Announcements

- **HW 1** due in 1 week (next Wed 8pm)

- **HW 0** grades released. Regrade requests by 9/30.

- **Worksheet** for linear regression posted.
- **Worksheet** and **Recitation** for logistic regression:
  - Towne 217 ALC on Friday Sep 27, 3pm to 4pm. Recording will be posted.

- **Office hour change:**
  - My OH this Thursday is moved to next Monday (9/30 at 3-4pm).

Lecture 7: Neural Networks (Part 2)

CIS 4190/5190

Fall 2024

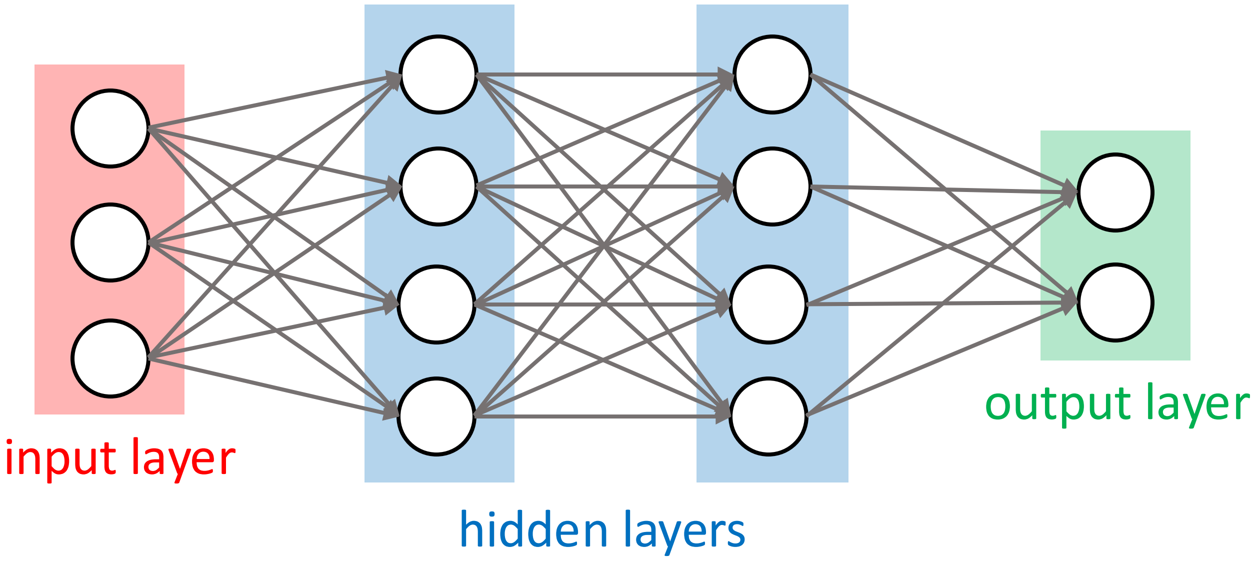
Slides adapted from Chris Callison-Berch and Luke Zettlemoyer and Fei-Fei Li

Agenda

- Recap
- Neural network tips and tricks
- Hyperparameter tuning
- Implementation

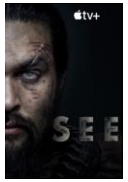
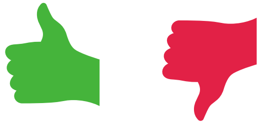
Recap

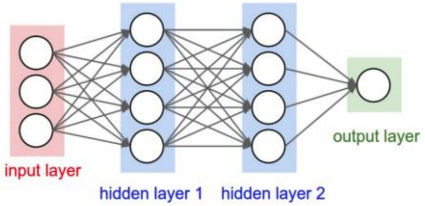
- **Representation Learning**: automatically learn good features for tasks
- **Deep Learning**: learn multiple levels of representation at increasing levels of complexity
- **Feedforward Neural Networks**:



Supervised Learning Setup

Specification

- Data  : 

- Model 

- Loss - function of (model parameters, data)
  - Maximize the probability of the data

End-to-end Learning

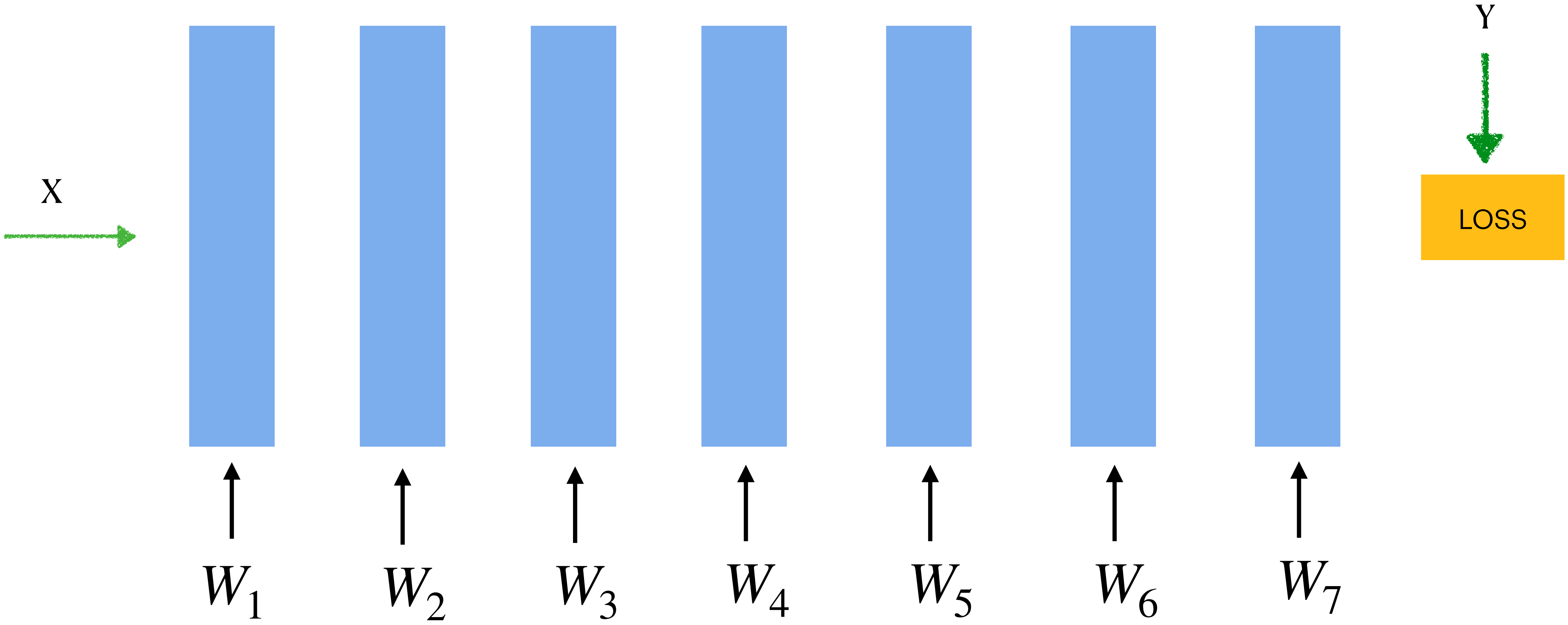
- Optimize objective over data
  - Learn All Network Parameters

- Gradient Based Optimization

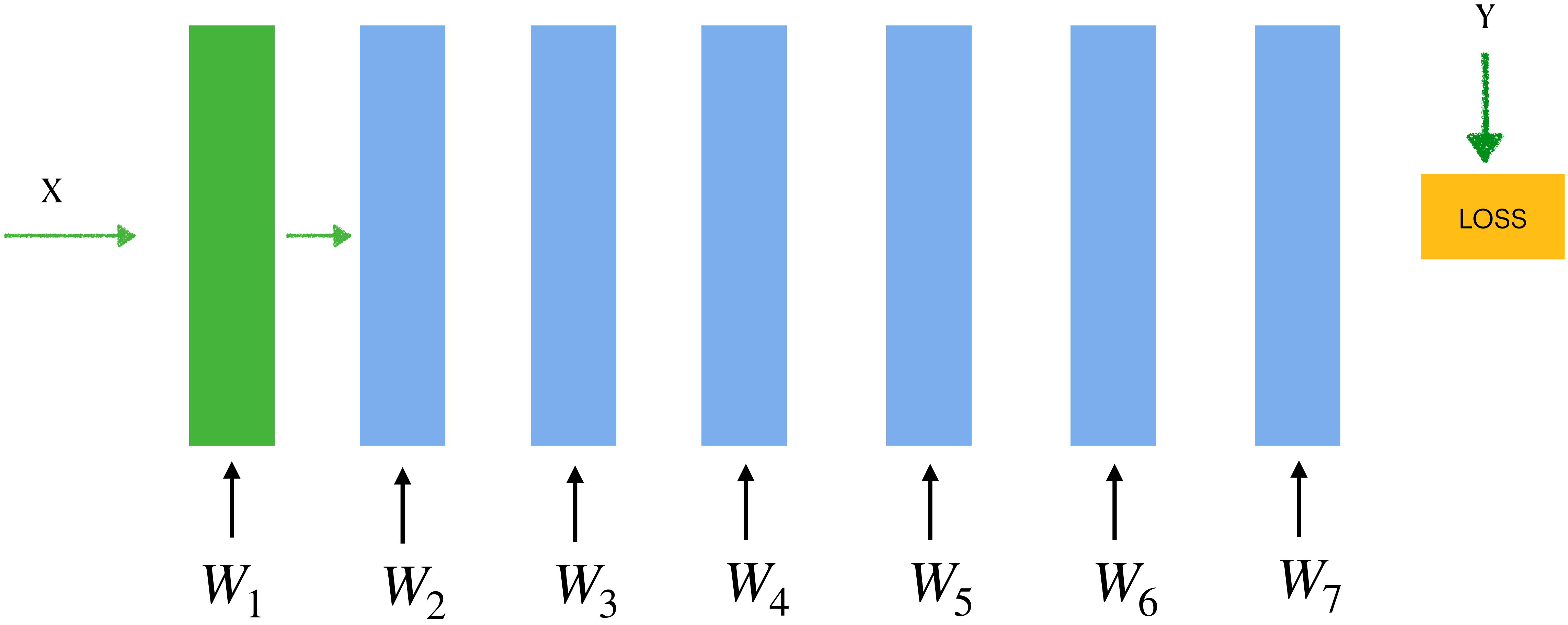
$$\theta^{(t+1)} = \theta^{t} - \mu \nabla_{\theta} L(\theta)$$

- Gradients via **Backpropagation**

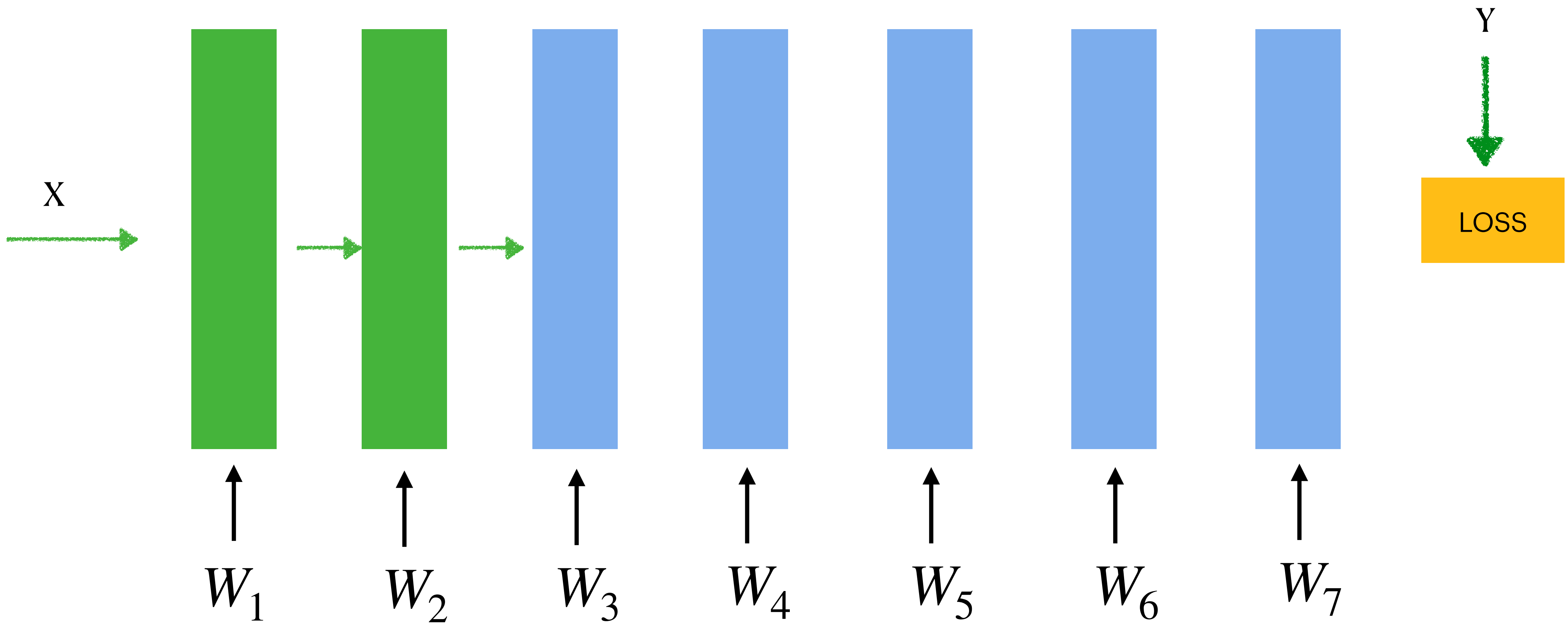
Forward Computation



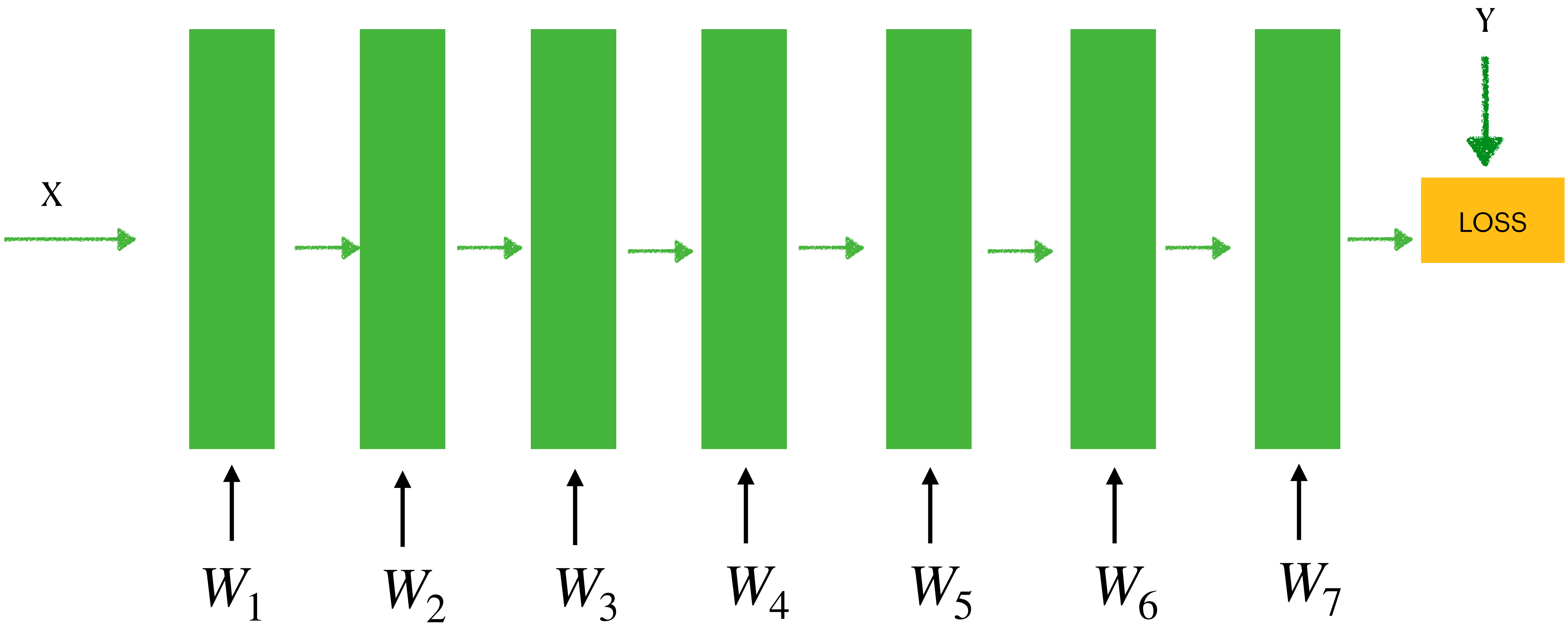
Forward Computation



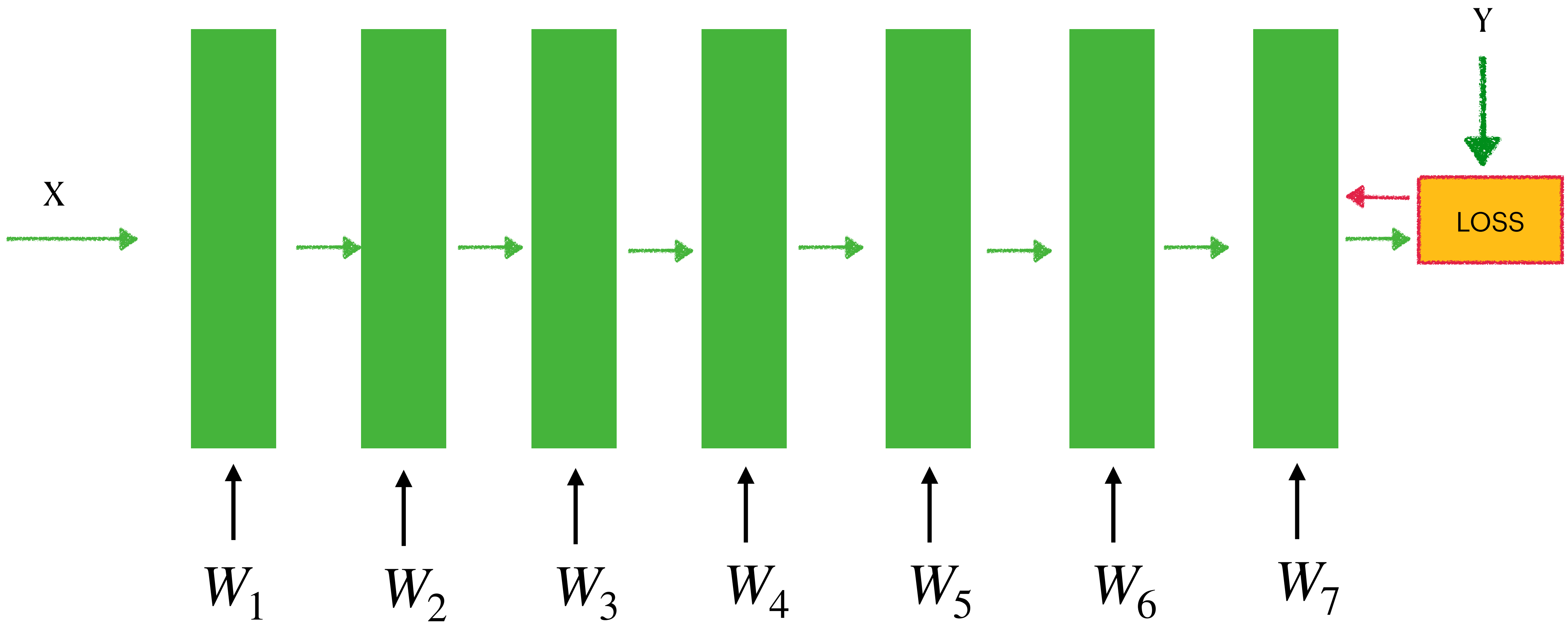
Forward Computation



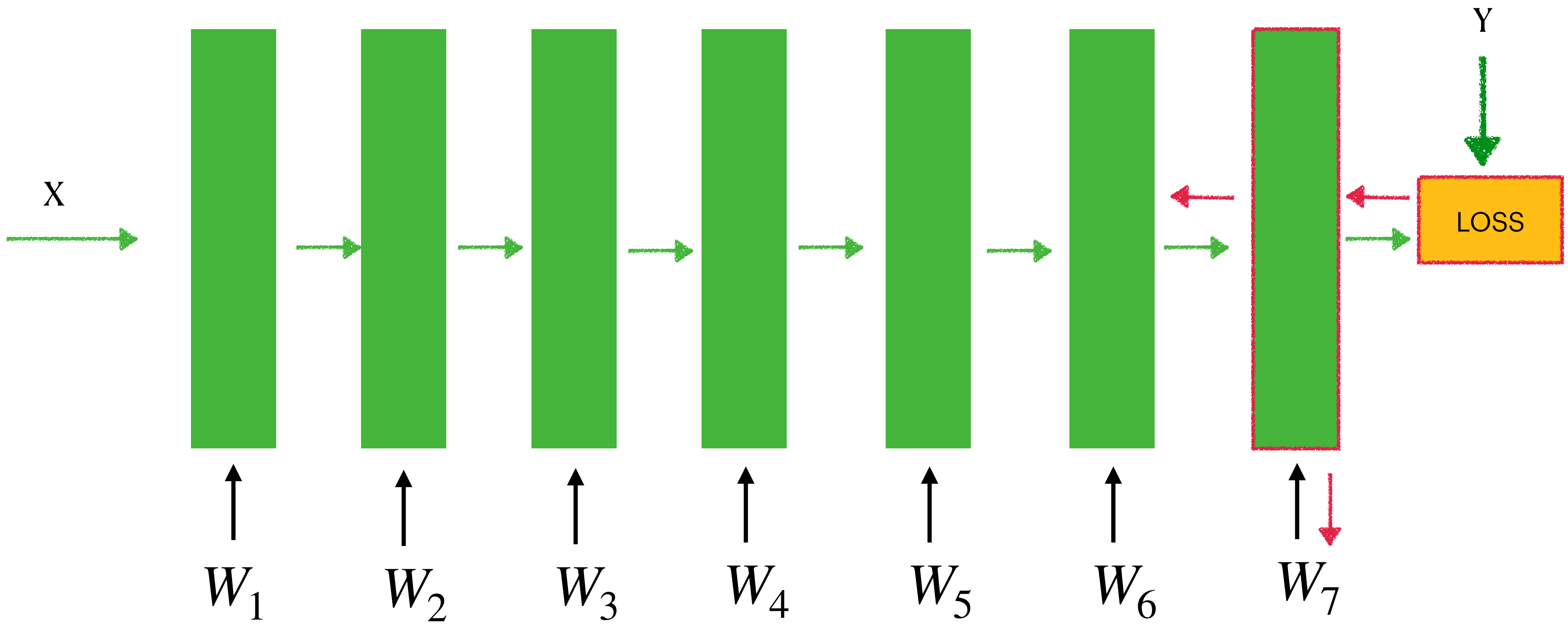
Forward Computation



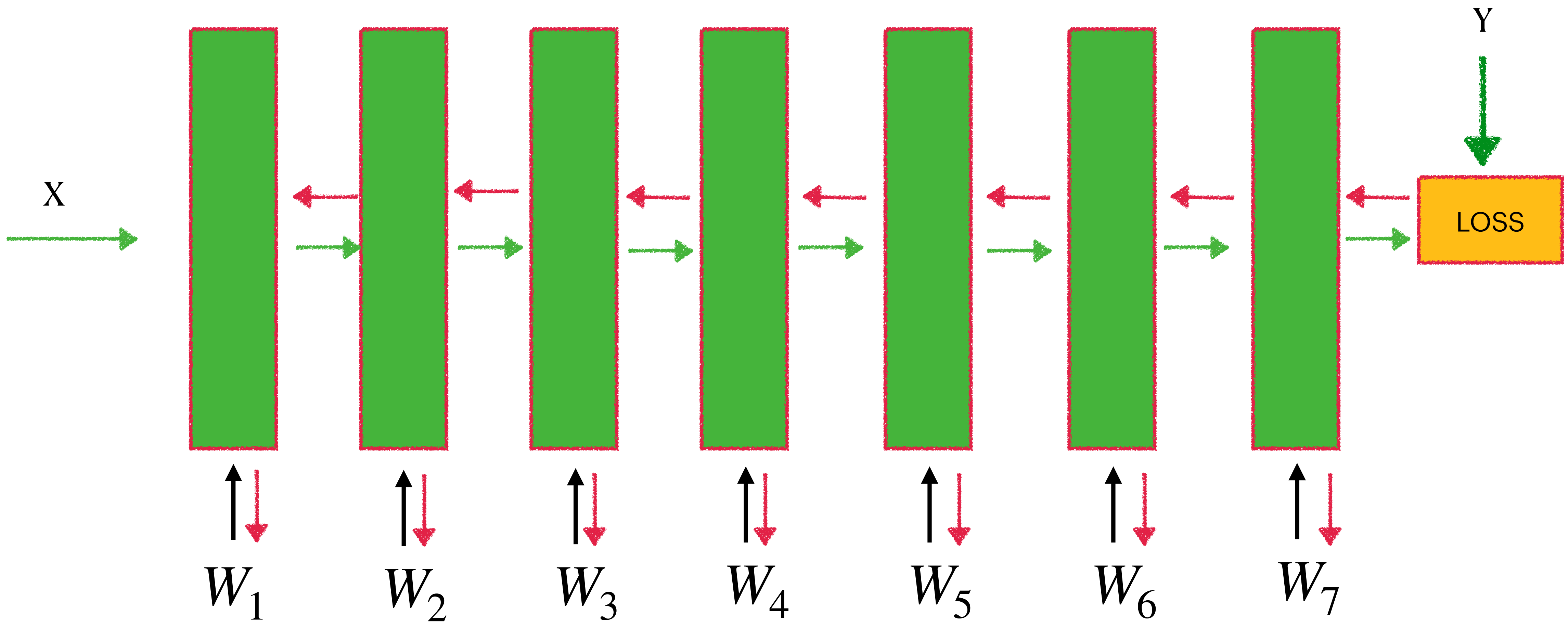
Backward Computation



Backward Computation

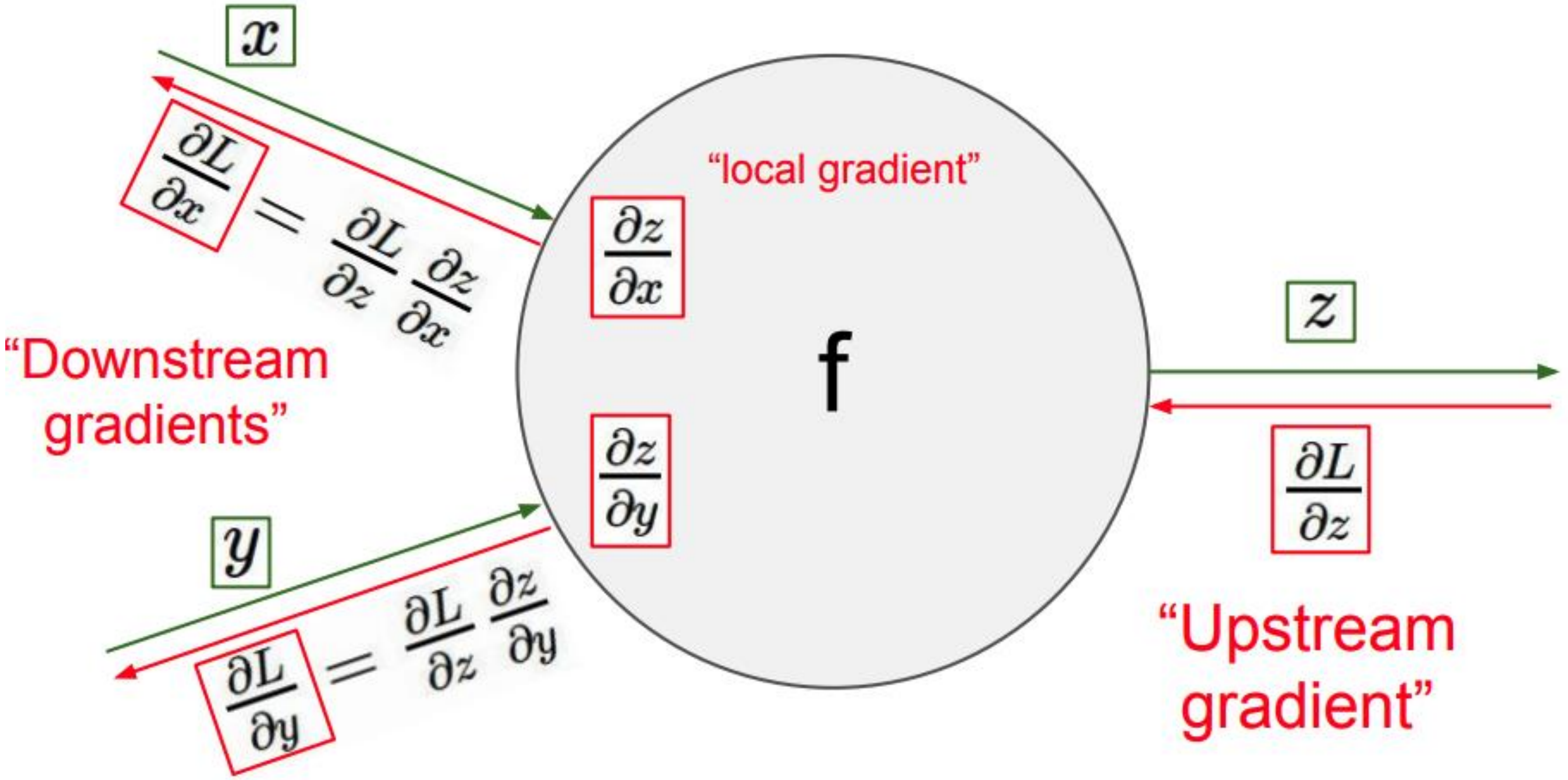


Backward Computation



Backward Pass

General Rule



If x and y are inputs and parameters, we are done.
Otherwise, continue propagating gradients backward

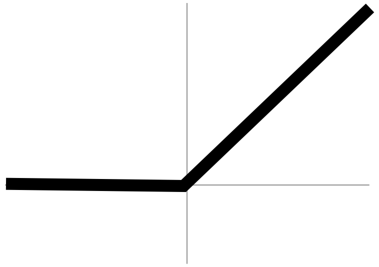
http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

Agenda

- Recap
- Neural network tips and tricks
- Hyperparameter tuning
- Implementation

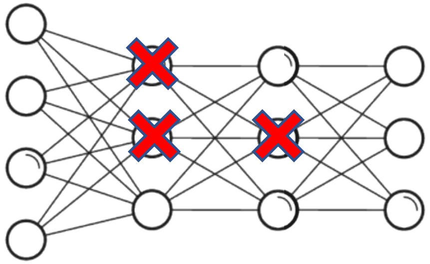
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

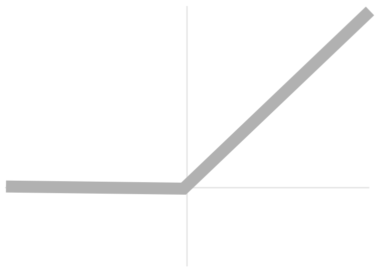


Dropout

Managing Training

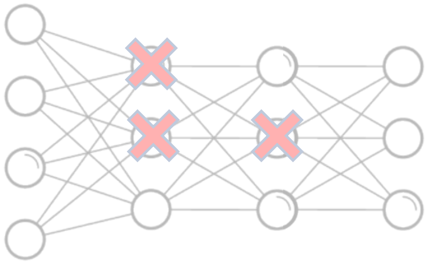
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



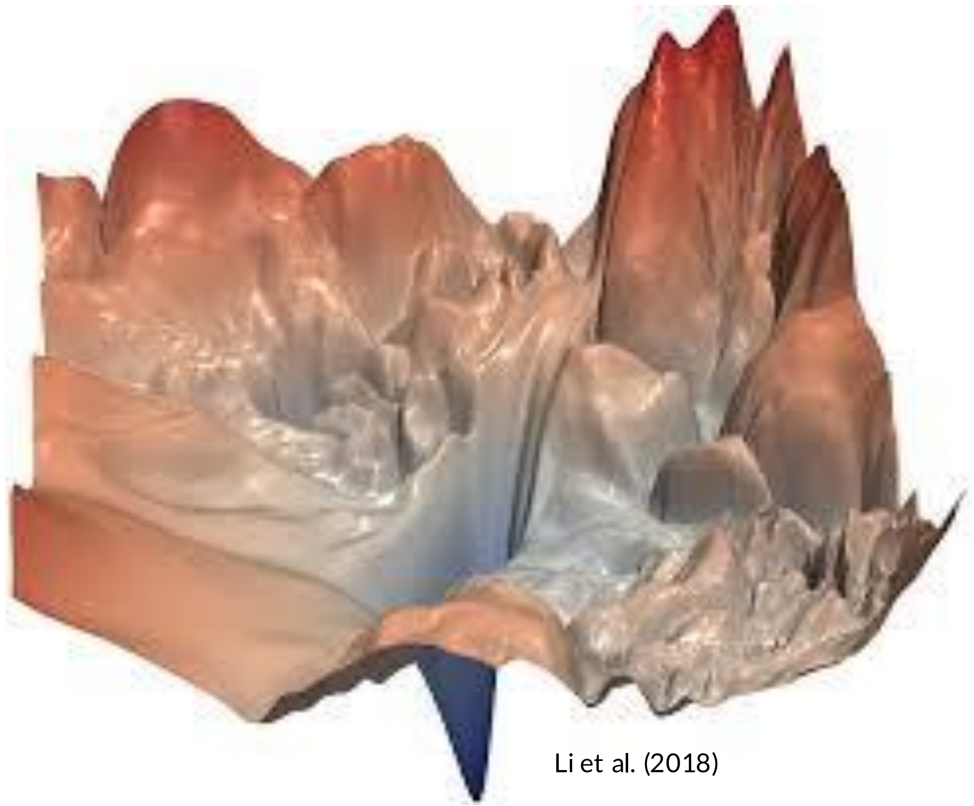
Dropout

Managing Training

Optimization Challenges

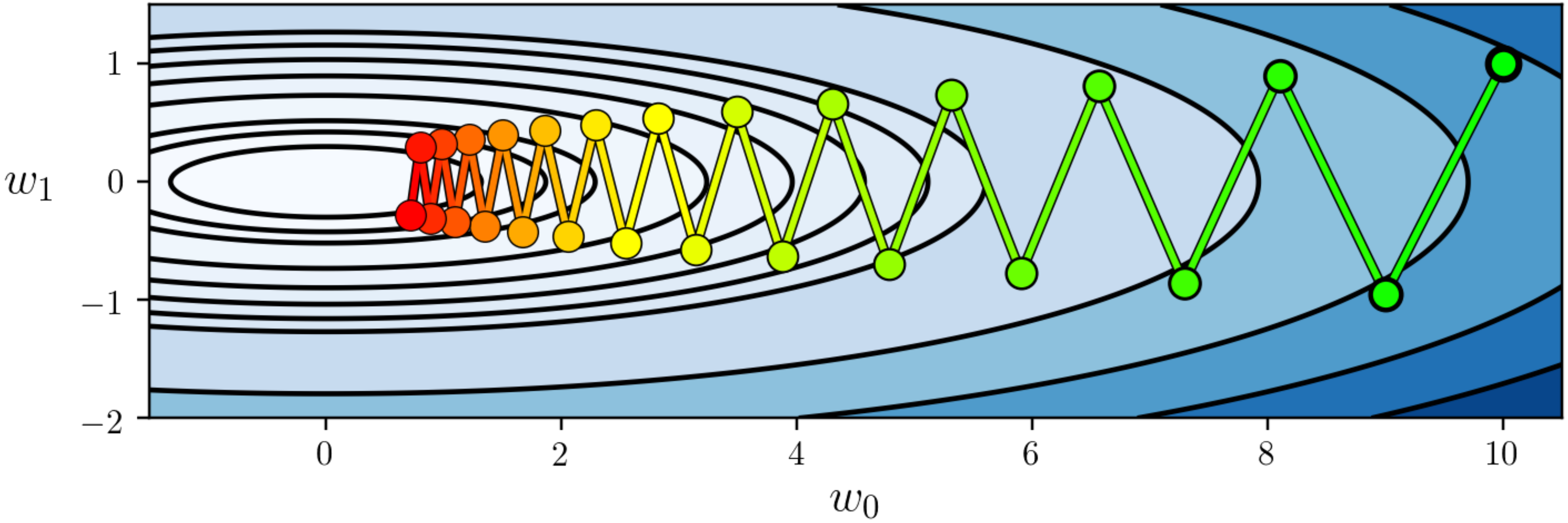
- **Challenges**
  - Local minima, saddle points due to non-convex loss
  - Exploding/vanishing gradients
  - Ill-conditioning

- Have heuristics that work in common cases (but not always)



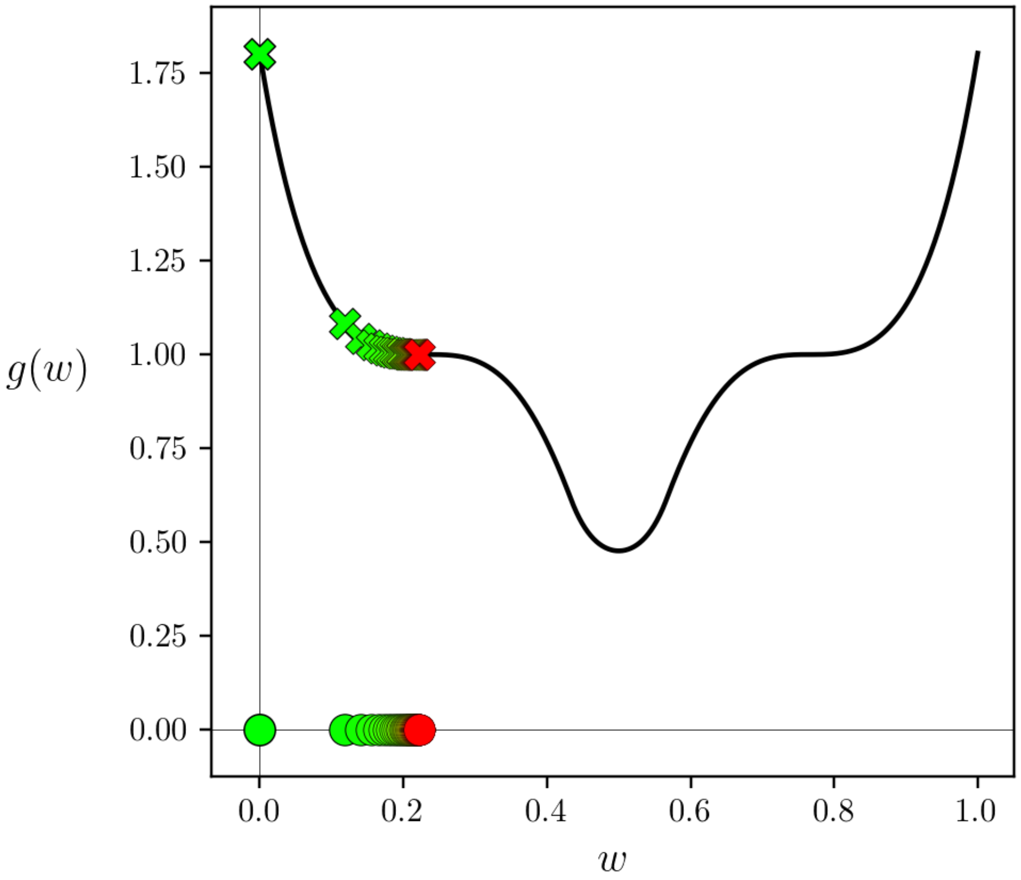
Li et al. (2018)

Challenge 1: Narrow Valleys



https://jermwatt.github.io/machine_learning_refined/

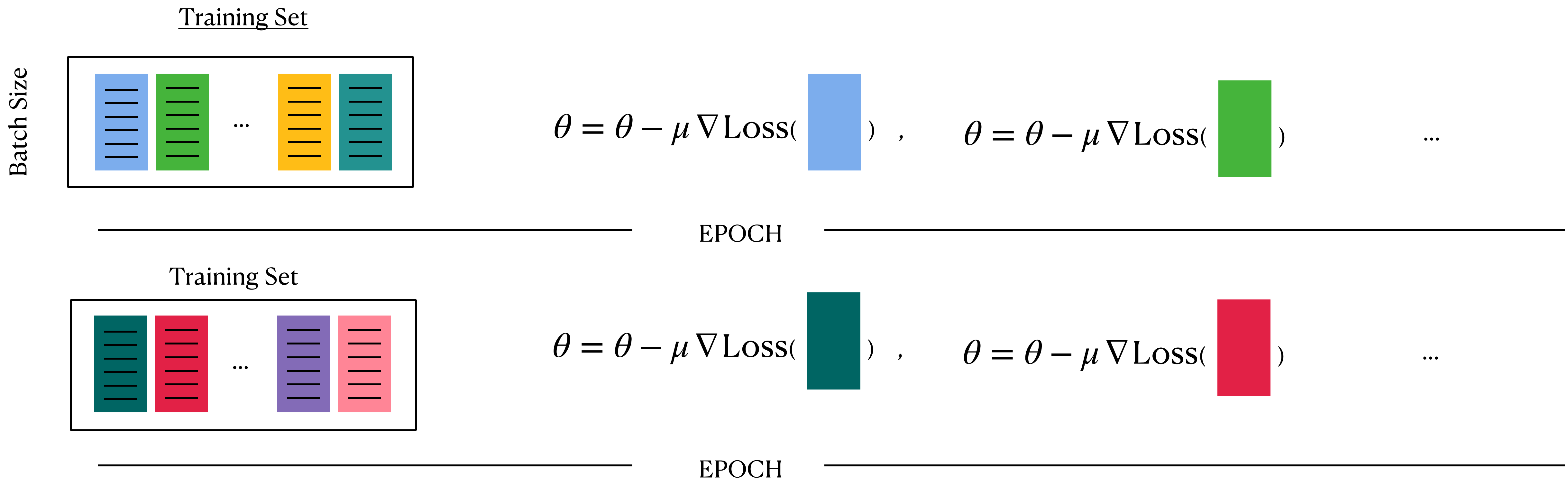
Challenge 2: Saddle Points



https://jermwatt.github.io/machine_learning_refined/

How Do We Optimize?
Gradient Descent with Mini-Batches

While not converged, on dev, sample data in pieces (batches) updating model



Accelerated Gradient Descent

- Vanilla gradient descent:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta L(f_\theta(x), y)$$

- Accelerated gradient descent (momentum):

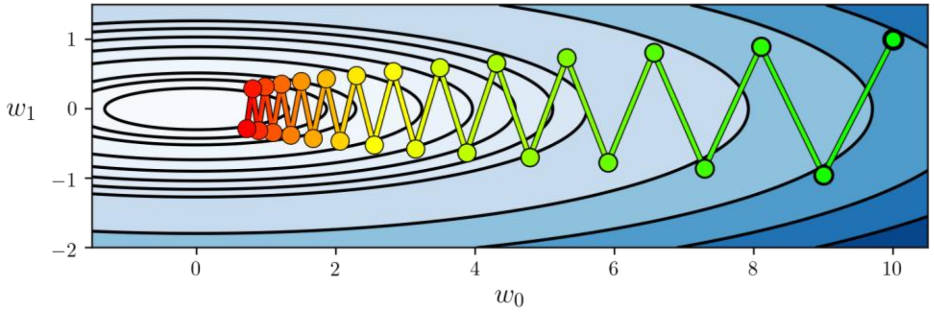
$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\theta L(f_\theta(x), y)$$
$$\theta \leftarrow \theta + \rho$$

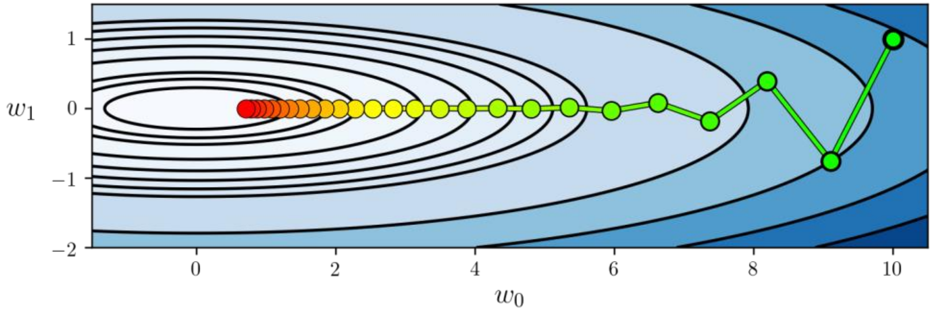
Accelerated Gradient Descent

- **Intuition:** ρ holds the previous update $\alpha \cdot \nabla_\theta L(f_\theta(x), y)$, except it "remembers" where it was heading via momentum

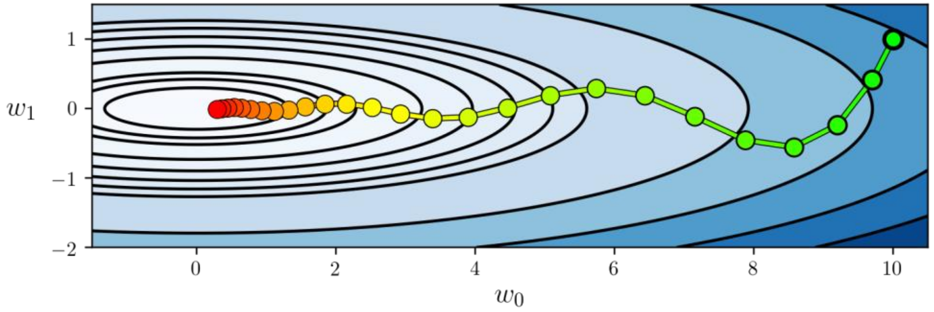
- New hyperparameter μ (typically $\mu = 0.9$ or $\mu = 0.99$)

Accelerated Gradient Descent





$\mu = 0.7$



$\mu = 0.2$

https://jermwatt.github.io/machine_learning_refined/

Nesterov Momentum

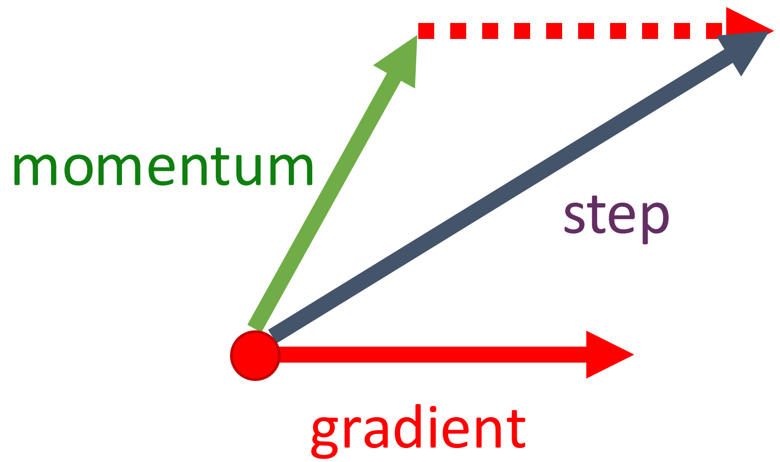
- Accelerated gradient descent:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\theta L(f_\theta(x), y)$$
$$\theta \leftarrow \theta + \rho$$

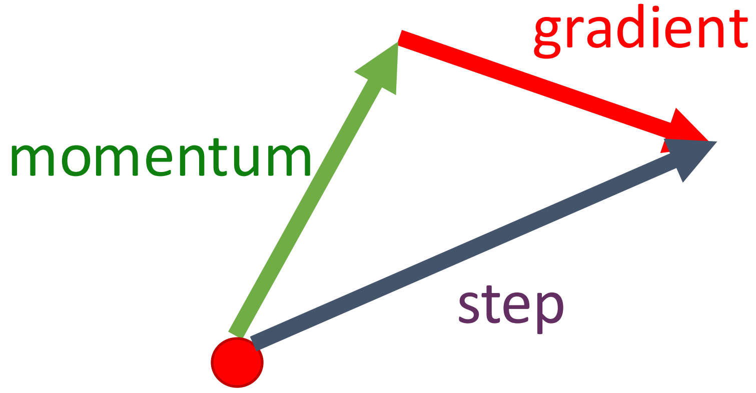
- Nesterov momentum:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\theta L\left(f_{\theta + \mu \cdot \rho}(x), y\right)$$
$$\theta \leftarrow \theta + \rho$$

Nesterov Momentum



vanilla momentum



Nesterov momentum

“Lookahead” helps avoid overshooting when close to the optimum

Adaptive Learning Rates

- **AdaGrad:** Letting $g = \nabla_\beta L\big(f_\beta(x), y\big)$, we have

$$G \leftarrow G + g^2 \quad \text{and} \quad \theta \leftarrow \theta - \frac{\alpha}{\sqrt{G}} \cdot g$$

vector

- **RMSProp:** Use exponential moving average instead:

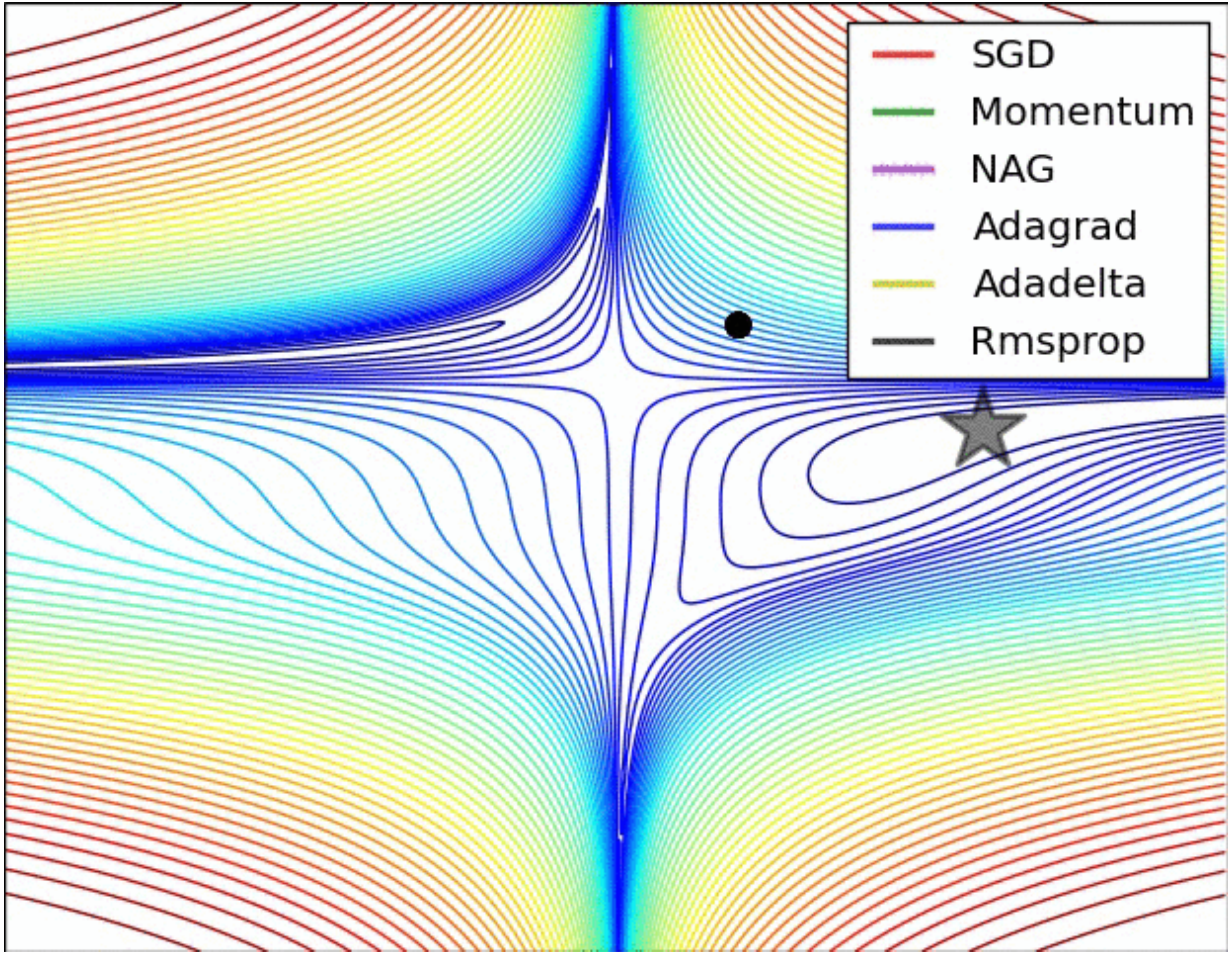
$$G \leftarrow \lambda \cdot G + (1-\lambda) g^2 \quad \text{and} \quad \beta \leftarrow \beta - \frac{\alpha}{\sqrt{G}} \cdot g$$

Adaptive Learning Rates

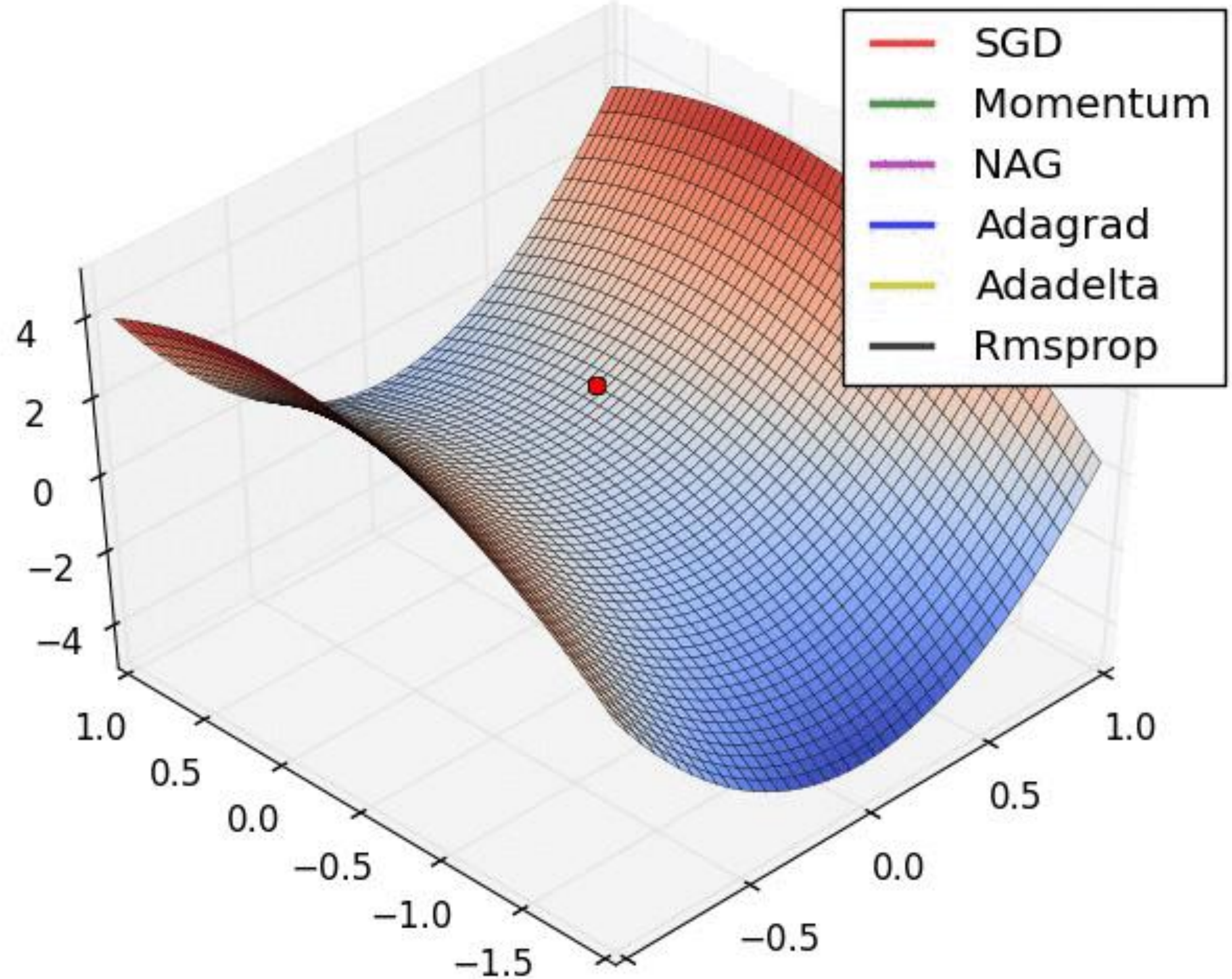
- **Adam:** Similar to RMSprop, but with both the first and second moments of the gradients

$$G \leftarrow \lambda \cdot G + (1-\lambda) \cdot g^2$$
$$g' \leftarrow \lambda' \cdot g' + (1-\lambda') \cdot g$$
$$\theta \leftarrow \theta - \alpha \cdot \frac{g'}{\sqrt{G}}$$

- **Intuition:** RMSProp with momentum
- Most commonly used optimizer



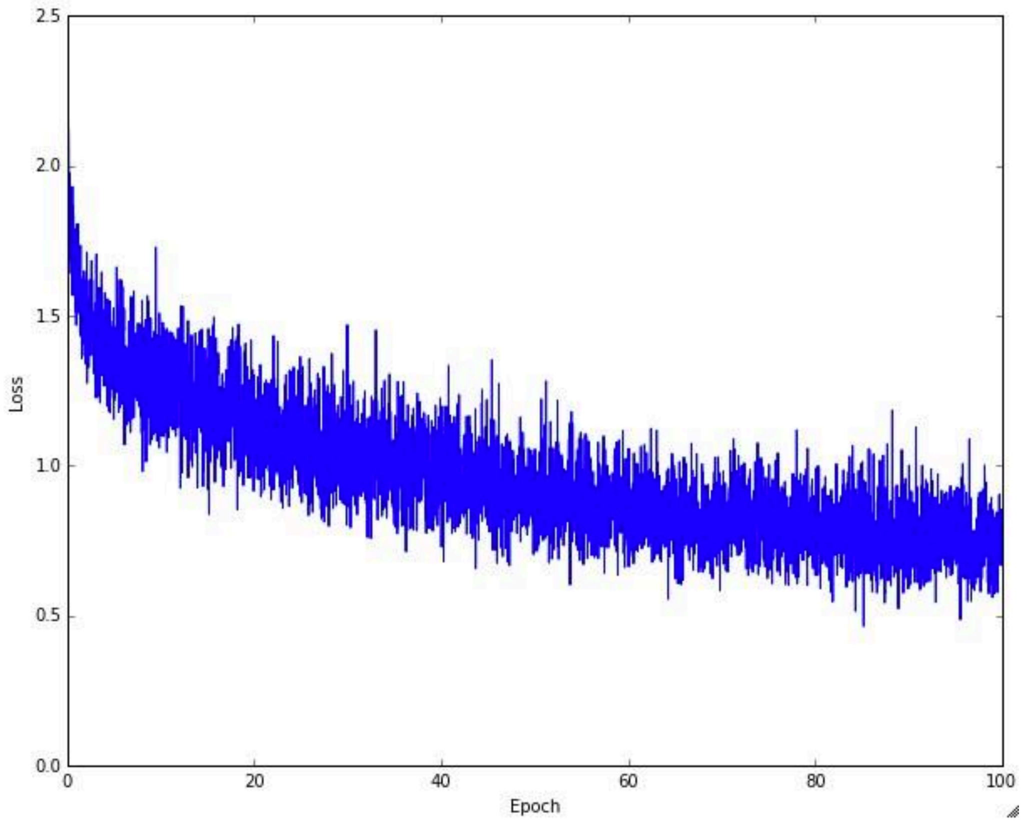
http://cs231n.github.io/neural-networks-3 (Alec Radford)

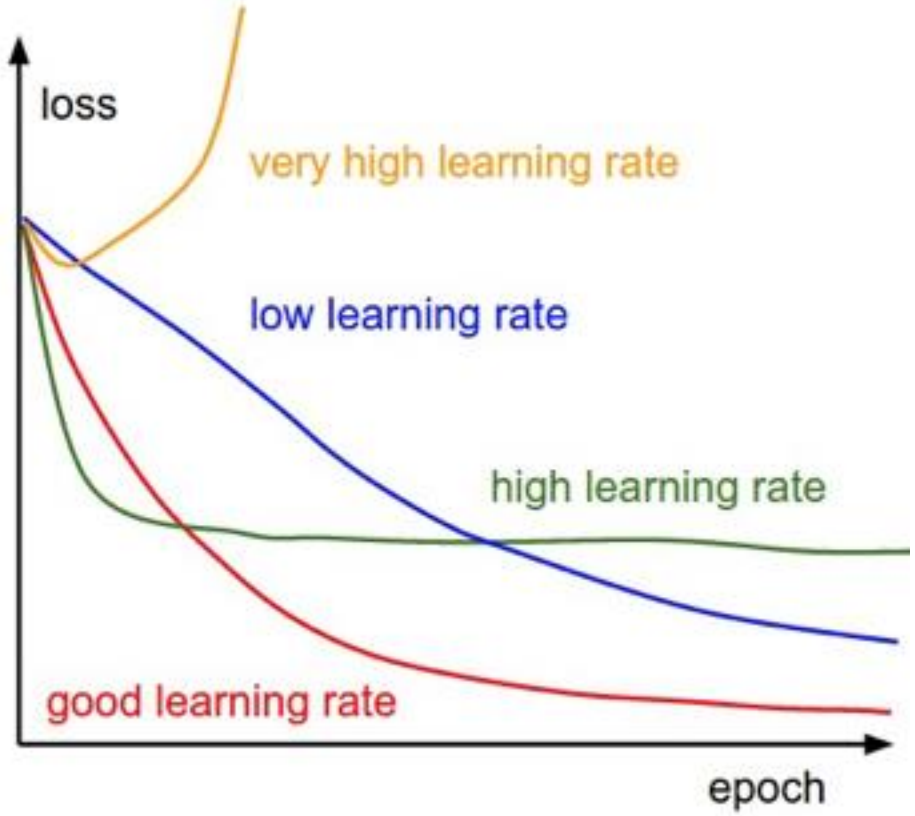


http://cs231n.github.io/neural-networks-3 (Alec Radford)

Learning Rate

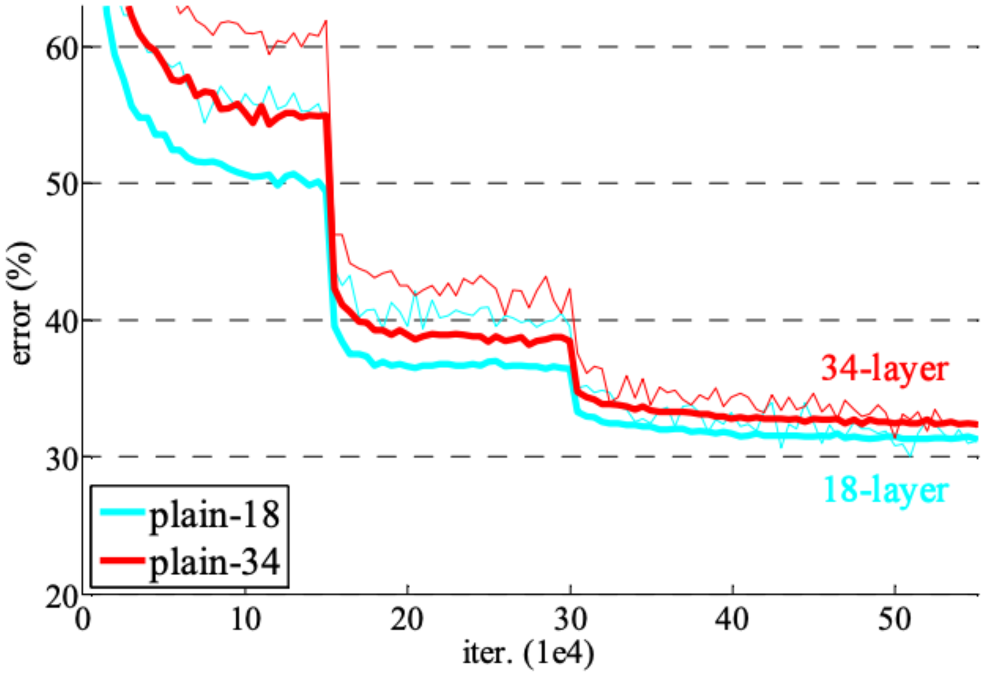
- Most important hyperparameter; tune by looking at training loss





Learning Rate

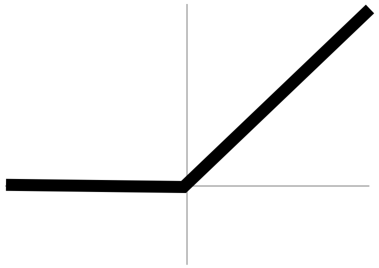
- **Schedules:** Reducing the learning rate every time the validation loss stagnates can be very effective for training



He et al, Residual Networks, 2015

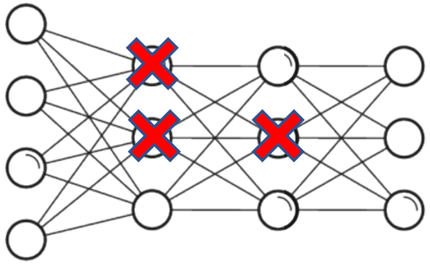
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

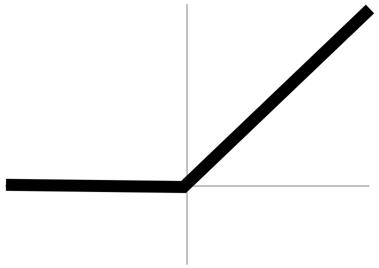


Dropout

Managing Training

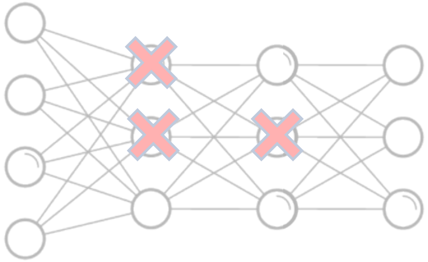
Neural Network Tips & Tricks

Optimization



Activation Functions

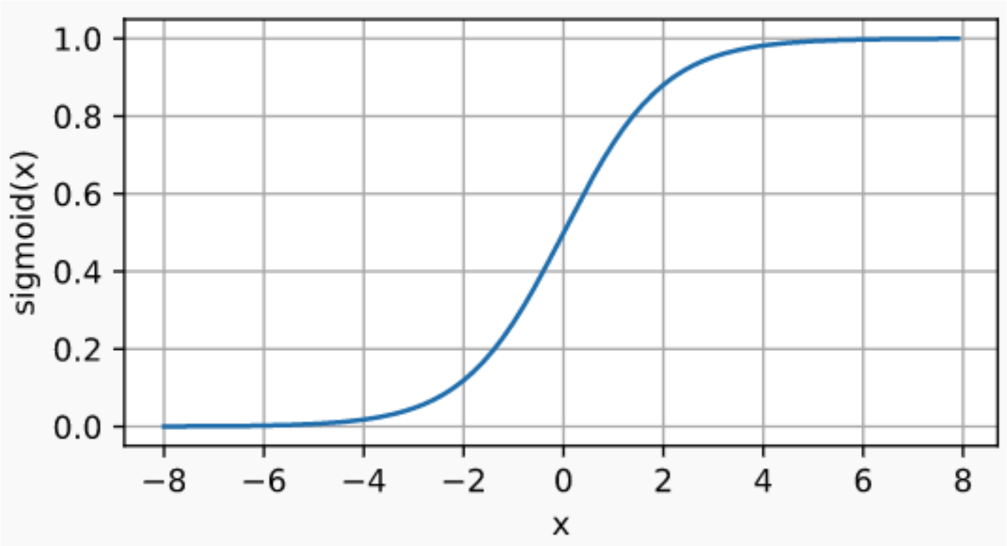
Managing Weights



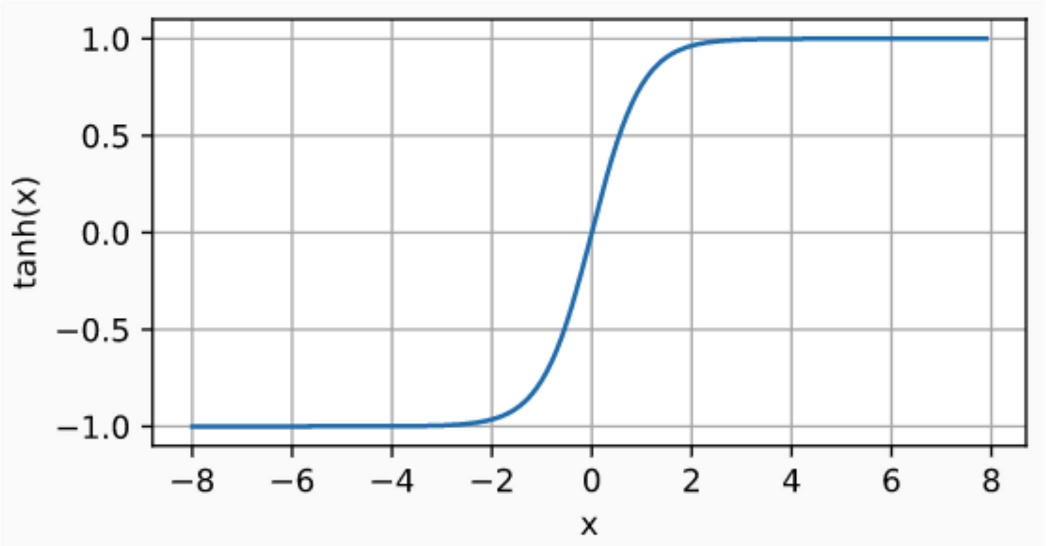
Dropout

Managing Training

Historical Activation Functions



sigmoid



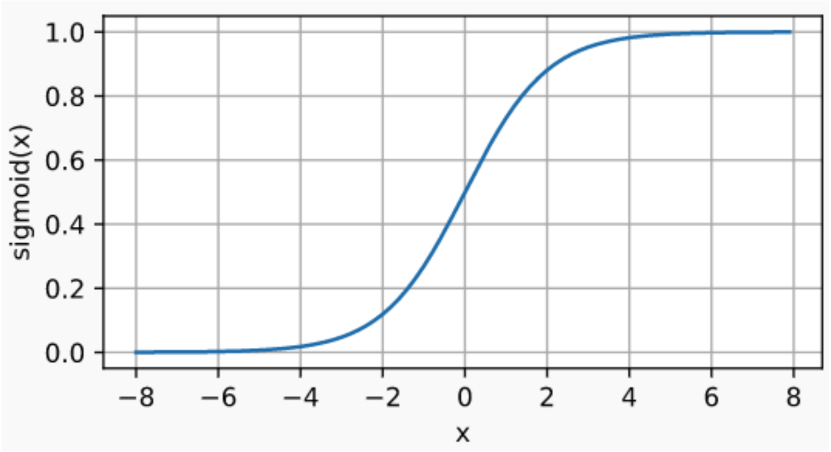
tanh

Vanishing Gradient Problem

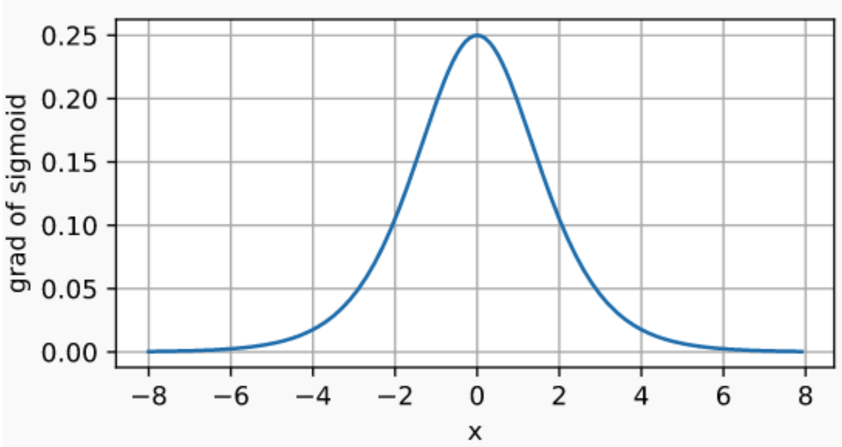
- The gradient of the sigmoid function is often nearly zero

- **Recall:** In backpropagation, gradients are products of local gradients

- **Quickly multiply to zero!**
  - Early layers update very slowly



sigmoid



sigmoid gradient

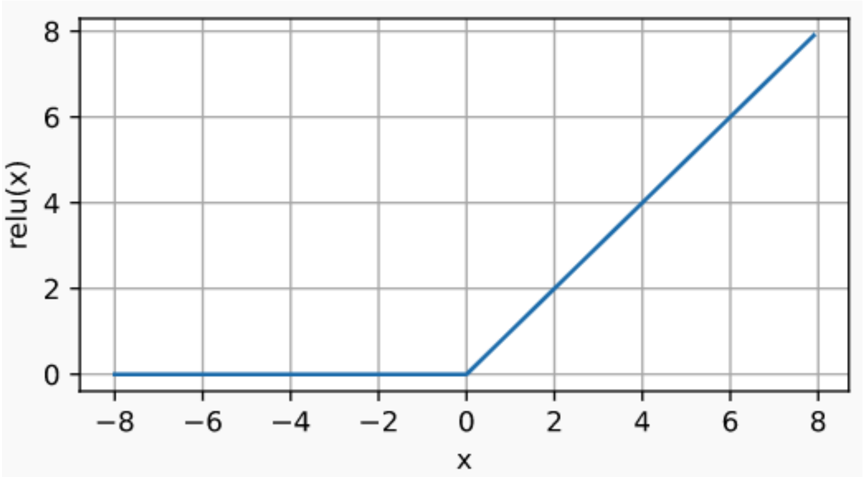
ReLU Activation

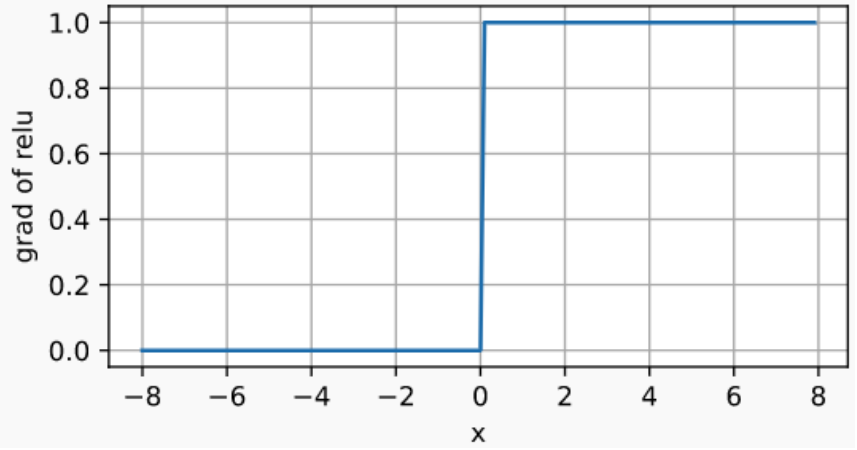
- Activation function

$$g(z) = \max\{0, z\}$$

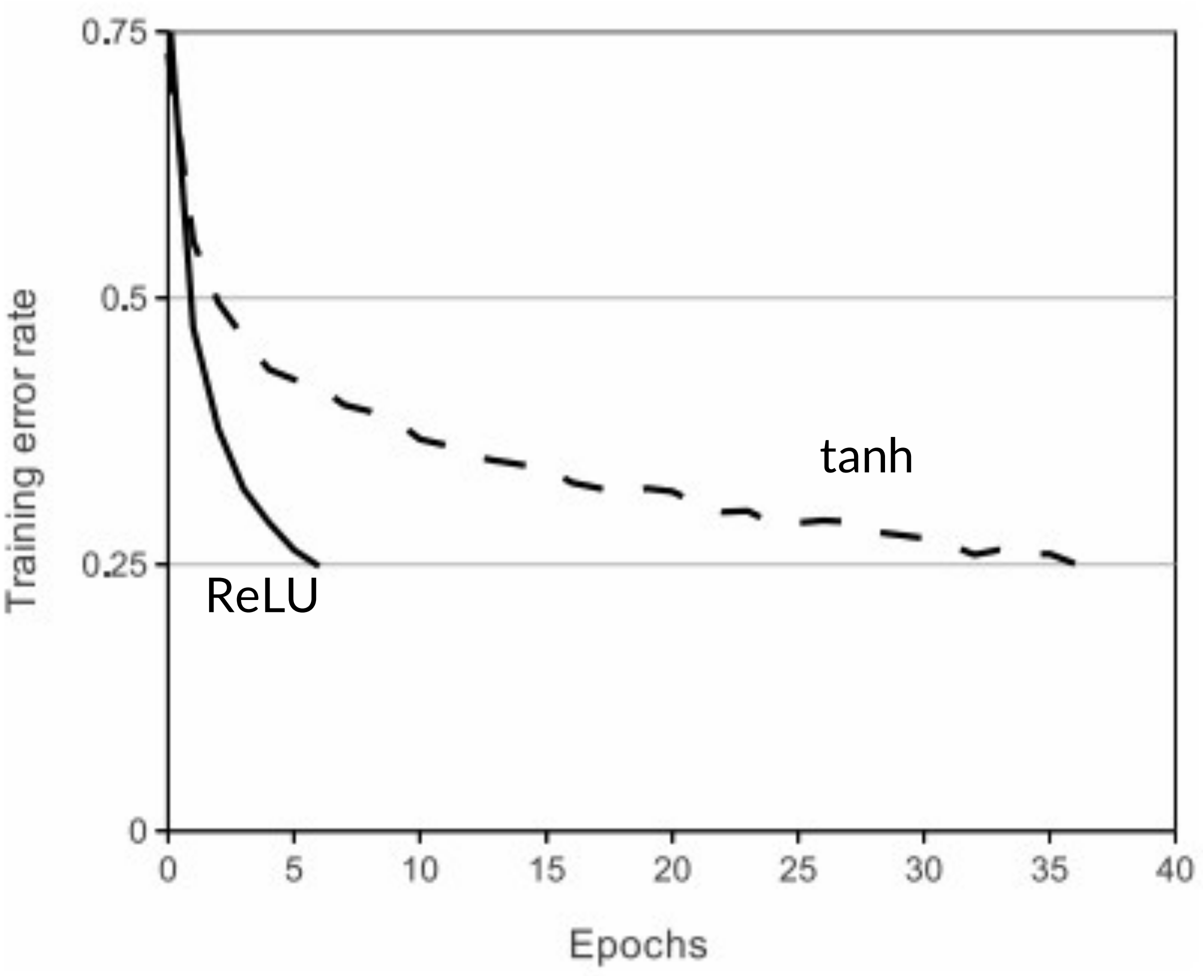
- Gradient now positive on the entire region $z \geq 0$

- Significant performance gains for deep neural networks

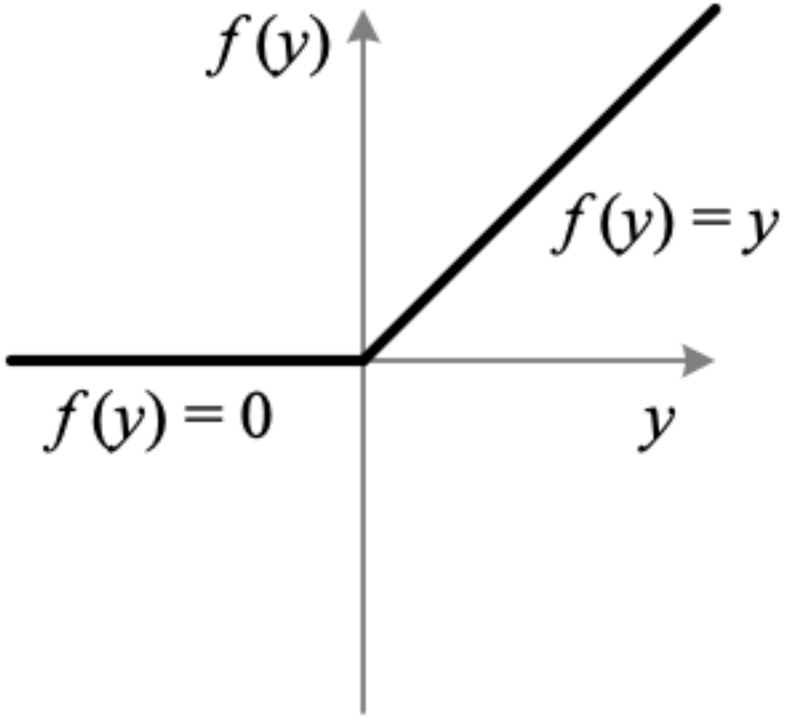


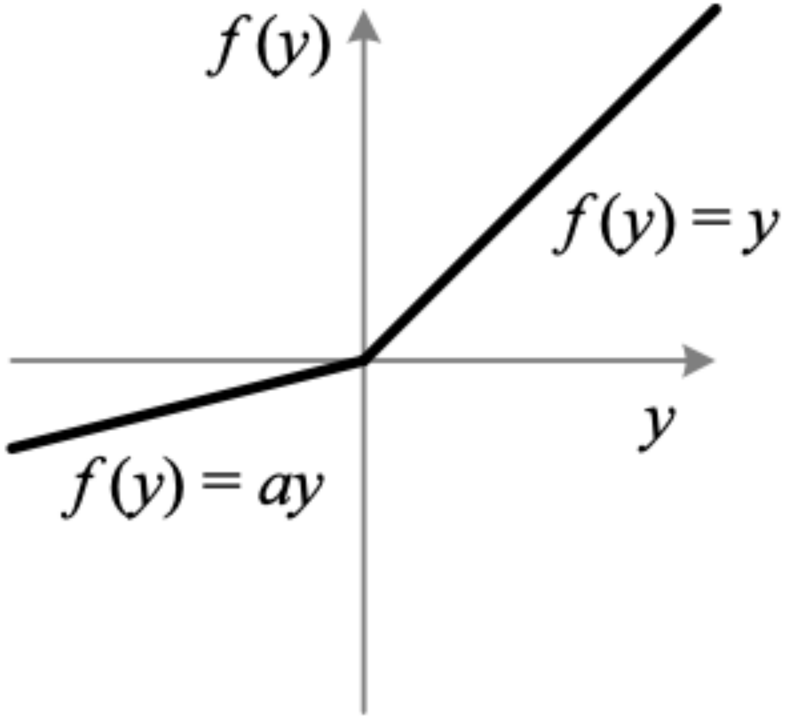


ReLU Activation



Leaky ReLU Activation



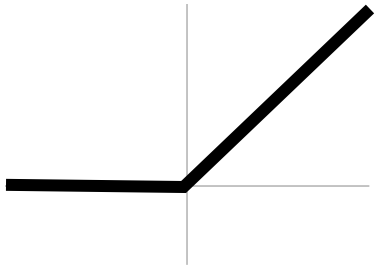


Activation Functions

- ReLU is a good standard choice
- Tradeoffs exist, and new activation functions are still being proposed

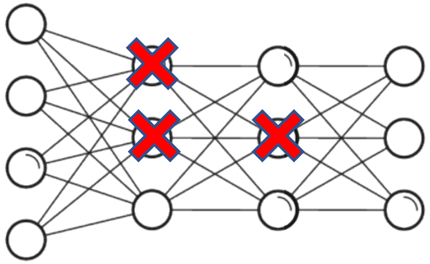
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

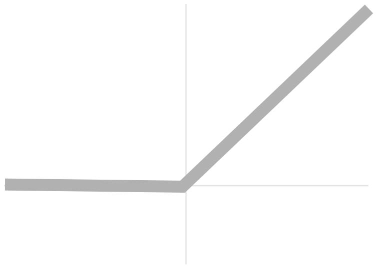


Dropout

Managing Training

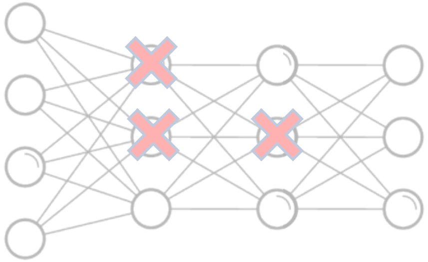
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

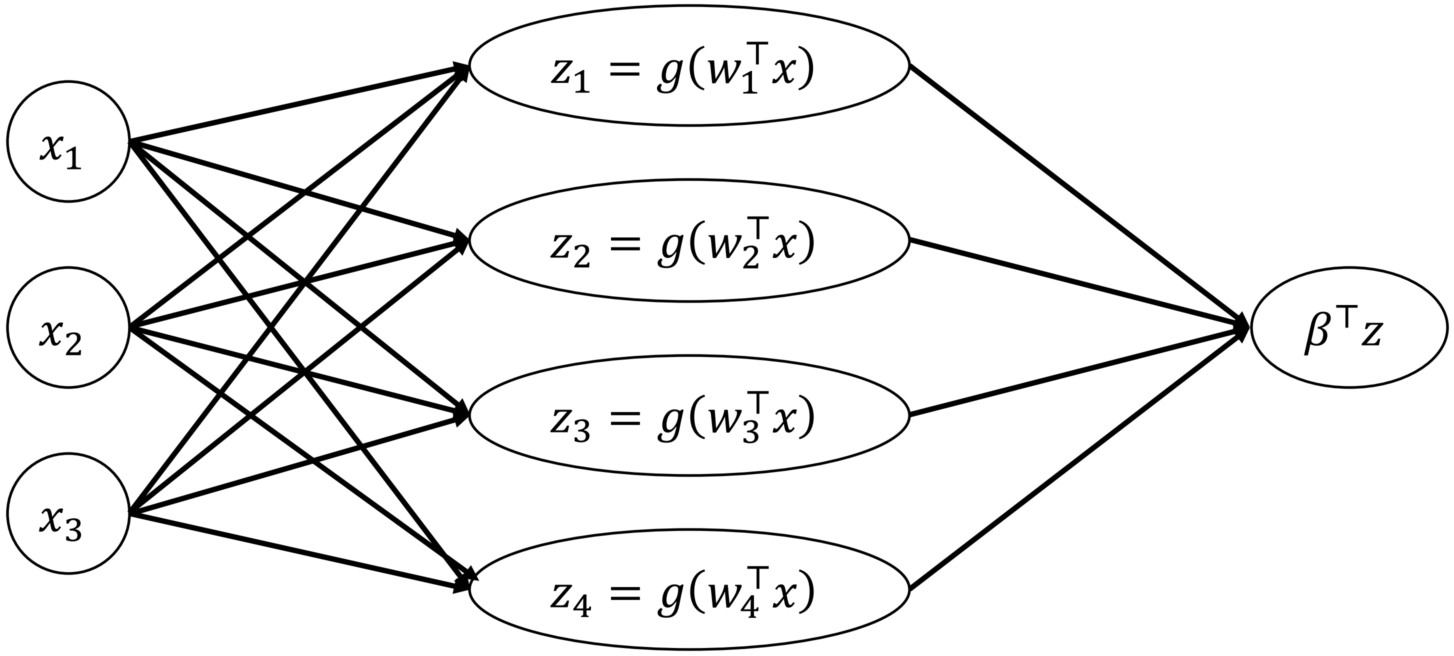


Dropout

Managing Training

Weight Initialization

- **Zero initialization:** Very bad choice!
  - All neurons $z_i = g\left(w_i^\top x\right)$ in a given layer remain identical
  - **Intuition:** They start out equal, so their gradients are equal!



Weight Initialization

- Long history of initialization tricks for W_j based on "fan in" d_{in}
  - Here, d_{in} is the dimension of the input of layer W_j
  - **Intuition:** Keep initial layer inputs $z^{(j)}$ in the "linear" part of sigmoid
  - **Note:** Initialize intercept term to 0

- **Kaiming initialization (also called "He initialization")**
  - For ReLU activations, use $W_j \sim N\left(0, \frac{2}{d_{\text{in}}}\right)$

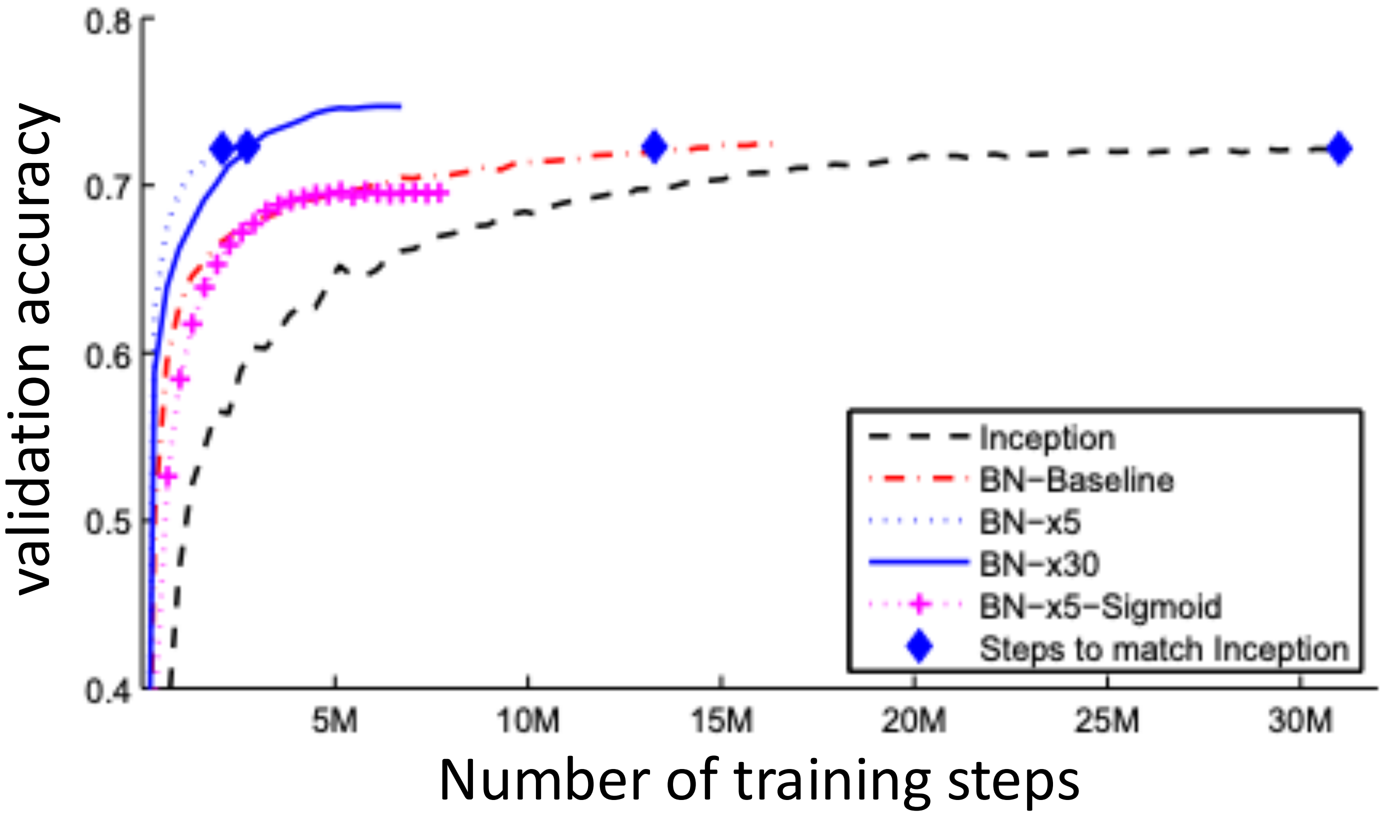
- **Xavier initialization**
  - For tanh activations, use $W_j \sim N\left(0, \frac{1}{d_{\text{in}}+d_{\text{out}}}\right)$ (d_{out} is output dimension)

Batch Normalization

- **Problem**
  - During learning, the distribution of inputs to each layer are shifting (since the layers below are also updating)
  - This cause the objective to have a lot irregularity and hard to take large steps in the loss landscape

- **Solution**
  - As with feature standardization, standardize inputs to each layer to $N(0, I)$
  - **Batch norm:** Compute mean and standard deviation of current minibatch and use it to normalize the current layer (this is differentiable!)
  - **Note:** Needs nontrivial mini-batches or will divide by zero
  - Apply after every layer (typically before activation)

Batch Normalization



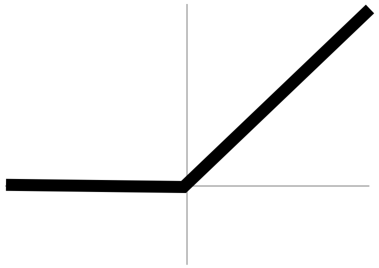
Regularization

- Can use L_1 and L_2 regularization as before
  - As before, do not regularize any of the intercept terms!
  - L_2 regularization more common

- Applied to "unrolled" weight matrices
  - Equivalently, Frobenius norm $\left\|W_j\right\|_F^2 = \sum_{i=1}^{k} \sum_{i'=1}^{h} W_{i,i'}^2$

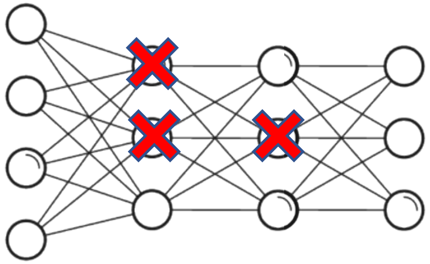
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

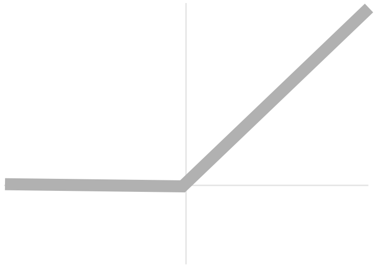


Dropout

Managing Training

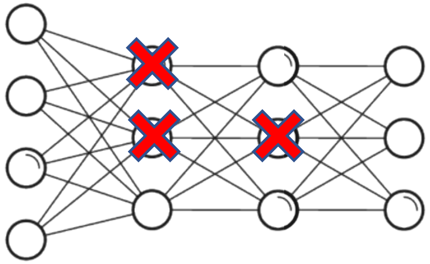
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



Dropout

Managing Training

Dropout

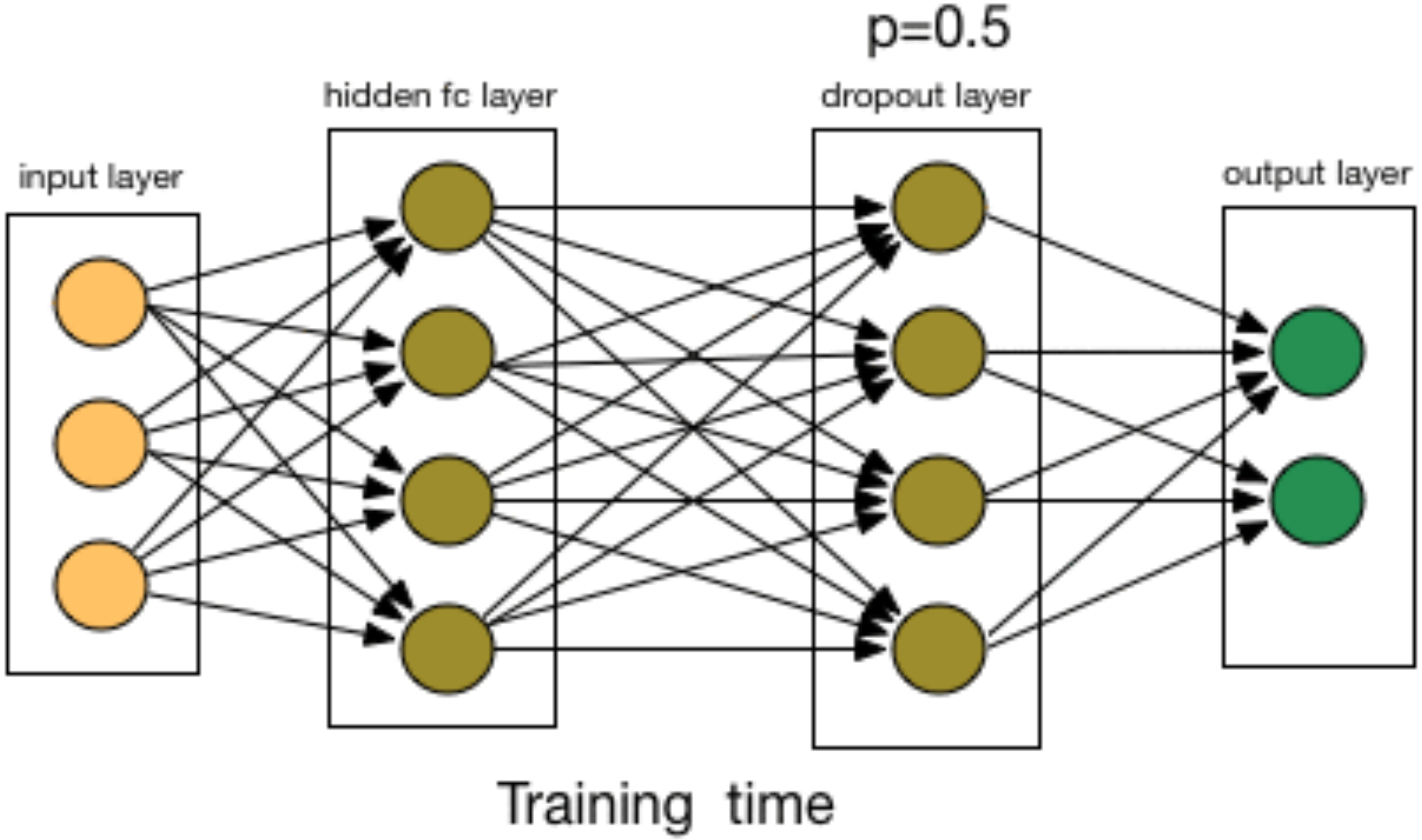
- **Idea:** During training, randomly "drop" (i.e., zero out) a fraction p of the neurons $z_i^{(j)}$ (usually take $p = \frac{1}{2}$)

- Implemented as its own layer

$$\mathrm{Dropout}(z) = \begin{cases} z & \text{with prob. } p \\ 0 & \text{otherwise} \end{cases}$$

- Usually include it at a few layers just before the output layer

Dropout



p=0.5
hidden fc layer
dropout layer
input layer
output layer
Training time

Intuition: Dropout as regularization

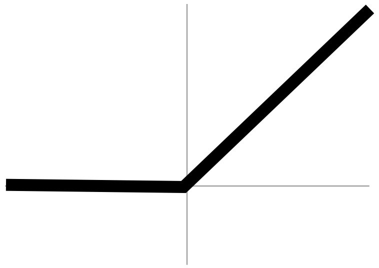
- Encourages robustness to missing information from the previous layer
- Each neuron works with many different kinds of inputs
- Makes them more likely to be individually competent

Dropout at Test Time

- **Naïve strategy:** Stop dropping neurons
  - **Problem:** Not the distribution the layer was trained on
- **Naïve strategy:** Average across all possible predictions
  - **Problem:** There are $2^{\#\text{neurons}}$ possible realizations of the randomness
- **Solution:** Turn off dropout but multiply the outgoing weights by p
  - Good approximation of the geometric mean of all $2^{\#\text{neurons}}$ predictions
- **Note:** Can also leave dropout on, sample multiple realizations of the randomness, and report distribution to help quantify uncertainty

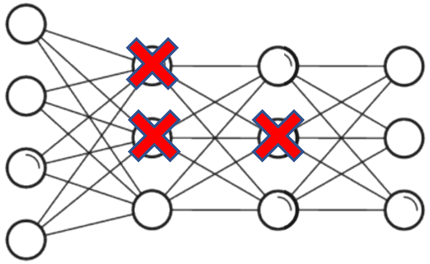
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

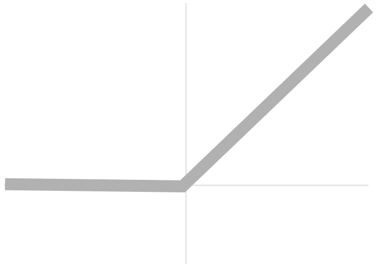


Dropout

Managing Training

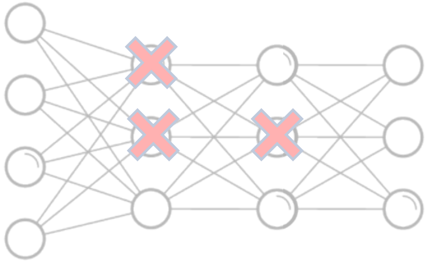
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

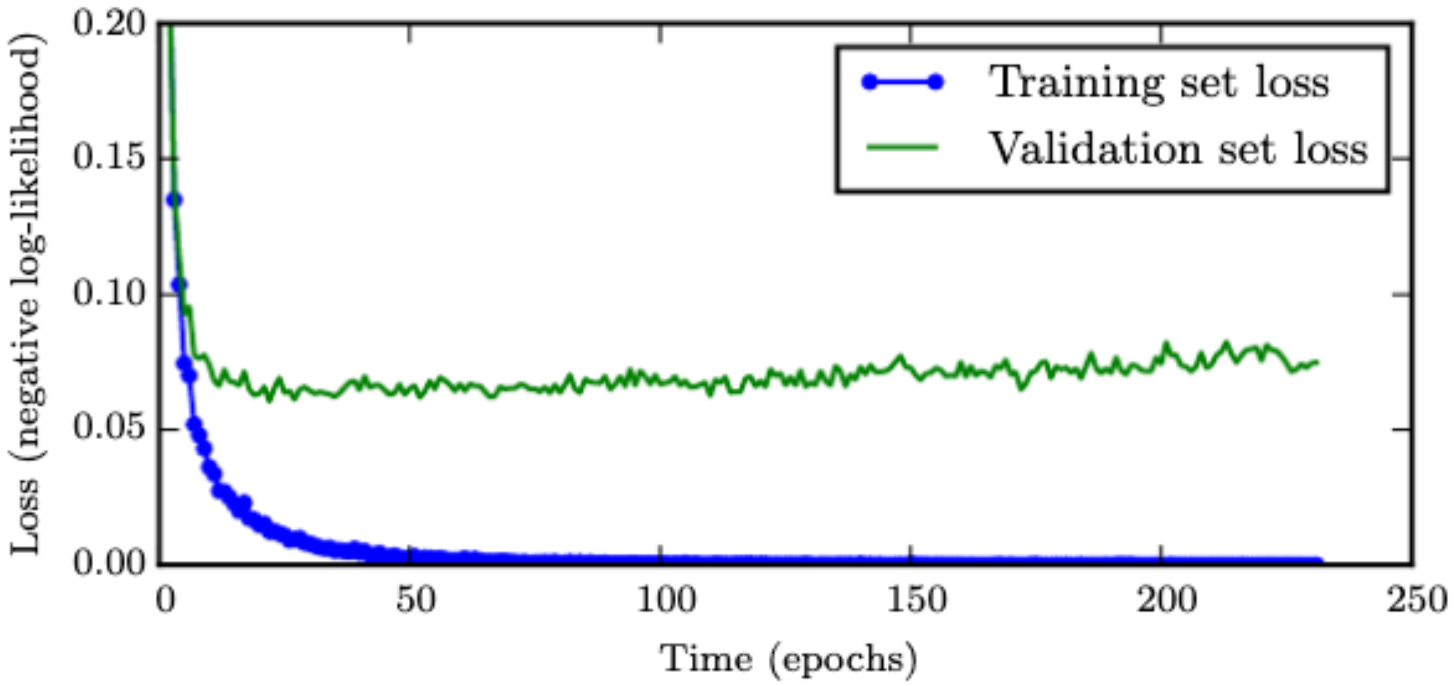


Dropout

Managing Training

Early Stopping

- Stop when your validation loss starts increasing (alternatively, finish training and choose best model on validation set)
  - Simple way to introduce regularization



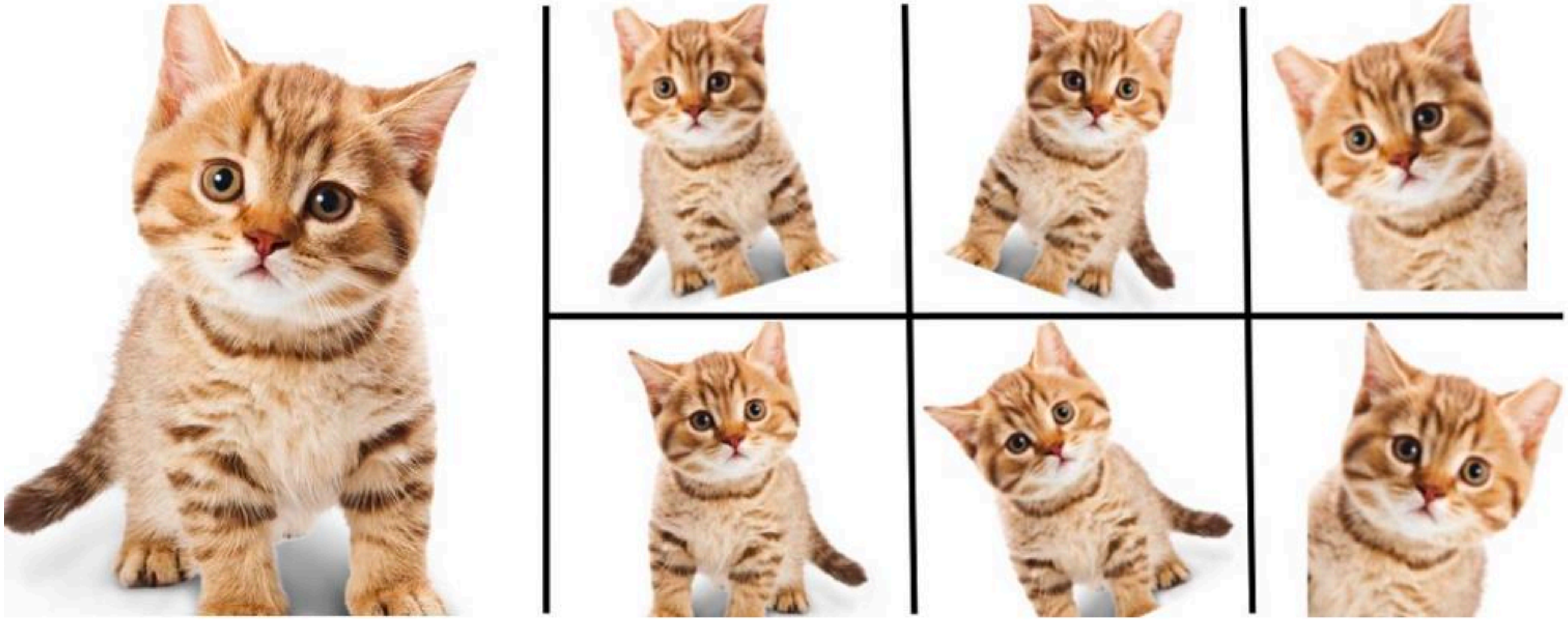
Data Augmentation

- **Data augmentation:** Generate more data by modifying training inputs

- Often used when you know that your output is robust to some transformations of your data
  - **Image domain:** Color shifts, add noise, rotations, translations, flips, crops
  - **NLP domain:** Substitute synonyms, generate examples (doesn’t work as well but ongoing research direction)
  - Can combine primitive shifts

- **Note:** Labels are simply the label of original image

Data Augmentation



Agenda

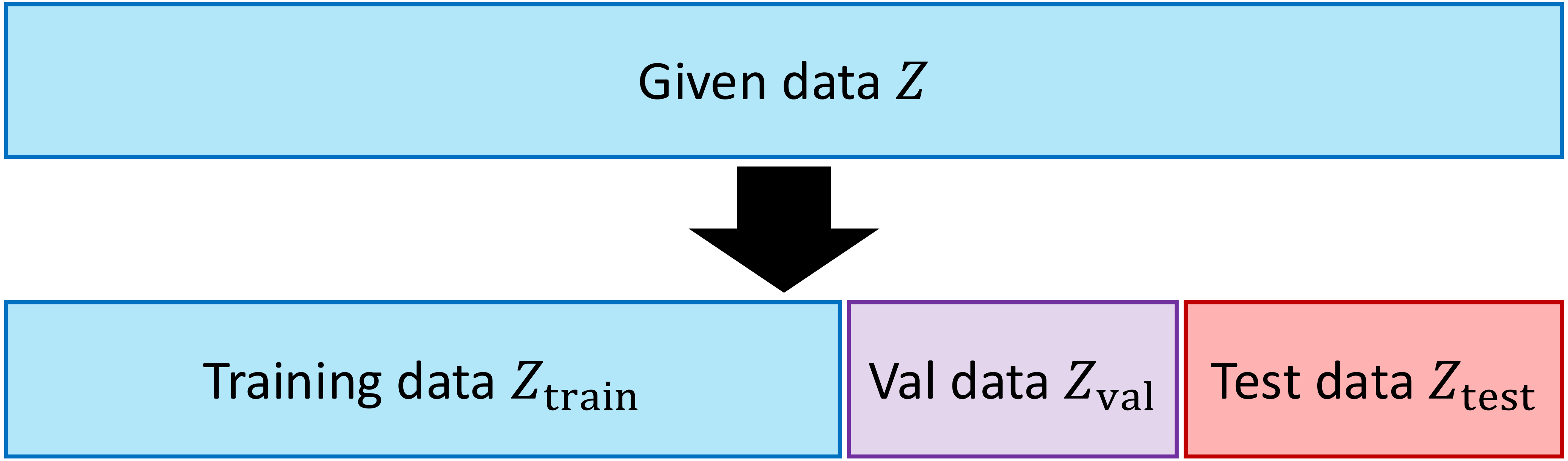
- Recap
- Neural network tips and tricks
- Hyperparameter tuning
- Implementation

(Default) Hyperparameteter Choices

- **Architecture:** Stick close to tried-and-tested architectures (esp. for images)
- **SGD variant:** Adam, second choice SGD + 0.9 momentum
- **Learning rate:** 3e-4 (Adam), 1e-4 (for SGD + momentum)
- **Learning rate schedule**: Divide by 10 every time training loss stagnates
- **Weight initialization**: “Kaiming” initialization (scaled Gaussian)
- **Activation functions**: ReLU
- **Regularization**: BatchNorm (& cousins), L2 regularization + Dropout on some or all fully connected layers
- **Hyperparameter Optimization**: Random sampling (often uniform on log scale), coarse to fine

Hyperparameter Optimization

- **Recall:** Use cross-validation to tune hyperparameters!
  - Typically use one held-out validation set for computational tractability
  - E.g., 60/20/20 split
  - Can use smaller validation/test sets if you have a very large dataset



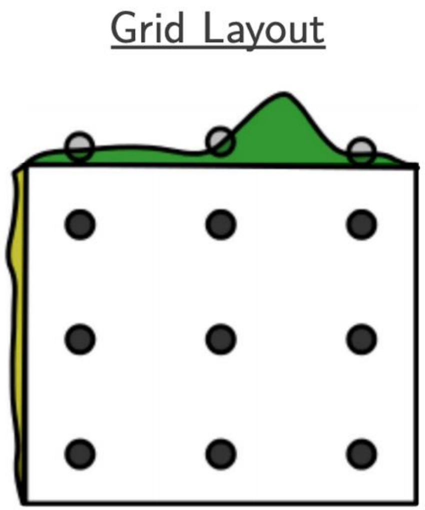
Hyperparameter Optimization Tips

- Keep the number of hyperparameters as small as possible
  - **Most important:** Learning rate, batch size

- **Strategy:** Automatically search over grid of hyperparameters and choose the best one on the validation set
  - Easy to parallelize across many machines
  - Record hyperparameters of all runs carefully!
  - Use the same random seeds for all runs

Hyperparameter Optimization Tips

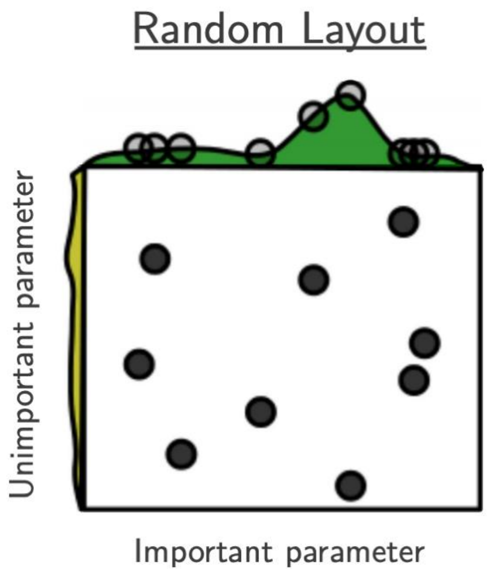
- **What about multiple hyperparameters?**
  - For 2 or 3 hyperparameters, do a systematic “grid search”



[Bergstra & Bengio, JMLR 2012]

Hyperparameter Optimization Tips

- **What about multiple hyperparameters?**
  - For >3 hyperparameters, do random search

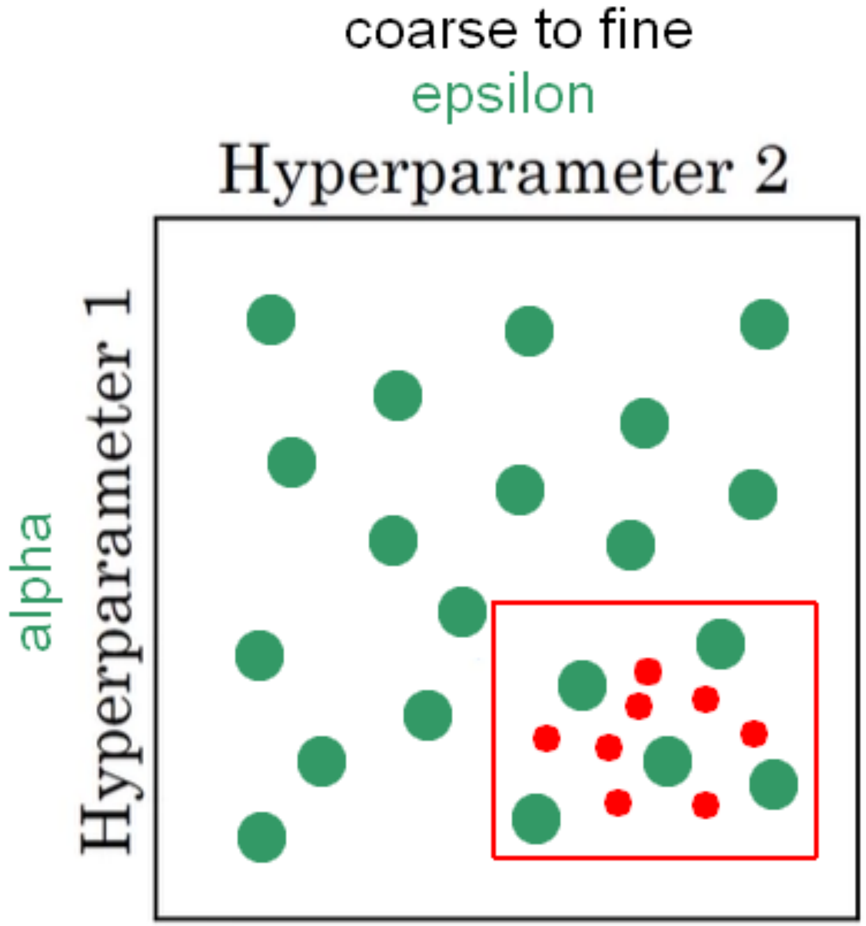


[Bergstra & Bengio, JMLR 2012]

Hyperparameter Optimization Tips

- **Coarse-to-find search**
    - Iteratively search over a window of hyperparameters
    - If the best results are near the boundary, center it on best hyperparameters
    - Otherwise, set a smaller window centered on the best hyperparameters

- **Bayesian optimization:** ML-guided search across hyperparameter trials to find good choices



https://www.andreaperlato.com/aipost/hyperparameters-tuning-in-ai/

Practical tips for training neural nets

- See Andrej Karpathy's blog post: http://karpathy.github.io/2019/04/25/recipe/
    - Fix random seed during debugging
    - Overfit a tiny dataset first
    - With everything (architecture, learning algorithm, data etc.), start simple and build complexity slowly over iterations.
    - Plot weight and gradient magnitudes to detect vanishing/exploding gradients.

- Assigned reading: Chapter 11 of the Deep Learning textbook: "Practical Methodology" https://www.deeplearningbook.org/contents/guidelines.html

Agenda

- Recap
- Neural network tips and tricks
- Hyperparameter tuning
- Implementation

Pytorch

- Open source packages have helped democratize deep learning

Pytorch: Defining a network “architecture”

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms


class Net(nn.Module):
    def __init__(self, in_features=10, num_classes=2, hidden_features=20):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.fc2 = nn.Linear(hidden_features, num_classes)

    def forward(self, x):
        x1 = self.fc1(x)
        x2 = F.relu(x1)
        x3 = self.fc2(x2)
        log_prob = F.log_softmax(x3, dim=1)

        return log_prob
```

Common parent class: nn.Module

Constructor: Defining layers of the network

Forward propagation: Defining f(x) through the layers

What about backward propagation?

Autograd

Good news: Chain rule based gradient computation is implemented in pytorch naturally! (True for all the important libraries today, including Tensorflow, Jax). No need to implement `backward()` !

`loss.backward()` simply backtracks through the computational graph, applying the chain rule, computing gradients with respect to all `tensor`s involved.

Does not apply any gradient descent updates yet.

Pytorch: Training function

```python
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

Looping over mini-batches

Flush out all old gradients

Runs forward pass model.forward(data)

Loss computation

Full gradient computation

Update all parameters