NEURAL NETWORKS

Deep learning = Deep neural networks = neural networks

DNNs (Deep Neural Networks)

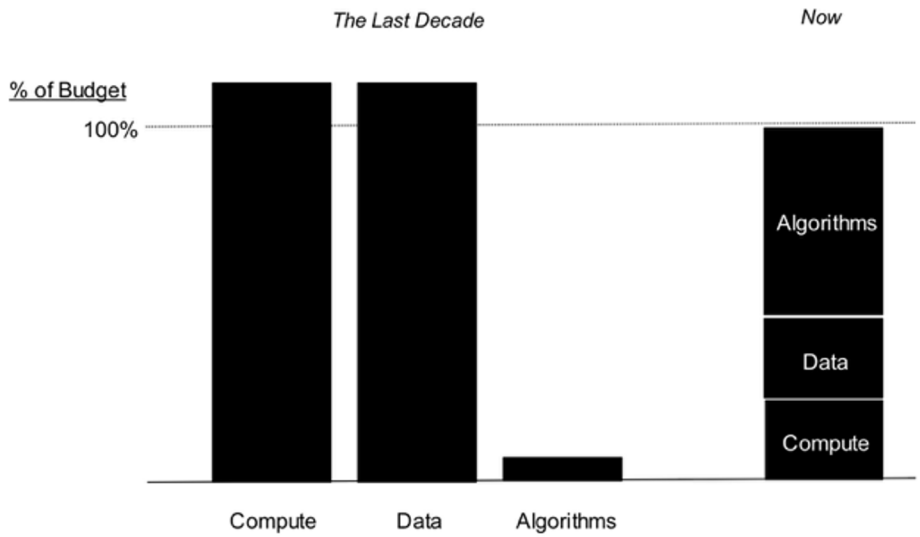
- Why deep learning?
- Greatly improved performance in ASR and other tasks (Computer Vision, Robotics, Machine Translation, NLP, etc.)
- Surpassed human performance in many tasks

Task	Previous state-of-the-art	Deep learning (2012)	Deep learning (2019)
TIMIT	24.4%	20.0%	13.8%
Switchboard	23.6%	16.1%	5.0%
Google voice search	16.0%	12.3%	4.9%
MOOC (Thai)	38.7%		19.6%

Geoffrey Hinton, Deep Neural Networks for Acoustic Modeling in Speech Recognition, 2012

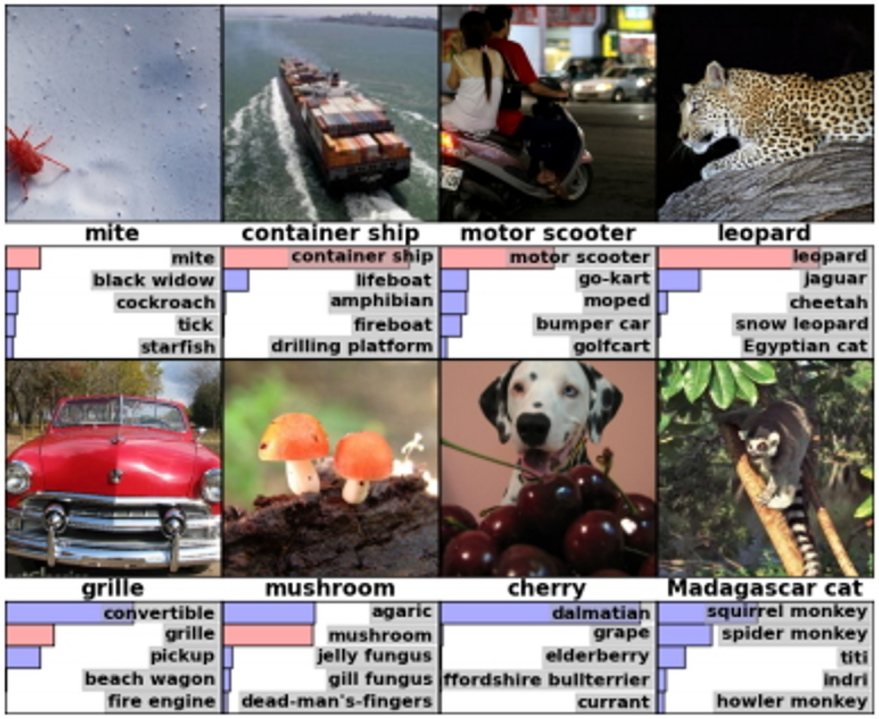
Why now

- Neural Networks has been around since 1990s
- Big data – DNN can take advantage of large amounts of data better than other models
- GPU – Enable training bigger models possible
- Deep – Easier to avoid bad local minima when the model is large



http://www.kdnuggets.com/2017/06/practical-guide-machine-learning-understand-differentiate-apply.html

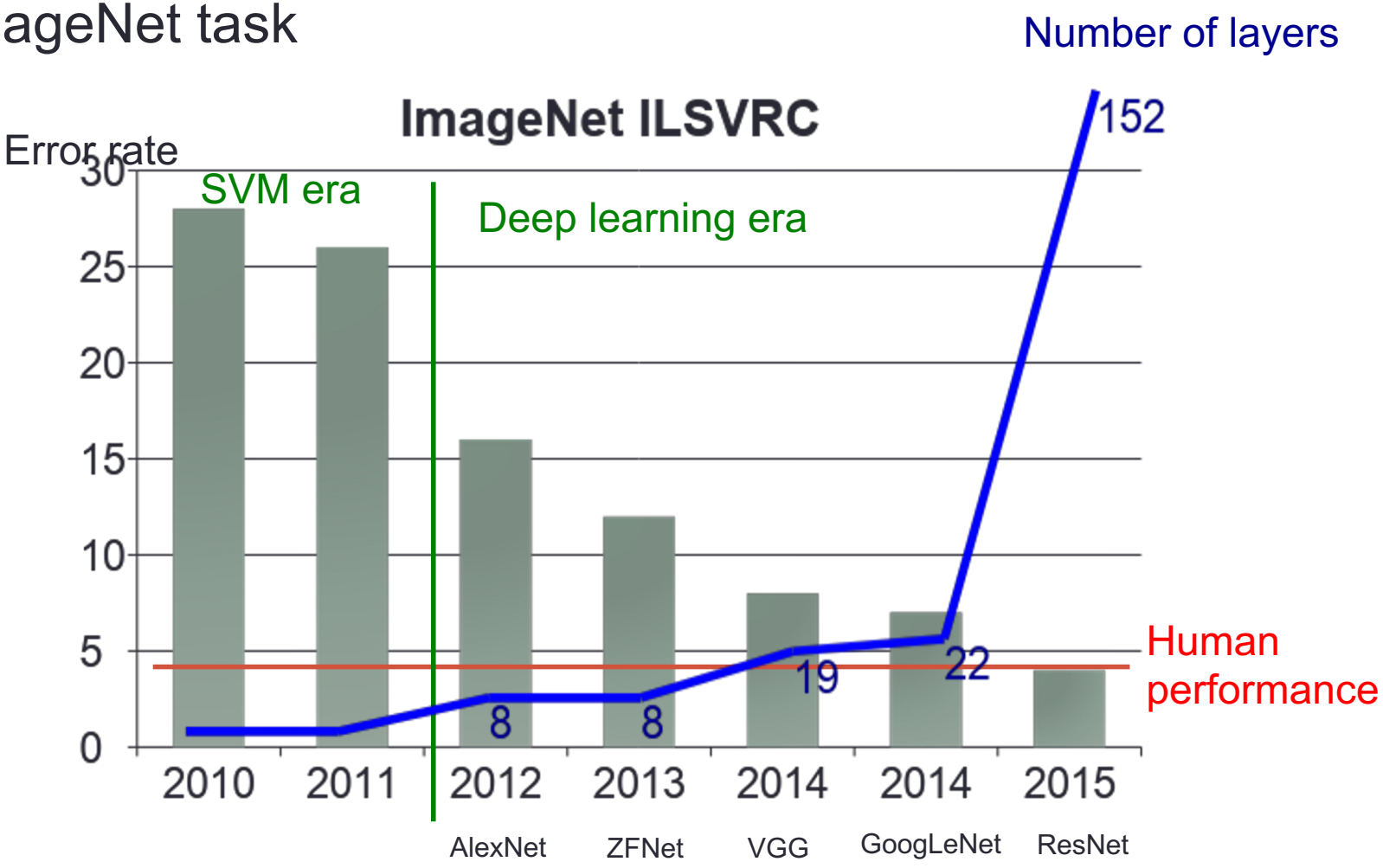
ImageNet - Object classification



Alex, Krizhevsky, Imagenet classification with deep convolutional neural networks, 2012

Wider and deeper networks

- ImageNet task



Olga Russakovsky, ImageNet Large Scale Visual Recognition Challenge, 2014 https://arxiv.org/abs/1409.0575

Statistics > Machine Learning

Dynamical Isometry and a Mean Field Theory of CNNs: How to Train 10,000-Layer Vanilla Convolutional Neural Networks

Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel S. Schoenholz, Jeffrey Pennington

(Submitted on 14 Jun 2018)

In recent years, state-of-the-art methods in computer vision have utilized increasingly deep convolutional neural network architectures (CNNs), with some of the most successful models employing hundreds or even thousands of layers. A variety of pathologies such as vanishing/exploding gradients make training such deep networks challenging. While residual connections and batch normalization do enable training at these depths, it has remained unclear whether such specialized architecture designs are truly necessary to train deep CNNs. In this work, we demonstrate that it is possible to train vanilla CNNs with ten thousand layers or more simply by using an appropriate initialization scheme. We derive this initialization scheme theoretically by developing a mean field theory for signal propagation and by characterizing the conditions for dynamical isometry, the equilibration of singular values of the input-output Jacobian matrix. These conditions require that the convolution operator be an orthogonal transformation in the sense that it is norm-preserving. We present an algorithm for generating such random initial orthogonal convolution kernels and demonstrate empirically that they enable efficient training of extremely deep architectures.

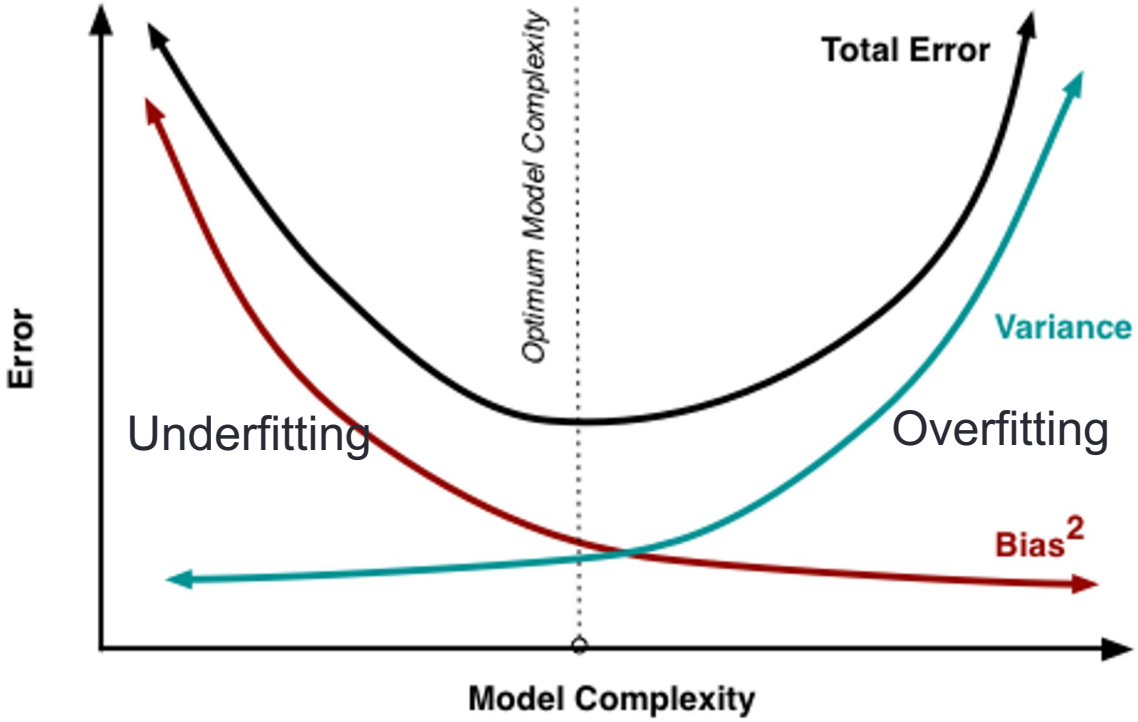
Comments: ICML 2018 Conference Proceedings
Subjects: **Machine Learning (stat.ML)**; Machine Learning (cs.LG)
Cite as: arXiv:1806.05393 **[stat.ML]**
(or **arXiv:1806.05393v1 [stat.ML]** for this version)

Submission history

From: Samuel Schoenholz [view email]
[v1] Thu, 14 Jun 2018 07:04:15 GMT (6734kb,D)

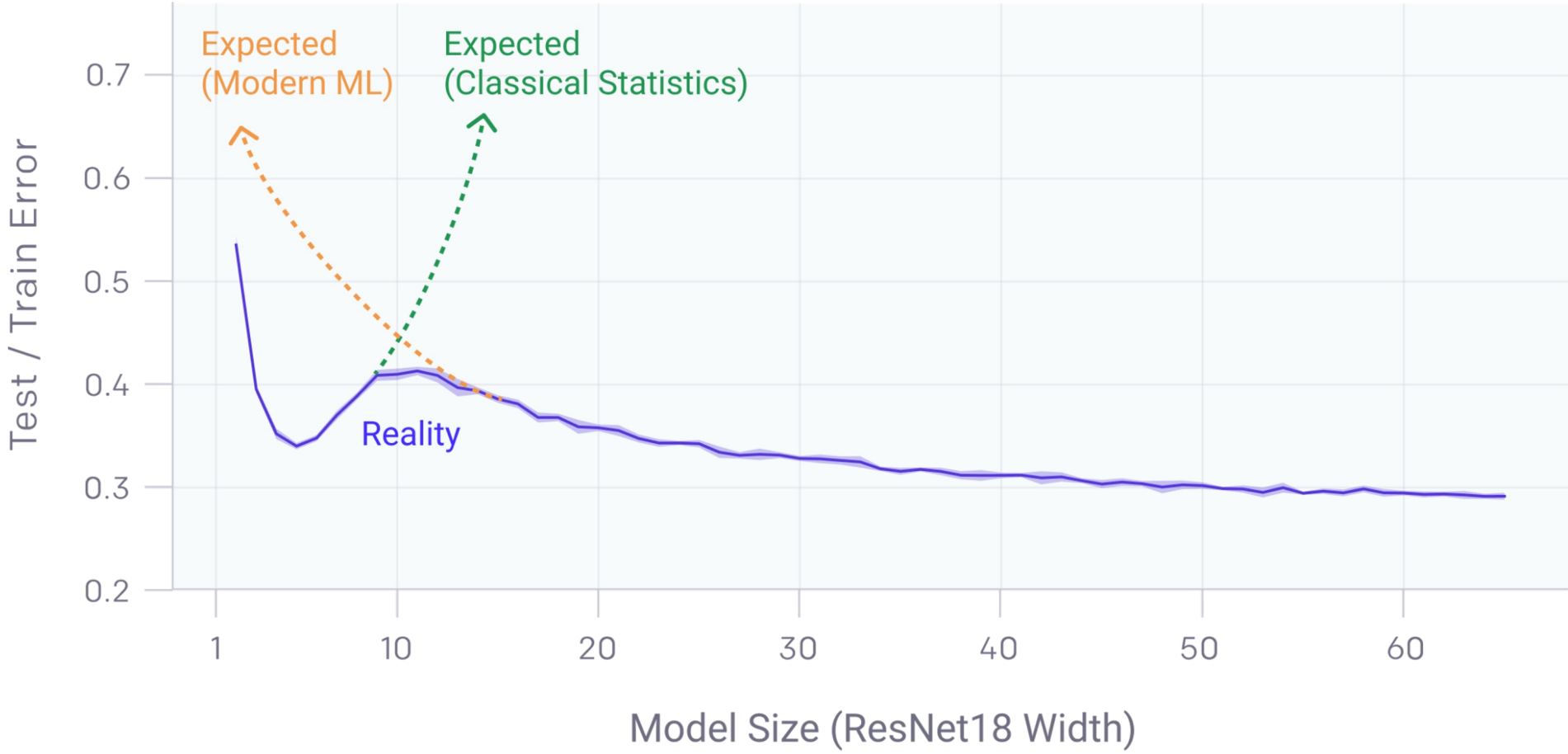
Bias-Variance Underfitting-Overfitting

- Usually if you try to reduce the bias of your model, the variance will increase, and vice versa.
- Called the bias-variance trade-off



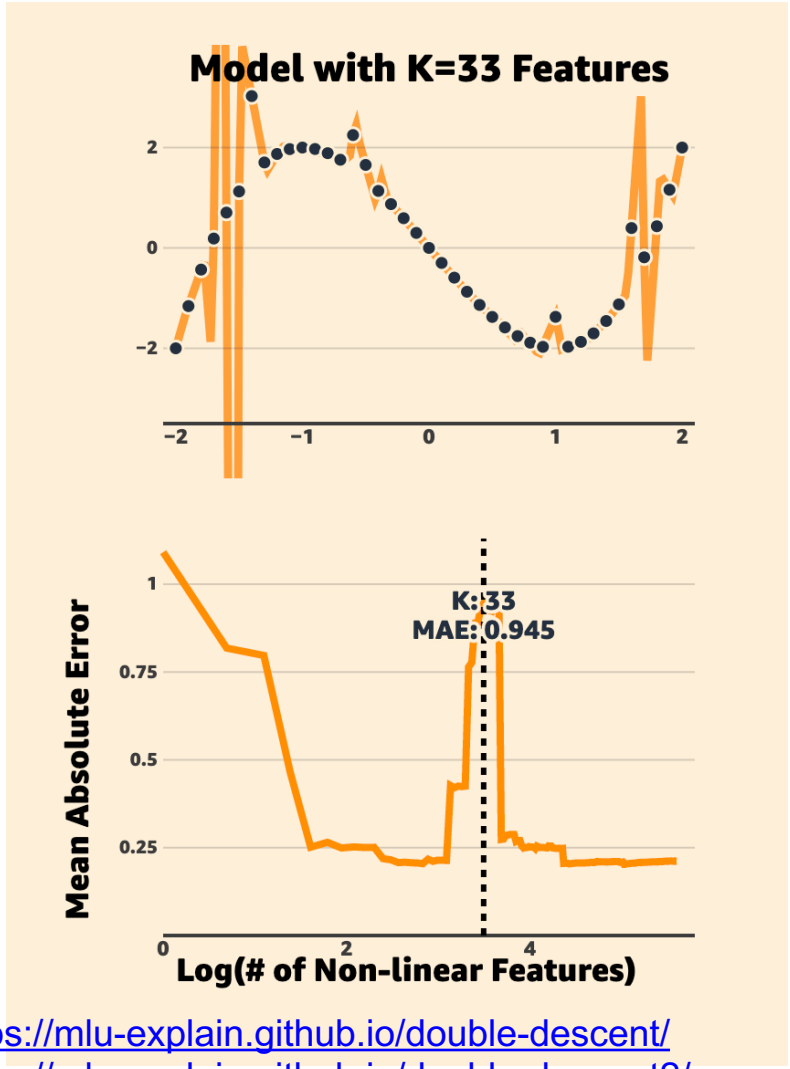
http://www.cs.cornell.edu/courses/cs4780/2017sp/lectures/lecturenote11.html

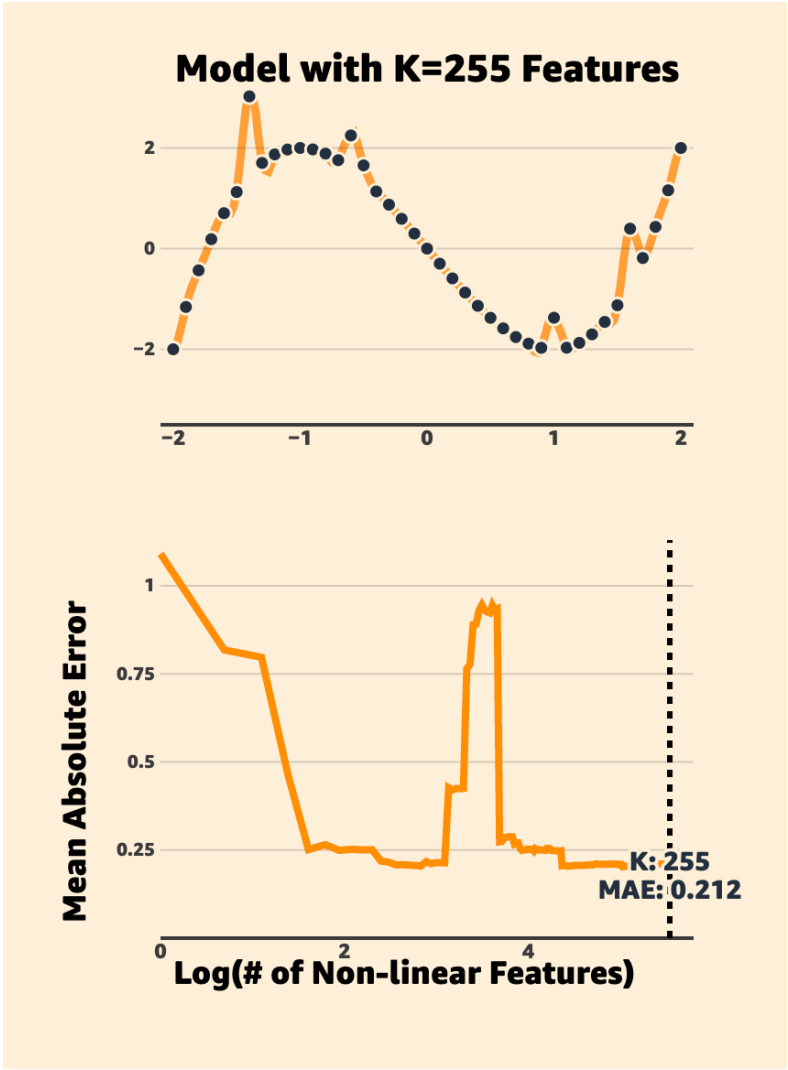
The double descent problem



https://openai.com/blog/deep-double-descent/

The double descent problem





https://mlu-explain.github.io/double-descent/
https://mlu-explain.github.io/double-descent2/

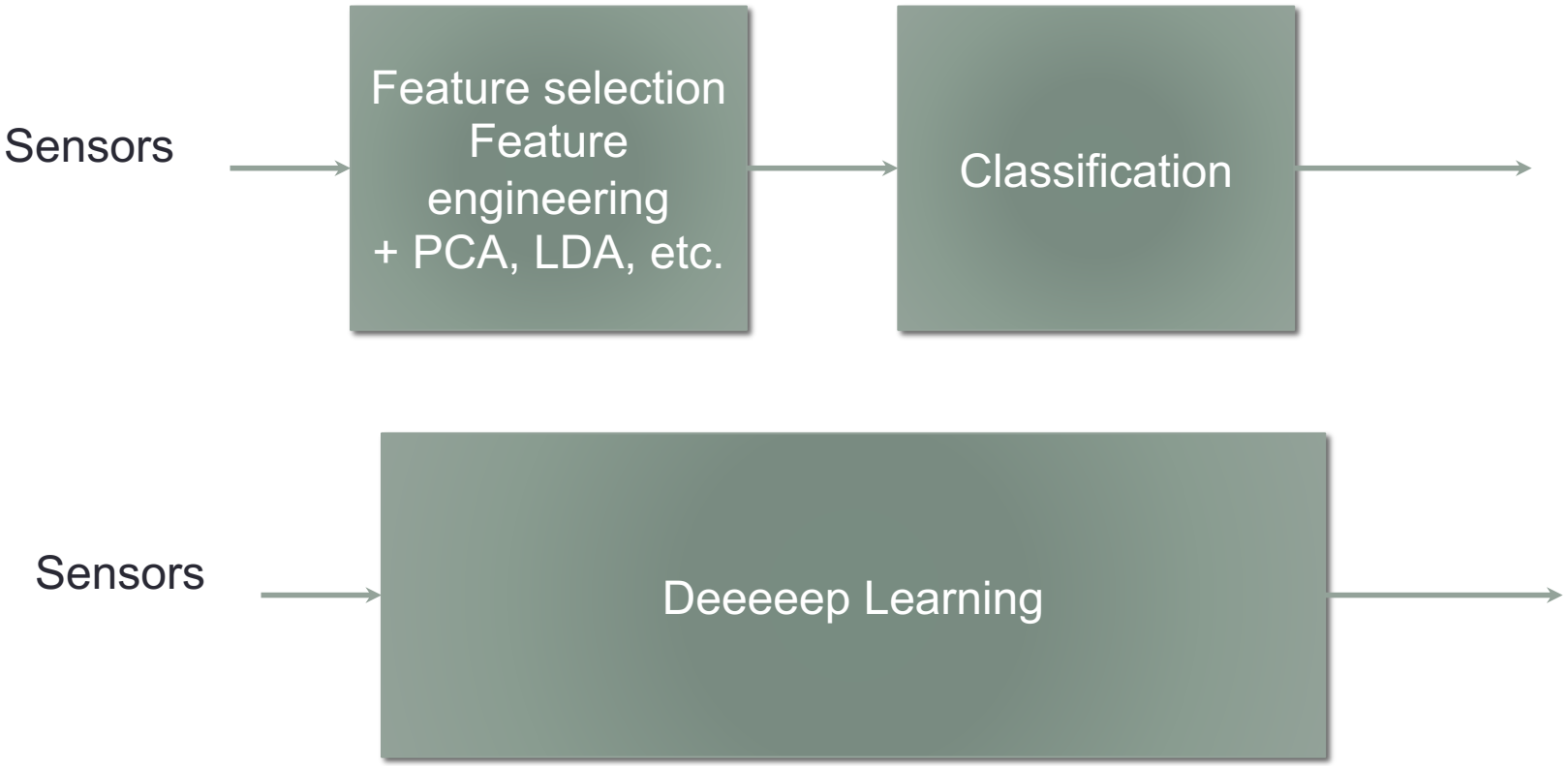
Inductive bias and ML

- The **inductive bias** of a learning algorithm is a set of assumptions that the algorithm used to generalize to new inputs.

http://www.cs.cmu.edu/~tom/pubs/NeedForBias_1980.pdf

- Your choice of model forces a certain type of behavior
  - Tells the model how to "overfit" the training data
- Putting inductive bias into deep learning model is easier than other models
  - ~~Domain knowledge to construct features~~
  - Domain knowledge to encourage certain learning behaviors

Traditional VS Deep learning

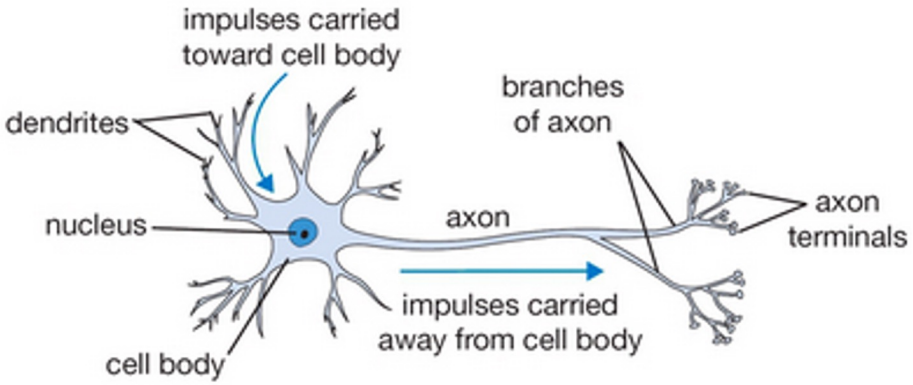


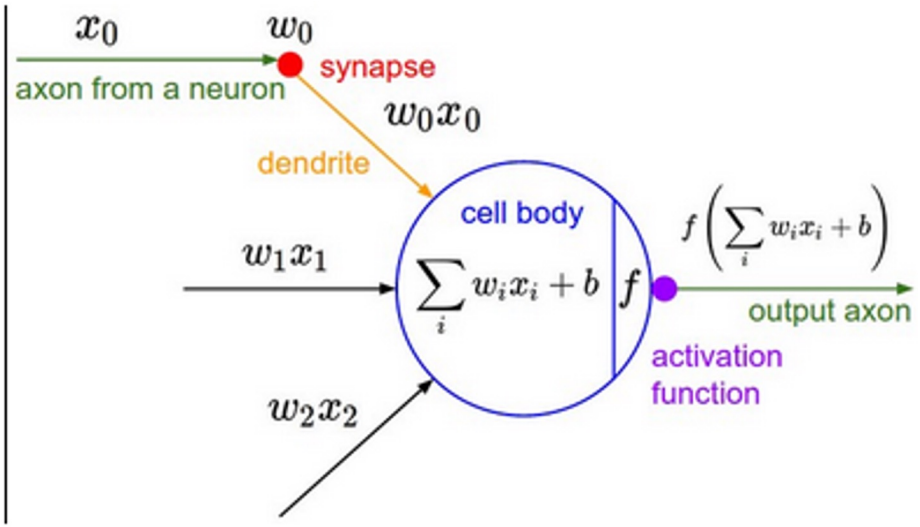
Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function and regularization
  - SGD and backprop
  - Learning rate
  - Overfitting – dropout, batchnorm
- CNN, RNN, LSTM, GRU

Fully connected networks

- Many names: feed forward networks or deep neural networks or multilayer perceptron or artificial neural networks
- Composed of multiple neurons

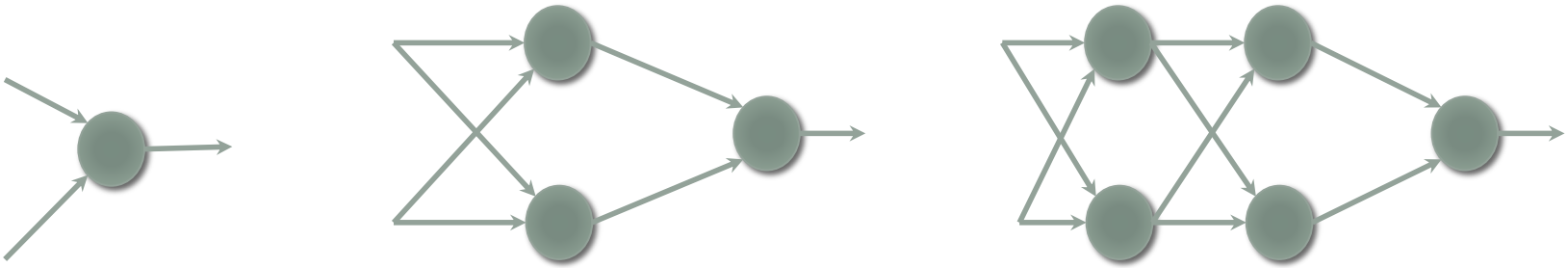


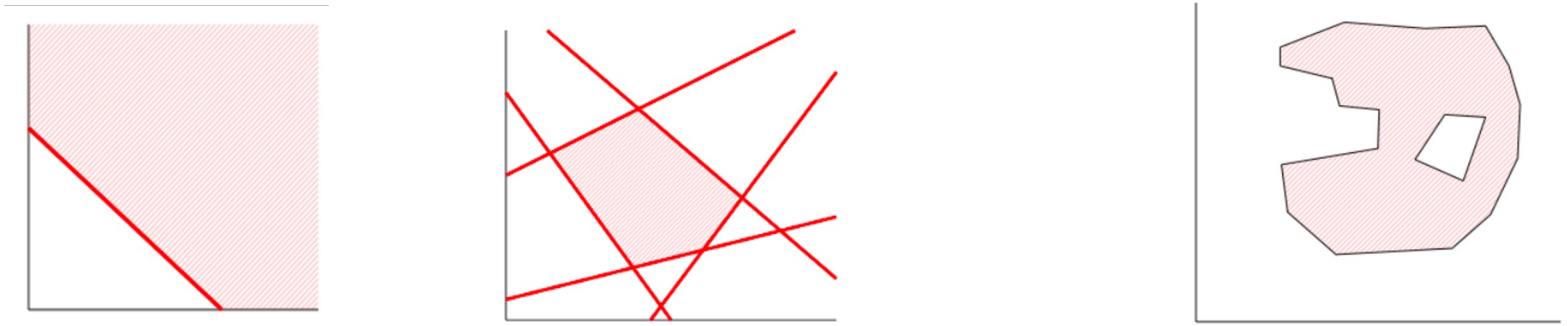


http://cs231n.github.io/neural-networks-1/

Combining neurons

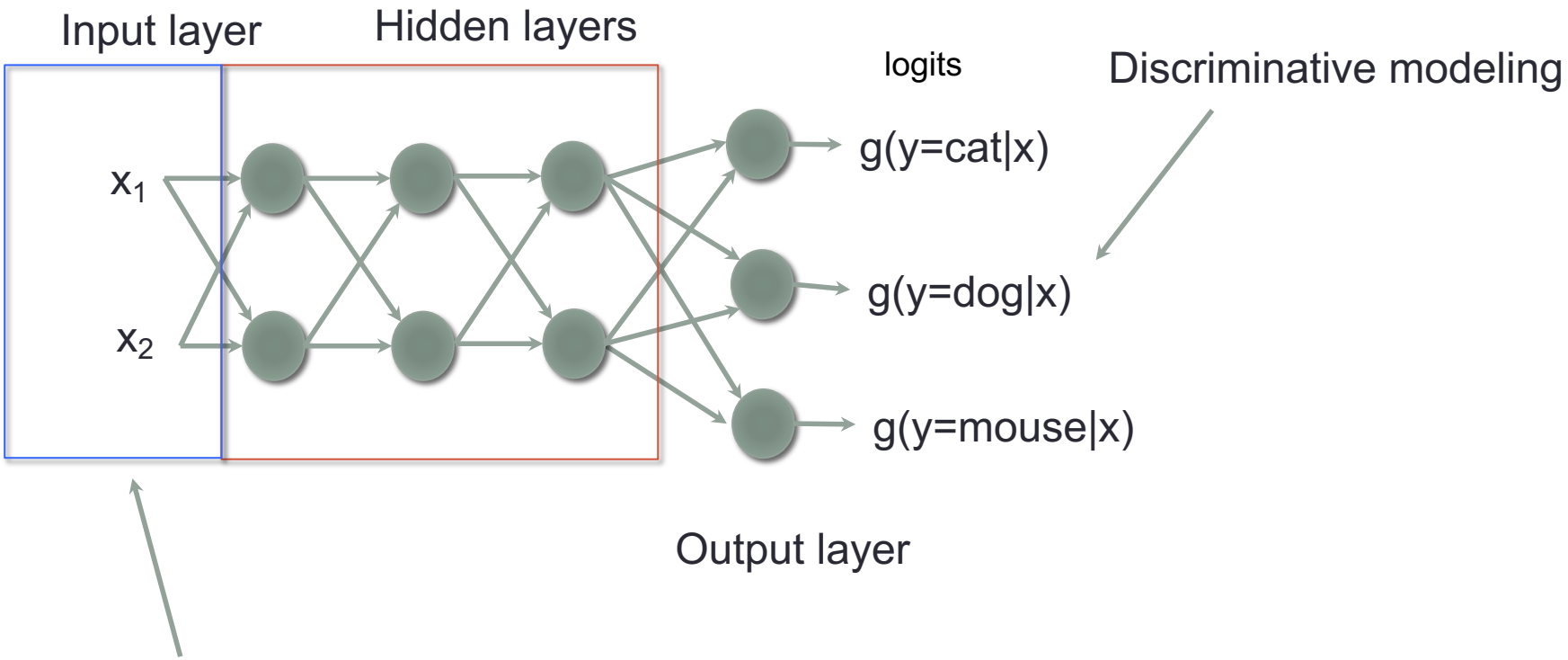
- Each neuron splits the feature space with a hyperplane
- Stacking neuron creates more complicated decision boundaries
- More powerful but prone to overfitting





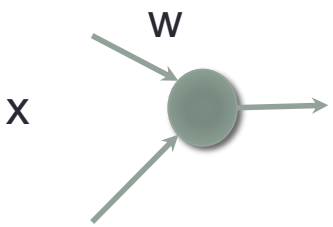
Terminology

Deep in Deep neural networks means many hidden layers



Projections and Neural network weights

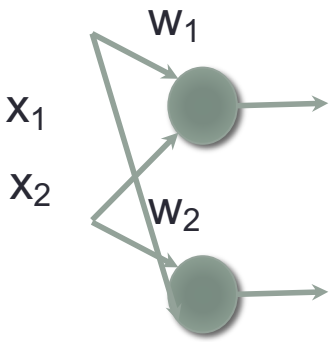
- $w^T x$

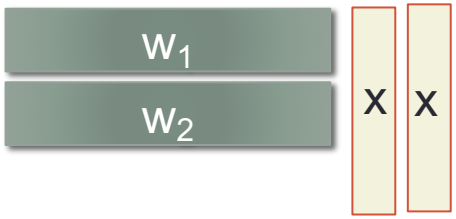


Projections and neural network weights

- $W^T[x_1, x_2]$

You can pack multiple inputs together to do one single matrix multiply

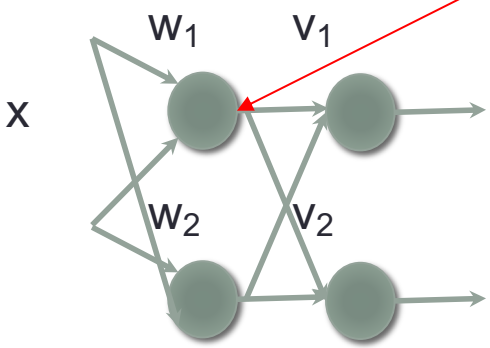




Neural network layer acts as nonlinear feature transform

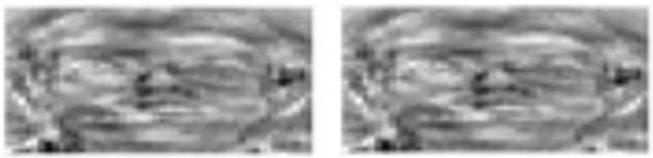
- $W^T x$

Without the nonlinearity the two matrices combine into one operation

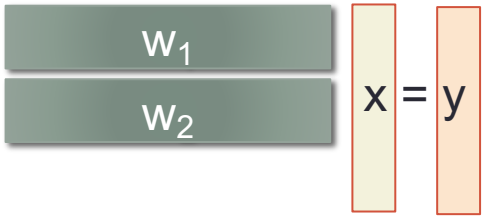


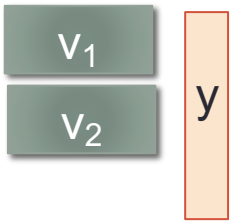
fischer projection = $V^T W^T x$
$= (WV)^T x$

Wv_1 Wv_2



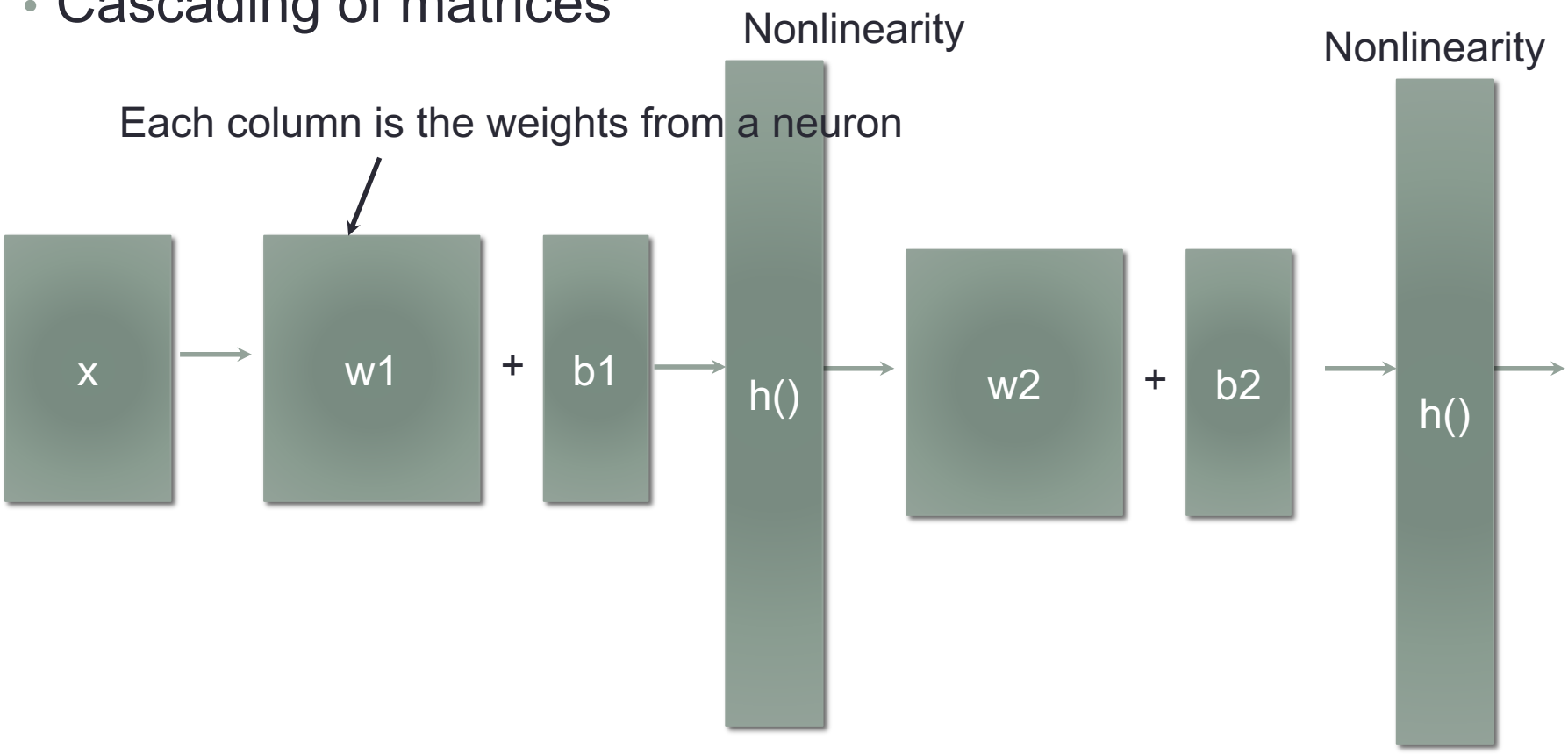
LDA projections





More linear algebra

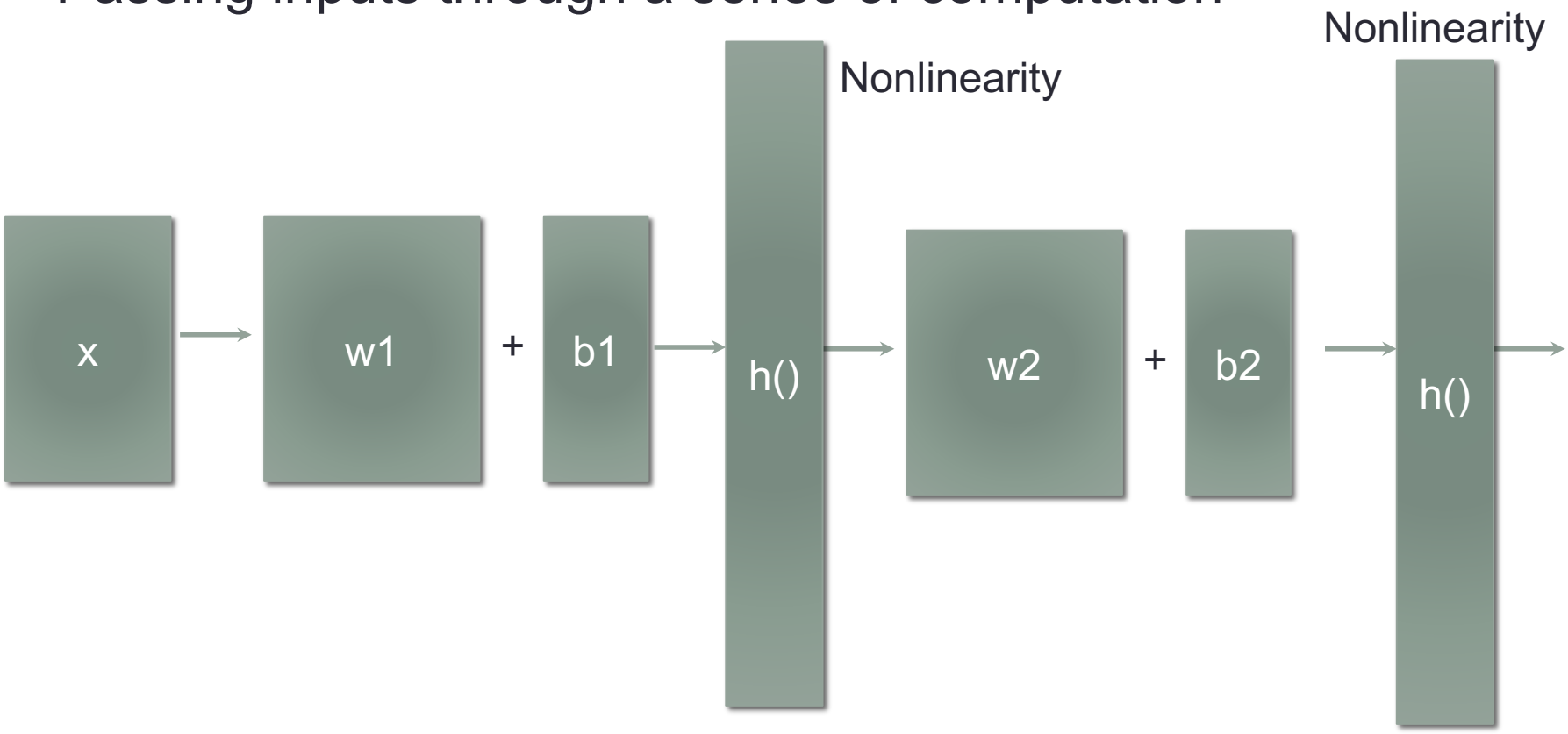
- Cascading of matrices



$h(W_2^T\, h(W_1^T X + \mathbf{b_1}) + \mathbf{b_2})$

Computation graph

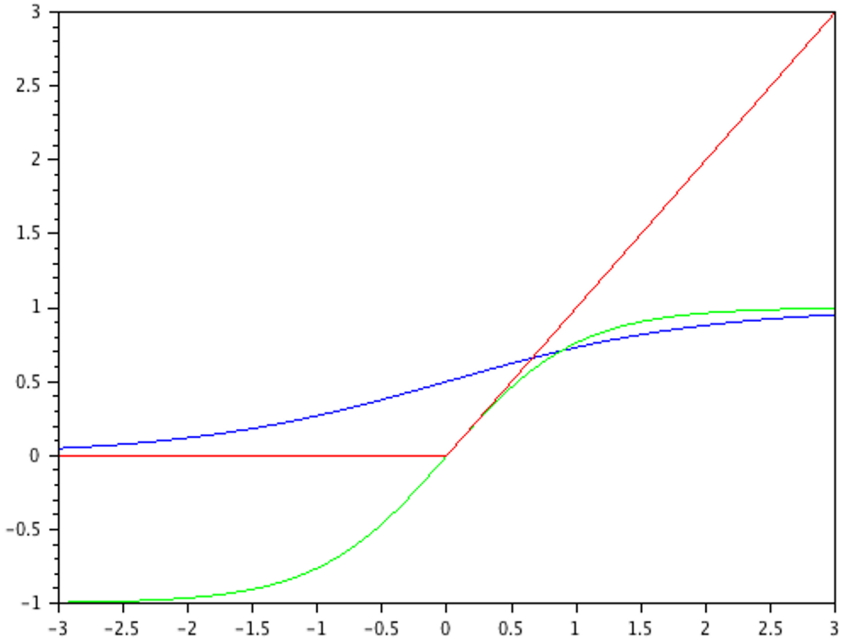
- Passing inputs through a series of computation



$h(W_2^T\, h(W_1^T X + \mathbf{b_1}) + \mathbf{b_2})$

Non-linearity

- The Non-linearity is important in order to stack neurons
- Sigmoid or logistic function
- tanh
- Rectified Linear Unit (ReLU)
  - LeakyReLU, ELU, PreLU
- Sigmoid Linear Units (SiLU)
  - Swish, Mish, GELU
- Most popular is ReLU and its variants



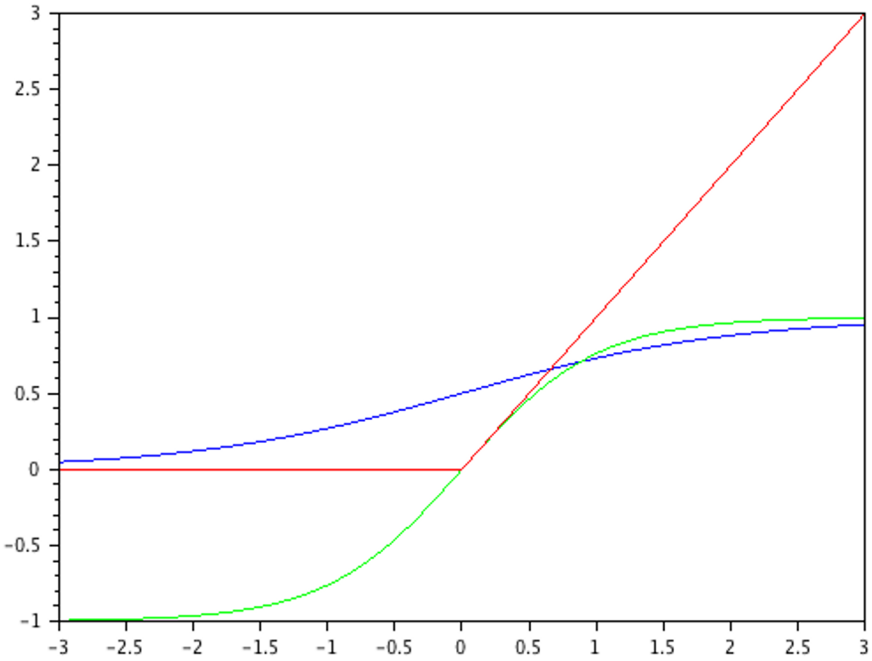
Non-linearity

- Sigmoid $\frac{1}{1+e^{-x}}$
- tanh

$$tanh(x)$$

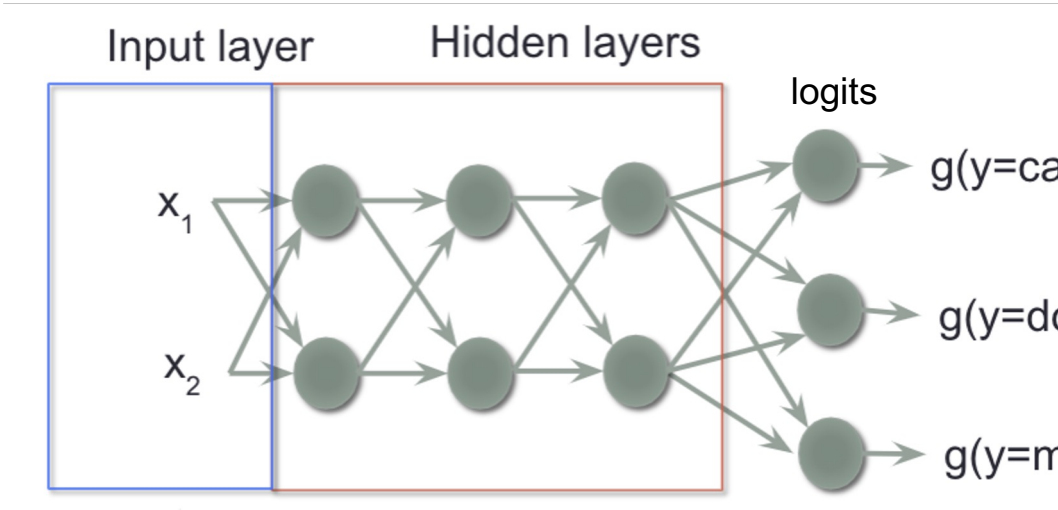
- Rectified Linear Unit (ReLU)

$$max(0, x)$$



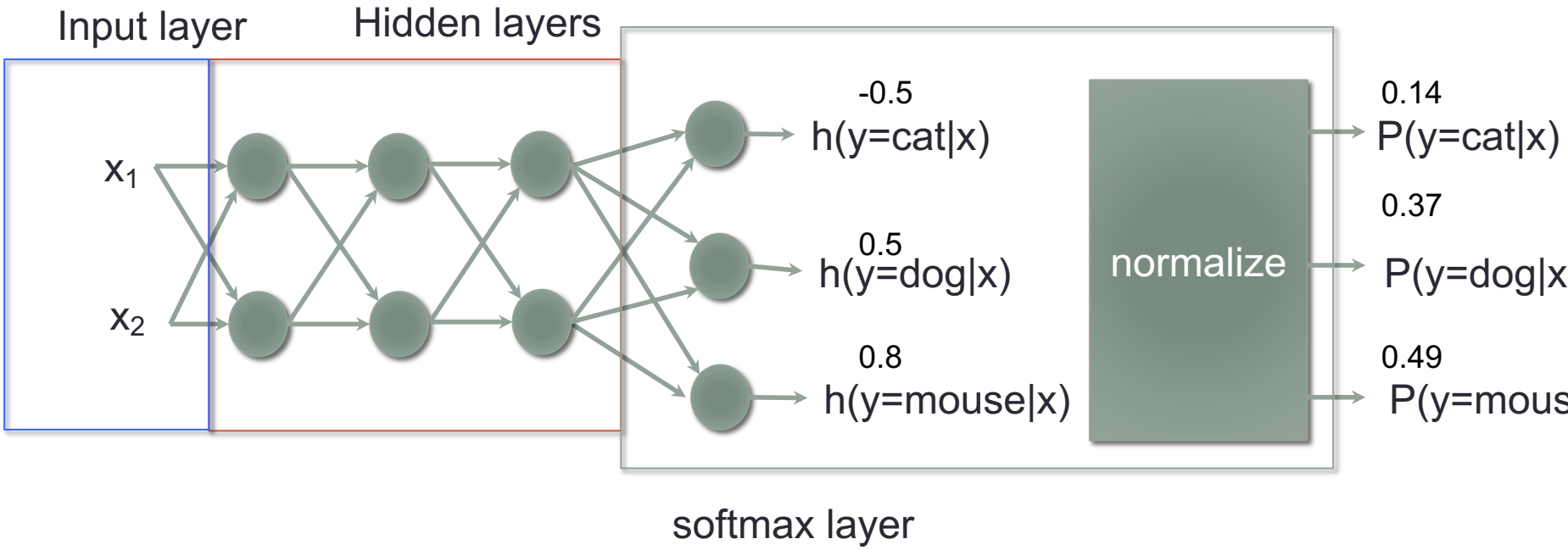
Output layer – Softmax layer

- We usually wants the output to mimic a probability function (0<=P<=1,sums to 1)
- Current setup has no such constraint
- The current output should have highest value for the correct class.
  - Value can be positive or negative number
- Takes the exponent
- Add a normalization



Softmax layer

$$P(y=j|x) = \frac{e^{h(y=j|x)}}{\sum_y e^{h(y|x)}}$$



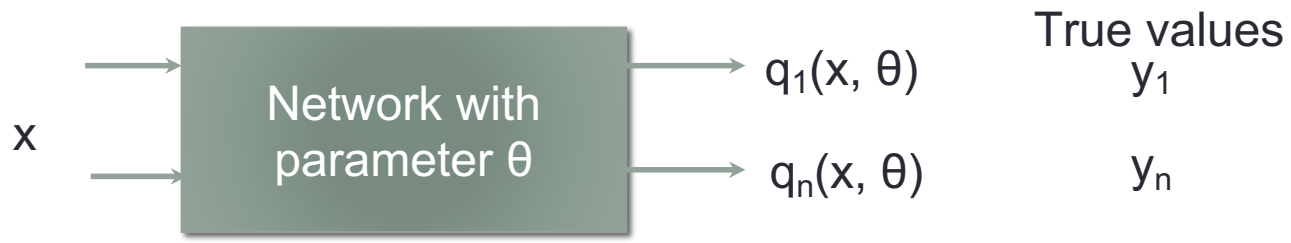
Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function and regularization
  - SGD and backprop
  - Learning rate
  - Overfitting – dropout, batchnorm
- CNN, RNN, LSTM, GRU <- Next class

Objective function (Loss function)

- Can be any function that summarizes the performance into a single number
- Cross entropy

- Sum of squared errors



Cross entropy loss

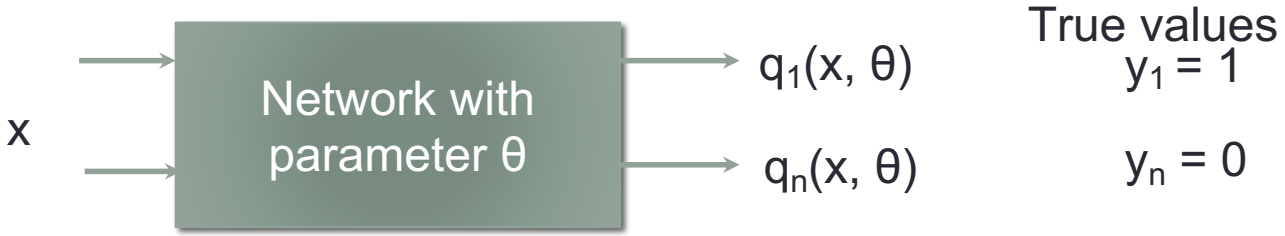
- Used for softmax outputs (probabilities), or classification tasks

Note Eq is calculated on one sample
Summed over whole batch for total loss

$$L = -\Sigma_n y_n log q_n(x, \theta)$$

- Where y_n is 1 if data x comes from class n
  0 otherwise

- L only has the term from the correct class
- L is non-negative with highest value when the output matches the true values, a “loss” function



Cross entropy loss & Logarithmic Loss (log loss)

- Minimizing the CE can be considered as the maximizing the log likelihood

$$L = -\Sigma_n y_n log q_n(x, \theta)$$

- Where y_n is 1 if data x comes from class n
  0 otherwise
- For binary class: $L(x_n) = \begin{cases} -\log(h(x_n)) & \text{, if } y_n = 1 \\ -\log(1-h(x_n)) & \text{, if } y_n = 0 \end{cases}$

$$L = [y_n log(h(x_n))] + [(1 - y_n) log(1 - h(x_n))]$$

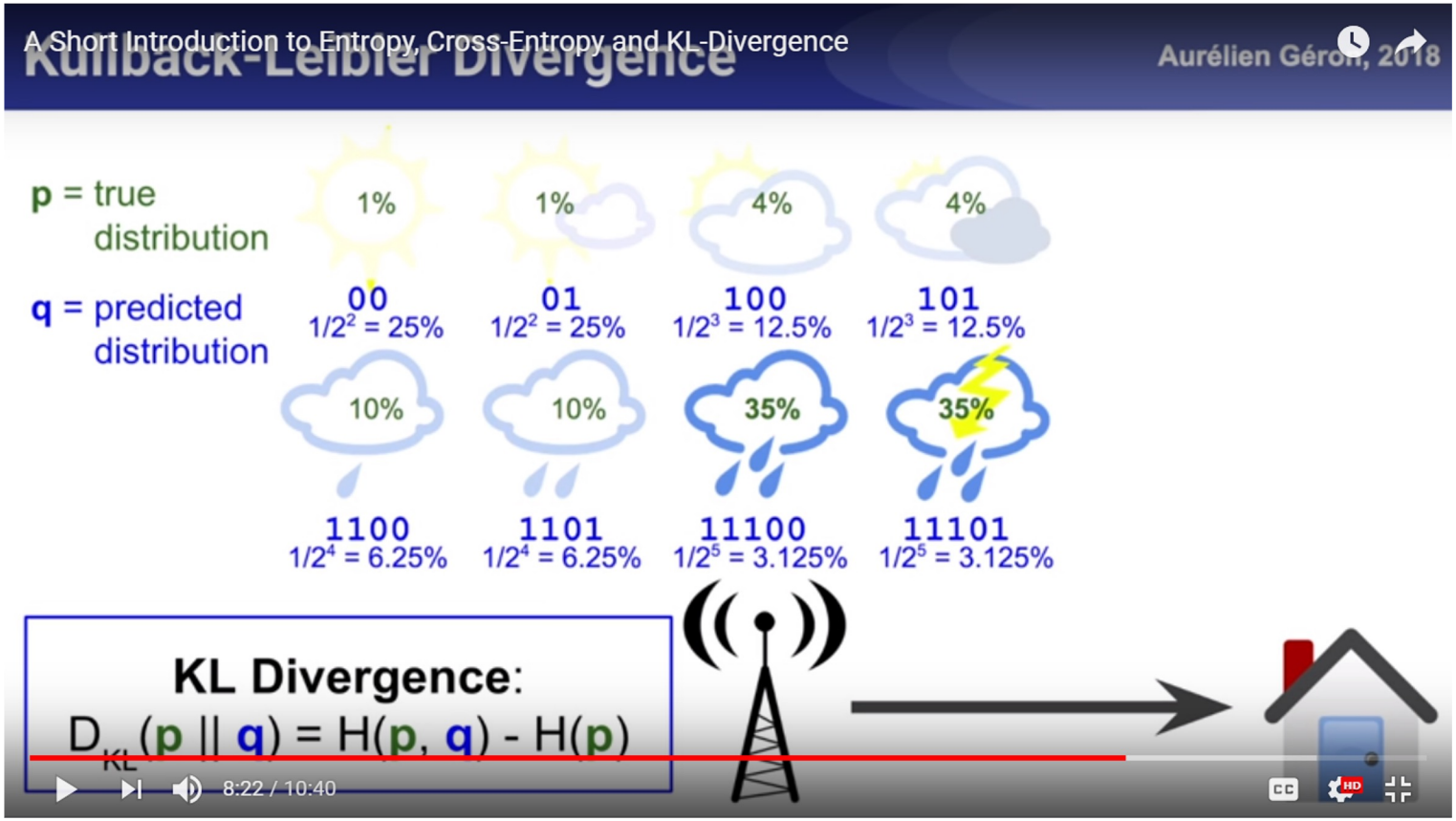
Same as log likelihood of logistic regression
Negative in front because we are minimizing the loss vs maximizing the probability

$$p(y \mid x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}$$

Other views of CE loss

Relationship between Entropy, CE, and KL Divergence

https://www.youtube.com/watch?v=ErfnhcEV1O8



Notes on CE loss

- If classes is ordered, it might not be ideal
  - Class 0: Perfect, Class 1: Good, Class 2: Average, Class 3: Bad
  - Soln: use Squared EMD loss https://arxiv.org/abs/1611.05916

- You can have better loss but worse accuracy
  - Groundtruth [1, 0, 0].
    - Model A [0.4, 0.3, 0.3], Model B [0.45, 0.5, 0.05]
  - Soln: monitor accuracy as well as loss

- Leads to overconfidence
  - Soln: label smoothing, https://paperswithcode.com/method/label-smoothing, Calibration https://arxiv.org/abs/1706.04599

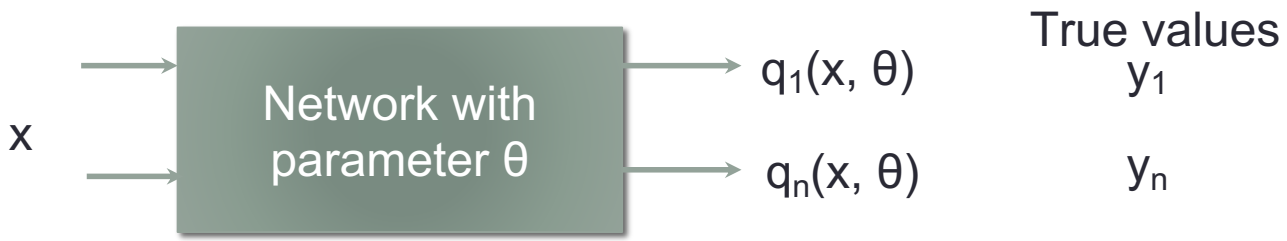
Sum of squared errors (MSE)

- Used for any real valued outputs such as regression

$$L = \frac{1}{2}\Sigma_n(y_n - q_n(x,\theta))^2$$

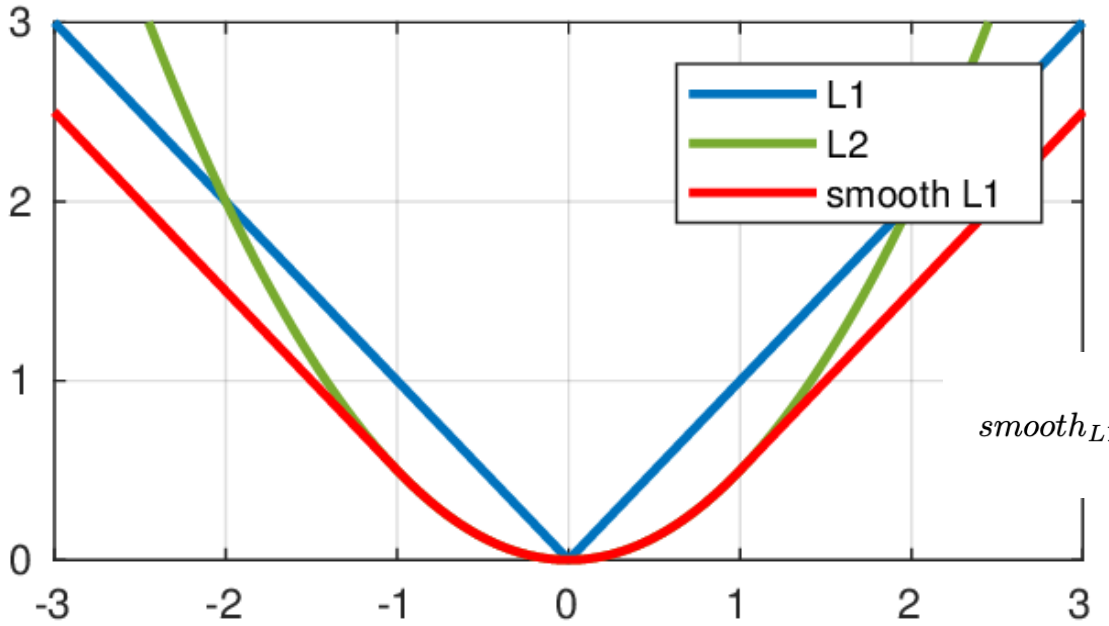
n is output dimension index
Note Eq is calculated on one sample
Summed over whole batch for total loss

- Nonnegative, the better the lower the loss



Notes on MSE loss

- L1 vs L2 loss
  - L1 robust to outliers
  - L2 easier to optimize (smooth gradient)
- Smoothed L1 combination of L1 and L2

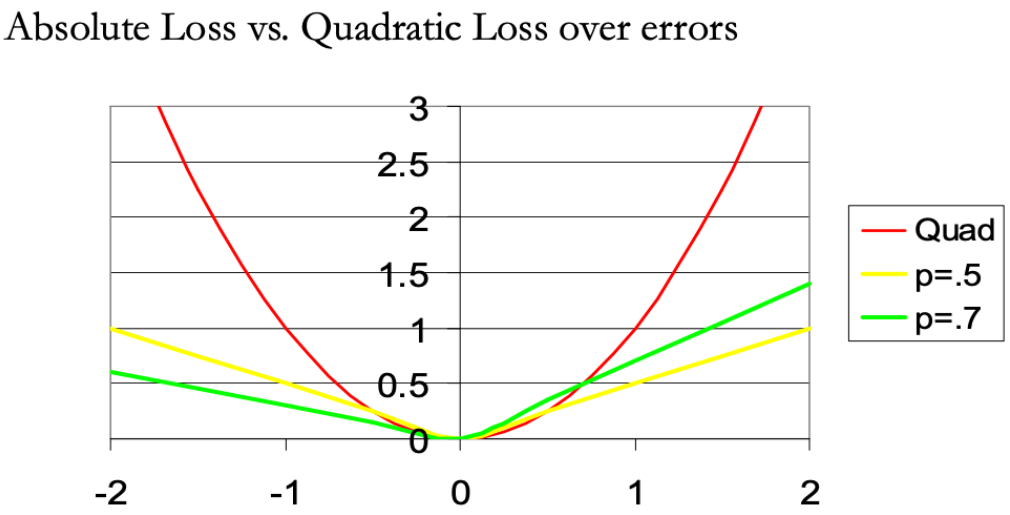


$$smooth_{L1}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases}, \qquad (4)$$

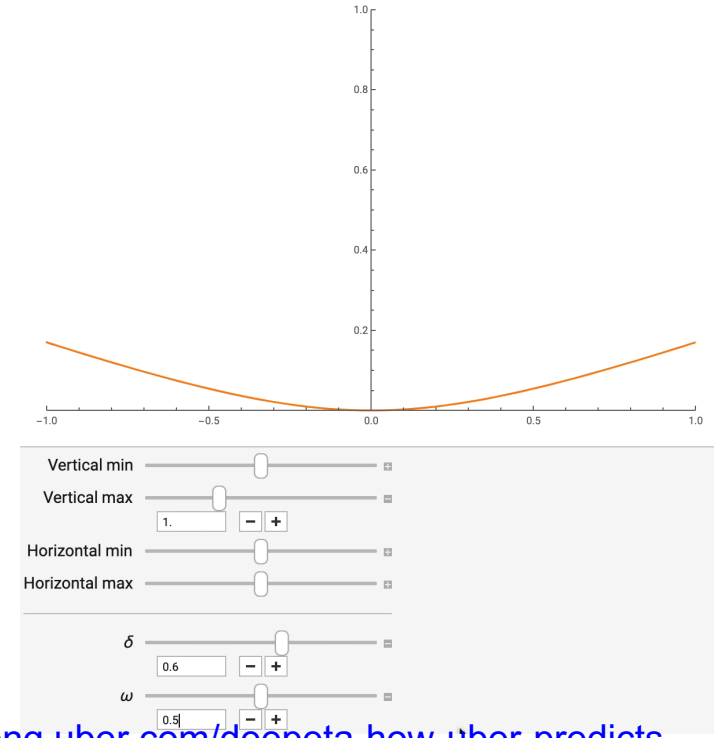
https://www.researchgate.net/publication/321180616_Wing_Loss_for_Robust_Facial_Landmark_Localisation_with_Convolutional_Neural_Networks

Notes on MSE loss

- Loss weights underestimate and overestimate to be the same
  Soln: asymetric loss (Quantile loss – L1, Huber loss – smoothed L1)



https://www.bauer.uh.edu/rsusmel/phd/ec1-25.pdf



https://eng.uber.com/deepeta-how-uber-predicts-arrival-times/

Regularization

Regularization: trading model bias for model variance

There two main approaches to regularize neural networks

- Explicit regularization
  Deals with the loss function
- Implicit regularization
  Deals with the network

Famous types of regularization

- L1 regularization: Regularizing term is a sum

$$\frac{1}{2}\Sigma_{i=1}^{m}(y_i - \theta^T \mathbf{x_i})^2 \; + \Sigma_j \mid \theta_j \mid$$

- L2 regularization: Regularizing term is a sum of squares

$$\frac{1}{2}\Sigma_{i=1}^{m}(y_i - \theta^T \mathbf{x_i})^2 \; + \Sigma_j \mid \theta_j \mid^2$$

(Not to be confused with L1 and L2 losses)

http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/

L1 L2 regularization notes

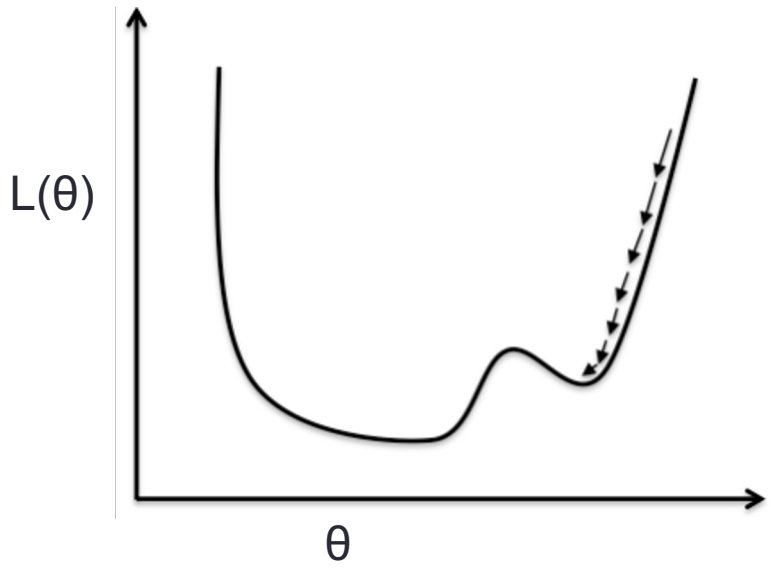
- Can use both at the same time
  - People claim L2 is superior
  - Weight decay can be considered as L2 regularization (under simple optimization techniques)
    - https://arxiv.org/abs/1711.05101
- I found them useless in practice for deep neural networks
  - Works when data is small (transfer learning)
- Other regularization methods exist (we will go over these later)

Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function and regularization
  - SGD and backprop
  - Learning rate
  - Overfitting – dropout, batchnorm
- CNN, RNN, LSTM, GRU

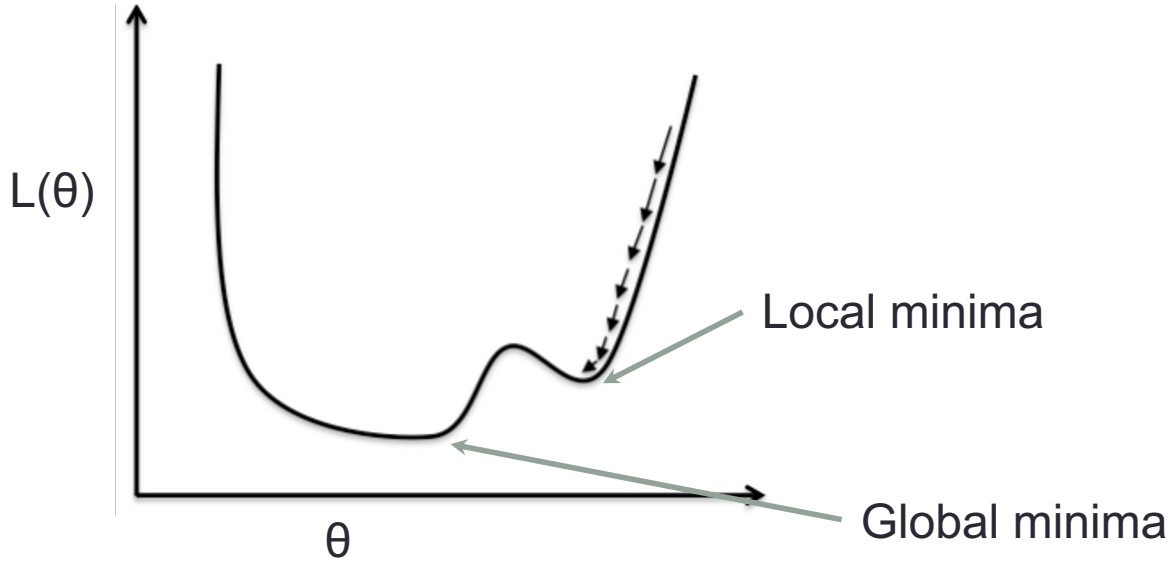
Minimization using gradient descent

- We want to minimize L with respect to θ (weights and biases)
  - Differentiate with respect to θ
  - Gradients passes through the network by Back Propagation



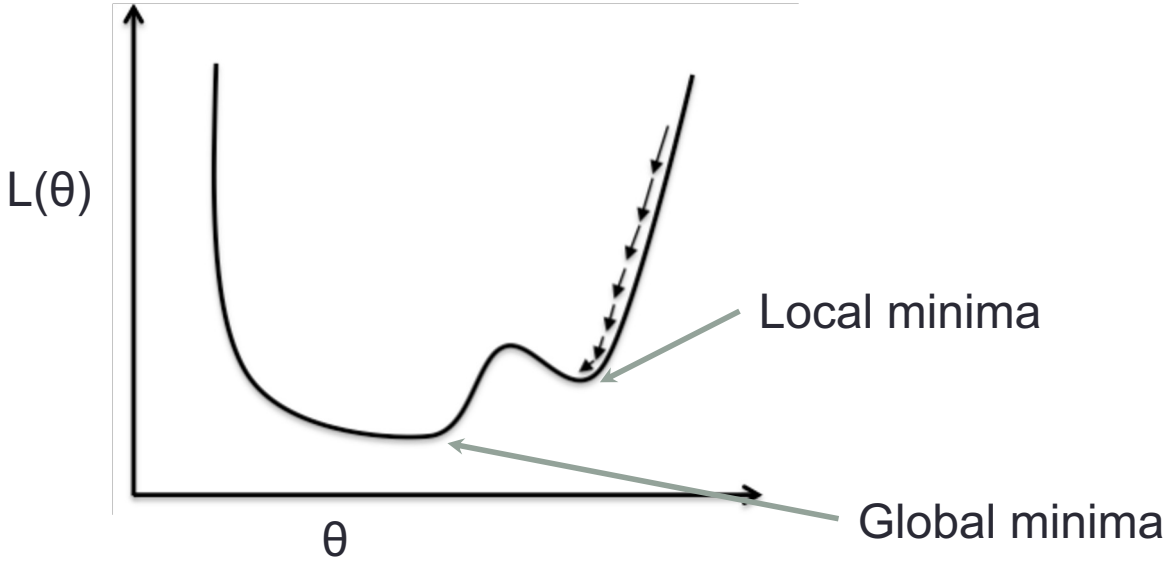
Deep vs Shallow

- The loss function of neural network is non-convex (and non-concave)
  - Local minimas can be avoided with convexity
    - Linear regression, SVM are convex optimization
  - Convexity gives easier training
    - Does not imply anything about the generalization of the model
    - The loss is optimized by the training set



Deep vs Shallow

- If deep, most local minimas are the global minima!
  - Always a way to lower the loss in the network with millions of parameters
  - Enough parameters to remember every training examples
  - Does not imply anything about generalization



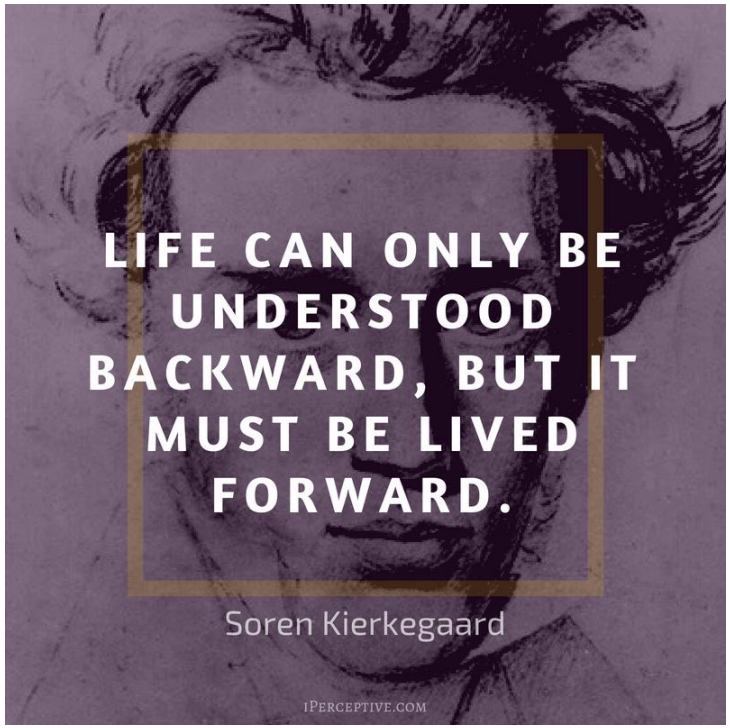
Kenji Kawaguchi,Deep Learning without Poor Local Minima, 2016 https://arxiv.org/abs/1605.07110

Differentiating a neural network model

- We want to minimize loss by gradient descent
- A model is very complex and have many layers! How do we differentiate this!!?

Back propagation

- Forward pass
  - Pass the value of the input until the end of the network
- Backward pass
  - Compute the gradient starting from the end and passing down gradients using chain rule



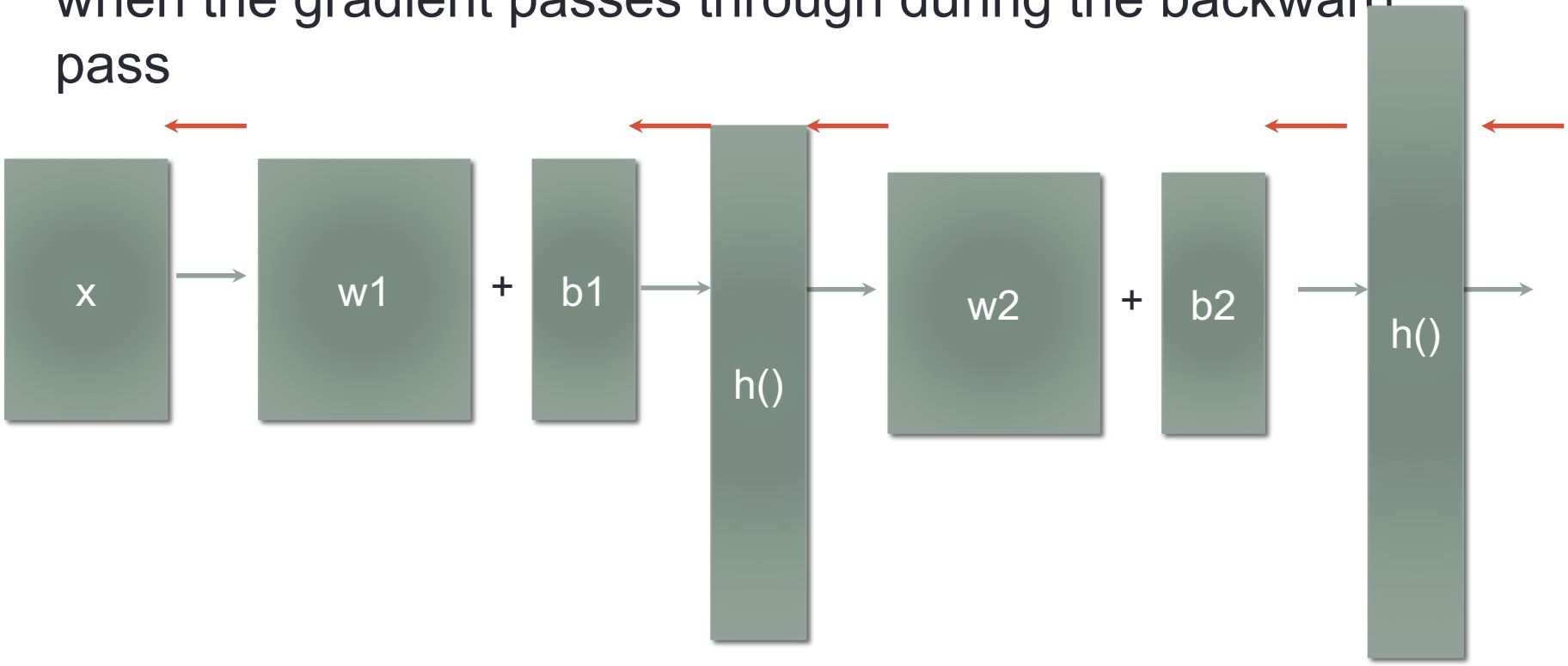
Examples to read

https://alonalj.github.io/2016/12/10/What-is-Backpropagation/

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Backprop and computation graph

- We can also define what happens to a computing graph when the gradient passes through during the backward pass

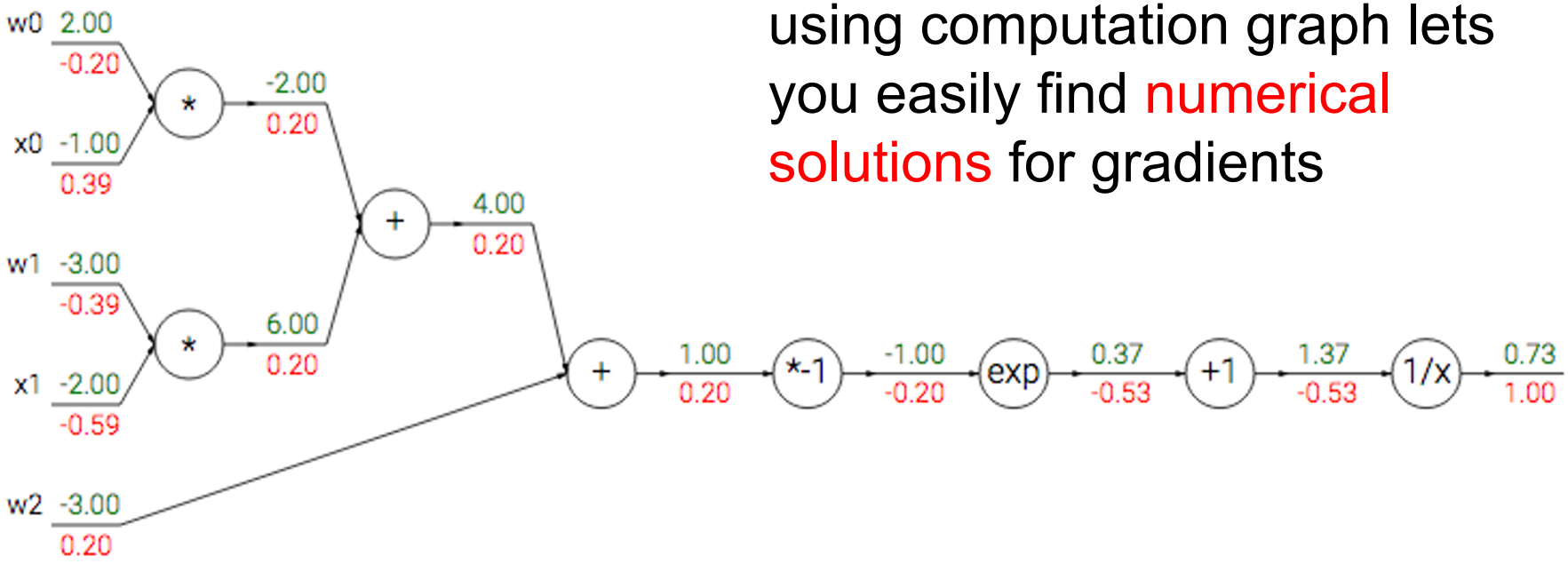


This lets us to build any neural networks without having to redo all the derivation as long as we define a forward and backward computation for the block.

Numerical gradient flow

- Let's find the gradient of $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

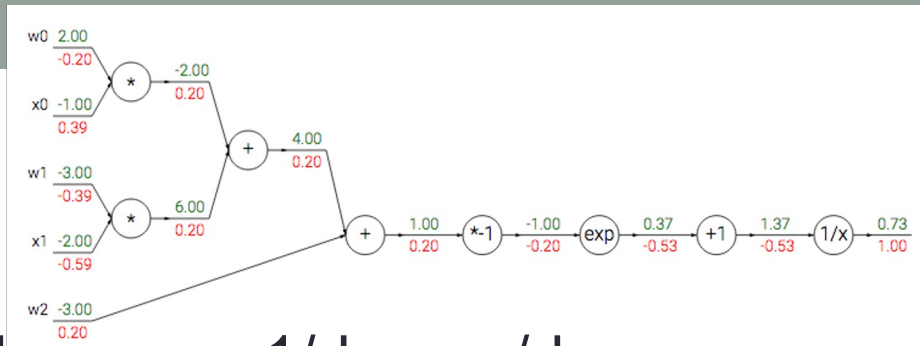
Computation graph



Doing backprop (chain rule) by using computation graph lets you easily find numerical solutions for gradients

http://cs231n.github.io/optimization-2/

$$f(w,x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

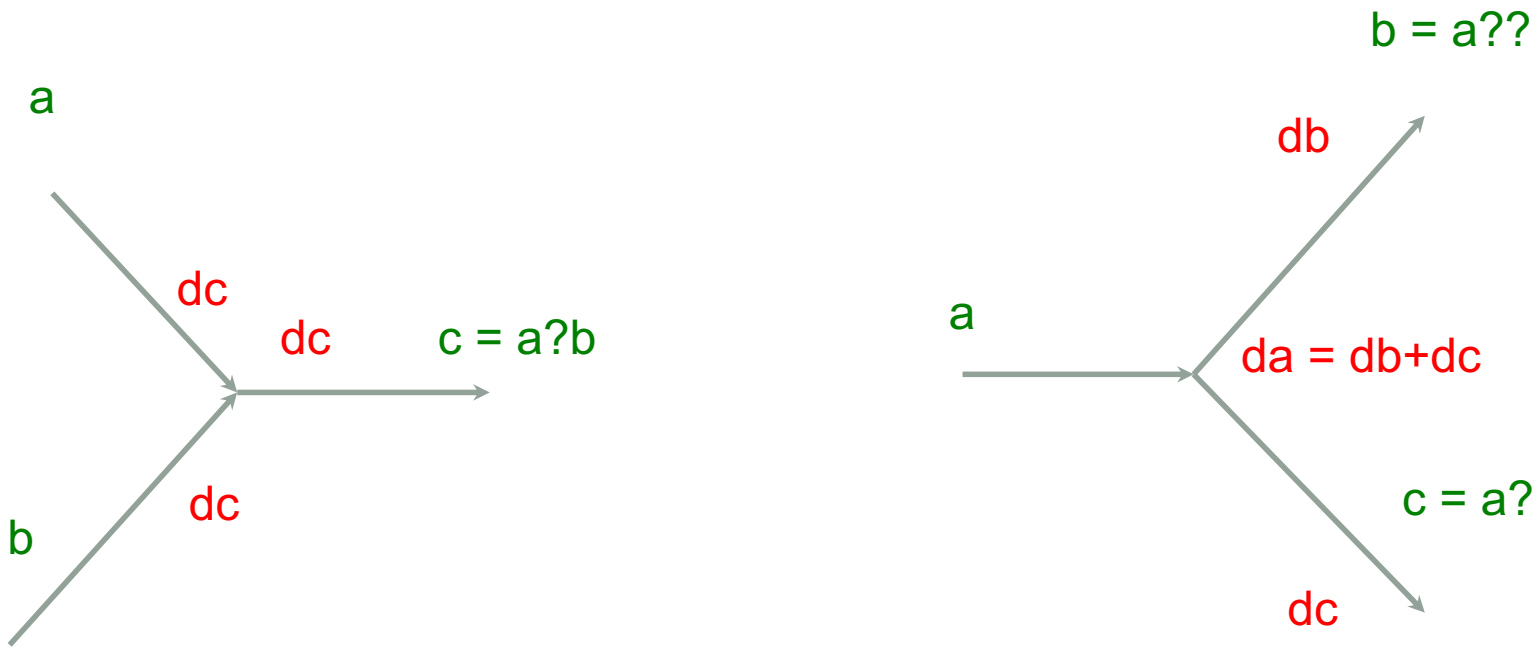


- w = [0, -3, -3]
- x = [-1, -2]
- t_0 = w[0]*x[0]
- t_1 = w[1]*x[1]
- $t_{01} = t_0 + t_1$
- $t_{012} = t_{01} + w[2]$
- $n_t = -t_{012}$
- e = exp(n_t)
- denom = e+1
- f = 1/denom

- ddenom = -1/denom/denom
- de = 1*ddenom
- dn_t = exp(n_t)*de
- $dt_{012} = -dn_t$
- $dw_2 = 1*dt_{012}$
- $dt_{01} = 1*dt_{012}$
- $dt_0 = 1*dt_{01}$
- $dt_1 = 1*dt_{01}$
- $dw_1 = x[1]dt_1$
- $dx_1 = w[1]dt_1$
- $dw_0 = x[0]dt_0$; $dx_0 = w[0]dt_0$

Perform backward pass in reverse order. No need to explicitly find overall derivative
This technique is called autograd. Many frameworks use it today.

Gradient flow at forks



Forward and backward pass acts differently at forks

Gradient and non-linearities

We can now talk about how good a non-linearity is by looking at the gradients.

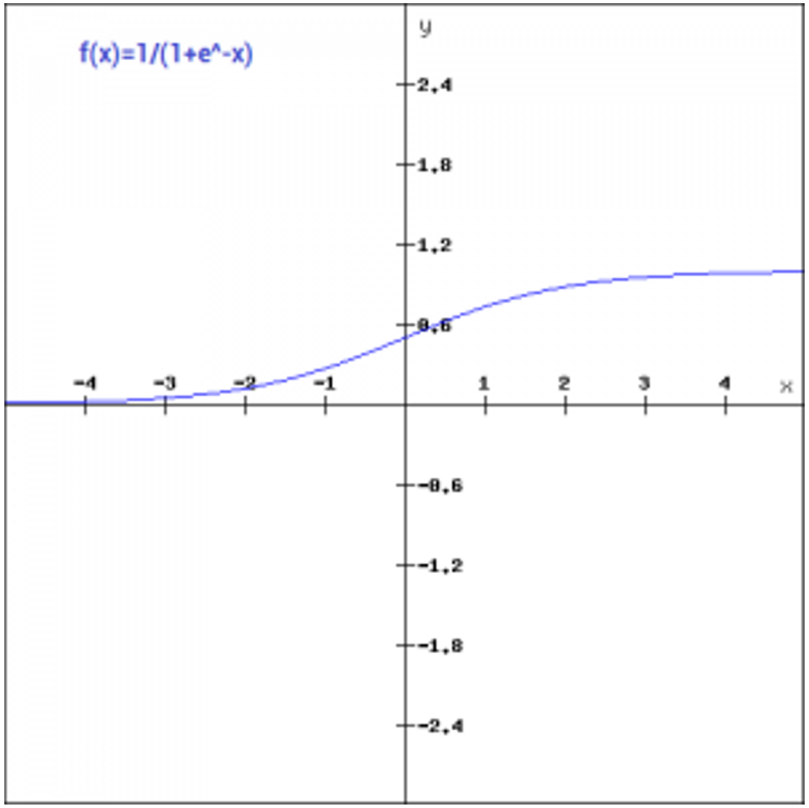
We want

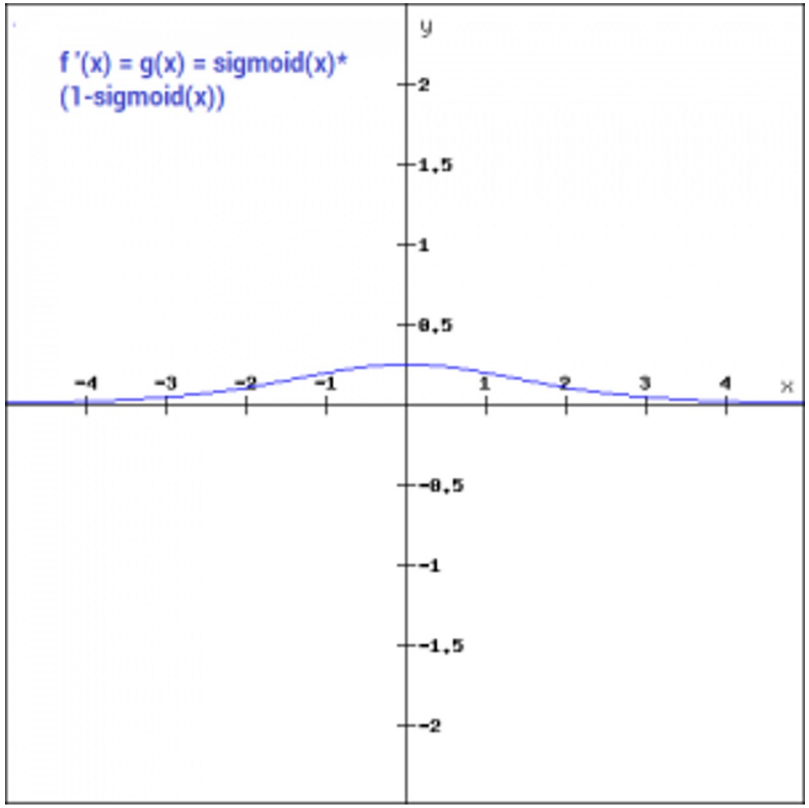
- Something that is differentiable numerically
- Cheap to compute
- Big gradients at every point

Notes on non-linearity

- Sigmoid

  Models get stuck if fall go far away from 0. Output always positive



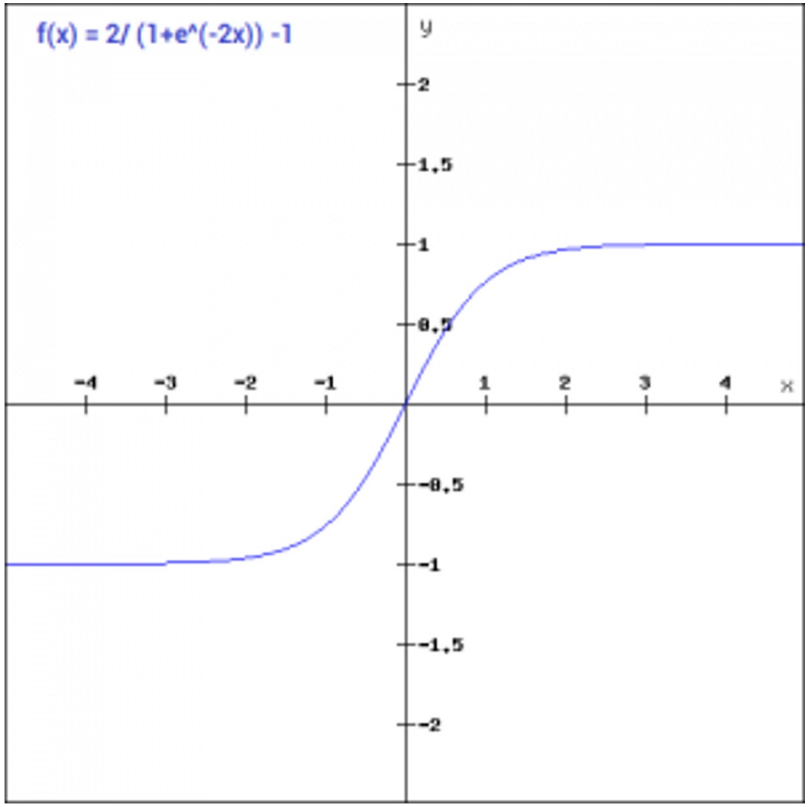


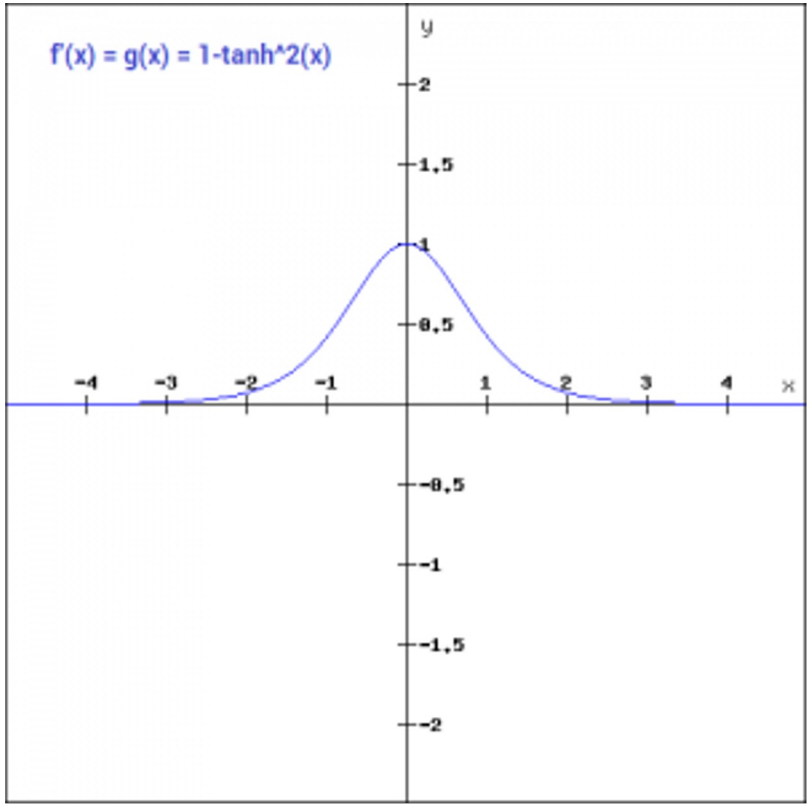
https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/

Notes on non-linearity

- Tanh

Output can be +-. Models get stuck if far away from 0



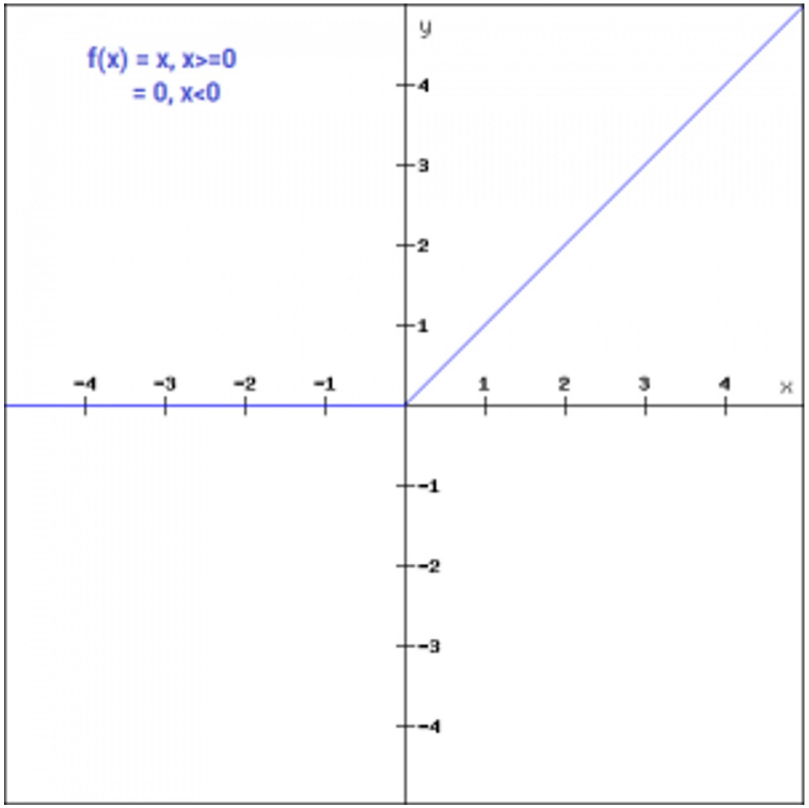


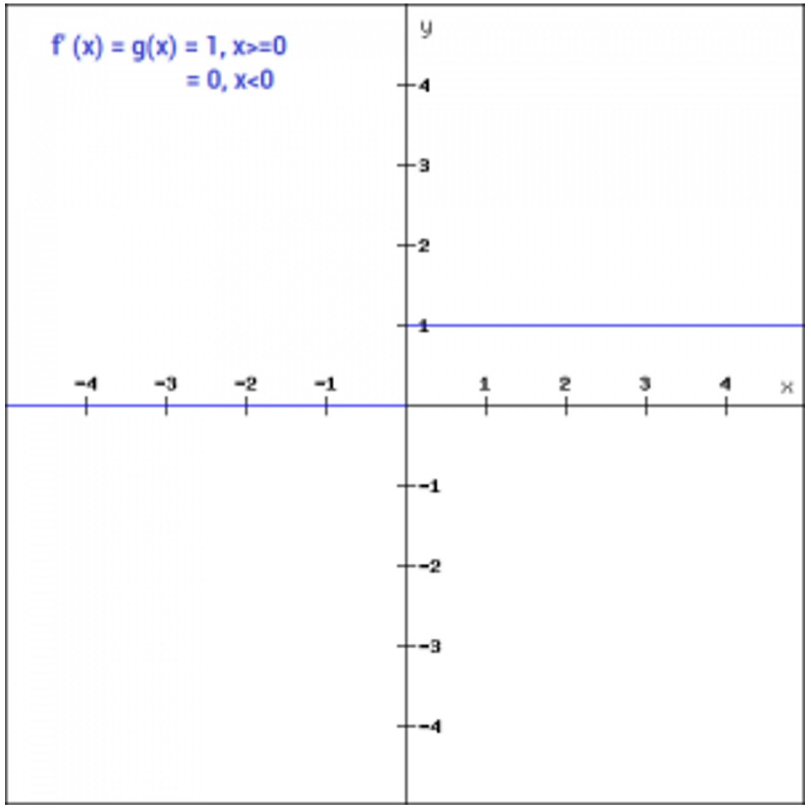
https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/

Notes on non-linearity

- ReLU

  High gradient in positive. Fast compute. Gradient doesn't move in negative



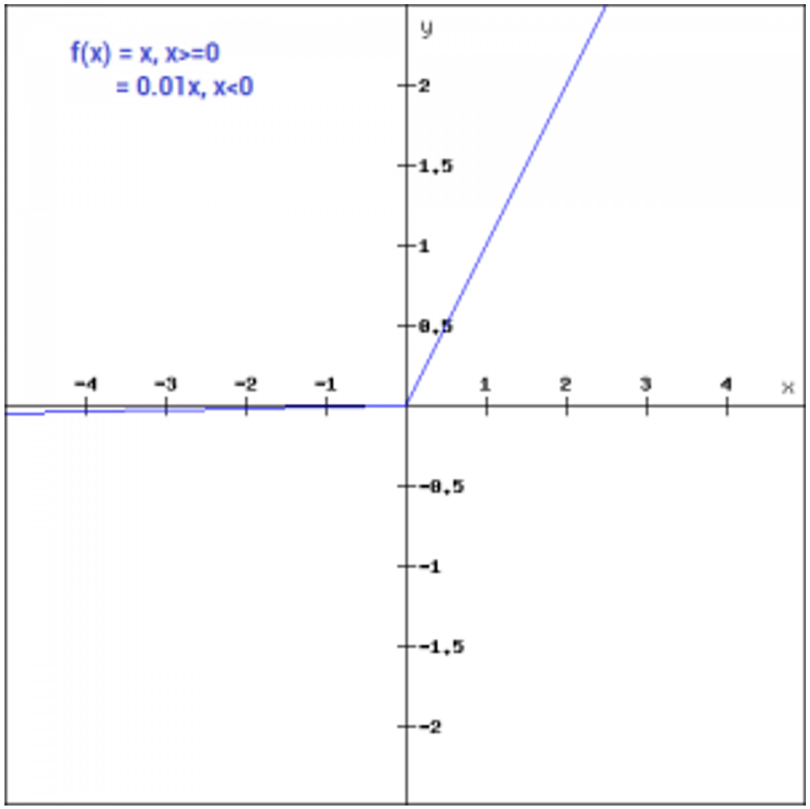


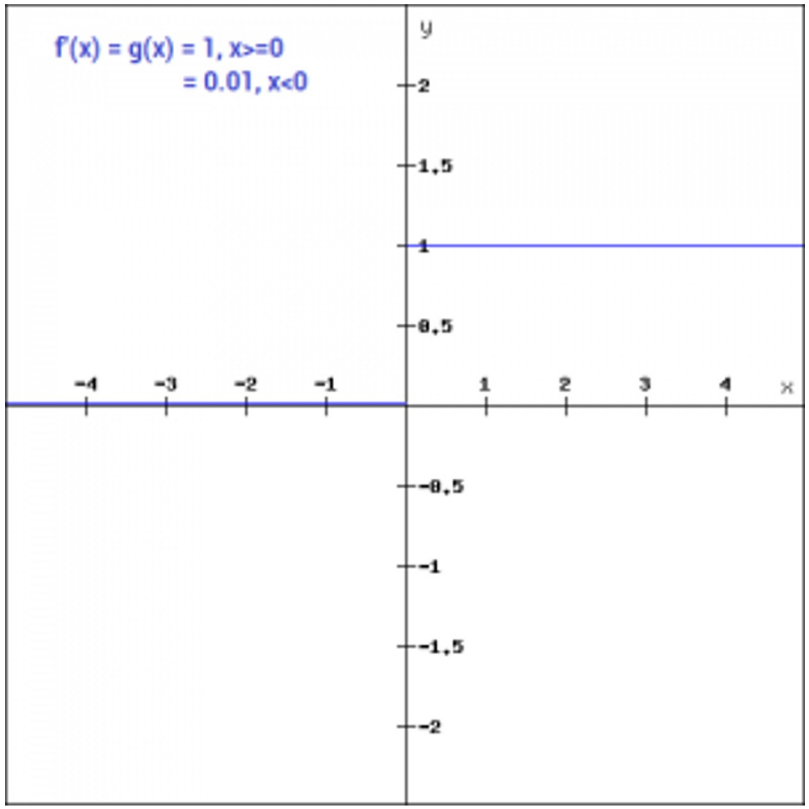
https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/

Notes on non-linearity

- Leaky ReLU

  Negative part now have some gradient. Small improvements depending on tasks





https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/

Notes on non-linearity

- Leaky ReLU
  Fixed slope
- PreLU
  Slope is learnable
  Different layer can have difference slope

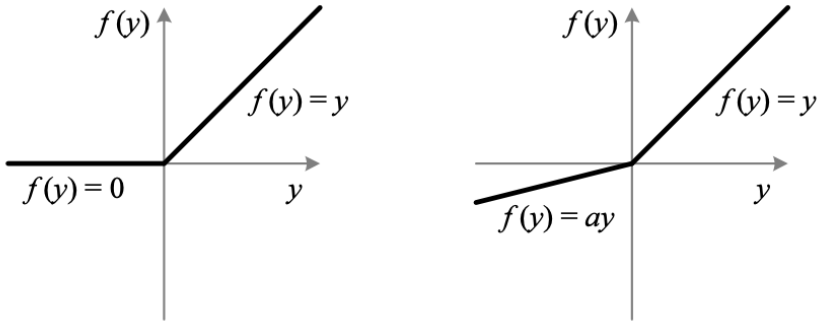


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

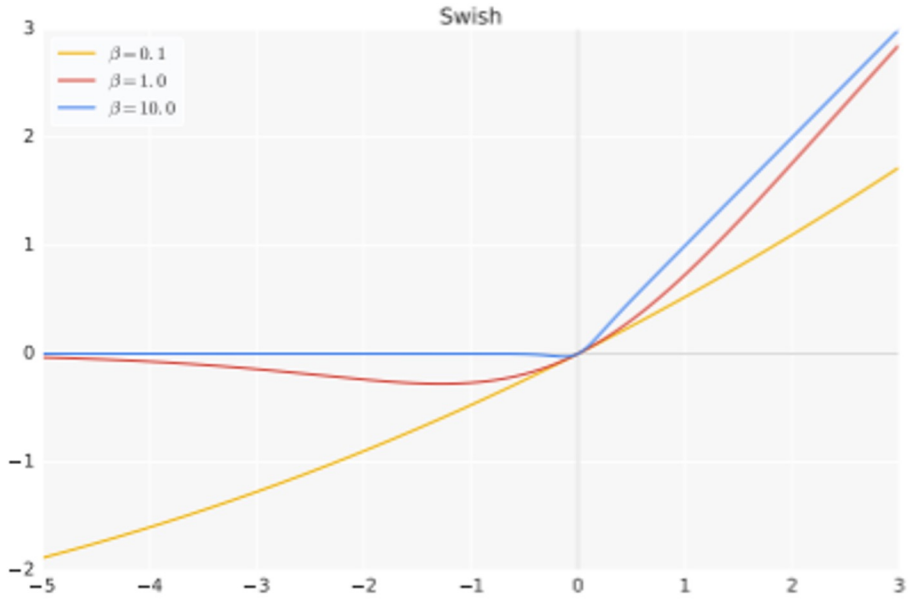
https://paperswithcode.com/paper/delving-deep-into-rectifiers-surpassing-human

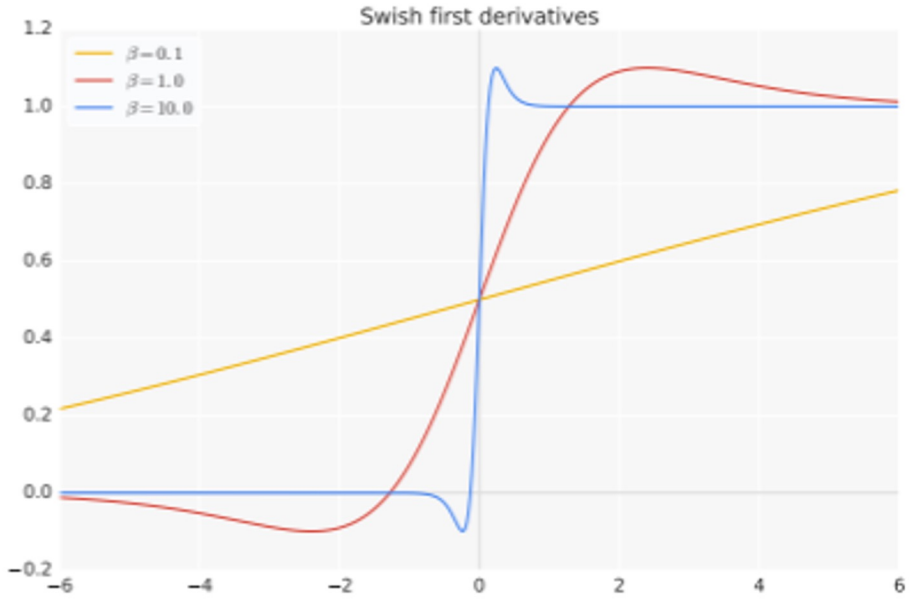
Notes on non-linearity

$$x \cdot sig(\beta x)$$

- Swish

  Nonnegative everywhere. Not monotonic. Learnable Beta or set to 1

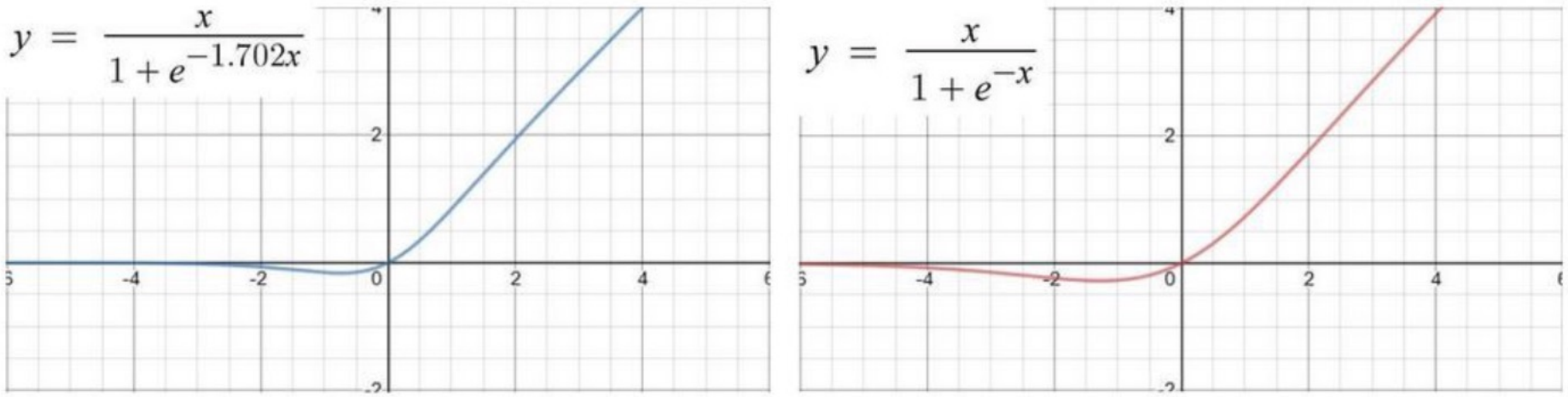




[Searching for Activation Functions - arXiv]

Notes on non-linearity

- Swish
  Nonnegative everywhere. Not monotonic. Learnable Beta or set to 1
- Gaussian Error Linear Units (GELU)
  Set beta to 1.702 (approximation of Gaussian distribution)
  Trains faster somehow, might be due to initialization strategy(?)



GELU (left) vs. Swish-1 (right): Image by Author

https://towardsdatascience.com/on-the-disparity-between-swish-and-gelu-1ddde902d64b

Notes on non-linearity

- Start with ReLU
  - No parameter to tune, many papers/settings uses this so it might work better with the matched settings
  - Try LeakyReLU or GELU afterwards

Initialization

- The starting point of your descent
- Important due to local minimas
- Not as important with large networks AND big data
- Now usually initialized randomly
  - One strategy (Xavier init)

$$\text{var}(w) = 2/(\text{fan\_in} + \text{fan\_out})$$

  - For ReLUs (He init)

$$\text{var}(w) = 2/(\text{fan\_in})$$

  - Or use a pre-trained network as initialization

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010
https://proceedings.mlr.press/v9/glorot10a.html
Kaiming He, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015
https://arxiv.org/abs/1502.01852

Stochastic gradient descent (SGD)

- Consider you have one million training examples
  - Gradient descent computes the objective function of all samples, then decide direction of descent
    - Takes too long
  - SGD computes the objective function on subsets of samples
    - The subset should not be biased and properly randomized to ensure no correlation between samples
- The subset is called a mini-batch
- Size of the mini-batch determines the training speed and accuracy
  - Usually somewhere between 32-1024 samples per mini-batch
    - Generally, works well with default settings
- Definition: 1 batch vs 1 epoch

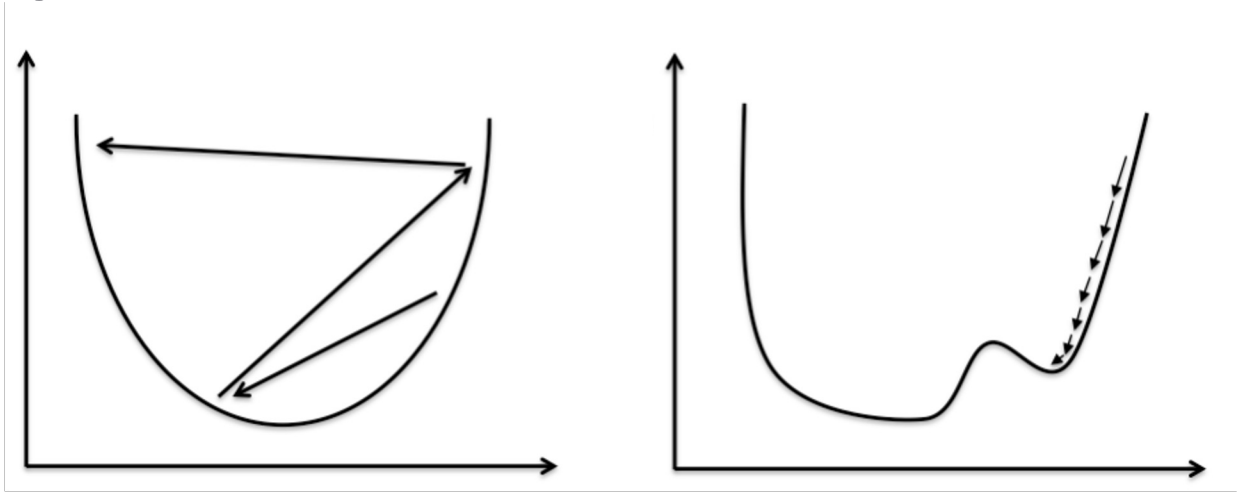
Self regularizing property of SGD

- SGD by its randomized nature does not overfit (as fast)
  - Considered as an implicit regularization (no change in the loss)

https://cbmm.mit.edu/sites/default/files/publications/CBMM-Memo-067-v3.pdf

Learning rate

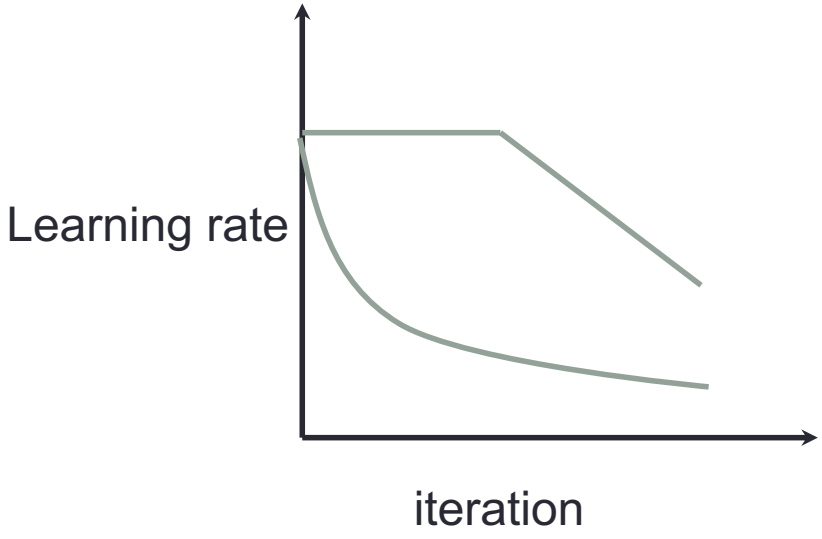
- How fast to go along the gradient direction is controlled by the learning rate
- Too large models diverge
- Too small the model get stuck in local minimas and takes too long to train



http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

Learning rate scheduling

- Usually starts with a large learning rate then gets smaller later
- Depends on your task
- Automatic ways to adjust the learning rate : Adagrad, Adam, etc. (still need scheduling still)



Learning rate strategies (annealing)

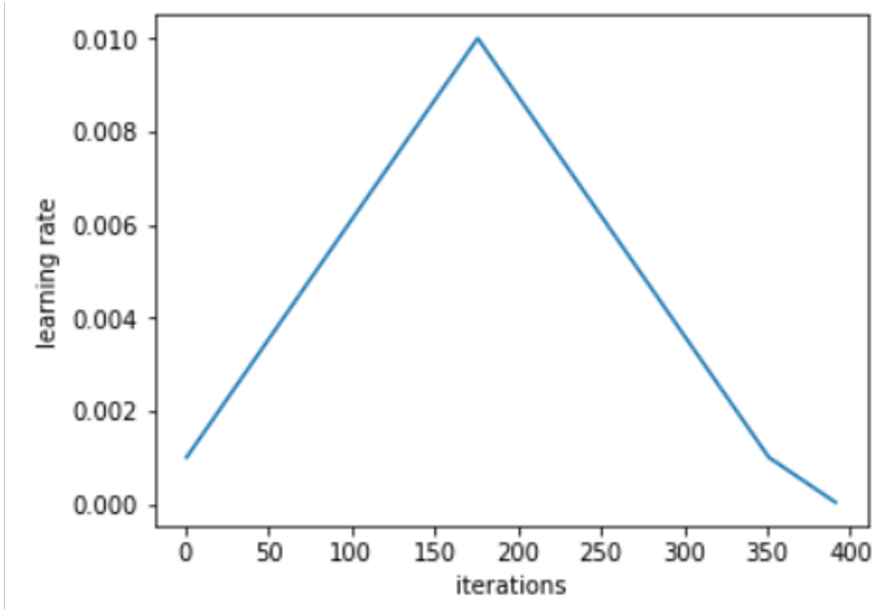
- Step decay: reduce learning rate by x after y epochs
- New bob method: half learning rate every time the validation error goes up. Only plausible in larger tasks
- Exponential decay: multiplies the learning rate by exp(-rate*epoch number)

Learning rate warm up

Initial point of the network can be at a bad spot.

Try not to go to fast - has a warmup period.

Useful for large datasets, or adaption (transfer learning)



Potentially leads to faster convergence and better accuracy – see paper below on why

See links below for methods to select the shape of the triangle

https://sgugger.github.io/the-1cycle-policy.html#the-1cycle-policy

Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Cyclical Learning Rates for Training Neural Networks

https://arxiv.org/abs/1908.03265 On the Variance of the Adaptive Learning Rate and Beyond

Notes on SGD

- SGD minimizes the loss function using just the first order derivative that is

$$L(\theta + \alpha\Delta\theta) = L(\theta) + \alpha L'(\theta)\Delta\theta + \frac{1}{2!}\alpha^2 L''(\theta)(\Delta\theta)^2 + \ldots$$

is approximate as

$$L(\theta + \alpha\Delta\theta) \simeq L(\theta) + \alpha L'(\theta)\Delta\theta$$

$$\nabla^2 f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f}{\partial\theta_1\partial\theta_1} & \cdots & \frac{\partial^2 f}{\partial\theta_1\partial\theta_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial\theta_n\partial\theta_1} & \cdots & \frac{\partial^2 f}{\partial\theta_n\partial\theta_n} \end{bmatrix}$$

because computing the second order gradient is very expensive.

Newer techniques use tricks to try to compensate for this approximation

Momentum

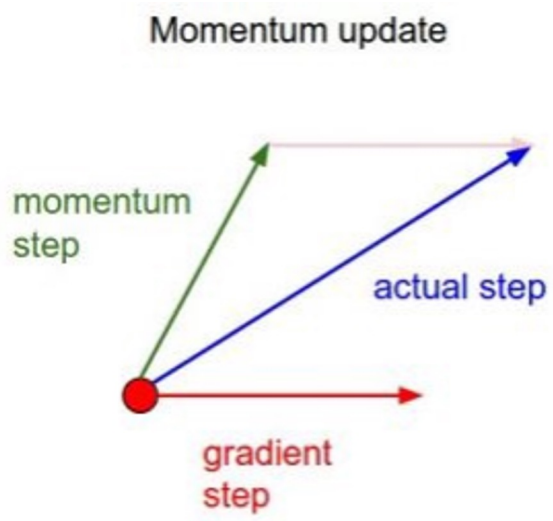
- Gradient descent can get stuck on small local minimas
  - Or slow down at saddle points
- Have concept of speed
- Can also consider as moving average of gradient estimate

speed Momentum rate gradient

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$

$$W = W - \alpha V_t$$

learning rate



http://cs231n.github.io/neural-networks-3/

Nesterov Momentum

Old

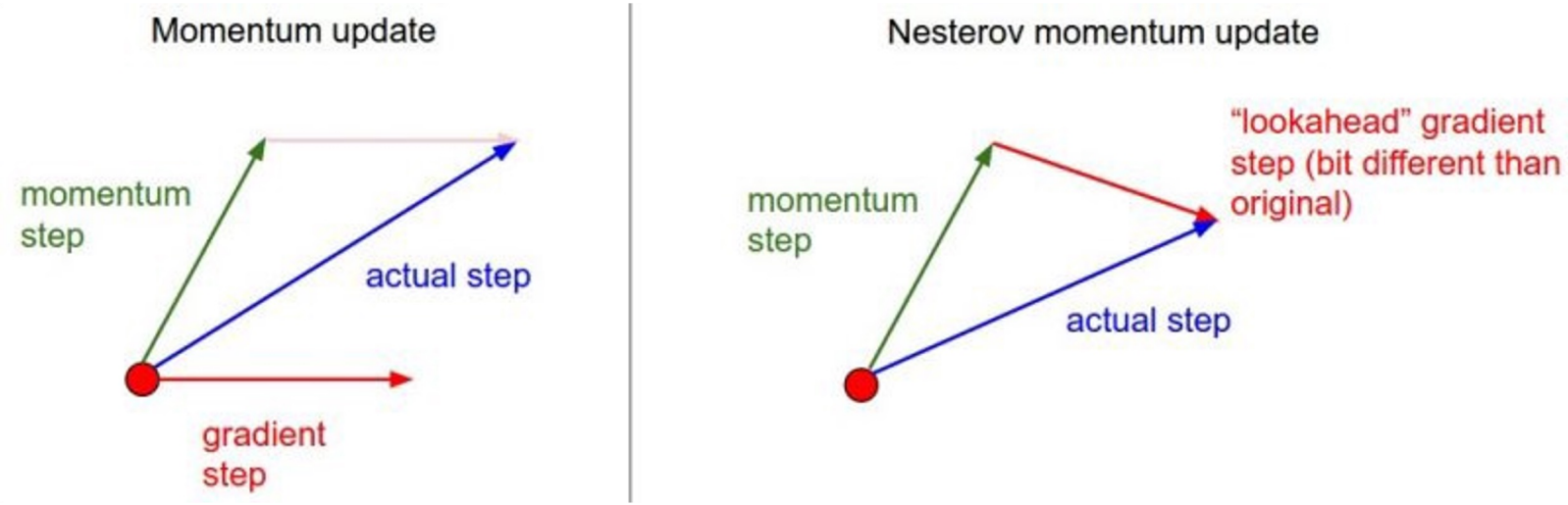
$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$
$$W = W - \alpha V_t$$

W is about to be at W-βV due to momentum

- Momentum with look ahead.
  - Compute gradient as if we took an additional step

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W - \beta V_{t-1}, X, y)$$
$$W = W - V_t$$

gradient is computed as if we took a step



http://cs231n.github.io/neural-networks-3/

Adaptive learning rates

How to have the updates be different for different layers?

LR of each weight is scaled by the size of the gradient

Can we trust the moment estimates?

Decay running estimates of the gradient size/momentums

RMSProp (normalized by gradient + decay)

Adam (normalized by gradient + decay both grad and momentum)

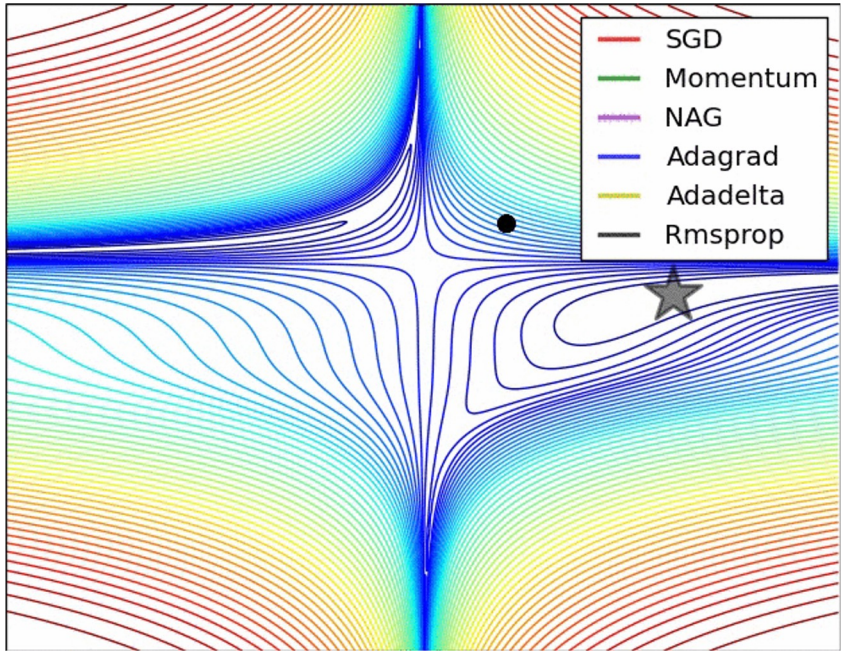
AdamW (Adam but deal with weight decay differently)

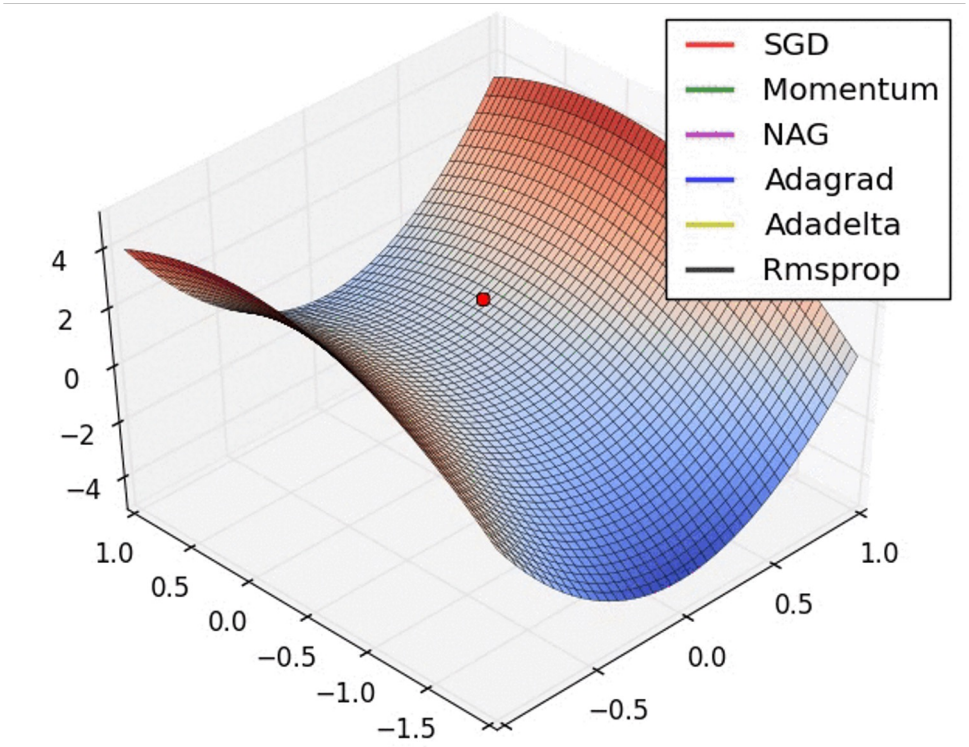
You still need to do learning rate scheduling

More details see

https://www.ruder.io/optimizing-gradient-descent/
https://www.ruder.io/deep-learning-optimization-2017/
https://towardsdatascience.com/why-adamw-matters-736223f31b5d

Optimization method and speed





http://ruder.io/optimizing-gradient-descent/index.html#whichoptimizertochoose

Learning rate tricks

- At least decay the learning rate
  - Monitor validation set performance
- If the loss never goes down -> decrease the learning rate (by factor of 10)
- Start with ADAM or ADAMW. Also try RMSprop and SGD with Nesterov Momentum if you have time.

- Learning rate is the most important hyperparameter that will affect your model performance.

Learning rate and batch size

- Learning rate and batch size interacts with each other
  - Larger batch has low variance -> can use larger learning rate
    - Increase batch size by k, increase learning rate by k (can also scale the momentum etc, see papers)
      - https://arxiv.org/abs/1706.02677
      - https://arxiv.org/abs/1711.00489

- Set batch size first
  - largest to fit the GPU (trains faster, and get better accuracy)
- Then set your learning rate
  - Still need to tune. Above theory is just a guideline

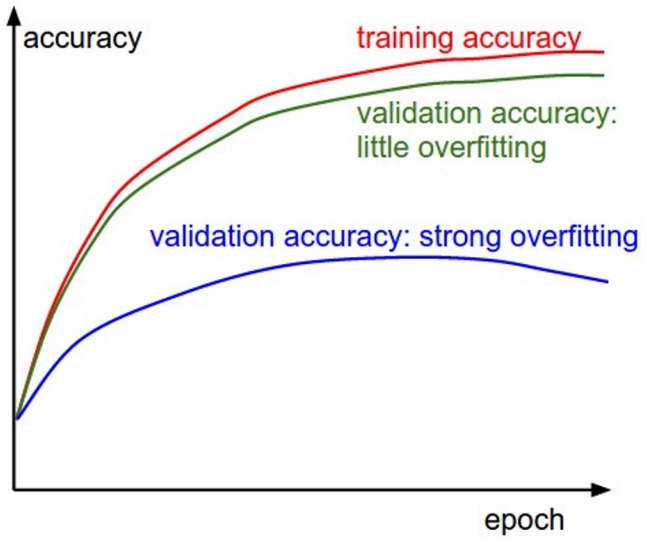
Small batchsize training

- Gradient accumulation
  - Accumulate gradients over multiple small batches before applying an parameter update
- Retune all learning rate hyperparamters

How to Train BERT with an Academic Budget https://arxiv.org/pdf/2104.07705.pdf

Overfitting

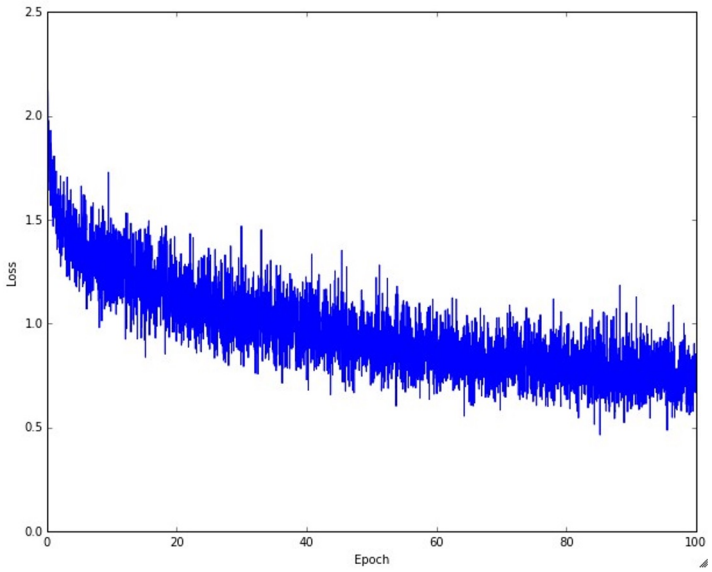
- You can keep doing back propagation forever!
- The training loss will always go down
- But it overfits
- Need to monitor performance on a held-out set
- Stop or decrease learning rate when overfit happens



http://cs231n.github.io/neural-networks-3/

Monitoring performance

- Monitor performance on a dev/validation set
  - This is NOT the test set
- Can monitor many criterions
  - Loss function
  - Classification accuracy
- Sometimes these disagree
- Actual performance can be noisy, need to see the trend



http://cs231n.github.io/neural-networks-3/

Dropout

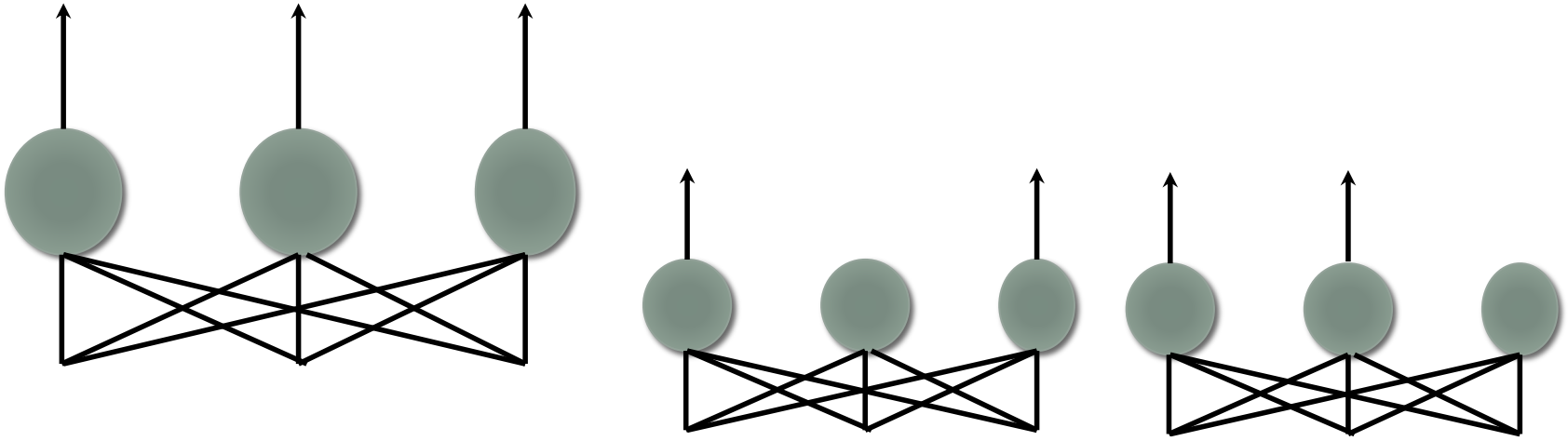
An implicit regularization technique for reducing overfitting

Randomly turn off different subset of neurons during training

Network no longer depend on any particular neuron

Force the model to have redundancy – robust to any corruption in input data

A form of performing model averaging (ensemble of experts)



Model

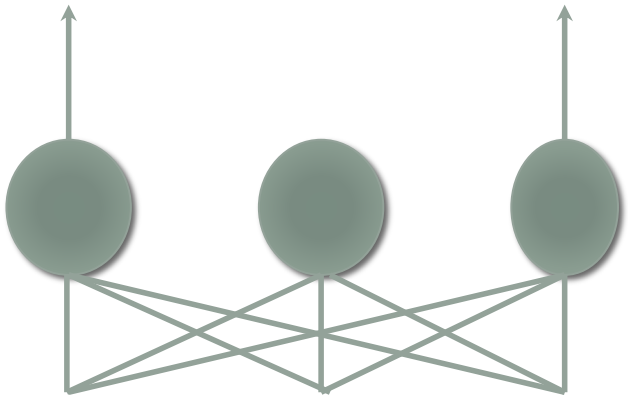
Dropout rate of 0.33

Hinton, Geoffrey "Improving neural networks by preventing co-adaptation of feature detectors" 2012

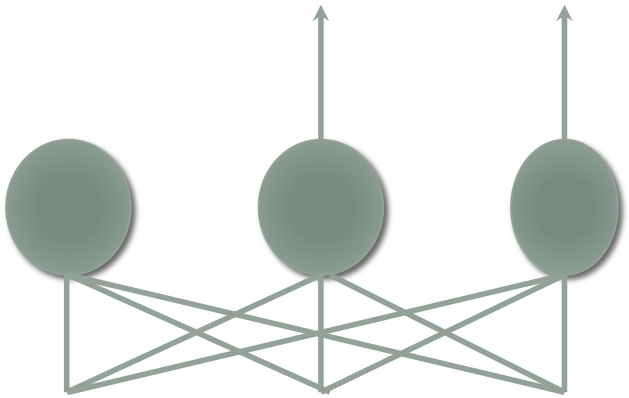
Dropout training time

Drop out rate 0.33

Randomly removed outputs
for each training sample



x1

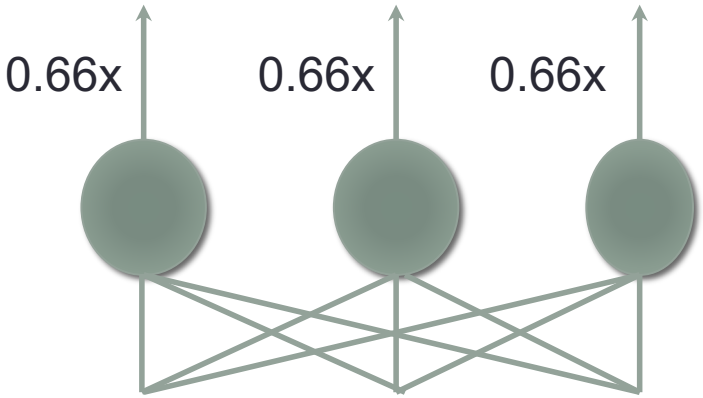


x2

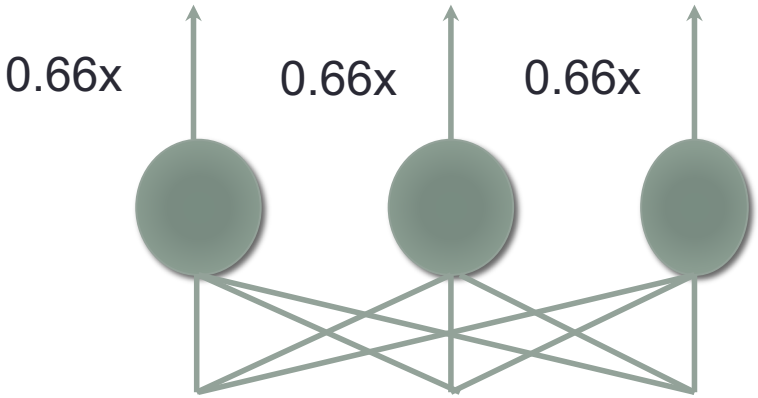
Dropout test time

Dropout rate 0.33

Scale outputs so the output contribution is around the same

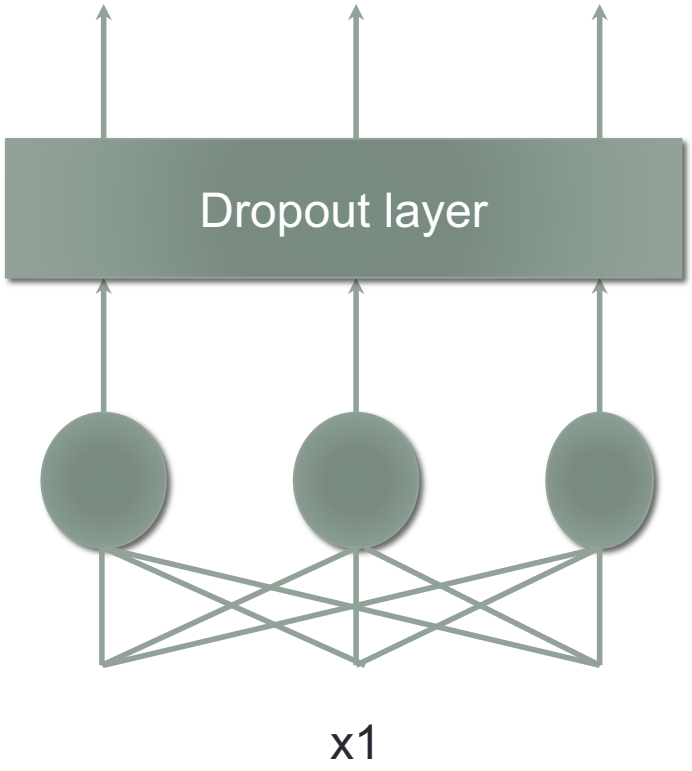


x1



x2

Dropout implementation

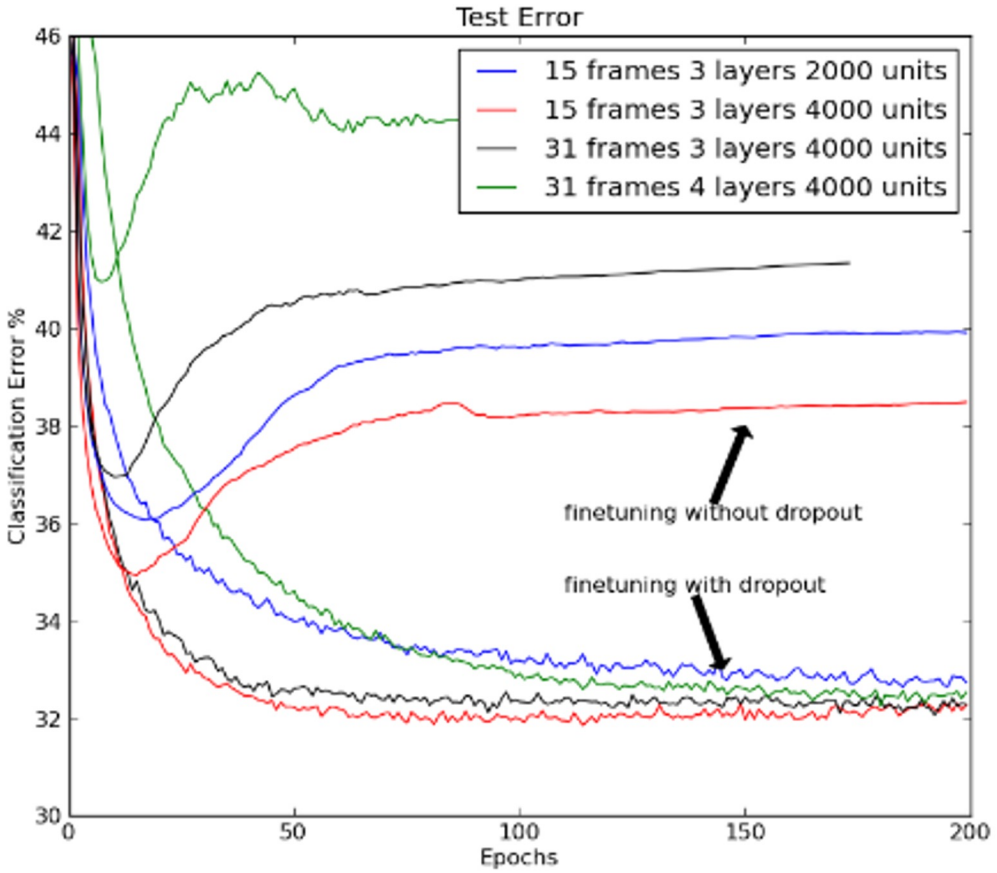


Dropout layer
Just another layer that drops inputs

Inverted dropout
A variant of dropout that scales the dropout at training time, so that you don't have to scale at test time.

Dropout on TIMIT

- A phoneme recognition task



Hinton, Geoffrey "Improving neural networks by preventing co-adaptation of feature detectors" 2012

Batch normalization

- Recent technique for (implicit) regularization
- Normalize every mini-batch at various batch norm layers to standard Gaussian (different from global normalization of the inputs)
- Place batch norm layers before non-linearities
- Faster training and better generalizations

For each mini-batch that goes through batch norm

1. Normalize by the mean and variance of the mini-batch for each dimension
2. Shift and scale by learnable parameters

Replaces dropout in some networks

$$\hat{x} = \frac{x - \mu_b}{\sigma_b}$$

$$y = \alpha\hat{x} + \beta$$

https://arxiv.org/abs/1502.03167

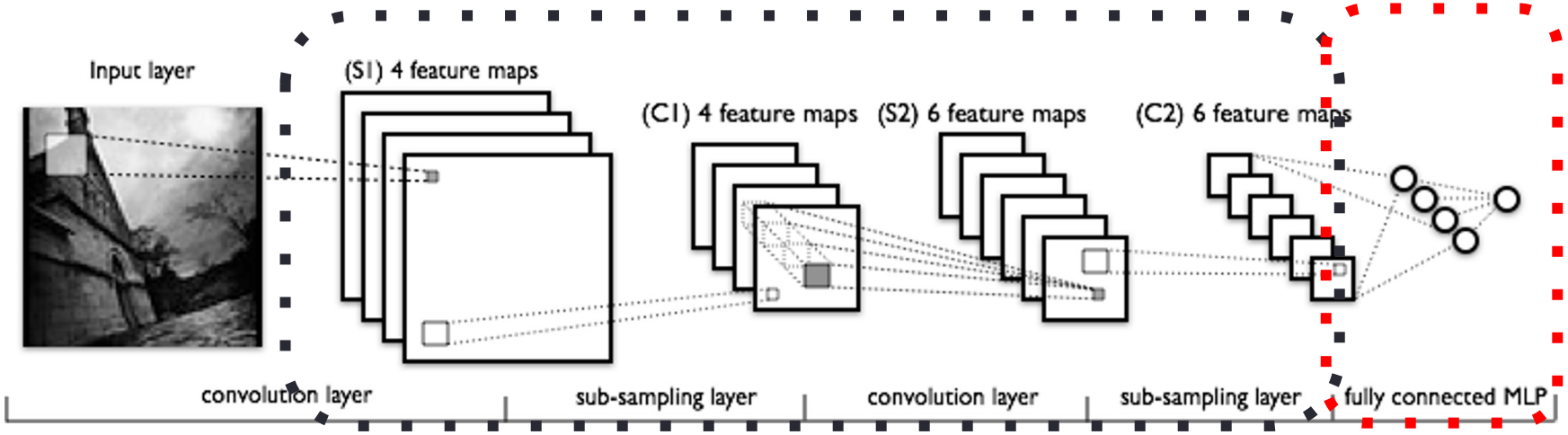
Tips to tune

- Feeling and experience ¯\(°_o)/¯
- Take numbers from papers
- Grid search
  - Heuristic search
  - Random search
  - Genetic Algorithm
- Picking the right type of model is more important than picking the right number of neurons
  - Inductive bias
- Tips for debugging (will make more sense next class)
  - http://karpathy.github.io/2019/04/25/recipe/

Neural networks

- Fully connected networks
  - Neuron
  - Non-linearity
  - Softmax layer
- DNN training
  - Loss function and regularization
  - SGD and backprop
  - Learning rate
  - Overfitting – dropout, batchnorm
- CNN, RNN, LSTM, GRU

Convolution Neural Network (CNN)

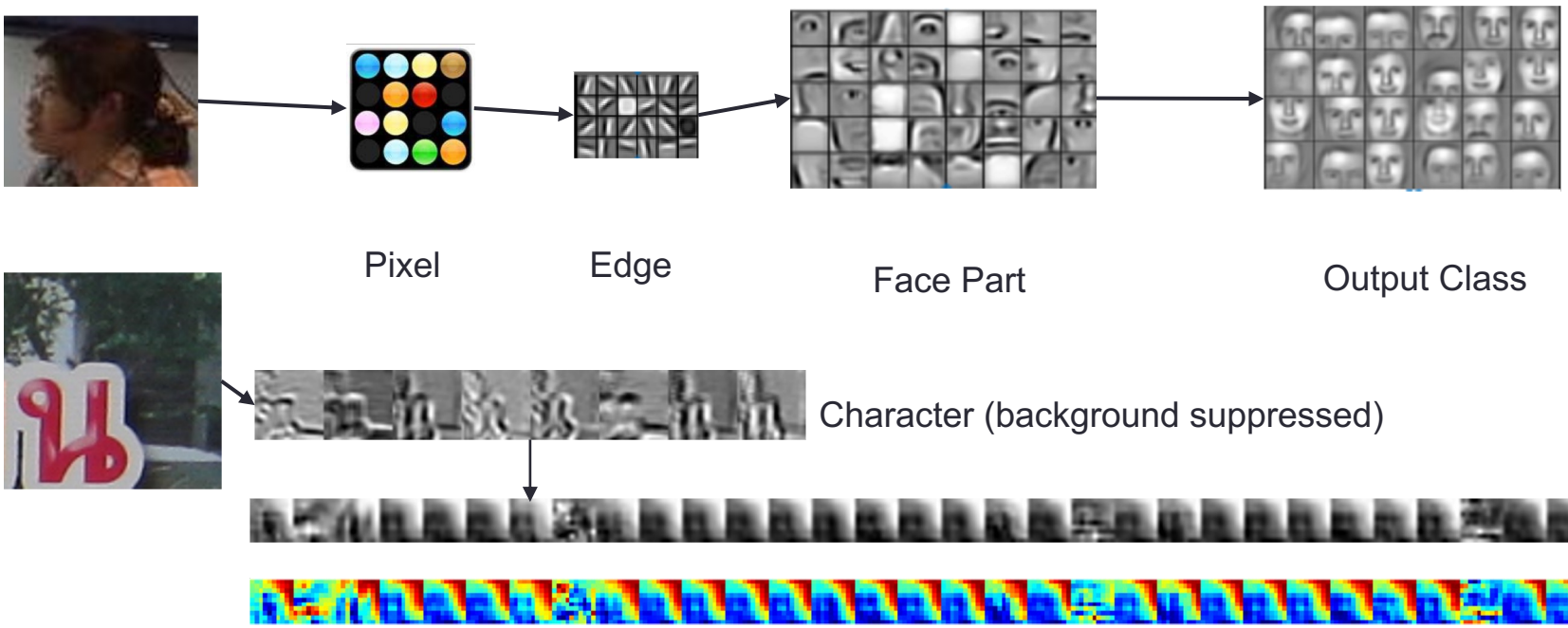


Trainable Feature Extractor Layer **Classification Layer**

Convolution layers followed by sub-sampling (pooling) layers
Output of each layer is called a feature map.

Convolution Neural Network (CNN)

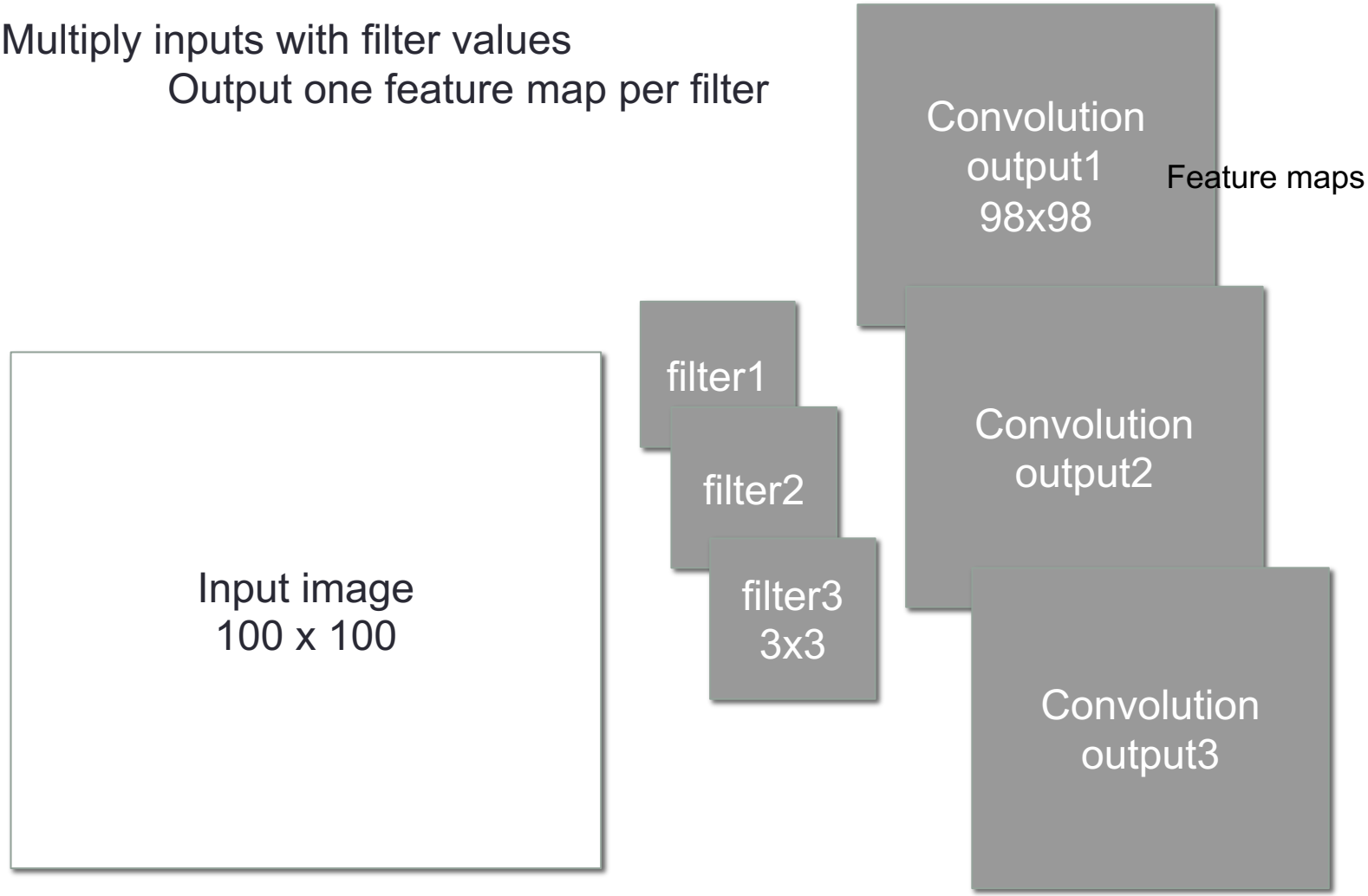
- Hierarchy of representations with increasing level of abstraction
- Each stage is a kind of trainable feature transform
- Image recognition: Pixel →edge → texture → part → object



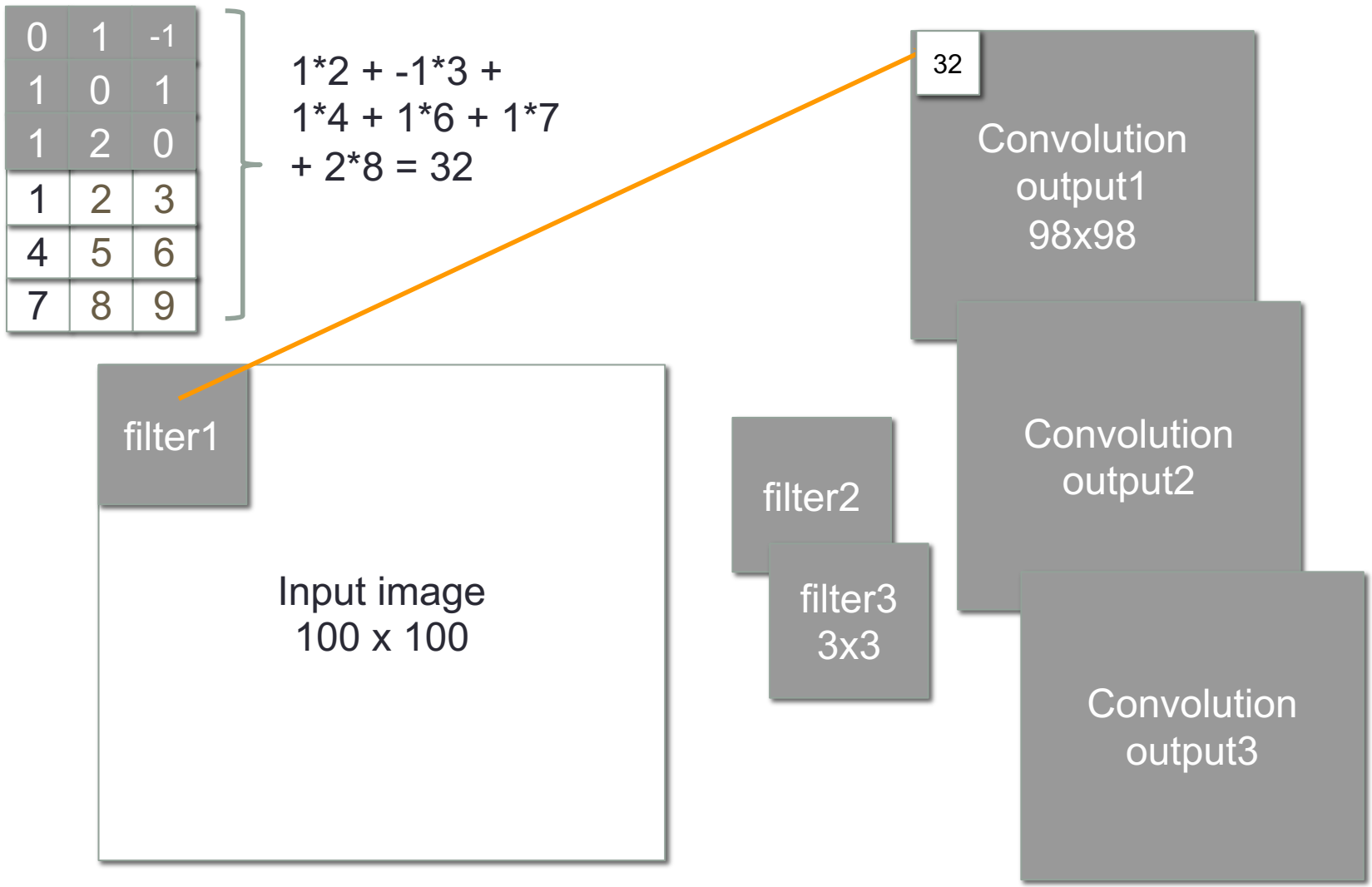
Convolution filters

Multiply inputs with filter values

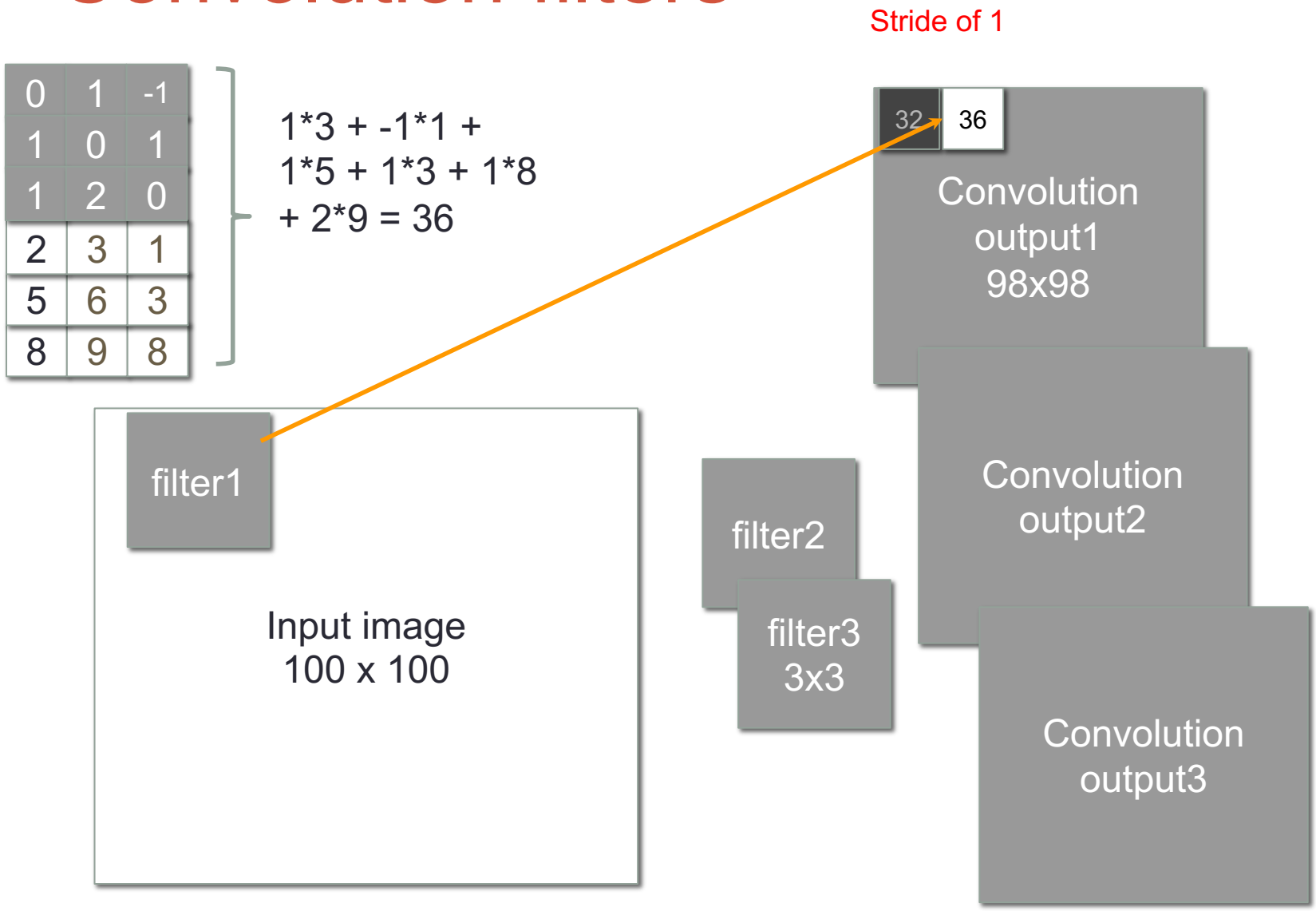
Output one feature map per filter



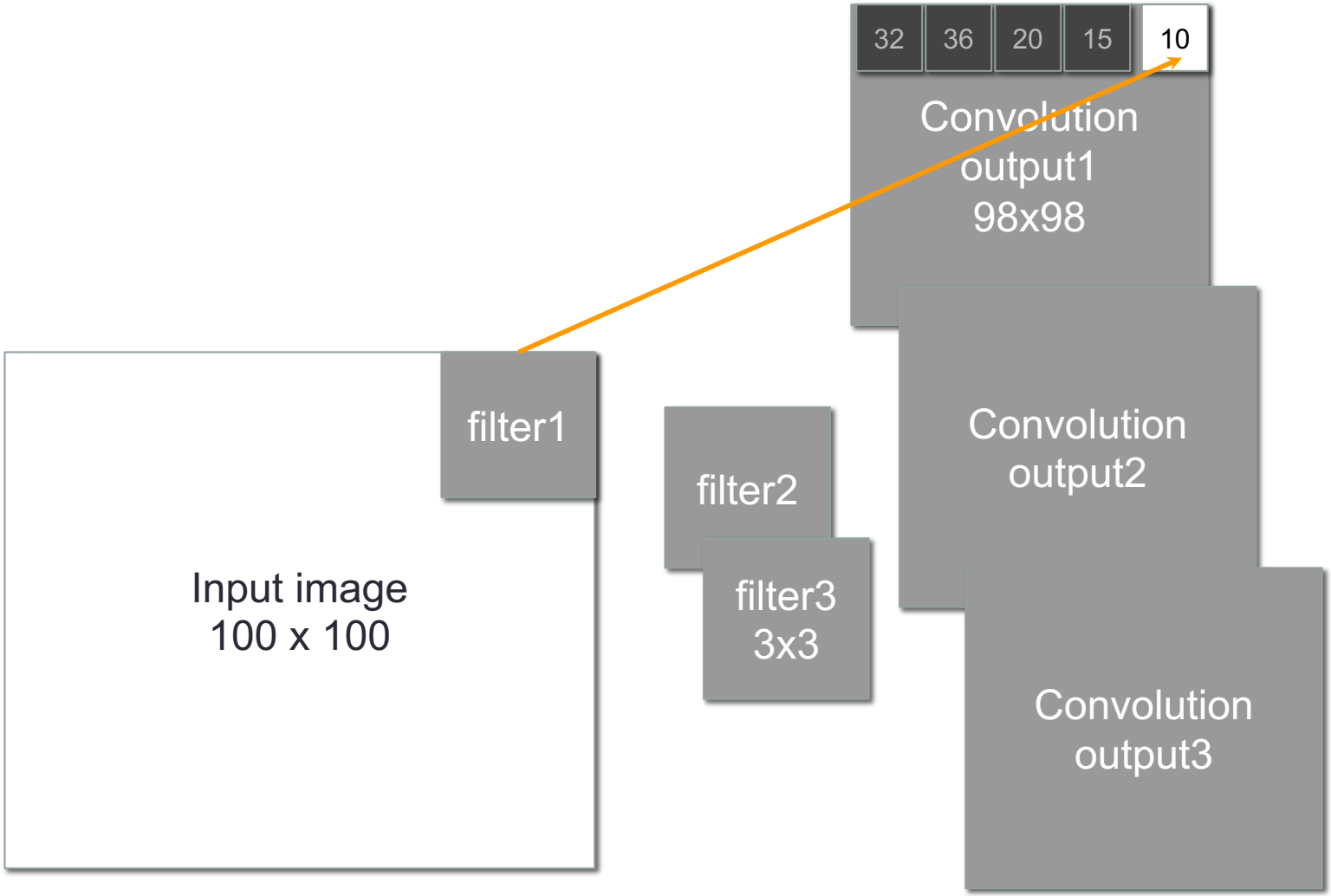
Convolution filters



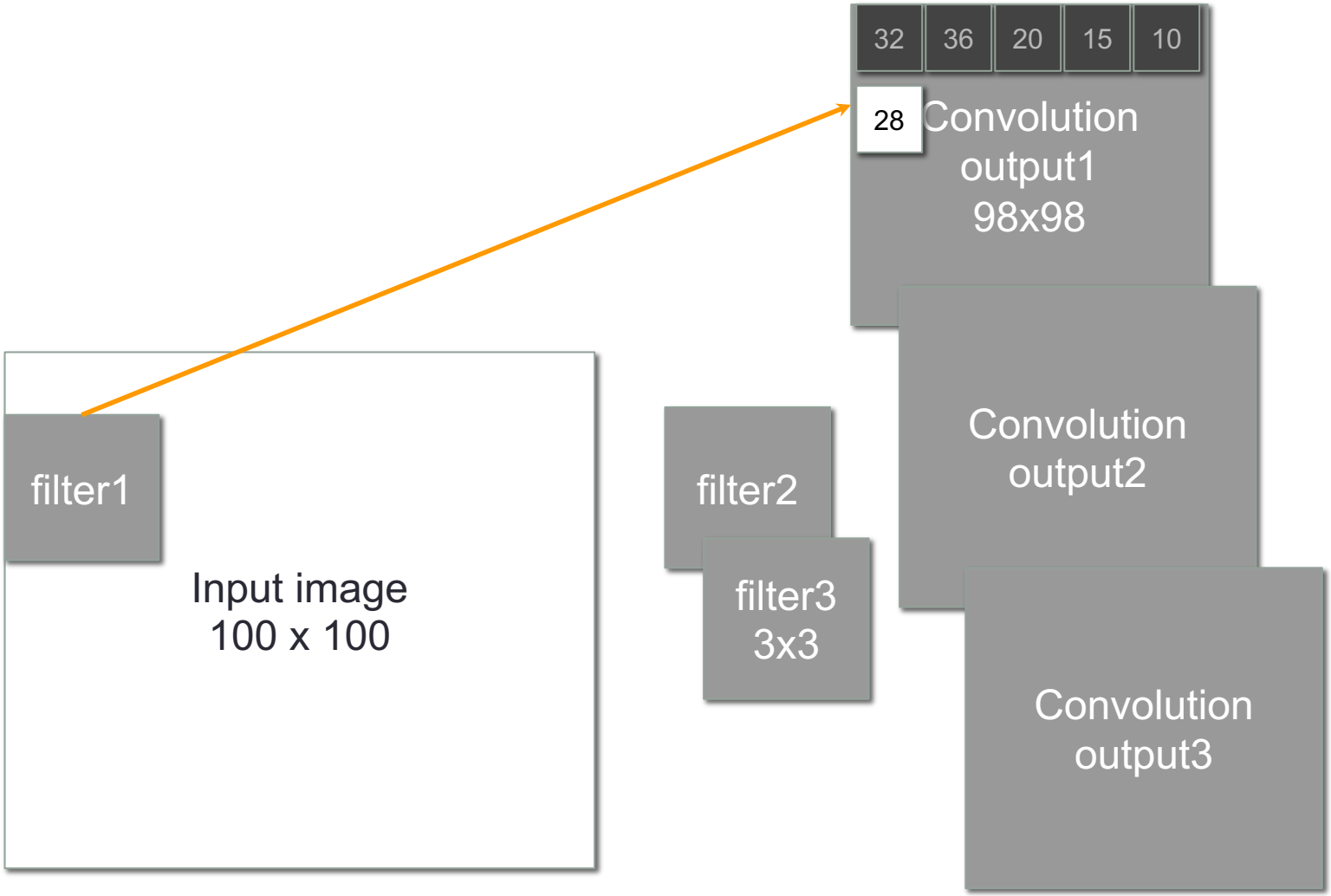
Convolution filters



Convolution filters

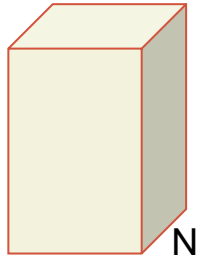


Convolution filters



Convolution filters

N filters means N feature maps
You get a 3 dimensional output



32	36	20	15	10
28	72	0	12	50
32	36	18	9	2
17	6	2	17	1

16

filter2

filter1

filter3
3x3

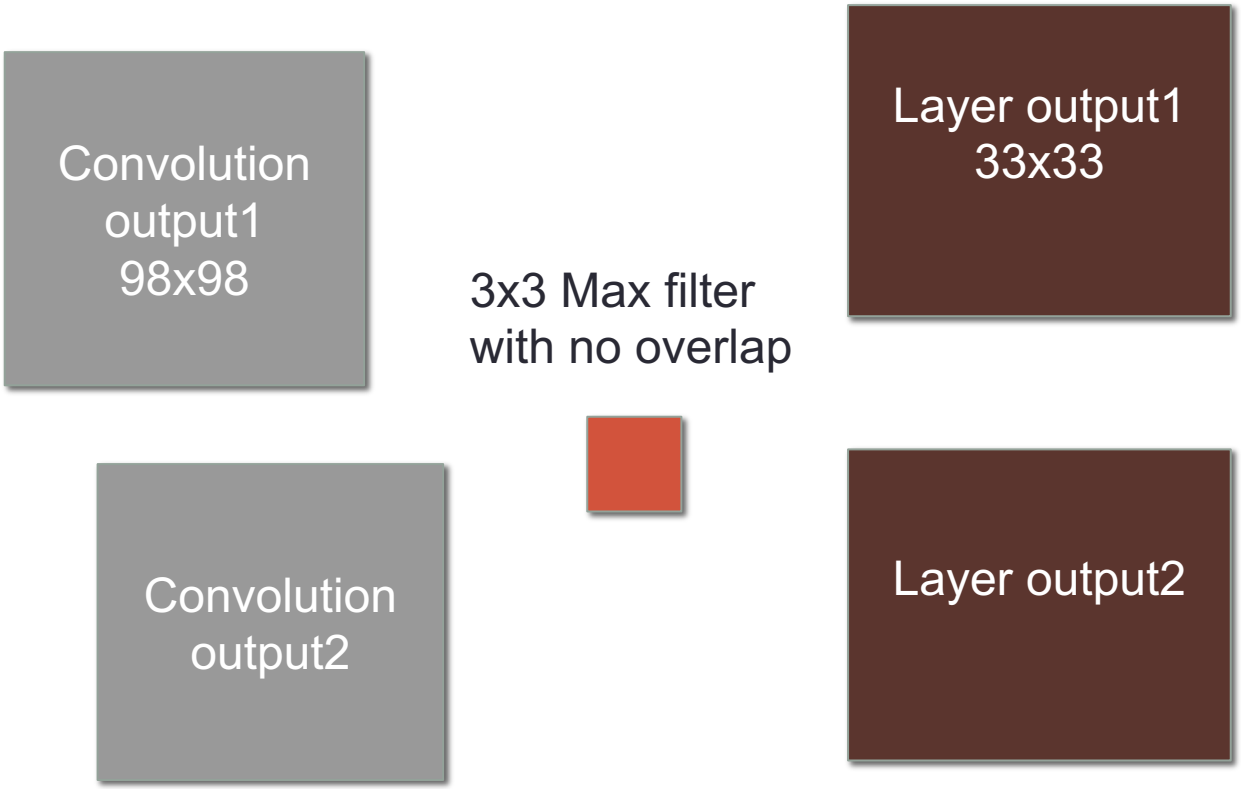
Input image
100 x 100

Convolution
output2

Convolution
output3

Pooling/subsampling

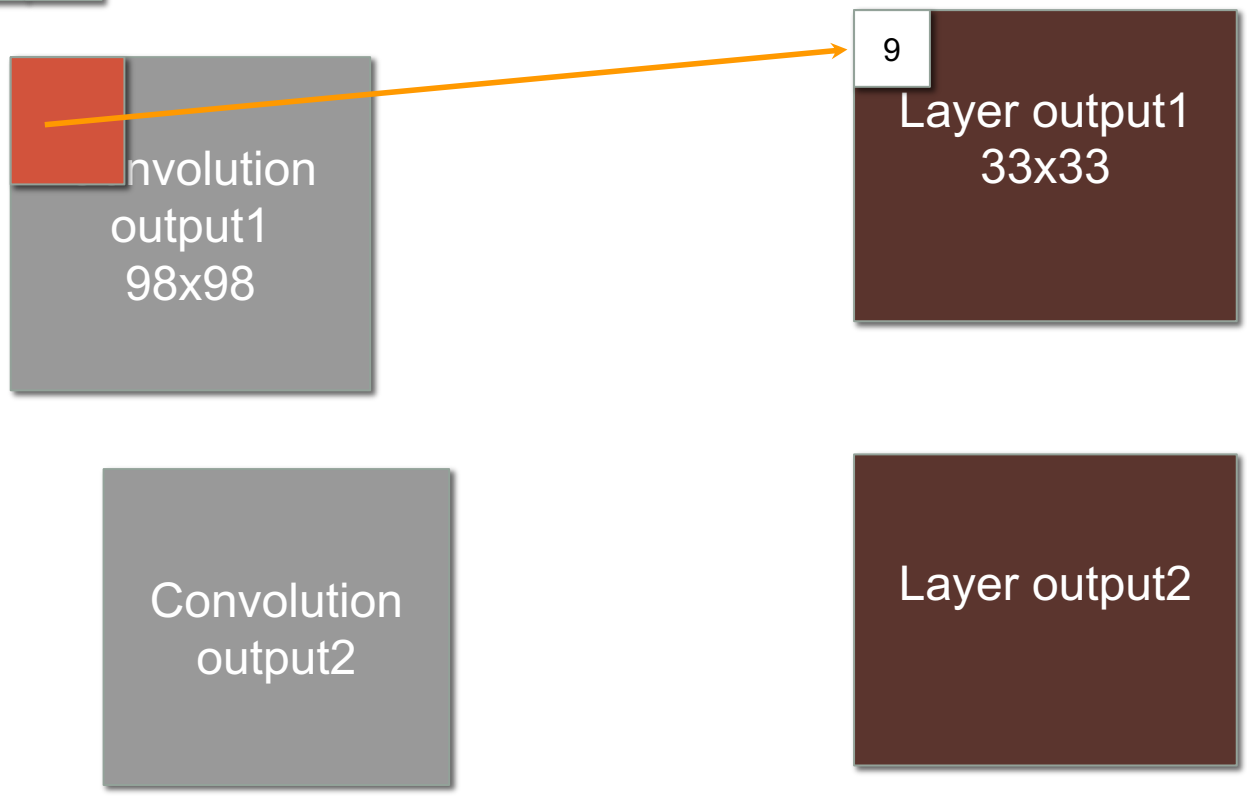
Reduce dimension of the feature maps



Pooling/subsampling

1	2	3
4	5	6
7	8	9

Max = 9

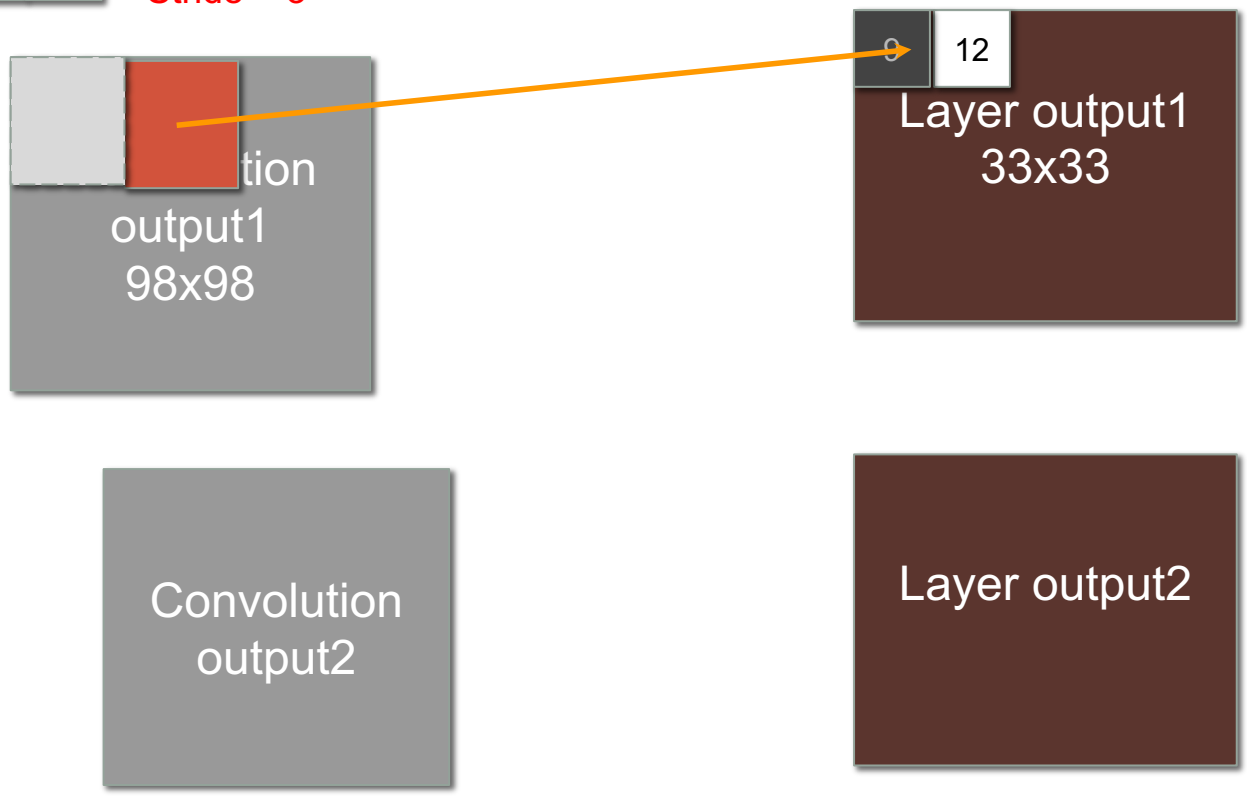


Pooling/subsampling

5	2	1
5	7	1
9	5	12

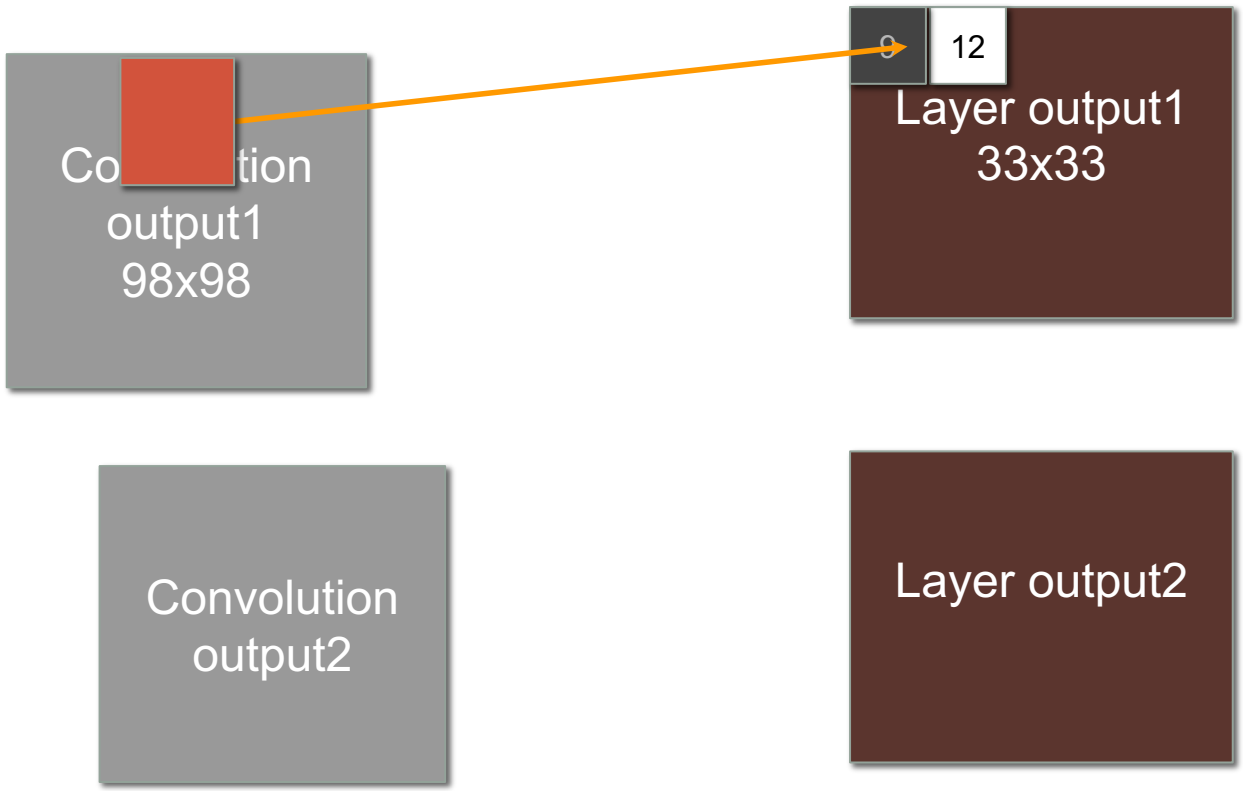
Max = 12

Stride = 3



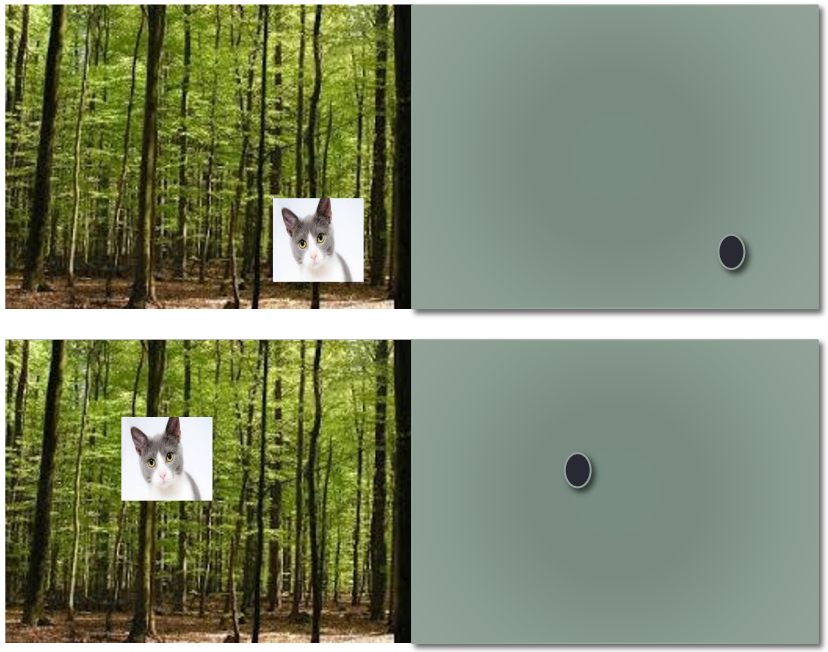
Pooling/subsampling

Can use other functions besides max
Example, average



Notes on convolution

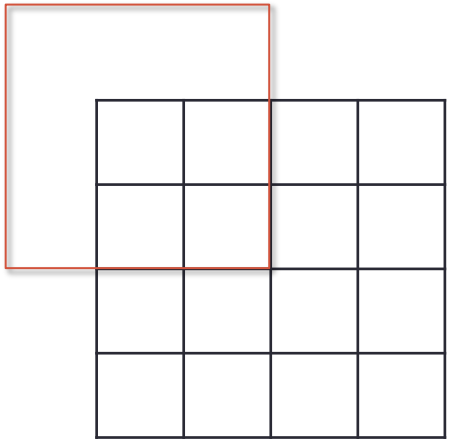
- Convolution layer gives us shift-equivariant (shifting the input, will shift the output equally)
- Using Max pooling (globally) after the convolution layers will give us shift-invariant (shifting the input, will yield the same output)



Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

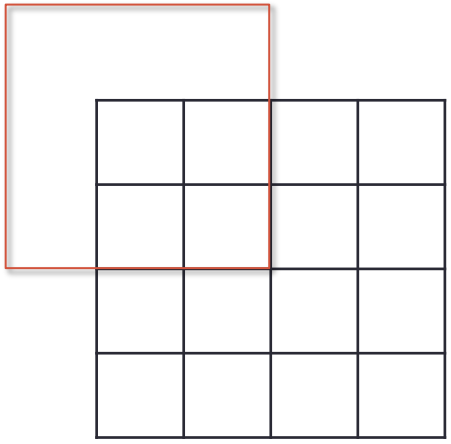
What is the output size?

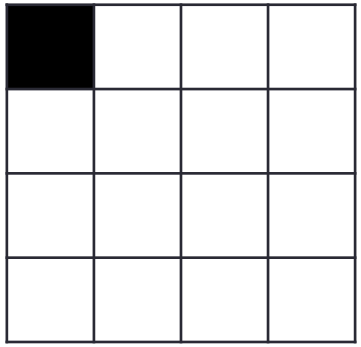


Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

What is the output size?

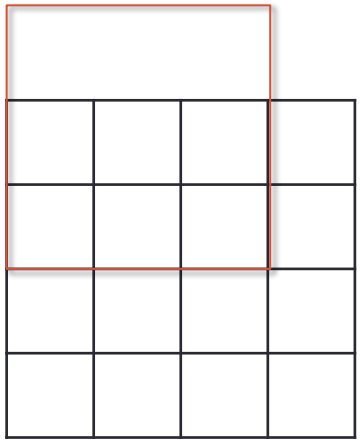


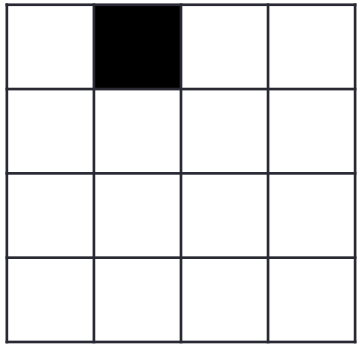


Convolution puzzle

5 filters 3x3 filter pad, stride 1, pad 1

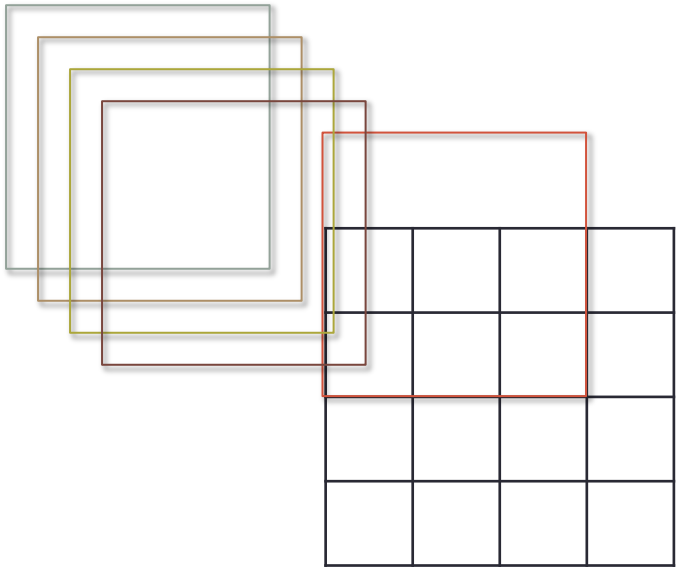
What is the output size?

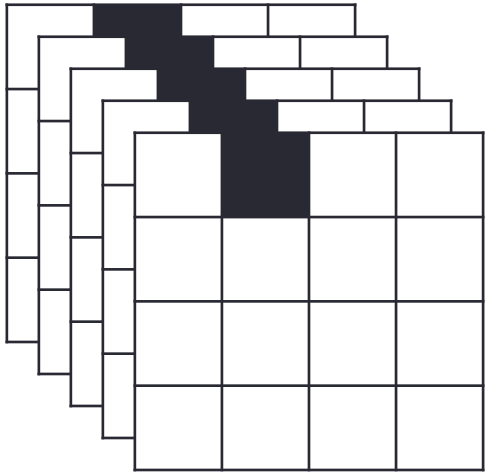




Convolution puzzle

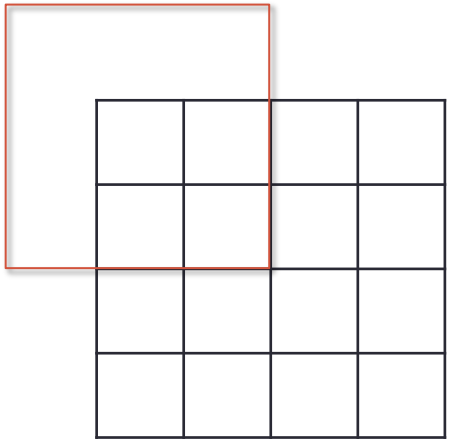
5 filters 3x3 filter pad, stride 1, pad 1





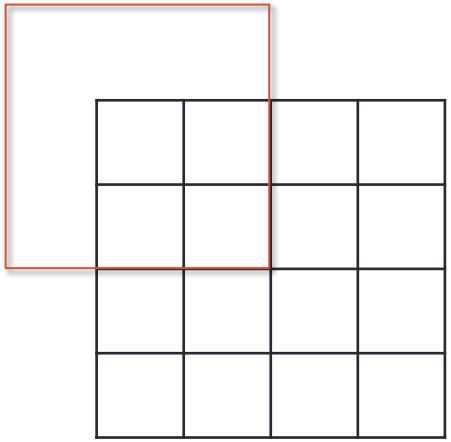
Convolution puzzle

3x3 filter pad, stride 2, pad 1



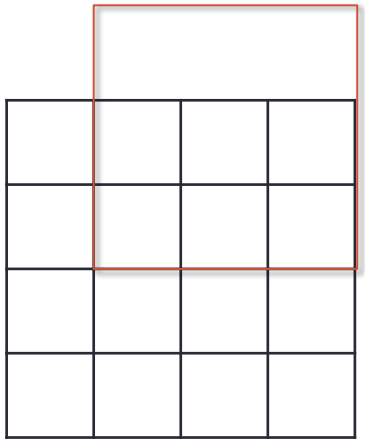
Convolution puzzle

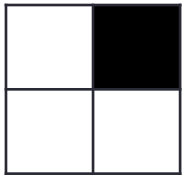
3x3 filter pad, stride 2, pad 1



Convolution puzzle

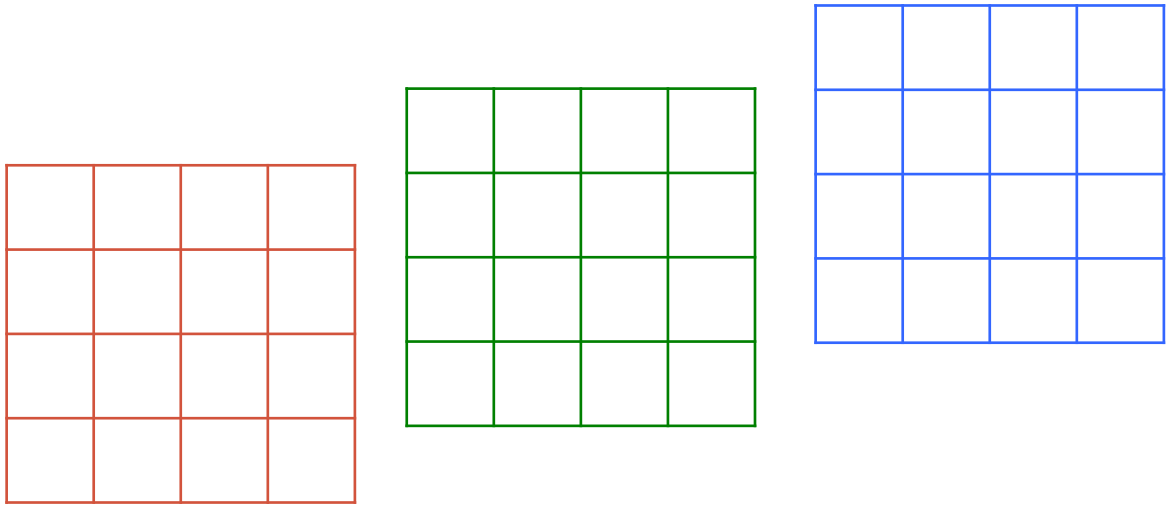
3x3 filter pad, stride 2, pad 1





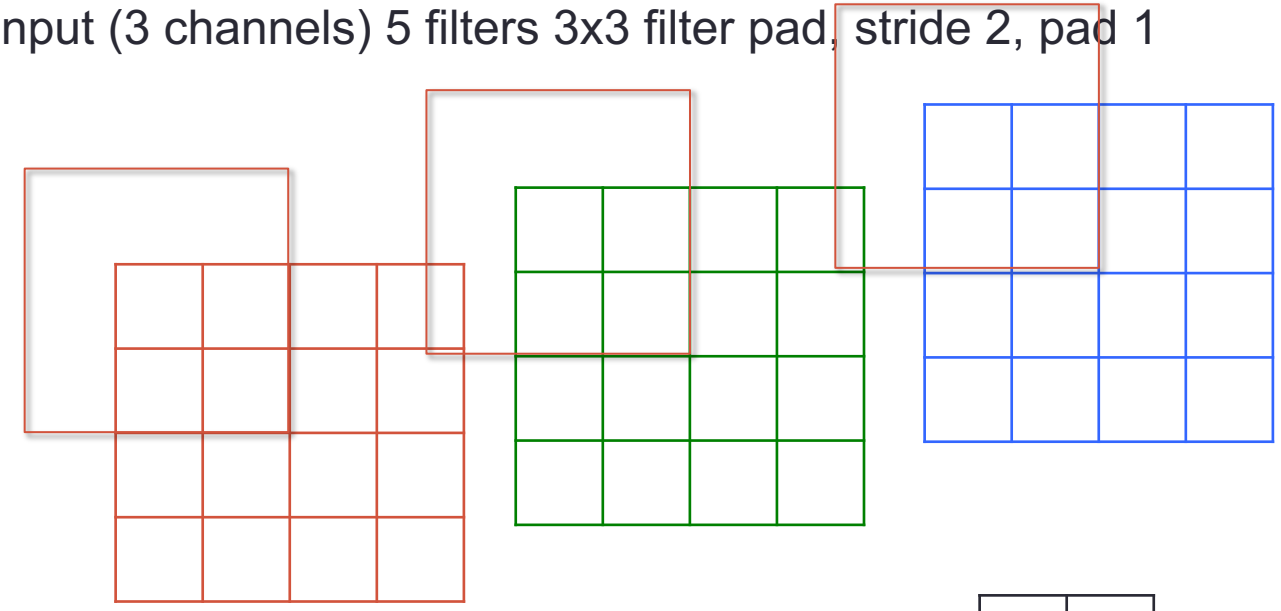
Convolution puzzle

RGB input (3 channels) 5 filters 3x3 filter pad, stride 2, pad 1

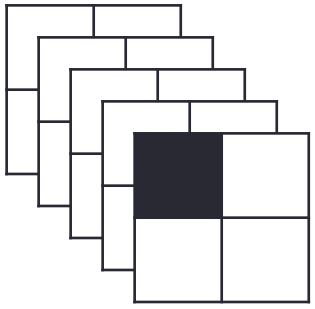


Convolution puzzle

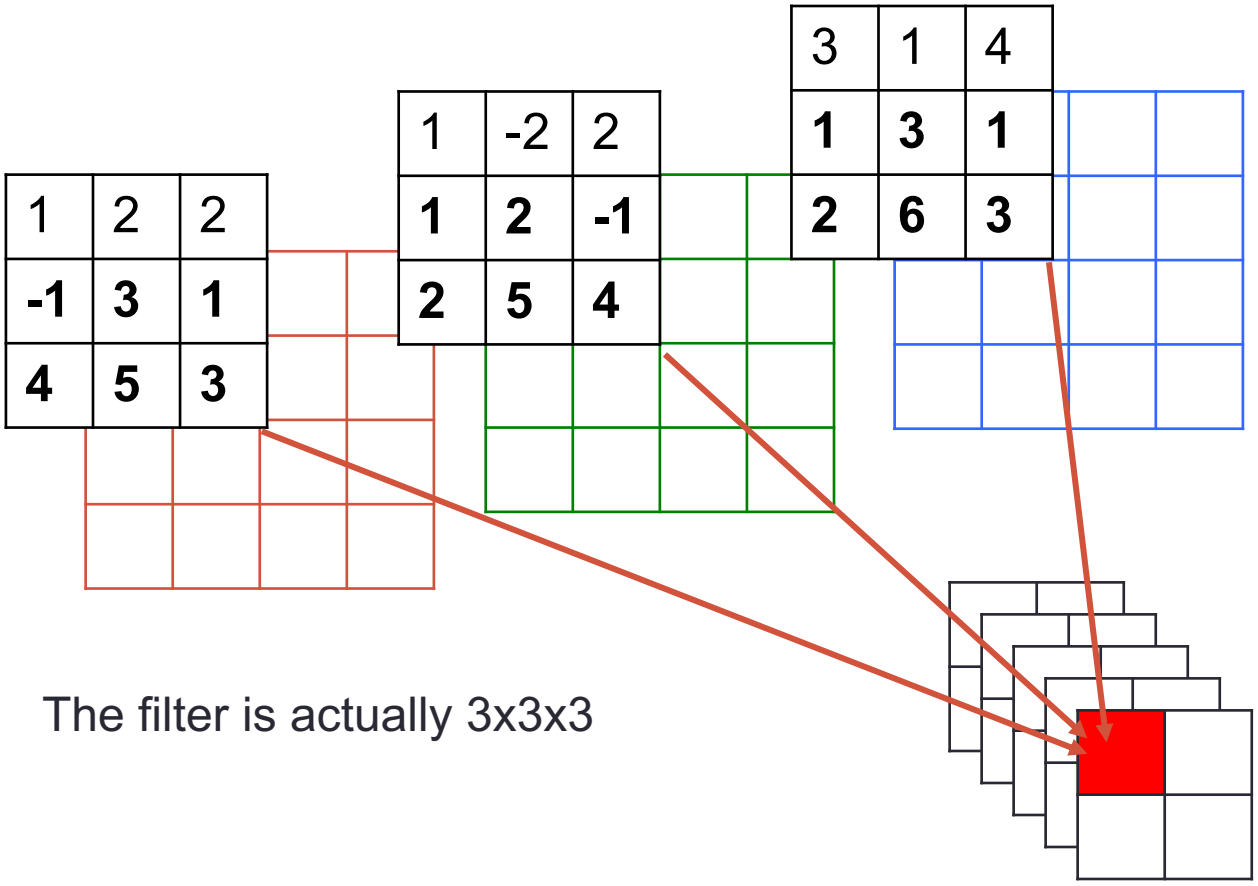
RGB input (3 channels) 5 filters 3x3 filter pad, stride 2, pad 1



The filter is actually 3x3x3

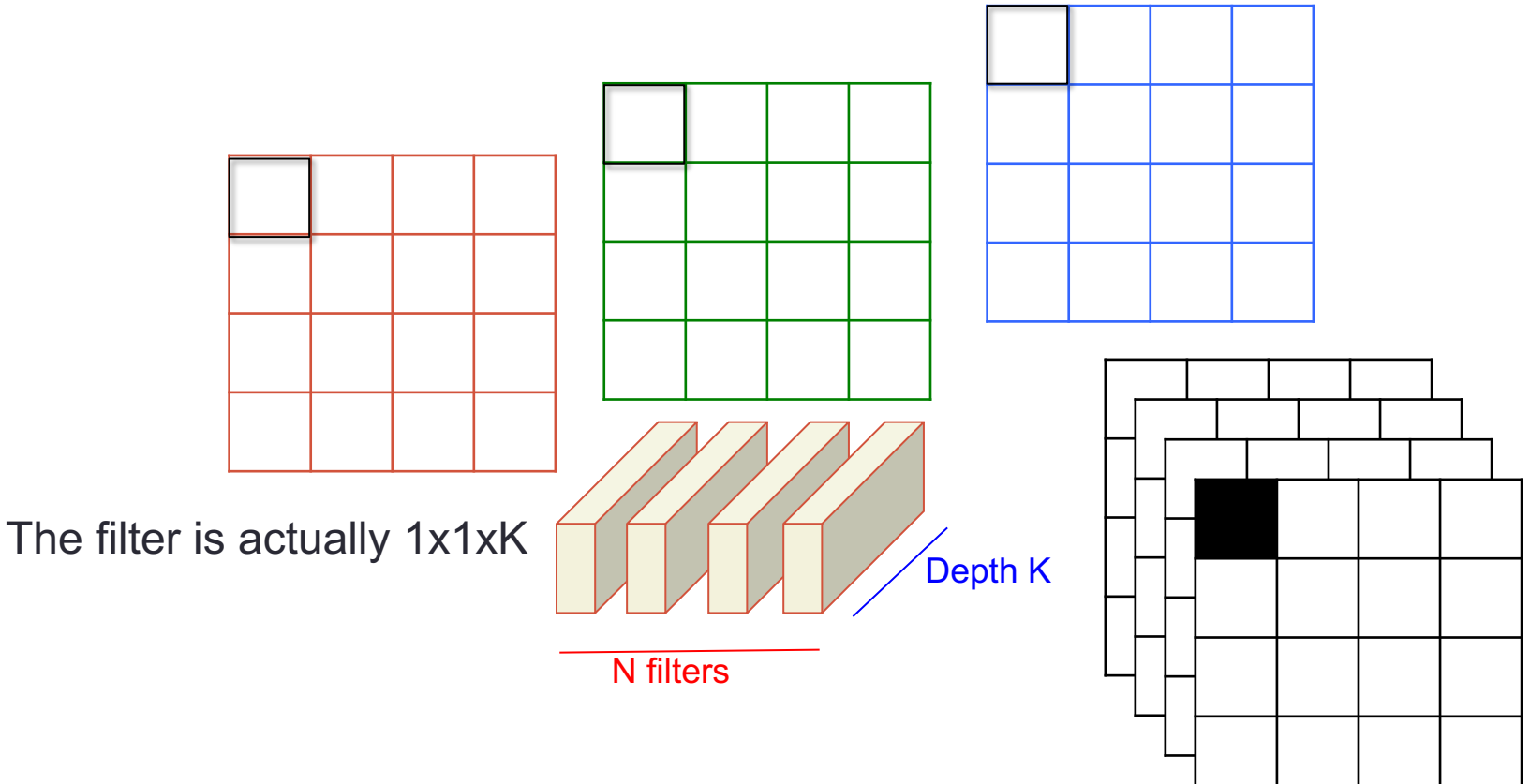


Convolution puzzle



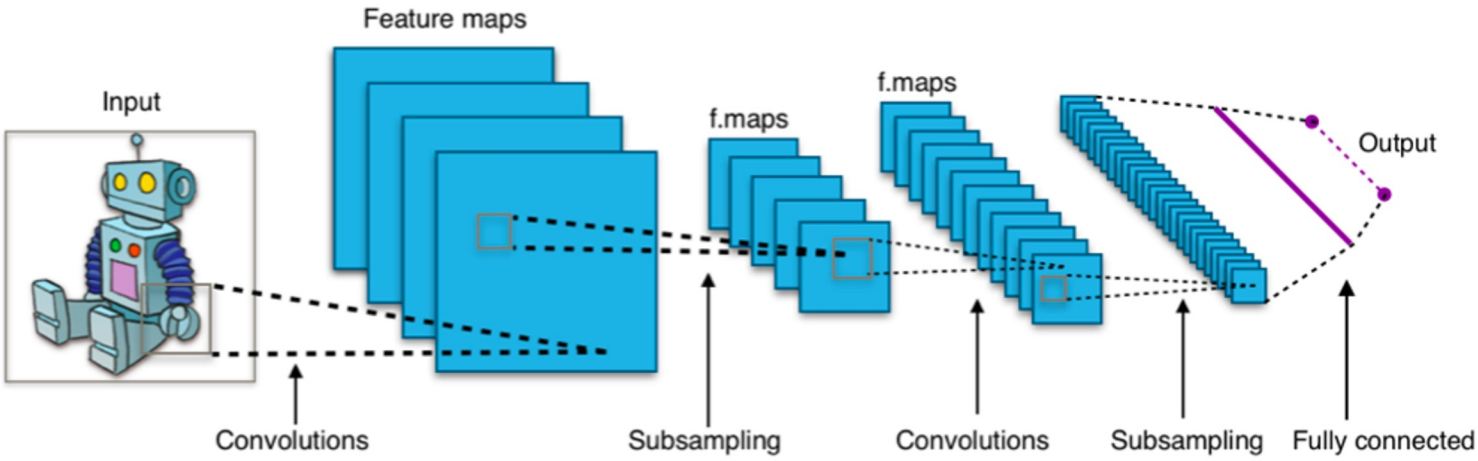
1x1 convolution

Reduces the dimension of feature maps



CNN overview

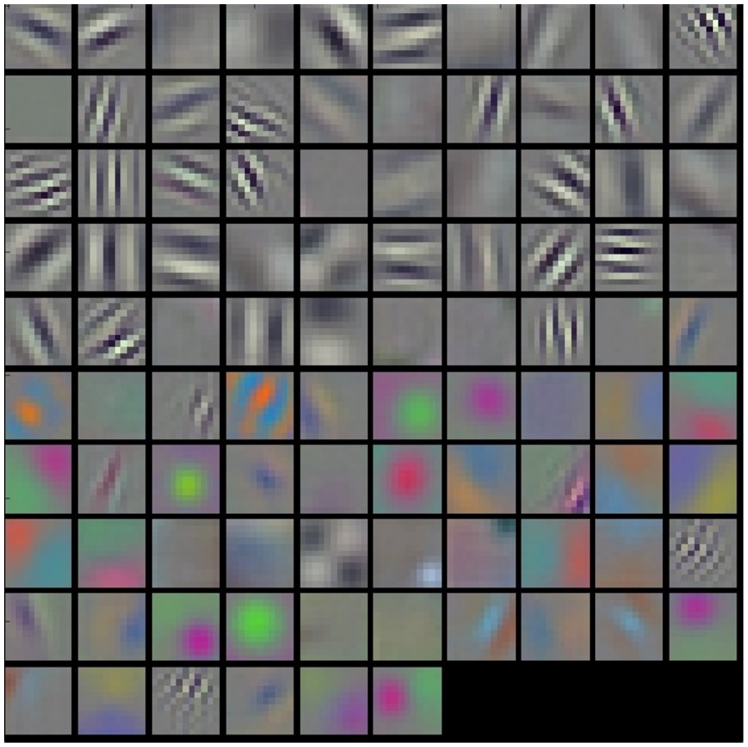
- Filter size, number of filters, filter shifts, and pooling rate are all parameters
- Usually followed by a fully connected network at the end
  - CNN is good at learning low level features
  - DNN combines the features into high level features and classify



https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Typical_cnn.png

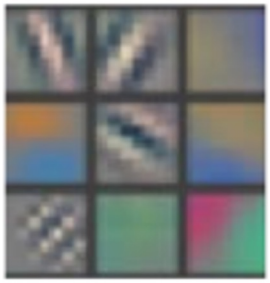
Visualizing convolutional layers

- Just like PCA, we can visualize the weights of a transform
- “Matched filters”

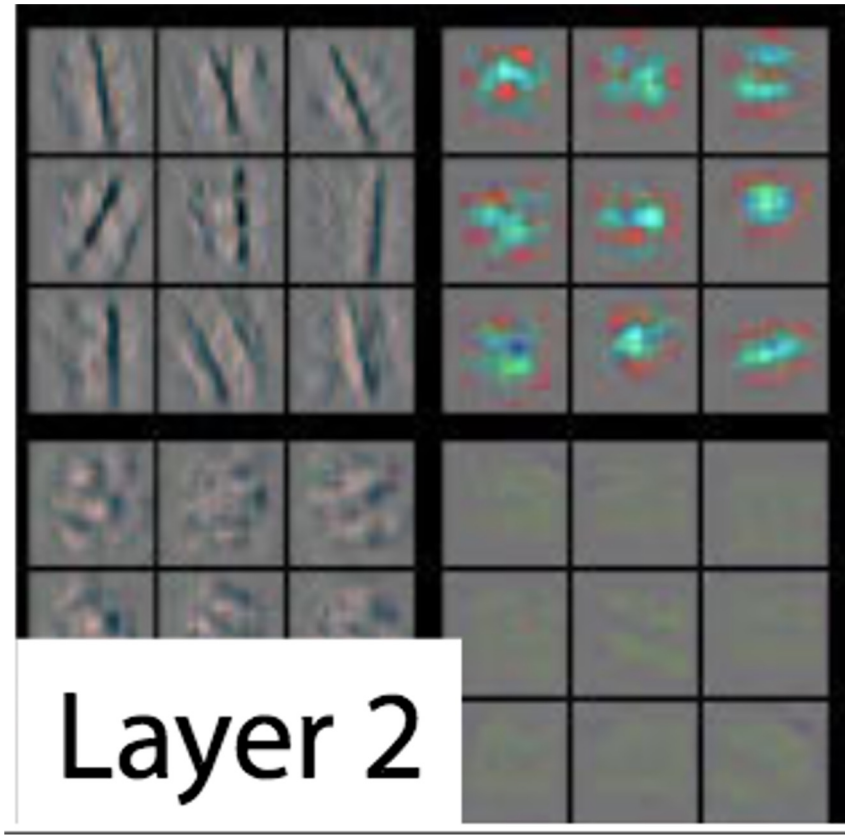


http://cs231n.github.io/understanding-cnn/

Higher layer captures higher-level concepts

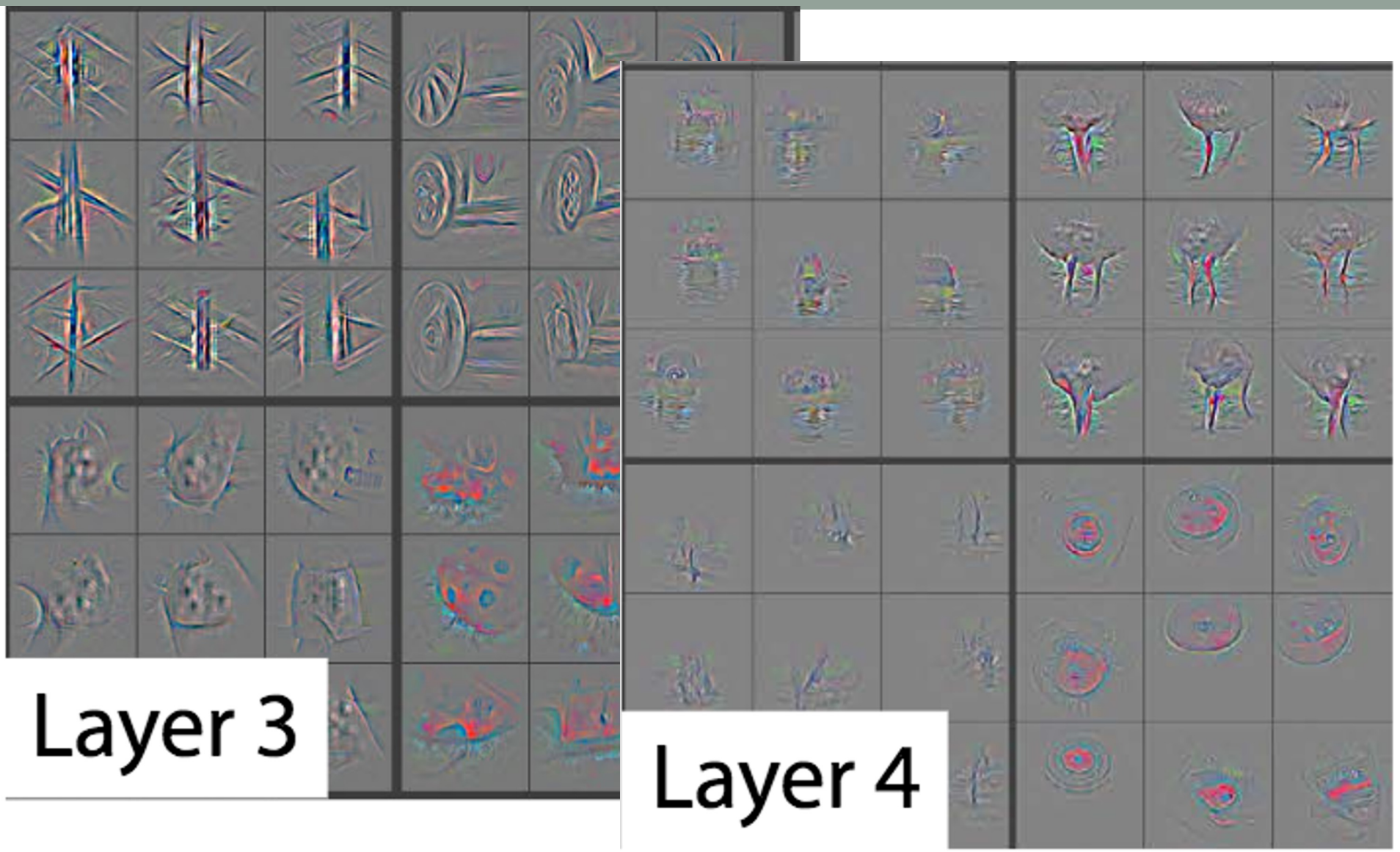


Layer 1



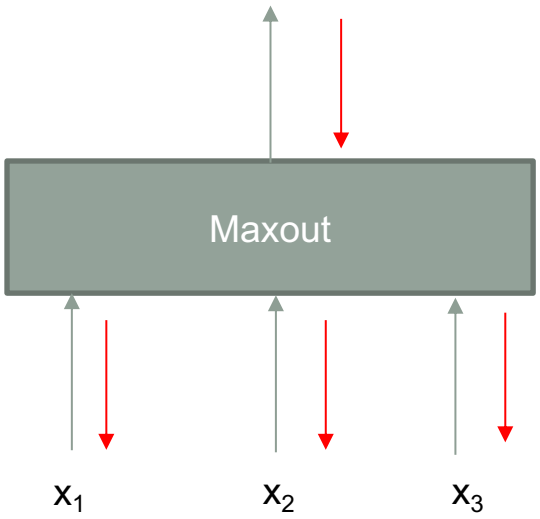
Layer 2

https://arxiv.org/pdf/1311.2901.pdf


Layer 3
Layer 4

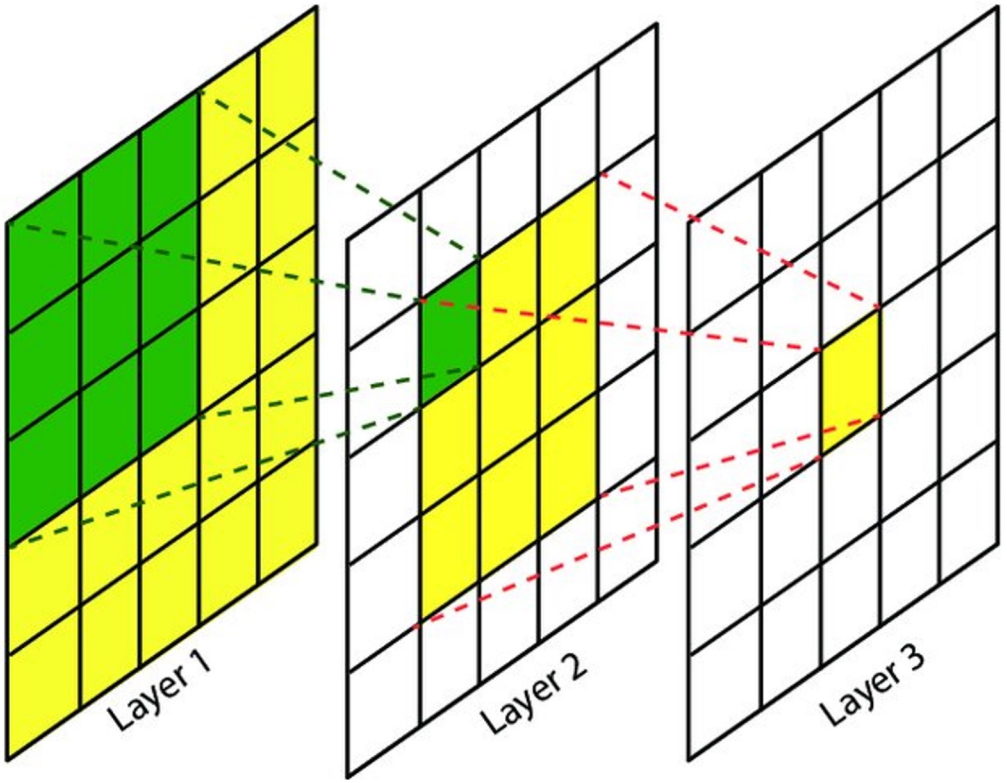
Pooling/subsampling

- Max filter -> Maxout
  - Backward pass?



Receptive field

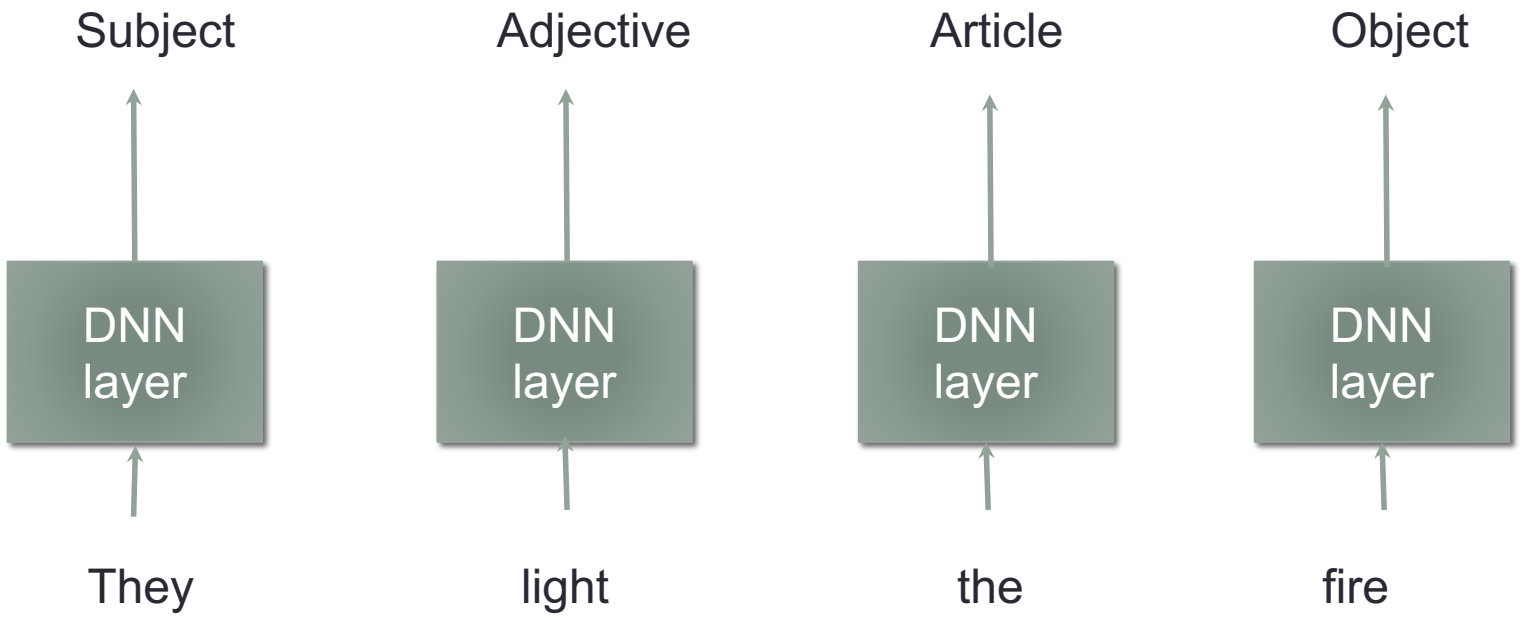
You might want to consider about how large is your pattern when designing your network



source image

Recurrent neural network (RNN)

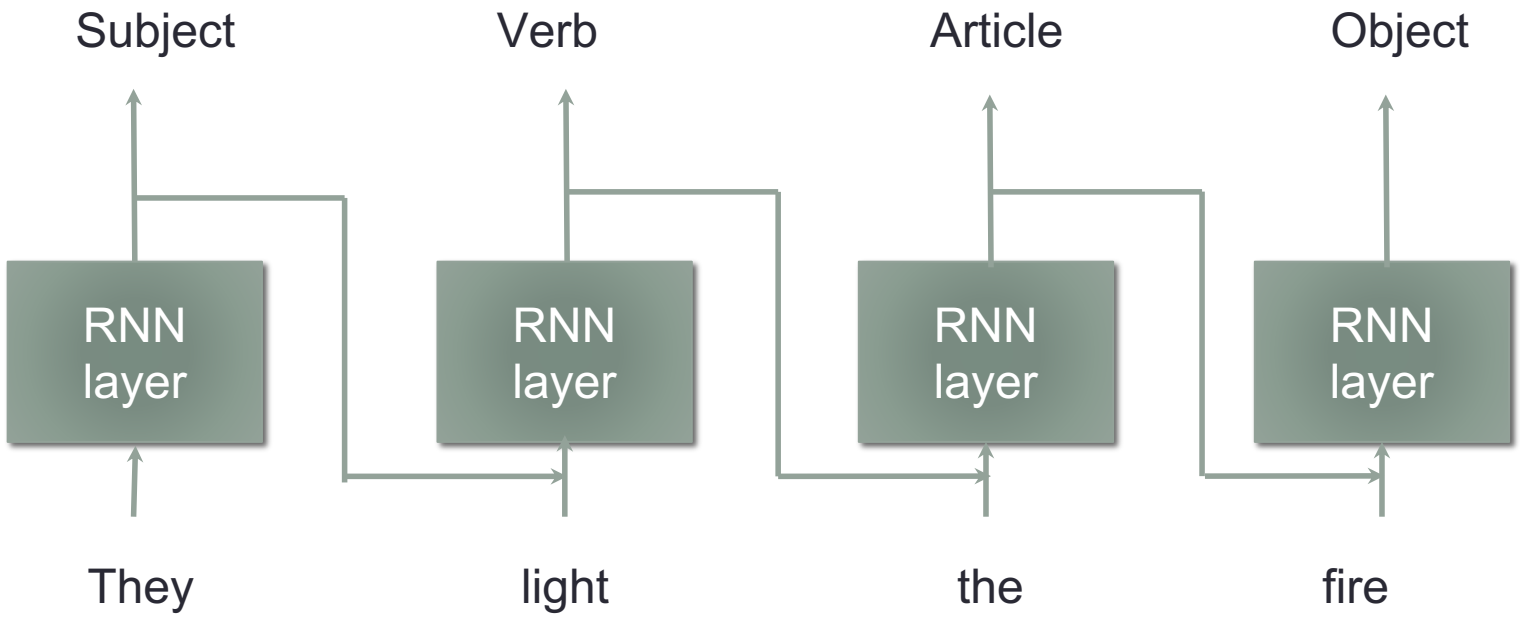
- DNN framework



Problem 1: need a way to remember the past

Recurrent neural network (RNN)

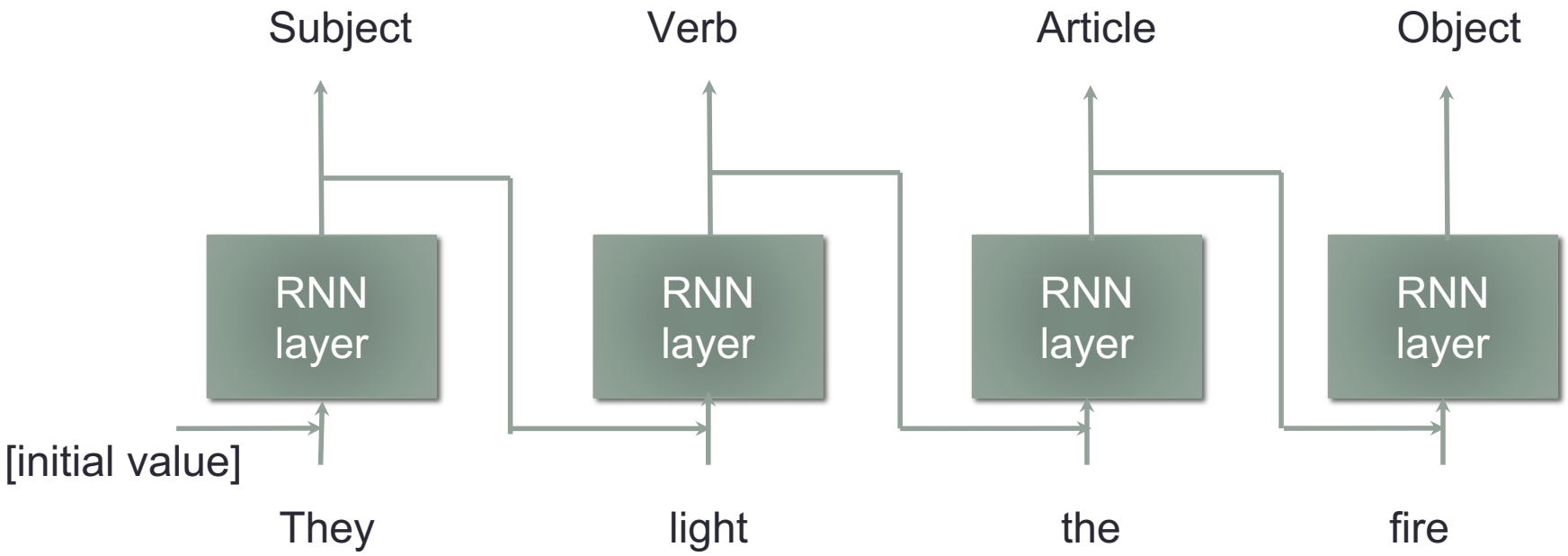
- RNN framework



Output of the layer encodes something meaningful about the past

Recurrent neural network (RNN)

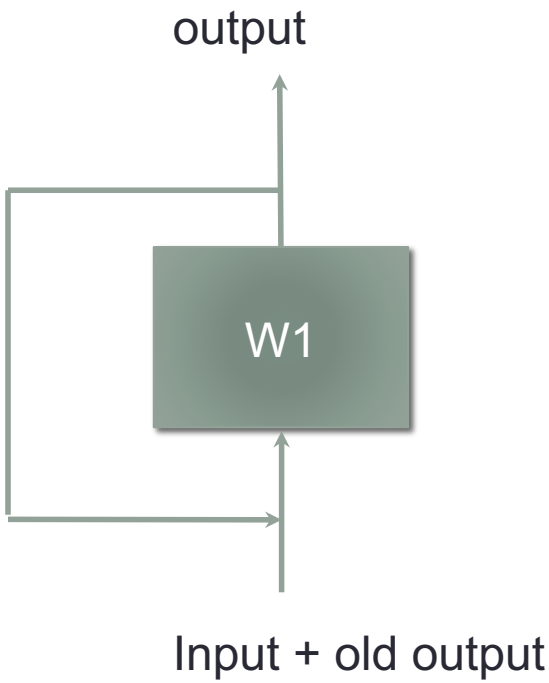
- RNN framework



New input feature = [original input feature, output of the layer at previous time step]

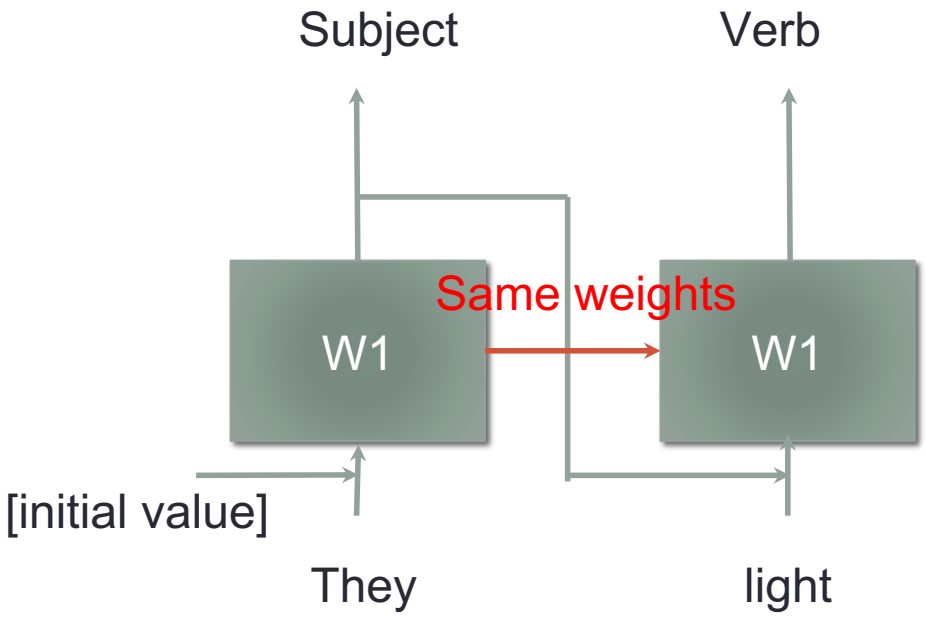
Recurrent neural network (RNN)

- Unrolling of a recurrent layer.



Recurrent neural network (RNN)

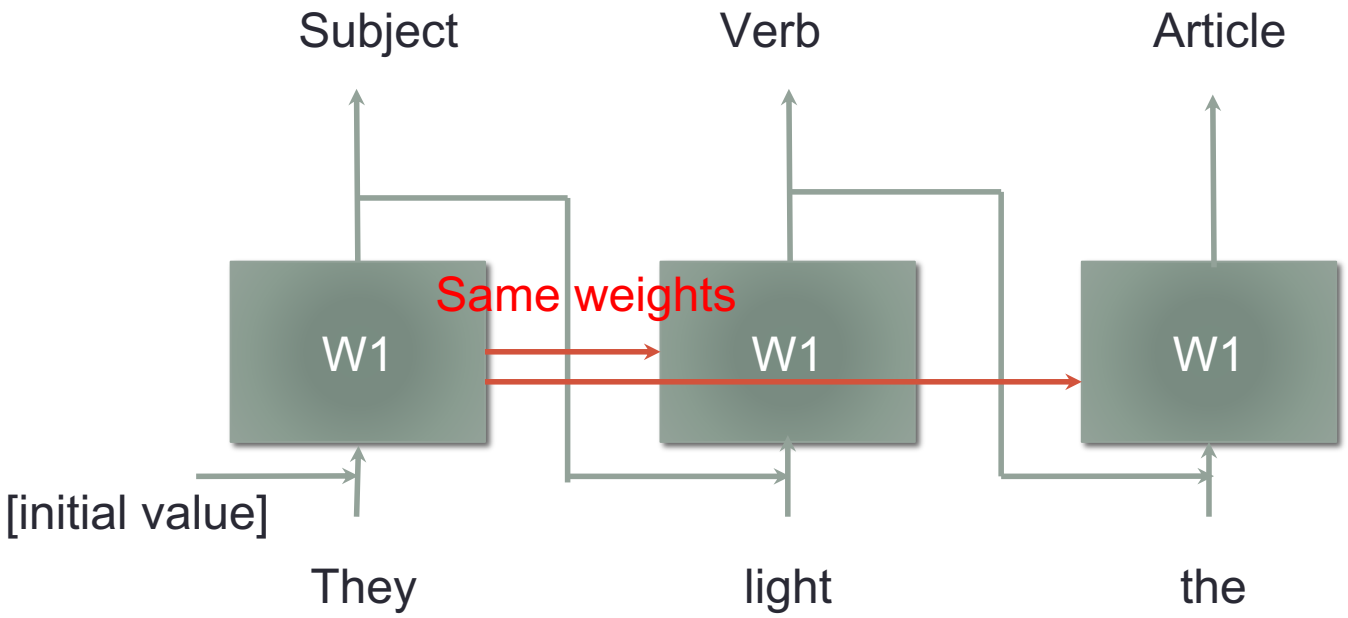
- Unrolling of a recurrent layer.



Parameter sharing across time

Recurrent neural network (RNN)

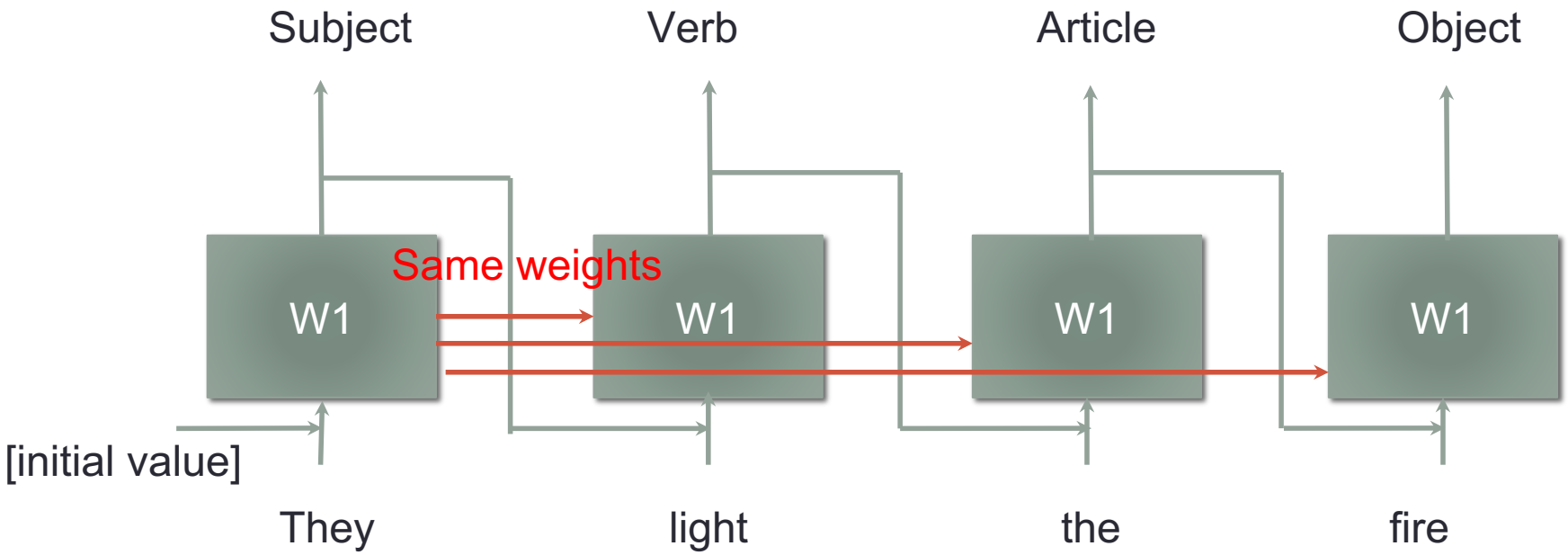
- Unrolling of a recurrent layer.



Parameter sharing across time

Recurrent neural network (RNN)

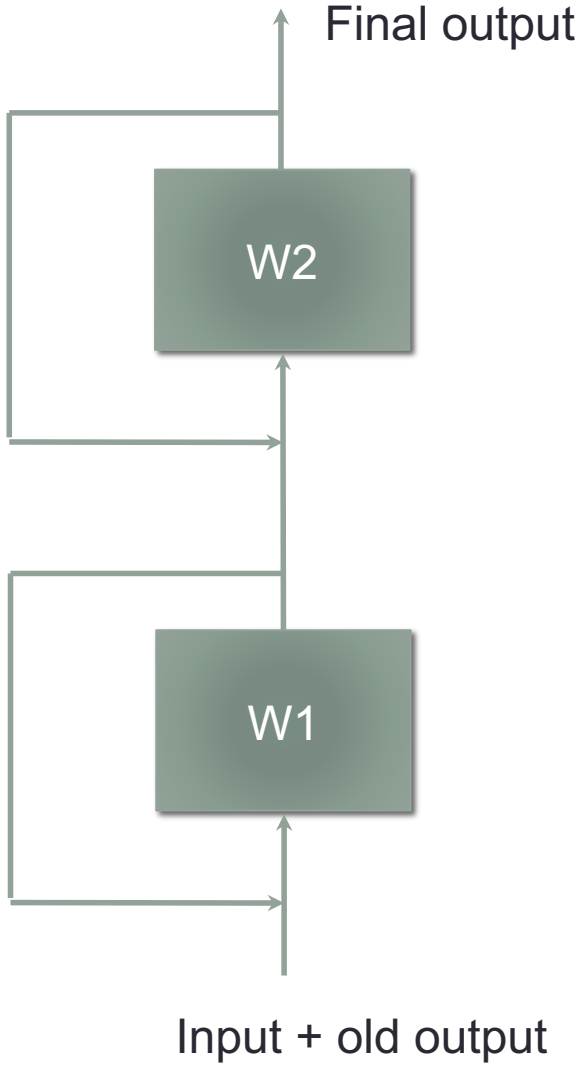
- Unrolling of a recurrent layer.



Parameter sharing across time

Recurrent neural network (RNN)

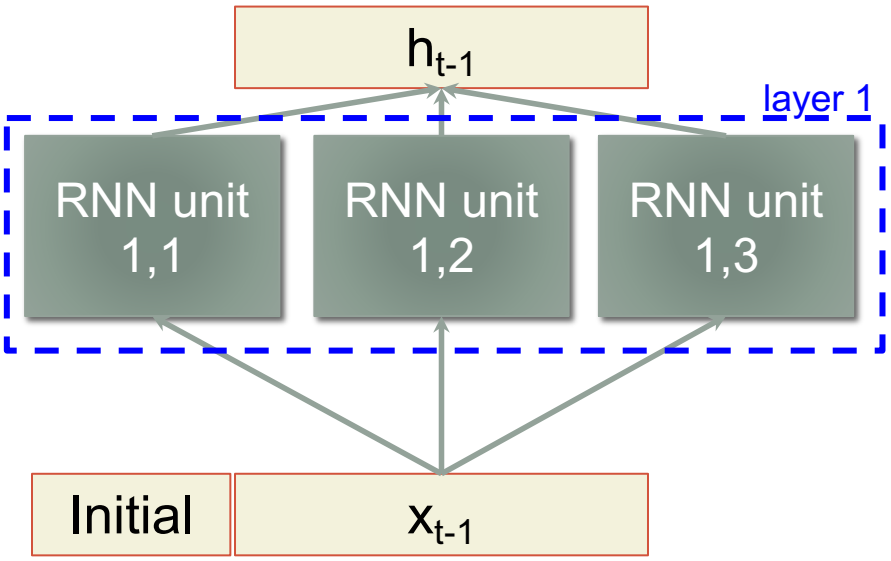
- Stacks of recurrent layer



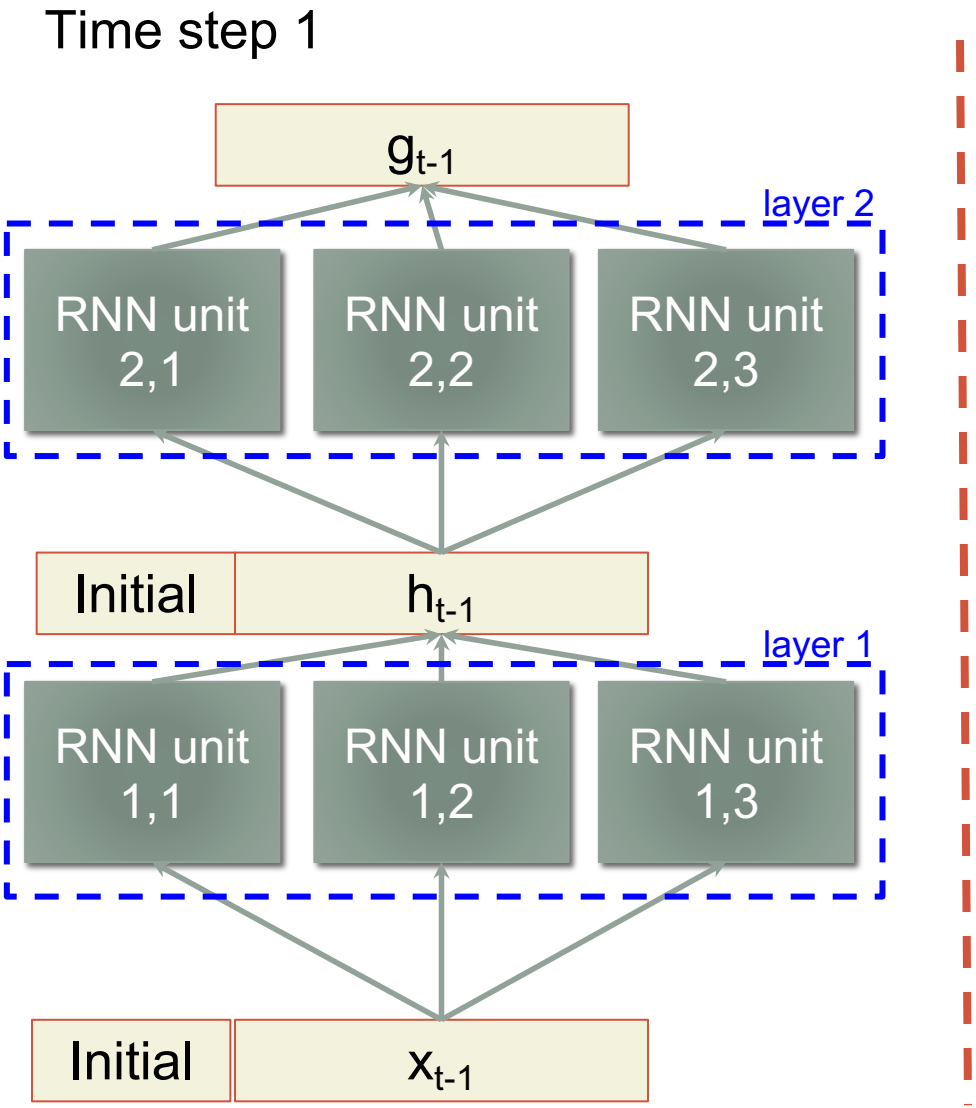
RNN layers (expanded in time)

Time step 1

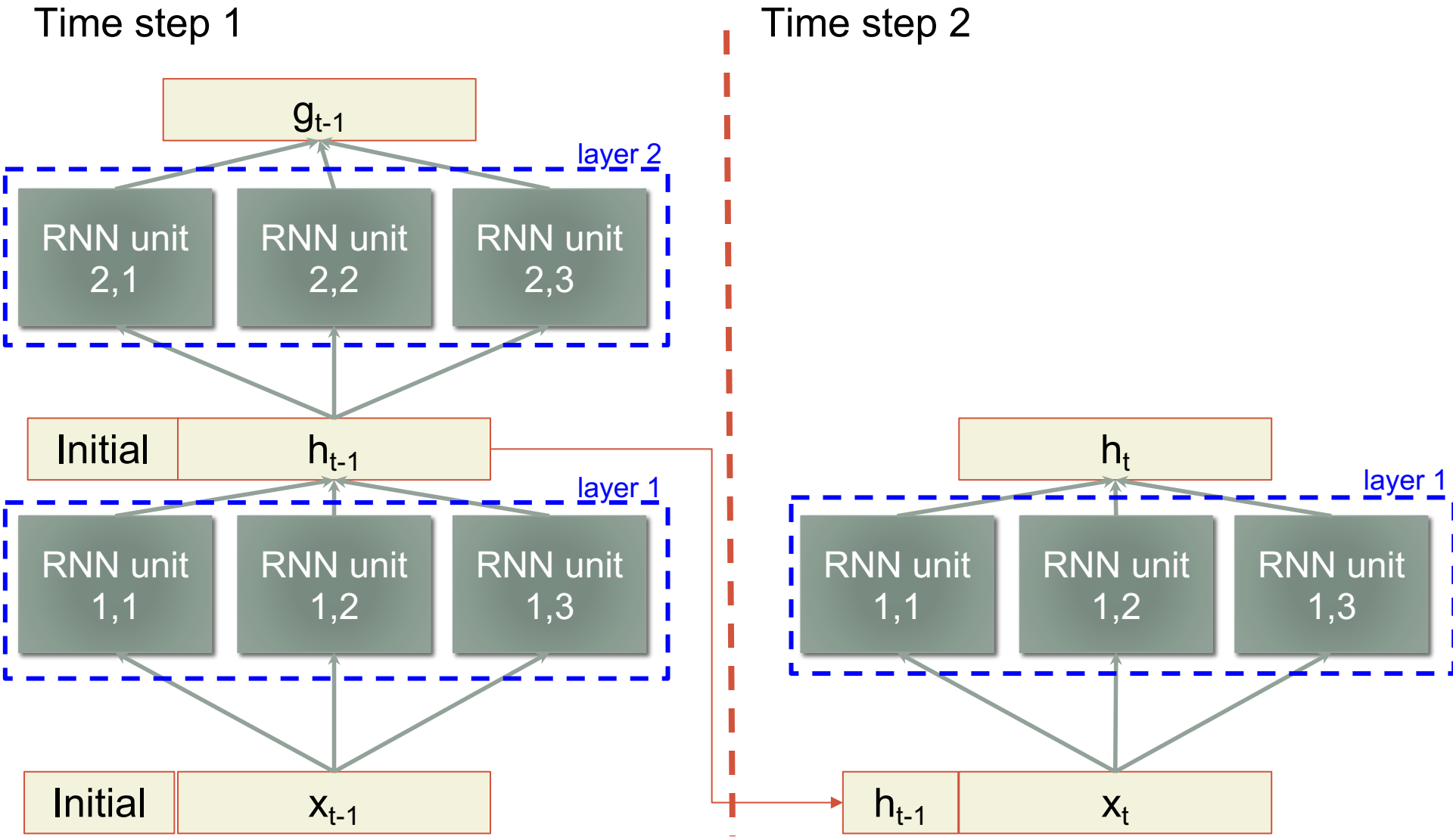
Time step 2



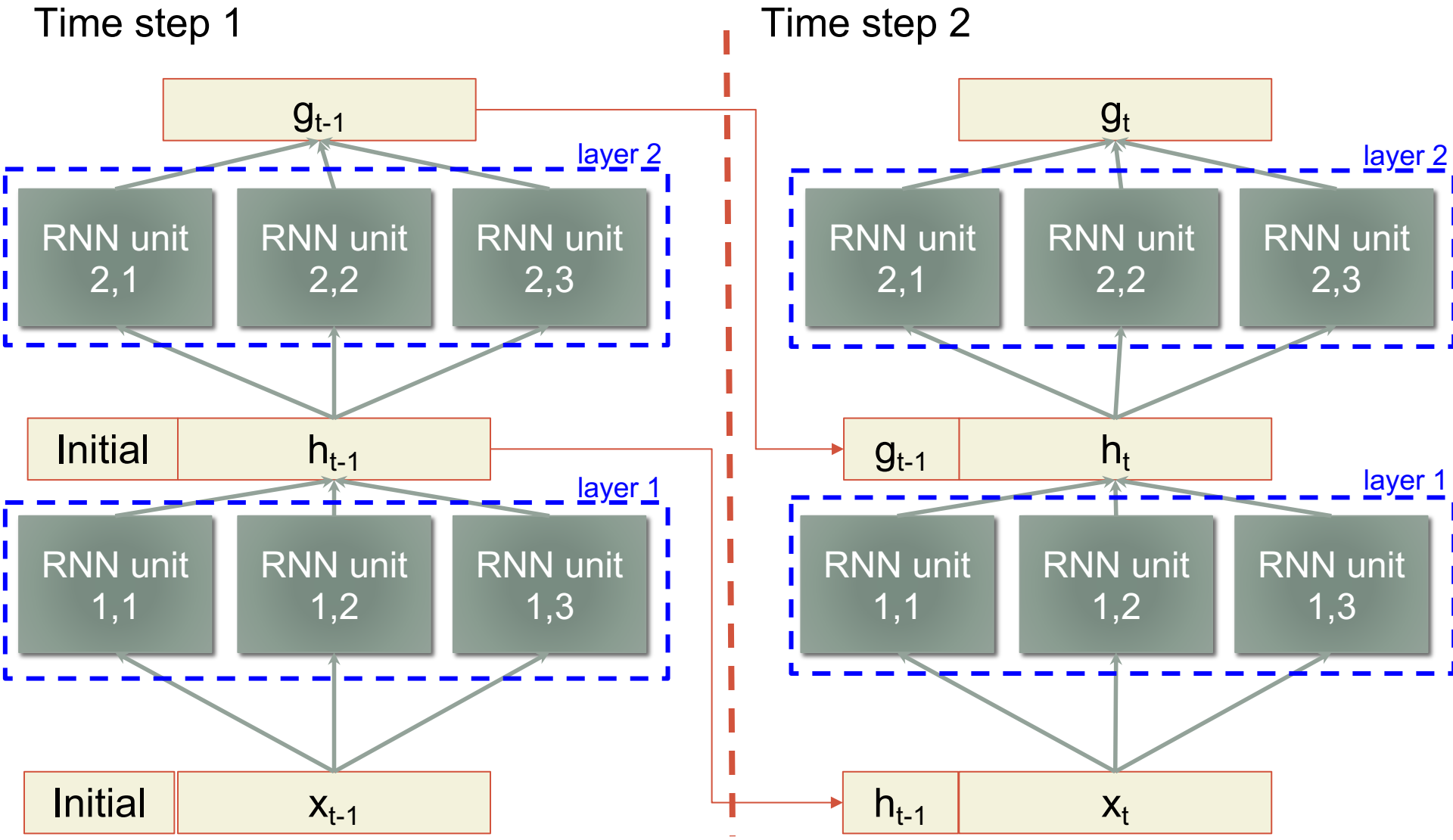
RNN layers (expanded in time)



RNN layers (expanded in time)

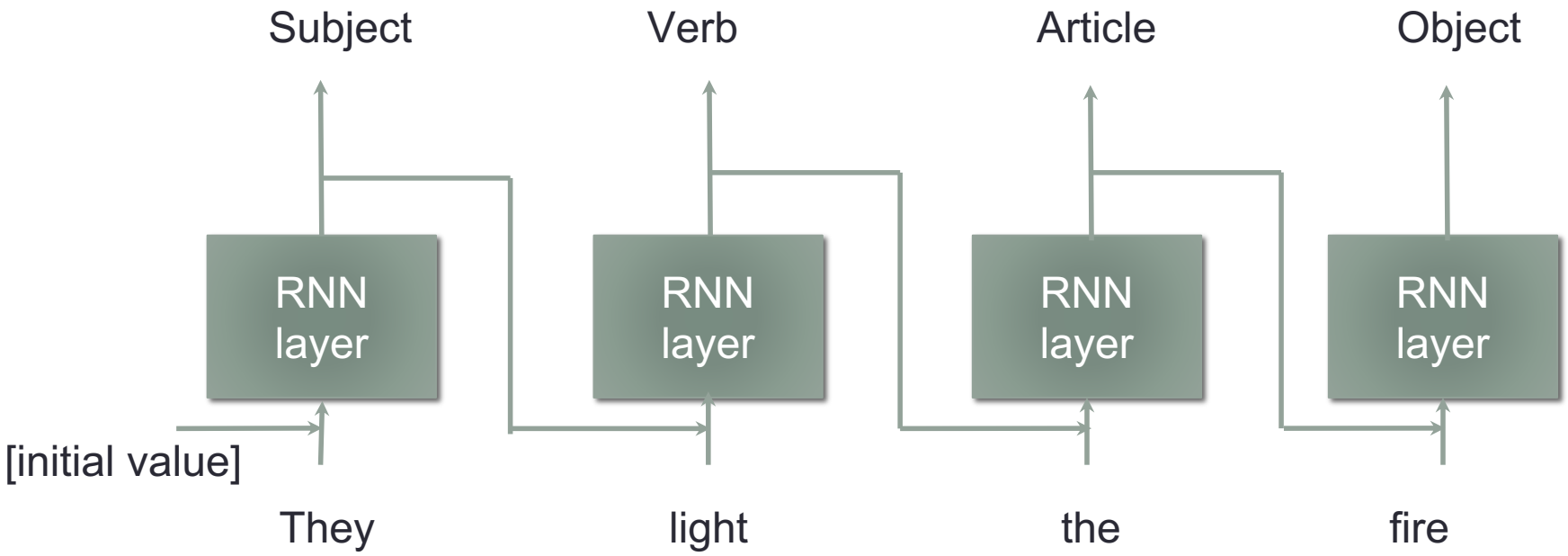


RNN layers (expanded in time)



Training a recurrent neural network

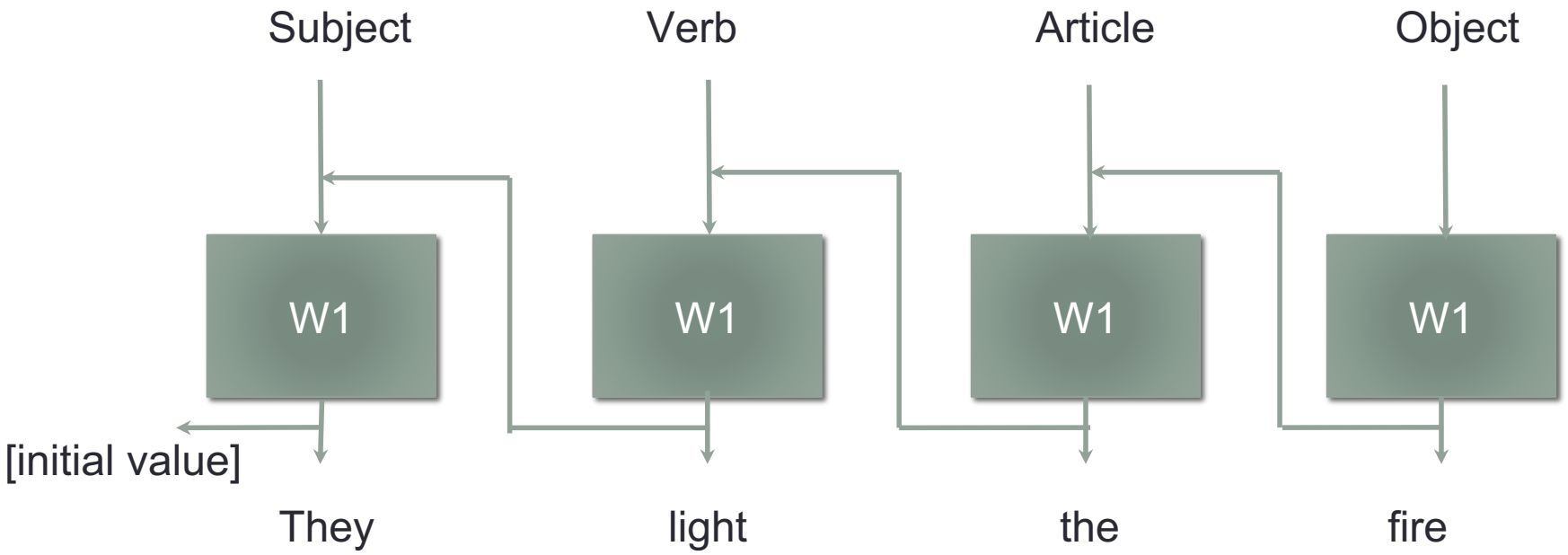
- RNN framework



New input feature = [original input feature, output of the layer at previous time step]

Training a recurrent neural network

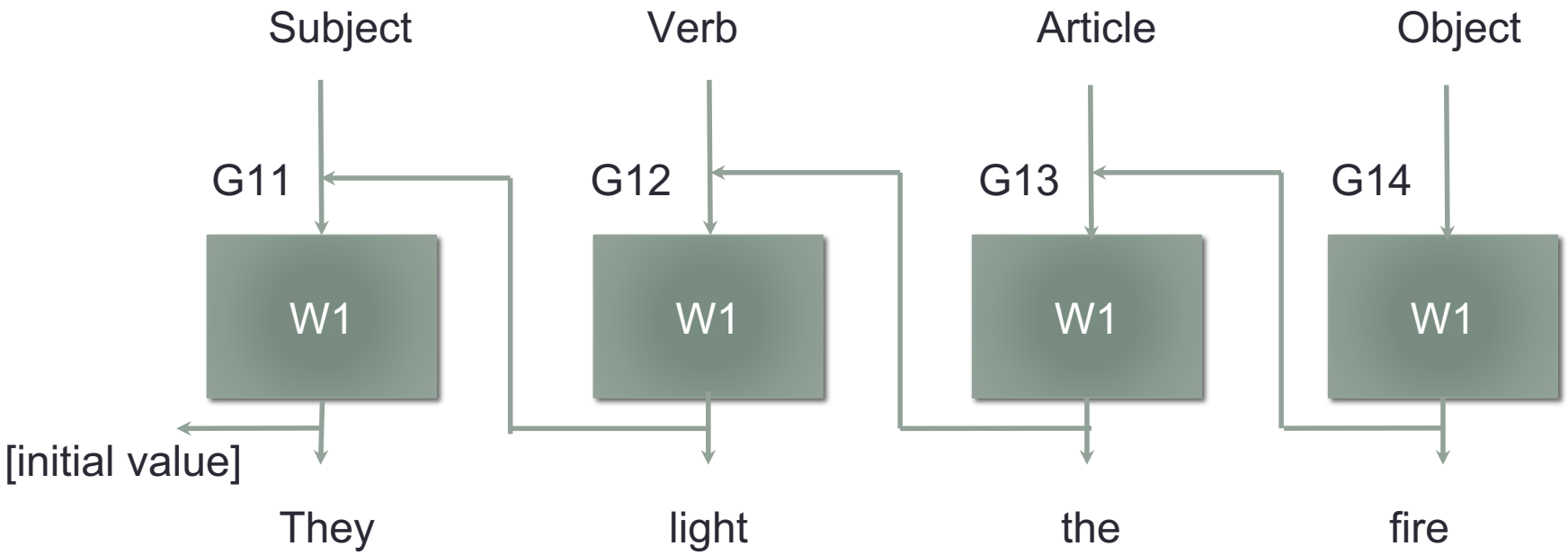
- Backward Computation graph



Backpropagation through time (BPTT)

BPTT

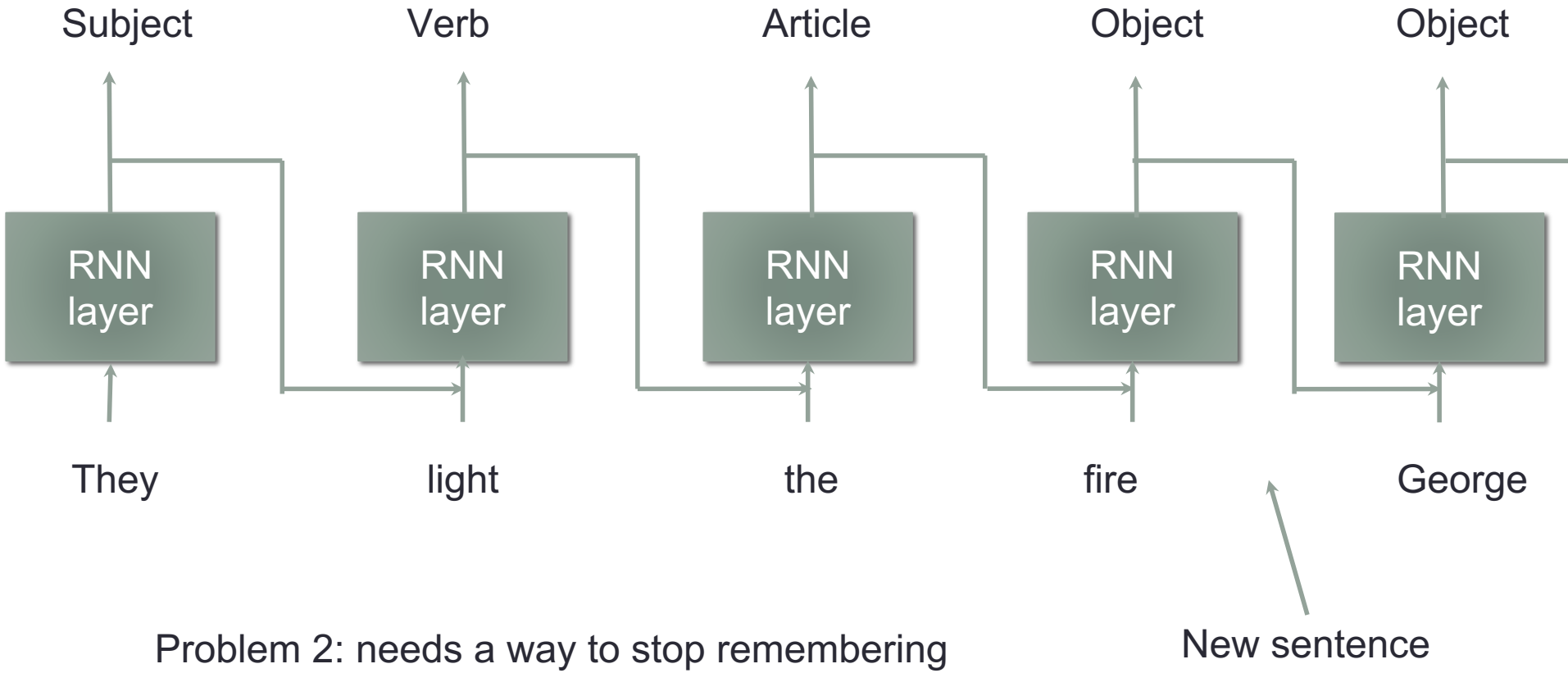
- Backward Computation graph



W1 <- W1 + G11 + G12 + G13 + G14

Problem 1: cannot deal with infinitely long recurrent
Gradient explosion, vanishing gradient

Recurrent neural network (RNN)

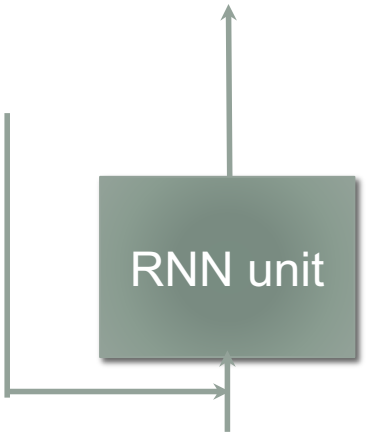


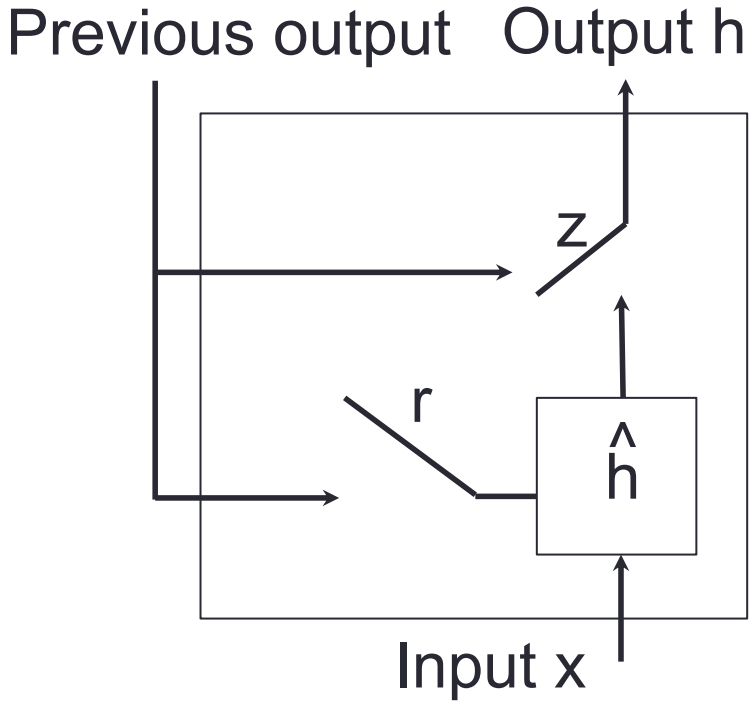
Problem 2: needs a way to stop remembering

Can the network learn when to start and stop remembering things?

Gated Recurrent Unit (GRU)

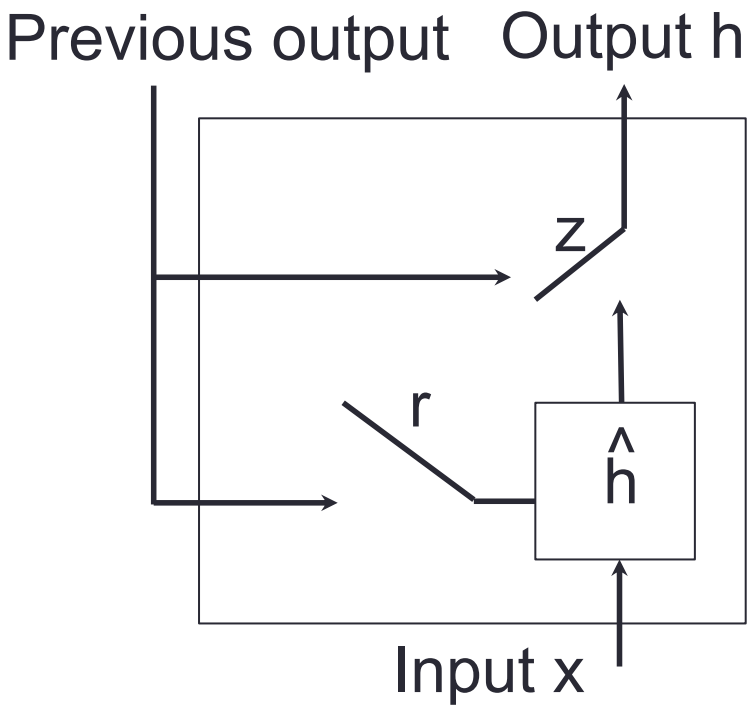
- Forms a Gated Recurrent Neural Networks (GRNN)
- Add gates that can choose to reset (r) or update (z)

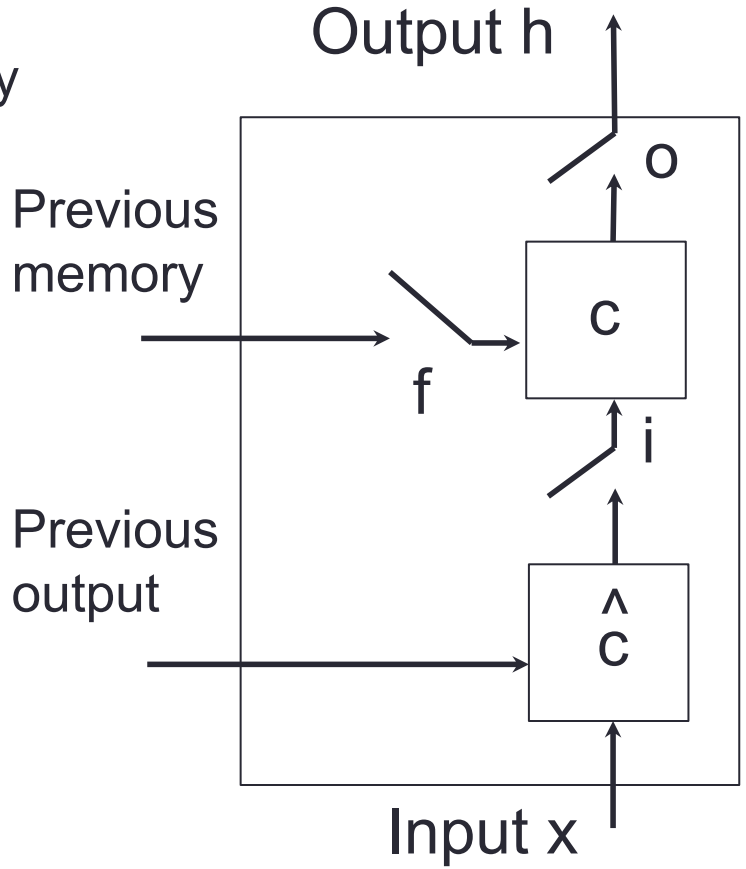




Long Short-Term Memory (LSTM)

- Have 3 gates, forget (f), input (i), output (o)
- Has an explicit memory cell (c)
  - Does not have to output the memory



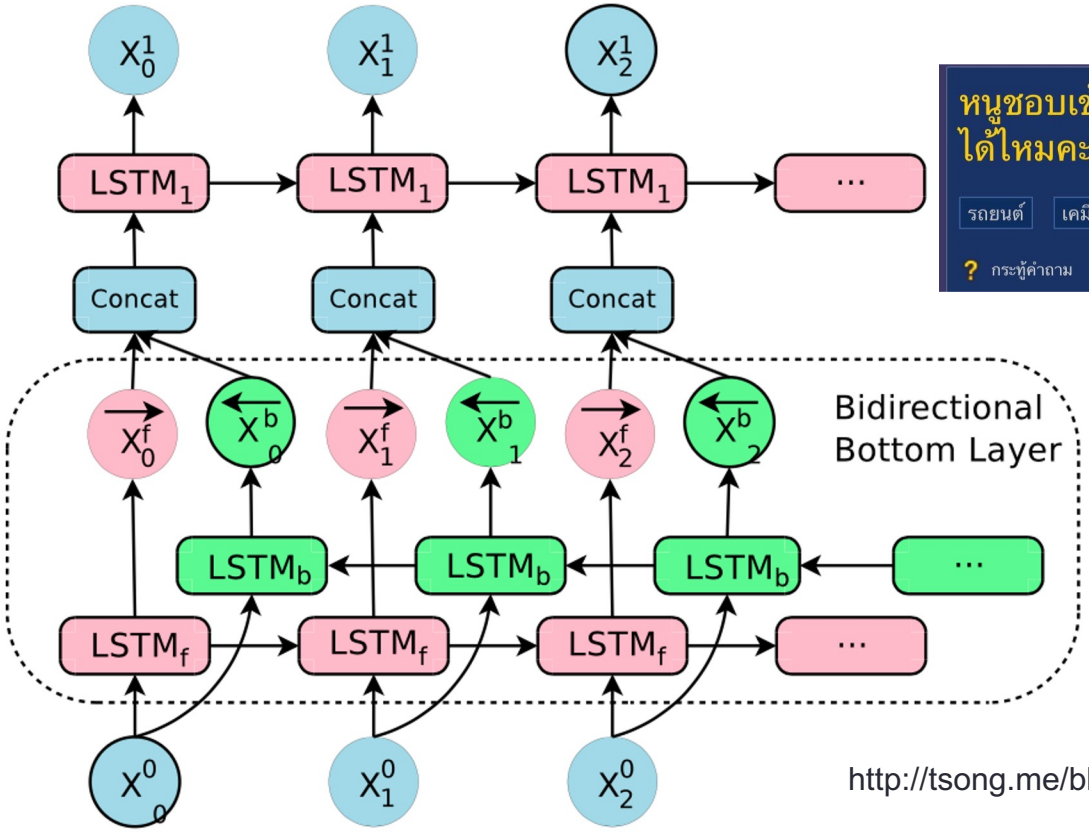


GRU vs LSTM

- GRU and LSTM offers the same performance with large dataset
  - GRU better for smaller dataset (less parameters)
  - GRU faster to train and faster runtime (smaller model)
- Use GRUs!

Bi-directional LSTM

- The previous GRU/LSTM only goes backward in time (uni-directional)
- Most of the time information from the future is useful for predicting the current output



http://tsong.me/blog/google-nmt/

Embeddings

- A way to encode information to a lower dimensional space
  - PCA
  - We learn about this lower dimensional space through data

One hot encoding

- Categorical representation is usually represented by one hot encoding

- Categorical representations examples:
  - Words in a vocabulary, characters in Thai language

Apple -> 1 -> [1, 0, 0, 0, …]

Bird -> 2 -> [0, 1, 0, 0, …]

Cat -> 3 -> [0, 0, 1, 0, …]

- Sparse representation
  - Sparse means most dimension are zero

One hot encoding

- Sparse – but lots of dimension
  - Curse of dimensionality
- Does not represent meaning.

Apple -> 1 -> [1, 0, 0, 0, …]

Bird -> 2 -> [0, 1, 0, 0, …]

Cat -> 3 -> [0, 0, 1, 0, …]

|Apple – Bird| = |Bird – Cat|

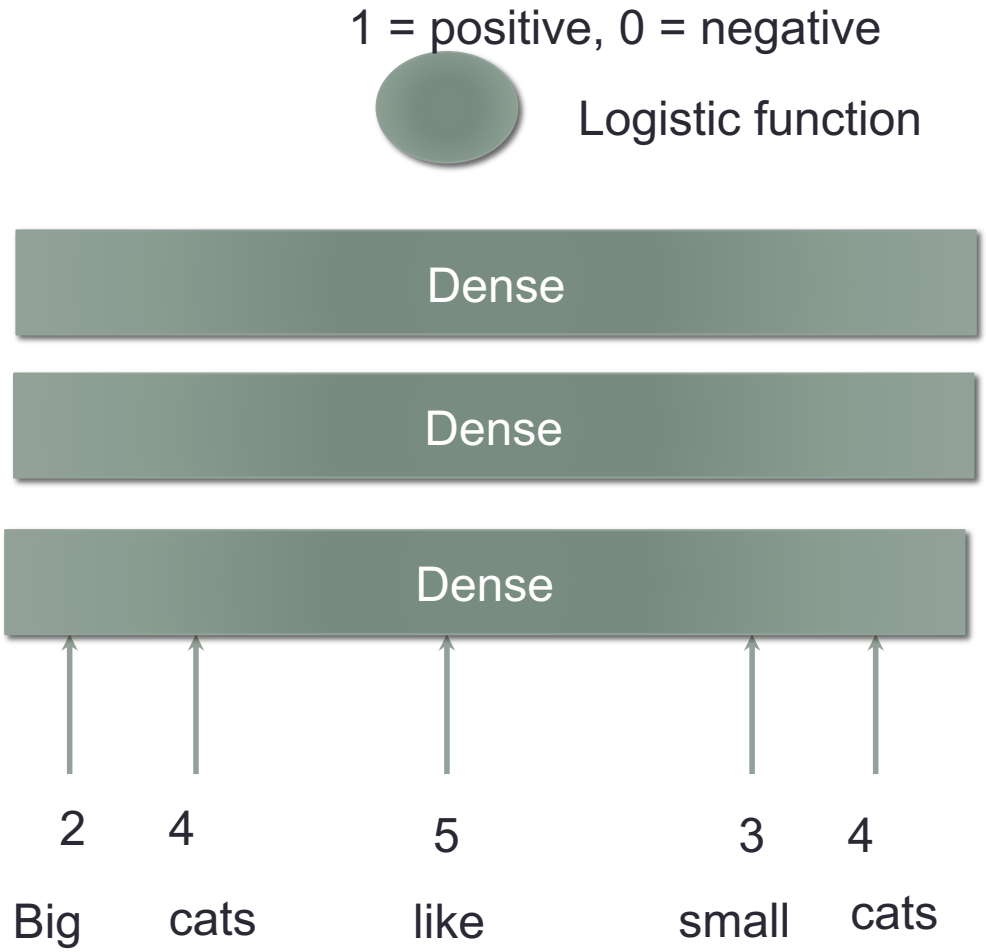
Dense representation

- We can encode sparse representation into a lower dimensional space
  - F: R^N -> R^M, where N>M

Apple -> 1 -> [1, 0, 0, 0, …] -> [2.3, 1.2]
Bird -> 2 -> [0, 1, 0, 0, …] -> [-1.0, 2.4]
Cat -> 3 -> [0, 0, 1, 0, …] -> [-3.0, 4.0]

- We can do this by using an embedding layer

Sentiment analysis with fully connected networks

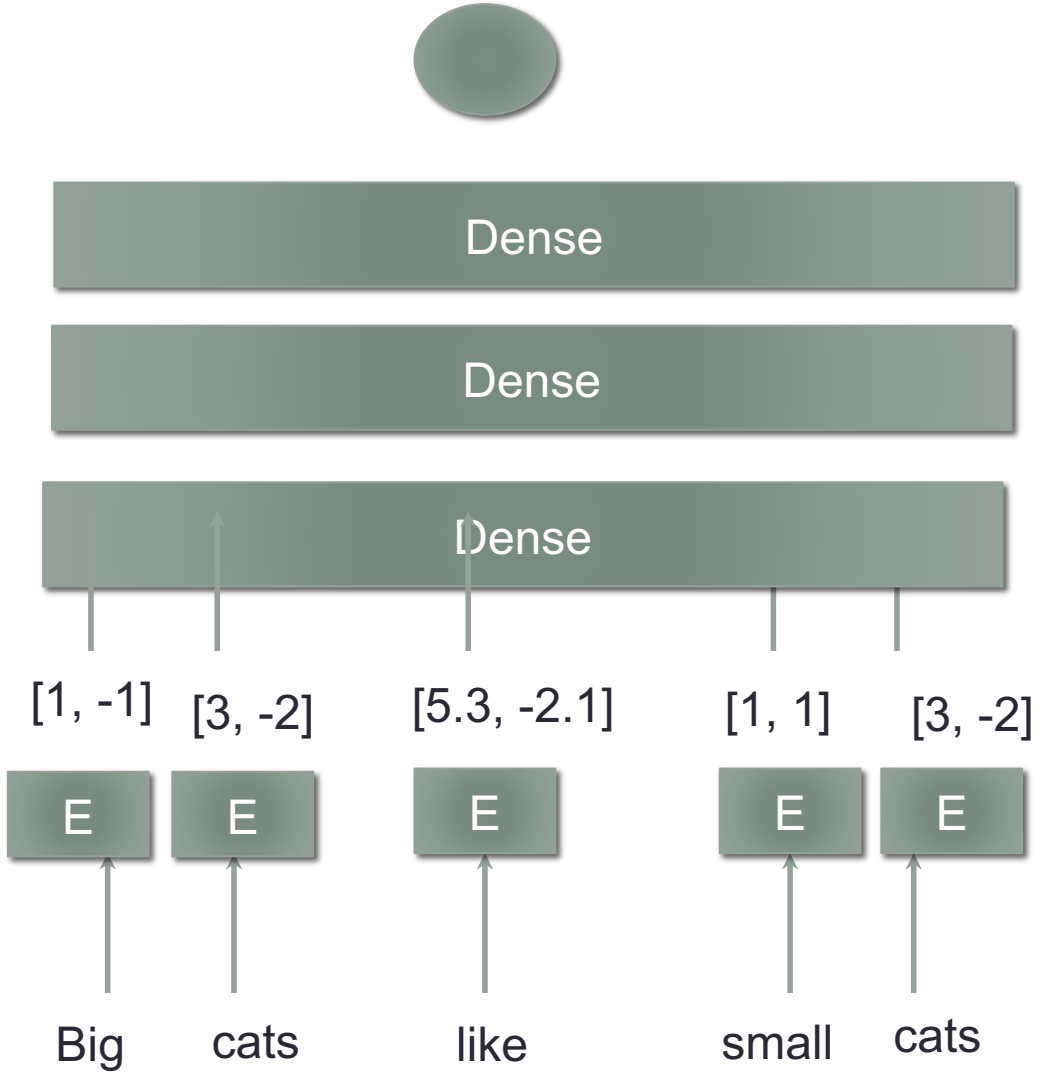


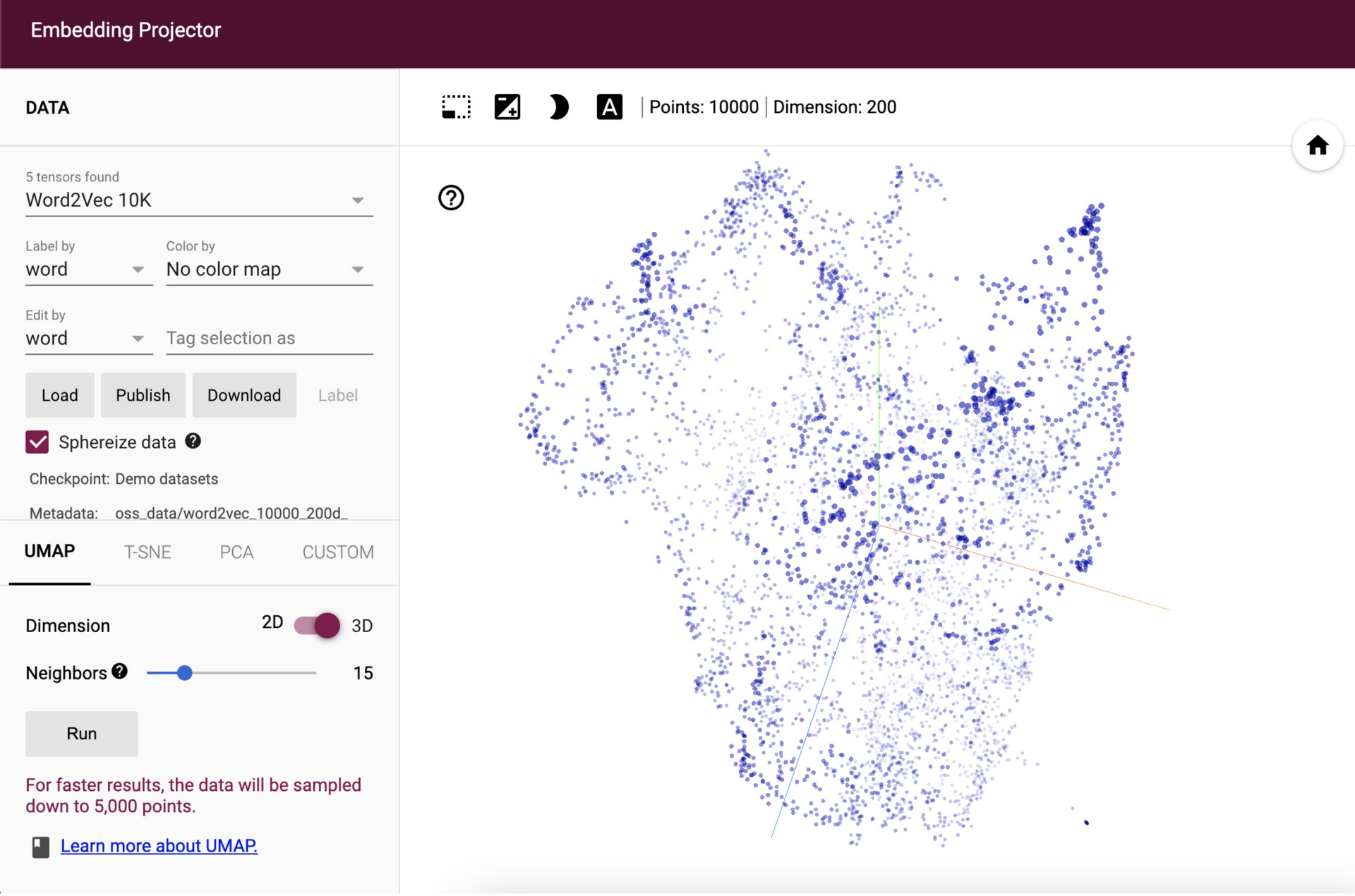
Adding embedding layer

Embedding layer shares the same weights

Parameter sharing!

More on embeddings in the next two lectures!




Embedding Projector
DATA
Points: 10000 | Dimension: 200
5 tensors found
Word2Vec 10K
Label by
word
Color by
No color map
Edit by
word
Tag selection as
Load
Publish
Download
Label
Sphereize data
Checkpoint: Demo datasets
Metadata: oss_data/word2vec_10000_200d_
UMAP
T-SNE
PCA
CUSTOM
Dimension
2D
3D
Neighbors
15
Run
For faster results, the data will be sampled down to 5,000 points.
Learn more about UMAP.

Embedding vs PCA

- PCA – unsupervised method
- Embedding – can be trained supervised
  - Embedding should be superior with the task
- PCA – linear
- Embedding – potentially non-linear
  - Should be more powerful

- You can learn an embedding on one task (with lots of data) and use it on another task
  - Embedding learns meaningful feature representations

Summary

- Layers
  - Dense/fully connected
  - Softmax
  - Nonlinearity
  - CNN, pooling
  - LSTM, GRU
  - Embedding
  - Dropout
  - Batchnorm
- Loss
  - training loss vs validation loss
  - Cross entropy vs MSE
- Optimizers
  - Backprop (autograd)
  - SGD, Adam, AdamW
- Learning rate scheduling

Appendix

Adam & AdamW

Image from https://stackoverflow.com/questions/64621585/adamw-and-adam-with-weight-decay

Adam vs AdamW

Adam

for $t = 1$ **to** ... **do**

  if *maximize*:

    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$

  else

    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

  if $\lambda \neq 0$

    $g_t \leftarrow g_t + \lambda\theta_{t-1}$

  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$

  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$

  if *amsgrad*

    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$

    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$

  else

    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

1. g is the gradient

2. g is augmented with weight decay term

3. m (momentum) is the moving average of g

4. v (variance) is the moving average of the gradient size

5. v and m are biased correct because moving average will overemphasize the starting time steps

Amsgrad is a tweak from Adam

AdamW

for $t = 1$ **to** ... **do**

  if *maximize*:

    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$

  else

    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$

  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$

  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$

  if *amsgrad*

    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$

    $\theta_t \leftarrow \theta_t - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$

  else

    $\theta_t \leftarrow \theta_t - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

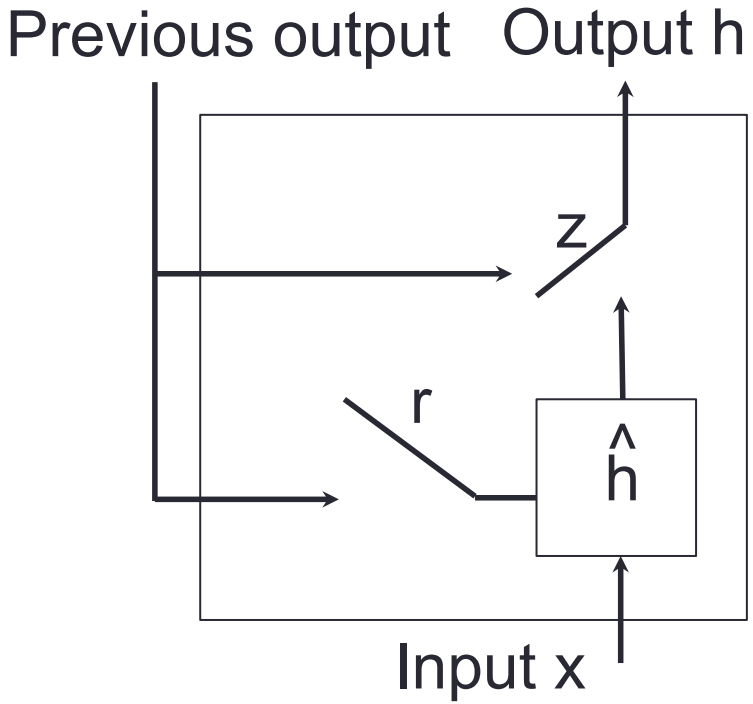
In AdamW weight decay term is used to change the weights directly, so it does not effect the momentum

6. The learning rate is normalized by the size of the gradient

Appendix

GRU & LSTM equations

Gated Recurrent Unit (GRU)

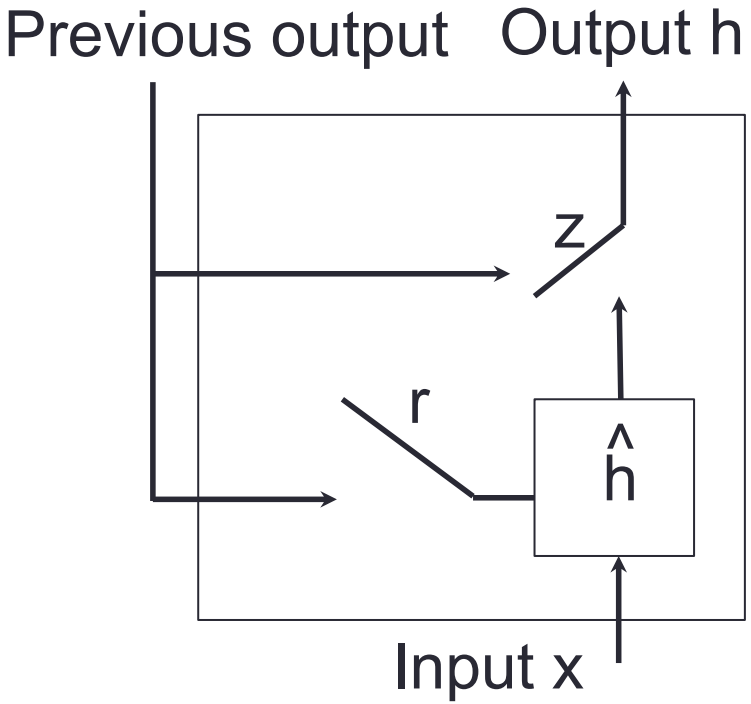


Neuron index

$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \hat{h}_t^j$$

time index

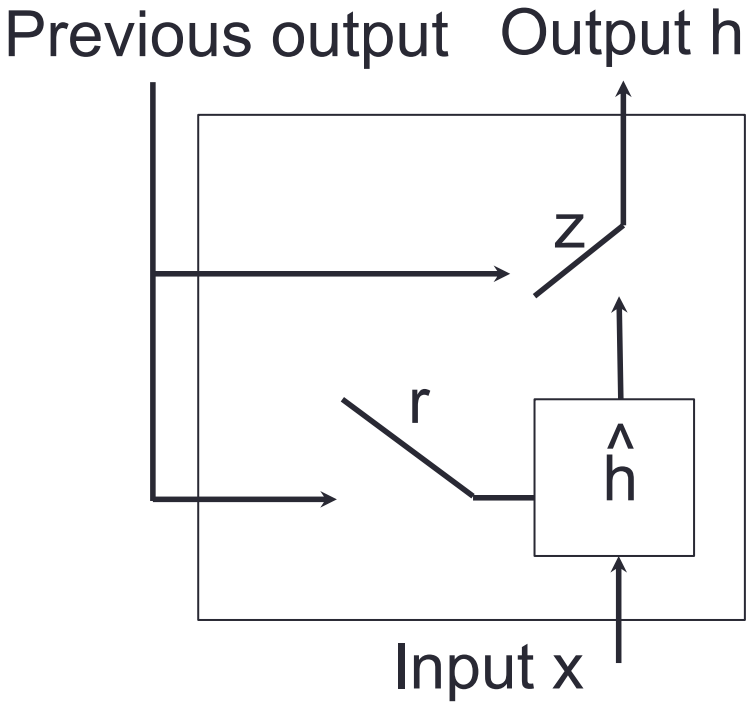
Gated Recurrent Unit (GRU)



$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \hat{h}_t^j$$

One GRU neuron output (scalar)

Gated Recurrent Unit (GRU)



$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

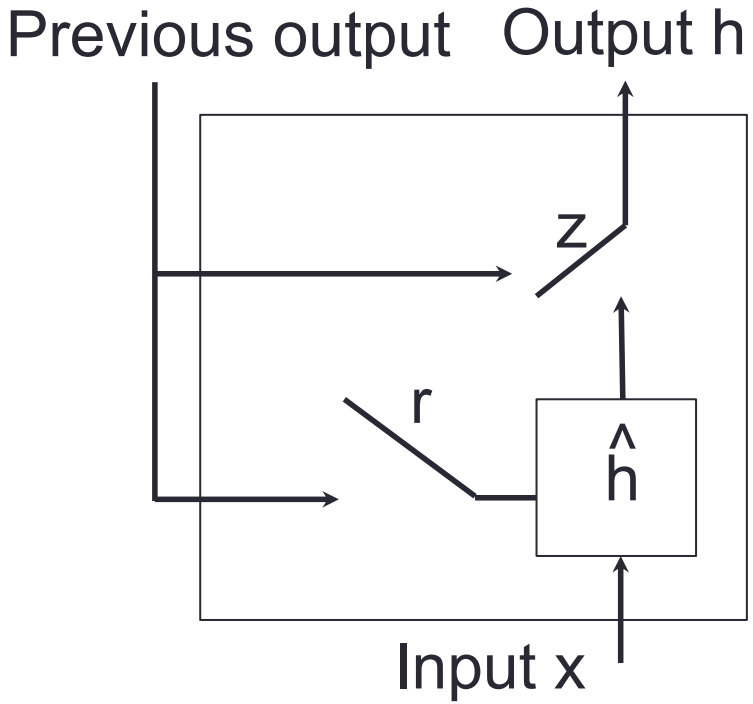
Element-wise product

Linear transform with matrix multiply

Vector (each value from each GRU unit in the previous layer)

$x_t^j = h_t^j$

Gated Recurrent Unit (GRU)



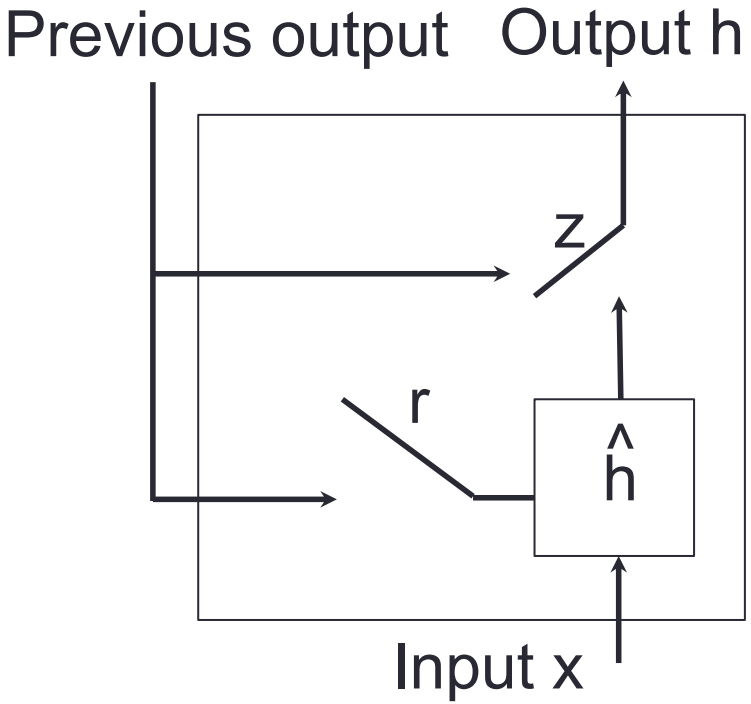
$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

Takes the j-th element

Bounds the output

Gated Recurrent Unit (GRU)



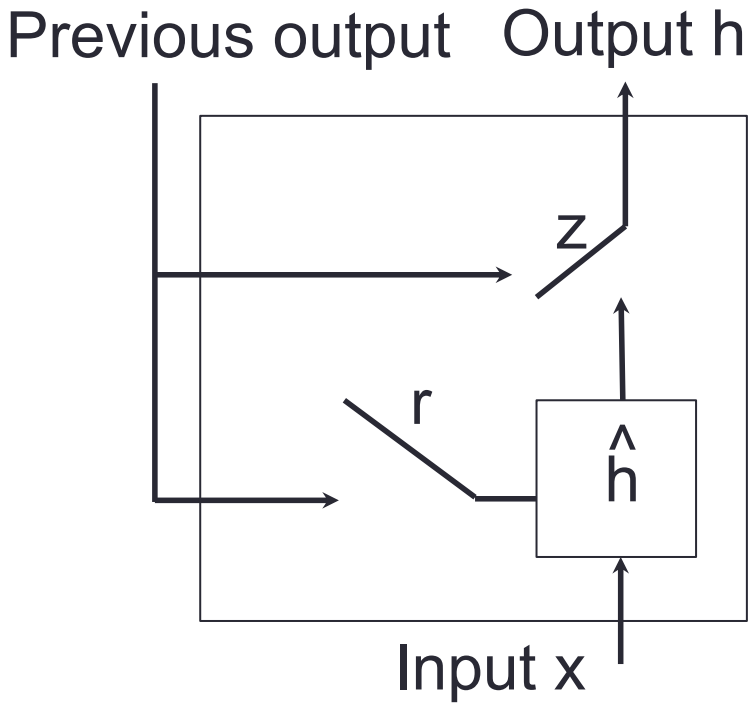
$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

$$z_t^j = sigmoid^j(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1})$$

Indicates a different set of weights

Gated Recurrent Unit (GRU)



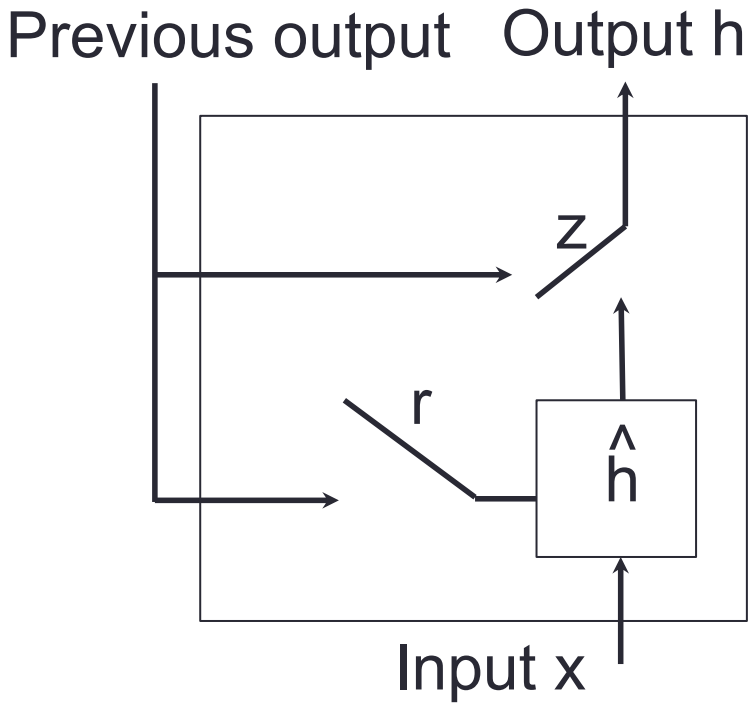
$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \hat{h}_t^j$$

$$\hat{h}_t^j = tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

$$z_t^j = sigmoid^j(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1})$$

Bounds the output to 0 to 1 for interpolation

Gated Recurrent Unit (GRU)



$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \hat{h}_t^j$$

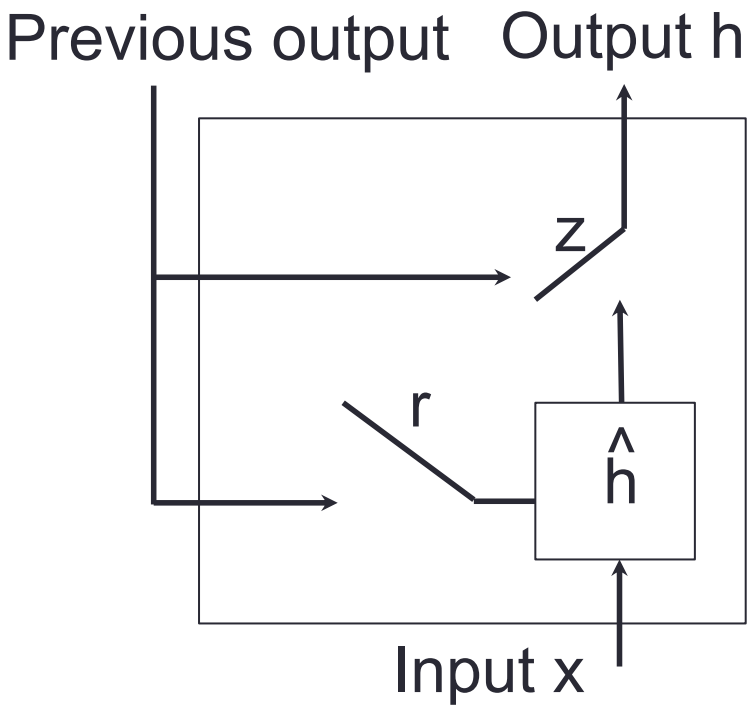
$$\hat{h}_t^j = tanh^j(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}))$$

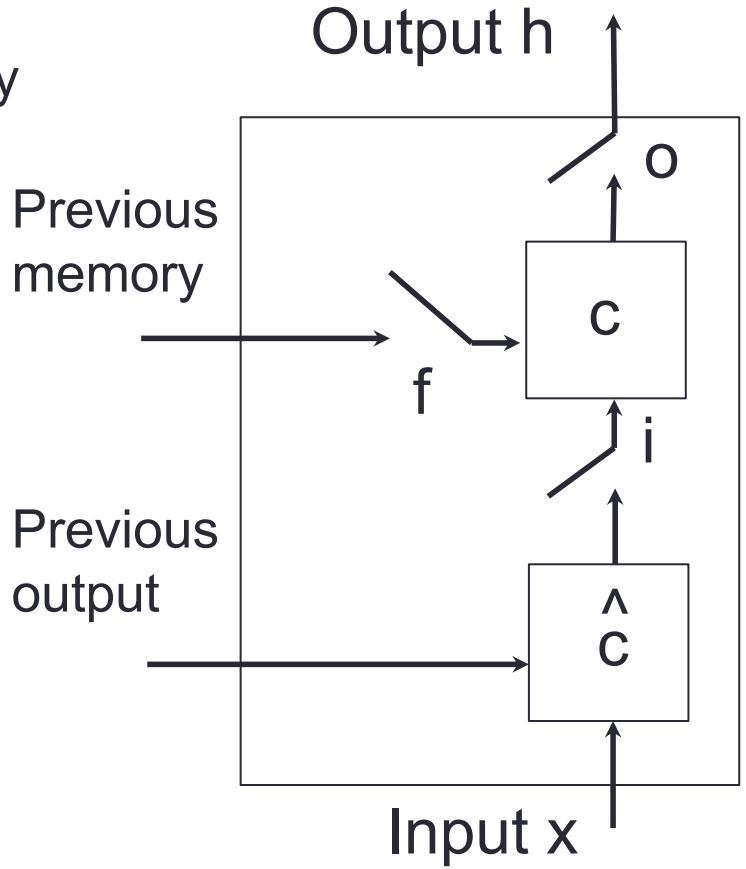
$$z_t^j = sigmoid^j(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1})$$

$$r_t^j = sigmoid^j(W_r\mathbf{x}_t + U_r\mathbf{h}_{t-1})$$

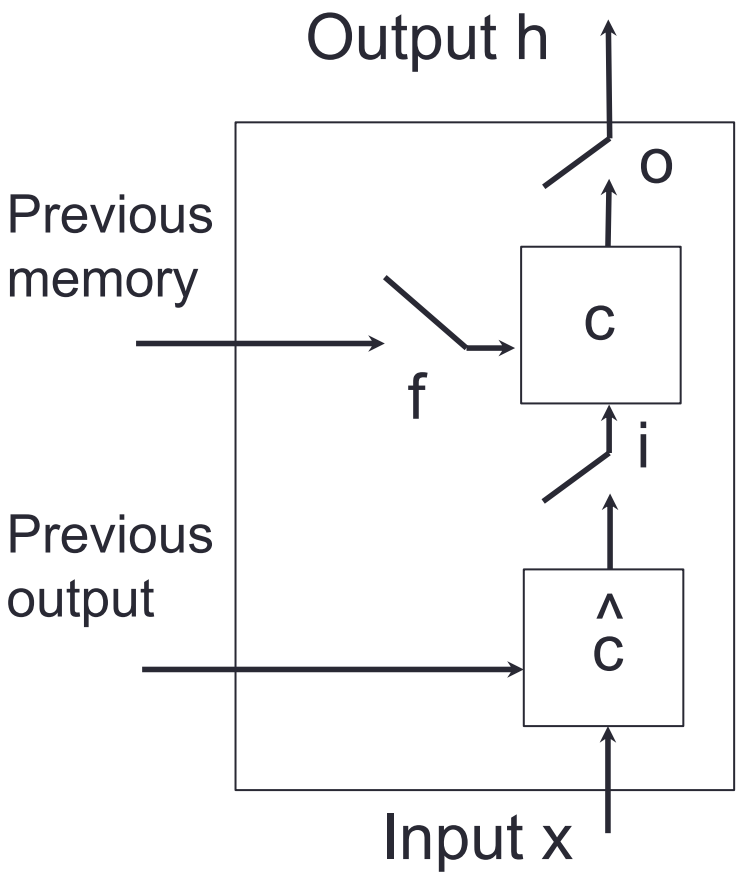
Long Short-Term Memory (LSTM)

- Have 3 gates, forget (f), input (i), output (o)
- Has an explicit memory cell (c)
  - Does not have to output the memory





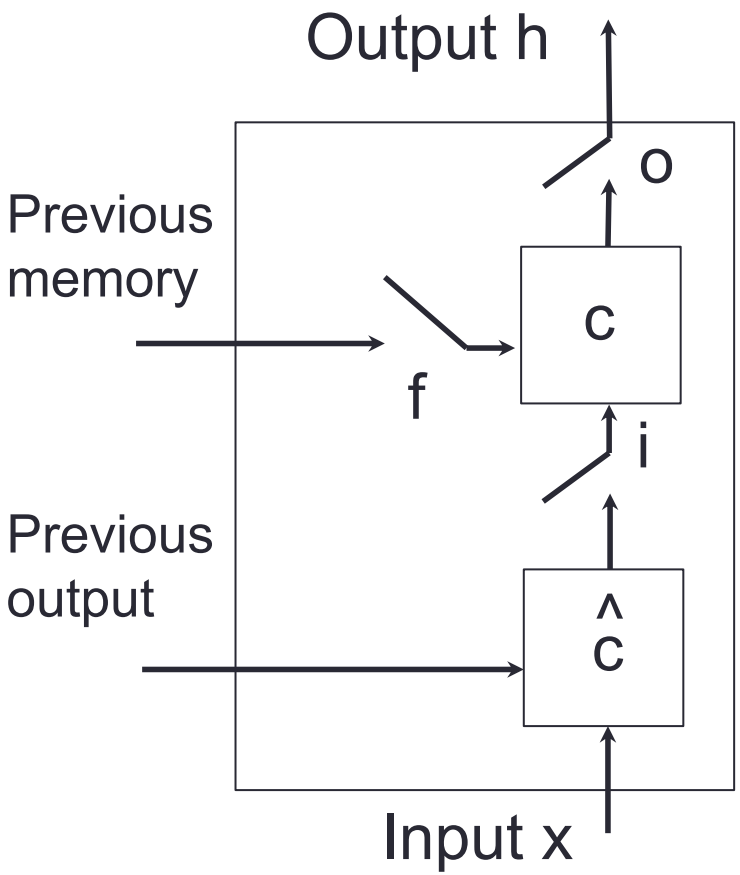
Long Short-Term Memory (LSTM)



$$i_t^j = F^j(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c_{t-1}})$$
$$o_t^j = F^j(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c_t})$$
$$f_t^j = F^j(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_j \mathbf{c_{t-1}})$$

Contribution from memory “Peephole connection”
Vs are diagonal matrices (Each cell can only see its own memory)

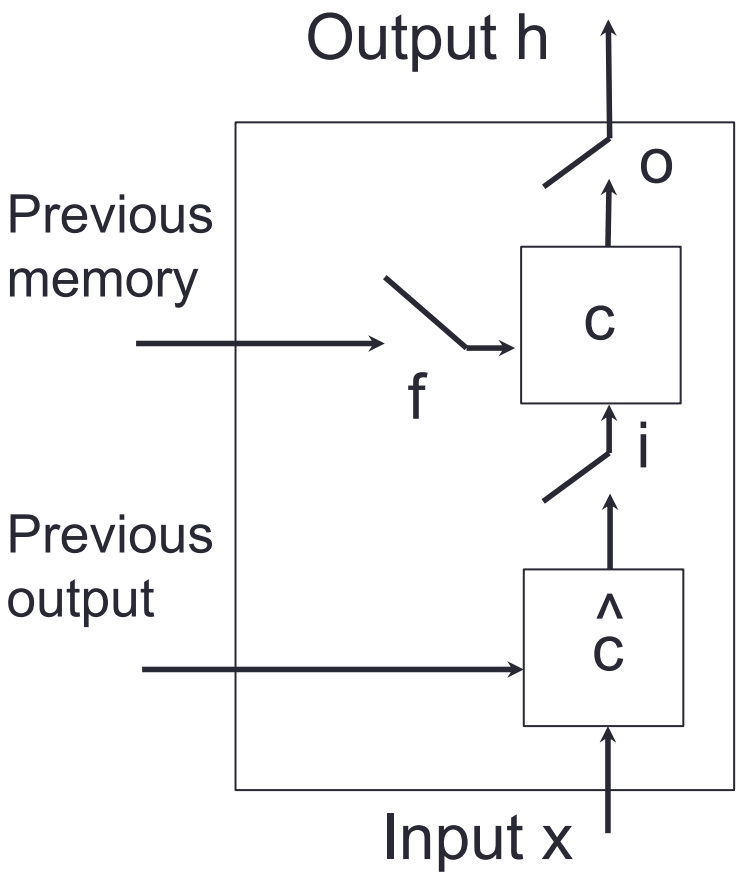
Long Short-Term Memory (LSTM)



$$i_t^j = F^j(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c_{t-1}})$$
$$o_t^j = F^j(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c_t})$$
$$f_t^j = F^j(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_j \mathbf{c_{t-1}})$$

Sigmoid function to bounds to 0-1

Long Short-Term Memory (LSTM)



$$i_t^j = F^j(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + V_i \mathbf{c_{t-1}})$$

$$o_t^j = F^j(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + V_o \mathbf{c_t})$$

$$f_t^j = F^j(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + V_j \mathbf{c_{t-1}})$$

$$h_t^j = o_t^j tanh(c_t^j)$$

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \hat{c}_t^j$$

$$\hat{c}_t^j = tanh^j(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1})$$