Lecture 12: Deep Learning

Andreas Wichert

Department of Computer Science and Engineering

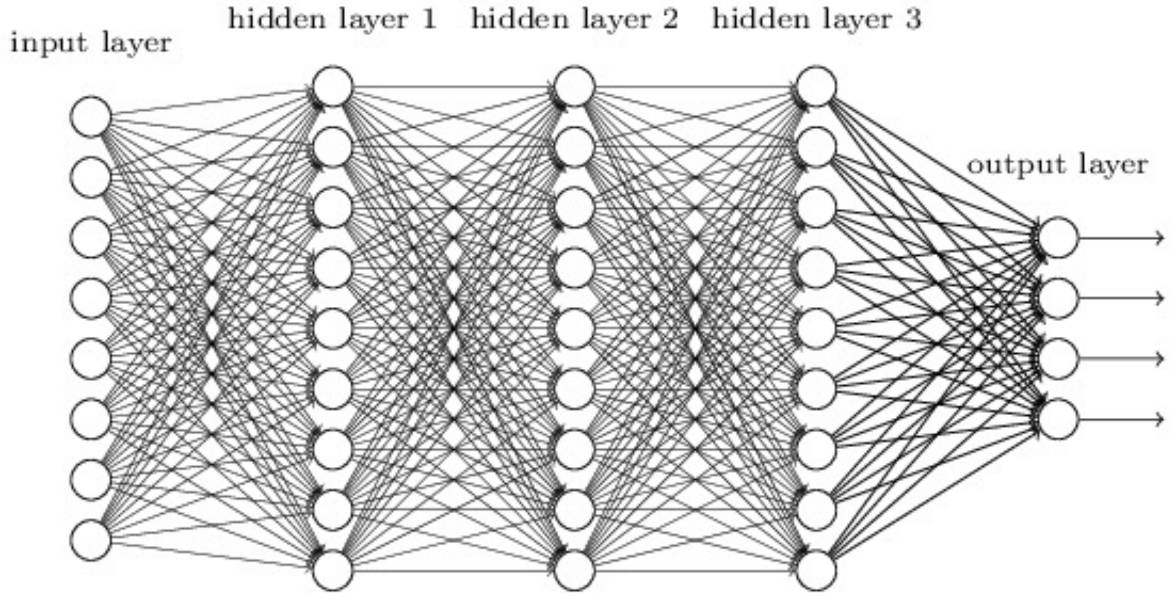
Técnico Lisboa

Deep Learning and Backpropagation

- According to the universality theorem, a neural network with a single hidden layer is capable of approximating any continuous function
- However, attempting to build a network with only one layer to approximate complex functions often requires a very large number of nodes

- Why does deep learning not have a local minimum?
- Is this true?

- It is assumed that an artificial neural network with several hidden layers is less likely to be stuck in a local minima and it is easier to find the right parameters as demonstrated by empirical experiments



Mini-Batch

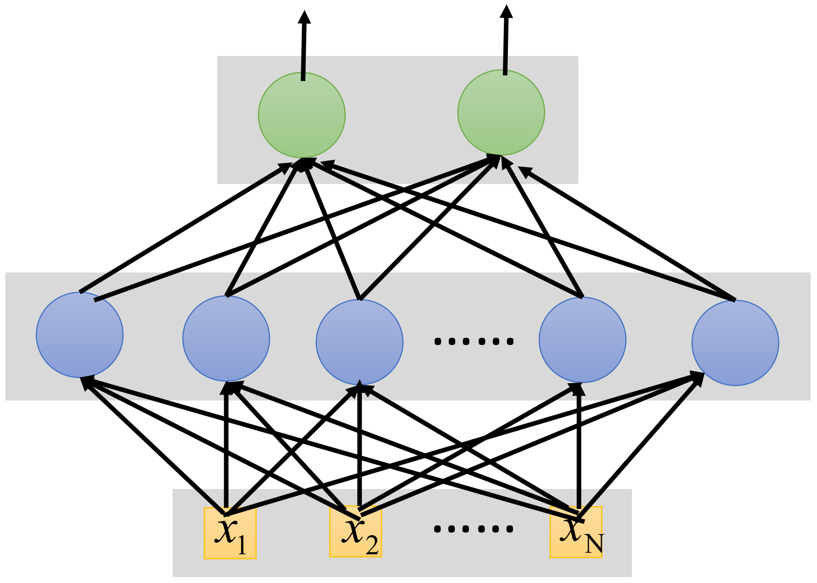
- A gradient is usually determined ver the whole training data set.
- This is called the batch gradient descent
- The model updates parameters after processing the whole training data (one epoch)

- In deep learning the training data set can be too big to fit the computer memory and the gradient cannot be computed efficiently

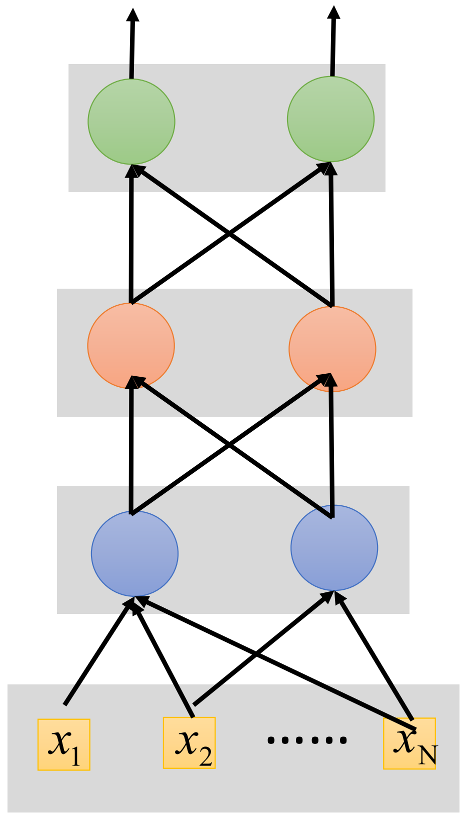
Mini-Batch

- In stochastic gradient descent one updates model parameters after processing every instance, however the model updates are noisy and for big training data sets not computationally efficient
- Therefore, mini batch gradient descent is introduced as a trade-off, learning is preformed in small groups.
  - For example, if the training data has 50000 instances, and the size of a mini batch is set to 50, then there will be 1000 mini batches.
  - They are as well called mini batch stochastic methods or stochastic methods

Fat + Short v.s. Thin + Tall

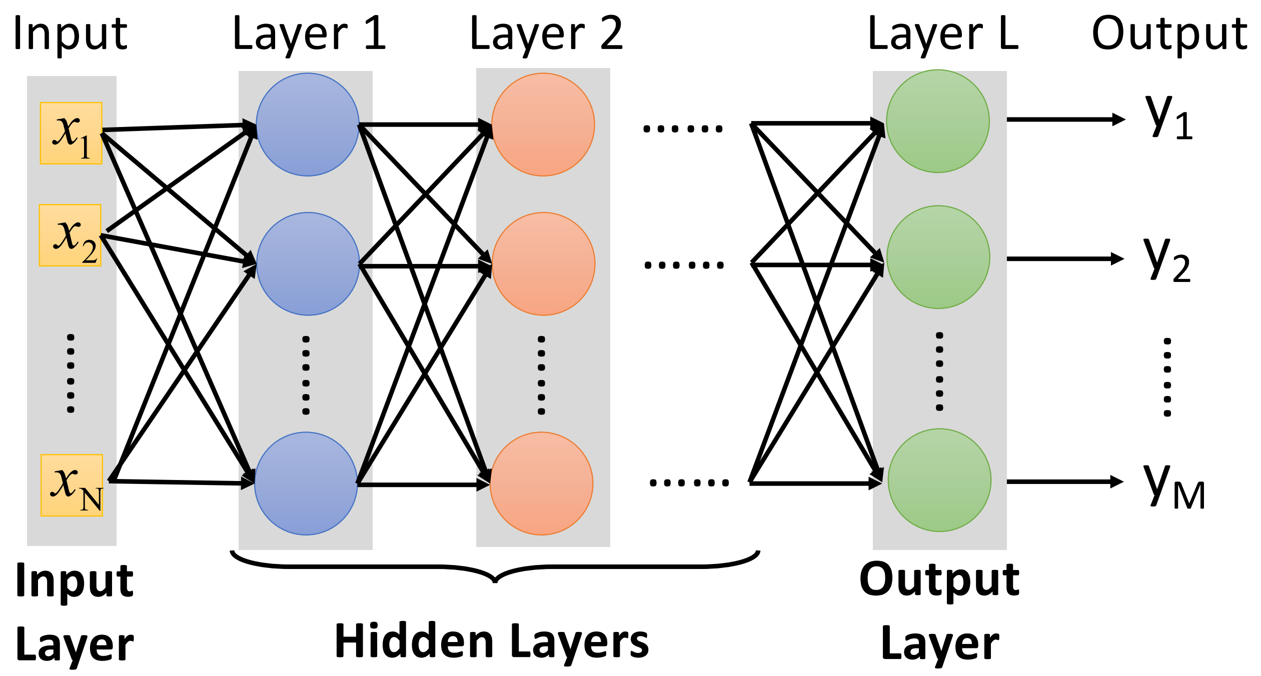


Shallow



Deep

Deep Means Many Hidden Layers



Why: Hierarchical Organization

- The idea of hierarchical structures is based on the decomposition of a hierarchy into simpler parts.
- Hierarchy offers a more efficient way of representing informa- tion. Deep learning enables high-level abstractions in data by architectures composed of multiple nonlinear transformations.
- It offers a natural progression from low-level structures to high-level structure, as demonstrated by natural complexity

Why: Boolean Functions

- Any Boolean function can be represented by a truth table.
- For D variables there are 2^D raws in the table.
  - A table can be described by a disjunctive normal form (DNF).
  - A disjunctive normal form can is described as an OR of ANDs.
- Each AND operation can be implemented by a perceptron.
  - All the AND perceptrons are ordered in one hidden layer.
  - Their output is feed into one perceptron that implements an OR operation. This kind of representation can lead to an exponential explosion of the AND operations.
- One could try to represent the formula by a circuit of bigger depth but lesser complexity.

Can represent Big Training Sets

- Deep networks may be trained on big training sets since they have many free parameters.
- “Flat” networks cannot do it, since for big training lead to the the curse of dimensionality.

Why: Curse of dimensionality

- In a “fat” neural networks there are many hidden neurons, its many outputs represent a high dimensional vector that is classified by the output neurons.
- The high dimensionality of the vector influences negatively the classification of the output neurons during learning and generalization.
    - It corresponds to the curse of dimensionality.
- Deep neural networks can ovoid the curse of dimensionality problem by constraining the number of hidden neurons.

Why: Local Minima

- It is assumed that an artificial neural network with several hidden layers is less likely to be stuck in a local minima as demonstrated by empirical experiments.
- It was commonly thought that simple gradient descent would get trapped in poor local minima.
  - In practice, poor local minima are rarely a problem with large networks but a big problem with small networks.
- Most local minima are equivalent in large networks and close to global minimum Chomoranksa et. al (2015).
  - However this is not true for small networks where bad local minima are present Swirscz et. al. (2016).
- For large networks, the loss function may have a large number of saddle points where the gradient is zero Dauphin et. al (2015).

Why: **Efficient Model Selection**

- To overcome overfitting one has to use a model that has the right capacity.
- However this task is difficult and costly since it involves the search trough many different architectures and parameters.
- Many experiments with different number of neurons and hidden layers have to be done.
- Instead one choses a deep over-parameterized neural network.
  - The search for the model of the right capacity is done by a search for the correct regularization value.
  - This kind of search is easier to implement

Why: Efficient Model Selection

- This kind of search is easier to implement since for example the l_2 or l_1 regularization are described by **one** variable α

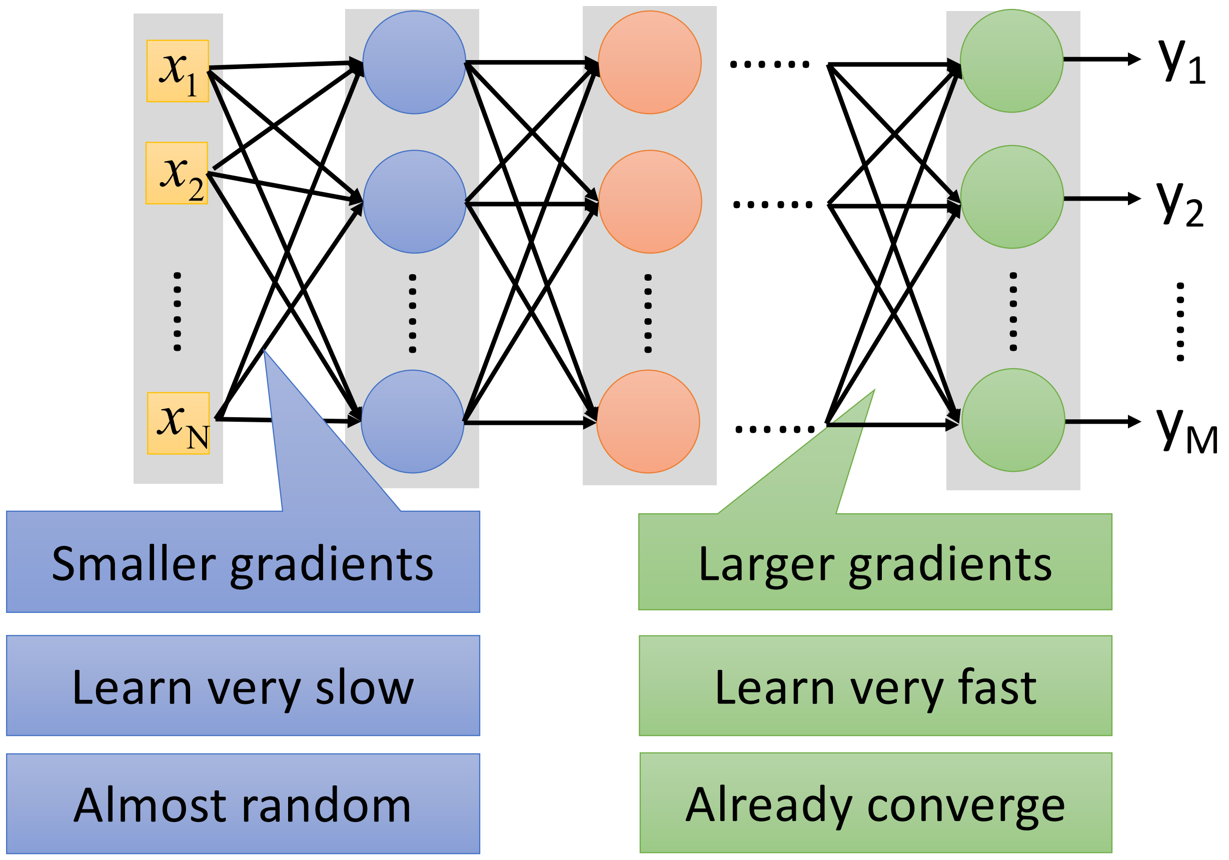
$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot w_j \qquad \Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot sign(w_j).$$

- The search for the correct model complexity can be done efficiently, by em-pirical experiments for searching for just one correct α value.
- This leads to a model with the optimal predictive capability is the one that leads to the best balance between bias and variance.

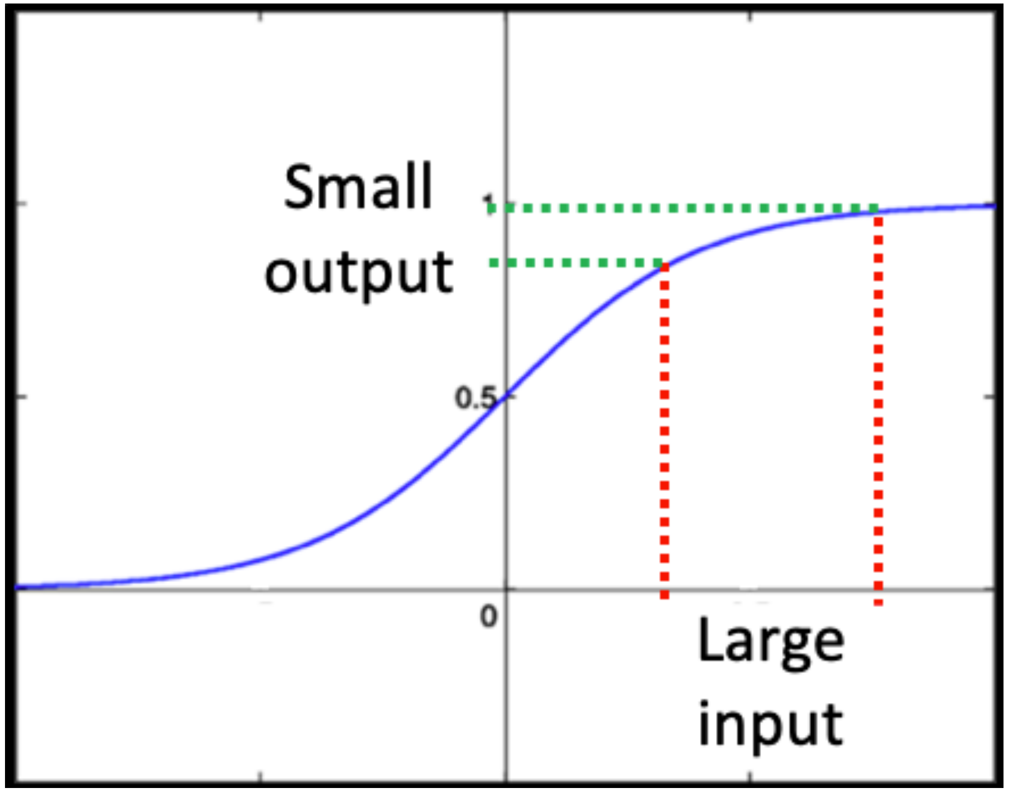
Why Not?

- It seems as well that the deep learning revolution results mainly from brute force, it is not based on new mathematical models and appears to be biologically unlikely.
- Deep neural networks require a very large labeled sample training set that can become a bottleneck, since in many special application it is difficult to generate big labeled sample training sets.
- Often these huge sets have to be manually labeled by some experts which results in high costs.

Vanishing Gradient Problem



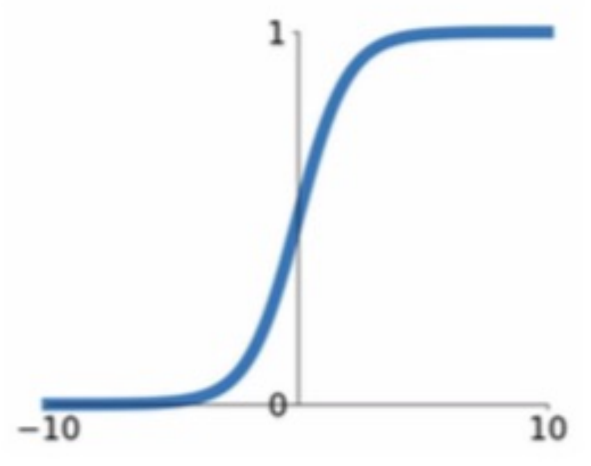
Vanishing Gradient Problem, sigmoid



Sigmoid

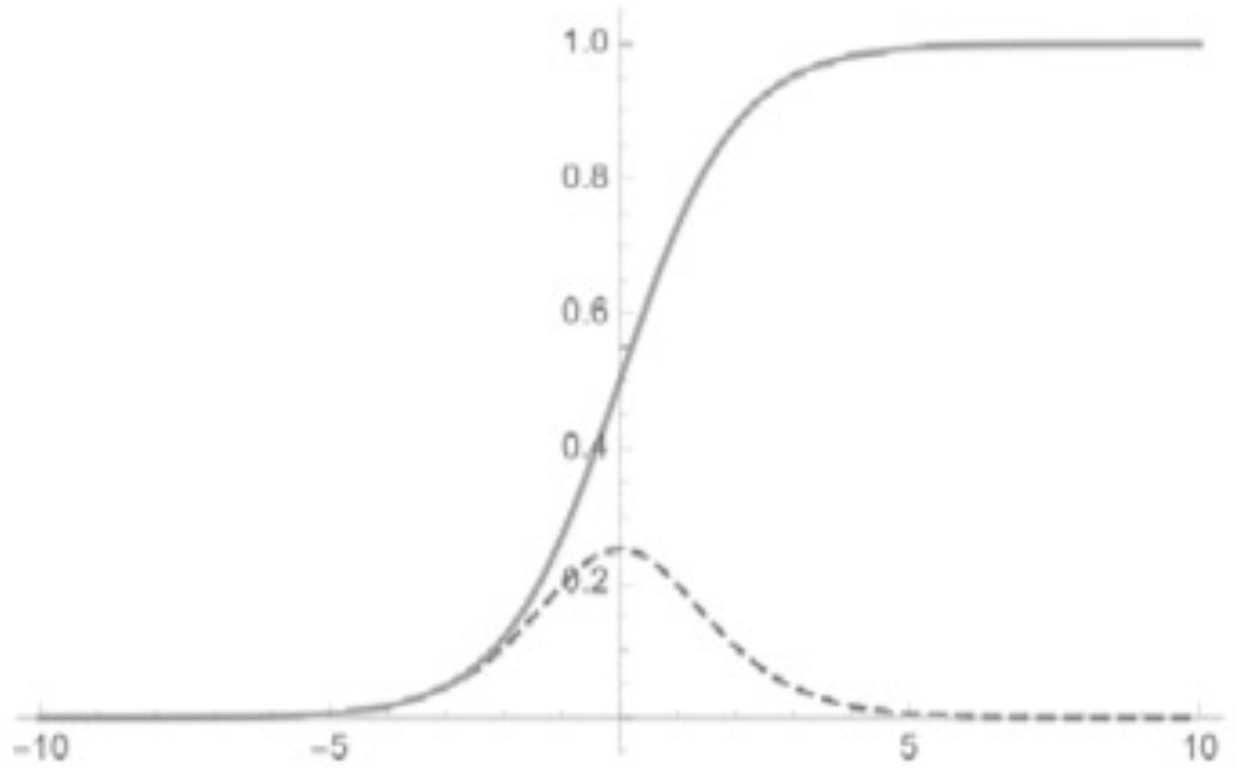
$$f(x) = \sigma(x) = \frac{1}{1 + e^{(-\alpha \cdot x)}}$$

$$f^{'}(x) = \sigma^{'}(x) = \alpha \cdot \sigma(x) \cdot (1 - \sigma(x))$$



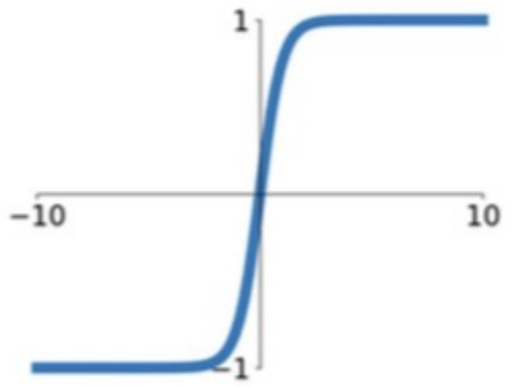
Sigmoid

- Squashes numbers to range [0,1]
- Historically popular
- Have nice interpretation as a saturating “firing rate” of a neuron
- 3 problems:
  - Saturated neurons “kill” the gradients
  - Sigmoid outputs are not zero-centered
  - exp() is a bit compute expensive



- The sigmoid function and the derivative indicated by doted line.

$$f(x) = \tanh(\alpha \cdot x)$$
$$f^{'}(x) = \alpha \cdot (1 - f(x)^2)$$



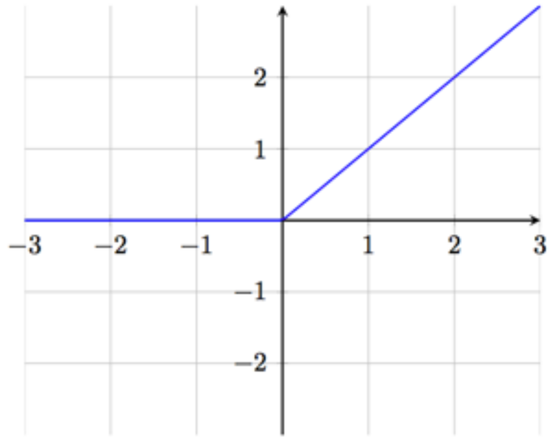
tanh(x)

- Squashes numbers to range [-1,1]
- - zero centered (nice)
- - kills gradients when saturated ☹

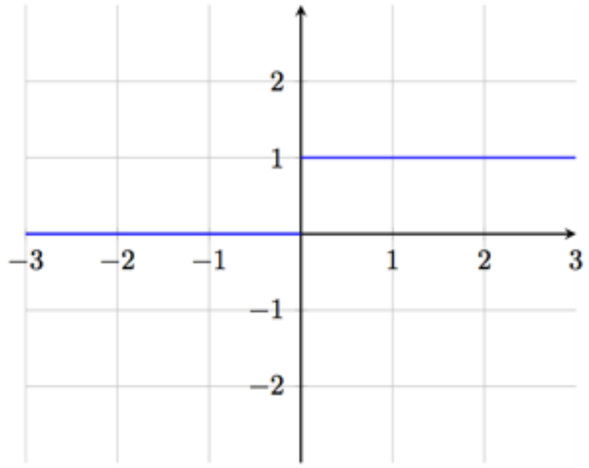
Rectified Linear Unit (ReLU)

- *f(x) = max(0,x)*
  - Function defined as the positive part of its argument
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- More biologically plausible

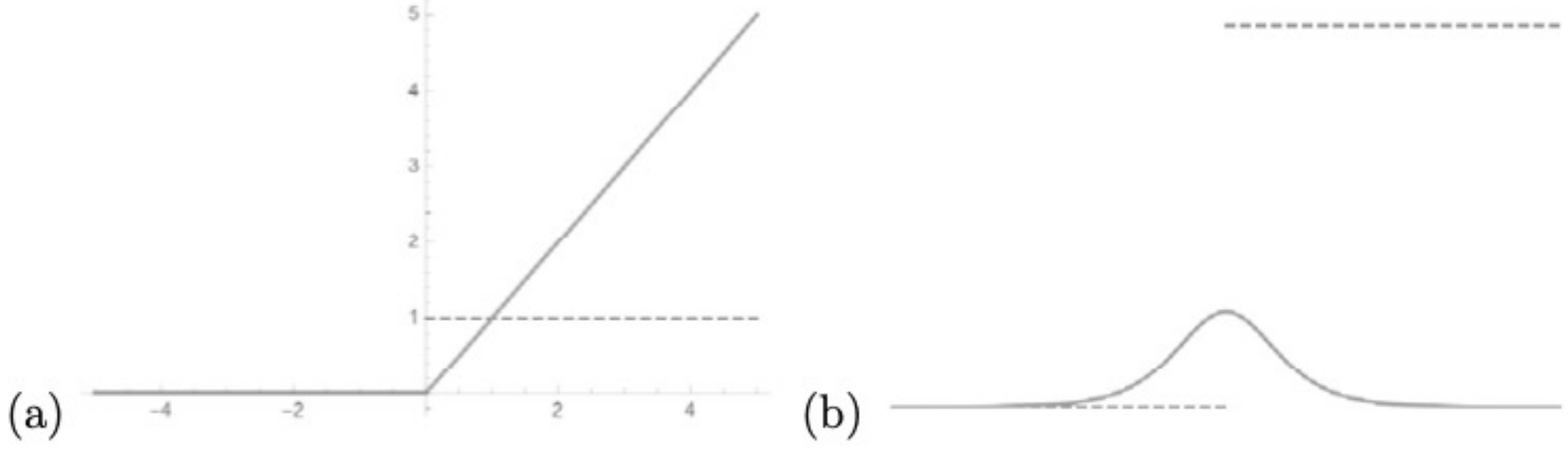
- But: Not zero-centered output ☹
- Non-differentiable at zero; however it is differentia a value of 0 or 1



ReLu function

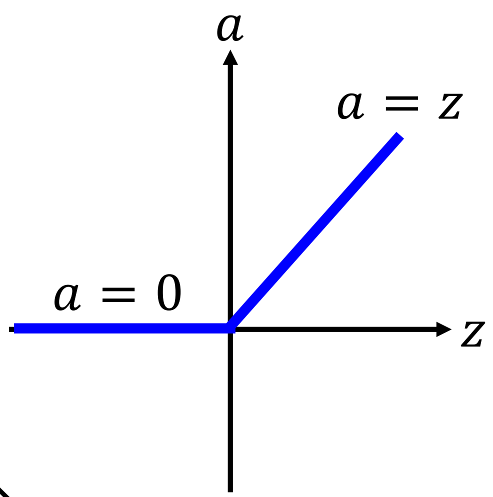


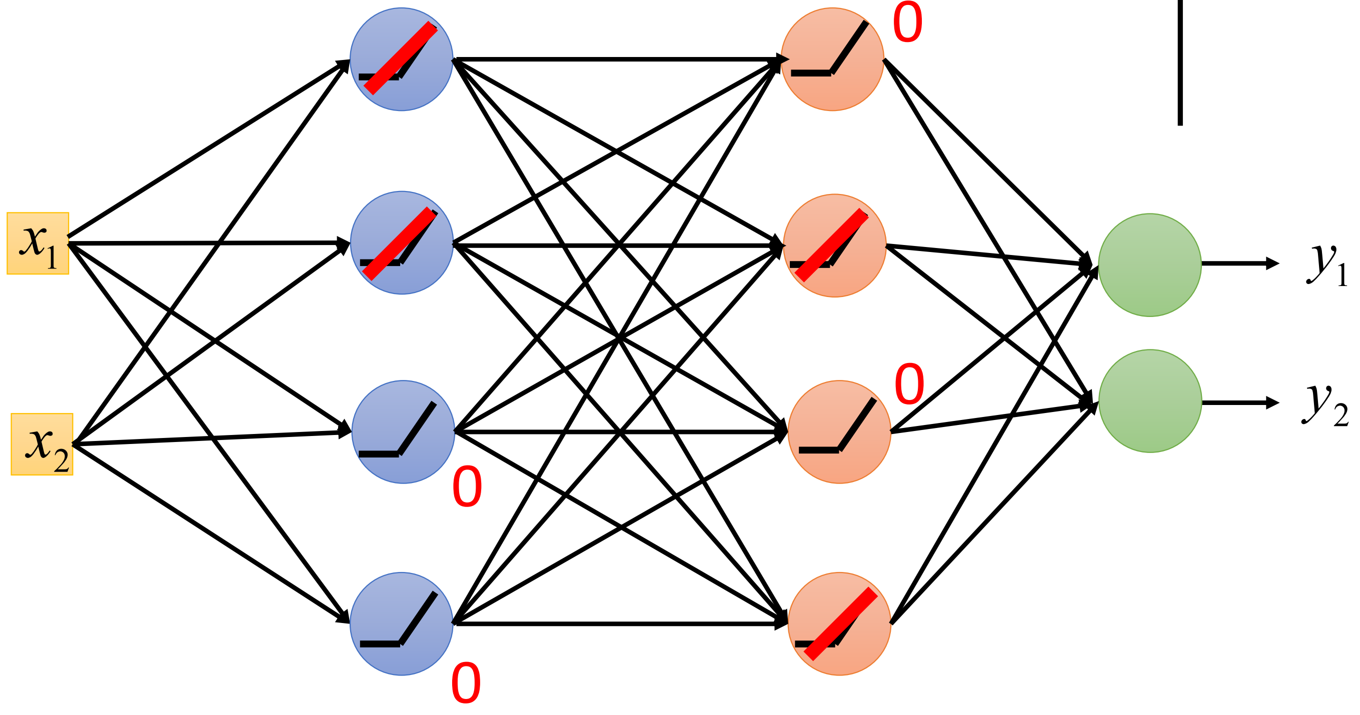
Derivative of ReLu function



- (a) Rectifier activation function (ReLU), the derivative is in- dicated by the doted line. (b) Comparing the the derivative of the sigmoid activation function and the rectifier activation function indicated by a doted line.

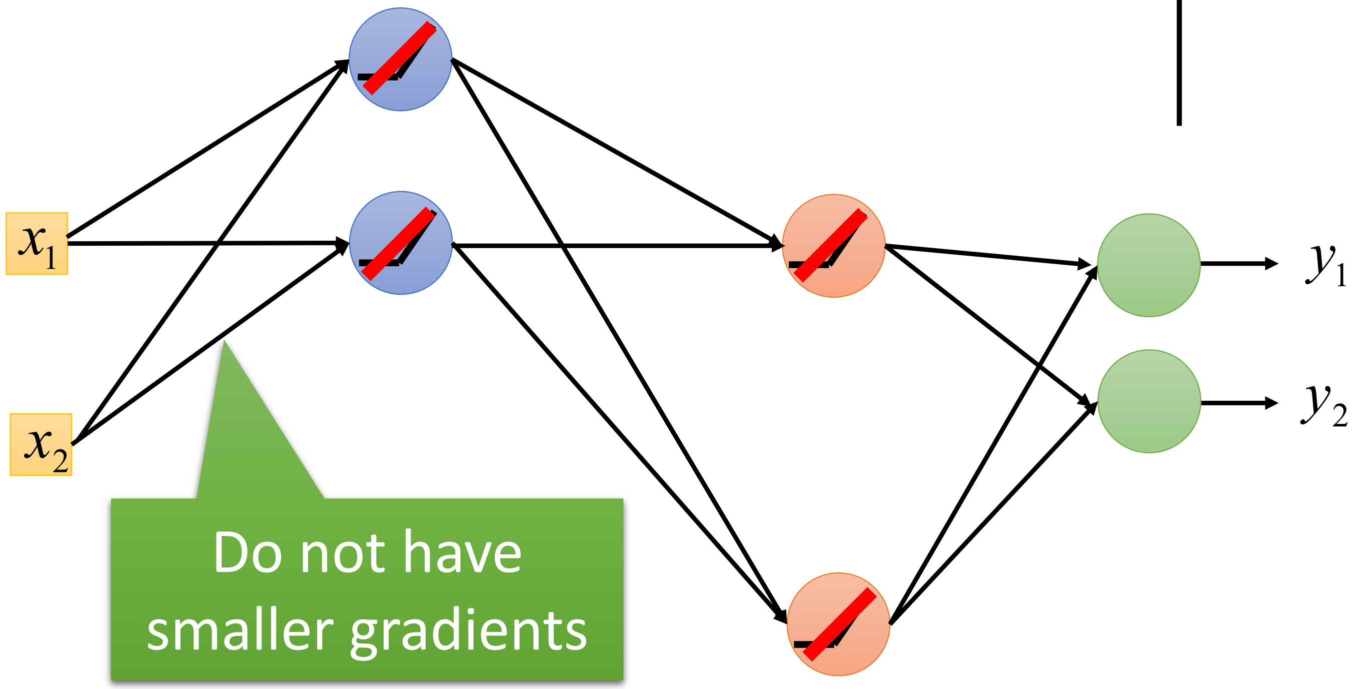
ReLU



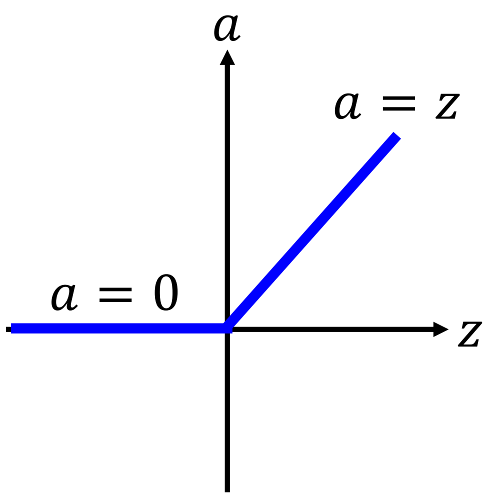


ReLU

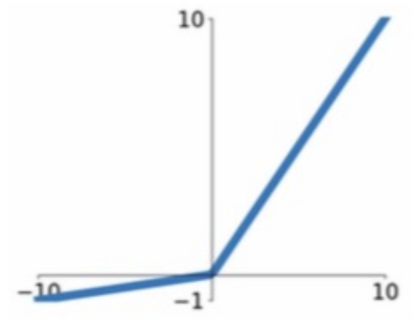
A Thinner linear network



Do not have smaller gradients



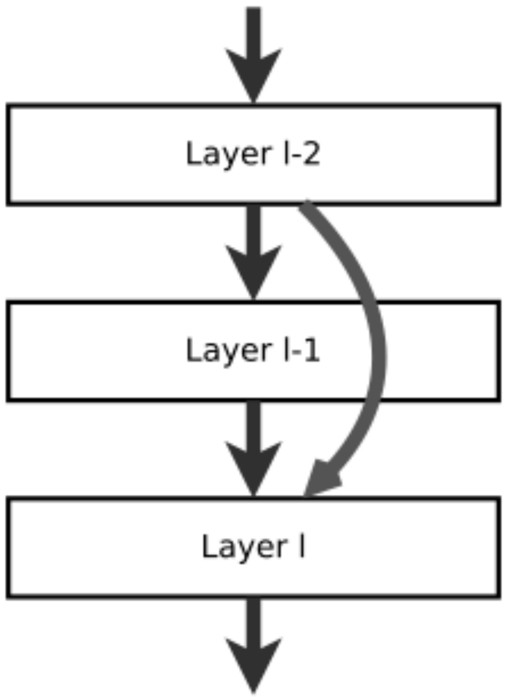
Leaky ReLU

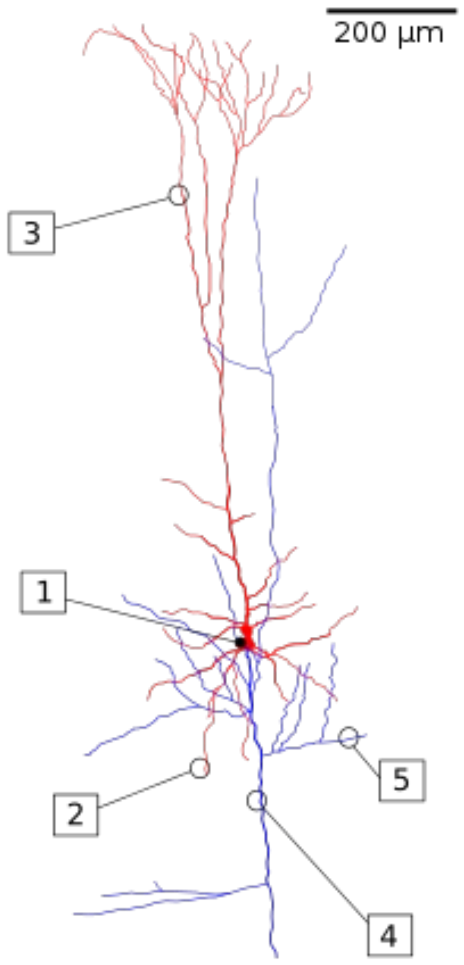


$$f(x) = \max(0.01x, x)$$

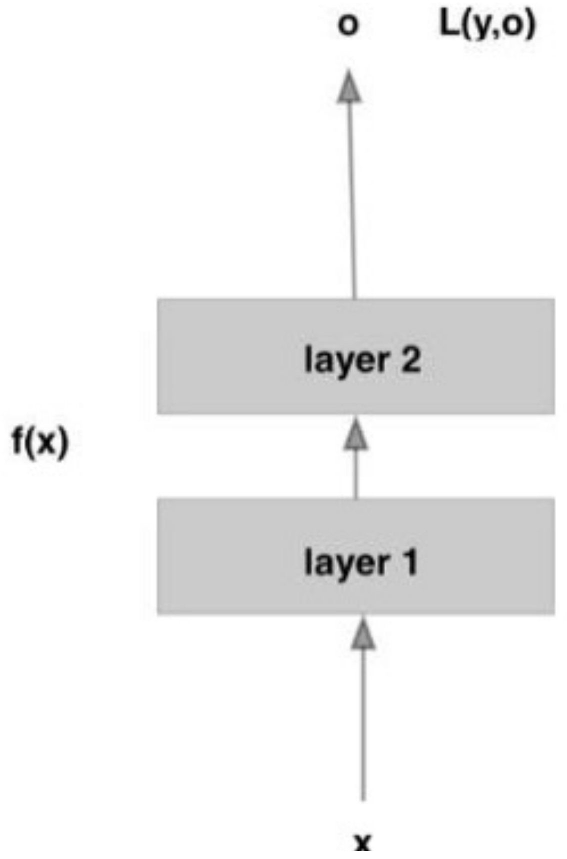
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- Will not die

Residual neural network

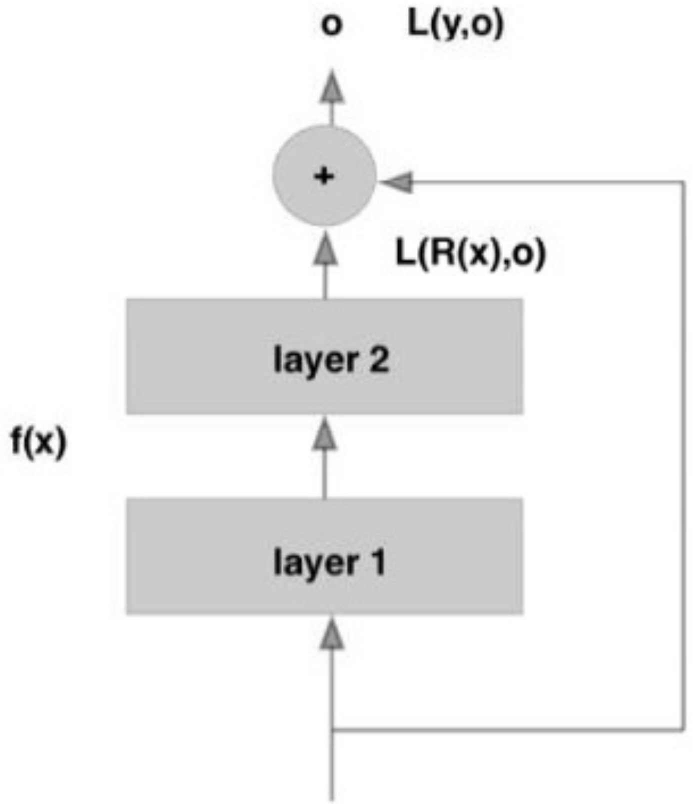




Pyramid Cells



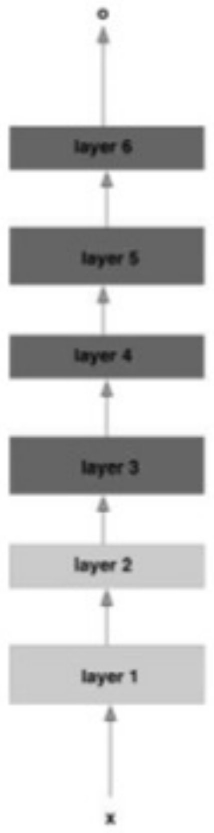
$$o = f(\mathbf{x}, \theta_i)$$

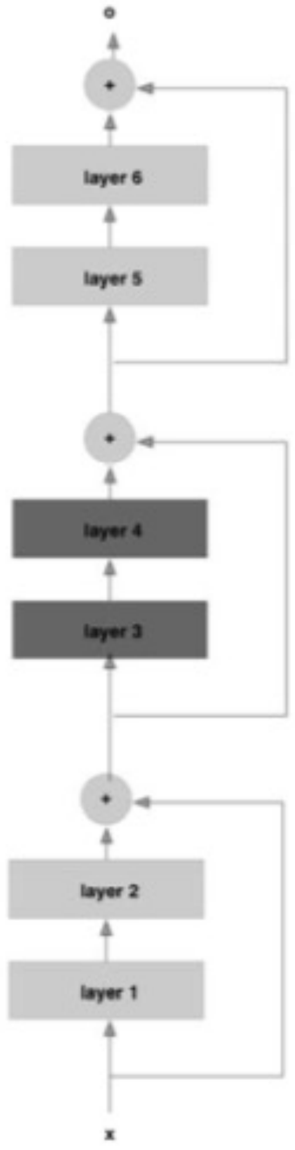


$$o = f(\mathbf{x}, \theta_i) + \mathbf{x}.$$

building block

$$o_\theta = f(\mathbf{x}, \theta)$$


o
layer 6
layer 5
layer 4
layer 3
layer 2
layer 1
x


o
+
layer 6
layer 5
+
layer 4
layer 3
+
layer 2
layer 1
x

- Residual neural networks do this by utilizing *skip connections*, or *short-cuts* to jump over some layers.
- One motivation for skipping over layers is to avoid the problem of vanishing gradients, by reusing activations from a previous layer until the adjacent layer learns its weights.
- During training, the weights adapt to mute the upstream layer, and amplify the previously-skipped layer.
  - In the simplest case, only the weights for the adjacent layer's connection are adapted, with no explicit weights for the upstream layer.

Weight Initialization

- **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)
- Works for small networks, but problems with deeper networks.

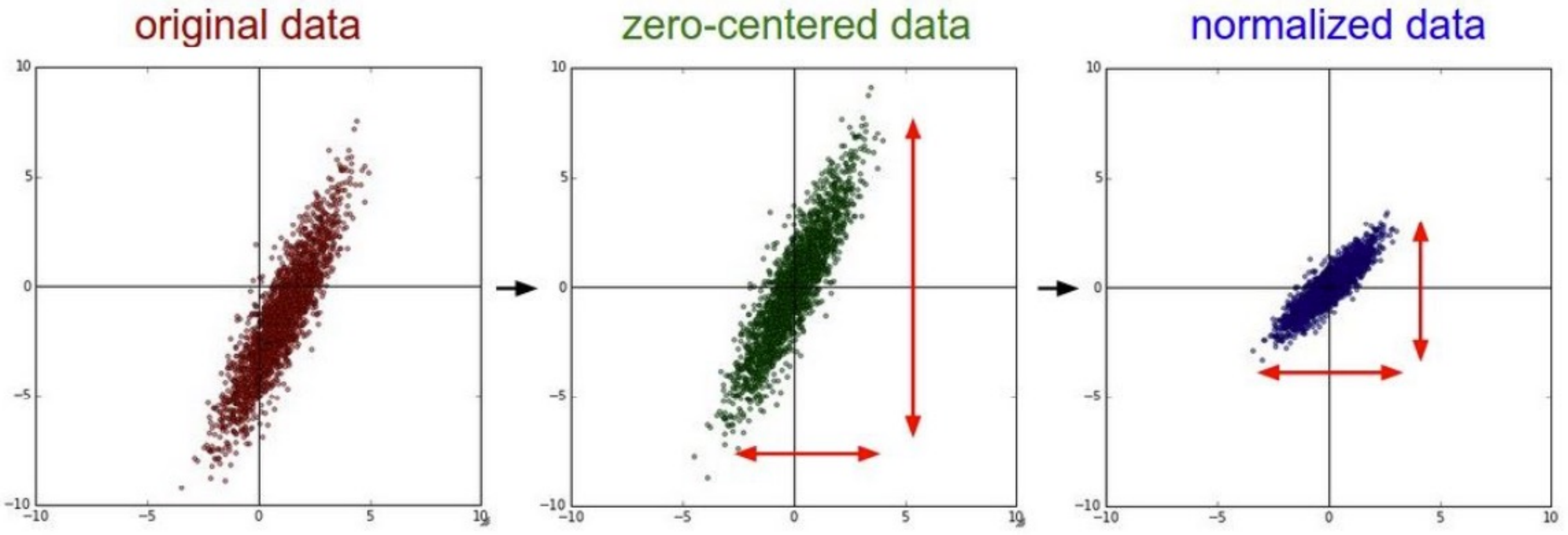
- Deeper networks:
  - All activations can become zero!
  - Almost all neurons completely saturated, either -1 and 1.
  - Gradients will be all zero.

Xavier initialization

- Weights from a Gaussian distribution with
  - zero mean
  - variance of 1/N
  - N specifies the number of input neurons.
  - when using the ReLU nonlinearity it breaks ☹

- Better [He et al., 2015]:
  - Weights from a Gaussian distribution with
  - zero mean
  - variance of 2/N
  - Does not break with ReLU

Preprocess the data



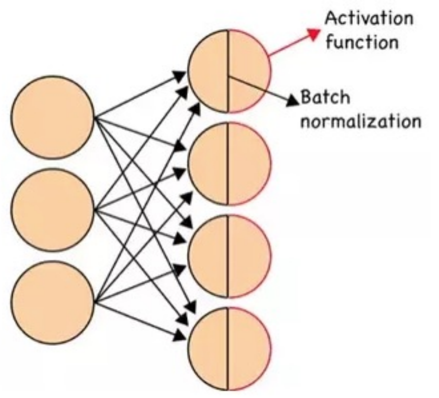
Batch Normalization

- We normalize all training data so that it resembles a normal distribution (that means, zero mean and a unitary variance)

- In the intermediate layers the distribution of the activations is constantly changing during training
  - This slows down the training process because each layer must learn to adapt themselves to a new distribution in every training step.
  - Batch normalization is a method we can use to normalize the inputs of each layer, in order to fight the internal covariate shift problem

Batch Normalization

- During training time, a **batch normalization layer** does the following:
    - Calculate the mean and variance of the layers input
    - Normalize the layer inputs using the previously calculated batch statistics
    - Scale and shift in order to obtain the output of the layer
    - *γ* and *β* are **learned during training** along with the original parameters of the network.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

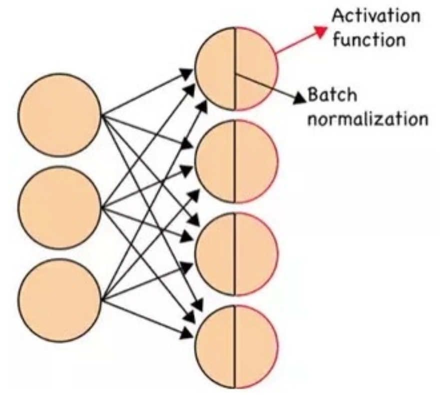
$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

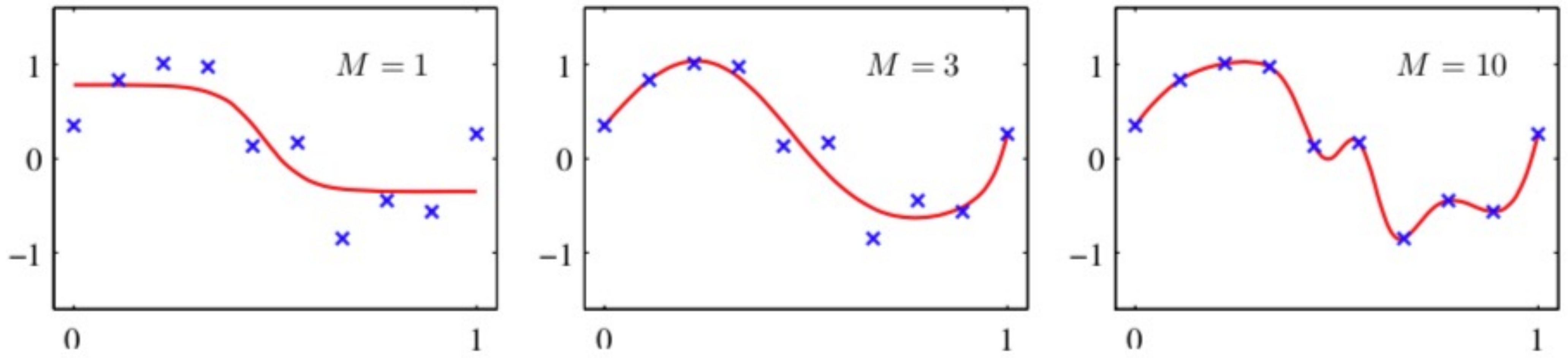
Test



- During test (or inference) time, the mean and the variance are fixed. They are estimated using the previously calculated means and variances of each training batch.

Overfitting

- The training data contains information about the regularities in the mapping from input to output.
- But it also contains noise
  - The target values may be unreliable.
  - There is sampling error
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error
  - If the model is very flexible it can model the sampling error really well. This is a disaster



- Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having M = 1, 3 and 10 hidden units, respectively, by minimizing a sum-of-squares error function

Preventing overfitting

- Get more data!
  - Always the best
- Use a model that has the right capacity:
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)
- Early stopping
  - Start with small weights and stop the learning before it overfits
- Weight-decay: Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
- Use Dropout that drops randomly some weights
- Noise: Add noise to the weights or the activities

Using a Validation Set

- Divide the total dataset into three subsets:
- Training data is used for learning the parameters of the model.
- Validation data is not used of learning but is used for deciding what type of model and what amount of regularization works best.
- Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data

l_2 Regularization

- The standard L_2 weight penalty involves adding an extra term to the cost function that penalizes the squared weights
- This keeps the weights small unless they have big error derivatives.
- It prevents the network from using weights that it does not need.
  - This can often improve generalization a lot because it helps to stop the network from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.

l_2 Regularization

$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \cdot \sum_{k=1}^{N} (y_k - o_k)^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2, \;\; or \;\; \tilde{E}(\mathbf{w}) = -\sum_{k=1}^{N} y_k \log o_k + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

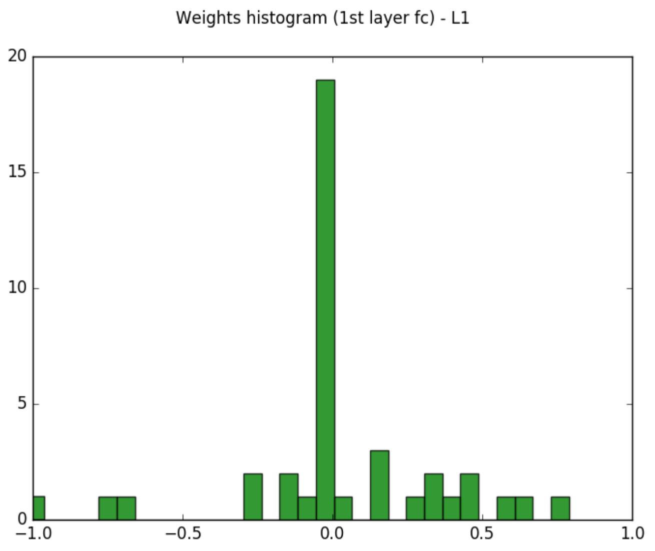
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

$$\frac{\partial \tilde{E}}{\partial w_j} = \frac{\partial E}{\partial w_j} + \lambda \cdot w_j.$$

$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} + \lambda \cdot w_j \right) = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \eta\lambda \cdot w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot w_j$$

l_1 Regularization

- Sometimes it works better to penalize the absolute values of the weights.
  - This makes some weights equal to zero which helps interpretation.



l_1 Regularization

$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \cdot \sum_{k=1}^{N} (y_k - o_k)^2 + \lambda \cdot \|\mathbf{w}\|_1, \ or \ \ \tilde{E}(\mathbf{w}) = -\sum_{k=1}^{N} y_k \log o_k + \lambda \cdot \|\mathbf{w}\|_1$$

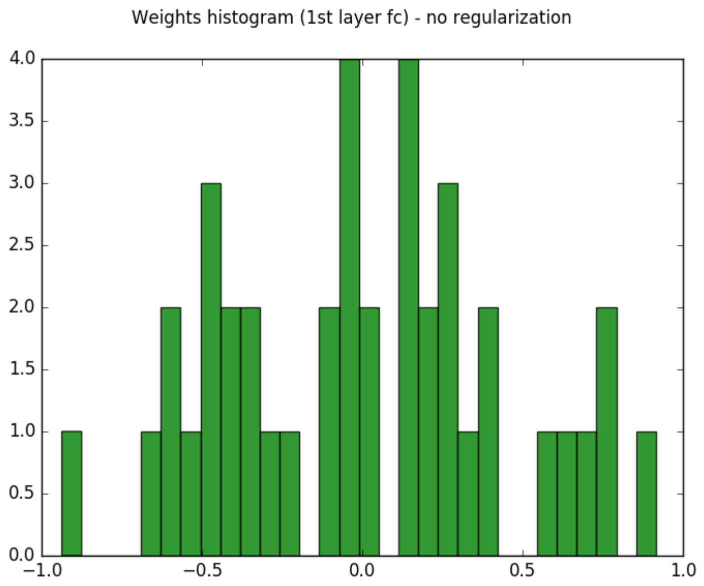
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$$

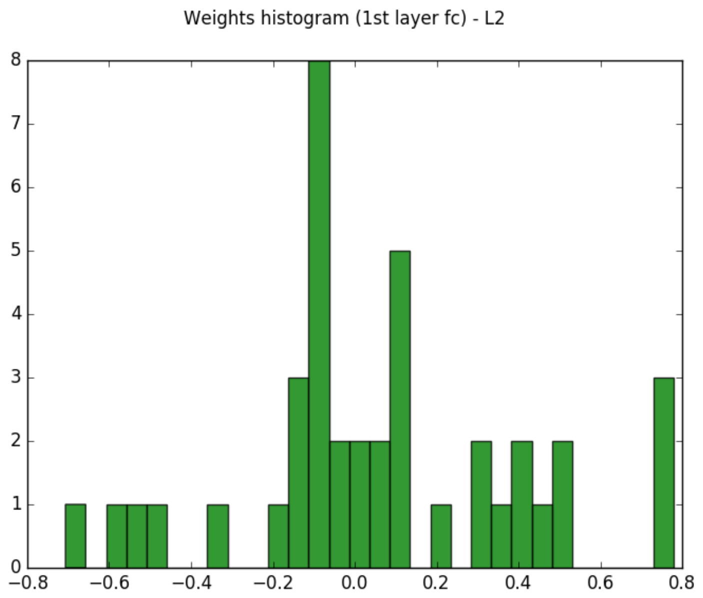
$$\frac{\partial \tilde{E}}{\partial w_j} = \frac{\partial E}{\partial w_j} + \lambda \cdot sign(w_j).$$

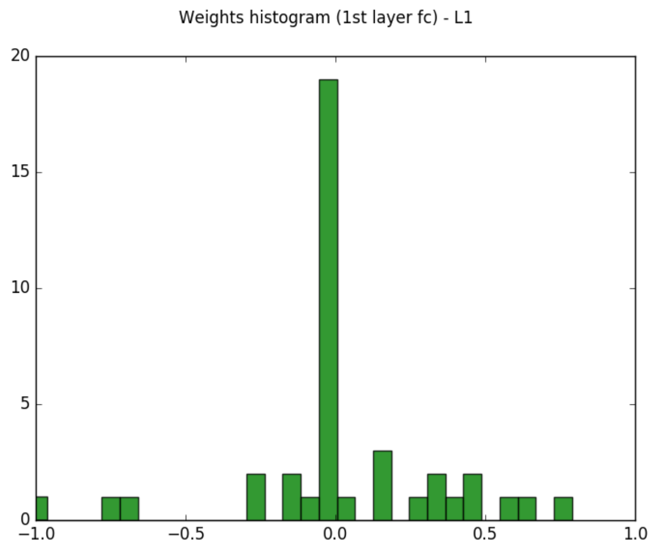
$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} + \lambda \cdot sign(w_j) \right) = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \eta\lambda \cdot sign(w_j)$$

$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot sign(w_j)$$

l_2 versus l_1 Regularization





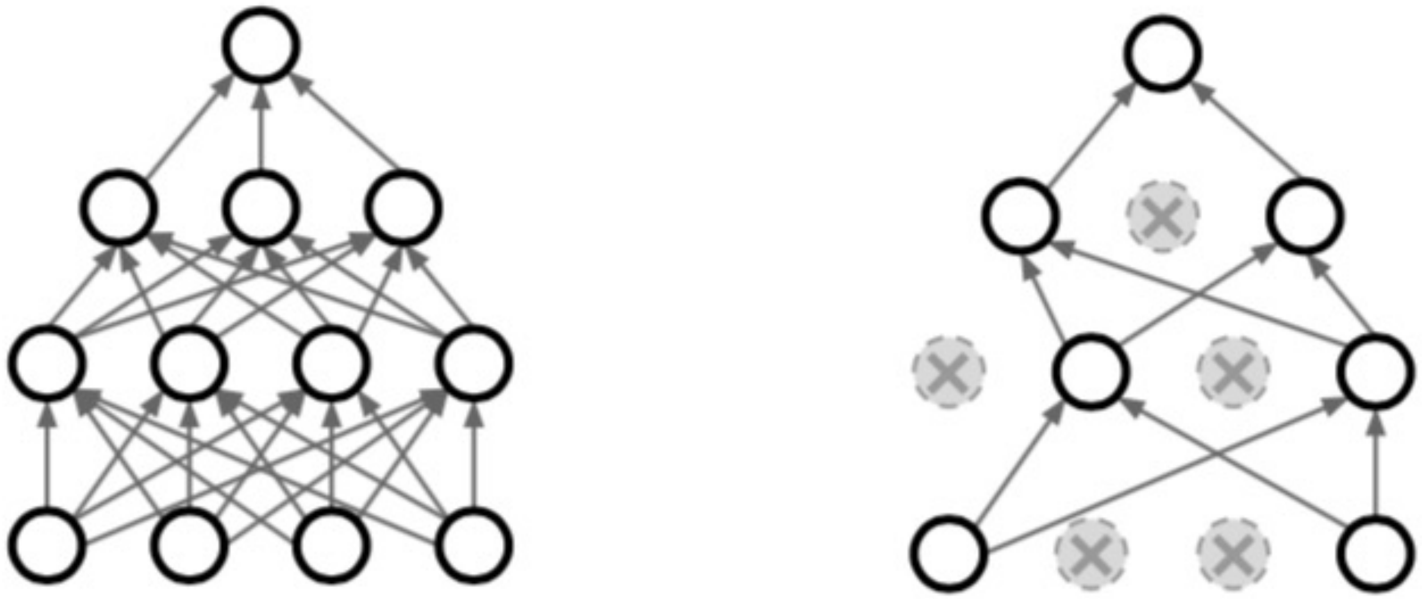


Regularization: Dropout

- Dropout is a stochastic regularization method.
- In each forward pass, randomly a set some neurons is set to zero (sleep) for one pass.
- The probability of dropping the set out is described by a hyper-parameter; usually 0.5 is commonly used.
  - For input nodes, this should be low, because information is directly lost when input nodes are ignored.

Regularization: Dropout

- In each forward pass, randomly set some neurons to zero (for one pass only)
  - Probability of dropping is a hyperparameter; 0.5 is common



Regularization: Dropout

- Only the reduced network is trained on the data in that stage
- The removed nodes are then reinserted into the network with their original weights.
  - For input nodes, this should be low, i because information is directly lost when input nodes are ignored.
- By avoiding training all nodes on all training data, dropout decreases overfitting.
  - It also significantly improves training speed.
  - It reduces node interactions, leading them to learn more robust features
- It is stochastic process, since each time a different set of neurons is drop out and not allowed to learn

Faster Optimizers: Momentum

- One can in some cases speed up trining by using a faster gradient descent as for example using the momentum α with τ indicating the time step of the algorithm

$$\Delta w_{ti}(\tau+1) = -\eta \frac{\partial E}{\partial w_{ti}} + \alpha \cdot \Delta w_{ti}(\tau)$$

$$w_{ti}^{new} = w_{ti}^{old} + \Delta w_{ti}(\tau+1).$$

- It prohibits fast changes of the direction of the gradient. The momentum parameter α is chosen between 0 and 1, usually 0.9 is a good value.

Nestrov Momentum

- The idea of Nesterov momentum optimization is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum $w_{ti} + \Delta w_{ti}(\tau)$

$$\Delta w_{ti}(\tau + 1) = -\eta \frac{\partial E(w_{ti} + \Delta w_{ti}(\tau))}{\partial w_{ti}} + \alpha \cdot \Delta w_{ti}(\tau),$$

$$w_{ti}^{new} = w_{ti}^{old} + \Delta w_{ti}(\tau + 1).$$

AdaGrad

- The AdaGrad algorithm scales down the gradient vector along the steepest dimensions and uses a different learning rate for every parameter w_i at every time step
  - It scales them inversely proportional to the square root of the sum of all squared values of the gradient representing the scale variable $s_{ti}(\tau)$
  - At the begin of learning with $\tau = 0$ the scale variable is initialized to *zero* $s_{ti}(0) = 0$.

$$s_{ti}(\tau + 1) = s_{ti}(\tau) + \frac{\partial E}{\partial w_{ti}} \cdot \frac{\partial E}{\partial w_{ti}}$$

and we preform the scaled gradient descent

$$w_{ti}^{new} = w_{ti}^{old} - \eta \cdot \frac{\frac{\partial E}{\partial w_{ti}}}{\sqrt{s_{ti}(\tau + 1) + \epsilon}}$$

ϵ is a smoothing term, a very tiny number to avoid division by zero

RMSProp

- he AdaGrad algorithm can slow down a bit too fast and may end up never converging to a minimum.
- RMSProp changes the gradient into an exponen tial weighted moving average.
  - It discard history from extreme past so that it can converge rapidly, it accumulating only the gradinents from the most recent iterations

$$s_{ti}(\tau+1) = \alpha \cdot s_{ti}(\tau) + (1-\alpha) \cdot \frac{\partial E}{\partial w_{ti}} \cdot \frac{\partial E}{\partial w_{ti}}$$

$$w_{ti}^{new} = w_{ti}^{old} - \eta \cdot \frac{\frac{\partial E}{\partial w_{ti}}}{\sqrt{s_{ti}(\tau+1) + \epsilon}}$$

α being the decay rate with a typical value of 0.9.

Adam

- Adam which stands for adaptive moment estimation, combines the ideas of momentum

$$\Delta w_{ti}(\tau+1) = -(1-\alpha_1)\frac{\partial E}{\partial w_{ti}} + \alpha_1 \cdot \Delta w_{ti}(\tau)$$

- and RMSProp

$$s_{ti}(\tau+1) = \alpha_2 \cdot s_{ti}(\tau) + (1-\alpha_2) \cdot \frac{\partial E}{\partial w_{ti}} \cdot \frac{\partial E}{\partial w_{ti}}$$

Adam

$$\Delta w_{ti}(\tau+1) = \frac{\Delta w_{ti}(\tau+1)}{1-\alpha_1^{\tau+1}}$$

and

$$s_{ti}(\tau+1) = \frac{s_{ti}(\tau+1))}{1-\alpha_2^{\tau+1}}.$$

Then we apply the update

$$w_{ti}^{new} = w_{ti}^{old} + \eta \cdot \frac{\Delta w_{ti}(\tau+1)}{\sqrt{s_{ti}(\tau+1)+\epsilon}}$$

Notation…

- In deep learning community one often uses a vector notation and indicates the free parameter as θ since beside the weights there are as well other parameters
- The notation for Adam would be for momentum

$$\mathbf{m} = \alpha_1 \cdot \mathbf{m} - (1 - \alpha_1)\nabla_\theta L(\theta)$$

- and RMSProp

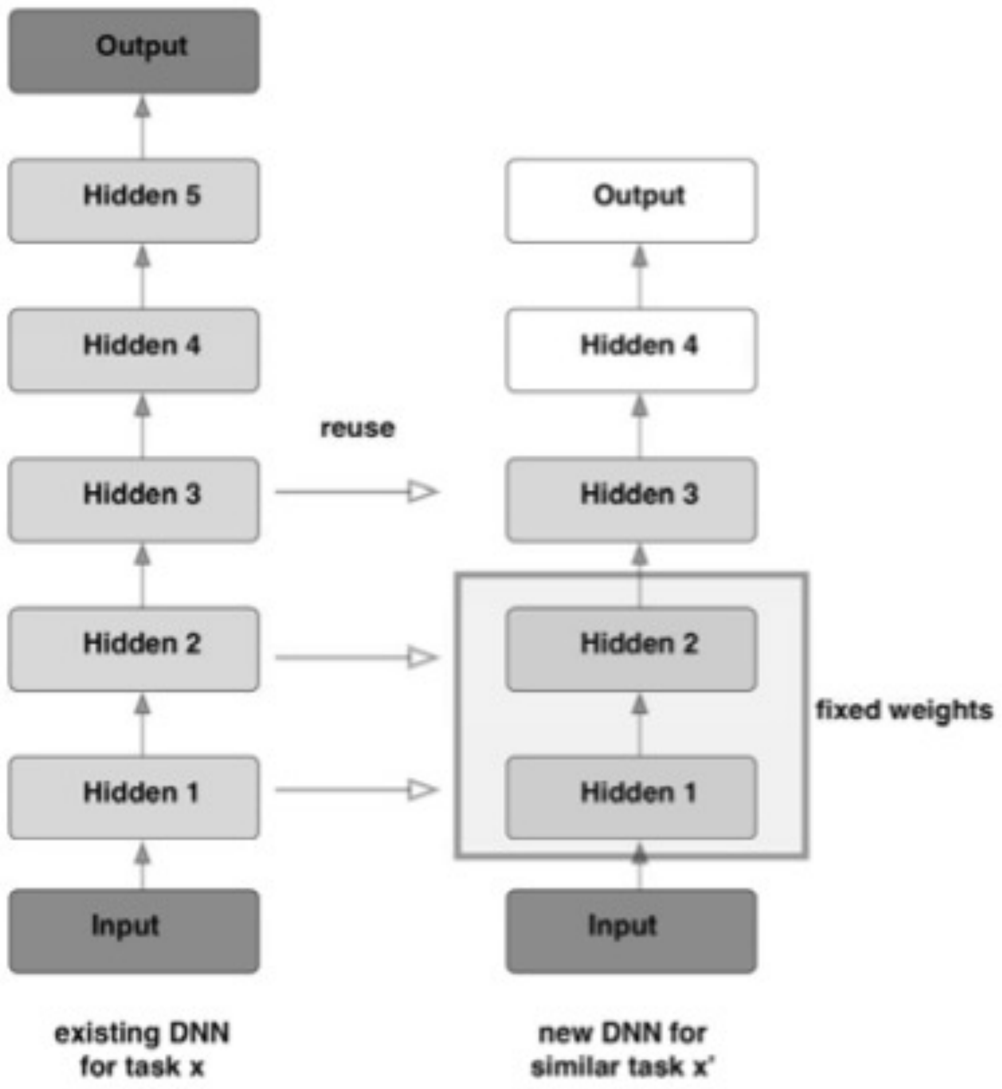
$$\mathbf{s} = \alpha_2 \cdot \mathbf{s} + (1 - \alpha_2)\nabla_\theta L(\theta) \odot \nabla_\theta L(\theta)$$

$$\mathbf{m} = \frac{\mathbf{m}}{1 - \alpha_1^\tau}$$

$\odot$ represents the element-wise multiplication

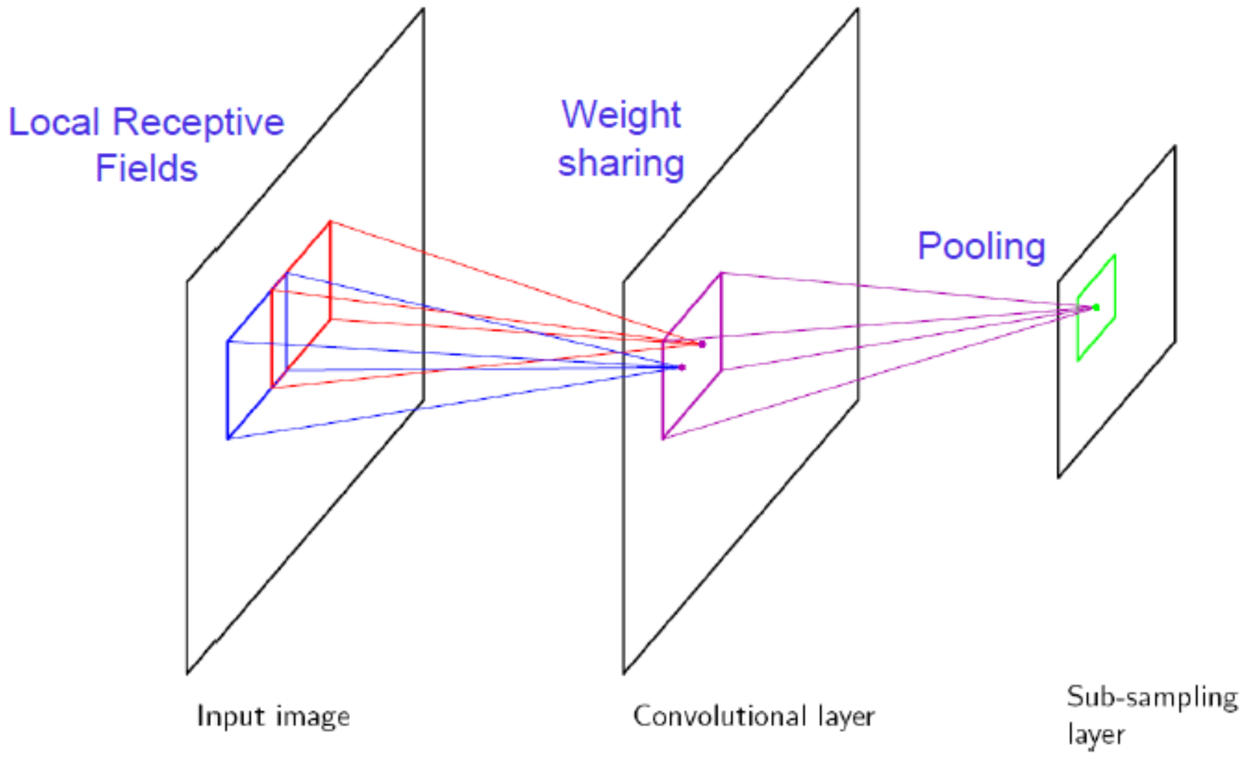
Transfer Learning

- You need a lot of a data if you want to train
- Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P_1) is exploited to improve generalization in another setting (say distribution P_2).
- We assume that many of the factors that explain the variations in P_1 are relevant to the variations that need to be captured for learning P_2.

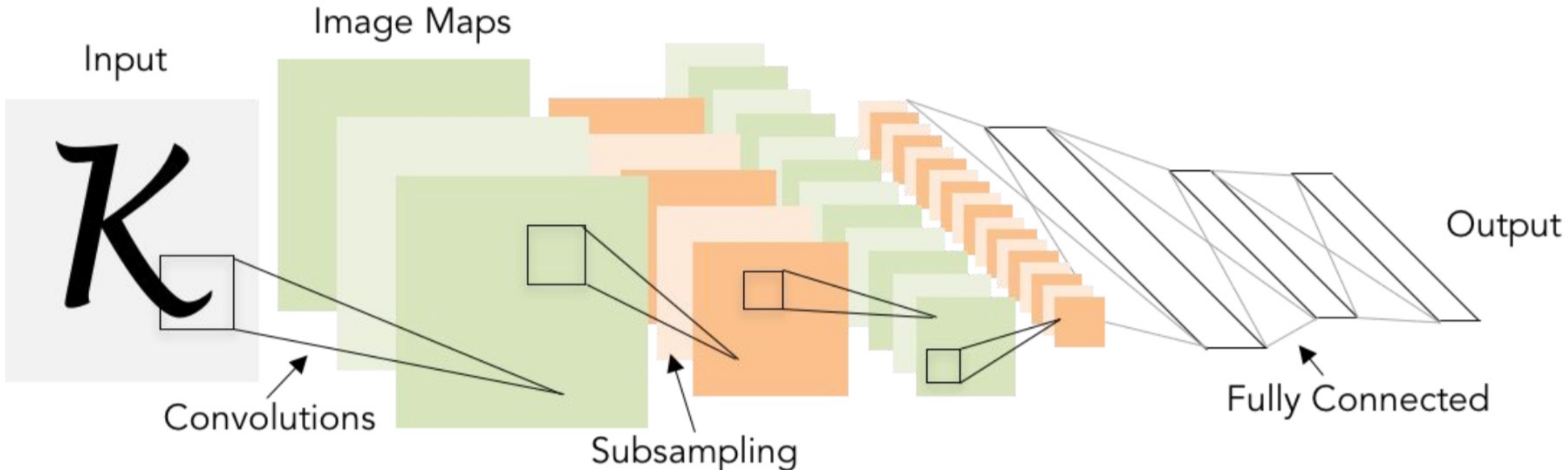

Output
Hidden 5
Hidden 4
Hidden 3
Hidden 2
Hidden 1
Input
reuse
Output
Hidden 4
Hidden 3
Hidden 2
Hidden 1
Input
fixed weights
existing DNN for task x
new DNN for similar task x'

Convolutional Neural Networks

(LeCun et al., 1989)



- Convolutional Neural Networks

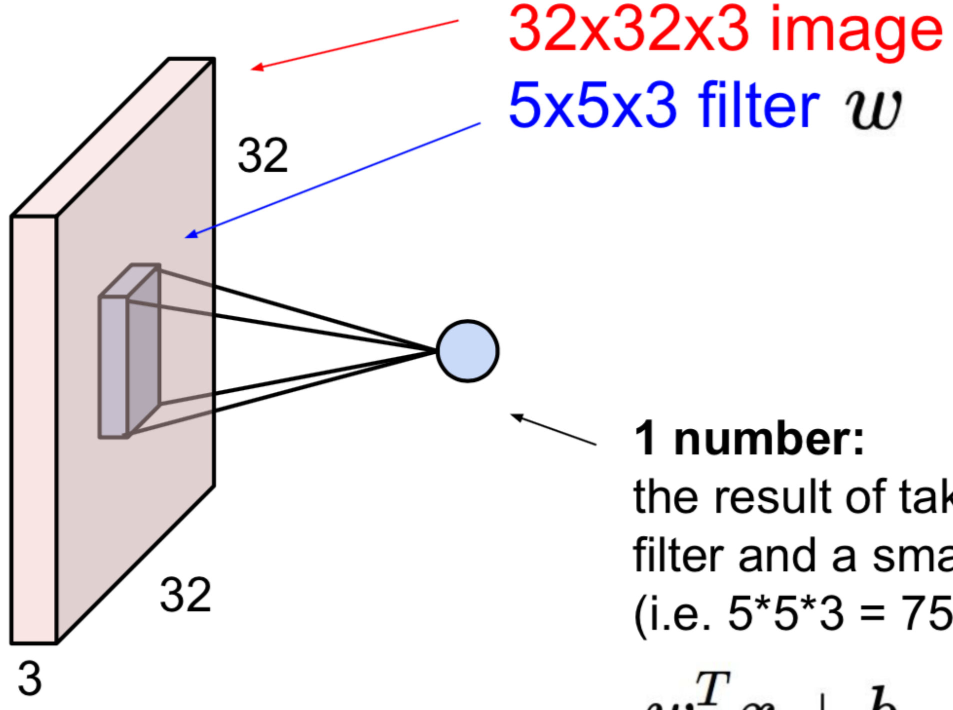


MNIST Data Set

→0, →3, →9, →0, →2, →1, →1, →3, →9
→4, →1, →2, →2, →1, →4, →8, →0, →4
→4, →7, →7, →2, →9, →6, →5, →5, →4
→8, →2, →5, →9, →5, →4, →1, →3, →7
→8, →0, →7, →4, →4, →7, →4, →7, →9
→8, →9, →9, →2, →2, →0, →1, →6, →5
→4, →4, →3, →9, →9, →1, →1, →5, →9
→2, →7, →0, →3, →4, →7, →5, →8, →7
→9, →0, →2, →8, →1, →2, →2, →7, →3

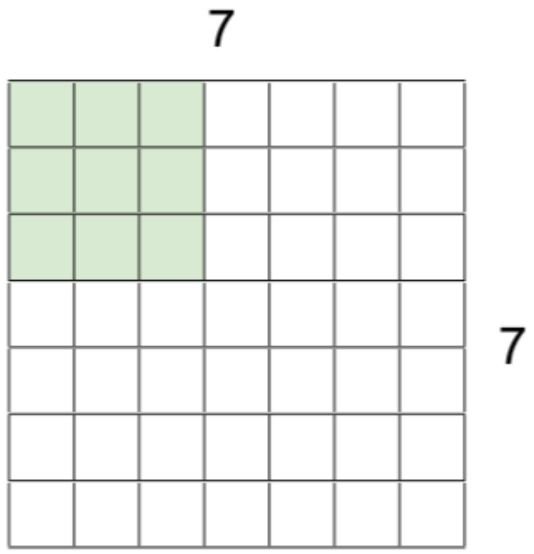
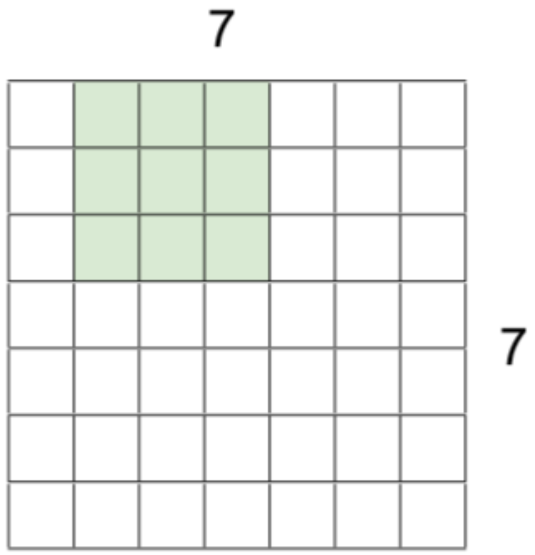
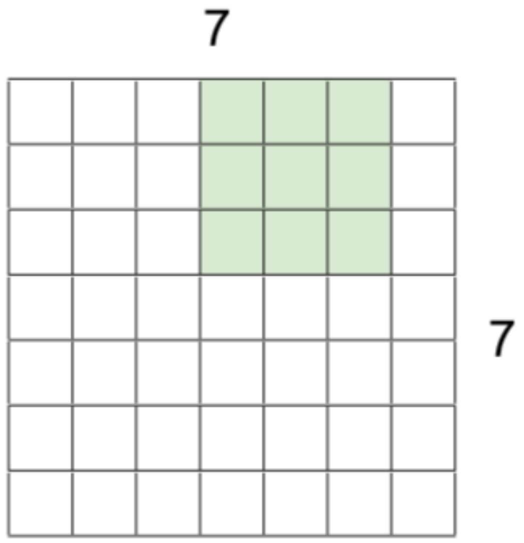
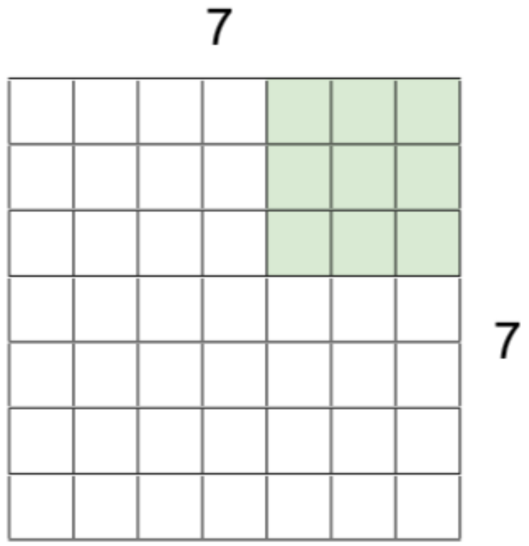
- The MNIST database contains 60, 000 training im- ages and 10, 000 testing images. The images of the digits contain grey levels represented by a 28 × 28 matrix resulting in 784 dimensional input vector

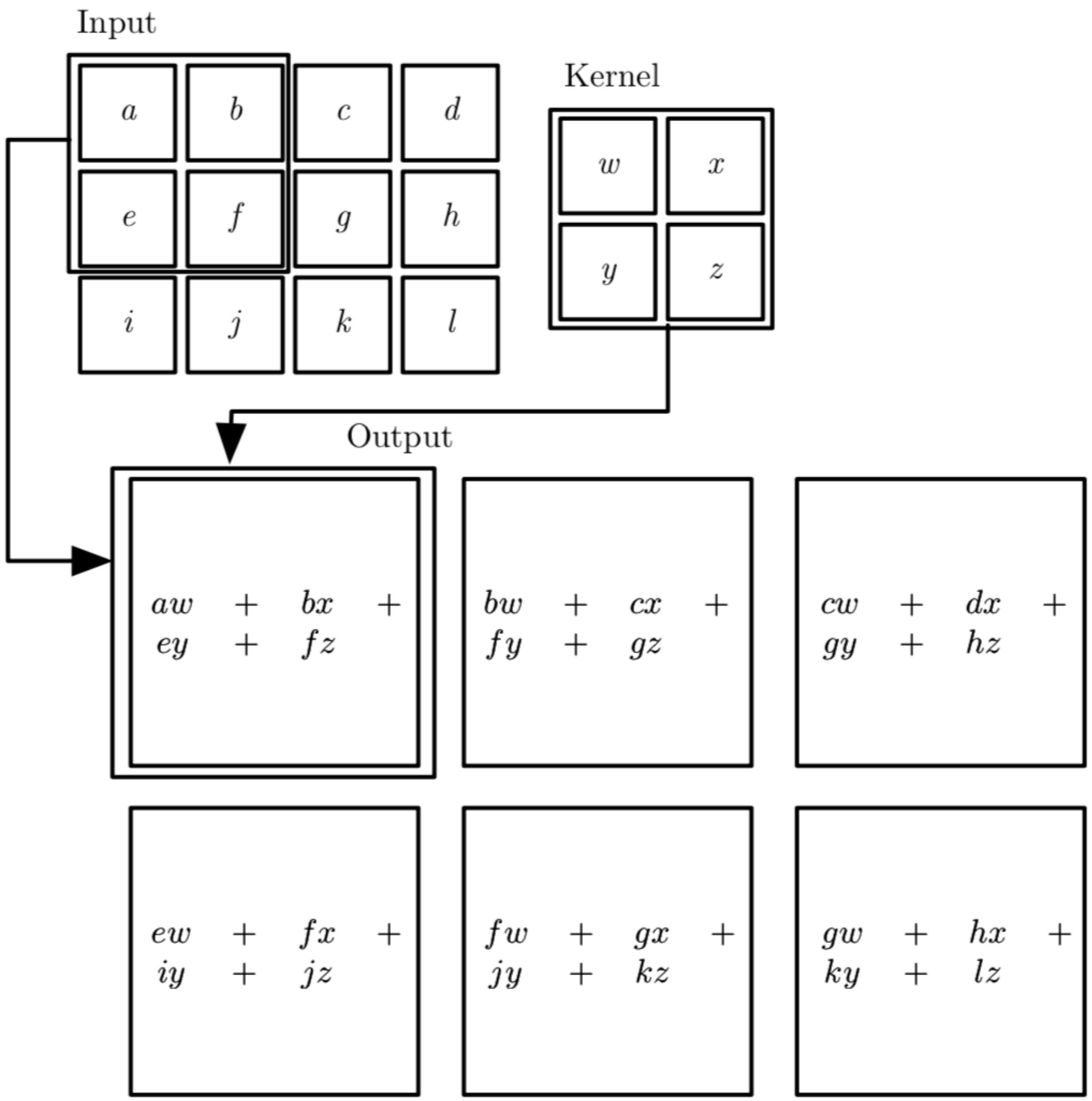
Convolution Layer



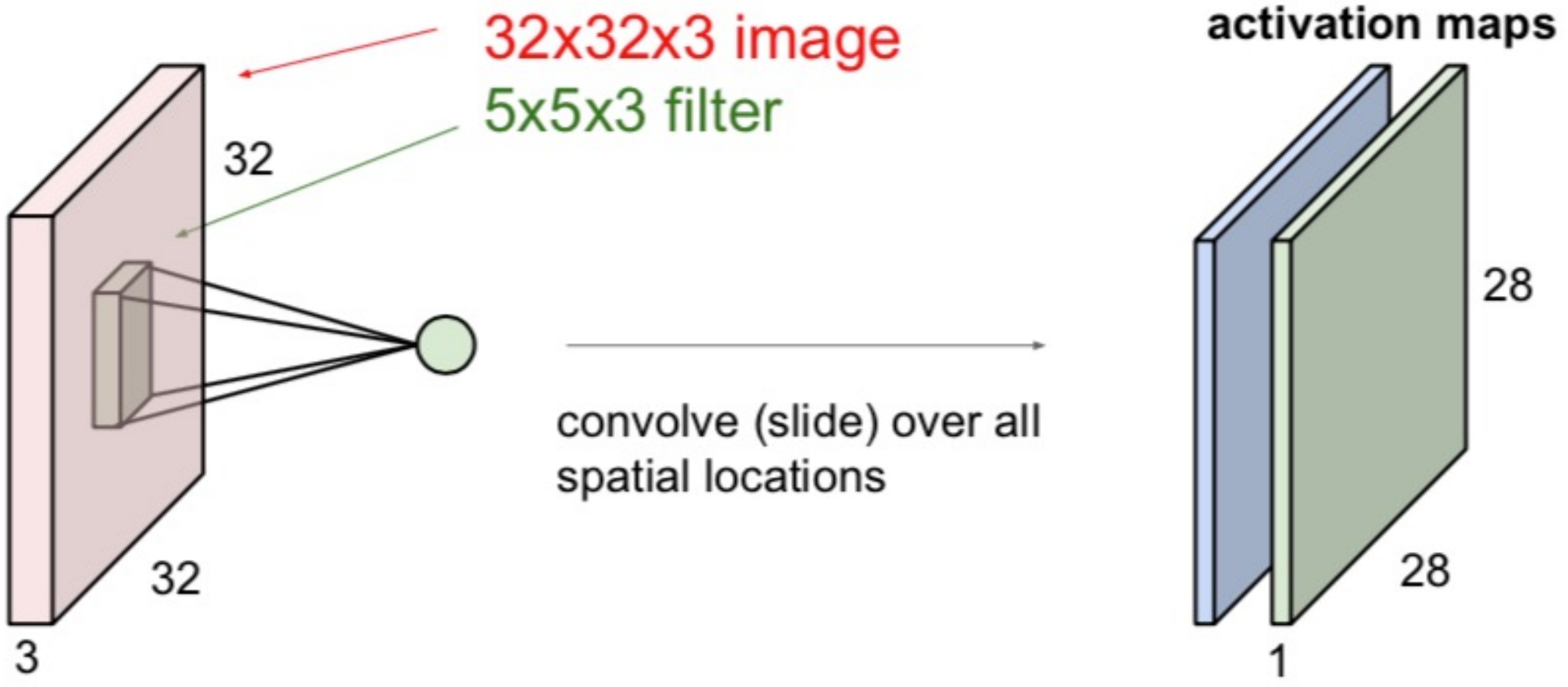
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

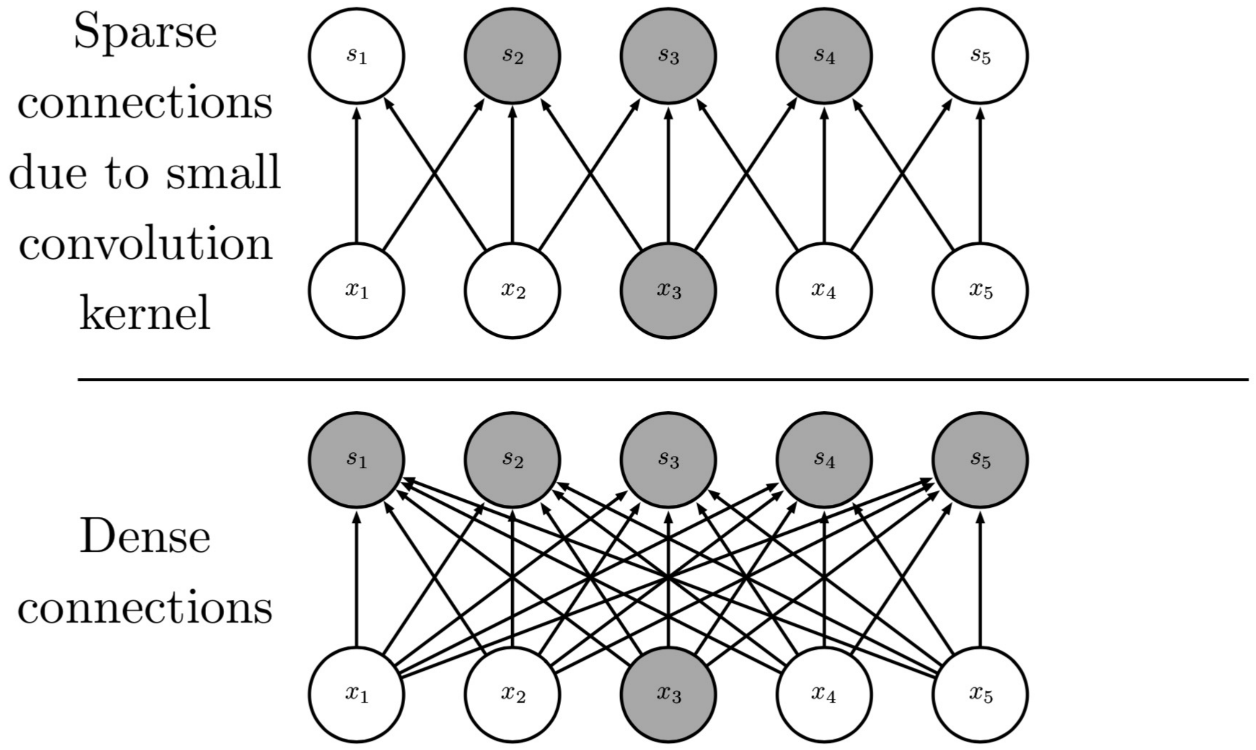

7
7

7
7

7
7

7
7


Input
a
b
c
d
e
f
g
h
i
j
k
l
Kernel
w
x
y
z
Output
$aw + bx + ey + fz$
$bw + cx + fy + gz$
$cw + dx + gy + hz$
$ew + fx + iy + jz$
$fw + gx + jy + kz$
$gw + hx + ky + lz$

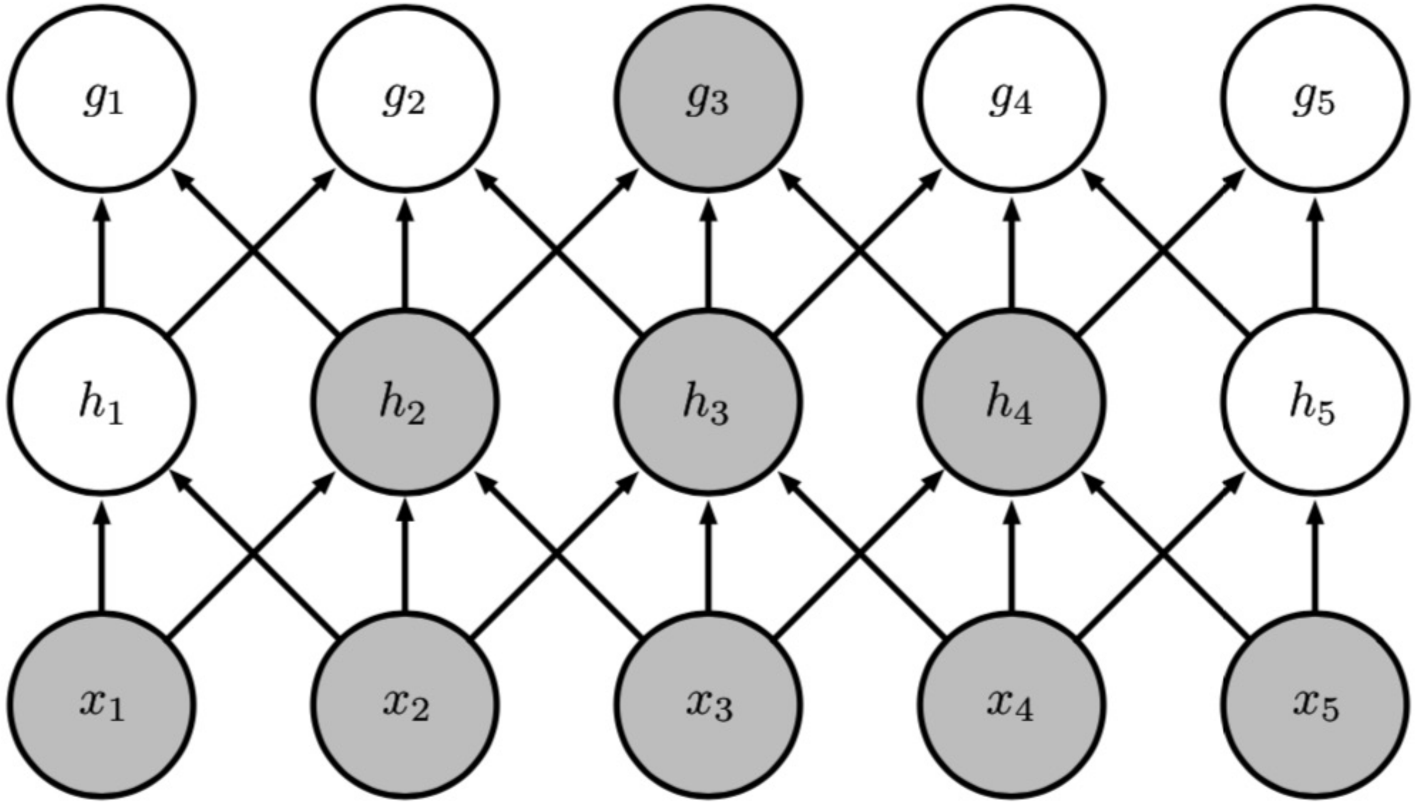
Convolution Layer



Sparse Connectivity

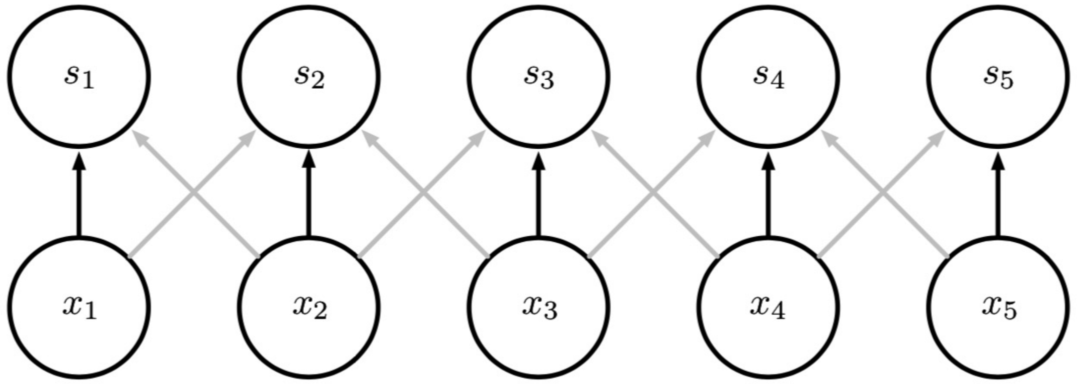


Growing Receptive Field

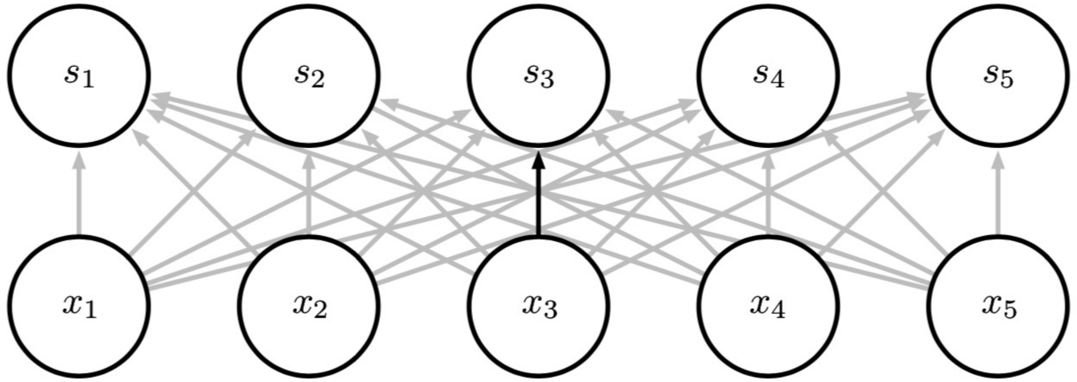


Parameter Sharing

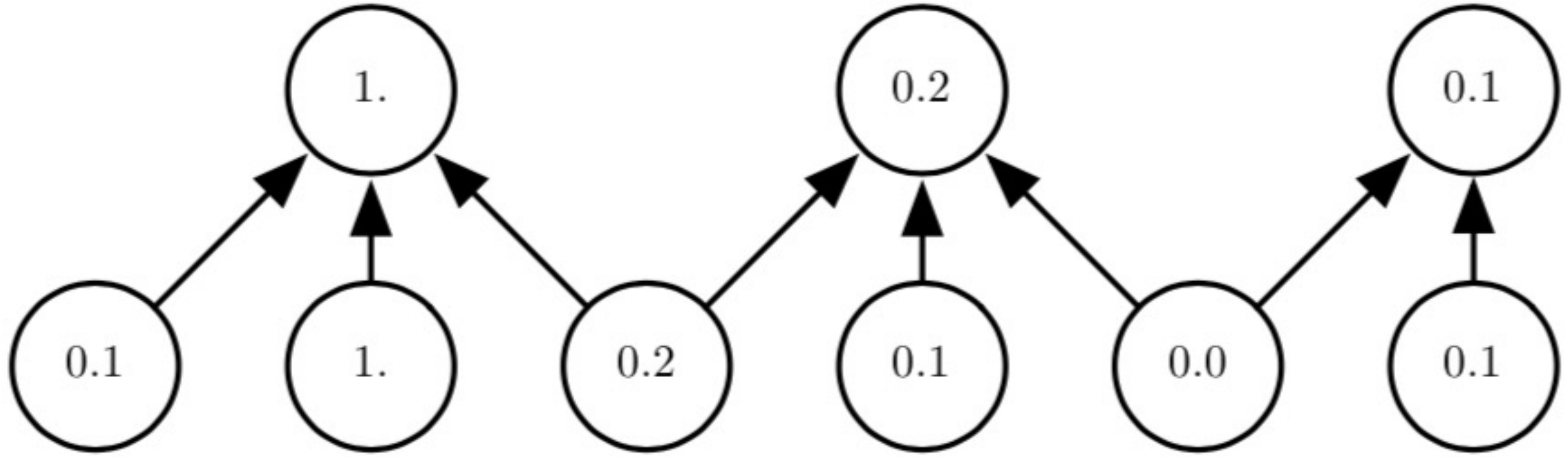
Convolution shares the same parameters across all spatial locations



Traditional matrix multiplication does not share any parameters

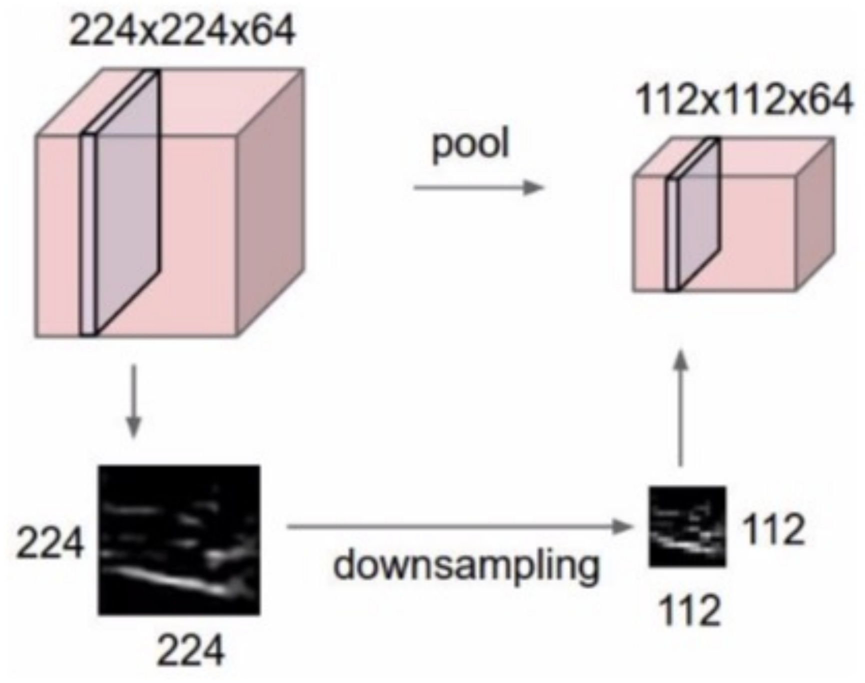


Pooling with Downsampling

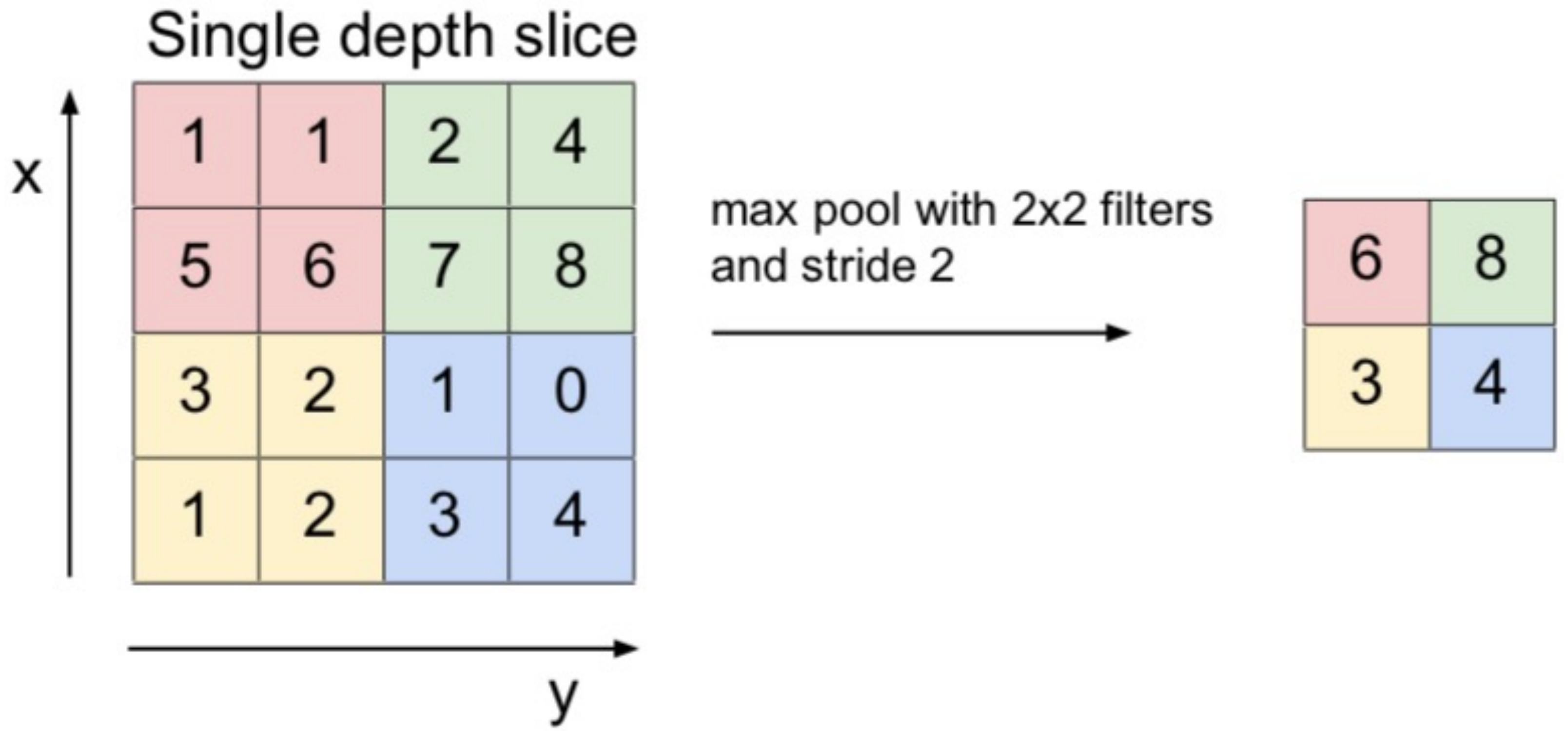


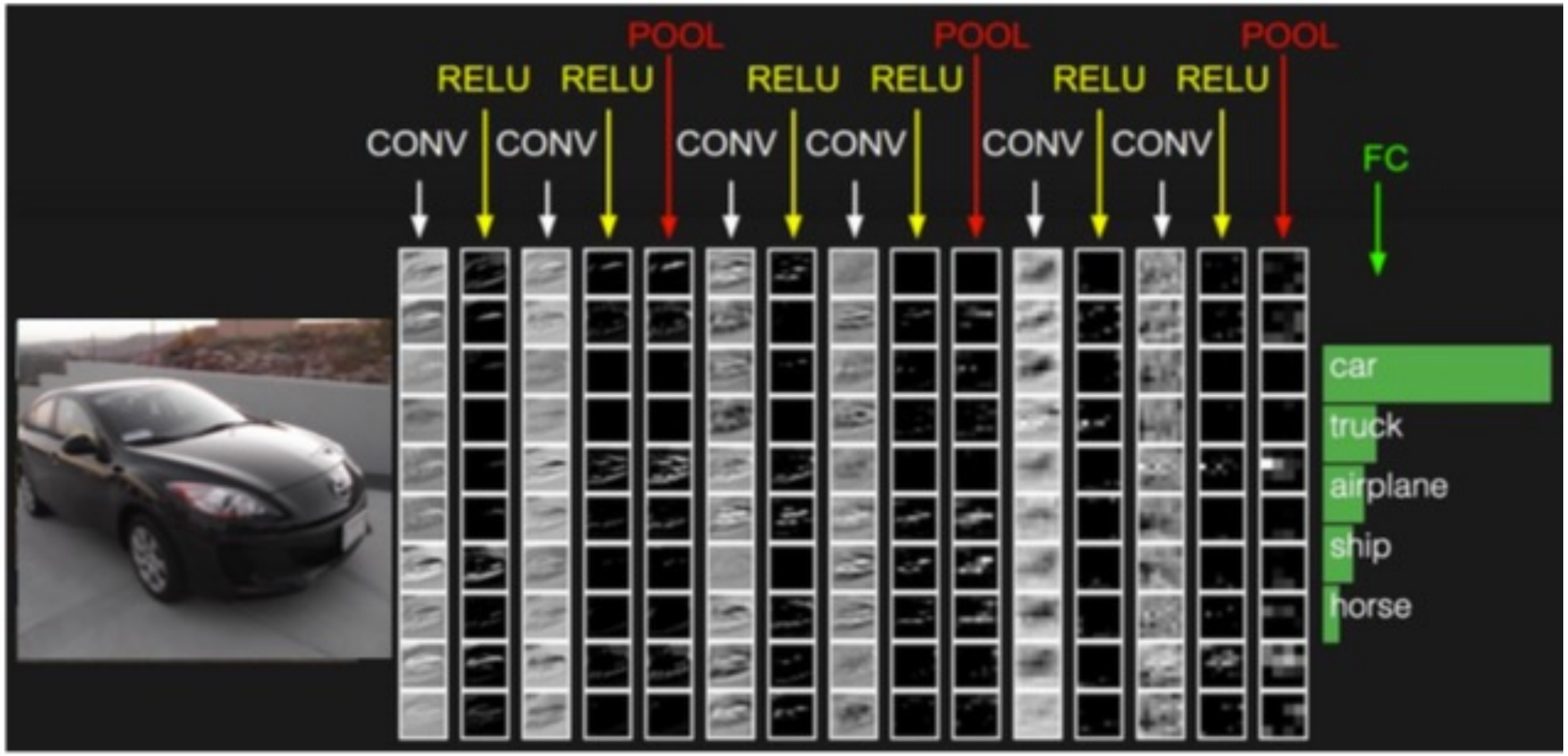
Pooling

- makes the representations smaller and more manageable
- operates over each activation map independently:

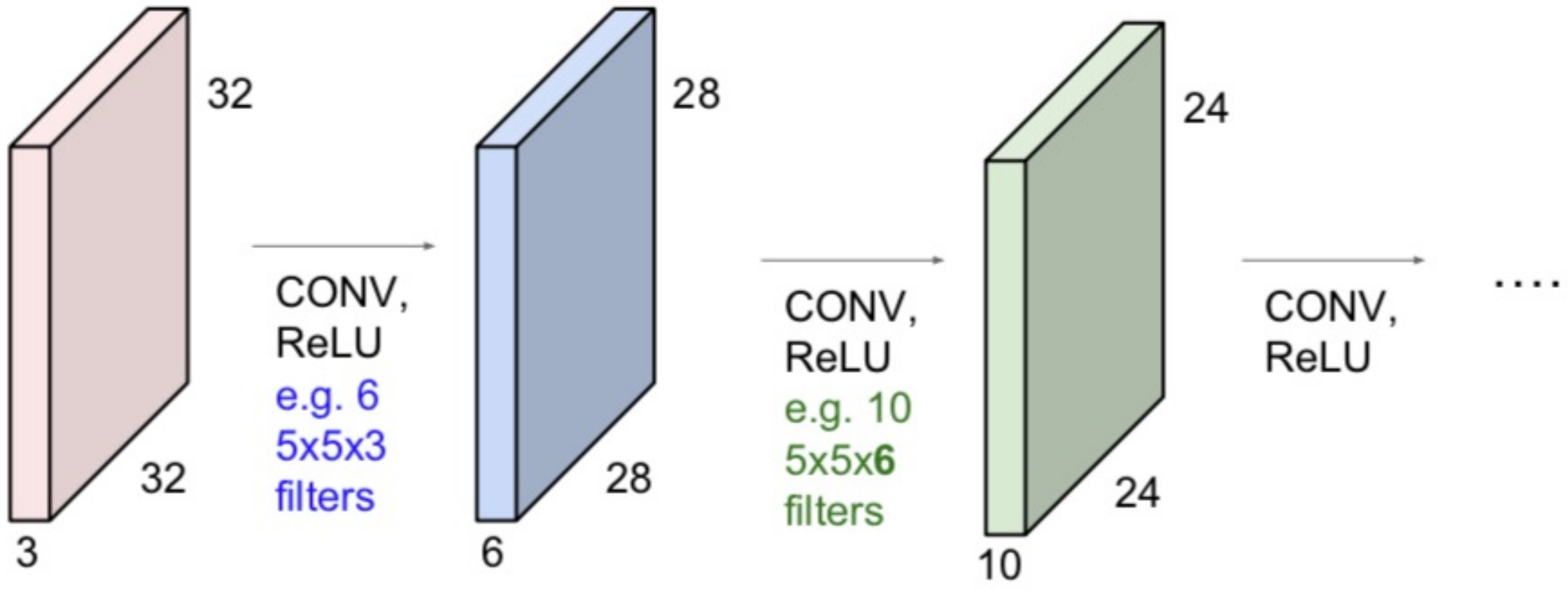


MAX POOLING


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

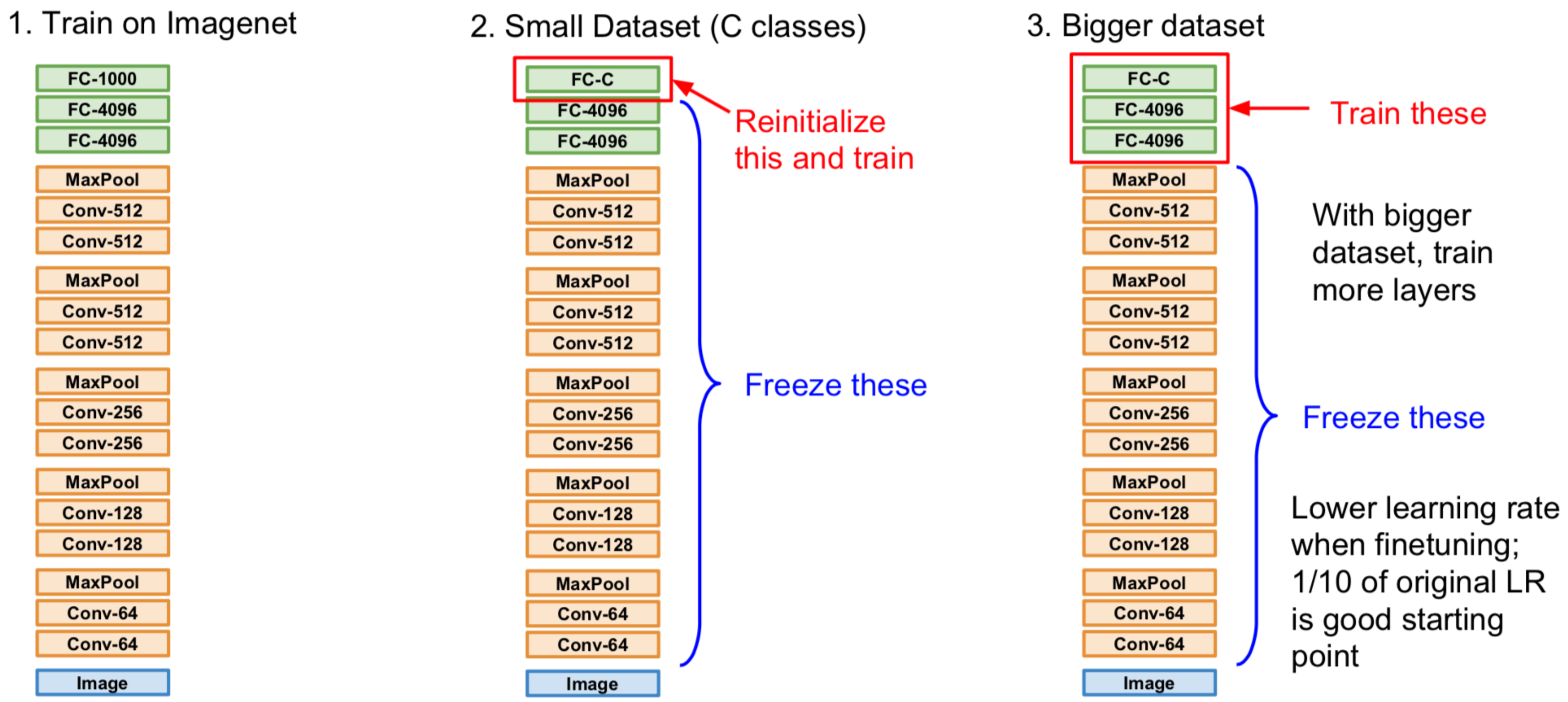
ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Transfer Learning

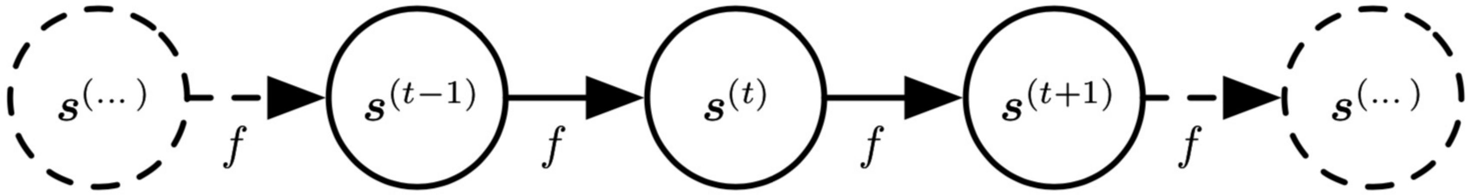
- You need a lot of a data if you want to train
- Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P_1) is exploited to improve generalization in another setting (say distribution P_2).
- We assume that many of the factors that explain the variations in P_1 are relevant to the variations that need to be captured for learning P_2.

Transfer Learning with CNNs



Classical Dynamical System

- Unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events.
- Unfolding this graph results in the sharing of parameters across a deep network structure.



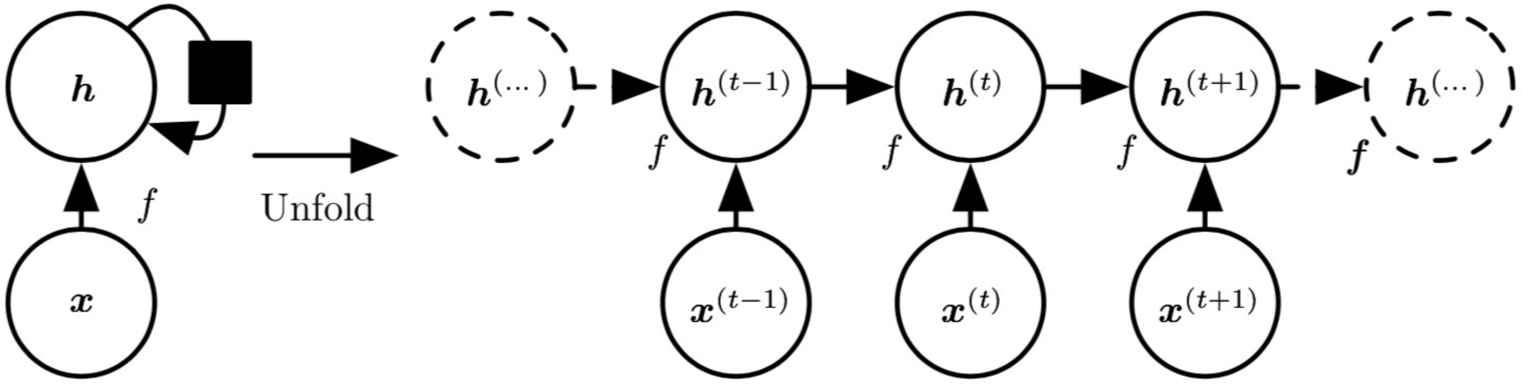
Dynamical system

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \mathbf{w}),$$

with $t = 3$

$$\mathbf{s}^{(3)} = f(\mathbf{s}^{(2)}; \mathbf{w}) = f(f(\mathbf{s}^{(1)}; \mathbf{w}); \mathbf{w})$$

Unfolding Computational Graphs



Variable $\mathbf{h}$ represents the state

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}; \mathbf{w}),$$

It maps an arbitrary length sequence

$$(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \cdots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$$

to a fixed length vector $\mathbf{h}^{(t)}$

Statistical Language Modeling

- For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, it may not be necessary to store all of the information in the input sequence up to time *t*,

- .. but rather only enough information to predict the rest of the sentence.

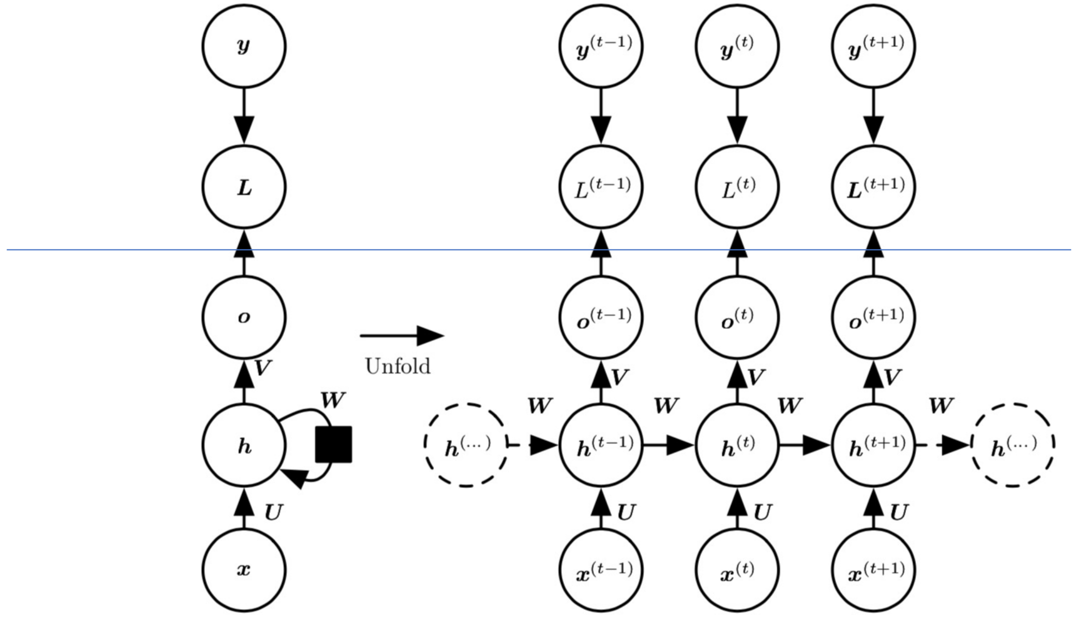
- We can represent the unfolded recurrence after *t* steps with a function g^t

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \cdot\cdot\cdot , \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{w})$$

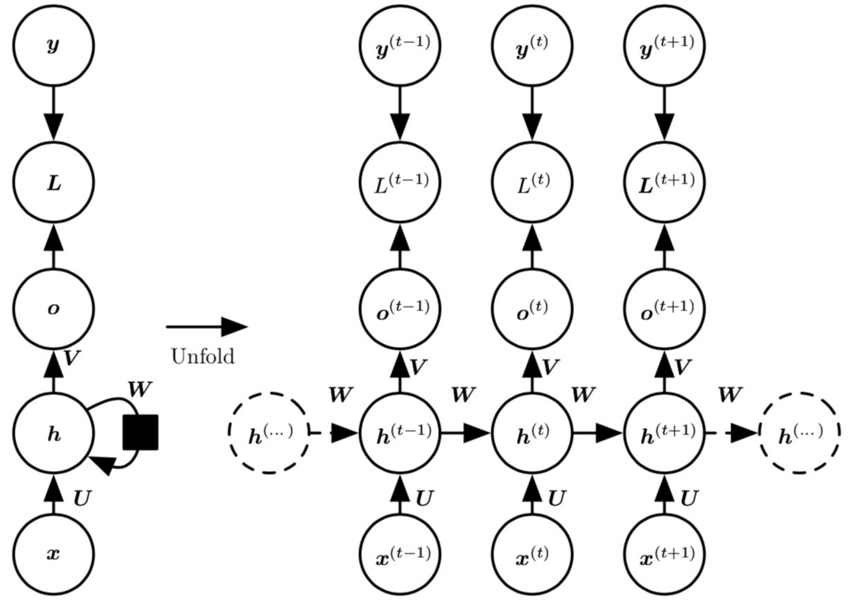
- Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.

Recurrent Neural Networks

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, Vanilla RNN or Elman RNN



Recurrent Hidden Units



With specified

$$\mathbf{h}^{(0)} = initial\ state$$

For each time step $t = 1$ to $t = \tau$

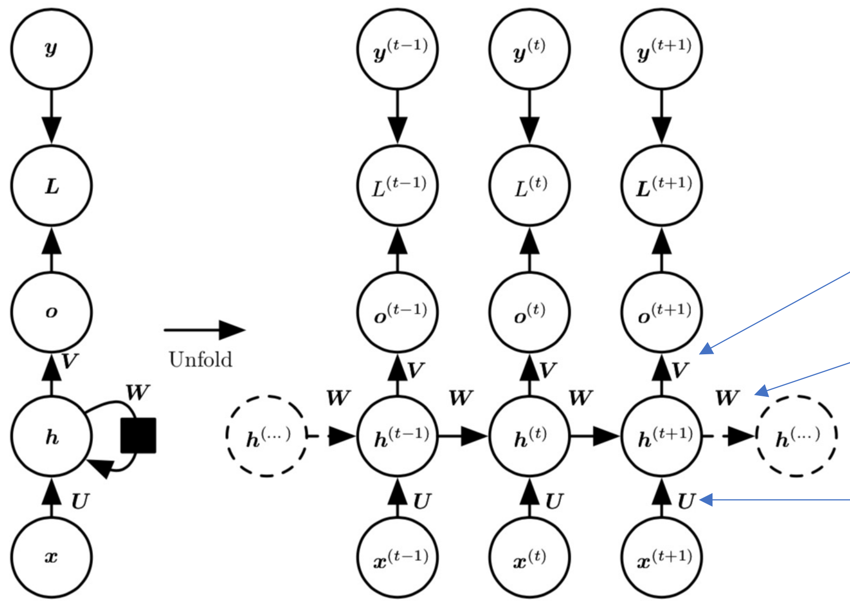
$$\mathbf{a}^{(t)} = \mathbf{b} + W \cdot \mathbf{h}^{(t-1)} + U \cdot \mathbf{x}^{(t)}$$

$$\mathbf{h}^{(t)} = tanh(\mathbf{a}^{(t)})$$

$$\mathbf{o}^{(t)} = softmax(\mathbf{c} + V \cdot \mathbf{h}^{(t)})$$

with $\mathbf{b}$ and $\mathbf{c}$ being the bias vectors and U, V, W the weight matrices

Recurrent Hidden Units

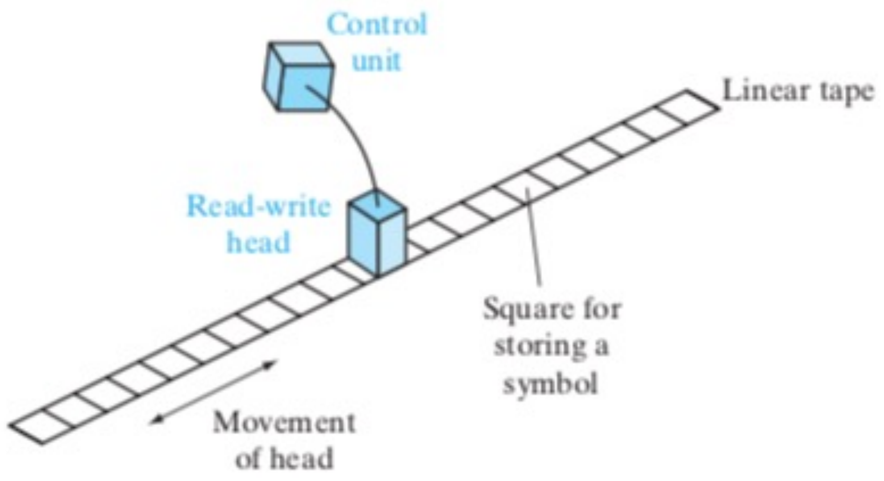


V weight matrix hidden to output layer

W weight matrix hidden to hidden layer *(new)*. Input is the preceding state.

U weight matrix input to hidden layer

Computational Power



- The recurrent neural network is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size.

Backpropagation Trough Time

- The back-propagation algorithm applied to the unrolled graph with *$O(\tau)$* cost is called back-propagation through time
- The training data for a recurrent neural network is an ordered sequence of τ input-output pairs
- Same idea as training over training set *D*, now the training set is *$D \cdot \tau$* with an additional input is the preceding state.

Backpropagation Trough Time

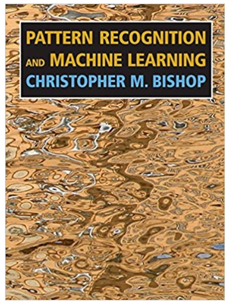
- The runtime is *O(τ)* and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may only be computed after the previous one.
- BPTT begins by unfolding a recurrent neural network in time. The unfolded network contains *τ* inputs and outputs, but every copy of the network shares the **same** parameters.
- Then the backpropagation algorithm is used to find the gradient of the cost with respect to all the network parameters

Conclusion

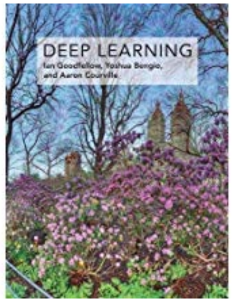
- Deep learning is: a **black box** but it is also a **black art**.
- Many approaches and hyperparameters:
  - activation functions,
  - learning rate,
  - momentum?

- Often these need tweaking, and you need to know what they do to change them intelligently.

Literature

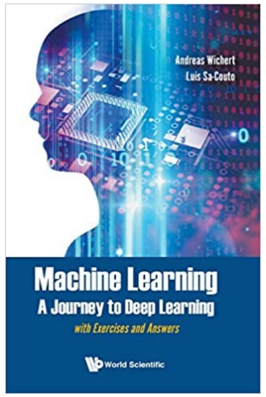


- Christopher M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics), Springer 2006
  - Chapter 5



- Deep Learning, I. Goodfellow, Y. Bengio, A. Courville MIT Press 2016
  - Chapter **6**, 7, 8

Literature



- Machine Learning - A Journey to Deep Learning, A. Wichert, Luis Sa-Couto, World Scientific, 2021
  - Chapter 12