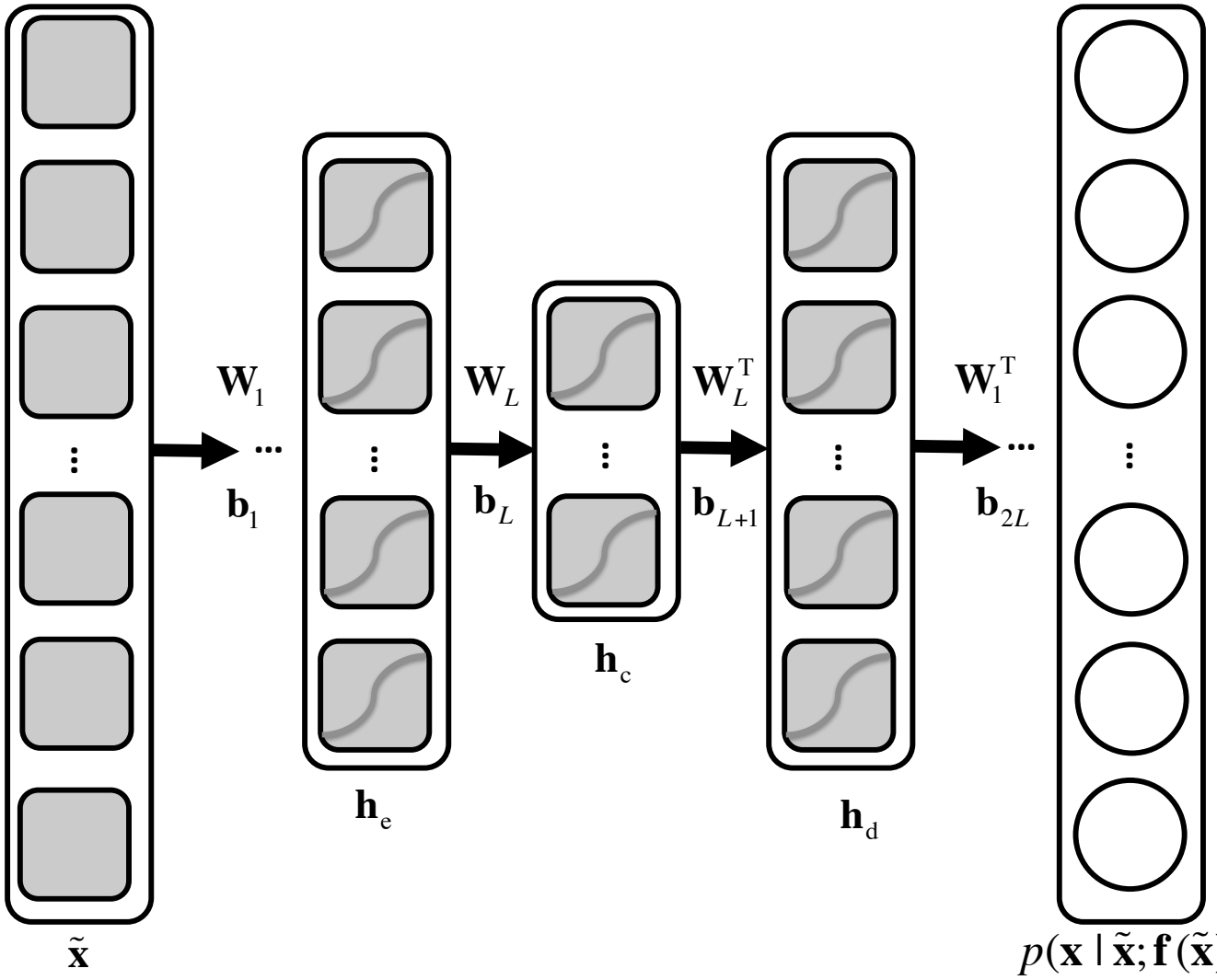
- Both the encoder activation function act() and the output activation function out() in the simple autoencoder model could be defined as the sigmoid function
- However, it can be shown that with no activation function, $\mathbf{h}^{(i)}=\mathbf{a}^{(i)}$, the resulting "linear autoencoder" will find the same subspace as PCA, (assuming a squared-error loss function and normalizing the data using mean centering)
  - Can be shown to be optimal in the sense that any model with a non-linear activation function would require a weight matrix with more parameters to achieve the same reconstruction error
- Even with non-linear activation functions such as a sigmoid, optimization finds solutions where the network operates in the linear regime, replicating the behavior of PCA
- This might seem discouraging; however, using a neural network with even one hidden layer to create much more flexible transformations, and
  - There is growing evidence deeper models can learn more useful representations

Deep autoencoders

- When building autoencoders from more flexible models, it is common to use a *bottleneck* in the network to produce an under-complete representation, providing a mechanism to obtain an encoding of lower dimension than the input.
- Deep autoencoders are able to learn low-dimensional representations with smaller reconstruction error than PCA using the same number of dimensions.
- Can be constructed by using L layers to create a hidden layer representation $\mathbf{h}_c^{(L)}$ of the data, and following this with a further L layers $\mathbf{h}_d^{(L+1)}$… $\mathbf{h}_d^{(2L)}$ to decode the representation back into its original form
- The j=1,…,2L weight matrices for each of the i=1,…,L encoding and decoding layers are constrained by

$$\mathbf{W}_{L+i} = \mathbf{W}_{L+1-i}^{\mathrm{T}}$$

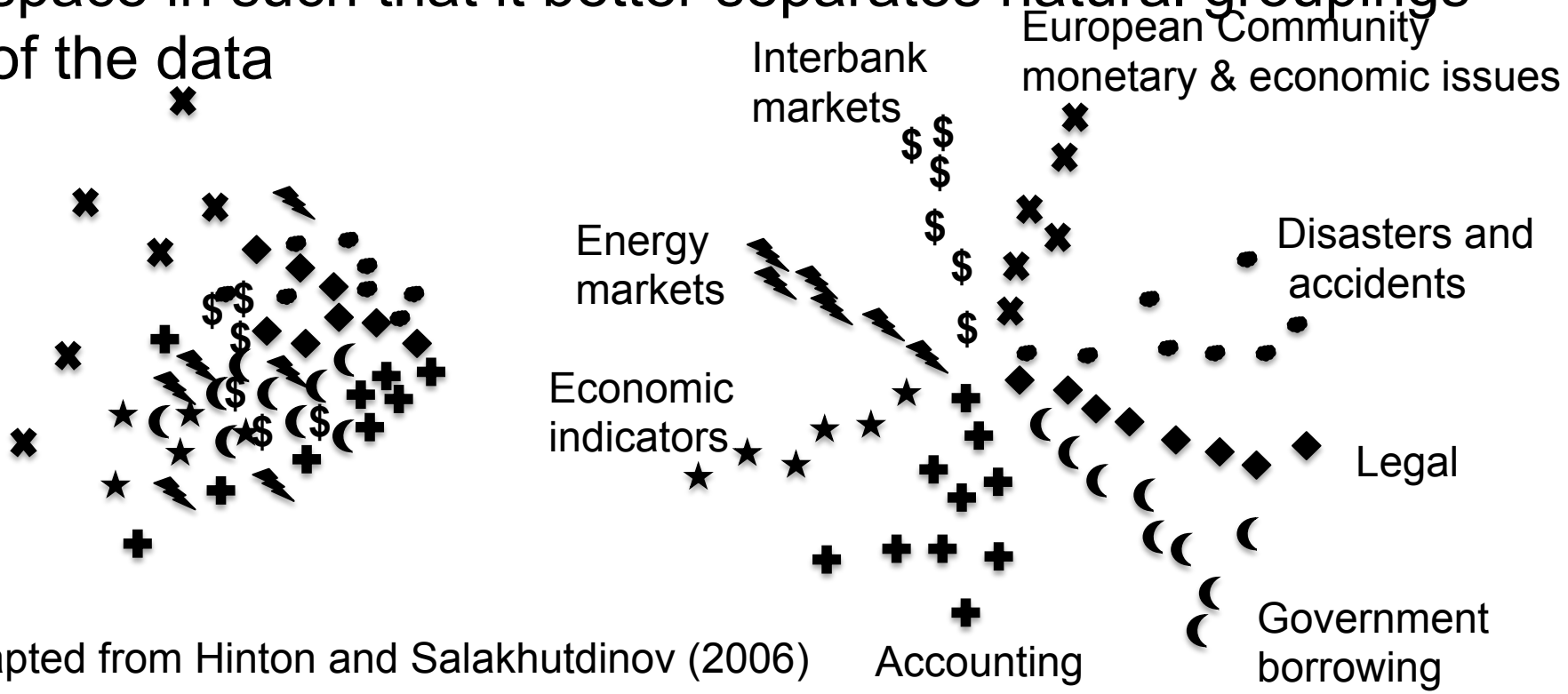
A deep autoencoder



$$\mathbf{f}(\mathbf{x}) = \mathbf{f}_{\mathbf{d}}(\mathbf{a}_{\mathbf{d}}^{(2L)}(\ldots \mathbf{h}_{\mathbf{d}}^{(L+1)}(\mathbf{a}_{\mathbf{d}}^{(L+1)}(\mathbf{h}_{\mathbf{c}}^{(L)}(\mathbf{a}_{\mathrm{e}}^{(L)}(\ldots \mathbf{h}_{\mathrm{e}}^{(1)}(\mathbf{a}_{\mathrm{e}}^{(1)}(\mathbf{x}))))))))).$$

Deep autoencoders

- A comparison of data projected into a 2D space with PCA (left) vs a deep autoencoder (right) for a text dataset
- The non-linear autoencoder can arrange the learned space in such that it better separates natural groupings of the data



Adapted from Hinton and Salakhutdinov (2006)

Training autoencoders

- Deep autoencoders are an effective framework for non-linear dimensionality reduction
- It can be difficult to optimize autoencoders; being careful about activation function choice and initialization can help
- Once such a network has been built, the top-most layer of the encoder, the code layer $\mathbf{h}_c$, can be input to a supervised classification procedure
- One can pre-train a discriminative neural net with an autoencoder
- One can also use a composite loss from the start, with a reconstructive (unsupervised) and a discriminative (supervised) criterion

$$L(\theta) = (1 - \lambda) L_{\text{sup}}(\theta) + \lambda L_{\text{unsup}}(\theta)$$

  where λ is a hyperparameter that balances the two objectives
- Another approach to training autoencoders is based on pre-training by stacking two-layered restricted Boltzmann machines RBMs

Denoising autoencoders

- Autoencoders can be trained layerwise, using autoencoders as the underlying building blocks
- One can use greedy layerwise training strategies involving plain autoencoders to train deep autoencoders, but attempts to do this for networks of even moderate depth has been problematic
- Procedures based on stacking denoising autoencoders have been found to work better
- Denoising autoencoders are trained to remove different types of noise that has been added synthetically to their input
- Autoencoder inputs can be corrupted with noise such as: Gaussian noise; masking noise, where some elements are set to 0; and salt-and-pepper noise, where some elements are set to minimum and maximum input values (such as 0 and 1)

Bibliographic Notes & Further Reading

Autoencoders

- Hinton and Salakhutdinov (2006) noted that it has been known since the 1980s that deep autoencoders, optimized through backpropagation, could be effective for non-linear dimensionality reduction.
- The key limiting factors were the small size of the datasets used to train them, coupled with low computation speeds; plus the old problem of local minima.
- By 2006, datasets such as the MNIST digits and the 20 Newsgroups collection were large enough, and computers were fast enough, for Hinton and Salakhutdinov to present compelling results illustrating the advantages of deep autoencoders over principal component analysis.
  - Their experimental work used generative pre-training to initialize weights to avoid problems with local minima.

Bibliographic Notes & Further Reading

Autoencoders

- Bourlard and Kamp (1988) provide a deep analysis of the relationships between autoencoders and principal component analysis.
- Vincent et al. (2010) proposed stacked denoising autoencoders and found that they outperform both stacked standard autoencoders and models based on stacking restricted Boltzmann machines.
- Cho and Chen (2014) produced state-of-the-art results on motion capture sequences by training deep autoencoders with rectified linear units using hybrid unsupervised and supervised learning.