



Language
Technologies
Institute

Carnegie
Mellon
University

Advanced Multimodal Machine Learning

Lecture 3.1: Optimization and Convolutional Neural Networks

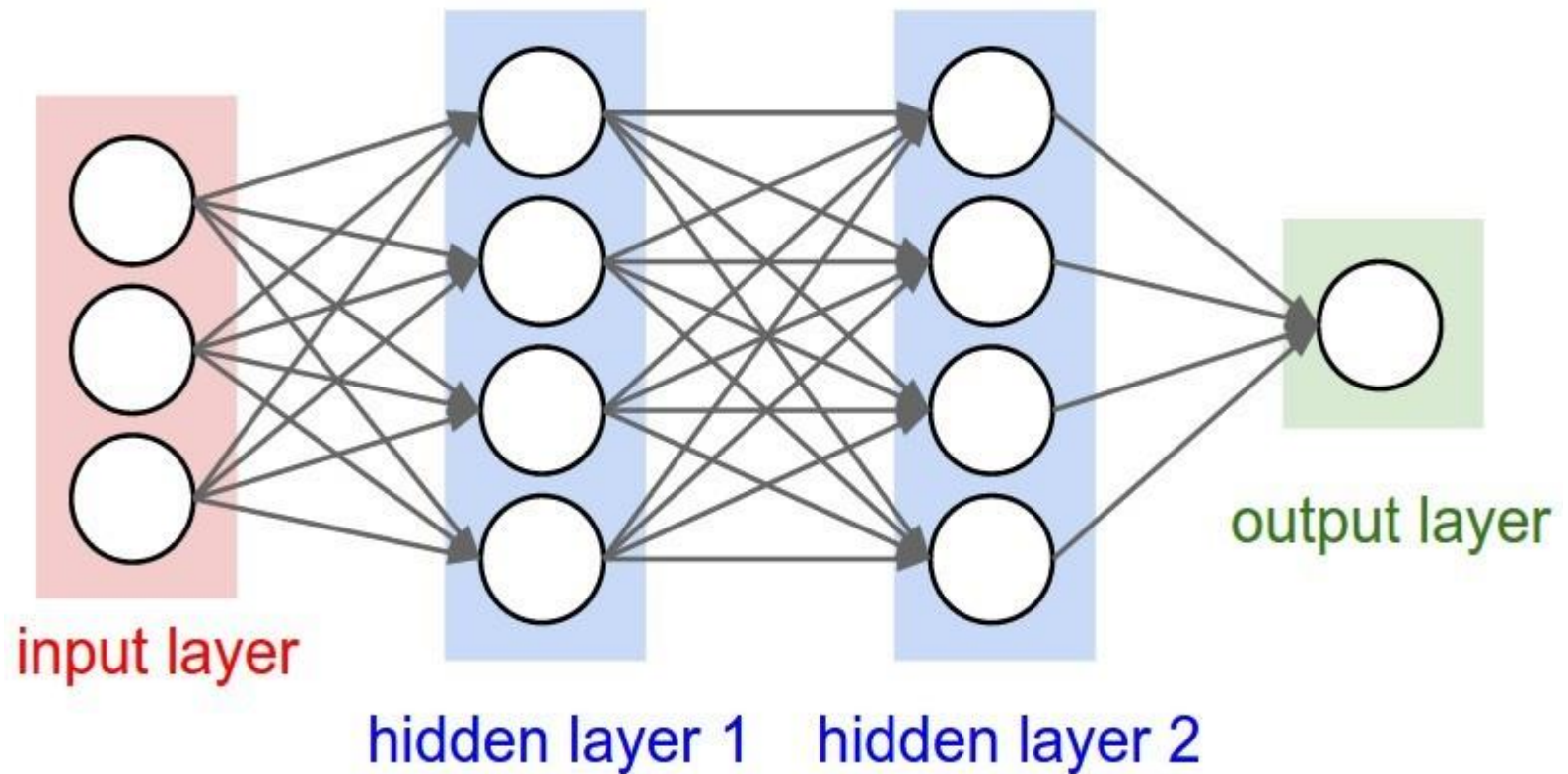
Louis-Philippe Morency
Tadas Baltrušaitis

Lecture Objectives

- Components of a neural network
- Learning the model
 - Optimization
 - Gradient computation
- Convolutional Neural networks
 - Convolution and pooling
 - Architectures
 - Training tricks

Neural Networks – recap

- Reminder of a Multi Layer Perceptron



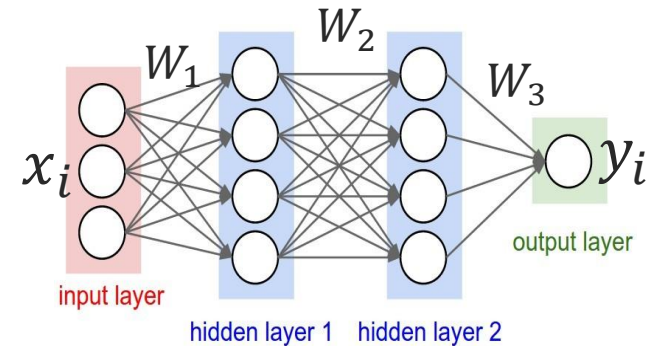
Neural Networks – recap

- Individual layers ($W = [W_1, b_1, W_2, b_2 \dots]$)

$$f_{1;W_1}(x) = \sigma(W_1x + b_1)$$

$$f_{2;W_2}(x) = \sigma(W_2x + b_2)$$

$$f_{3;W_3}(x) = \sigma(W_3x + b_3)$$



- The whole score function for a two hidden layer network

$$y_i = f(x_i, W) = f_{3;W_3}(f_{2;W_2}(f_{1;W_1}(x_i)))$$

Neural Networks inference and learning

- Inference (Testing)
 - Use the score function ($y = f(\mathbf{x}; W)$)
 - Have a trained model (parameters W)
- Learning model parameters (Training)
 - Loss function (L)
 - Gradient – will talk about today
 - Optimization – will talk about today



Loss function (1)

- Loss function is often made up of three parts

$$L = L_{data} + \lambda_1 L_{regularization} + \lambda_2 L_{constraints}$$

- Data term

- How well our model is explaining/predicting training data (e.g. cross-entropy loss, Euclidean loss)

$$\sum_i L_i = - \sum_i \log \left(\frac{e^{f_{y_i}(x_i; W)}}{\sum_j e^{f_j(x_i; W)}} \right)$$

$$\sum_i L_i = \sum_i (y_i - f(x_i, W))^2$$



Loss function (2)

- Loss function is often made up of three parts

$$L = L_{data} + \lambda_1 L_{regularization} + \lambda_2 L_{constraints}$$

- Regularization/Smoothness term
 - Prevent the model from becoming too complex
 - e.g. $\|W\|_2$ for parameters smoothness
 - e.g. $\|W\|_1$ for parameter sparsity
- λ_1 is a hyper-parameter
- Optional, but almost never omitted



Loss function (3)

- Loss function is often made up of three parts

$$L = L_{data} + \lambda_1 L_{regularization} + \lambda_2 L_{constraints}$$

- Additional constraints
 - Optional and not always used
 - Help with certain models (e.g. coordinated multimodal representation)
 - e.g. Triplet loss, hinge ranking loss, reconstruction loss
 - Will talk more during multimodal representation lecture



Learning model parameters

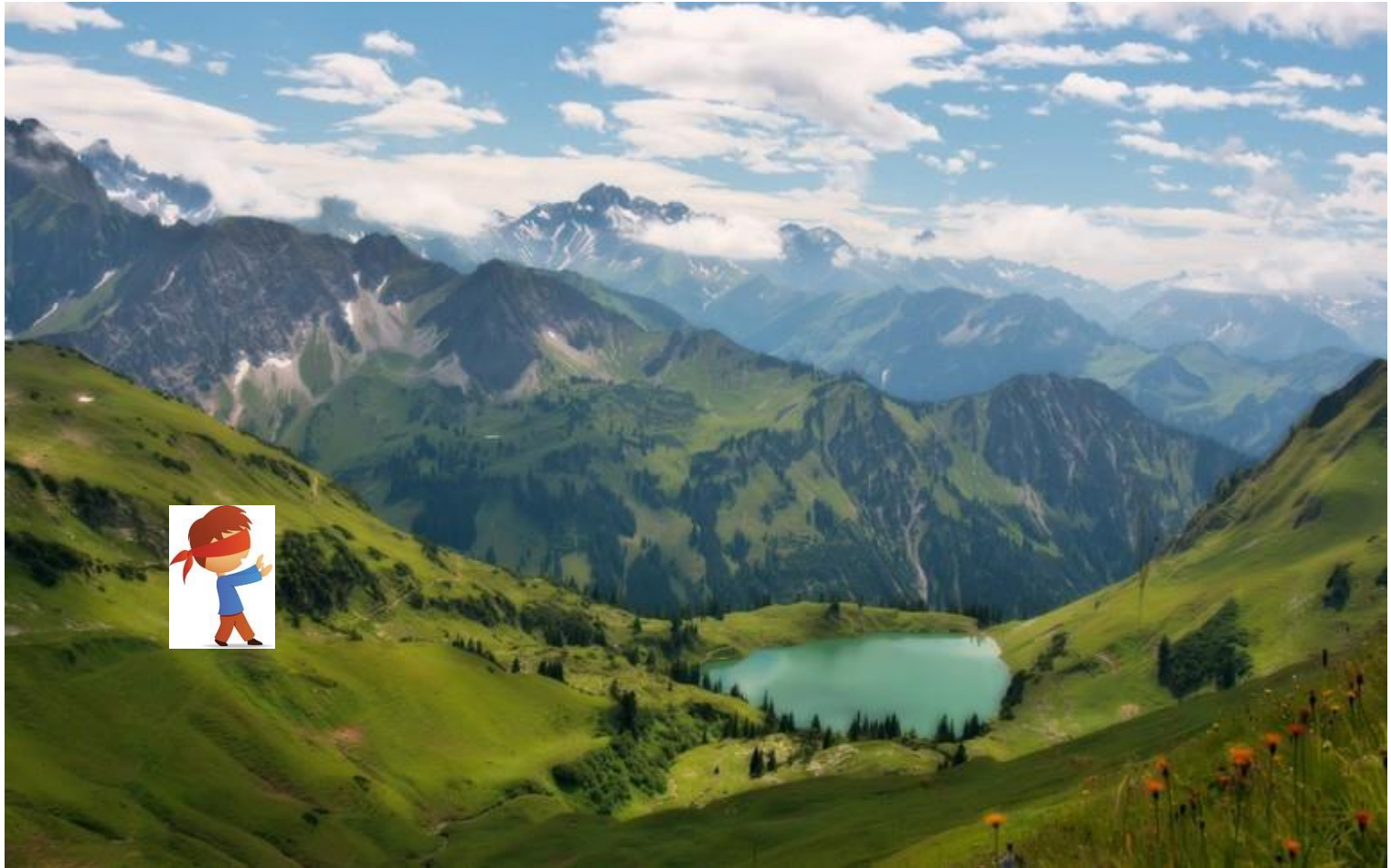


Learning model parameters

- We have our training data
 - $X = \{x_1, x_2, \dots, x_n\}$ (e.g. images, videos, text etc.)
 - $Y = \{y_1, y_2, \dots, y_n\}$ (labels)
 - Fixed
- We want to learn the W (weights and biases) that leads to best loss
$$\operatorname{argmin}_W [L(X, Y, W)]$$
- The notation means find W for which $L(X, Y, W)$ has the lowest value



Optimization



Optimizing a generic function

- We want to find a minimum of the loss function
- How do we do that?
 - Searching everywhere (global optimum) is computationally infeasible
 - We could search randomly from our starting point (mostly picked at random) and then refine the search region – impractical and not accurate
 - Instead we can follow the gradient



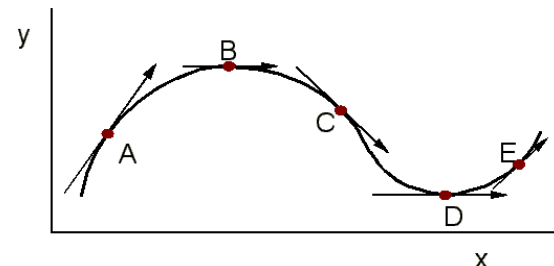
What is a gradient?

- Geometrically

- Points in the direction of the greatest rate of increase of the function and its magnitude is the slope of the graph in that direction

- More formally in 1D

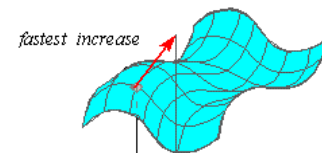
$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$



- In higher dimensions

$$\frac{\partial f}{\partial x_i}(a_1, \dots, a_n) = \lim_{h \rightarrow 0} \frac{f(a_1, \dots, a_i + h, \dots, a_n) - f(a_1, \dots, a_i, \dots, a_n)}{h}$$

- In multiple dimension, the **gradient** is the vector of (partial derivatives) and is called a **Jacobian**.



Numeric gradient

- Can set h to a very low number and compute:

$$\frac{df(x)}{dx} = \frac{f(x + h) - f(x)}{h}$$

- Slow and just an approximation
 - Need to compute score once (or even twice for central limit) for each parameter
 - Sensitive to choice of h
- h needs to be chosen as well - hyperparameter

Analytical gradient

- If we know the function and it is **differentiable**
 - Derivative/gradient is defined at every point in f
 - Sometimes use differentiable approximations
 - Some are locally differentiable
- Use Calculus (or Wikipedia)!
- Examples:

$$f(x) = \frac{1}{1 + e^{-x}}; \frac{df}{dx} = (1 - f(x))f(x)$$

$$f(x) = (x - y)^2; \frac{df}{dx} = 2(x - y)$$

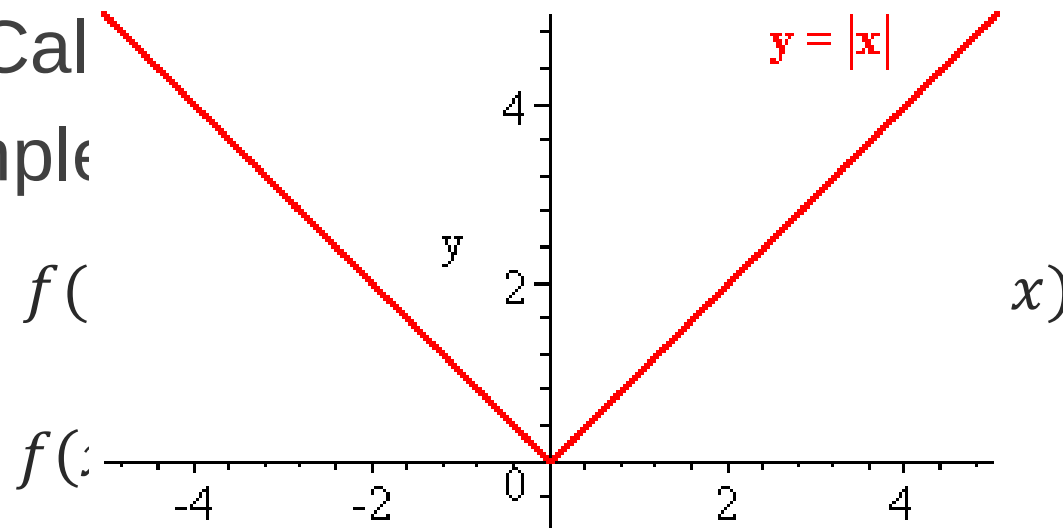


Analytical gradient

- If we know the function and it is **differentiable**
 - Derivative/gradient is defined at every point in f
 - Sometimes use differentiable approximations
 - Some are locally differentiable

- Use Calculus

- Example



Which one should we use?

- Numeric
 - Slow
 - Approximate
- Analytical
 - More error prone to implement (need to get the gradient right)
 - Can use automated tools to help – Theano, autograd, Matlab symbolic toolbox
- Have both, use analytical for speed but check using numeric
- [Why you should understand gradient](#)



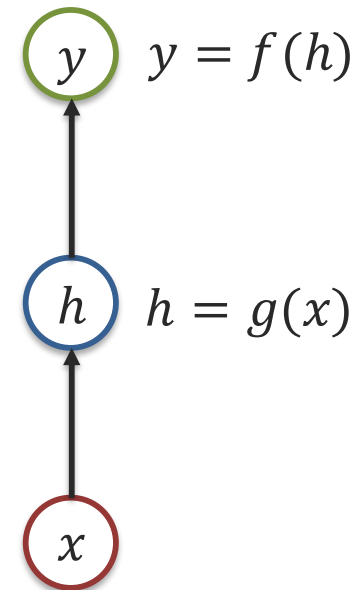
Neural Networks gradient



Gradient Computation

Chain rule:

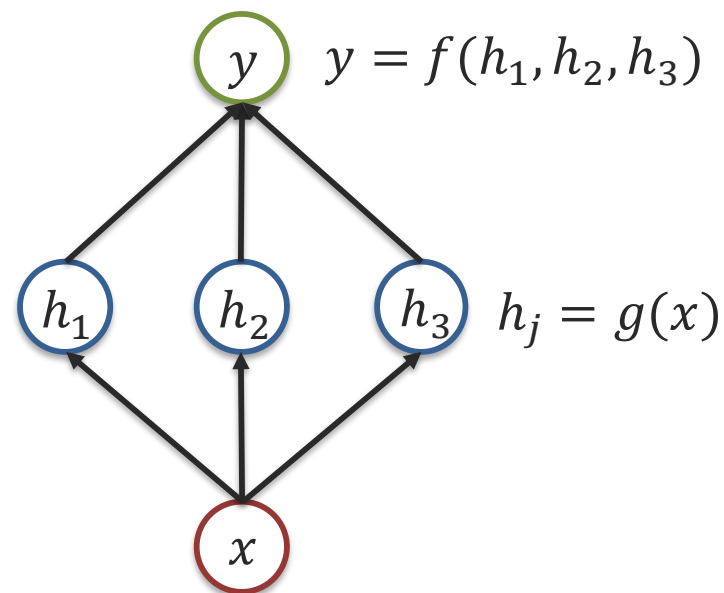
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial h} \frac{\partial h}{\partial x}$$



Optimization: Gradient Computation

Multiple-path chain rule:

$$\frac{\partial y}{\partial x} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x}$$



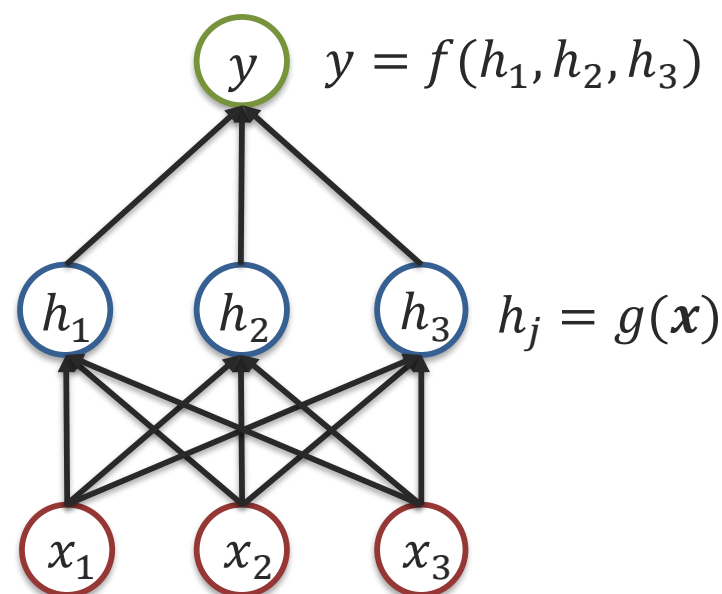
Optimization: Gradient Computation

Multiple-path chain rule:

$$\frac{\partial y}{\partial x_1} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x_1}$$

$$\frac{\partial y}{\partial x_2} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x_2}$$

$$\frac{\partial y}{\partial x_3} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x_3}$$



Optimization: Gradient Computation

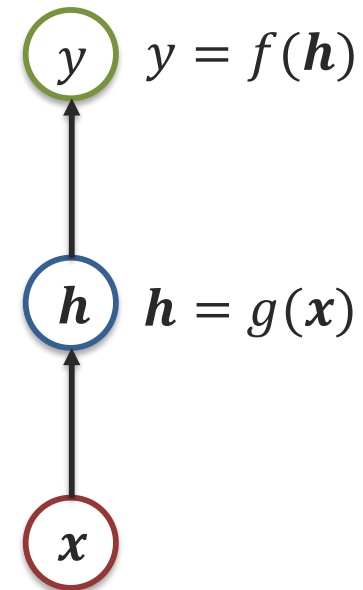
Vector representation:

Gradient $\nabla_x y = \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \frac{\partial y}{\partial x_3} \right]$

$\nabla_x y = \left(\frac{\partial \mathbf{h}}{\partial \mathbf{x}} \right)^T \nabla_h y$

“local” Jacobian
(matrix of size $|h| \times |x|$ computed using partial derivatives)

“backprop” Gradient



Backpropagation Algorithm (efficient gradient)

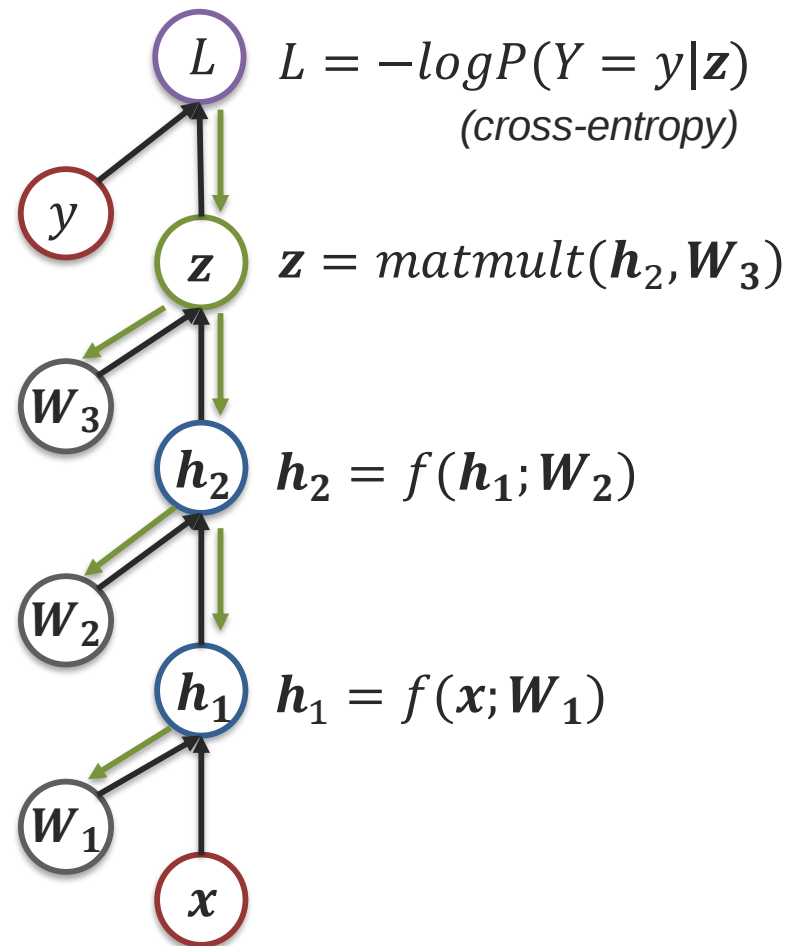
Forward pass

- Following the graph topology, compute value of each unit

Backpropagation pass

- Initialize output gradient = 1
- Compute “local” Jacobian matrix using values from forward pass
- Use the chain rule:

Gradient = “local” Jacobian $\times$
“backprop” gradient



Computational Graph: Multi-layer Feedforward Network

Computational unit:



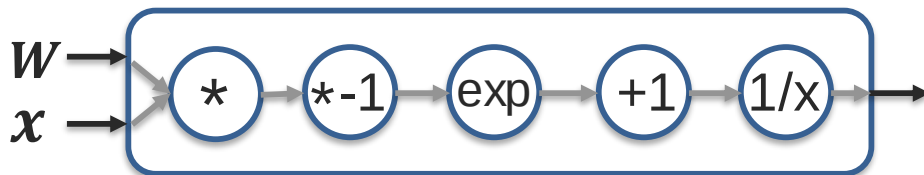
$$h = f(x; W)$$

- Multiple input
- One output
- Vector/tensor

▪ Sigmoid unit:

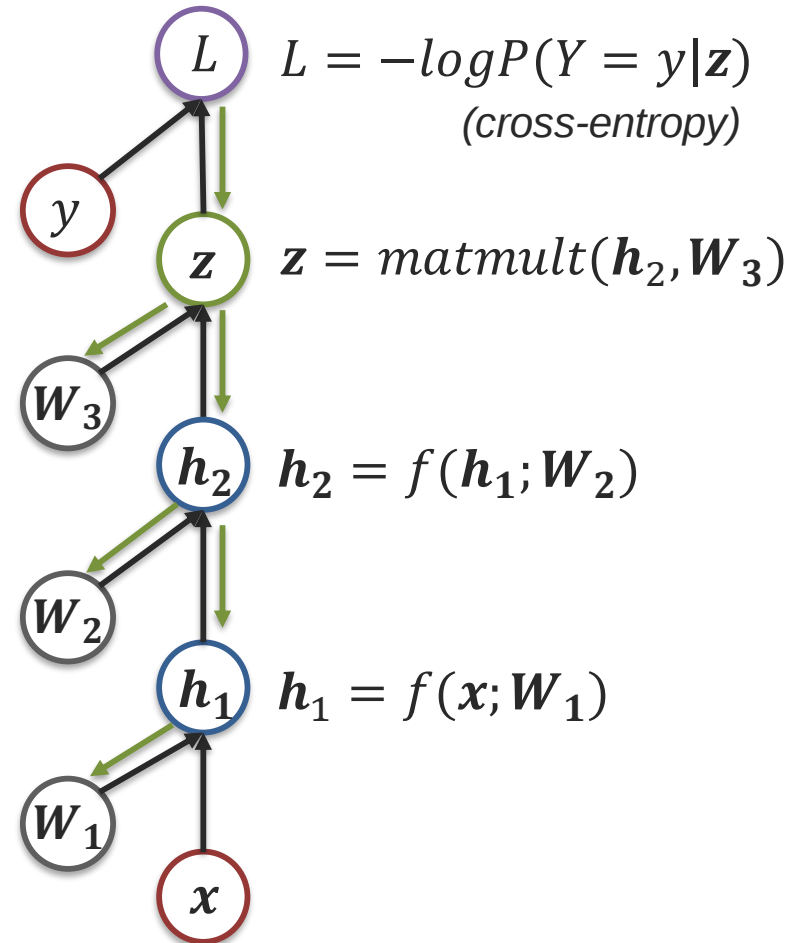


$$h_j = (1 + e^{-W_j x})^{-1}$$



Differentiable “unit” function!

(or close approximation to compute “local Jacobian”)



Gradient descent



How to follow the gradient

- Many methods for optimization
 - **Gradient Descent (actually the “simplest” one)**
 - Newton methods (use Hessian – second derivative)
 - Quasi-Newton (use approximate Hessian)
 - BFGS
 - LBFGS
 - Don’t require learning rates (fewer hyperparameters)
 - But, do not work with stochastic and batch methods so rarely used to train modern Neural Networks
- **All of them look at the gradient**
 - Very few non gradient based optimization methods

Parameter Update Strategies

Gradient descent:

$$\theta^{(t+1)} = \theta^t - \epsilon_k \nabla_{\theta} L$$

Annotations for the first equation:

- $\theta^{(t+1)}$: New model parameters
- θ^t : Previous parameters
- ϵ_k : Learning rate at iteration k
- $\nabla_{\theta} L$: Gradient of our loss function

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_{\tau}$$

Annotations for the second equation:

- ϵ_k : Learning rate at iteration k
- α : Decay
- ϵ_0 : Initial learning rate
- ϵ_{τ} : Decay learning rate linearly until iteration τ

- Extensions:
- Stochastic (“batch”)
 - with momentum
 - AdaGrad
 - RMSProp



Vanilla Gradient Descent

- Compute gradient with respect to loss and keep updating weights till convergence

```
while not converged:
```

```
    # compute gradients
```

```
    weights_grad = compute_gradient(loss_fun, data, weights)
```

```
    # perform parameter update
```

```
    weights += - step_size * weights_grad
```

```
    # (optionally update step size)
```



GD example



GD example



GD example



GD example



GD example

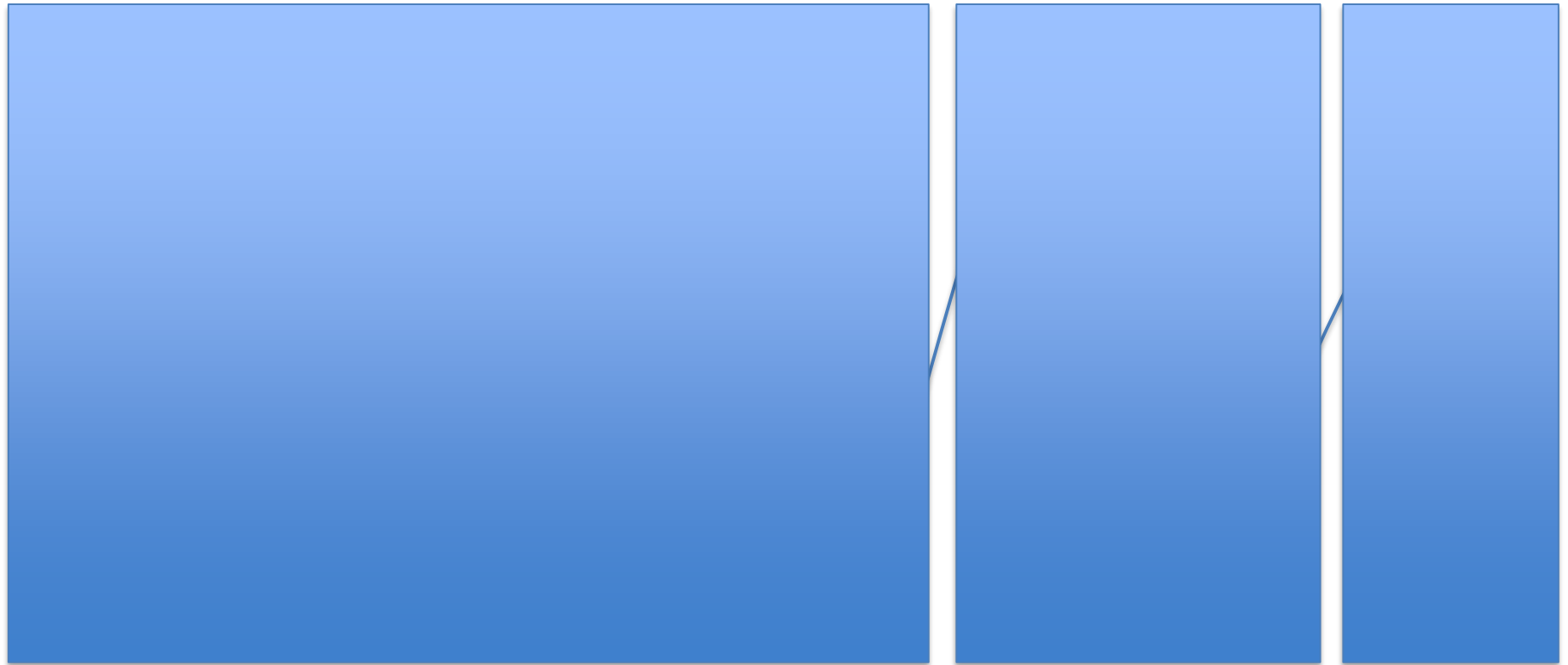
- Converged



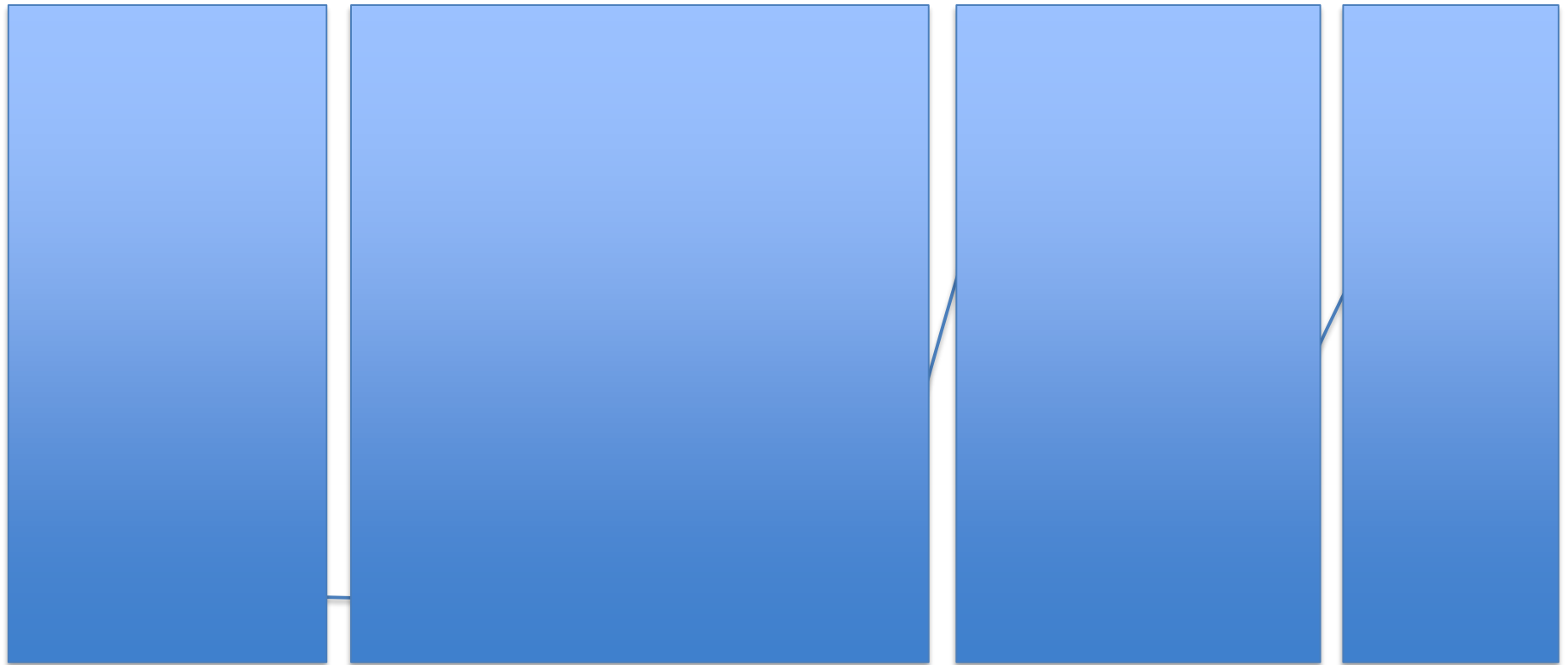
GD example 2



GD example 2

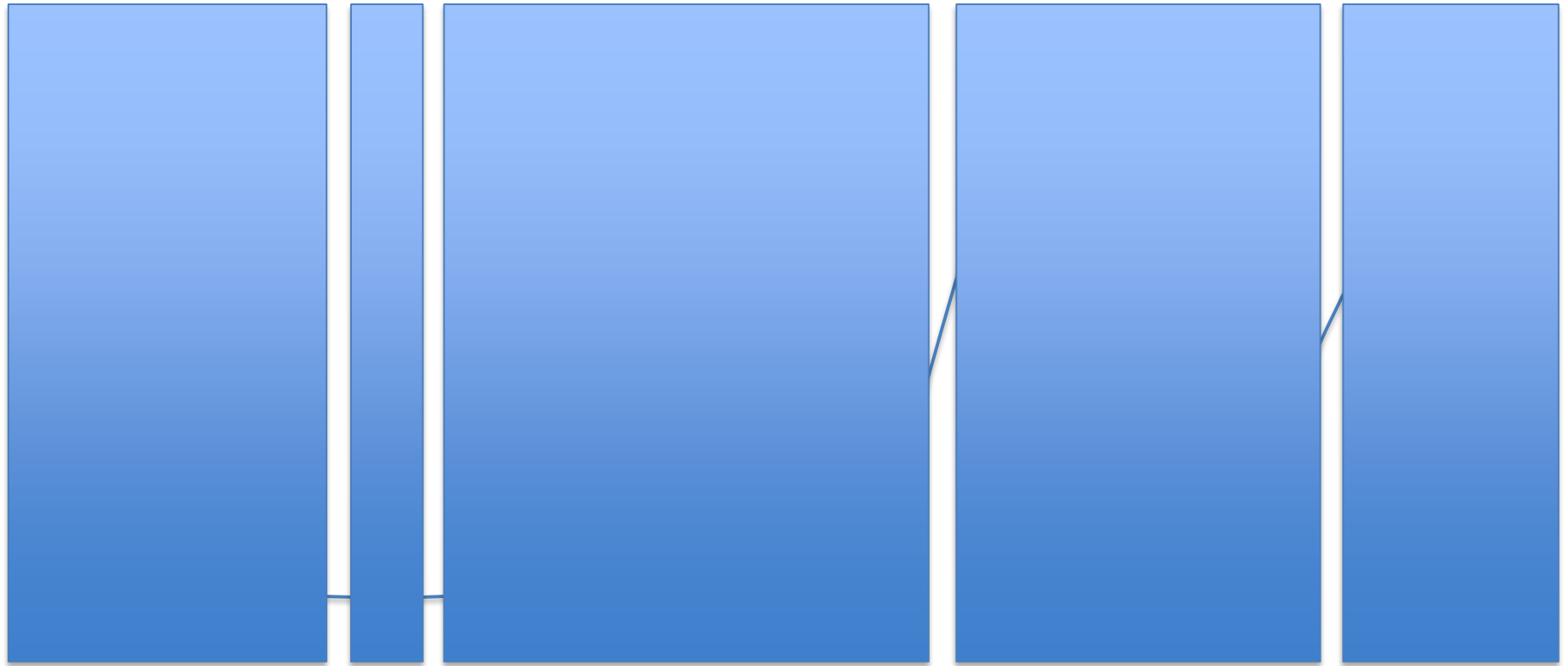


GD example 2



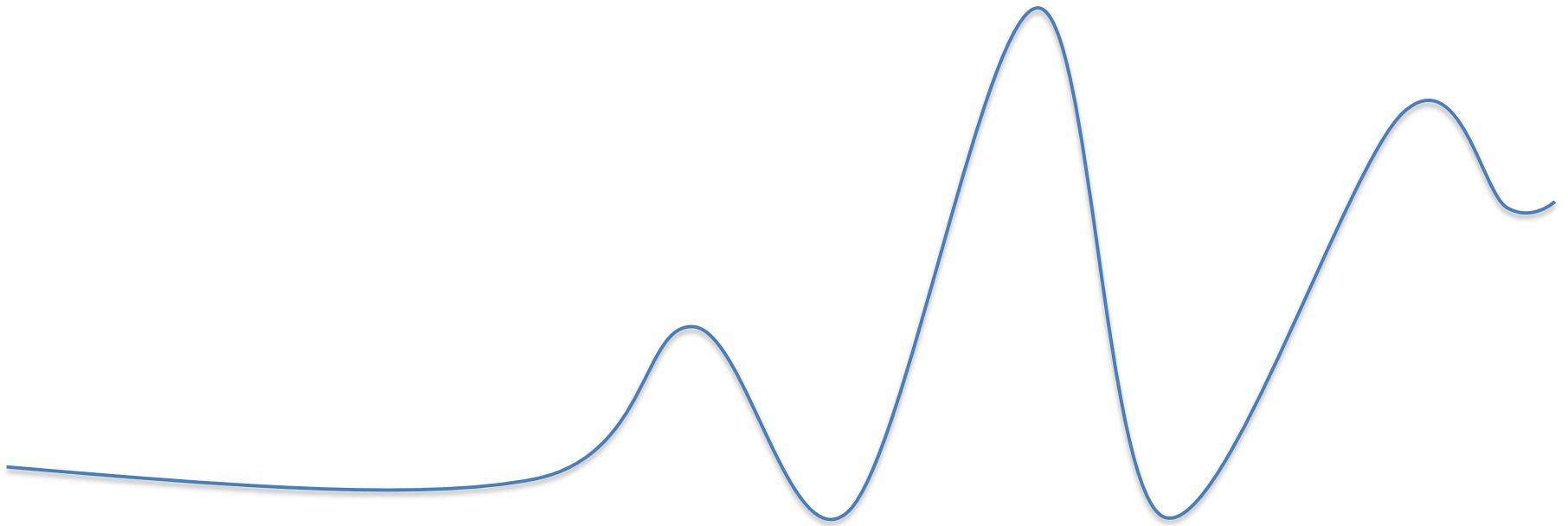
GD example 2

- Convergence reached



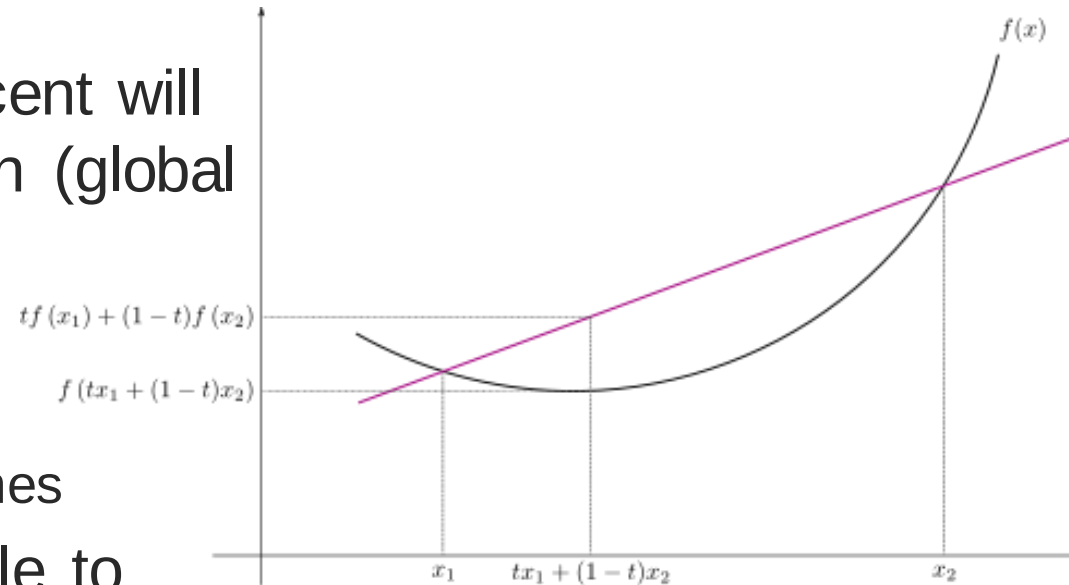
GD example

- We are looking at a potentially very complex surface through a pinhole and hope that we reach a **good enough** (not optimal) value



Convex vs. non-convex functions and local minima

- Convex – gradient descent will lead to a perfect solution (global optimum)
 - Logistic regression
 - Least squares models
 - Support vector machines
- Non-convex – impossible to guarantee that the solution is the best – will lead to local-minima
 - Neural networks
 - Various graphical models



Batch (stochastic) gradient descent

- Using all of data points might be tricky when computing a gradient
 - Uses lots of memory and slow to compute
- Instead use batch gradient descent
 - Take a subset of data when computing the gradient

```
while not converged:
```

```
    # Shuffle data
```

```
    data = randomize(data)
```

```
    # Split data into batches and update each batch individual
```

```
    for data_batch in data:
```

```
        weights_grad = backpropagation(loss_fun, data_batch , weights)
```

```
        # perform parameter update
```

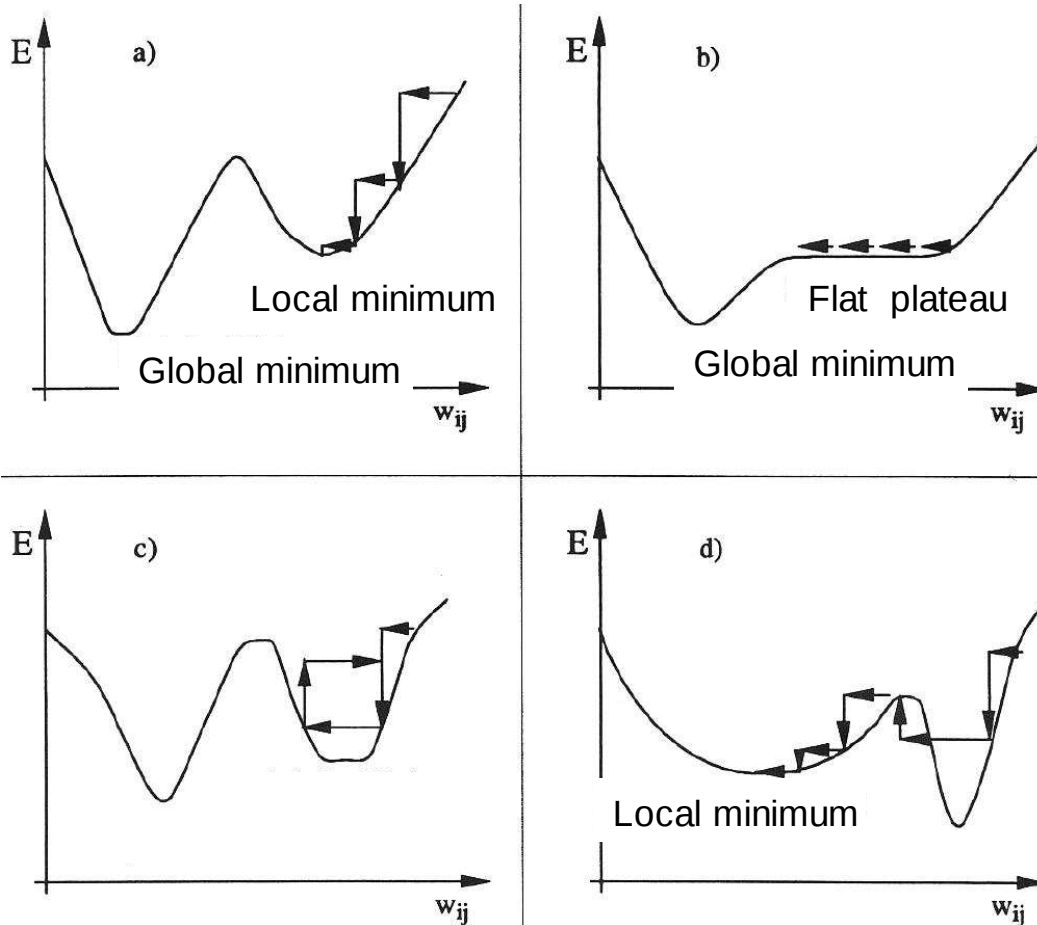
```
        weights += - step_size * weights_grad
```

Iteration

Epoch



Potential issues



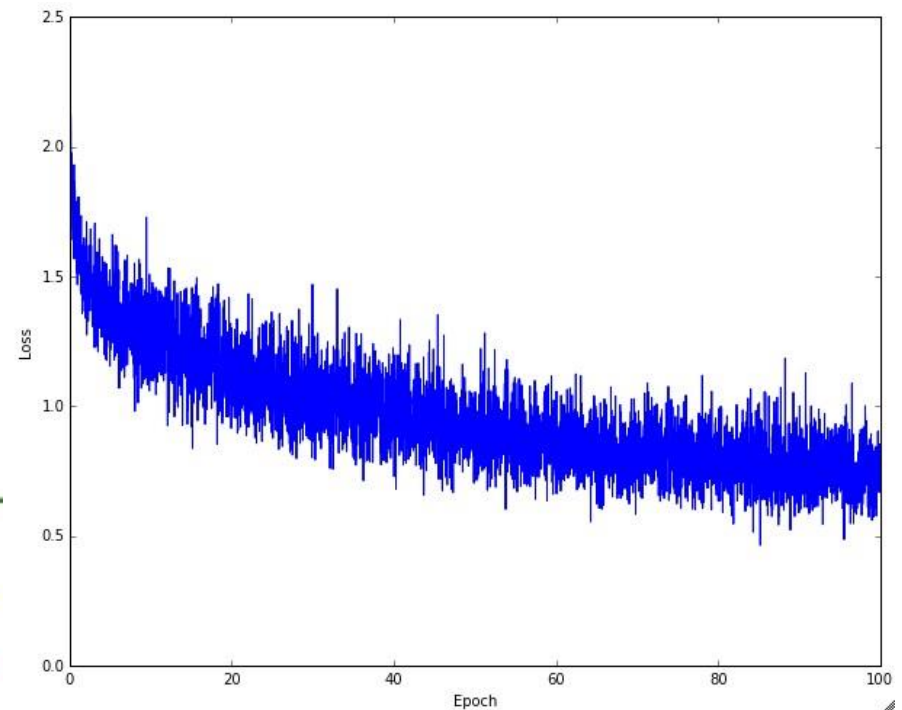
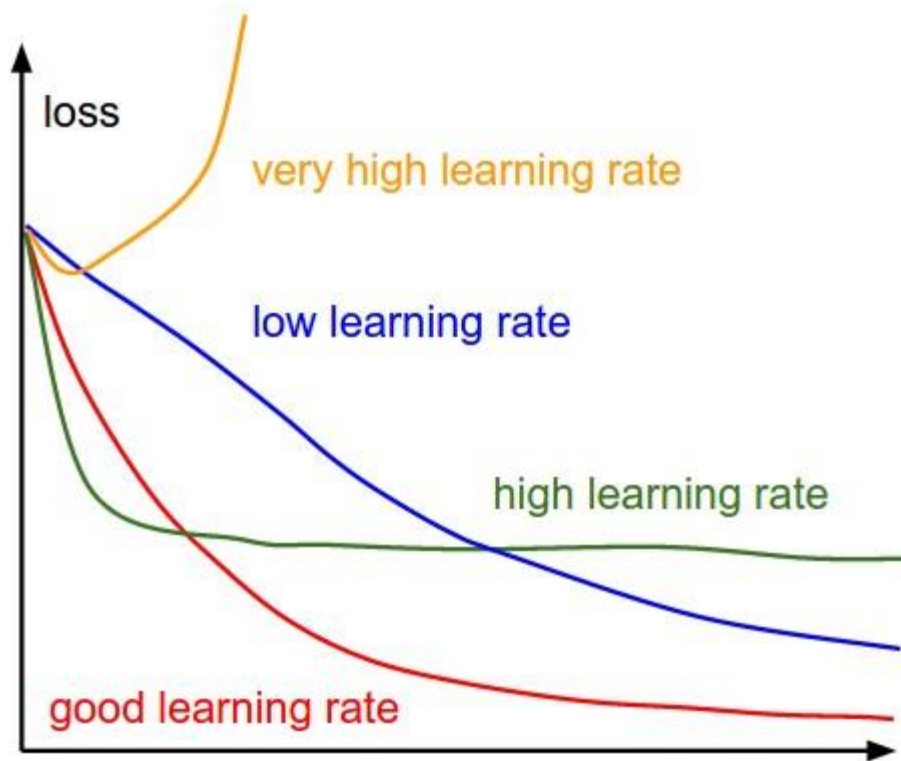
Problems that can occur?

- Getting stuck in local minima (global minimum is never found) (a)
- Getting stuck on flat plateaus of the error-plane (b)
- Oscillations in error rates (c)
- Learning rate is critical (d)

Some observations:

- Small steps are likely to lead to consistent but slow progress.
- Large steps can lead to better progress but are more risky.
- Note that eventually, for a large step size we will overshoot and make the loss worse.

Interpreting learning rates



Convolutional Neural Networks



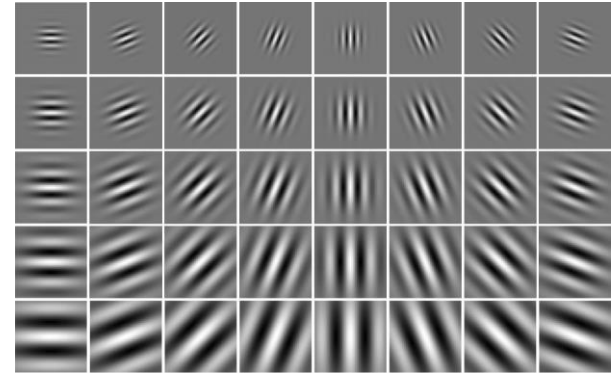
Why not just use an MLP for images (1)?

- MLP connects each pixel in an image to each neuron
- Does not exploit redundancy in image structure
 - Detecting edges, blobs
 - Don't need to treat the top left of image differently from the center
- Too many parameters
 - For a small 200×200 pixel RGB image the first matrix would have $120000 \times n$ parameters for the first layer alone



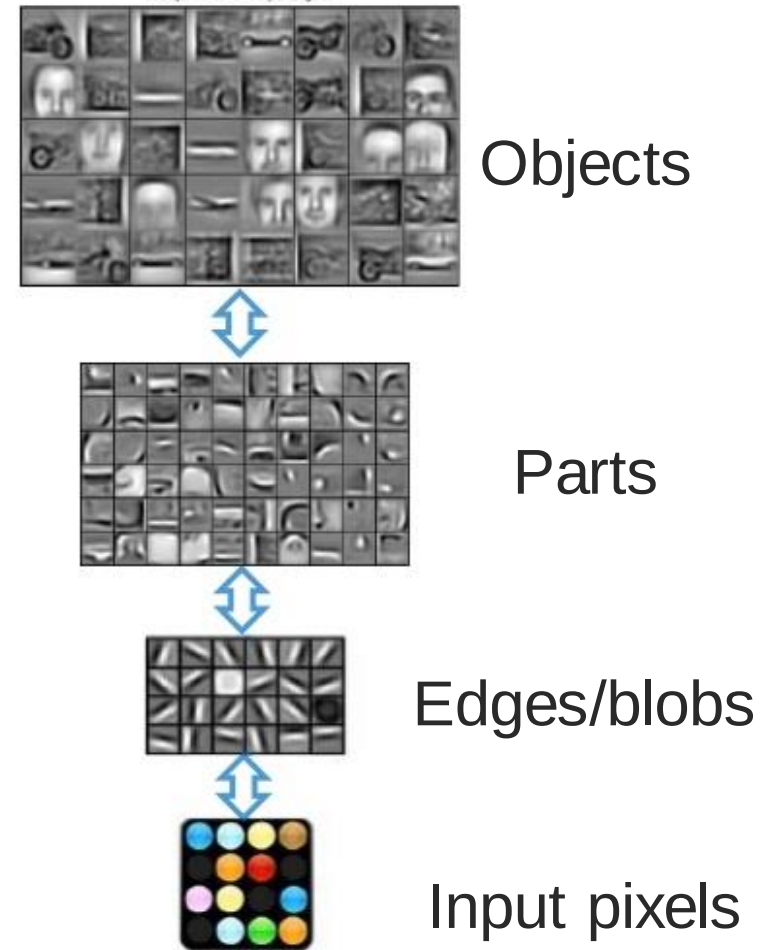
Why not just use an MLP for images (2)?

- Human visual system works in a filter fashion
 - First the eyes detect edges and change in light intensity
 - The visual cortex processing performs Gabor like filtering
- MLP does not exploit translation invariance
- MLP does not necessarily encourage visual abstraction



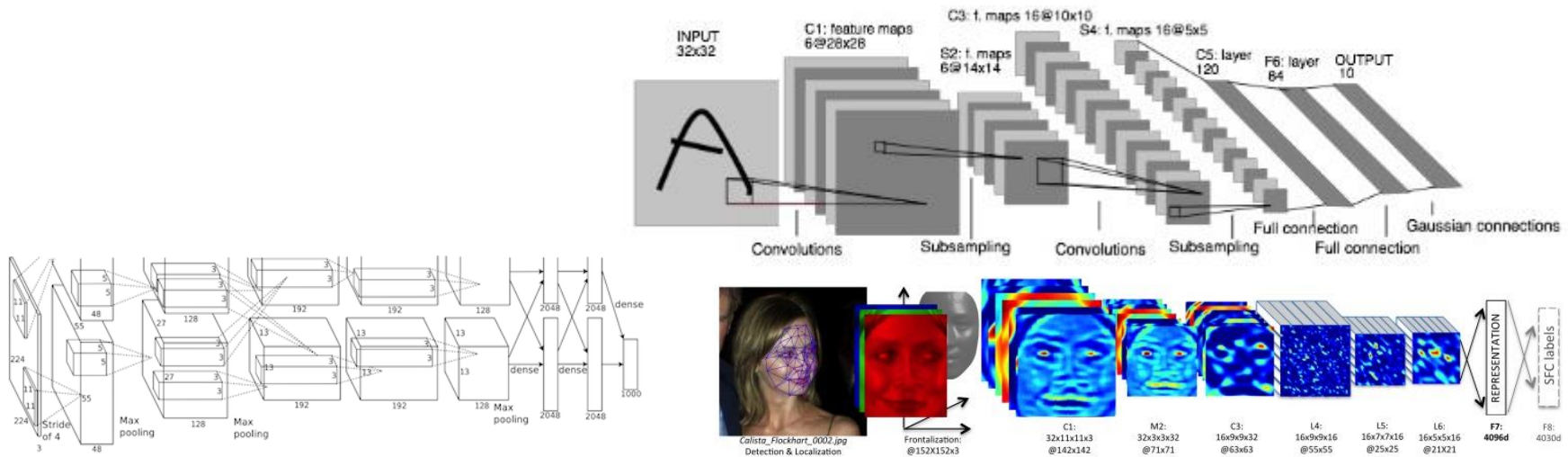
Why use Convolutional Neural Networks

- Using basic Multi Layer Perceptrons does not work well for images
- Intention to build more abstract representation as we go up every layer



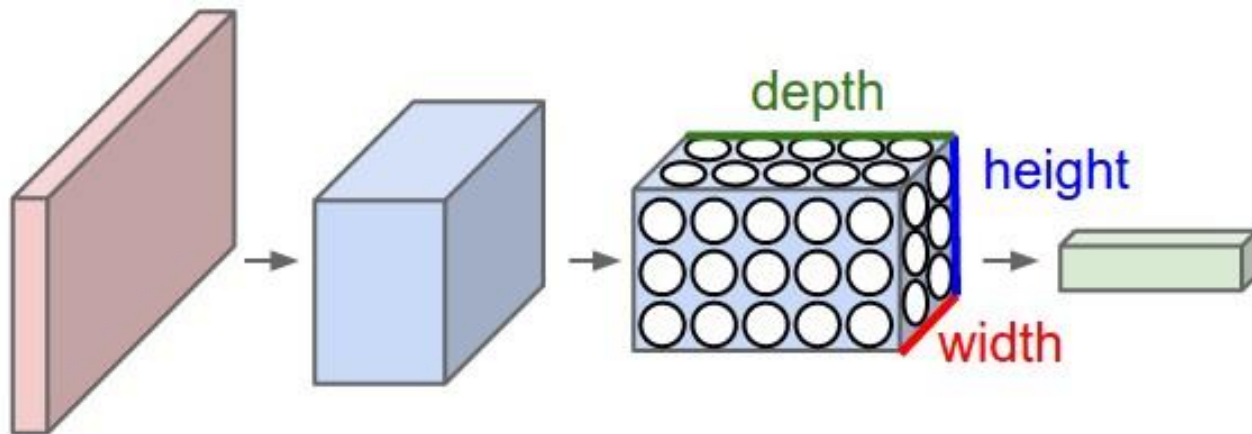
Convolutional Neural Networks

- They are everywhere that uses representation learning with images
- State of the art results – object recognition, face recognition, segmentation, OCR, visual emotion recognition
- Extensively used for multimodal tasks as well



Main differences of CNN from MLP

- Addition of:
 - Convolution layer
 - Pooling layer
- Everything else is the same (loss, score and optimization)
- MLP layer is called Fully Connected layer



Convolution



Convolutional definition

- A basic mathematical operation (that given two functions returns a function)

$$(f * g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]g[n - m]$$

- Have a continuous and discrete versions (we focus on the latter)



Convolution in 1D

- Why do we flip the signal?
 - Mathematical convention
 - Makes certain proofs and properties neater
 - The unflipped version is called correlation

$$(f * g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]g[n + m]$$



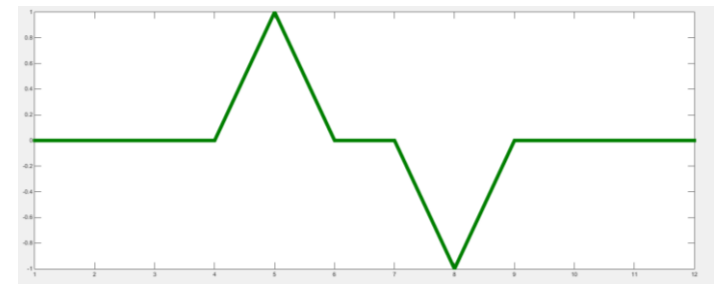
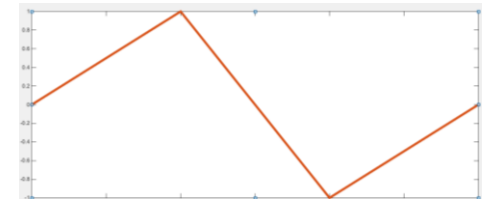
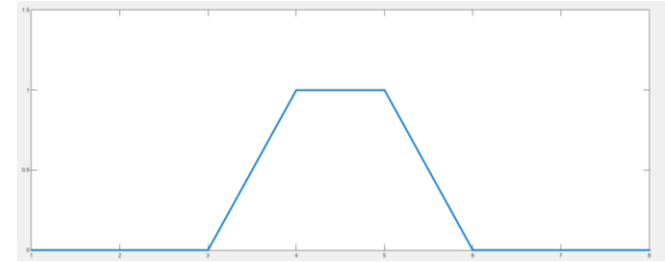
Convolution in 1D

- Example

- $f = [\dots, 0, 1, 1, 1, 0, 0, \dots]$

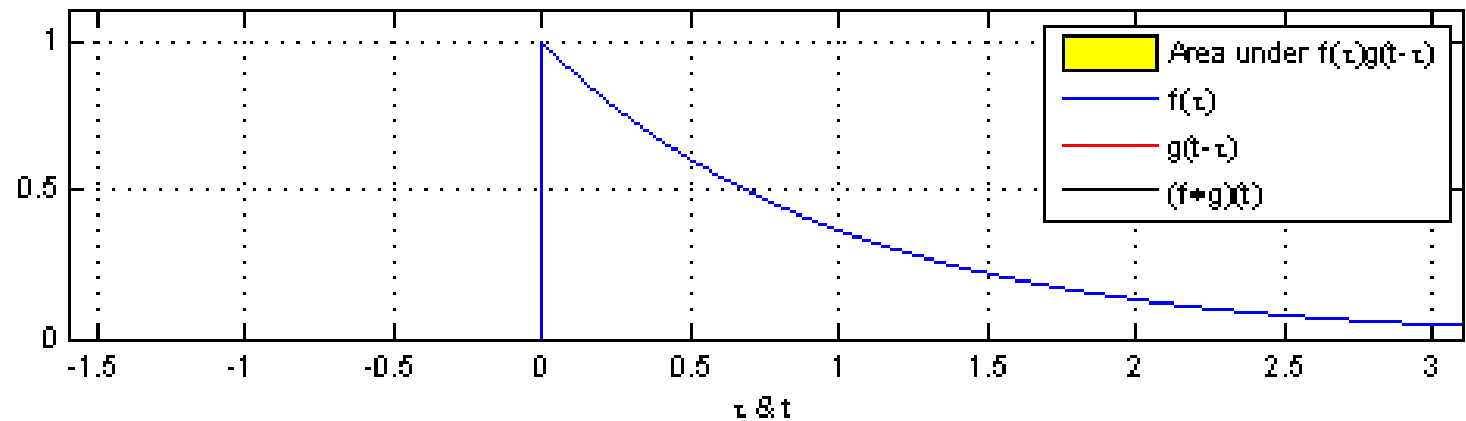
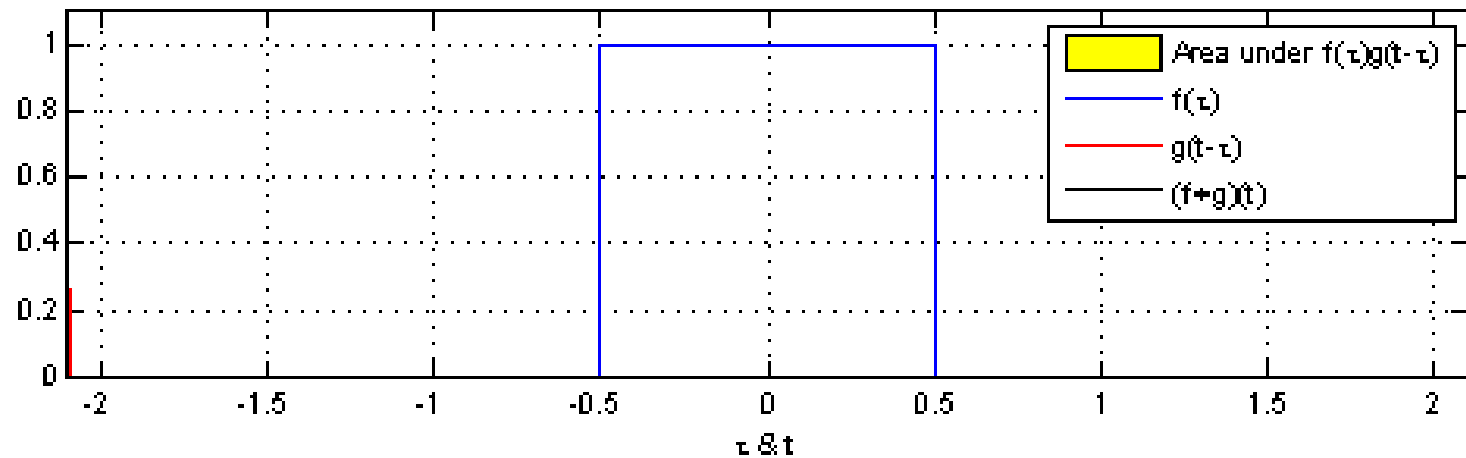
- $g = [\dots, 0, 1, -1, 0, \dots]$

- $f * g = [\dots, 0, 1, 0, 0, -1, 0, 0, \dots]$



$$(f * g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

Convolution in 1D



Convolution in practice

- In CNN we only consider functions with limited domain (not from $-\infty$ to ∞)
- Also only consider fully defined (valid) version
 - We have a signal of length N
 - Kernel of length K
 - Output will be length $N - K + 1$
- $f = [1, 2, 1]$, $g = [1, -1]$, $f * g = [1, -1]$



Convolution in practice

- If we want output to be different size we can add padding to the signal
 - Just add 0s at the beginning and end
- $f = [0,0,1,2,1,0,0]$, $g = [1, -1]$, $f * g = [0,1,1, -1, -1,0]$
- Also have strided convolution (the filter jumps over pixels or signal)
 - With stride 2
 - $f = [0,0,1,2,1,0,0]$, $g = [1, -1]$, $f * g = [0,1,-1,0]$

Convolution in 2D

- Example of image and a kernel



*



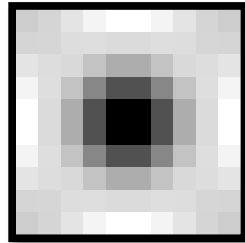
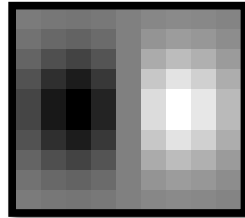
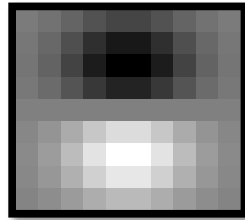
=



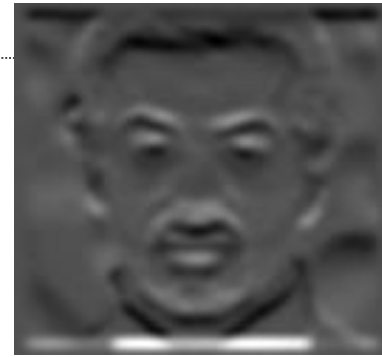
Convolution in 2D



*



=



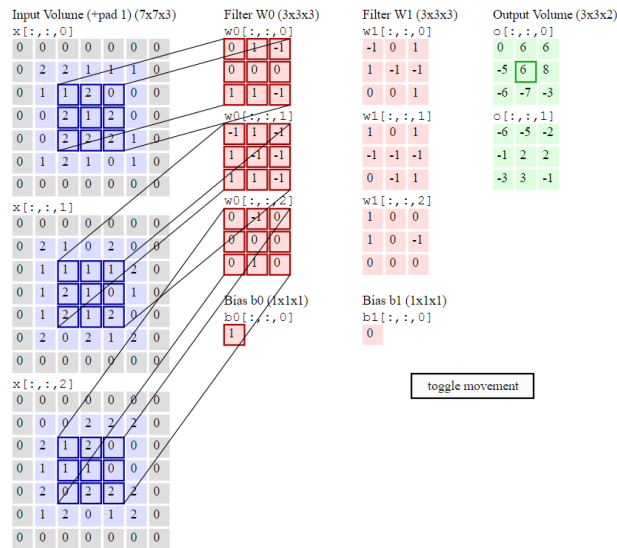
Convolution intuition

- Correlation/correspondence between two signals
 - Template matching
- Why are we interested in convolution
 - Allows to extract structure from signal or image
 - A very efficient operation on signals and images



Sample CNN convolution

- Great animated visualization of 2D convolution
- <http://cs231n.github.io/convolutional-networks/>



Some convolution properties

- Commutative - $f * g = g * f$
- Associative - $f * (g * h) = (f * g) * h$
- Distributive - $f * (g + h) = (f * g) + (f * h)$
- Can use tricks to make it really efficient
 - Fourier transform
 - GPU computation

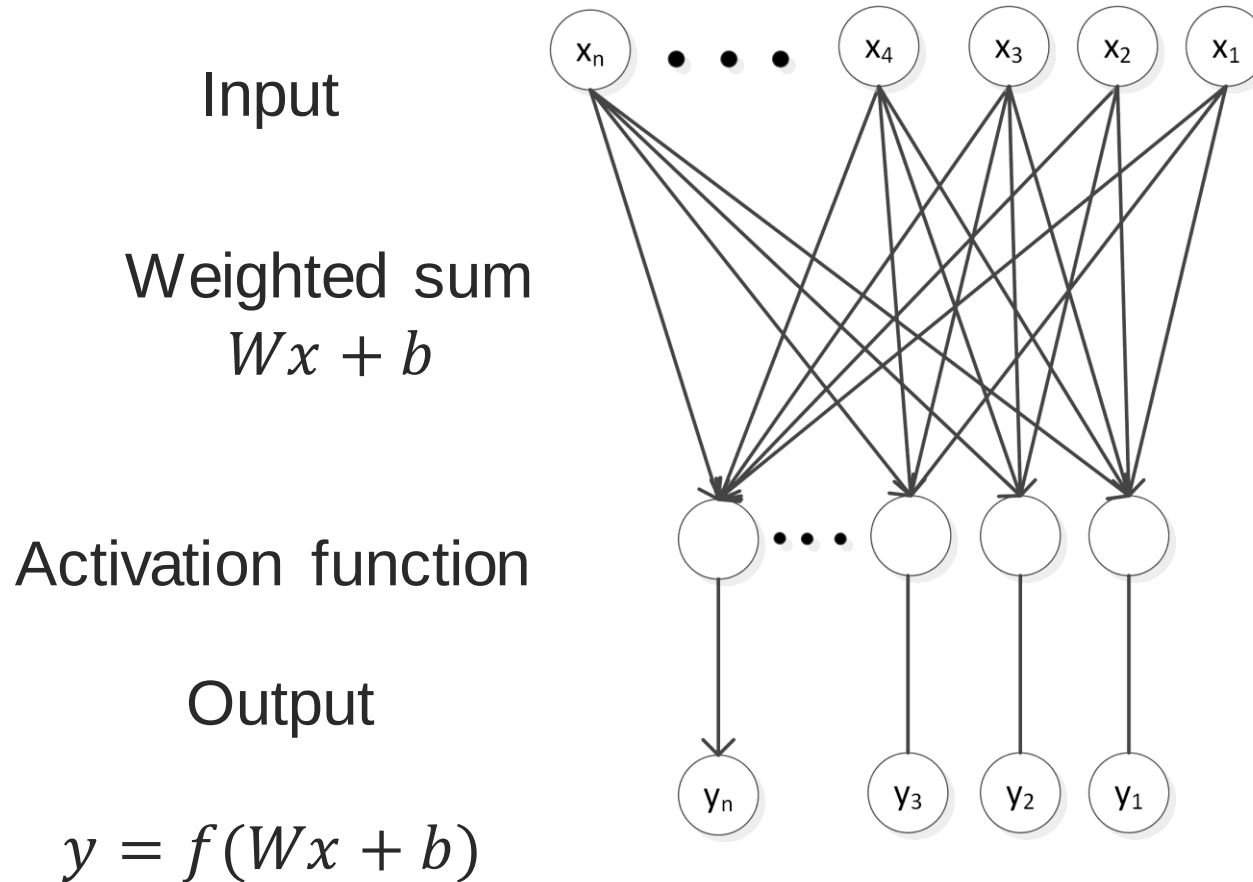


Convolution with MLP



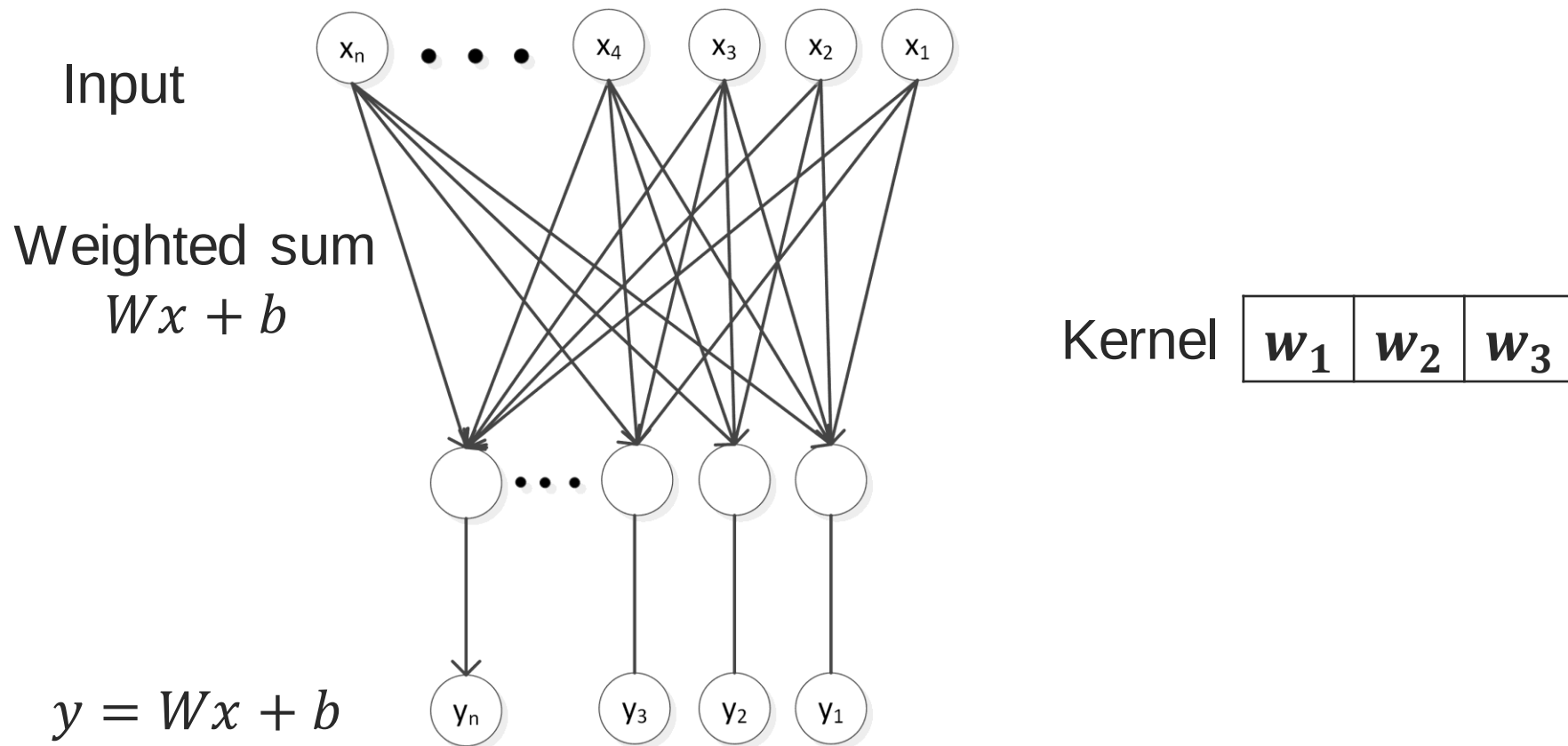
Fully connected layer

- Weighted sum followed by an activation function



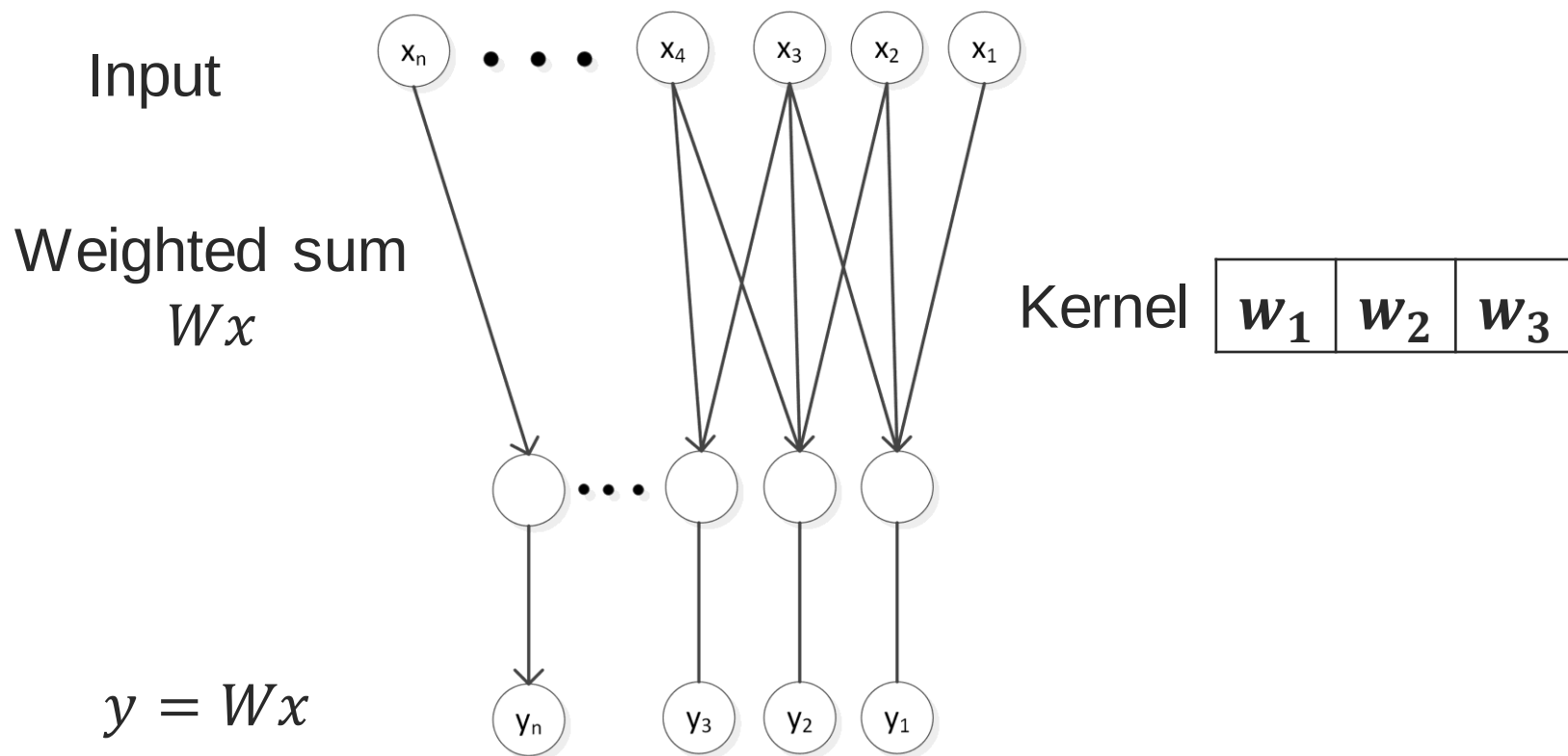
Convolution as MLP (1)

- Remove activation



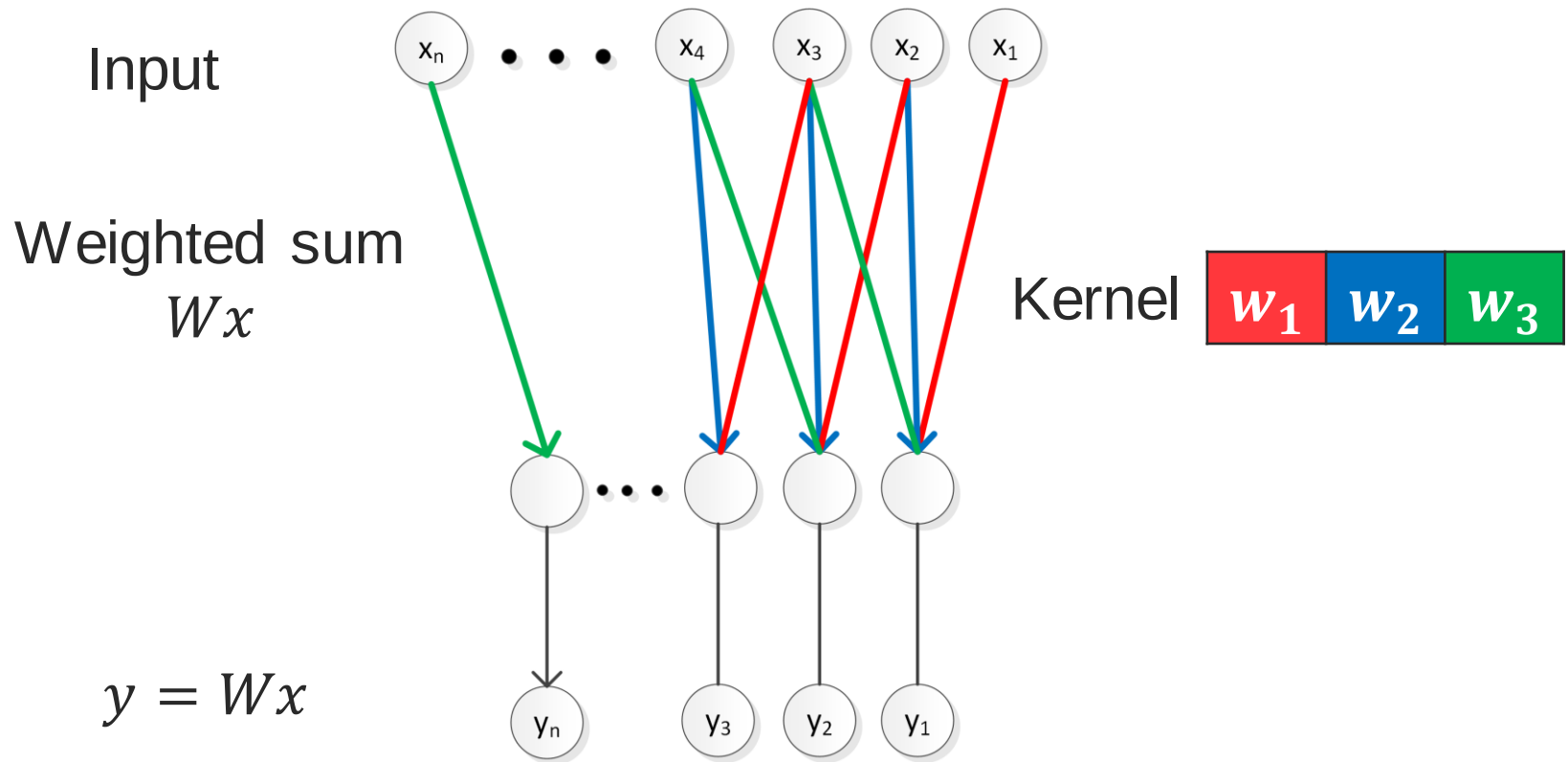
Convolution as MLP (2)

- Remove redundant links making the matrix W sparse (optionally remove the bias term)



Convolution as MLP (3)

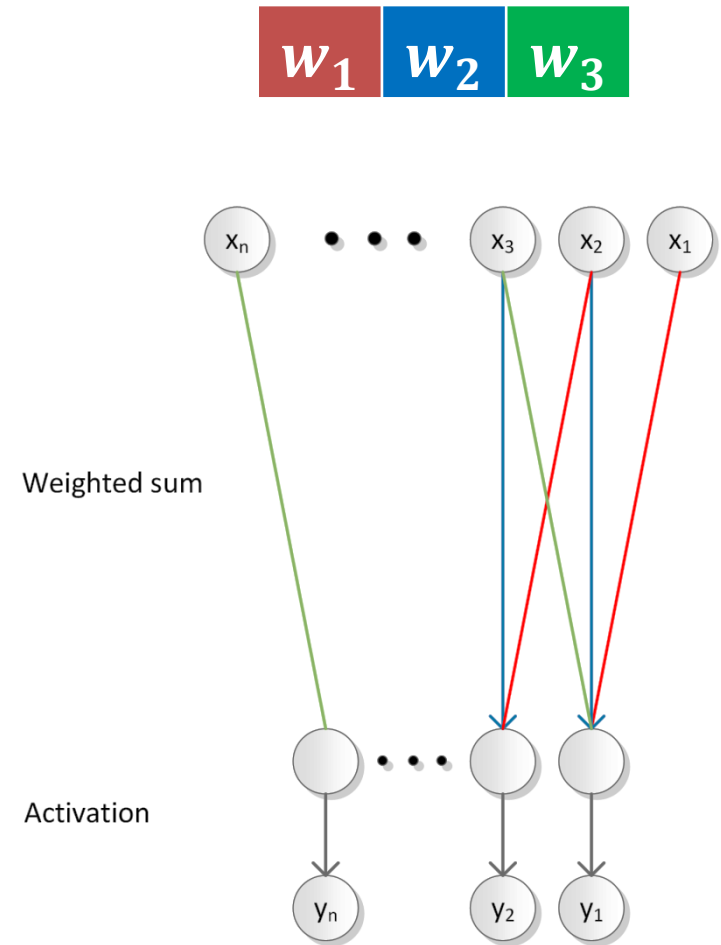
- We can also share the weights in matrix W not to do redundant computation



How do we do convolution in MLP recap

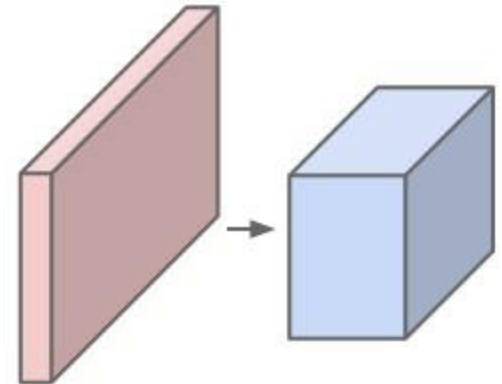
- Not a fully connected layer anymore
- Shared weights
 - Same colour indicates same (shared) weight

$$W = \begin{pmatrix} w_1 & w_2 & w_3 & & 0 & 0 & 0 \\ 0 & w_1 & w_2 & \dots & 0 & 0 & 0 \\ 0 & 0 & w_1 & & 0 & 0 & 0 \\ & \vdots & & \ddots & \vdots & & \\ 0 & 0 & 0 & & w_3 & 0 & 0 \\ 0 & 0 & 0 & \dots & w_2 & w_3 & 0 \\ 0 & 0 & 0 & & w_1 & w_2 & w_3 \end{pmatrix}$$



More on convolution

- Can expand this to 2D
 - Just need to make sure to link the right pixel with the right weight
- Can expand to multi-channel 2D
 - For RGB images
- Can expand to multiple kernels/filters
 - Output is not a single image anymore, but a **volume** (sometimes called a feature map)
 - Can be represented as a tensor (a 3D matrix)
- Usually also include a bias term and an activation

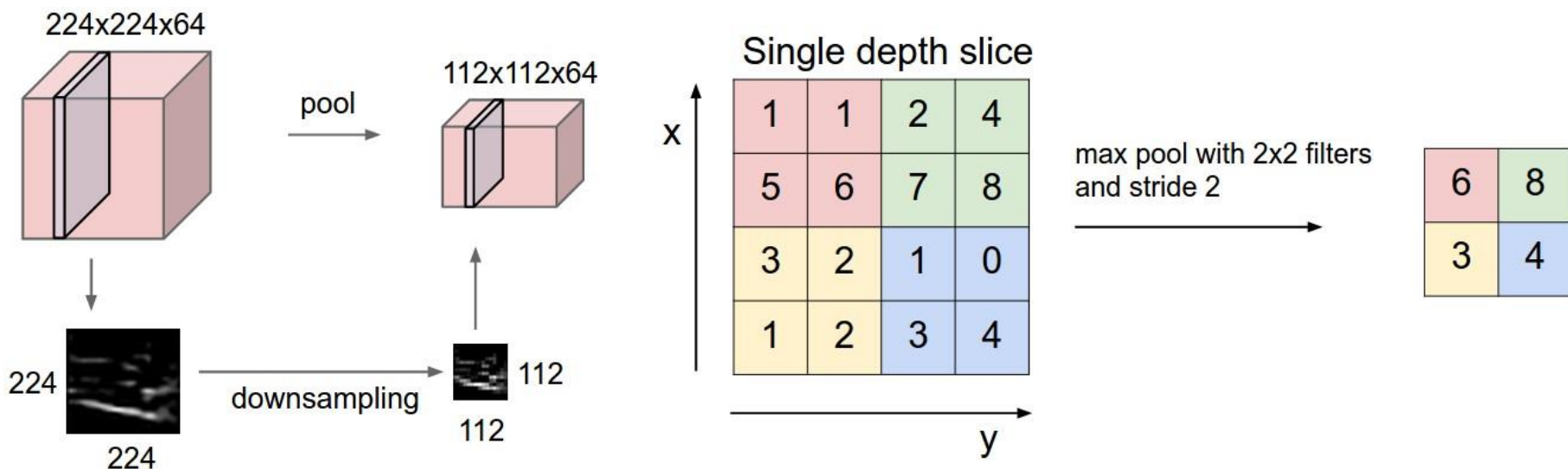


Pooling layer



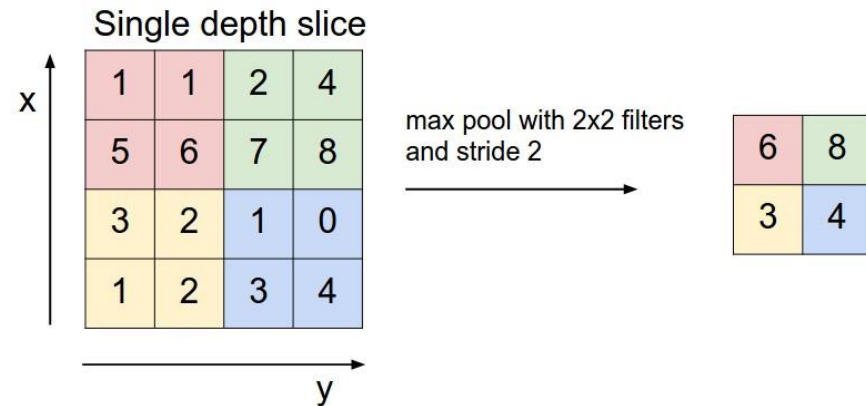
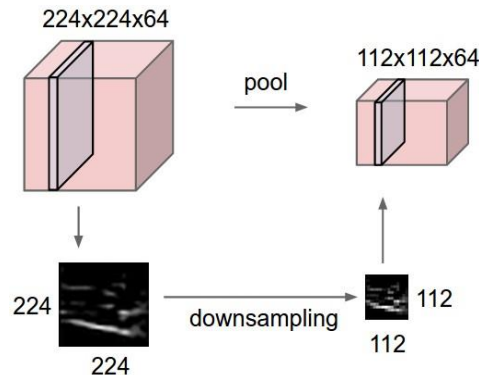
Pooling layer

- Image subsampling



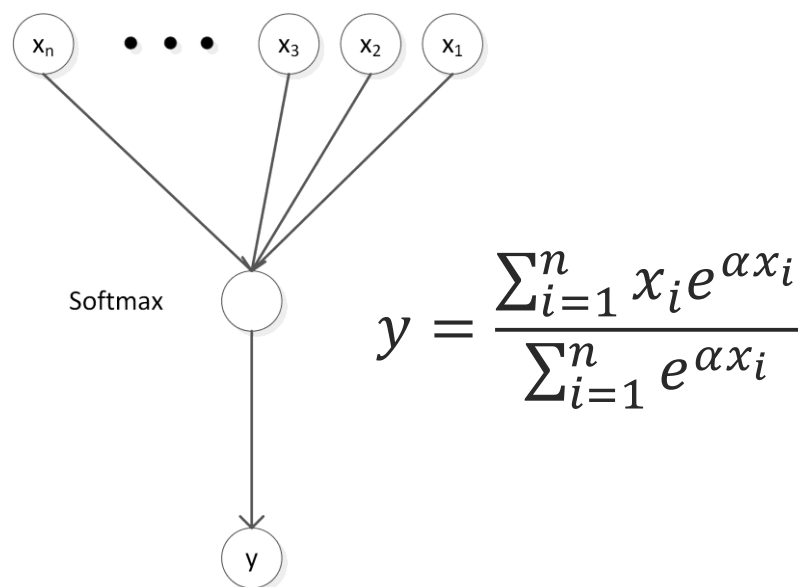
Pooling layer motivation

- Used for sub-sampling
 - Allows summarization of response
- Helps with translational invariance
- Have filter size and stride (hyperparameters)



Pooling layer gradient

1. Record during forward pass which pixel was picked and use the same in backward pass
2. Pick the maximum value from input using a smooth and differentiable approximation

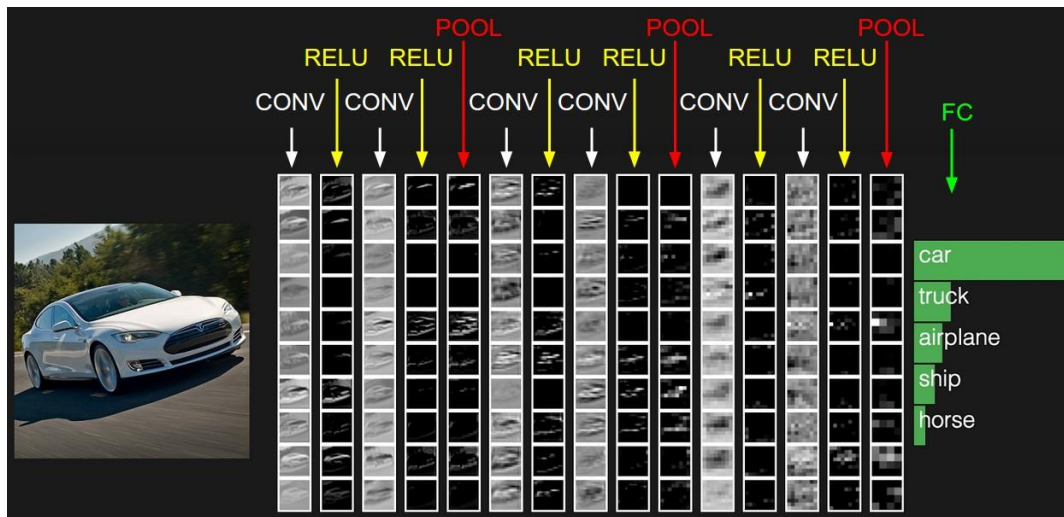


Putting it all together



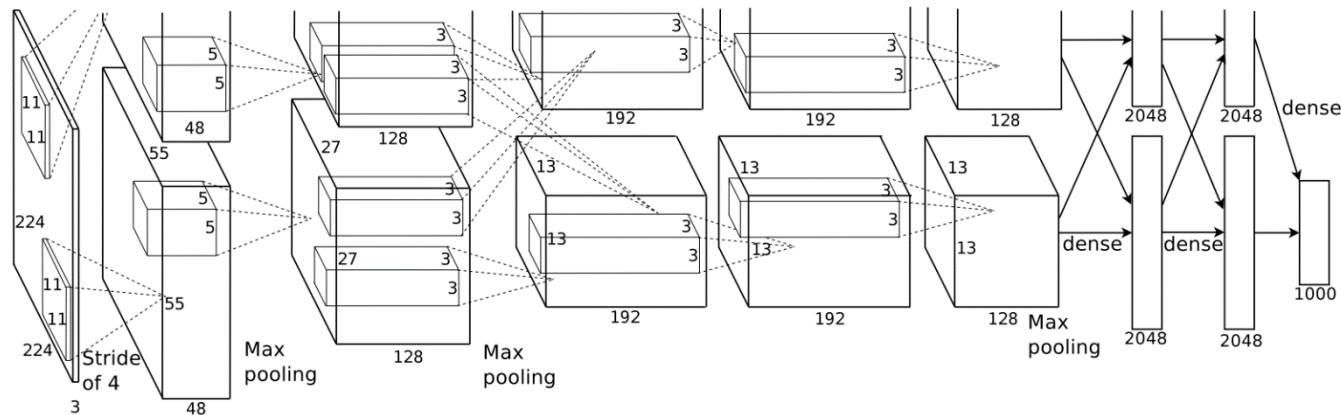
Common architectures

- Start with a convolutional layer follow by non-linear activation and pooling
- Repeat this several times
- Follow with a fully connected (MLP) layer



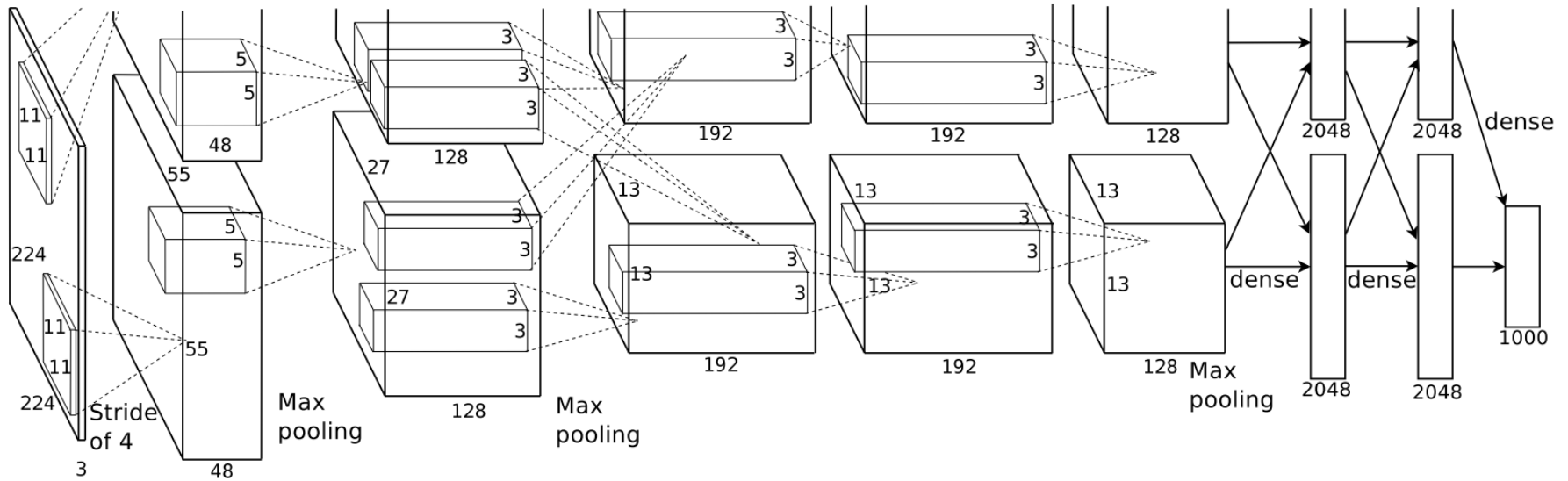
AlexNet model

- Used for object classification task
 - 1000 way classification task – pick one
- Architecture inspired partially by how much parameters can fit on a single GPU
- In total over 61M parameters



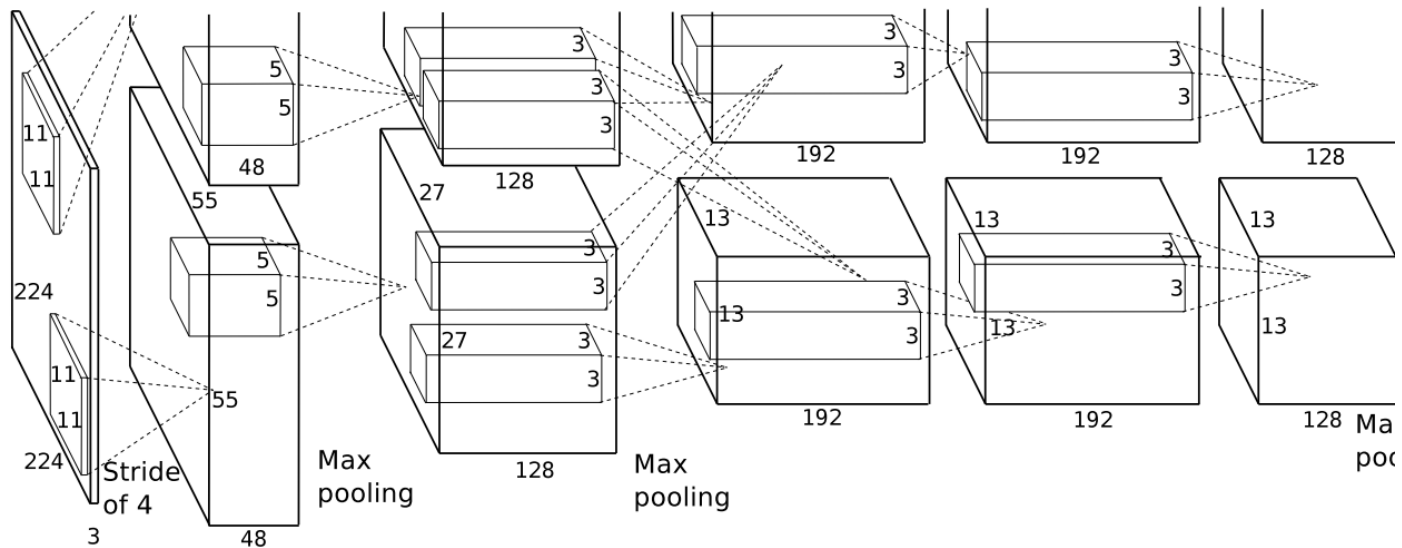
AlexNet architecture

- 7 Layer architecture



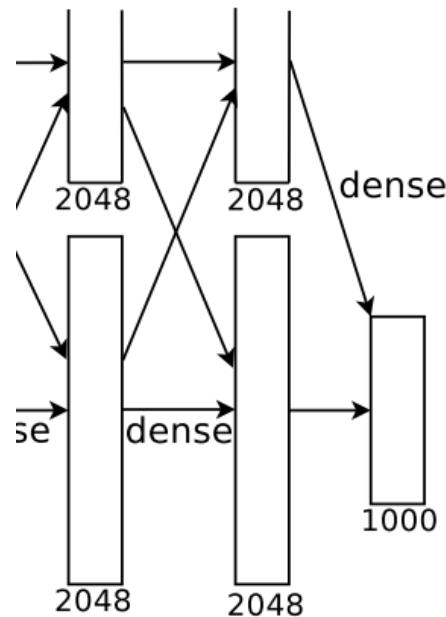
AlexNet architecture

- First five layers
 - Convolution with ReLU followed by max-pooling



AlexNet architecture

- Final layers fully connected (like MLP)



- In papers you will see people referring to FC7

What are the models learning

- Will discuss in the reading group on Thursday



Other popular architectures

- LeNet – an early 5 layer architecture for handwritten digit recognition
- VGGNet – a deep (19 layer) network for object recognition
- ResNet – an even deeper model from Microsoft
- DeepFace – Facebook's face recognition CNN
- **Already trained AlexNet and VGGNet models for object recognition can be found online**



Training tricks

- Data augmentation (Create more data)
 - Image scaling
 - Shifting
 - Rotation
 - Mirroring
- Optimization
 - Dropout
 - Regularization
 - Many more tricks/tips that we will discuss in Week 8



Fine tuning for specific tasks

- Often start with an existing architecture and an already trained network (for example AlexNet or VGGNet for object recognition)
- Discard the final layer score function and replace with your own (FC7)
- Perform gradient decent on it
 - Nice thing about neural networks is that we can continue training them with new data

