



Language
Technologies
Institute

Carnegie
Mellon
University

Multimodal Machine Learning

Lecture 2.2: Basic Concepts - Network Optimization

Louis-Philippe Morency

* Original version co-developed with Tadas Baltrusaitis

Lecture Objectives

- Learning neural networks
 - Optimization
 - Gradient computation
- Practical Deep Model Optimization
 - Adaptive Optimization Methods
 - Regularization
 - Co-adaptation
 - Multimodal Optimization



Learning model parameters

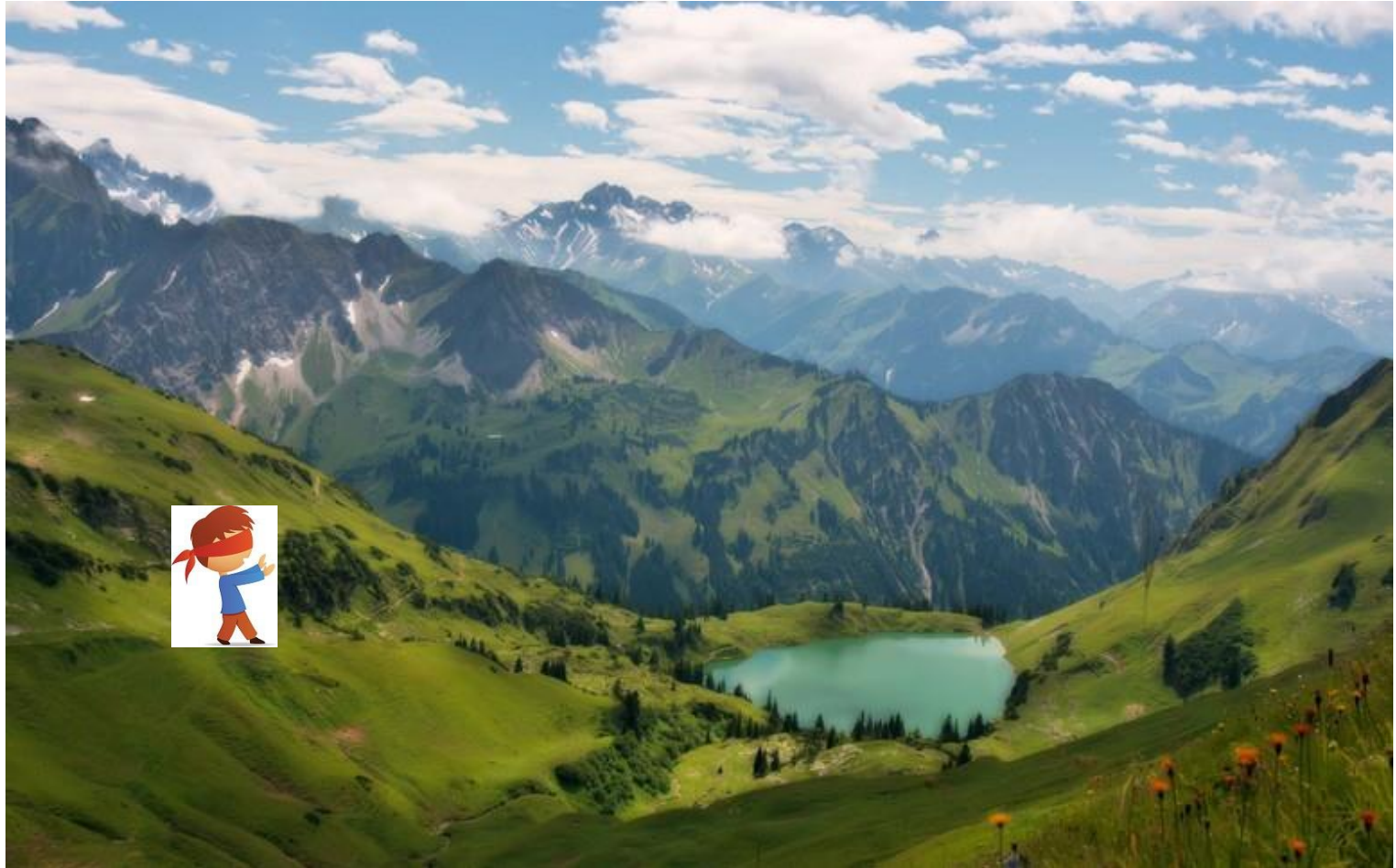


Learning model parameters

- 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



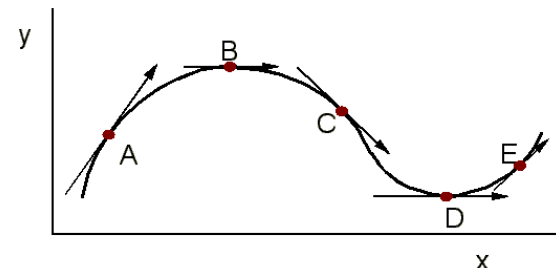
What is a gradient?

- 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

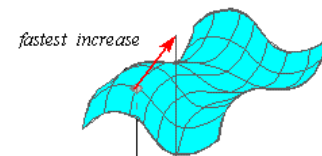
$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$



- 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:

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

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

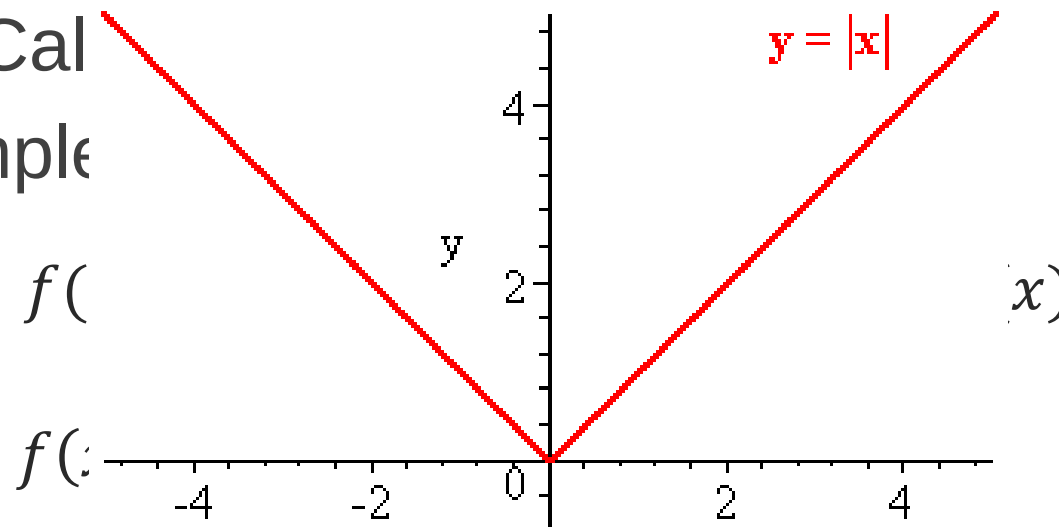


Analytical gradient

- 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](#)



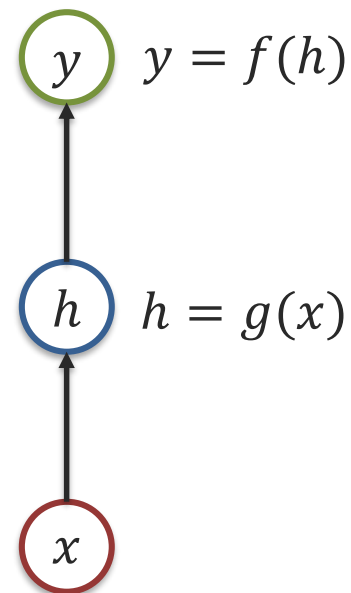
Neural Networks gradient



Gradient Computation

Chain rule:

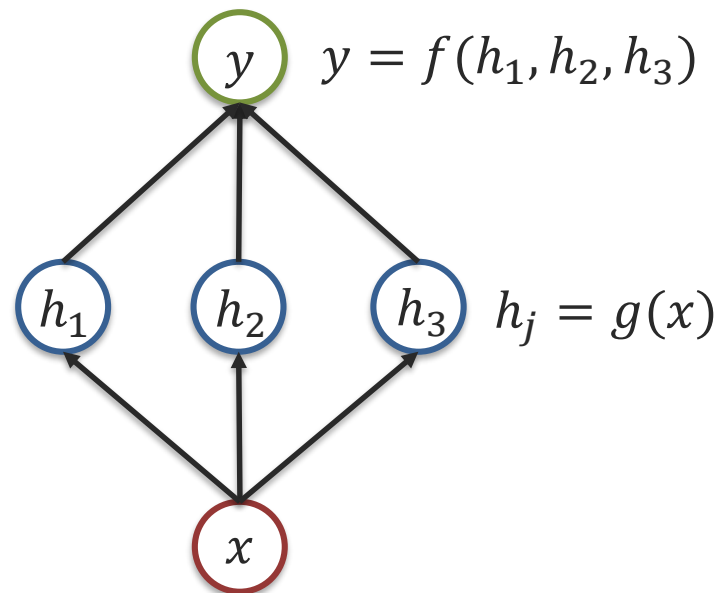
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial h} \frac{\partial h}{\partial x}$$



Optimization: Gradient Computation

Multiple-path chain rule:

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



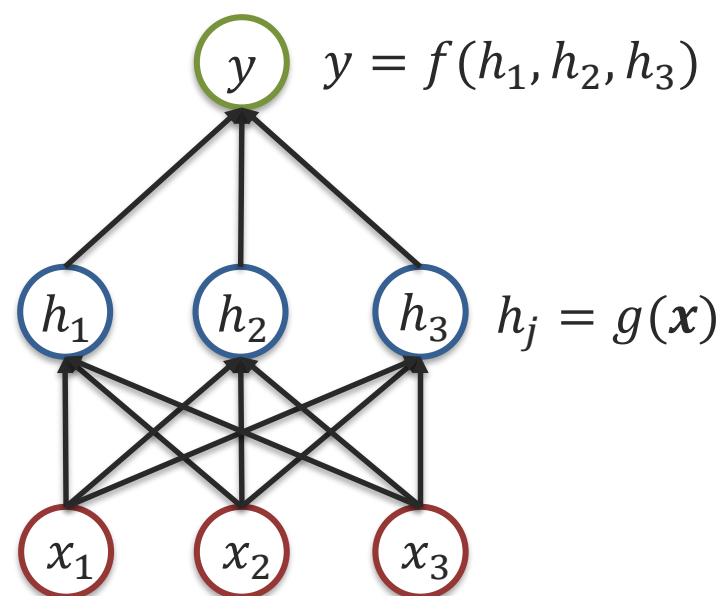
Optimization: Gradient Computation

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

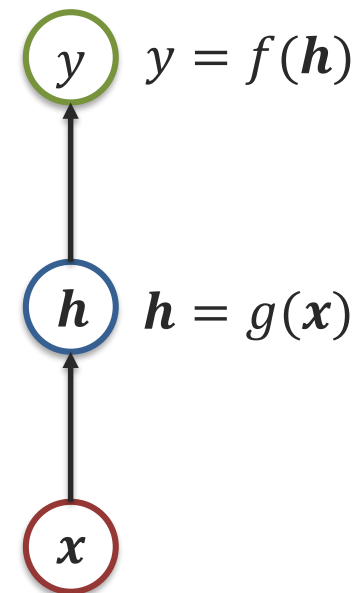
Vector representation:

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

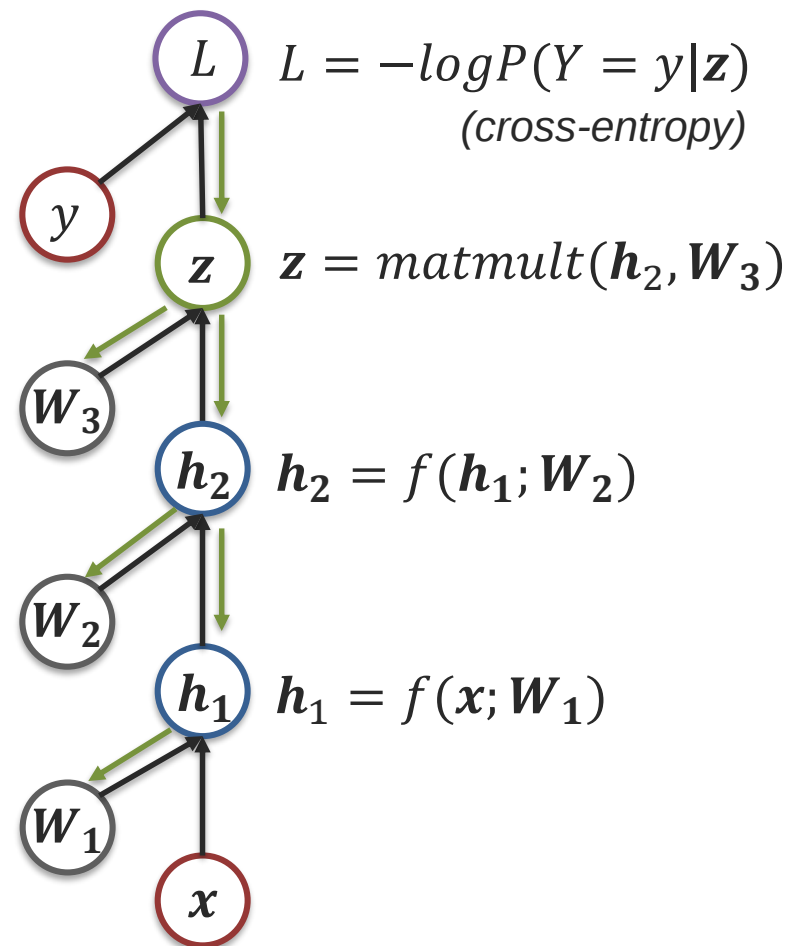
- Following the graph topology, compute value of each unit

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

- Why is this rule important?



Computational Graph: Multi-layer Feedforward Network

Computational unit:



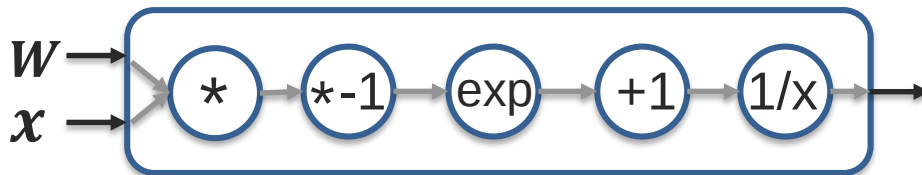
$$h = f(x; W)$$

- Multiple input
- One output
- Vector/tensor

▪ Sigmoid unit:

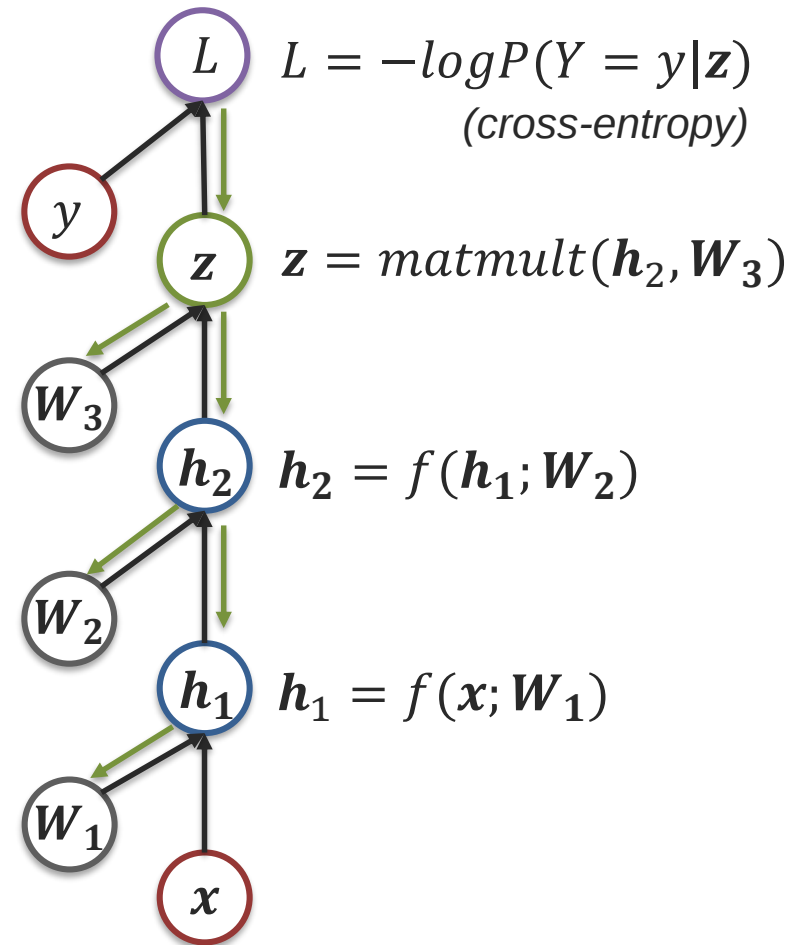


$$h_j = (1 + e^{-W_j x})^{-1}$$



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)
```



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
```

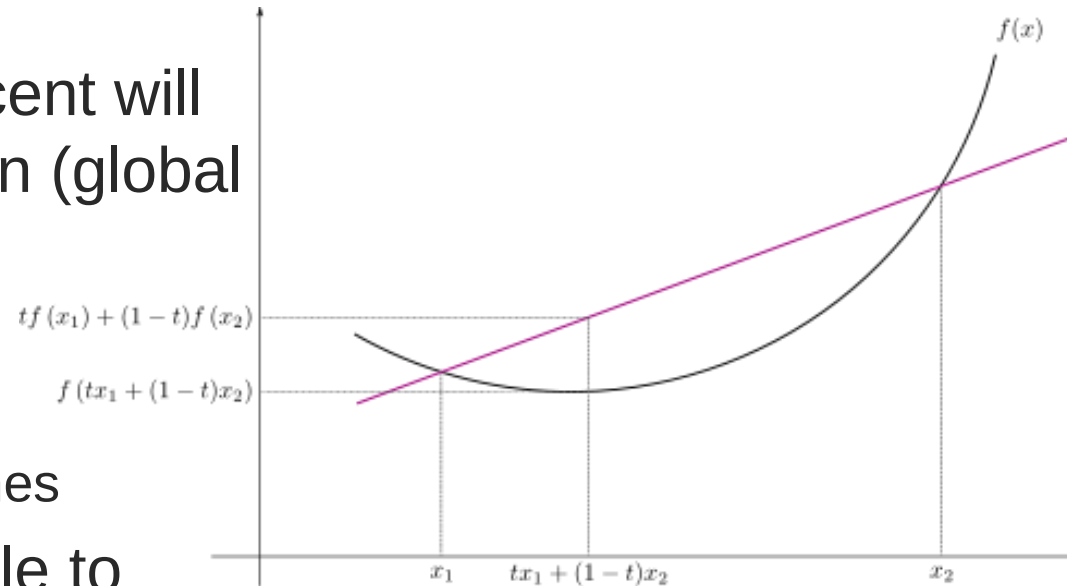
Iteration

Epoch

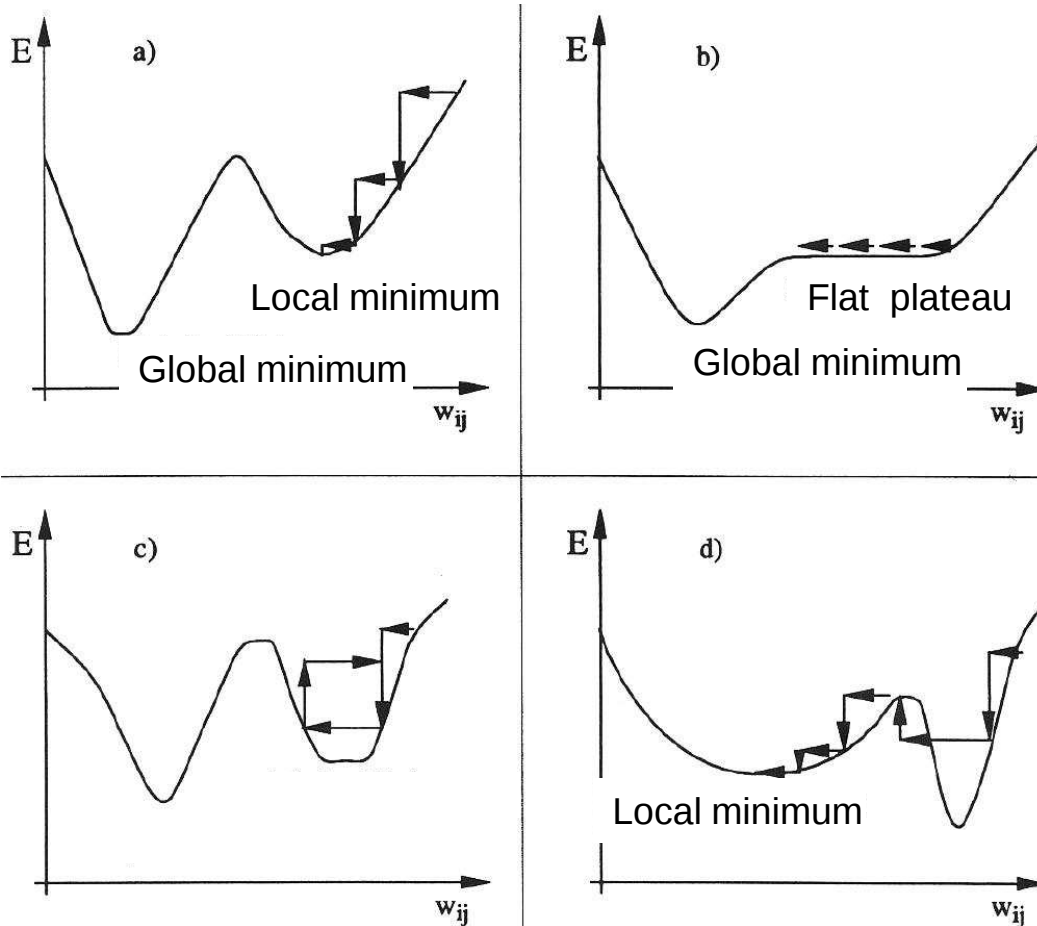


Convex vs. non-convex functions and local minima

- 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



Potential issues



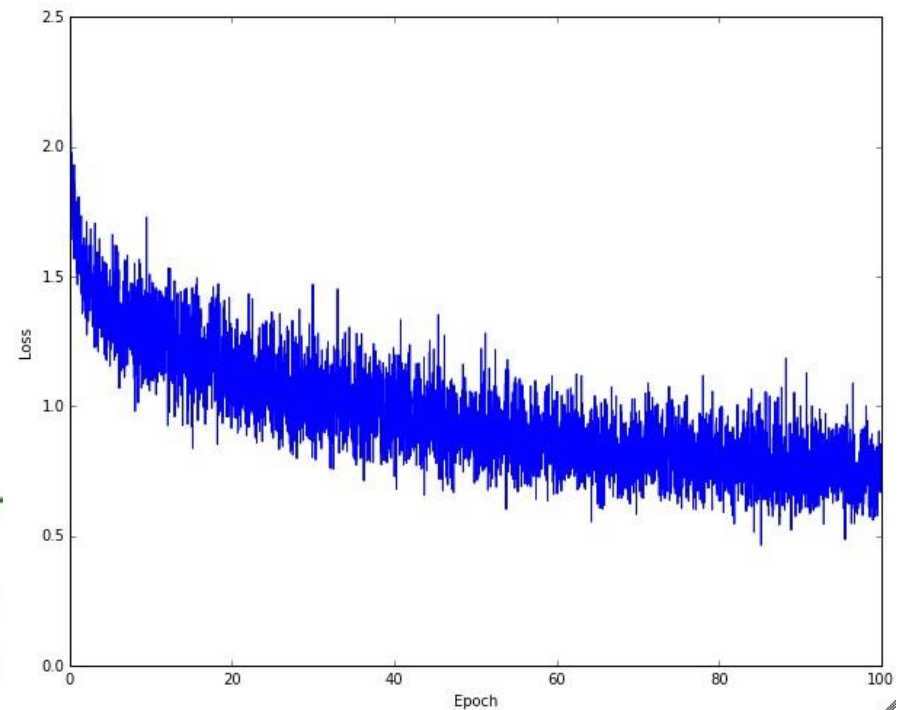
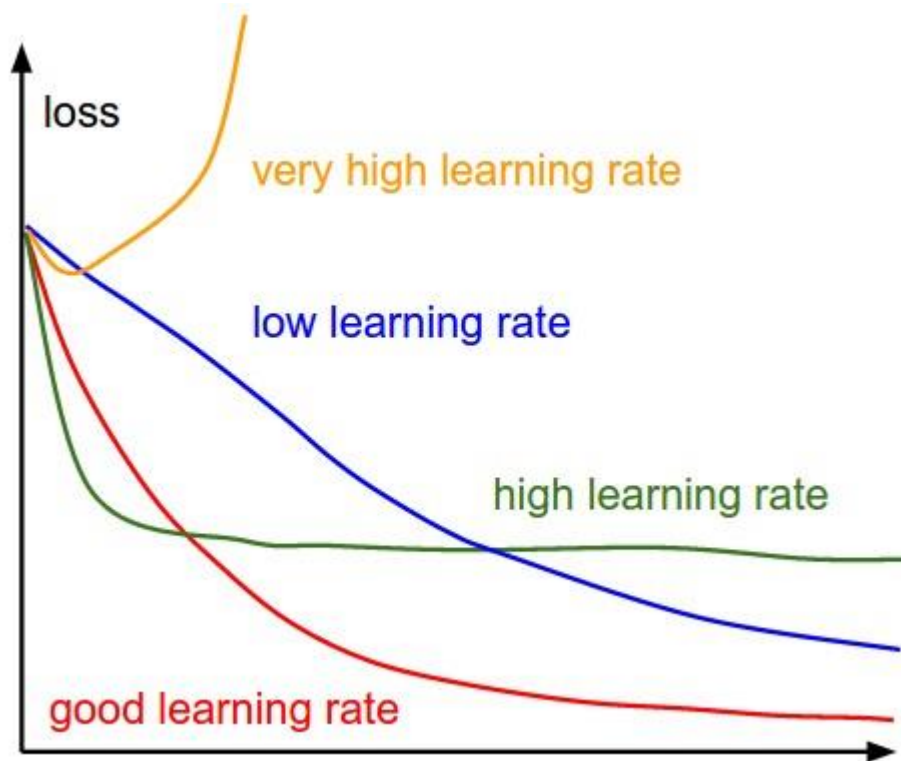
Problems that can occur?

- 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



Optimization – Practical Guidelines



Optimization – Practical Guidelines

- Adaptive Optimization Methods
- Regularization
- Co-adaptation
- Multimodal Optimization



Adaptive Learning Rate

General Idea: Let neurons who just started learning have huge learning rate.

Adaptive Learning Rate is an active area of research:

- Adadelta
- RMSProp

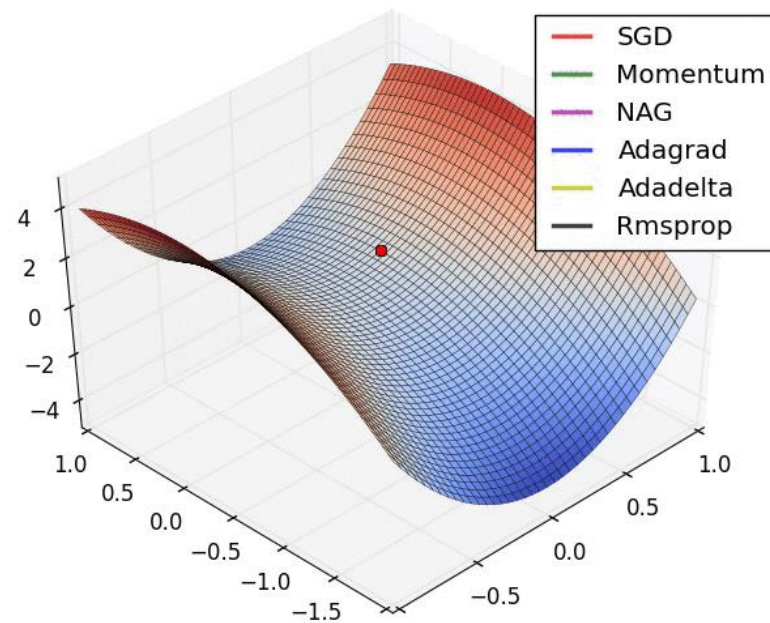
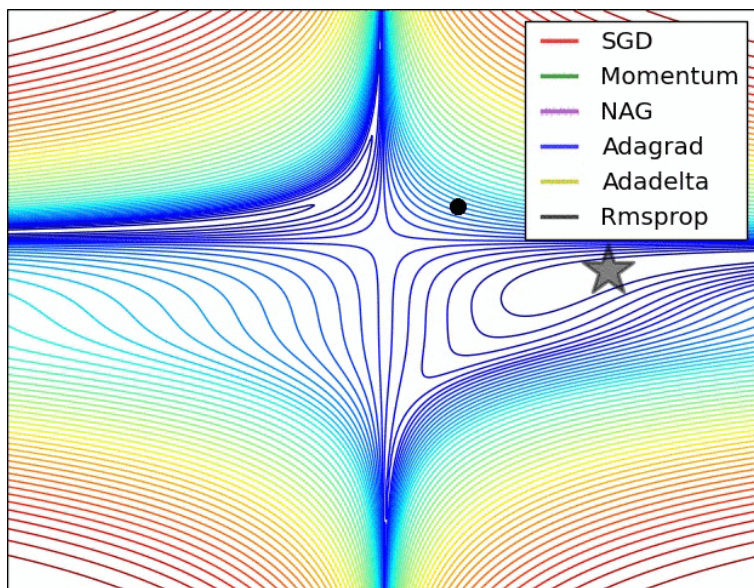
```
cache = decay_rate * cache + (1 - decay_rate) * dx**2  
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Adam

```
m = beta1*m + (1-beta1)*dx  
v = beta2*v + (1-beta2)*(dx**2)  
x += - learning_rate * m / (np.sqrt(v) + eps)
```

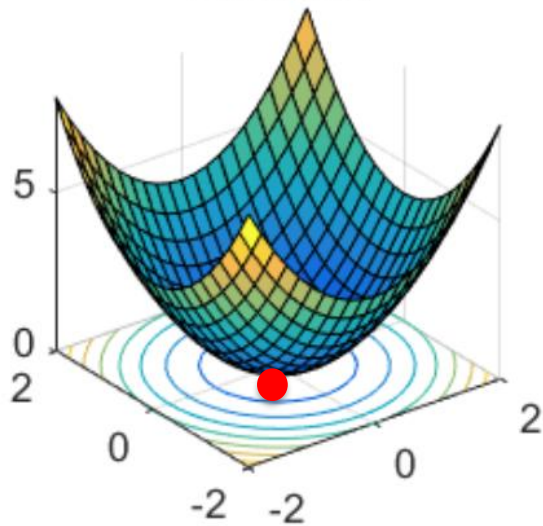


Comparison

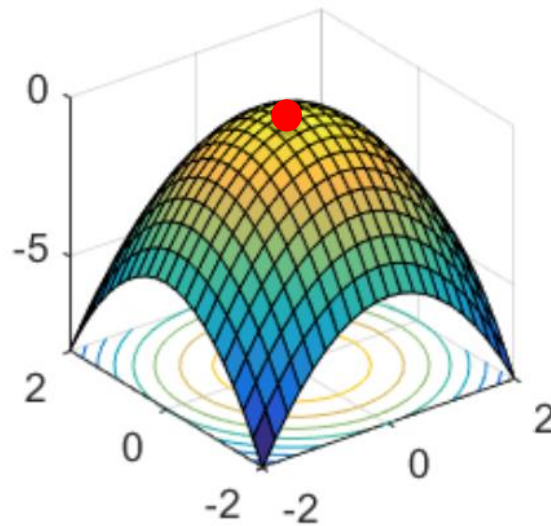


Critical Points

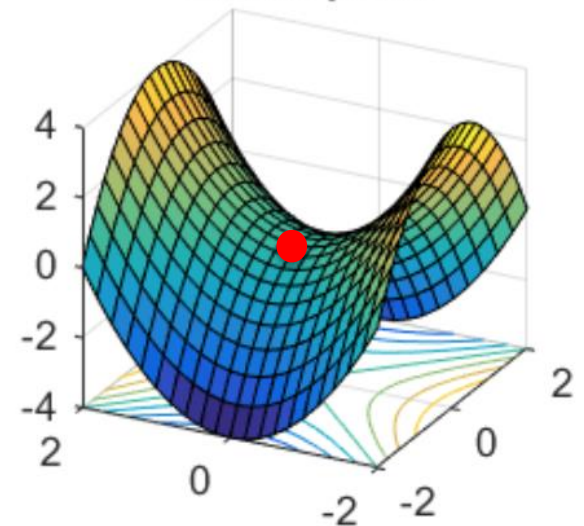
local min



local max



saddle point

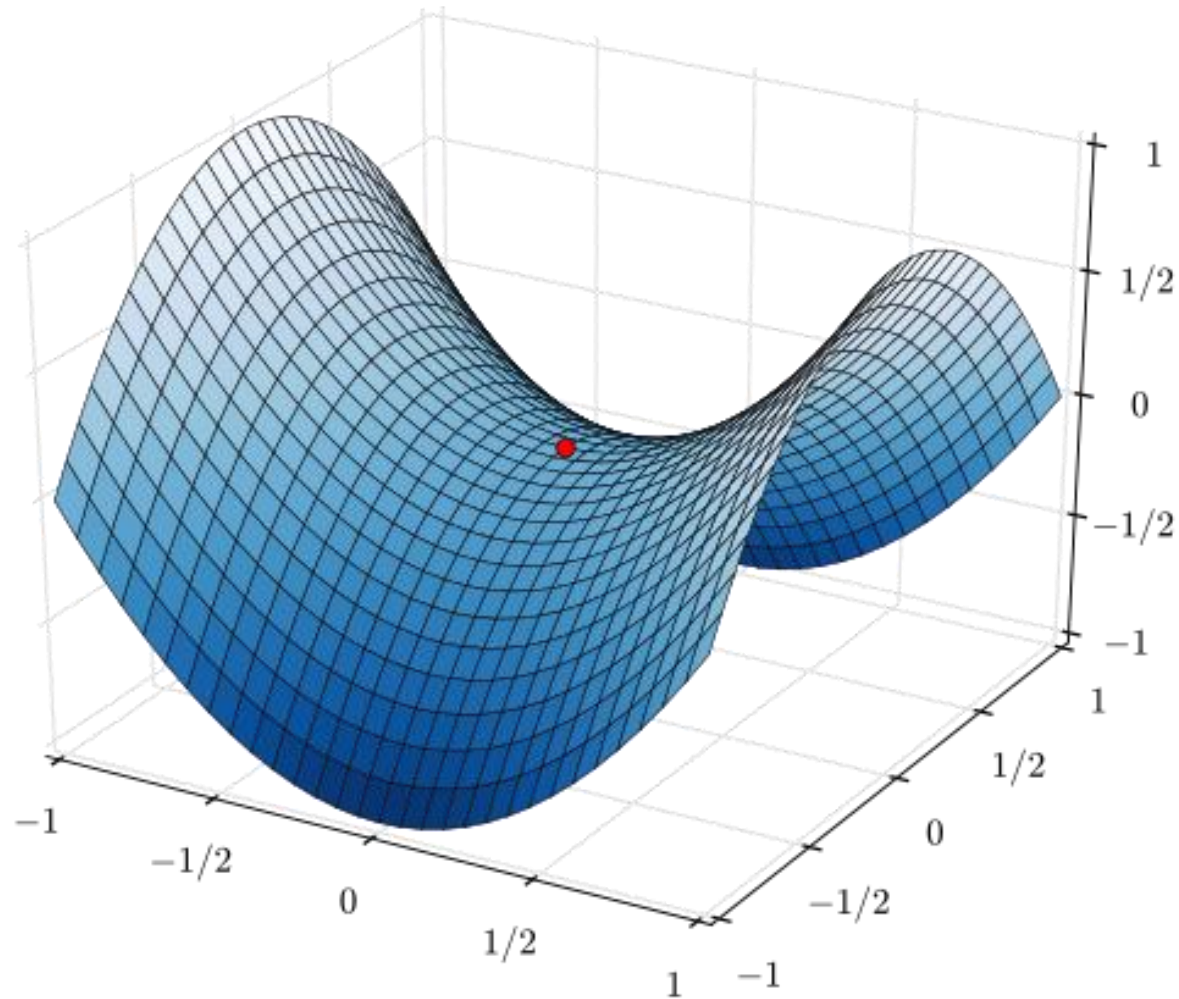


Saddle Points

- Deep Learning Optimization:
 - Deep Learning problems in general have many local minimas 
 - Many (not all) of them are actually almost as good as global minima due to parameter permutation 
 - However it is NP-hard to even find a local minima 
- Lots and lots of saddles in many deep learning problems.



Why Saddles are Bad

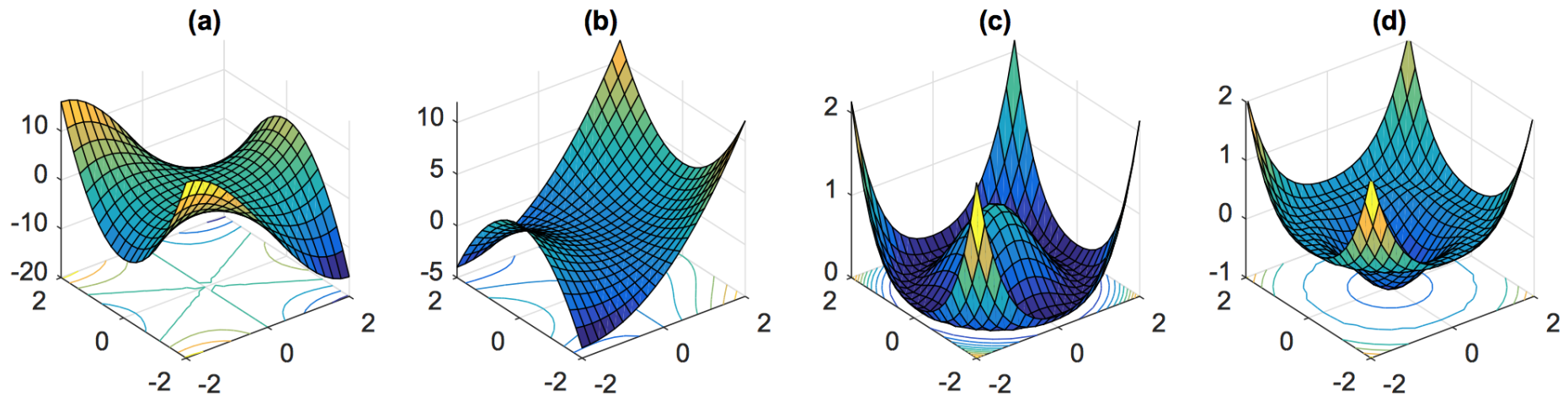


Detecting Saddles

- One way to detect saddles:
 - Calculate Hessian at point x
 - If Hessian is indefinite you have a saddle for sure.
 - If Hessian is not indefinite you really can't tell.
- My loss isn't changing:
 - You are definitely close to a critical point
 - You may be in a saddle point
 - You may be in the local minima/maxima
 - One trick: quickly check the surrounding
 - Best practical trick if Hessian is not indefinite.



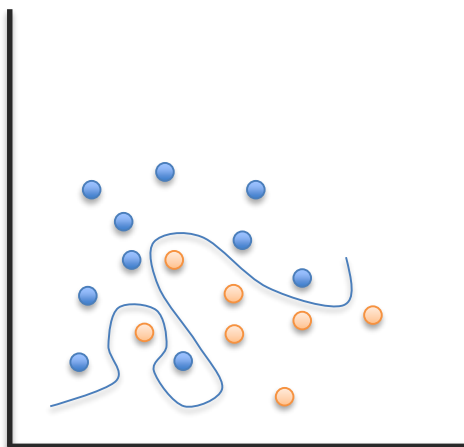
Bad Saddle Points



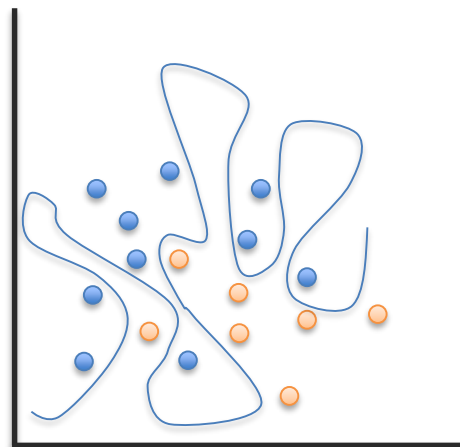
<https://arxiv.org/pdf/1602.05908.pdf>

Example

Real

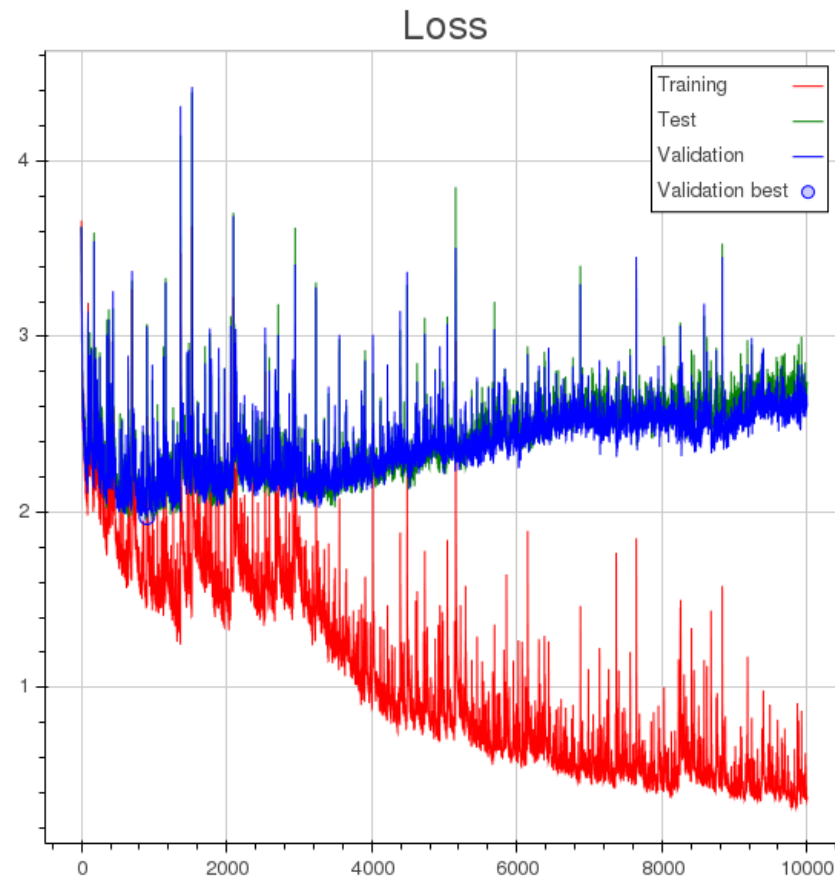


Our
Model



Not the fault of learning rate or momentum

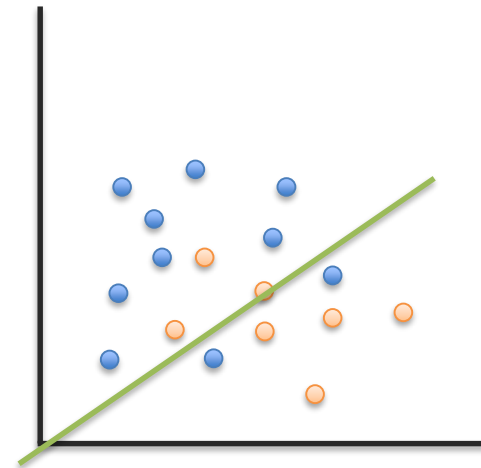
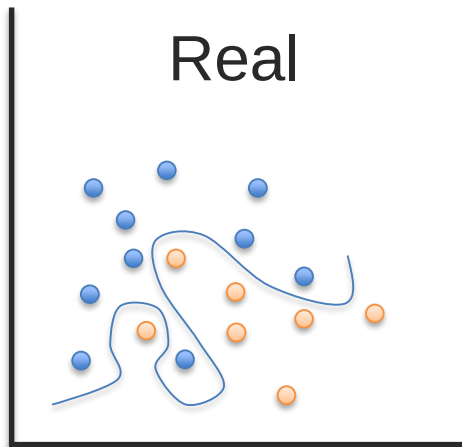
Example



Bias-Variance

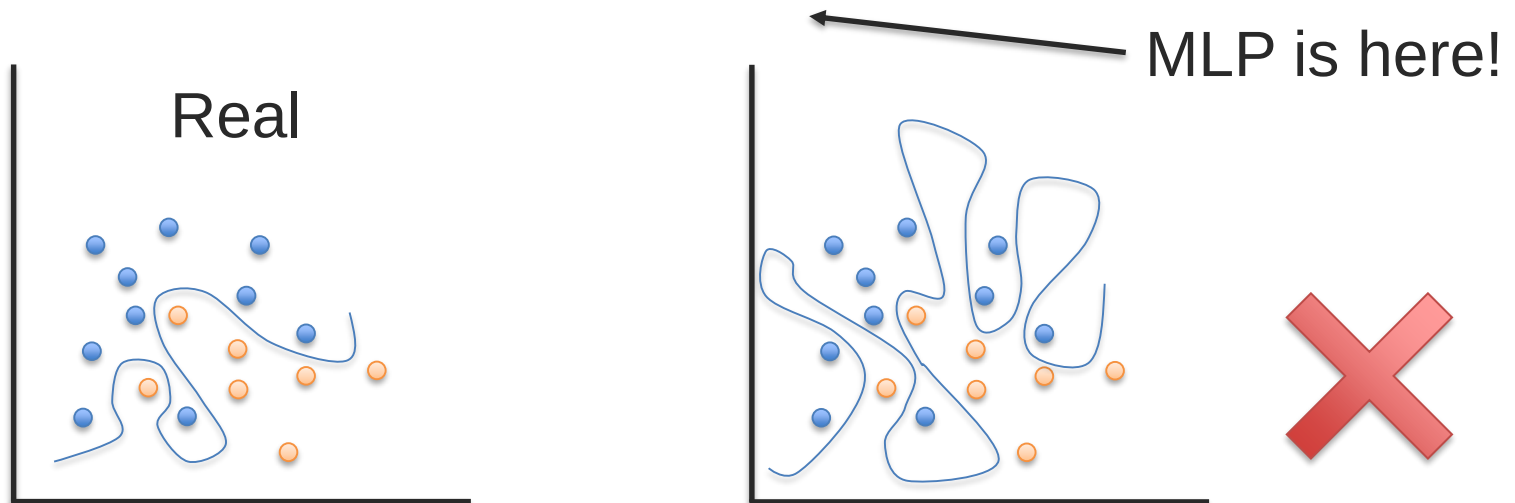
- Problem of bias and variance
 - Simple models are unlikely to find the solution to a hard problem, thus probability of finding the right model is low.

← No longer SOT!



Bias-Variance

- Problem of bias and variance
 - Simple models are unlikely to find the solution to a hard problem, thus probability of finding the right model is low.
 - Complex models find many solutions to a problem, thus probability of finding the right model is again low.



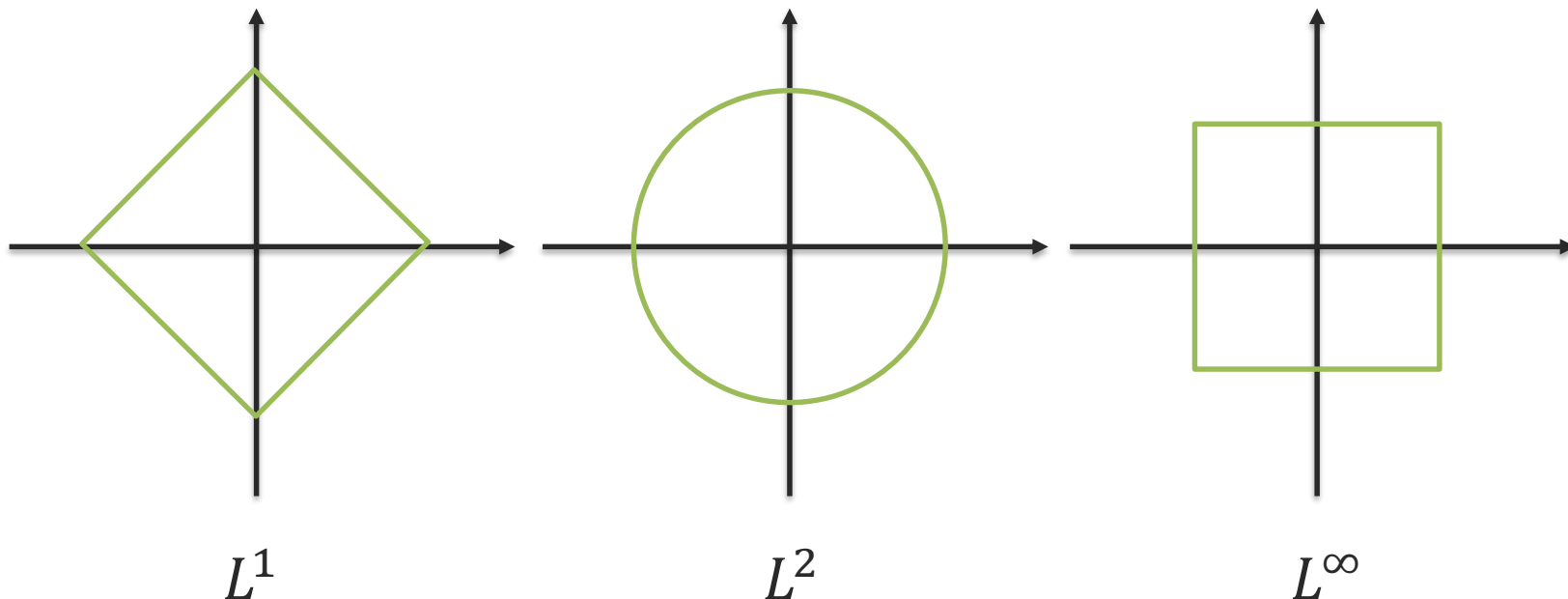
Optimization – Practical Guidelines

- Adaptive Optimization Methods
- Regularization
- Co-adaptation
- Multimodal Optimization



Regularization

- Parameter Regularization:
 - Adding prior to the network parameters
 - L^p Norms



$$\text{Minimize: } \text{Loss}(x; \theta) + \alpha \|\theta\|$$

Parameter Regularization

- Parameter Regularization:
 - L^1 (Lasso) and L^2 (Ridge) are the most famous norms used. Sometimes combined (Elastic)
 - Other norms are computationally ineffective.
- Maximum a posteriori (MAP) estimation:
 - Having priors on the model parameters
 - L^2 can be seen as a Gaussian prior on model parameters θ
 - A generalization of L^2 is called Tikhonov Regularization with Multivariate Gaussian prior on model parameters.
 - Assuming Correlation between parameters one can build a Mahalanobis variation of Tikhonov Regularization.



Structural Regularization

- Lots of models can learn everything.
- Go for simpler ones. 
- Use task specific models:
 - CNNs
 - RecNNs
 - LSTMs
 - GRUs

Occam's razor



Optimization – Practical Guidelines

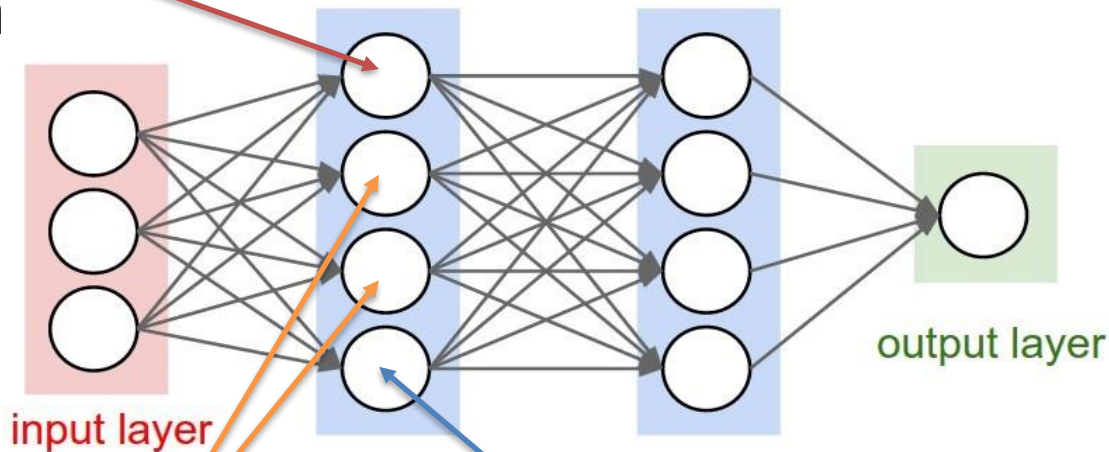
- Adaptive Optimization Methods
- Regularization
- Co-adaptation
- Multimodal Optimization



Example

- A neuron learns something that is not useful:
 1. Learn something useful
 2. Other neurons learn to mitigate it.

Useless
neuron



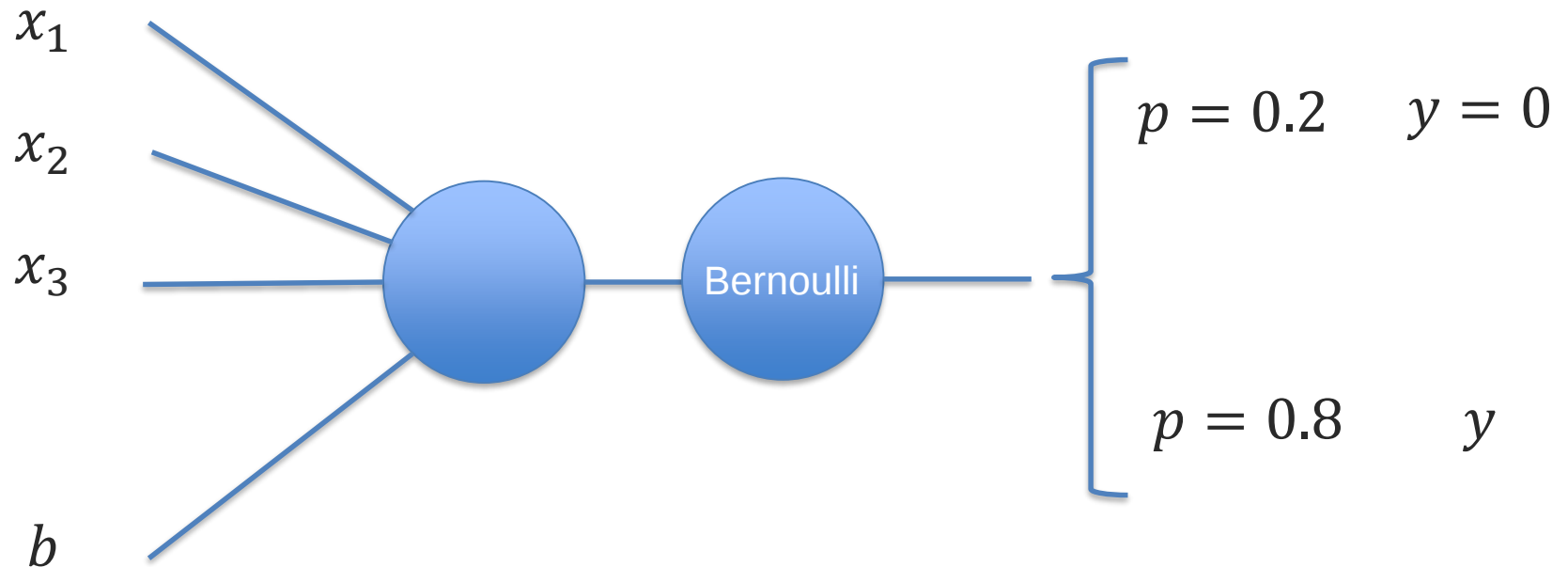
Learning to fight
useless neuron

Actually learning
something



Dropout

- Simply multiply the output of a hidden layer with a mask of 0s and 1s (Bernoulli)



Dropout

➡ Forward step: multiply with a Bernoulli distribution per epoch, batch or sample point. Question: which one works better?

⬅ Backward step: just calculate the gradients same as before. Question: some neurons are out of the network, so how does this work?

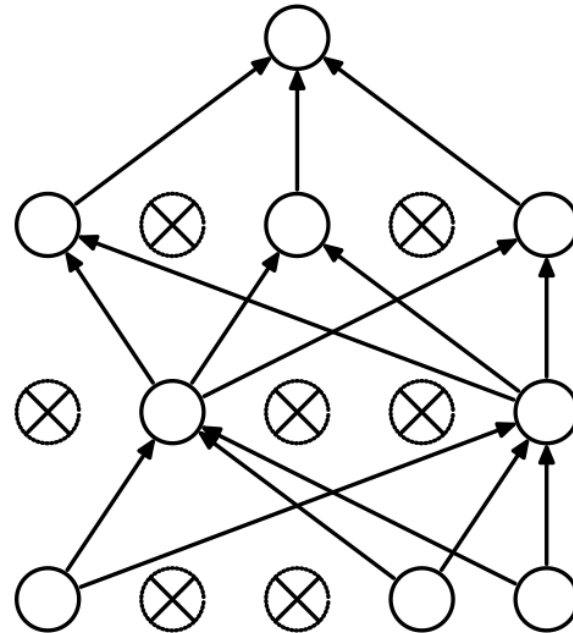
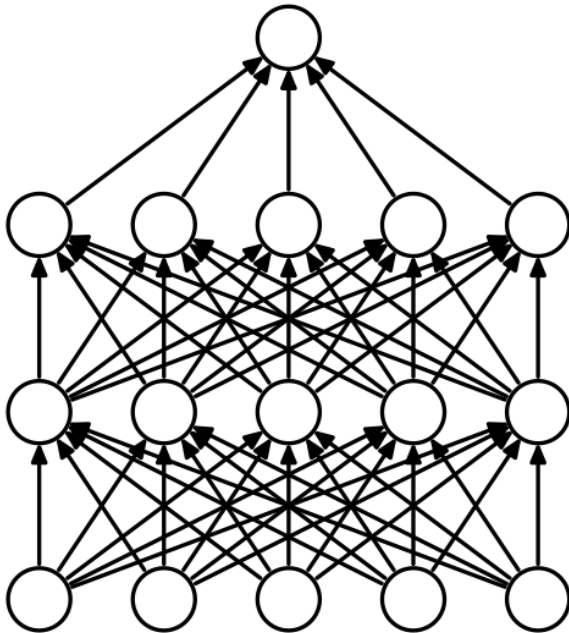
All good? Nope

+ Multiply the weights by $1 - p_i$



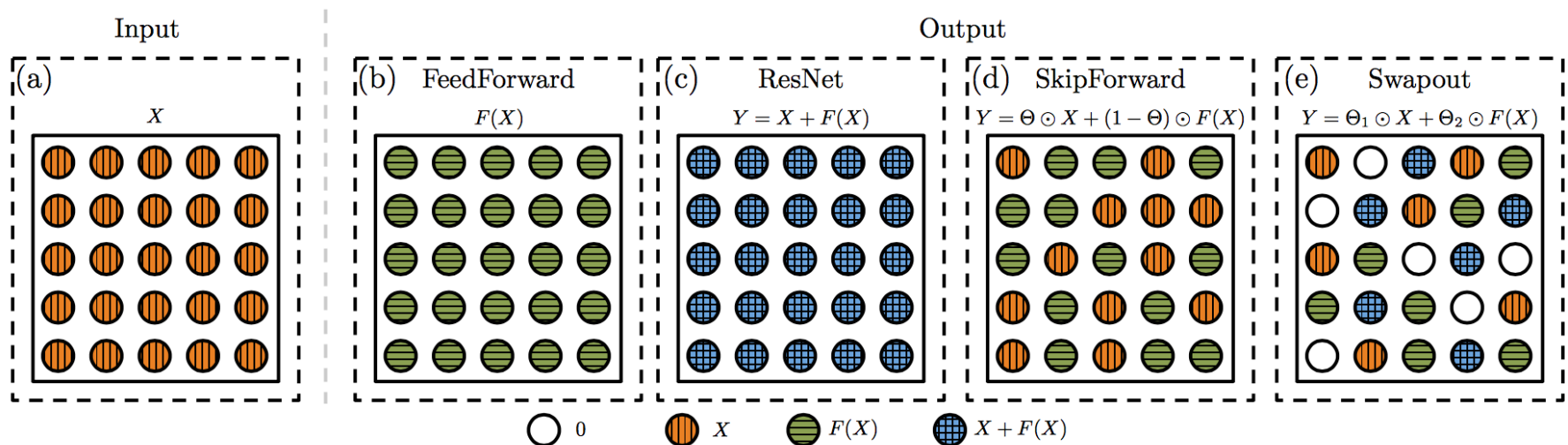
Dropout

Stop co-adaptation + learn ensemble



Other variations

- Gaussian dropout: instead of multiplying with a Bernoulli random variable, multiply with a Gaussian with mean 1.
- Swapout: Allow skip-connections to happen



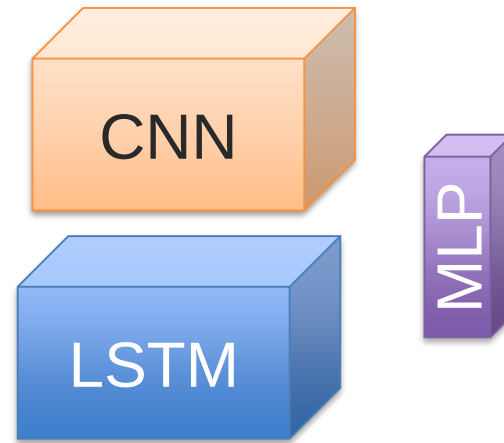
Optimization – Practical Guidelines

- Adaptive Optimization Methods
- Regularization
- Co-adaptation
- Multimodal Optimization



Multimodal Optimization

- Biggest Challenge:
 - Data from different sources
 - Different networks
- Example:
 - Question Answering: LSTM(s) connected to a CNN
 - Multimodal Sentiment: LSTM(s) fused with MLPs and 3D-CNNs
- CNNs work well with high decaying learning rate
- LSTMs work well with adaptive methods and normal SGD
- MLPs are very good with adaptive methods

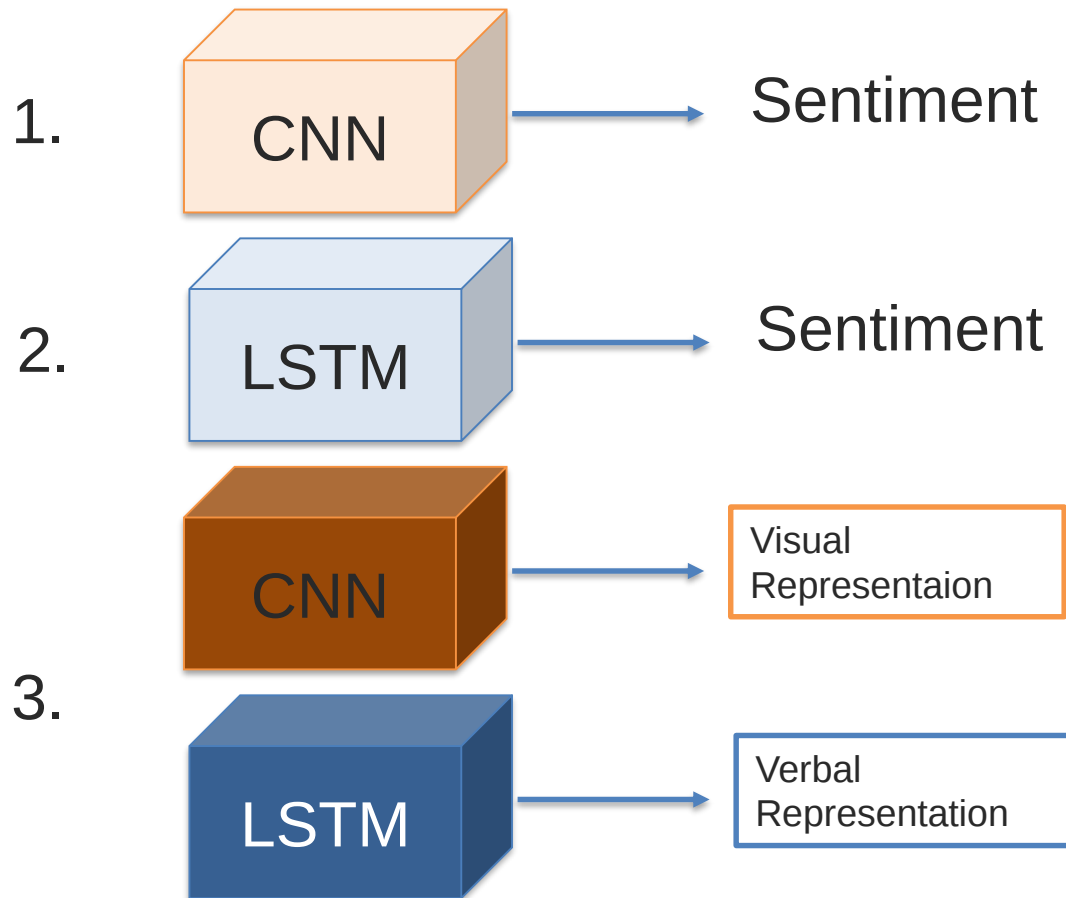


Multimodal Optimization

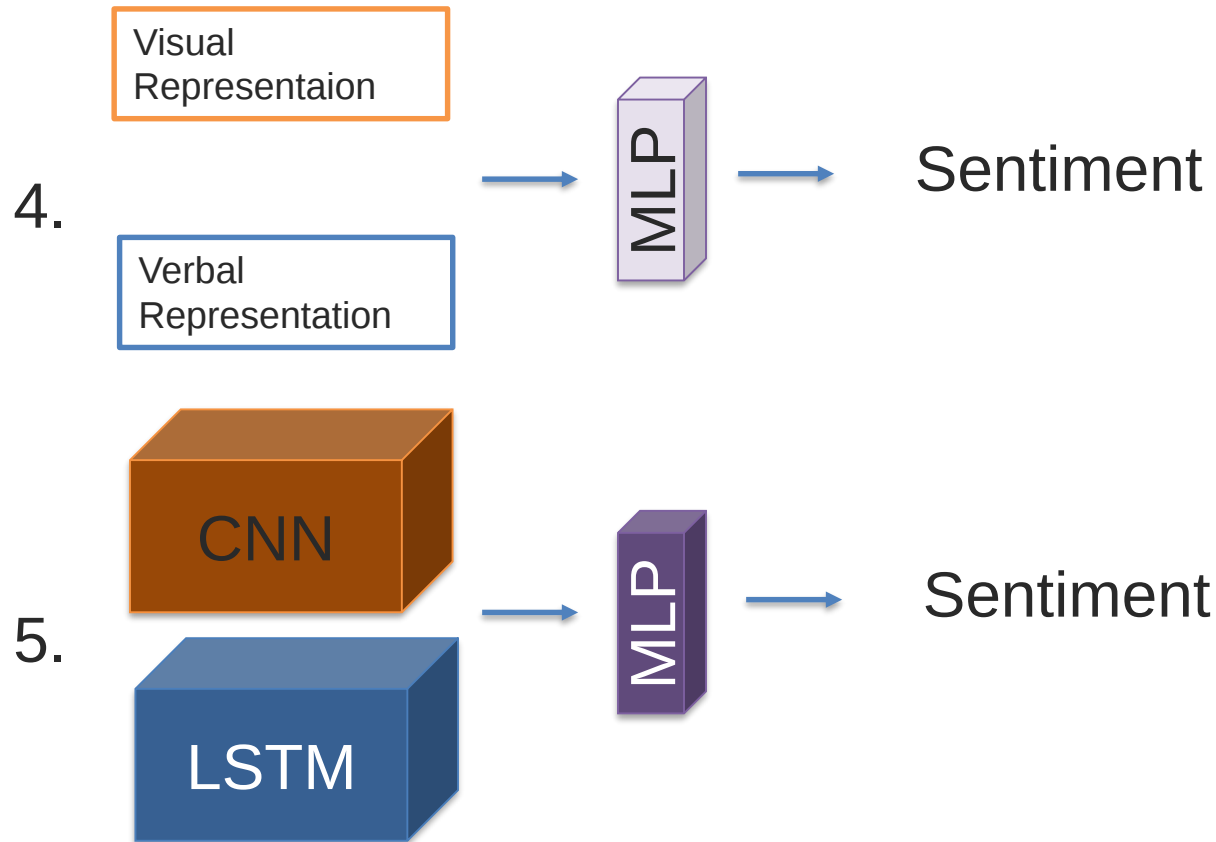
- How to work with all of them?
- Pre-training is the most straight forward way:
 - Train each individual component of the model separately
 - Put together and fine tune
- Example: Multimodal Sentiment Analysis



Pre-training



Pre-training



Pre-training Tricks

- In the final stage (5), it is better to not use adaptive methods such as Adam.
 - Adam starts with huge momentum on all the networks parameters and can destroy the effects of pretraining.
 - Simple SGD mostly helpful.
- Initialization from other pre-trained models:
 - VGG for CNNs
 - Language models for RNNs
 - Layer by layer training for MLPs

