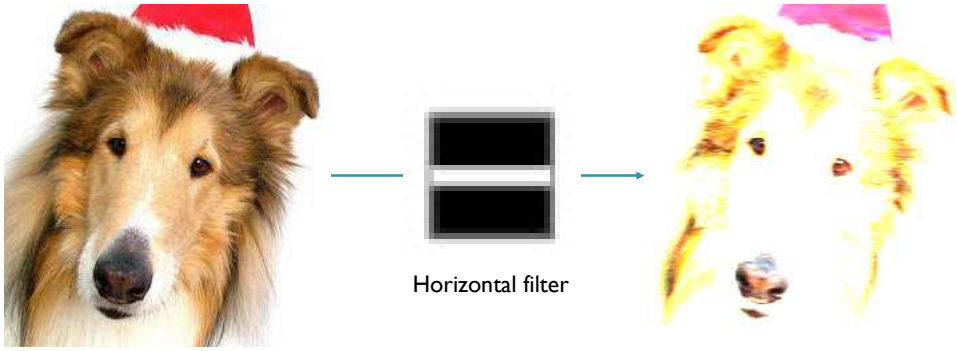
Convolutional Neural Networks (CNN)

A picture says a 1000 words

Feature selection

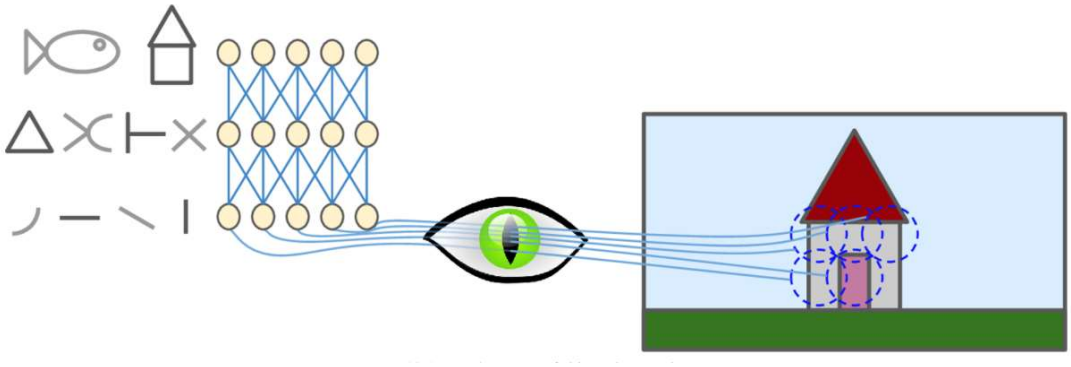


Why Convolutional Neural Networks?

Neural network analogy with human vision system and the architecture of the visual cortex

→Local receptive field: limited regions (e.g. 5 neurons/layer)

→Complex patterns are combinations of lower-level patterns



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

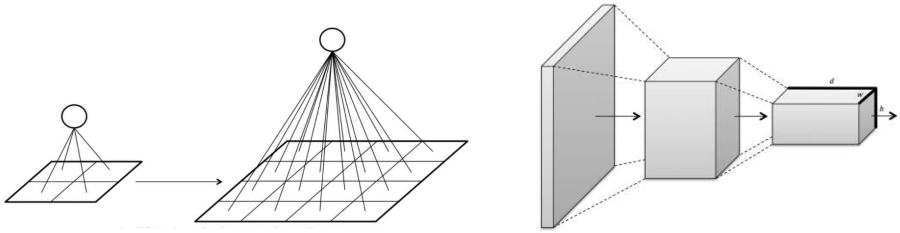
Why Convolutional Neural Networks?

Ideas have fed into & driven NN research

- One idea is effective scaling
  - Not all neurons fire from an image
  - Distribution based on features
- Second idea is NN can recognize faces in different lighting/angles/obscuration
- Third idea is that it is the combination of features that define recognition and works even for partially obscured images
- Fourth idea is that there are hierarchical relationships that can help learning

Why Convolutional Neural Networks?

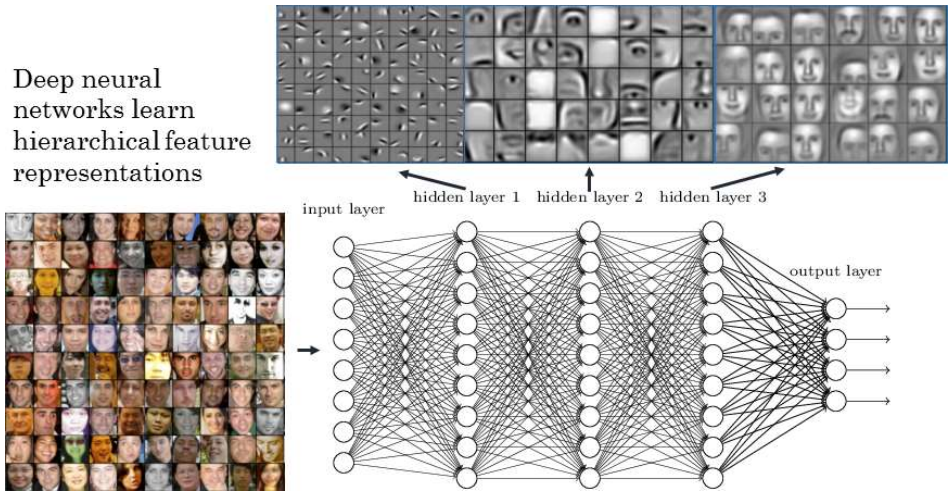
- MNIST uses 28x28 images→784 weights
- Scaling up to 256x256 would kill them
- Need a scalable solution
- 2012 ImageNet benchmark→25.7% error
  - Alex Krizhevsky (Geoffrey Hinton's Lab)
  - AlexNet used 1st CNN; later→ 16% error rate



from the book Fundamentals of Deep Learning, © Buduma, 2017

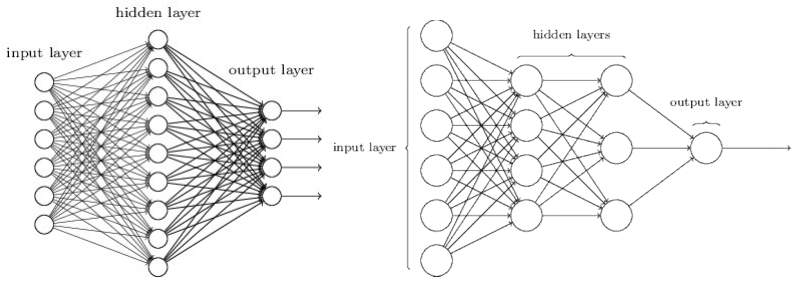
What Deep Learning is Doing

- http://www.rsipvision.com/exploring-deep-learning/



Layers vs. Hidden Layer Width

- Both might have the same number of nodes but which to choose?



- Generally #connections is the killer
- Wide hidden layer has more connections

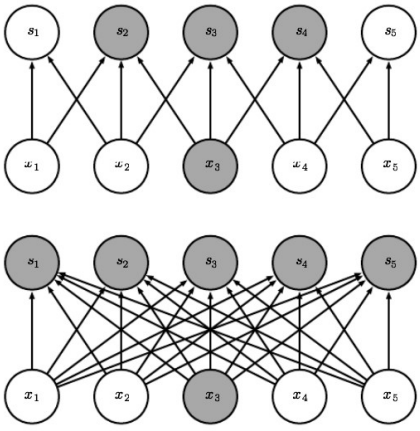
Observation

- Fully-connected layer: Connection goes from every node of previous layer to every node of next layer
- What if we relaxed that assumption?
- Brain uses regions of connected neurons
- NN with small regions of connectivity
  - Convolutional Neural Network
  - Also called “local networks”
  - Also called “locally connected networks”

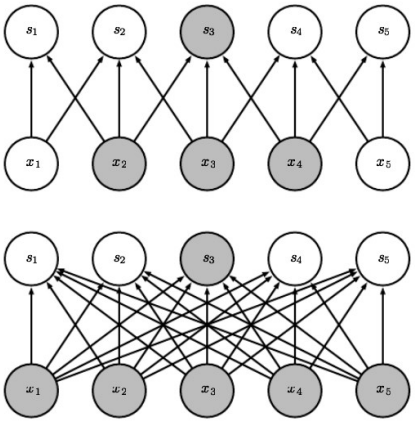
Densely connected vs. sparse connectivity

- kernel smaller than input → detect small, meaningful features
- store fewer weights, use less memory, improve statistical efficiency

view from below

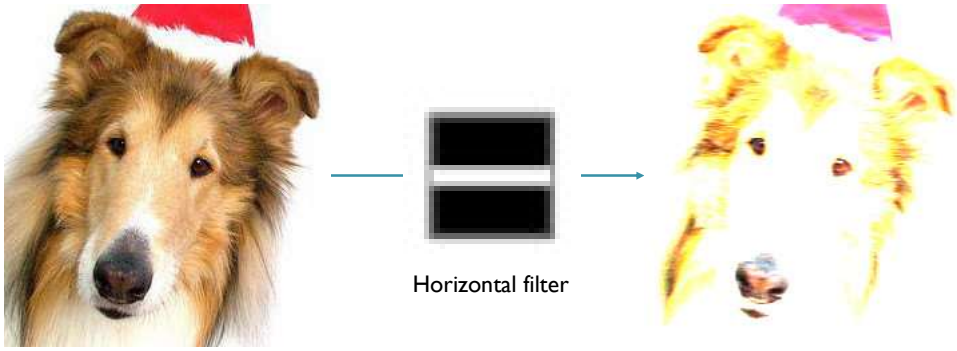


view from above



from the book Deep Learning, © Goodfellow et al., 2016

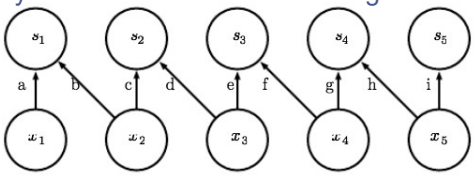
Weight Sharing



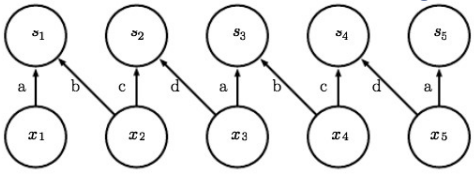
Weight Sharing

- a locally connected (sparse) network is not obliged to share weights
- in tiled convolutions, kernels rotate – different filters are applied to neighboring neurons
- sharing weights in a traditional Convolutional Neural Network has only one kernel that passes everywhere

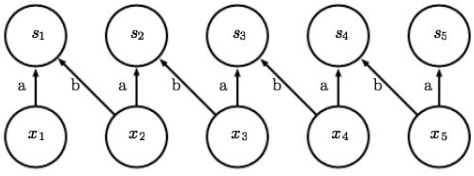
locally connected network – no weight sharing



tiled – 2 neurons have different weights

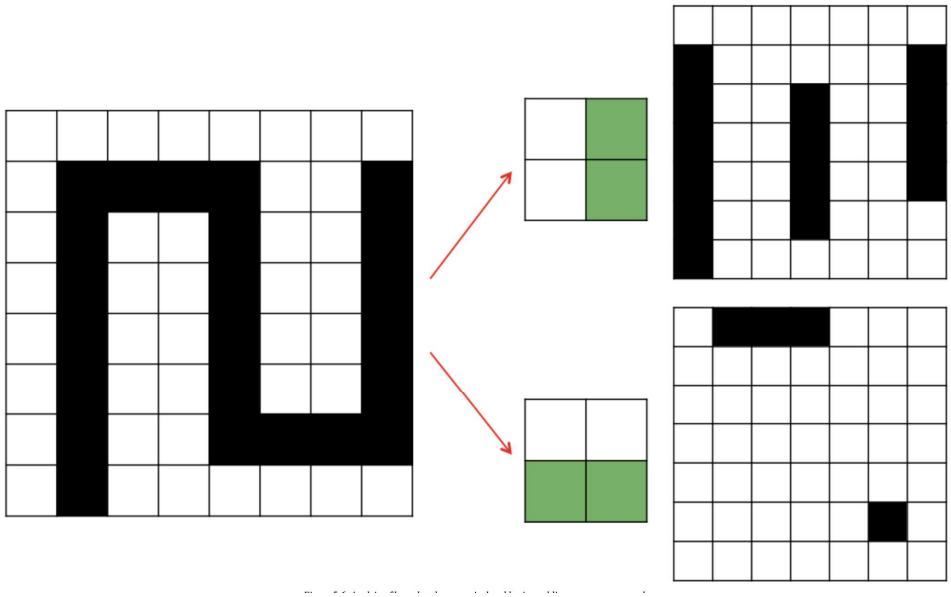


traditional convolution – tile is 1 neuron



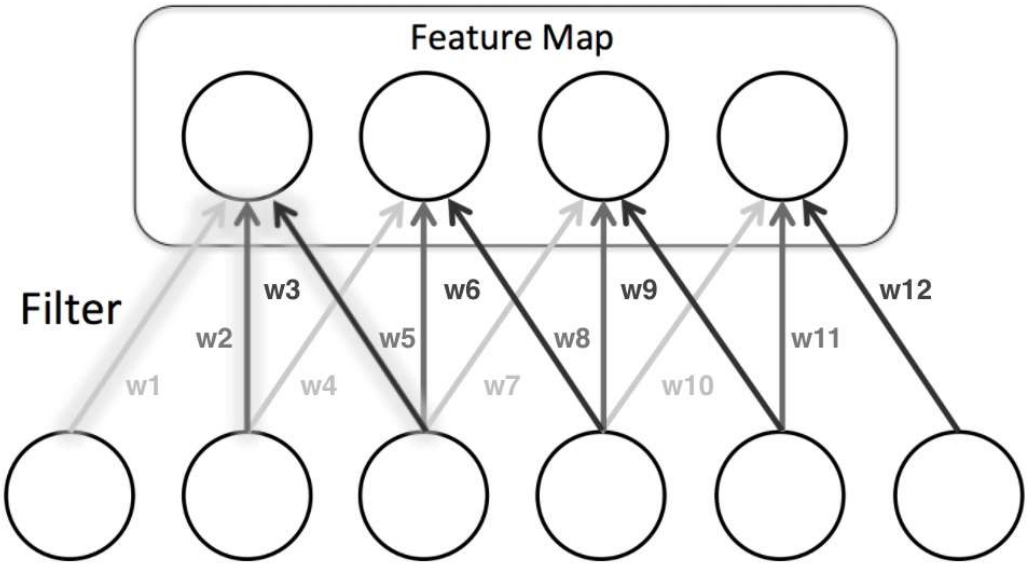
from the book Deep Learning, © Goodfellow et al., 2016

Weight Sharing



from the book Fundamentals of Deep Learning, © Buduma, 2017

Weight Sharing



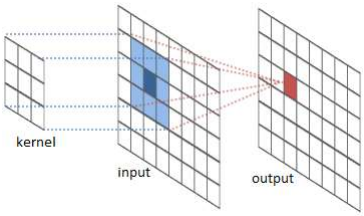
from the book Fundamentals of Deep Learning, © Buduma, 2017

Weight sharing

- Learning weights but shared (i.e. equal)
- Shifting by 3 here each time:
  - w1 = w4 = w7 = w10
  - w2 = w5 = w8 = w11
  - w3 = w6 = w9 = w12

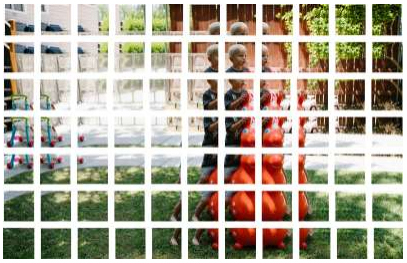
Convolution in 2D

- Use 2D mask
- Apply to all locations
  - Multiply mask
  - Add products
  - Store result to center pixel
- Goal: Find a head

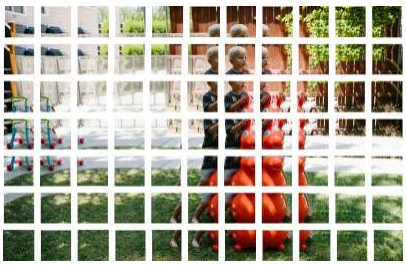


http://stackoverflow.com/questions/15356153/how-do-convolution-matrices-work

https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721#.nltvl4pk4



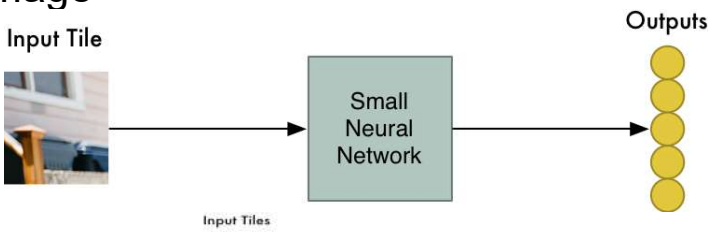
Convolution

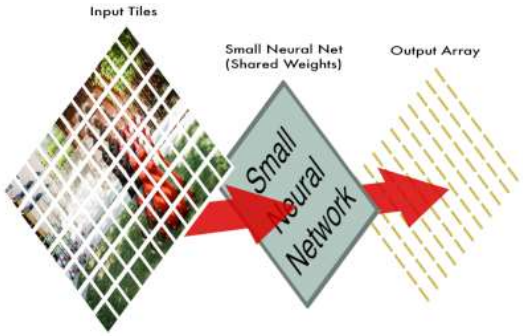


- We could have shifted every pixel
- Typically take a “stride” that will overlap
  - Stride = distance of the shift

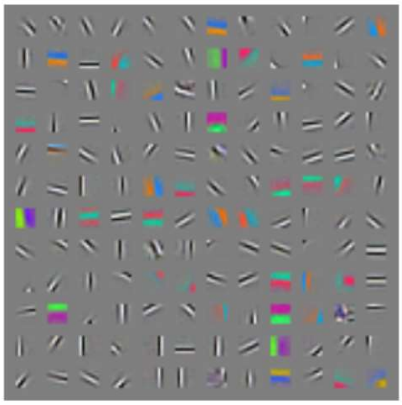
Feature Reduction

- Train neural network to produce outputs for each subimage





Weight sharing

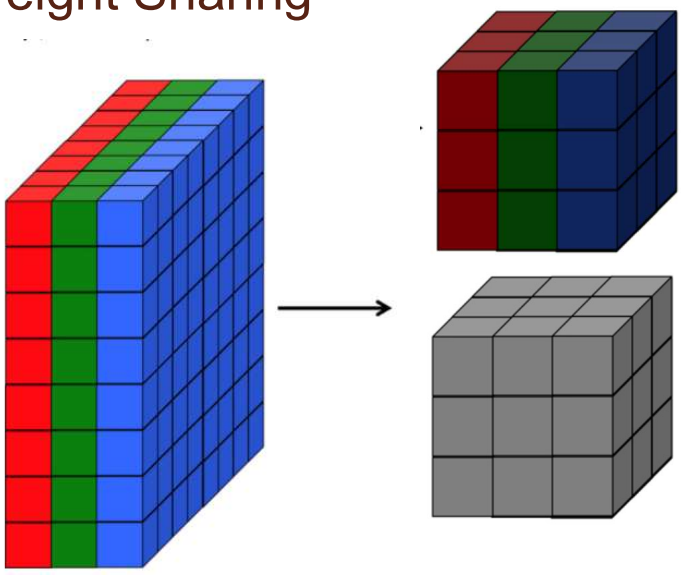


Convolutional kernels (filters) learned in a CNN

Elements in one of the feature maps are combinations of filter multiplications over all slices of the previous layer's feature maps

from the book Deep Learning, © Goodfellow et al., 2016

Weight Sharing



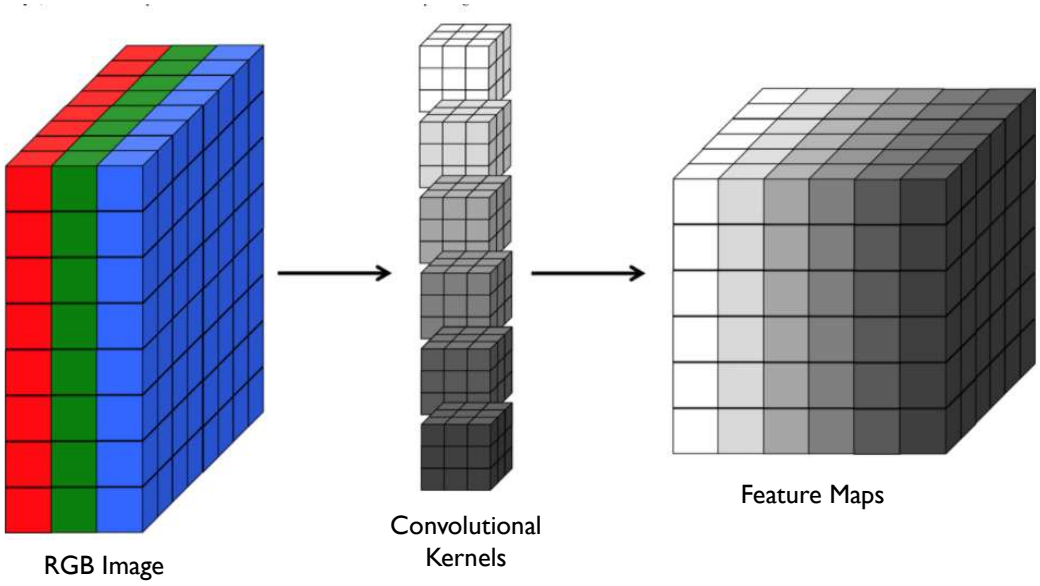
RGB Image

Convolutional Kernels

?

from the book Fundamentals of Deep Learning, © Buduma, 2017

Weight Sharing



from the book Fundamentals of Deep Learning, © Buduma, 2017

Convolution in 2D

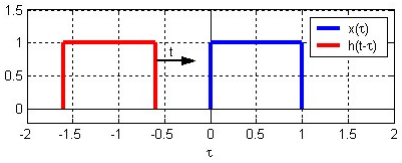
- In 2D, we can think of 2D patterns
- "Mask" could be used to identify one of these patterns
  - Vertical Lines
  - Horizontal Lines
  - Angled Lines
  - Curves
- Mask runs across image and measures match for given feature

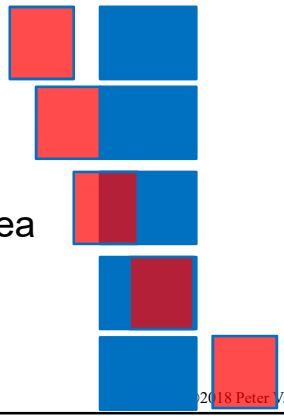
The mechanics of convolution

- Signal processing mechanism for matching filter against a stream

$$[f * g](t) \equiv \int_0 f(\tau)\, g(t-\tau)\, d\tau,$$

- Video http://mathworld.wolfram.com/Convolution.html





- Convolution gives overlap area function of shift
- What shape will it be here?

2D convolution arithmetic

- A guide to convolution arithmetic in deep learning
- https://arxiv.org/pdf/1603.07285.pdf

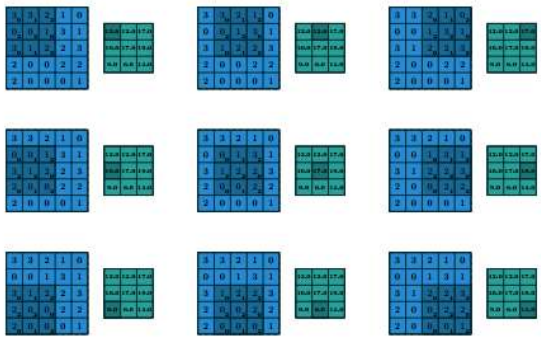
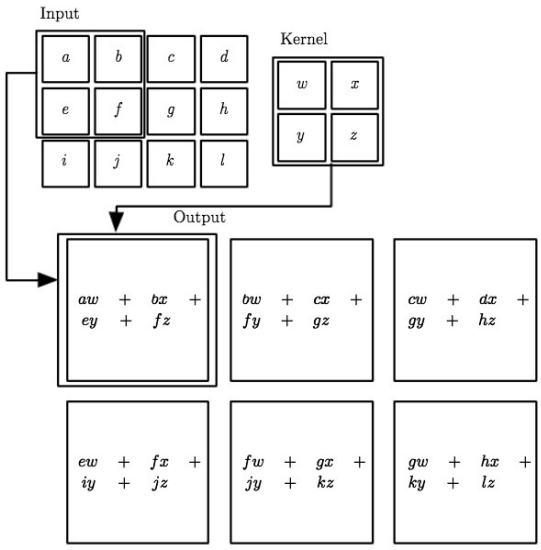


Figure 1.1: Computing the output values of a discrete convolution.

2D convolution arithmetic



from the book Deep Learning, © Goodfellow et al., 2016

- Input is 3 x 4 image
- kernel is 2 x 2 (patch)
- Output is called a "feature map": it is 2 x 3 and composed of the sum of element by element multiplications
- "VALID" padding: apply the kernel only entirely within the image – no padding

2D convolution arithmetic

Input 5 x 5

1	2	1	1	1
1	0	0	0	0
1	0	0	0	0
5	2	5	4	5
3	5	5	5	3

Kernel 2 x 2

-1	-1
1	1

Top left: -1*1 + -1*2 + 1*1 + 1*0 = -1

Feature Map 4 x 4

-1	-3	-2	-2
0	0	0	0
6	7	9	9
1	3	1	-1

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$

from the book Deep Learning, © Goodfellow et al., 2016

2D convolution arithmetic

Online demos for convolution

- Setosa:

http://setosa.io/ev/image-kernels/

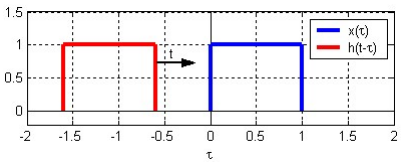
- Stanford CS231n:

https://cs231n.github.io/assets/conv-demo/index.html

And a nice blog from Chris Olah:

http://colah.github.io/posts/2014-07-Understanding-Convolutions/

What about the weights?



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

Gx

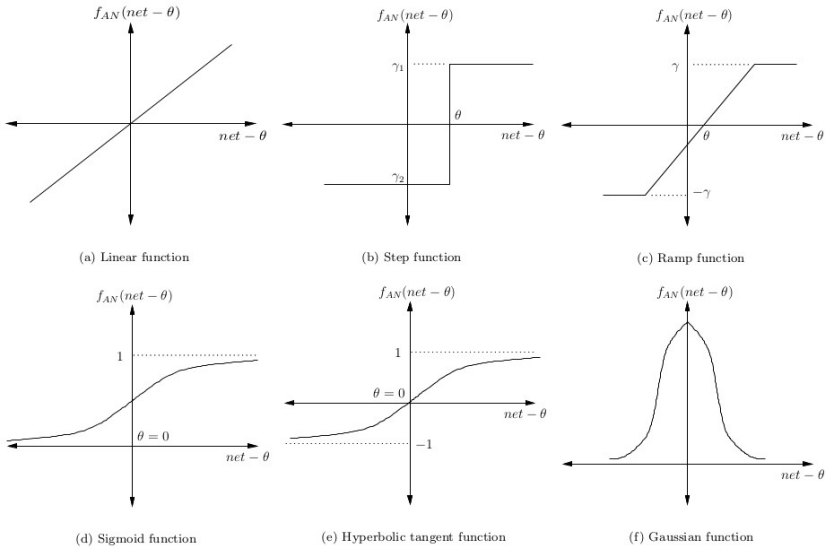
+1	+2	+1
0	0	0
-1	-2	-1

Gy

- Have masks running across the image focusing on a local region
- Where are the weights in all this?
- Is this a linear or non-linear operation?

Activation Functions

- http://www.turingfinance.com/misconceptions-about-neural-networks/

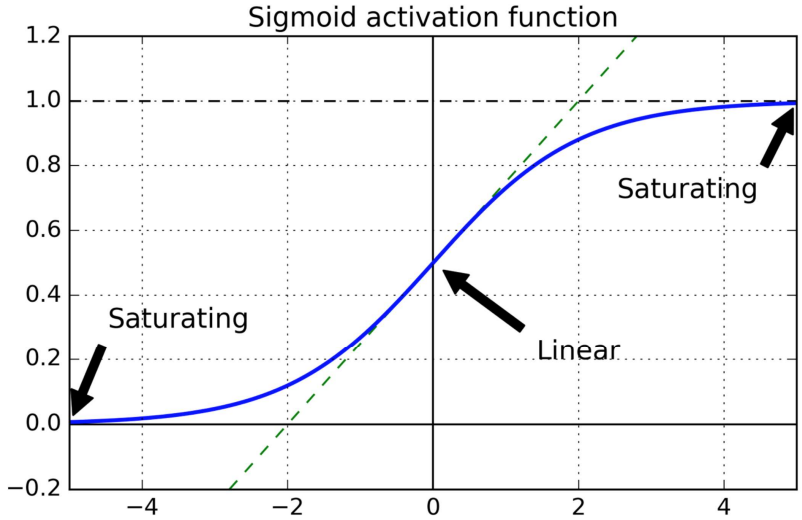


(a) Linear function (b) Step function (c) Ramp function

(d) Sigmoid function (e) Hyperbolic tangent function (f) Gaussian function

Activation Functions

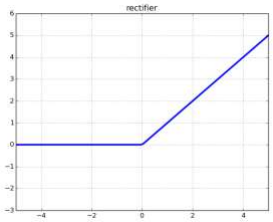
- Could Mother Nature have got it wrong?



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Sigmoid Calculation → ReLU

- Most of the past few decades have focused on sigmoid calculation
- Then use turned to ReLU and its variations
  - "Rectified Linear Unit"
- Softplus
  - Smooth version of this
  - f(x) = ln(1+exp(x))



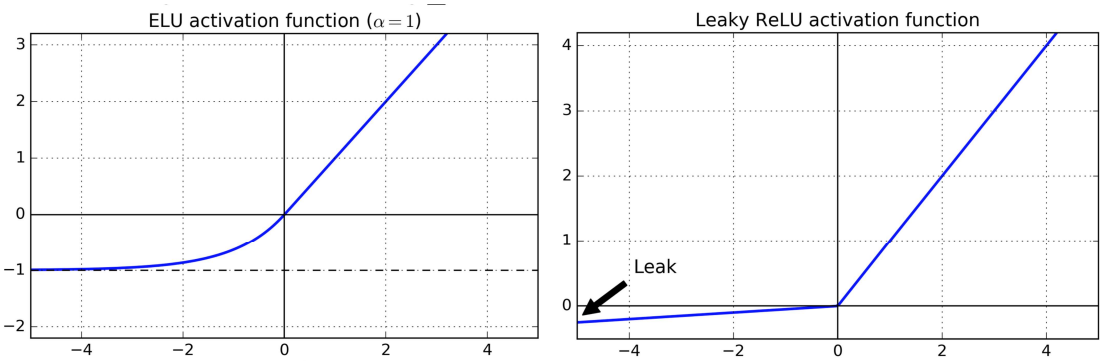
- Why ReLU?

Why ReLU

- Faster to calculate – not a concern for the brain
- Sigmoid gradients at plateaus
  - Tend to be small numbers not 0
  - Small numbers require calculation; not 0
    - With zeros, get sparser data and "it works"
  - ReLU accelerates stochastic gradient descent possibly since it doesn't saturate
- Some of the nodes die and never activate – always 0 vanishing/exploding gradients problem

Nonsaturating Activation Functions

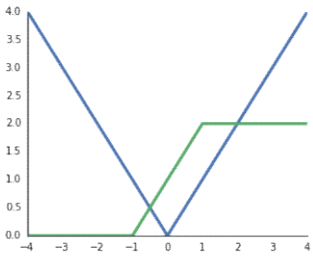
Leaky ReLU, Randomized Leaky ReLU (RReLU), Parametric Leaky (PReLU), Exponential Linear Unit (ELU), SWISH
A long coma is sometimes better than death



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

ReLU

- One reason for multiple layers was to include non-linearities
- Does the ReLU provide non-linearity?
- Answer:
  - 1) can across multiple layers
  - 2) doesn't have to be the only activation

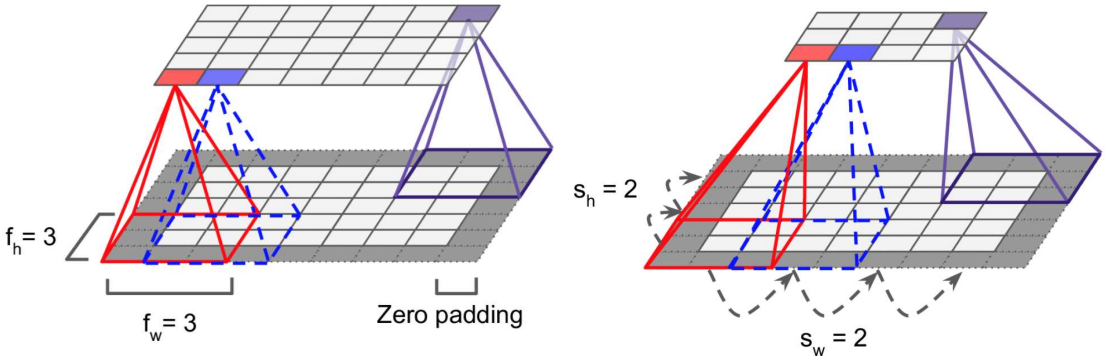


http://stats.stackexchange.com/questions/141960/deep-neural-nets-relus-removing-non-linearity

Padding and stride

3 x 3 patch with ”SAME” (i.e. with zeros) padding on top and bottom

A stride of 2 – in both the height and width-wise directions of the image



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Padding and stride

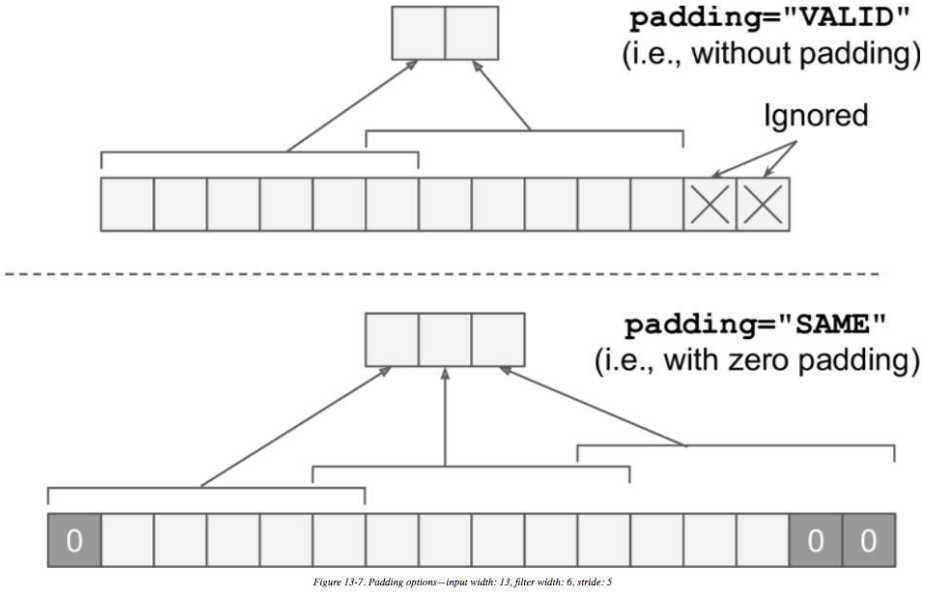


Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Stacking multiple feature maps

- Start with 3 channels (RGB);
- Each convolutional layer *l* has *n* filters (kernels), each of them learning a characteristic and producing its own feature map $k \rightarrow k = 1 \ldots n$ feature maps

Equation 13-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l-1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l-1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Stacking multiple feature maps

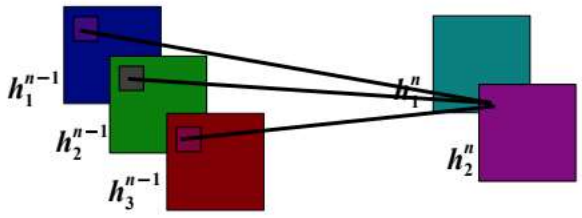
- Repeat about the red boxes once again, it's important…

$$h_j^n = max\left(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map

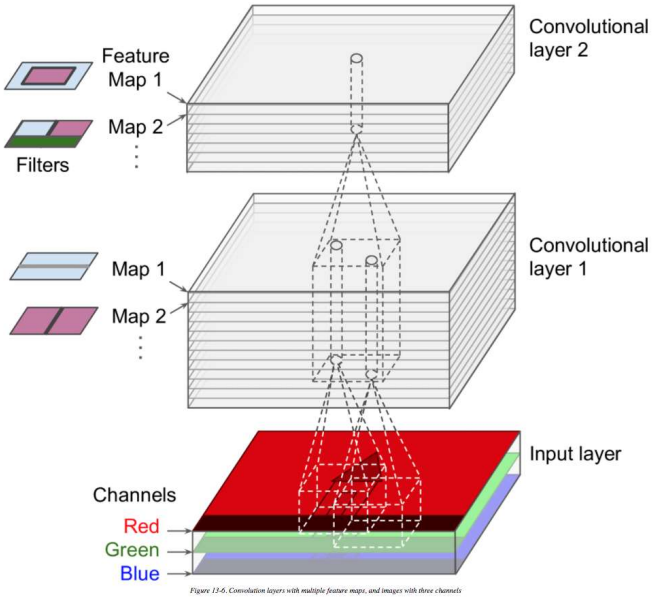
input feature map

kernel



slide 64 of this CVPR 2014 tutorial by Marc'Aurelio Ranzato

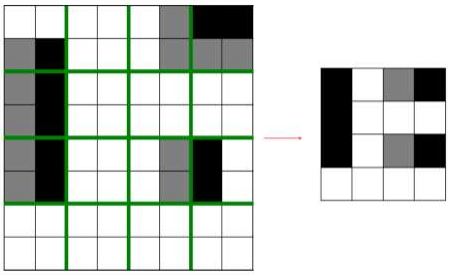
Stacking multiple feature maps



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Max Pooling

- Need to reduce dimensionality
- Add a layer called “Max Pooling”

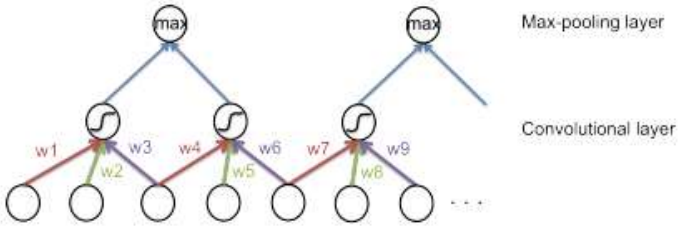


- Takes output, bins them, and takes max
- Locally invariant – small shifts are OK

Image from the book Fundamentals of Deep Learning, © Buduma, 2017

Max Pooling

Tutorial on Deep Learning Part 2: Autoencoder, Convolutional Neural Networks and Recurrent neural Networks by Quoc V. Le @ Google 2015

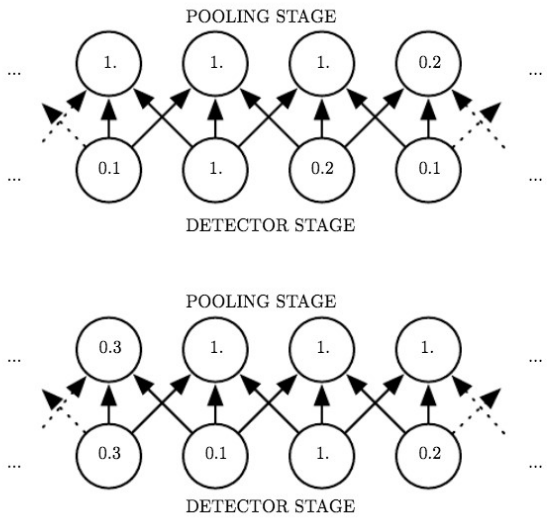


- Interweave max-pooling layer between convolutional layers
- Reduces the scale
- Shift/Translation invariance
  - Events in left or middle → same node
  - Defines local region

Max-Pooling

Introduces Shift/Translation invariance

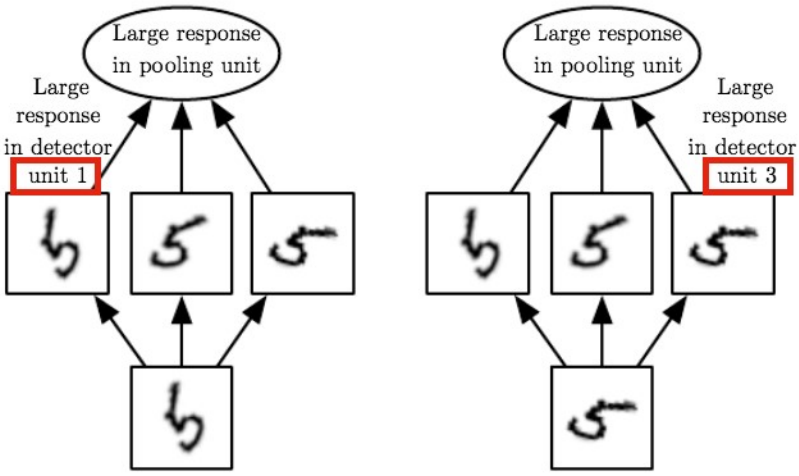
- Bottom row is output of activation function
- Unit that fires is most important (1.)
- (right) effect of a 1 position shift is reduced on top row



from the book Deep Learning, © Goodfellow et al., 2016

Other Pooling: across multiple channels

Pooling over multiple feature maps with separate learned parameters: units learn with transformation invariance: fires on 5 regardless of angle



from the book Deep Learning, © Goodfellow et al., 2016

CNN architectures: Stacked layers of functions

Choices i.e. hyperparameters:

- for layers: width, height, depth
- for filters/kernels: affect learned features
  - spatial extent *(w,h)*, depth, stride, zero-padding
- pooling: max-pooling or something else
- activation
- regularization
- initialization
- batch-normalization
- optimizer

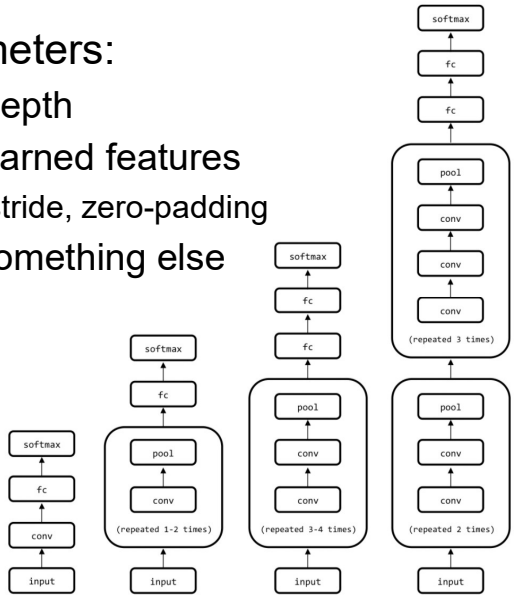
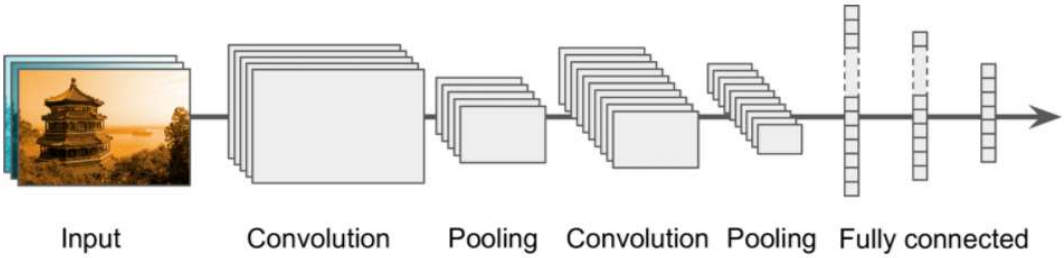


Image from the book Fundamentals of Deep Learning, © Buduma, 2017

CNN architectures: Stacked layers of functions



- Convolutions – filter dimensions, stride
- Multiple shapes, sequences
- Max pooling steps
- Fully-connected layers

from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Deep Neural Network Details for Implementation

Training CNN deep learning networks

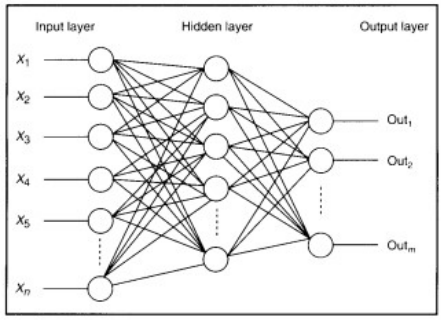
- Initializing the weights – Xavier & He initialization
- Batch normalization

Backpropagation Algorithm

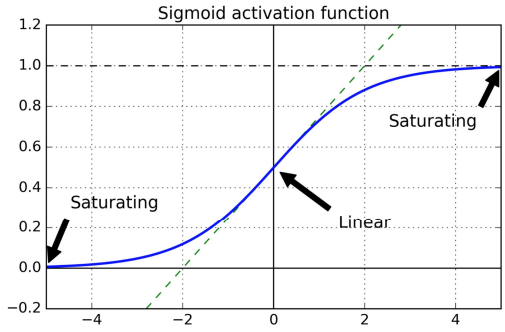
- Initialize weights random & small—why?

Backpropagation Algorithm

- Initialize weights random & small—why?



- N(0, 1/sqrt(#inputs))
- "Symmetry Breaking"



Vanishing and exploding gradients

- Often gradients get smaller and smaller as backprop progress to the lower layers
- Glorot & Bengio (2010) – major suspects:
  - Sigmoid activation (remember its mean = 0.5)
  - Weight initialisation - How do you think this was being done?
  - Combination of activation function and initialization meant variance outputs each layer > variance inputs each layer

Xavier & He initialization

- Equalize variance of outputs of each layer to its inputs
- Equalize variance of gradients after going through and coming back through layer
- How? --> random initialization of weights using:

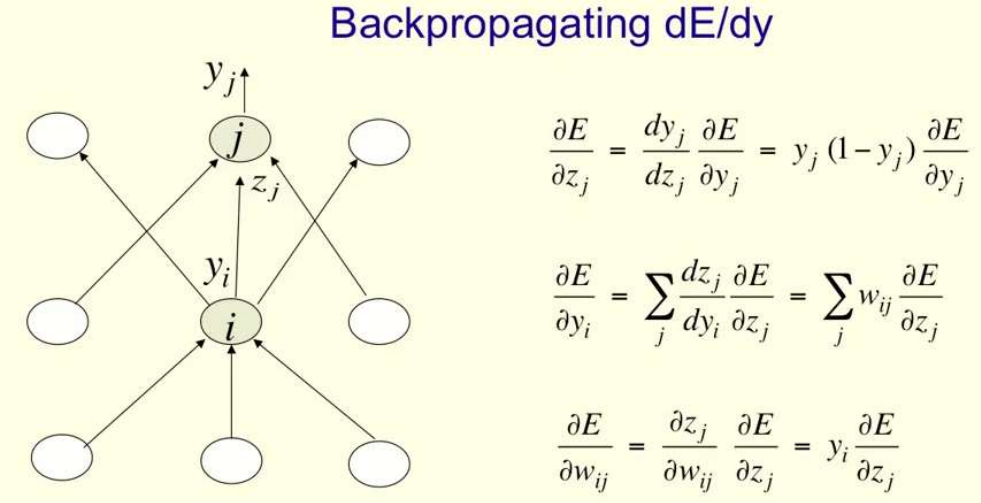
Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- Speeds up training of deep networks

from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Backpropagation on one slide

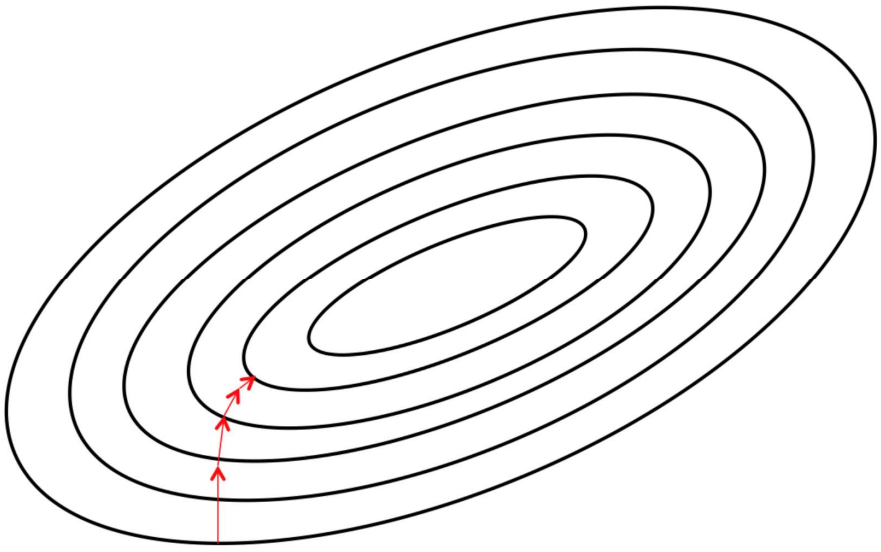


- Just the chain rule
- Hinton, Srivastava and Swersky slides

Backpropagation Algorithm

- Initialize weights random & small
- Loop
  - For each training example
    - Put into network
    - Compute network outputs
    - For each output unit k
      - $d_k = y_k (1-y_k)(t_k - y_k)$
    - For each hidden unit
      - $d_h = o_h (1-o_h) \sum_{k=out} d_k w_{hk}$ where o_h = output of hidden
    - Update weights $w_{ij} \leftarrow w_{ij} + \Delta\alpha\, d_j\, x_{ij}$

Gradient Descent



from the book Fundamentals of Deep Learning, © Buduma, 2017

Gradient Descent – it's slow

Why?

Gradient Descent – it's slow

1) Need to perform gradient descent over the entire network of weights
   - Multiple layers x wide inputs in many cases
     - 100x100 images → 10K weights from input
2) Not guaranteed convexity (global min)
   - What to do about this?
3) Sensitive: use slow learning rate
4) Present inputs & outputs sequentially
   - Epoch = one round through data
   - May need 1000s of epochs

Batch Mode

- Initialize all the weights w_i
- Loop
  - Present each training instance to the network
    - Input x vector
    - Compute the output y
  - Compare y to the t from the training data
  - Compute Δw_i $\quad \sum_{k=inputs}(t_k - y_k)\,(-x_{ki})$
  - Update the new w_i weights
- Epoch = presentation of all training data
- Batch mode updates after each epoch

How to Speed Gradient Descent?

- Idea: vary amount of data used to compute the perfect gradients

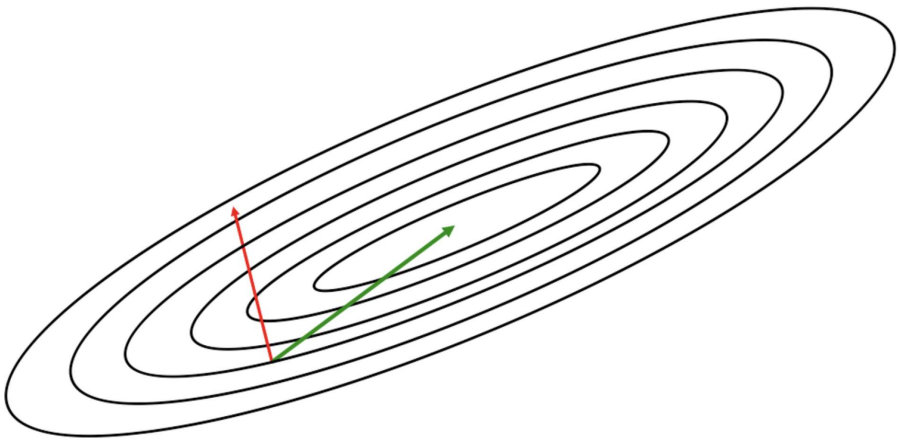
1) Batch gradient descent
   - Use epoch of data to update weights
   - Can't update with new data coming in
2) Stochastic gradient descent
   - Update weights with each training sample
   - Wrong & overshoots, but OK for small steps
3) Mini-batch
   - Update weights every n samples n ~ 256

Batch vs. Incremental Grad. Desc.

- Batch
  - + Good for well behaved error surfaces
  - - Slow to update
  - + Can get stuck in saddle-points (flat parts) and local minima
- Incremental (stochastic, "online", but also mini-batch)
  - + Faster convergence
  - + Can help avoid local minima
  - - Lose the guarantees of convergence

Transforming Inputs

- Scale the inputs so they are fairly equivalent in scale

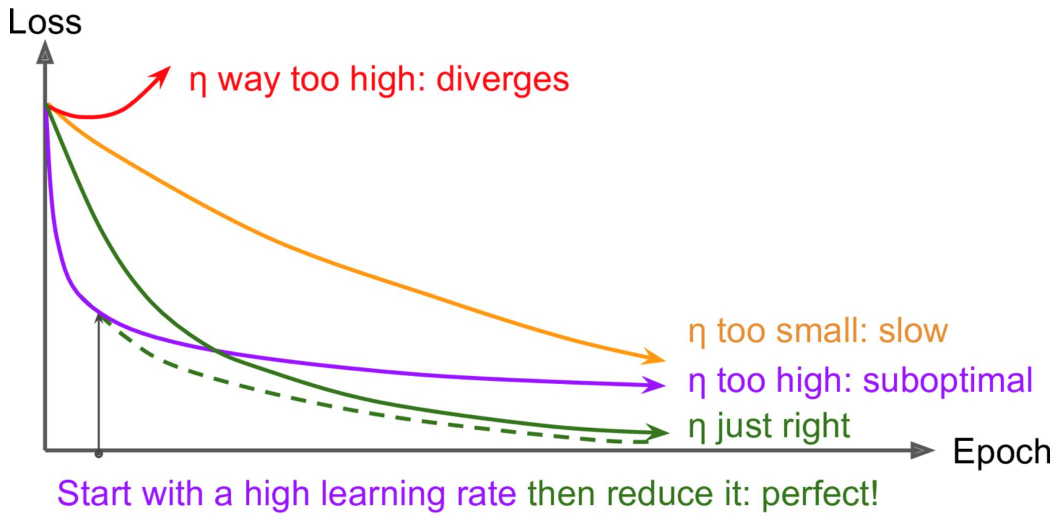


- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.7105&rep=rep1&type=pdf
- https://en.wiki2.org/wiki/Gradient_descent

from the book Fundamentals of Deep Learning, © Buduma, 2017

How to Speed Gradient Descent?

How to Speed Gradient Descent?



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

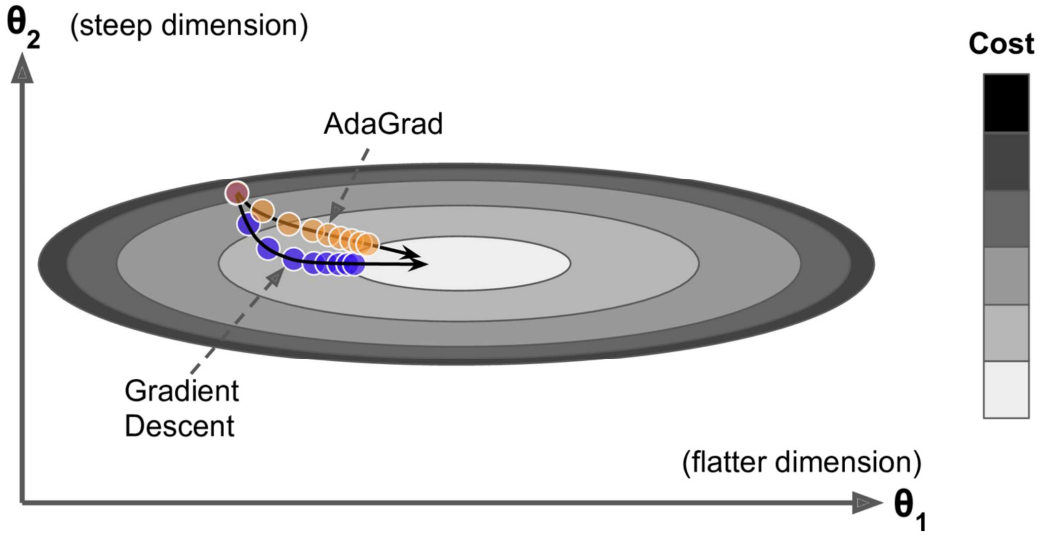
Momentum

- Simple extension of SGD – used for many years
- Helps in the case of a long, narrow valley (SGD slow)
  - Accelerate along dimensions in which gradient consistently points in the same direction
  - Slow down along dimensions where the sign of the gradient continues to change
- How? Keep track of past parameter updates with exponential decay

Adagrad & Adadelta Method

- Adagrad
  - Provides a gradient descent helper
  - Adjusts the step size based on the slope:
  - Large gradients = small learning rate & vice versa
  - Uses the stochastic updating
  - Applies different learning rate for each parameter separately
  - Incorporates past gradients as well
  - Accumulation with sum of squared past gradients:
    - learning rate continues to shrink - can → 0 change
- Adadelta
  - Fix to Adagrad using window of past gradients that fixes the → 0 issue

Adagrad & Adadelta Method



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

How to Speed Gradient Descent?

- Speed up the learning rate
  - "Momentum": exponential weighted average
    - $\mathbf{v}_i = m\,\mathbf{v}_{i-1} - \varepsilon\, g_i$
    - $\theta_i = \theta_i + \mathbf{v}_i$
    - Note i → i^{th}update
  - Nesterov method is version of this
  - Heuristics
    - Scheduled changes in learning rate based on #epochs
    - Increase learning rate if error is decreasing (annealing)
    - Near end, reduce learning rate but not early

Gradient Descent – it's slow

Why?

How to Speed Gradient Descent?

- Momentum optimization,
- Nesterov Accelerated Gradient,
- AdaGrad & Adadelta
- RMSProp
  - Decay parameter, like Adagrad accumulate squared past gradients
- Adam optimization.
  - Momentum and RMSProp combined

How to Speed Gradient Descent?

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \epsilon}$

- t represents the iteration number (starting at 1).

This only for information purposes

3 and 4 are correction hyperparameters to account for initialization bias

from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Batch Normalization

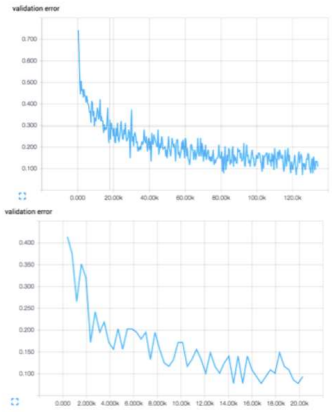
- What if the if the outputs of the neurons in the bottom part of the network shift?

Batch Normalization

- What if the if the outputs of the neurons in the bottom part of the network shift?
- What would happen to the upper layers?

Batch Normalization

shifts in the distribution of output from the neurons can have impact all the way up from the bottom-most to the top – like stacked boxes



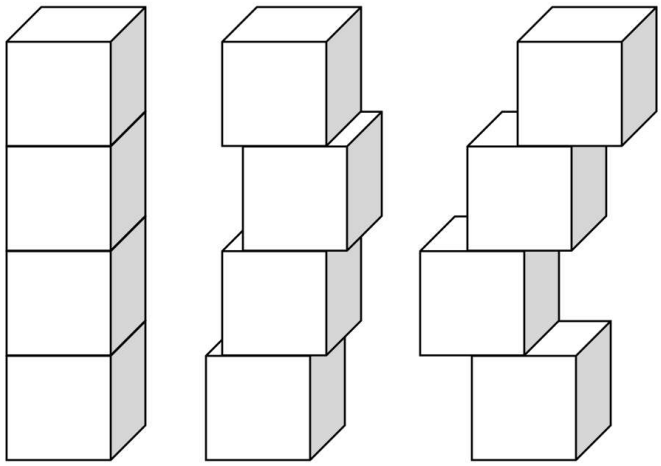


Figure 5-15. When blocks in a tower become shifted too drastically so that they no longer align, the structure can become very unstable

from the book Fundamentals of Deep Learning, © Buduma, 2017

Batch Normalization

- What if the if the outputs of the neurons in the bottom part of the network shift?
- What would happen to the higher layers?
  - Would have to remember old data
  - Would have to accommodate new data
- How could you solve the problem?

Batch Normalization

- Take logits out of layer before go into the non-linearity
- Z-Normalize each tuple of vector across all examples of mini-batch
- May have to modify the momentum calculations
- Claims:
  - Avoids need for regularization & dropout
  - Several fold faster learning

Overfitting

- Major problem in neural networks
- Lots of parameters to tweak
- How would you avoid it?

Avoid Overfitting

Regularization: penalize large weights

- Add an extra loss term for the overall model loss function to optimize
  - L_1 or L_2
  - ”Weight-decay”
  - *Error* + λ* *f(Weights)* -> each weight $\frac{1}{2}\lambda\ w^2$
  - Need to modify the derived weighting
  - Widely used
- ”Max-norm”: constrain incoming weights at each neuron by ”clipping” them such that the L_2 norm is less than *r* - in gradient descent, project vector back

Avoid Overfitting

Share weights

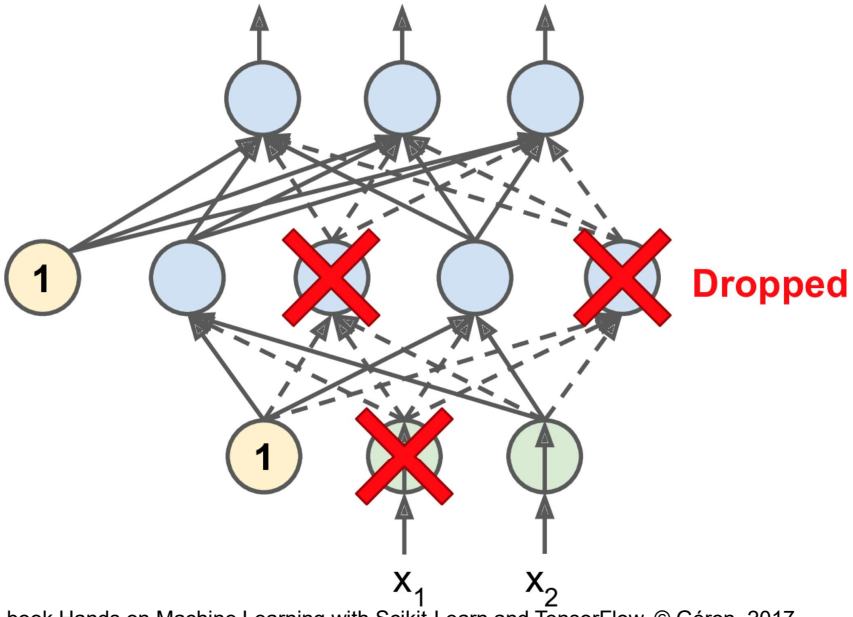
- We are already doing that as part of CNN

Avoid Overfitting

Dropout: randomly nodes & connections during training

- Randomly remove half of hidden nodes
  - Actually set the weights to 0
- Train network normally with updates
- Restore the nodes for next mini-batch

Avoid Overfitting - Dropout



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Avoid Overfitting - Dropout

Dropout (cont.):

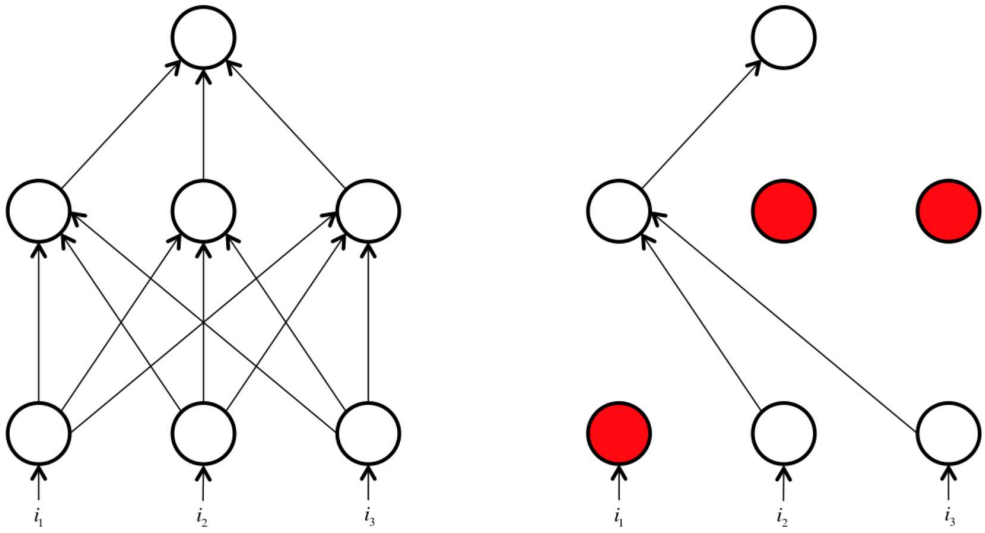
- Randomly remove half of hidden nodes
  - Actually set the weights to 0
  - But if you set half the weights to 0, what will happen to the subsequent activation?
- How to fix it?

Avoid Overfitting - Dropout

Dropout (cont.):

- Randomly remove half of hidden nodes
  - Actually set the weights to 0
  - But if you set half the weights to 0, what will happen to the subsequent activation?
- How to fix it?
  - When you run the trained model on the test data, multiply each neuron's input weight by the "keep probability" (1-p)
  - Or, during training, divide the output weights by the "keep probability" [preferred]

Avoid Overfitting - Dropout



from the book Fundamentals of Deep Learning, © Buduma, 2017

Avoid Overfitting

Stop early by tracking error as f(epoch)

- ? Use model averaging (like bagging)
- ? Use cross-validation to figure out when training error increases

Hyperparameter tuning:

- How do you know what is a good hyperparameter choice?

Hyperparameter tuning:

- How do you know what is a good hyperparameter choice?
- Capacity: ability to fit a wide variety of functions

Hyperparameter tuning:

- Training error – lower is better
- Generalization error – even lower is better, and you don't want this to start increasing
- Resource constraints – computing power, runtime, memory

Hyperparameter tuning:

- training error – lower is better
- generalization error – even lower is better
- resource constraints – computing power, runtime, memory

The effective "capacity" of a model depends on:

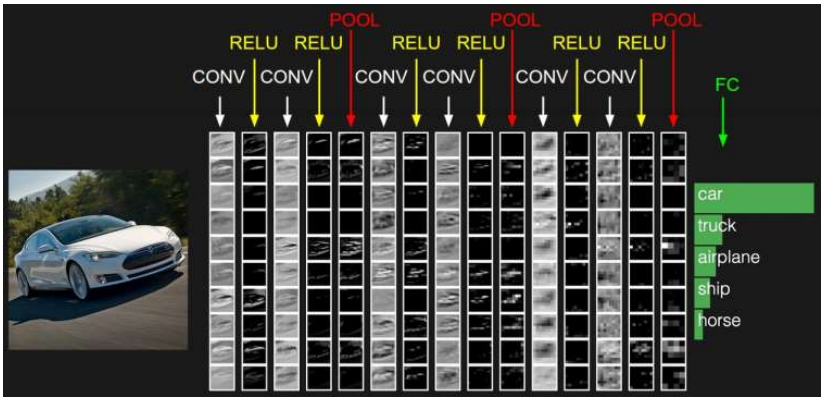
- representational capacity of model (e.g. # layers, depth)
- success of algorithm to minimize cost function in training
  - the ability to discover the functions that best give representation
- regularization of the cost function and training procedure

Hyper-parameter	Increases capacity when	Reason	Caveats
Number of hidden units	increased	Increasing the number of hidden units increases the representational capacity of the model.	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model.
Learning rate	tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure.	
Convolution kernel width (i.e. spatial extent, w x h)	increased	Increasing the kernel width increases the receptive field and therefore the number of parameters in the model = more distinctive features	A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost
Implicit zero padding	increased	Adding implicit zeros before convolution keeps the representation size large.	Increases time and memory cost of most operations.
Weight decay coefficient	decreased	Decreasing the weight decay coefficient frees the model parameters to become larger	
Dropout rate	decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set.	

from the book Deep Learning p.426, © Goodfellow et al., 2016

Putting it all together

- http://cs231n.github.io/convolutional-networks/



Deep Learning Toolsets & Architectures

Existing DeepLearning Tools

- Theano (U Montreal) more for research
- Torch (Facebook) written in Lua
- Caffe (Berkeley)
- Neon
- TensorFlow (Google)
- CNTK (Microsoft)
- MXNet (Apache)
- Front-ends
  - Keras (Tensorflow , CNT, Theano)
  - PyTorch (Torch)

TensorFlow

- Released in 2015 by Google – used by Google Brain
- APIs for Python C/C++ Java & Go
- Tensor = multidimensional array
  - How data are stored and enter into computations
- Includes TensorBoard for visualizing learning through the network graphs, scalars and histograms
- Works out-of-the-box on multiple CPUs and GPUs with acceleration (NVidia CUDA GPUs)
- Keras front-end is now built in: tf.keras
- current release 1.11

Resources

- TensorFlow:
  - https://www.tensorflow.org/tutorials/
  - https://developers.google.com/machine-learning/crash-course/
  - https://github.com/tensorflow/models
    https://www.tensorflow.org/hub/
  - https://keras.io/applications/
- Kaggle kernels
  - https://www.kaggle.com/kernels
- Model repositories, etc.
  - https://mxnet.apache.org/model_zoo/index.html
  - https://modelzoo.co/
  - https://modeldepot.io/

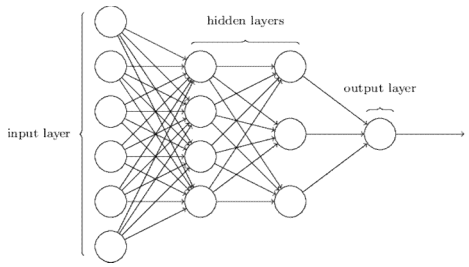
Types of layers in a CNN

- Convolution
  - Activation, Batch Normalization
- Pooling
- Dropout
- Fully Connected

Remember – what constitutes a ”layer”?

Layers

We have gone from this:



to this:

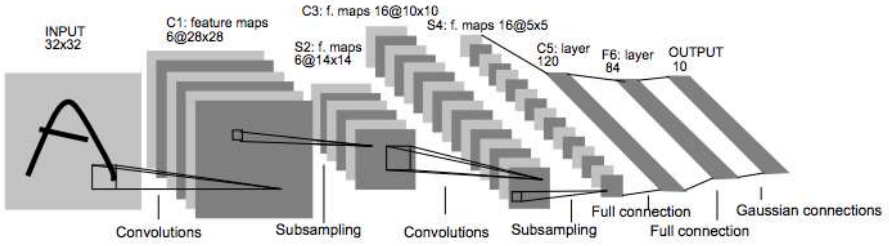


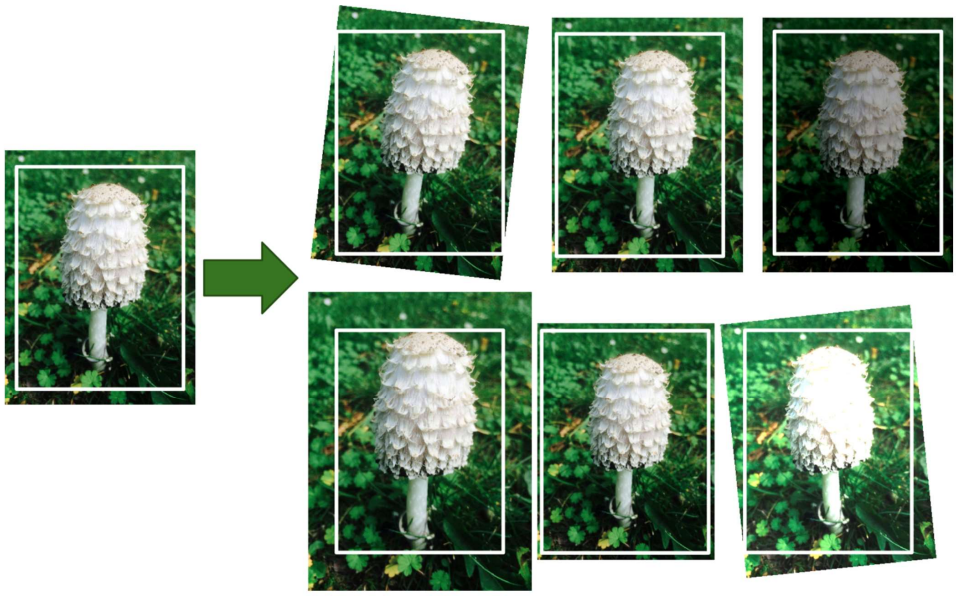
Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Convention: layers are the ones with weights

CNN preprocessing pipelines

- normalize the images – all images to same scale
  - all to [0,1] or [-1,1] but not mixing [0,1] **and** [0,255]
- most architectures require to have standard size
  - cropping, scaling to fit – dynamic pooling can adjust
- dataset augmentation
  - reduce generalization error by finding more training samples
- bounding box image segmentation into locations

Data Augmentation



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

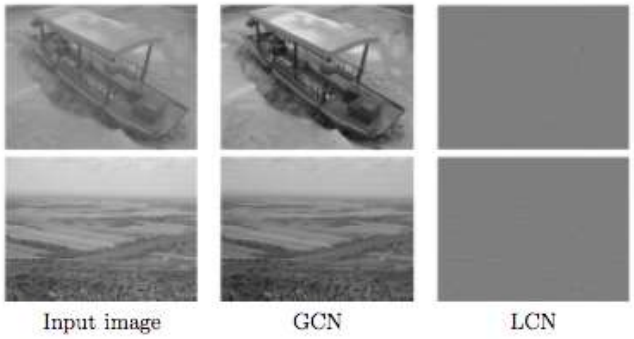
Data Augmentation

Essential to prevent overfitting in small datasets

- Mirroring – horizontal, vertical
- Rotations
- Random cropping
- Shearing
- Local warping
- Width/height shift
- Color shifting – add distortion to R G B channels
- PCA color augmentation – keep tint the same (AlexNet)

Contrast normalization (bright – dark pixels)

- Global Contrast Normalization $X'_{i,j,k} = s \frac{X_{i,j,k} - \bar{X}}{\max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}(X_{i,j,k} - \bar{X})^2}\right\}}$
- Whitening: all Prin. Components of PCA have equal variance
- Local Contrast Normalization



from the book Deep Learning, © Goodfellow et al., 2016

Some useful CNN architectures

- LeNet-5 – (arguably) the first CNN architecture - digits
- AlexNet – unparalleled performance on ImageNet
- GoogLeNet – Inception module:optimal local ConvNets
- ResNet – add skip connections to learn on residuals

CNN architectures: LeNet-5 1998

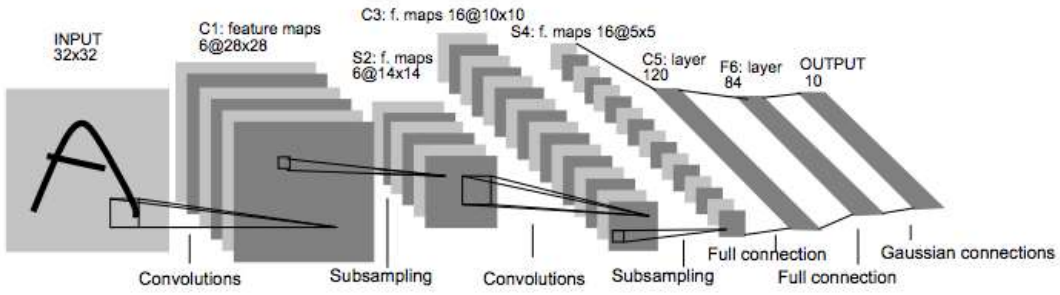


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet-5 schema from "Gradient-based learning applied to document recognition", LeCun et al, 1998

CNN architectures: LeNet-5 1998

Layer	Activation output		Number of
	Shape	Size	parameters
Input	(32,32,3)	3,072	0
Convolution Layer 1 (f=5, s=1)	(28,28,8)	6,272	208
Pooling Layer 1	(14,14,8)	1,568	0
Convolution Layer 5 (f=5, s=1)	(10,10,16)	1,600	416
Pooling Layer 2	(5,5,16)	40	0
Fully Connected Layer 3	(120,1)	120	48,001
Fully Connected Layer 4	(84,1)	84	10,081
Softmax Layer (output)	(10,1)	10	841

CNN architectures: AlexNet 2012: 17% top-5

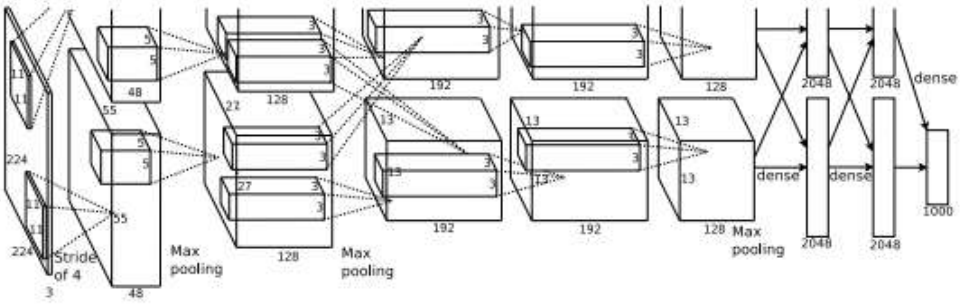


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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.

ImageNet schema from "ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky et al, 2012

CNN architectures: LeNet-5 and AlexNet

Table 13-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

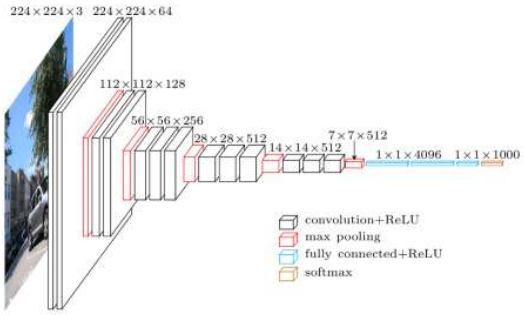
from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

CNN architectures: VGG-16 VGG-19

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					

Table 2: **Number of parameters** (in millions).

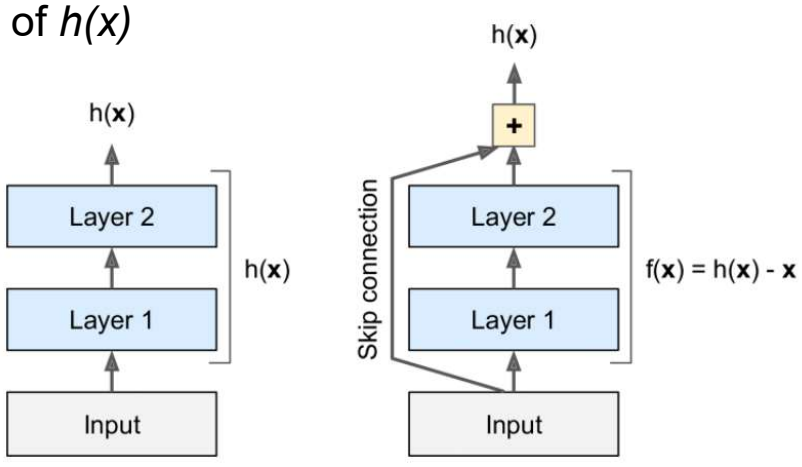
Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144



from the paper Simonyan & Zisserman, 2015 VGG - Visual Geometry Group, Oxford

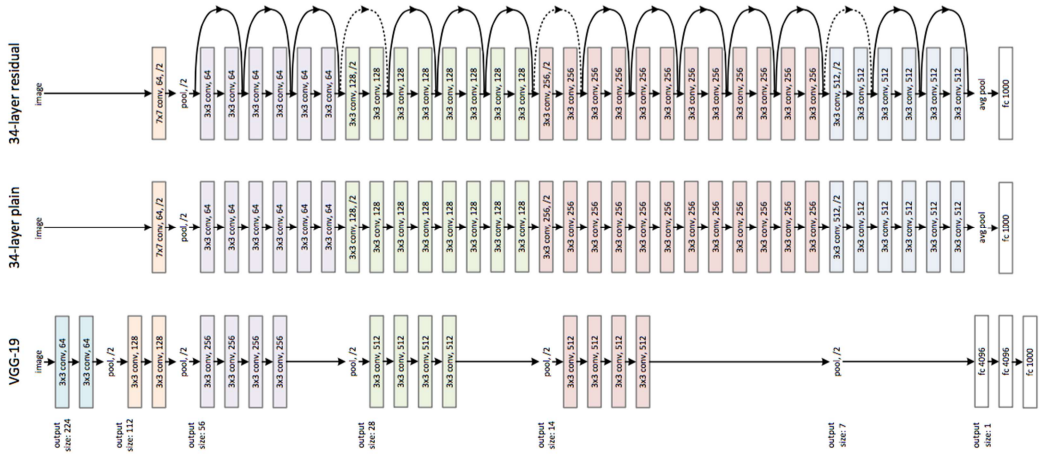
CNN architectures: ResNet 2015: 3.6% top-5

- Model residual learning through skip connections: add *x* to output, network is forced to model *f(x) = h(x) –x* instead of *h(x)*



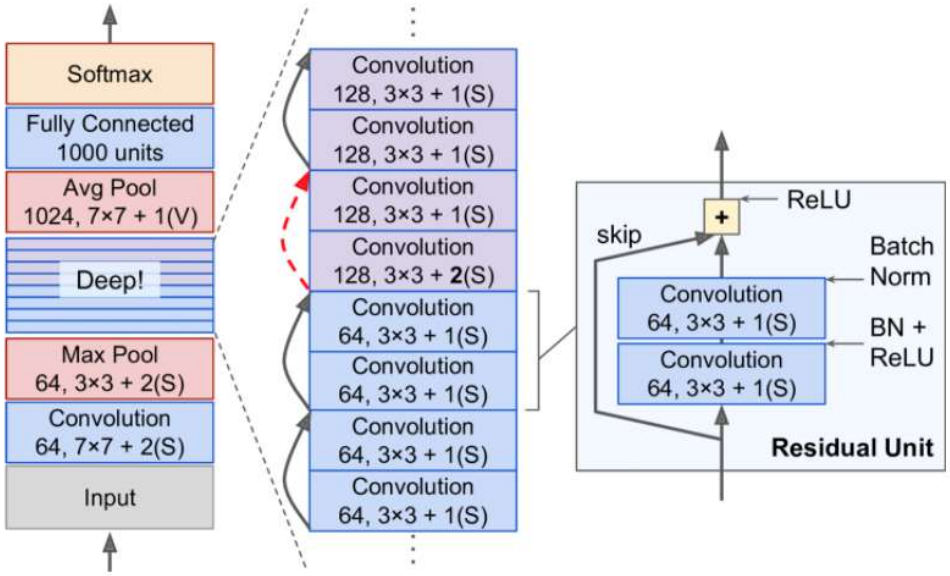
from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

CNN architectures: ResNet 2015: 3.6% top-5



from Deep Residual Learning for Image Recognition, He et al, 2015

CNN architectures: ResNet 2015



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

CNN architectures: ResNet 2015

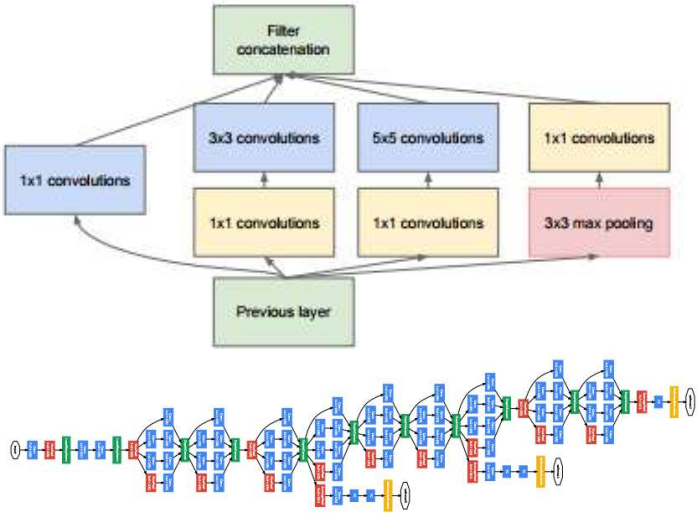
Questions

- What happens if the current layer weights are zero (or at least very small)?
- What constraint on the architecture are there on order to make activations ”skip” layers?

CNN architectures:

- How do we know how many convolutions, pooling layers, fully-connected layers?
- What size filters works best 5x5, 3x3, 1x1?
- With GPUs and distributed/parallel computing why bother – why not do them all?

CNN architectures: GoogLeNet / Inception: 7%



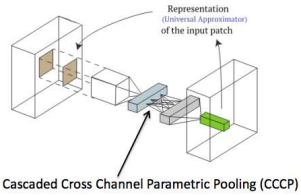
- ”Inception Modules” composed of various sized convolutions
- 1x1 convolutions
- configuration requires patches to align 1x1 3x3 5x5
- Also includes pooling paths
- 22 layers, but
- 12x fewer para-meters than AlexNet

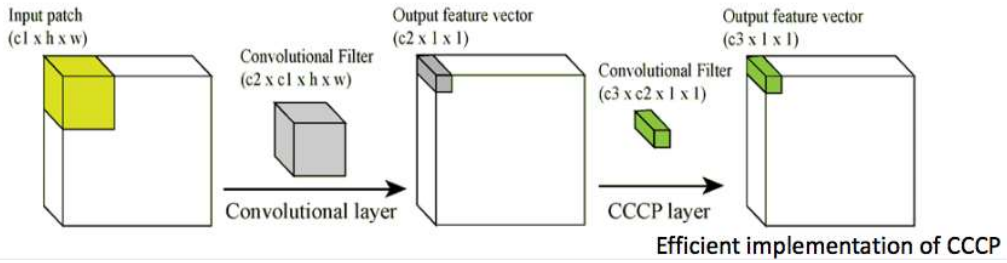
Inception Module composition from Going deeper with convolutions, Szegedy et al. 2014

CNN architectures: GoogLeNet / Inception

1 x 1 convolution

- Network-in-network (Lin et al 2012)
- What’s the point, anyway?
- How is it different from a fully-connected layer?





http://image-net.org/challenges/LSVRC/2014/slides/ILSVRC2014_NUS_release.pdf

CNN architectures: GoogLeNet / Inception

1 x 1 convolution

- What's the point, anyway?
  - increased network size means better performance
  - there is a limit to how deep you can go → "go deeper"
  - dimension reduction
  - adds non-linearity
- How is it different from a fully-connected layer?
  - at the last layer of a network, it isn't
  - any size input – not the full feature map
    - e.g. LeCun says it gets around fixed input size constraint

1 x 1 convolutions

E.g. 6 x 6 feature map and single weight, 3, in a 1 x 1 filter with stride 1 : just multiplying every element in the 6 x 6 map by 3. Why bother?

The point: never are doing it with a flat feature map, but a set of feature maps with depth, say, 20 feature maps all stacked in a 6 x 6 x 20 volume.

1 x 1 convolution filter is then 1 x 1 x 20 - with 20 different weights (i.e. numbers) in the filter to be trained, and each one multiplies its own distinct input feature map.

Go from 20 feature maps of size 6 x 6 down to 10 of them.

Each 1 x 1 convolution is like a mini network, since it has weights to learn and an activation function (no pooling or dropout though).

This is dimension reduction - doesn't it make you think of PCA?

1 x 1 convolutions

Limit to depth of whole network (number of layers)

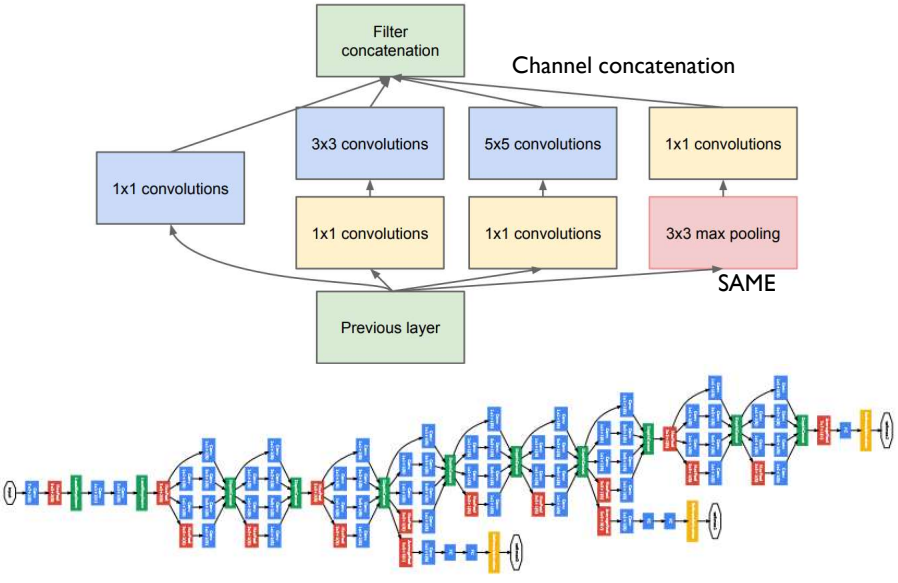
if you just keep stacking more and more feature maps for each new layer complexity would increase, having more and more weights to learn.

- 1 x 1 convolutions allow you to distill the learning and reduce the depth/complexity,
- they add an additional non-linearity, since they are a mini-network
- not many hyper-parameter choices - you just choose where/when to stick them in, and how many filters to use.
- 1 x 1 convolutions are a lot like flattening the feature maps and doing a Fully Connected layer.

If you used them at the end of a CNN, this is true, they would be essentially the same.

- 1 x 1 convolutions can be used with any sized input, since you are just passing them across all the pixels in the input maps/channels.

CNN architectures: GoogLeNet / Inception



Inception Module composition from Going deeper with convolutions, Szegedy et al. 2014

CNN architectures: GoogLeNet / Inception

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

Going deeper with convolutions, Szegedy et al. 2014

Convolutional Neural Networks

- Good
  - Top performance for their domain – image recognition
  - Representational capacity – highly flexible
  - Speed
- Bad
  - No memory → LSTM, memory networks, etc.
  - Fixed input size: good for images and other fixed-length vectors
  - Fixed output size – i.e. categories → extend pre-trained networks, outputs as features
  - Every new technique seems to add more hyper-parameters

Convolutional Neural Networks

- Ugly
  - ”Just made it up” – how to choose architecture?
    - Activation functions are at the heart but we just use what works
    - *“One must be cautious though: although the proposed architecture has become a success for computer vision, it is still questionable whether its quality can be attributed to the guiding principles that have lead to its construction.”* (Going deeper with convolutions)
  - Throwing away information: Is pooling needed?
    - equivariance not invariance
    - reflect the transformation as information
    - disentangle not discard (Hinton)

Transfer Learning

- Not everyone can train an Inception-v3 CNN with ImageNet images for 3 weeks with multiple GPUs
- Three scenarios:

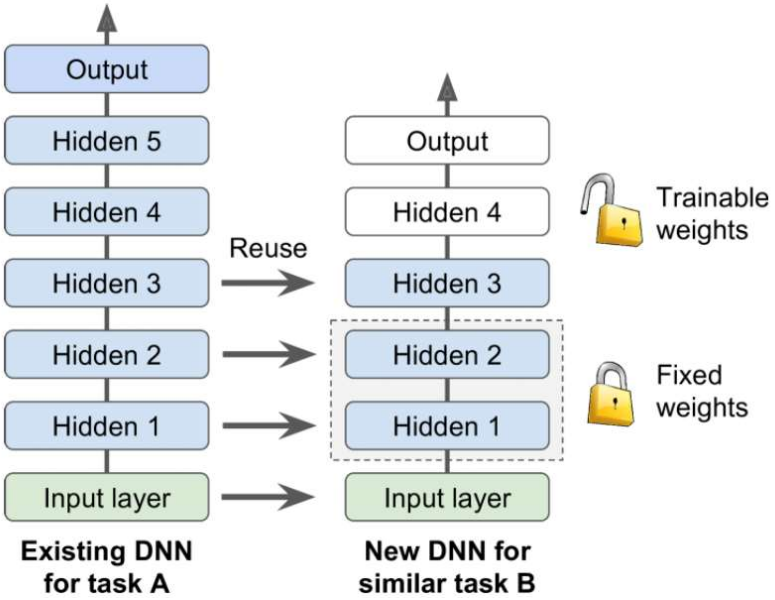
1. Use a pre-trained CNN → move on from checkpoint
2. Fine tune the CNN: replace the classifier at the top of the CNN and continue to train – backpropagation will fine tune the weights; keep some layers frozen
3. Use the CNN as a feature generator
   - Take a trained CNN, remove the last FC layer and use the output layer as features e.g. AlexNet = 4096 features
   - Train a classical linear classifier for the new images

Transfer Learning

Important question: how much data do you have?

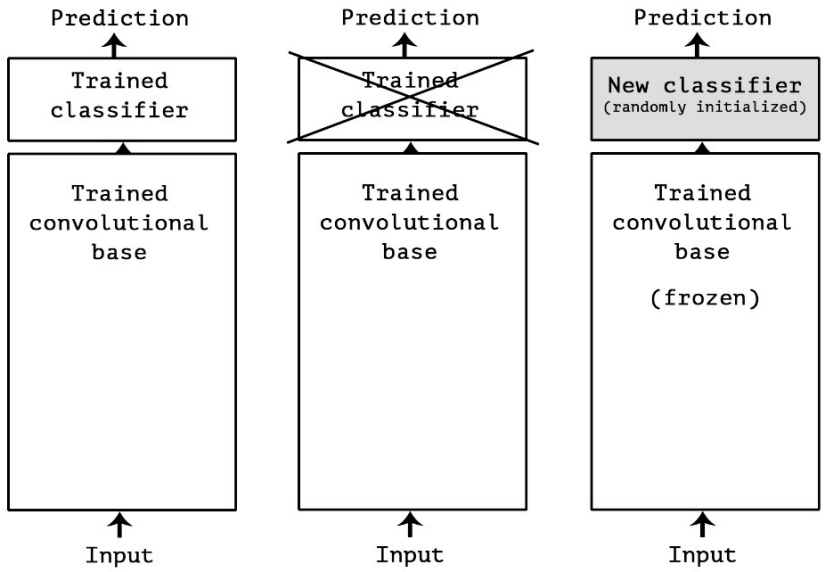
- Lots: Options 1 and 2 – pre-train & move on or fine tune the whole, or parts of, the base
- Not much: Option 3 – use a pre-trained CNN as a feature generator
  - 1. Run base once - save output to disk – use as input layer
  - 2. Re-run the (frozen) base every time – more computations, but allows you to do data augmentation
- Another important question: do you have a GPU?

Pre-trained layers/weights & Transfer Learning



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Pre-trained layers/weights & Transfer Learning



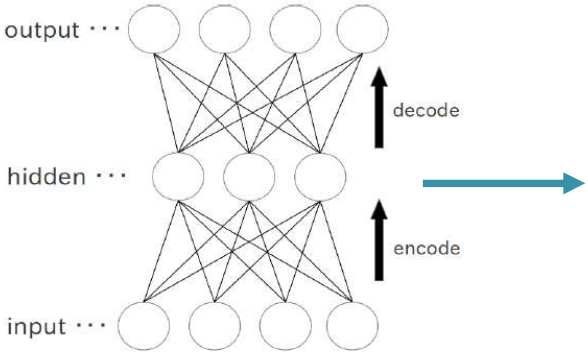
from the book Deep Learning with Python, © Chollet, 2018

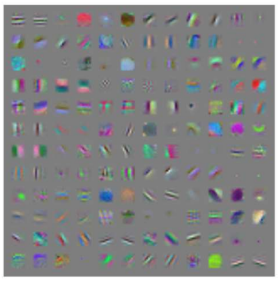
Autoencoder – who cares?

- Encoder: $\boldsymbol{h} = f(\boldsymbol{x})$ Decoder: $\boldsymbol{r} = g(\boldsymbol{h})$
- Pre-learn data
  - Reduces training time
- Perform dimensionality reduction
  - Leveraging the non-linearity of NN rather than the usual linear models (e.g. PCA)
  - Useful for visualization
  - Useful for searches for images, etc.
- Gain insight into how the NN learns

Autoencoder Neural Network

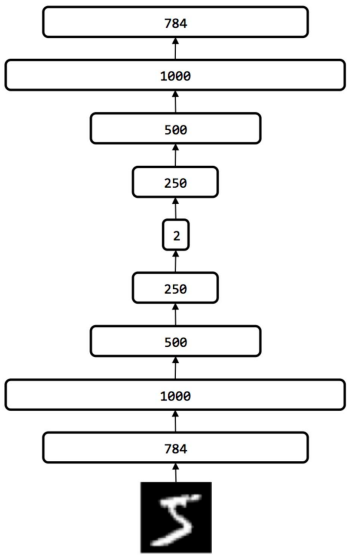
- Way of compressing data; denoising
- Set input = output and learn (not identity function)
- Constrain # hidden nodes: ceil(log2(n)) for n inputs
- Hidden layer learns the most salient pattern





Representation learned with sparsity penalty → gets useful features

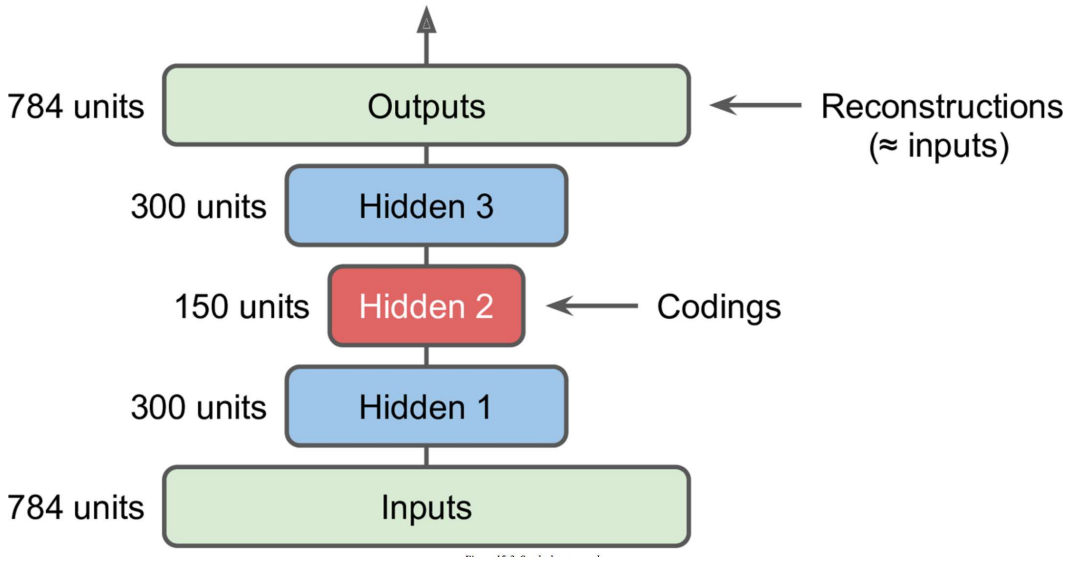
Sparse Autoencoder



from the book Fundamentals of Deep Learning, © Buduma, 2017

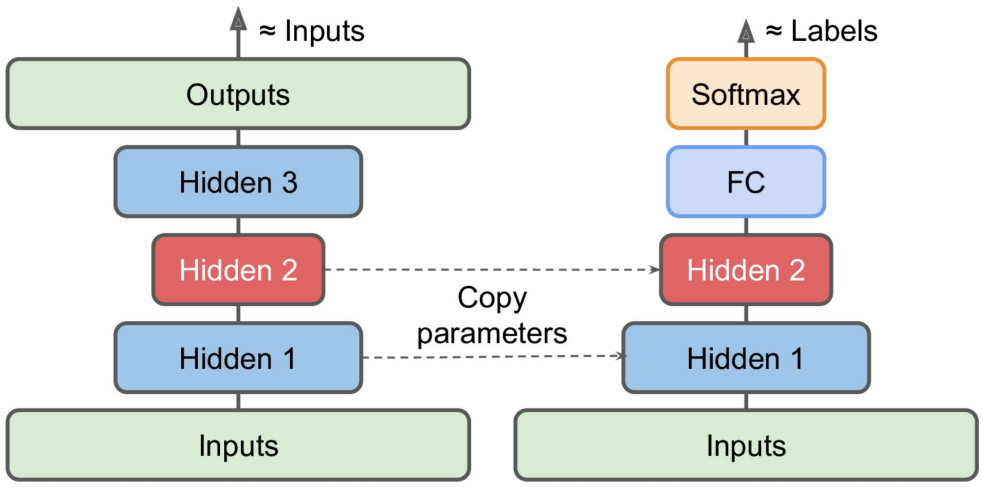
- Same image as input and output
- Learning to create the optimal sparse representation
- Here: MNIST digit

Sparse Autoencoder



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Autoencoder for unsupervised pre-training

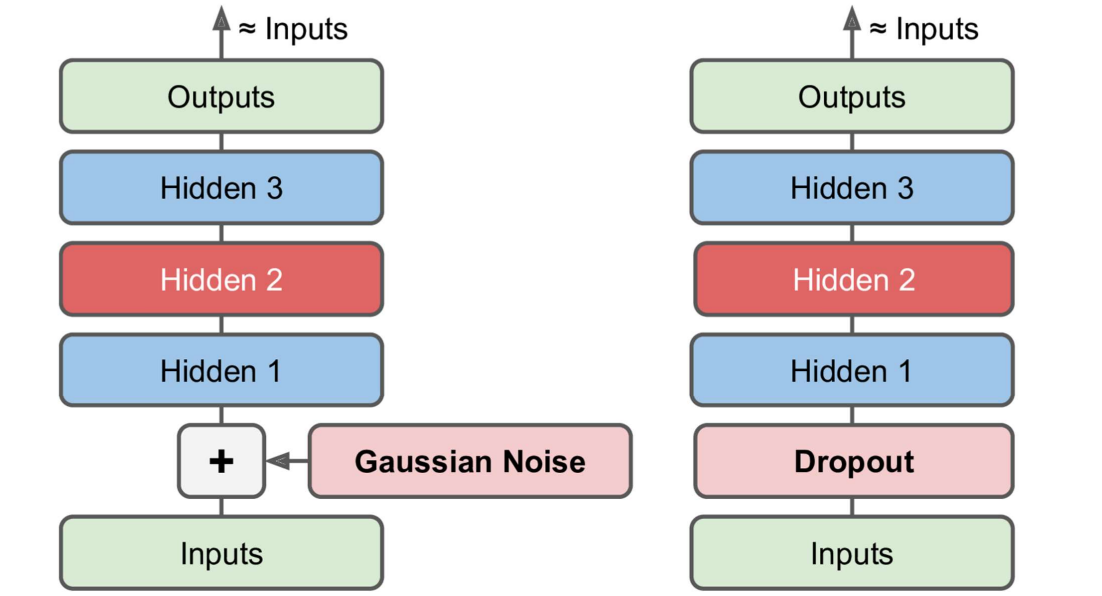


Phase 1
Train the autoencoder
using all the data

Phase 2
Train the classifier
on the *labeled* data

from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017

Autoencoder for denoising



from the book Hands on Machine Learning with Scikit-Learn and TensorFlow, © Géron, 2017