Announcements

- Project Milestone 1 due **Tonight at 8pm**
- Quiz due **tomorrow (Thursday, March 2) at 8pm**
- HW 4 due **Wednesday, March 15**
  - Please start early!

Lecture 14: Neural Networks (Part 2)

CIS 4190/5190

Spring 2023

Agenda

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

Recap: Neural Network Model Family

- Each **layer** is a parametric function $f_{W_j}: \mathbb{R}^k \to \mathbb{R}^h$ for some k, h
- Compose sequentially to form model family (a.k.a. **architecture**):

$$f_W = f_{W_m} \circ \cdots \circ f_{W_1}$$

- **Examples:**
  - **Linear:** $f_W(z) = Wz$
  - **Activation function:** $g(z) = \sigma(z)$
  - **Softmax:** $f(z) = \mathrm{softmax}(z)$

Recap: Optimization & Backpropagation

- Based on gradient descent, with a few tweaks
  - **Note:** Loss is nonconvex, but gradient descent works well in practice
- **Key challenge:** How to compute the gradient?
  - **Previous strategy:** Work out gradient for every model family
  - **Backpropagation:** Algorithm for computing gradient of an arbitrary programmatic composition of layers

Recap: Backpropagation by Example

- Consider a function $f(x, W, \beta) = f_2(f_1(x, W), \beta)$, where
  - $f_1(z, W) = g(Wz)$
  - $f_2(z, \beta) = \beta^\top z$

- Its derivatives are

$$
\begin{aligned}
D_\beta f(x, W, \beta) &= D_\beta f_2(f_1(x, W), \beta) \\
&= \partial_z f_2(f_1(x, W), \beta) D_\beta f_1(x, W) + \partial_\beta f_2(f_1(x, W), \beta) \\
&= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \partial_\beta f_2(f_1(x, W), \beta)
\end{aligned}
$$

Recap: Backpropagation by Example

- Consider a function $f(x, W, \beta) = f_2(f_1(x, W), \beta)$, where
  - $f_1(z, W) = g(Wz)$
  - $f_2(z, \beta) = \beta^\top z$

- Its derivatives are

$$
\begin{aligned}
D_W f(x, W, \beta) &= D_W f_2(f_1(x, W), \beta) \\
&= \partial_z f_2(f_1(x, W), \beta) D_W f_1(x, W) + \partial_W f_2(f_1(x, W), \beta) \\
&= \partial_z f_2(f_1(x, W), \beta) \partial_W f_1(x, W)
\end{aligned}
$$

Recap: Backpropagation

- **General case:** Consider a neural network

$$f_W(x) = f_{W_m} \circ f_{W_{m-1}} \circ \cdots \circ f_{W_1}(x)$$

- **Forward pass:**

$$z^{(j)} = f_{W_j} \circ \cdots \circ f_{W_1}(x)$$

- **Backward pass:**

$$D_{W_j} f_W(x) = \underbrace{\partial_z f_{W_m}\left(z^{(m-1)}\right) \ldots \partial_z f_{W_{j+1}}\left(z^{(j)}\right)}_{\text{shared across terms}} \partial_{W_j} f_{W_j}\left(z^{(j-1)}\right)$$

Recap: Backpropagation

$$\partial_z f_{W_m}(z) \partial_z$$

$$= \begin{bmatrix} \frac{\partial f_{W_{m,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,1}}}{\partial z_k}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m,h}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,h}}}{\partial z_k}(z) \end{bmatrix}$$

Recap: Backpropagation

$$\partial_z f_{W_m}(z)\partial_z f_{W_{m-1}}(z)$$

$$= \begin{bmatrix} \frac{\partial f_{W_{m,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,1}}}{\partial z_k}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m,h}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,h}}}{\partial z_k}(z) \end{bmatrix} \begin{bmatrix} \frac{\partial f_{W_{m-1,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,1}}}{\partial z_\ell}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m-1,k}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,k}}}{\partial z_\ell}(z) \end{bmatrix}$$

Recap: Backpropagation

$$\partial_z f_{W_m}(z)\partial_z f_{W_{m-1}}(z)\partial_z f_{W_{m-2}}(z)$$

$$= \begin{bmatrix} \frac{\partial f_{W_{m,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,1}}}{\partial z_k}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m,h}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,h}}}{\partial z_k}(z) \end{bmatrix} \begin{bmatrix} \frac{\partial f_{W_{m-1,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,1}}}{\partial z_\ell}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m-1,k}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,k}}}{\partial z_\ell}(z) \end{bmatrix} \begin{bmatrix} \frac{\partial f_{W_{m-1,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,1}}}{\partial z_m}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m-1,\ell}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,\ell}}}{\partial z_m}(z) \end{bmatrix}$$

Recap: Backpropagation

$$\partial_z f_{W_m}(z)\partial_z f_{W_{m-1}}(z)\partial_z f_{W_{m-2}}(z)\dots$$

$$= \begin{bmatrix} \frac{\partial f_{W_{m,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,1}}}{\partial z_k}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m,h}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m,h}}}{\partial z_k}(z) \end{bmatrix} \begin{bmatrix} \frac{\partial f_{W_{m-1,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,1}}}{\partial z_\ell}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m-1,k}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,k}}}{\partial z_\ell}(z) \end{bmatrix} \begin{bmatrix} \frac{\partial f_{W_{m-1,1}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,1}}}{\partial z_m}(z) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{W_{m-1,\ell}}}{\partial z_1}(z) & \cdots & \frac{\partial f_{W_{m-1,\ell}}}{\partial z_m}(z) \end{bmatrix} \dots$$

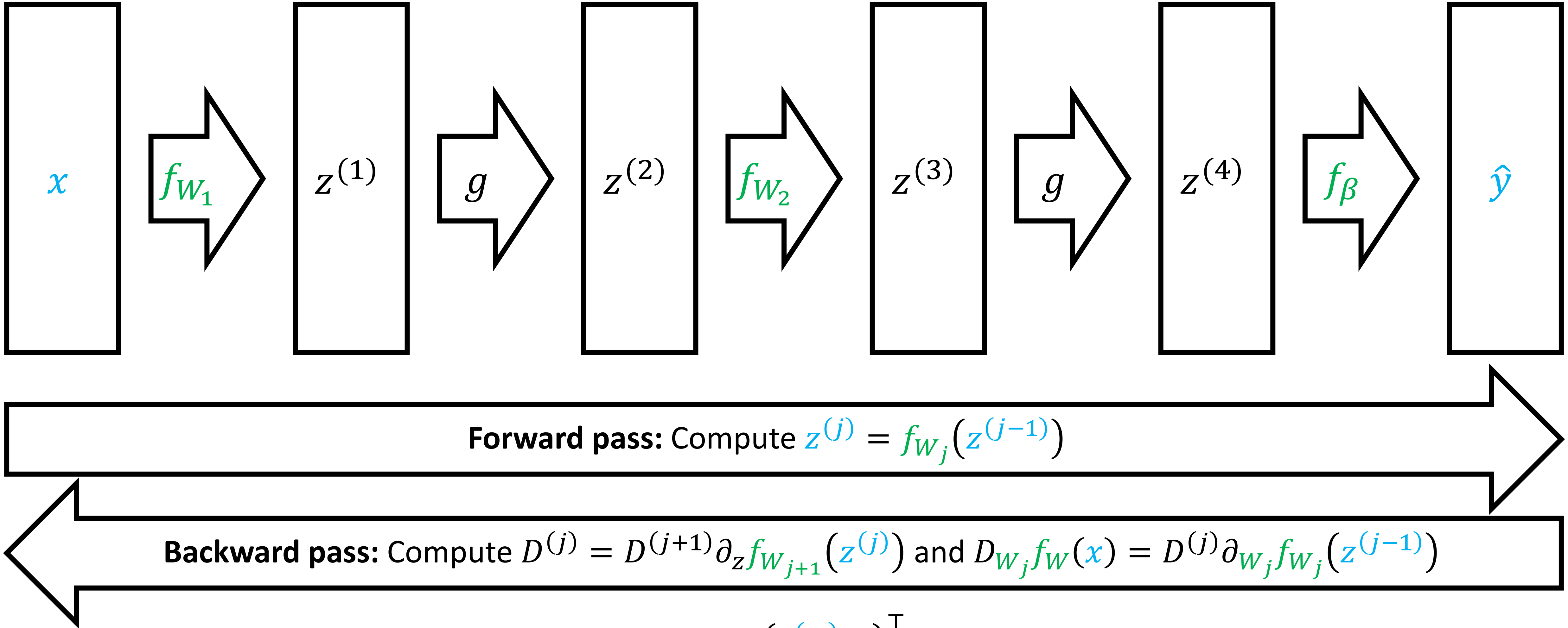
Recap: Backpropagation

- **Forward pass:** Compute forwards from $j = 0$ to $j = m$
  - $z^{(j)} = \begin{cases} x & \text{if } j = 0 \\ f_{W_j}\left(z^{(j-1)}\right) & \text{if } j > 0 \end{cases}$

- **Backward pass:** Compute backwards from $j = m$ to $j = 1$
  - $D^{(j)} = \begin{cases} 1 & \text{if } j = m \\ D^{(j+1)} \partial_z f_{W_{j+1}}\left(z^{(j)}\right) & \text{if } j < m \end{cases}$
  - $D_{W_j} f_W(x) = D^{(j)} \partial_{W_j} f_{W_j}\left(z^{(j-1)}\right)$

- **Final output:** $\nabla_{W_j} L(f_W(x), y)^\top = \nabla_{\hat{y}} L\left(z^{(m)}, y\right)^\top D_{W_j} f_W(x)$ for each j

Recap: Backpropagation



Final output: $\nabla_{\hat{y}} L\left(z^{(m)}, y\right)^\top D_{W_j} f_W(x)$

Gradient Descent

- $W_1 \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\dots\}$ **until** convergence:

$$W_{t+1,j} \leftarrow W_{t,j} - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_{W_j} L\big(f_{W_t}(x_i), y_i\big) \quad (\text{for each } j)$$

- **return** f_{W_t}

Gradient Descent

- $W_1 \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\dots\}$ **until** convergence:
  - Compute gradients $\nabla_{W_j} L\big(f_{W_t}(x_i), y_i\big)$ using backpropagation
  - Update parameters:

$$W_{t+1,j} \leftarrow W_{t,j} - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_{W_j} L\big(f_{W_t}(x_i), y_i\big) \quad \text{(for each } j\text{)}$$

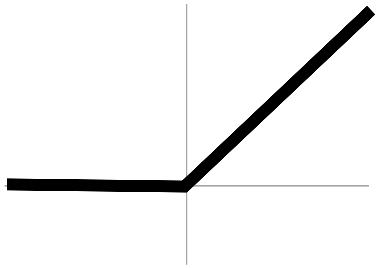
- **return** f_{W_t}

Agenda

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

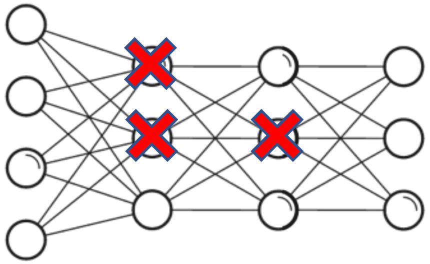
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

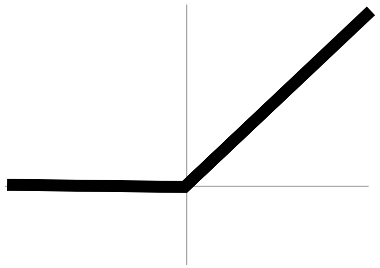


Dropout

Managing Training

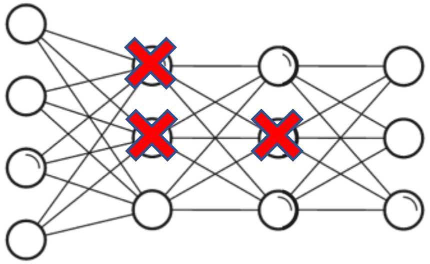
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



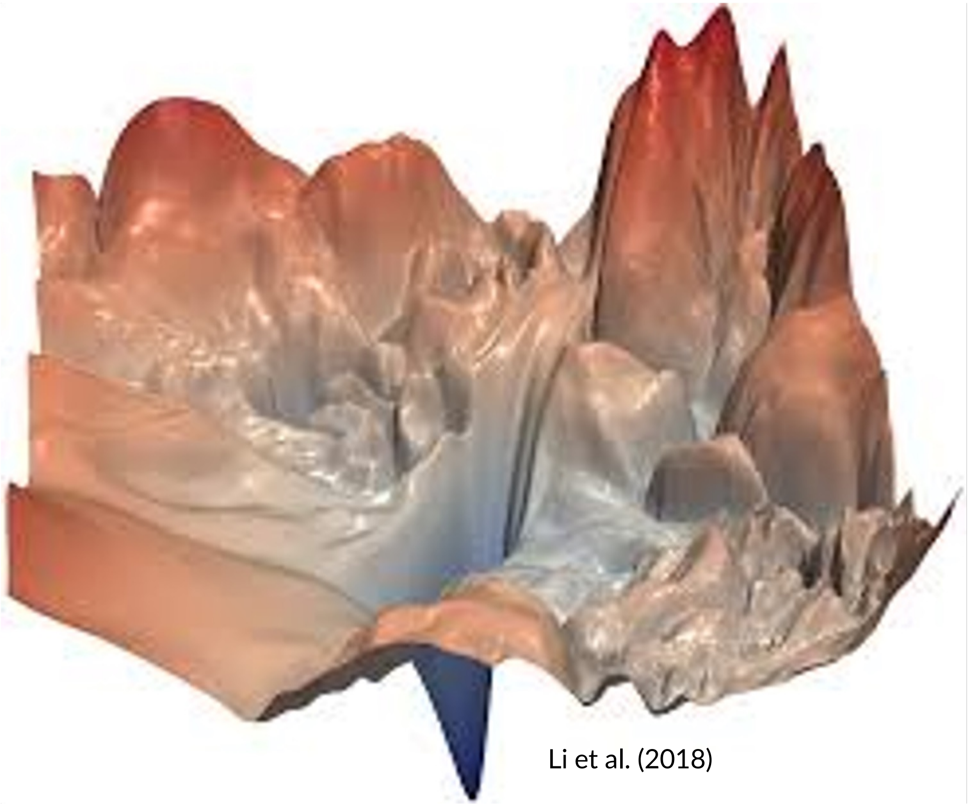
Dropout

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)

Gradient Descent

- $W \leftarrow \text{Initialize}(\ )$
- **for** $t \in \{1,2,\ldots,T\}$:

$$\beta \leftarrow \beta - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_\beta L\big(f_\beta(x_i), y_i\big)$$

- **return** f_β

Gradient Descent

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\ldots,T\}$:

$$\beta \leftarrow \beta - \frac{\alpha}{n} \cdot \sum_{i=1}^{n} \nabla_\beta L\big(f_\beta(x_i), y_i\big)$$

- **return** f_β

Stochastic Gradient Descent

usually $T \in \{1, \dots, 10\}$

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2, \dots, T\}$:
    - **for** $i \in \{1,2, \dots, n\}$:

$$\beta \leftarrow \beta - \alpha \cdot \nabla_\beta L\big(f_\beta(x_i), y_i\big)$$

- **return** f_β

Minibatch Stochastic Gradient Descent

- $W \leftarrow \text{Initialize}(\,)$
- **for** $t \in \{1,2,\ldots,T\}$:
  - **for** $i' \in \left\{1,2,\ldots,\frac{n}{k}\right\}$:

$$\beta \leftarrow \beta - \frac{\alpha}{k} \cdot \sum_{i=i'k}^{i'(k+1)-1} \nabla_\beta L\big(f_\beta(x_i), y_i\big) \quad (\text{for each } j)$$

- **return** f_β

Accelerated Gradient Descent

- Vanilla gradient descent:

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

- Accelerated gradient descent:

$$\begin{aligned} &\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\beta L\big(f_\beta(x), y\big) \\ &\beta \leftarrow \beta + \rho \end{aligned}$$

Accelerated Gradient Descent

- Vanilla gradient descent:

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

- Accelerated gradient descent:

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

Accelerated Gradient Descent

- Vanilla gradient descent:

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

- Accelerated gradient descent:

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

Accelerated Gradient Descent

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

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

Nesterov Momentum

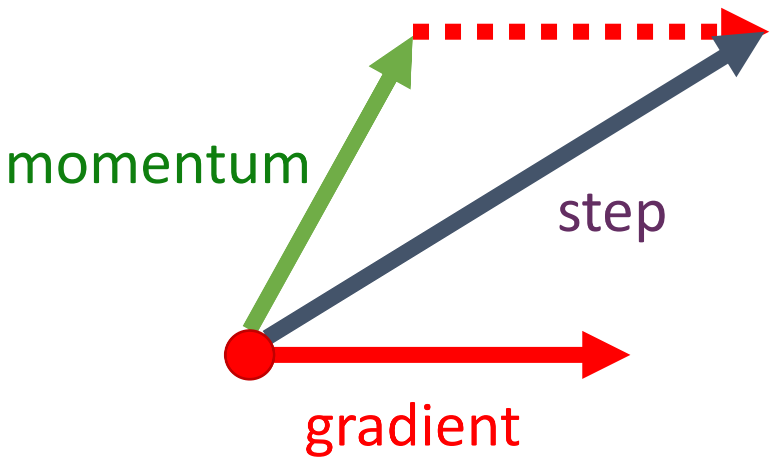
- Accelerated gradient descent:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\beta L\big(f_{\beta}(x), y\big)$$
$$\beta \leftarrow \beta + \rho$$

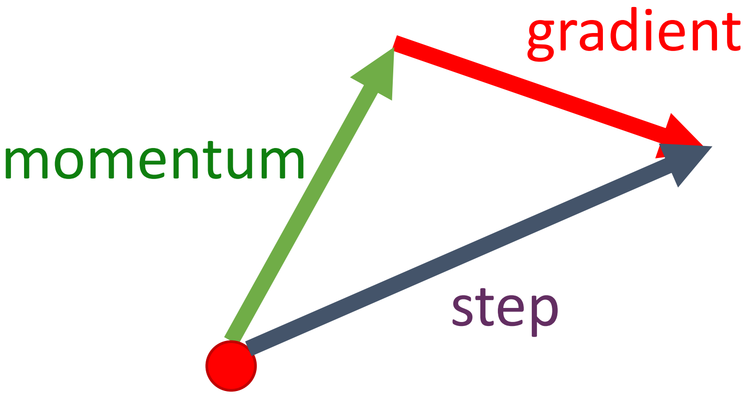
- Nesterov momentum:

$$\rho \leftarrow \mu \cdot \rho - \alpha \cdot \nabla_\beta L\big(f_{\beta+\mu\cdot\rho}(x), y\big)$$
$$\beta \leftarrow \beta + \rho$$

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

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

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

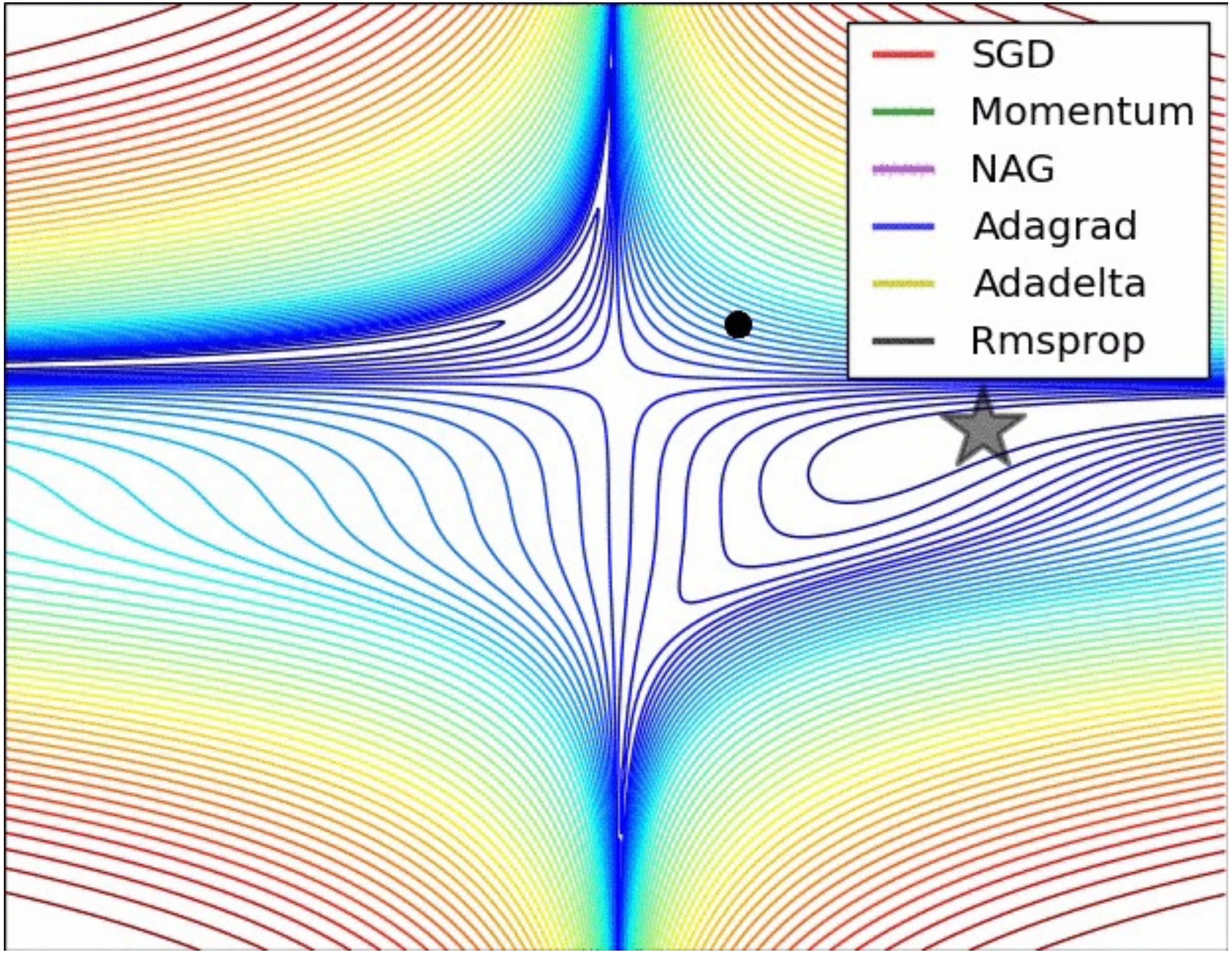
- **Adam:** Similar to RMSprop, but with both the first and second moments of the gradients

$$G \leftarrow \lambda \cdot G + (1 - \lambda) \cdot g^2$$

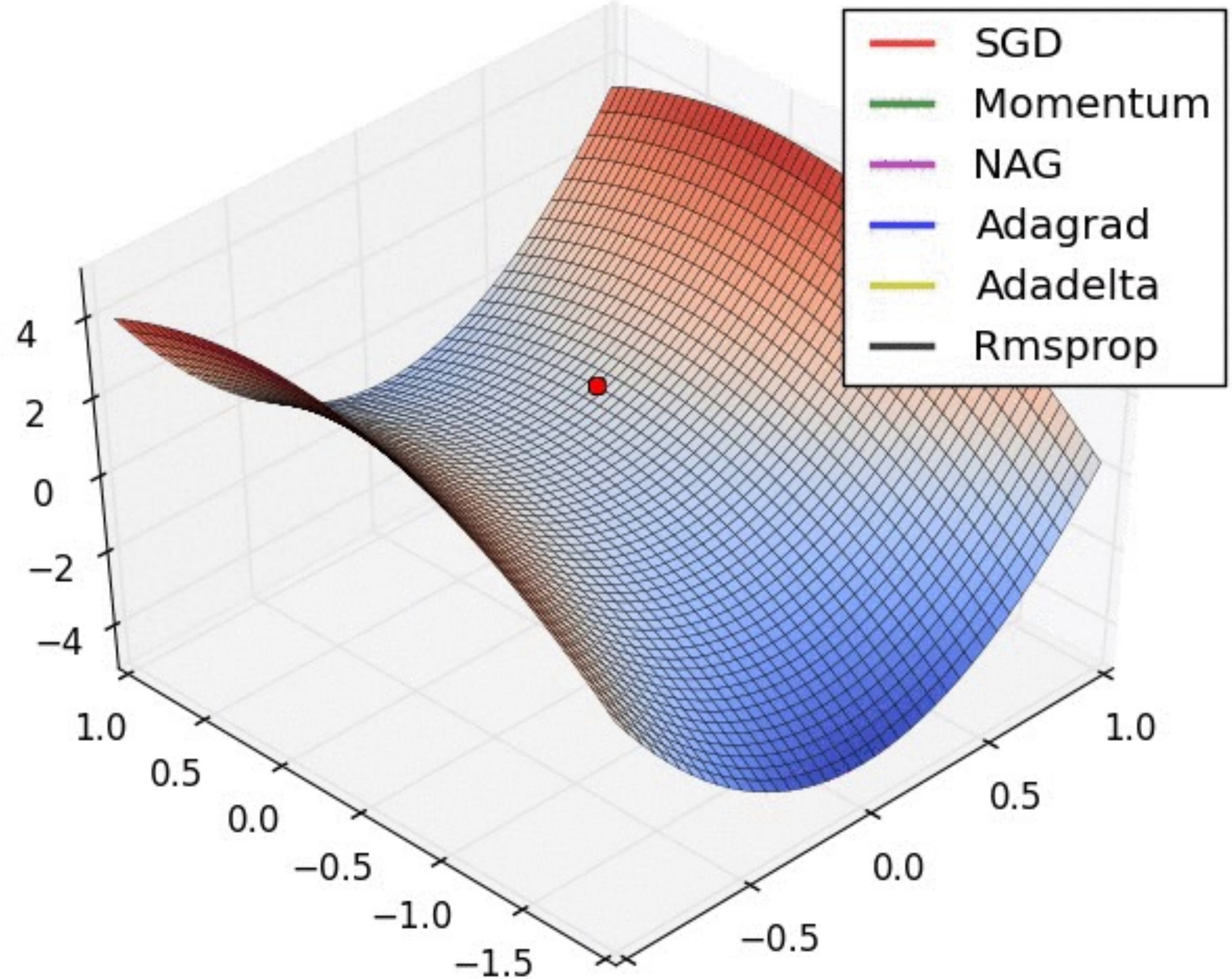
$$g' \leftarrow \lambda' \cdot g' + (1 - \lambda') \cdot g$$

$$\beta \leftarrow \beta - \frac{g'}{\sqrt{G}}$$

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



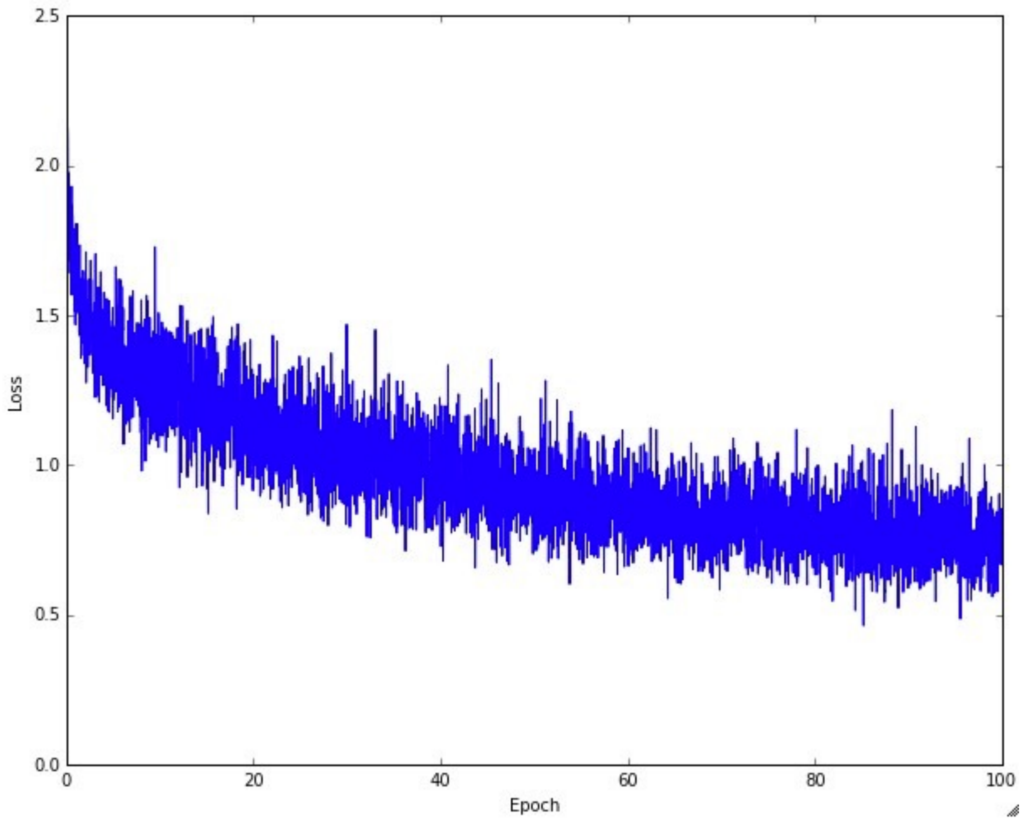
http://cs231n.github.io/neural-networks-3 (Alec Radford)

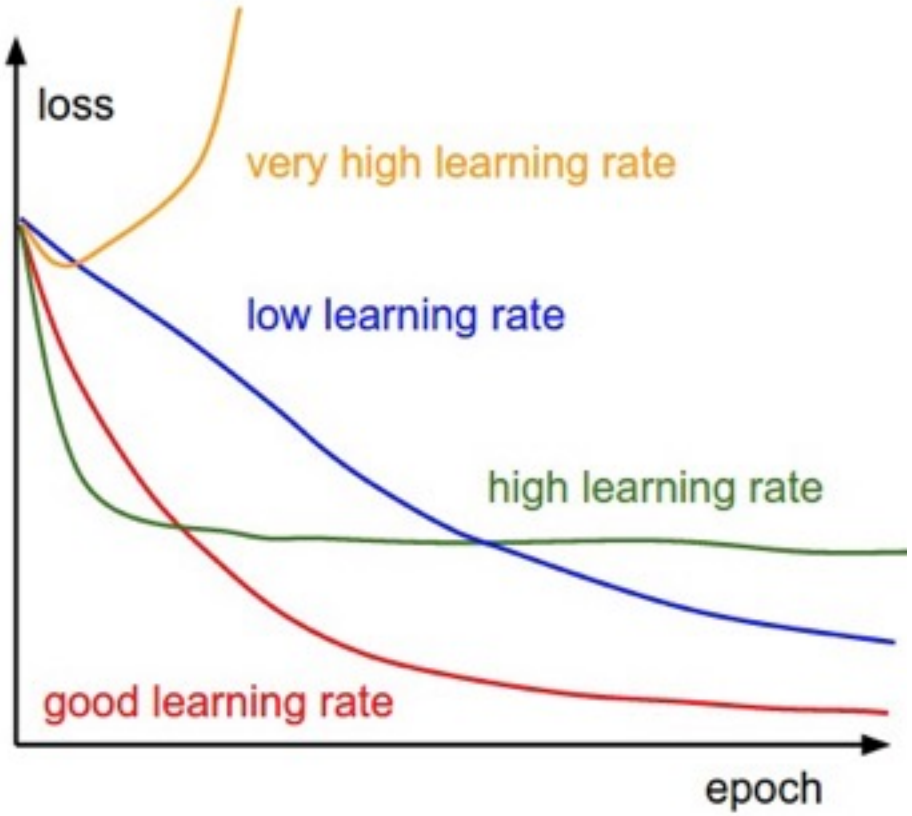


http://cs231n.github.io/neural-networks-3 (Alec Radford)

Learning Rate

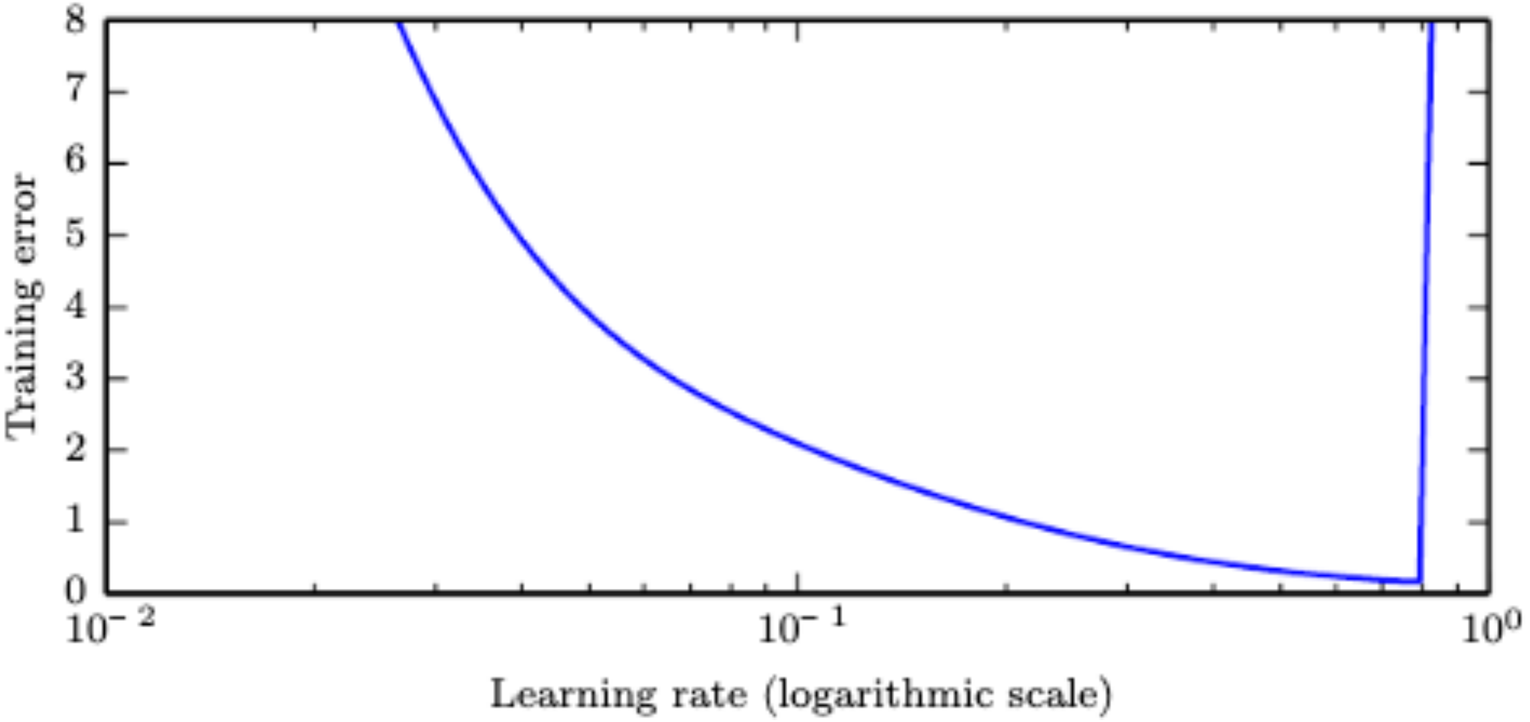
- Most important hyperparameter; tune by looking at training loss





Learning Rate

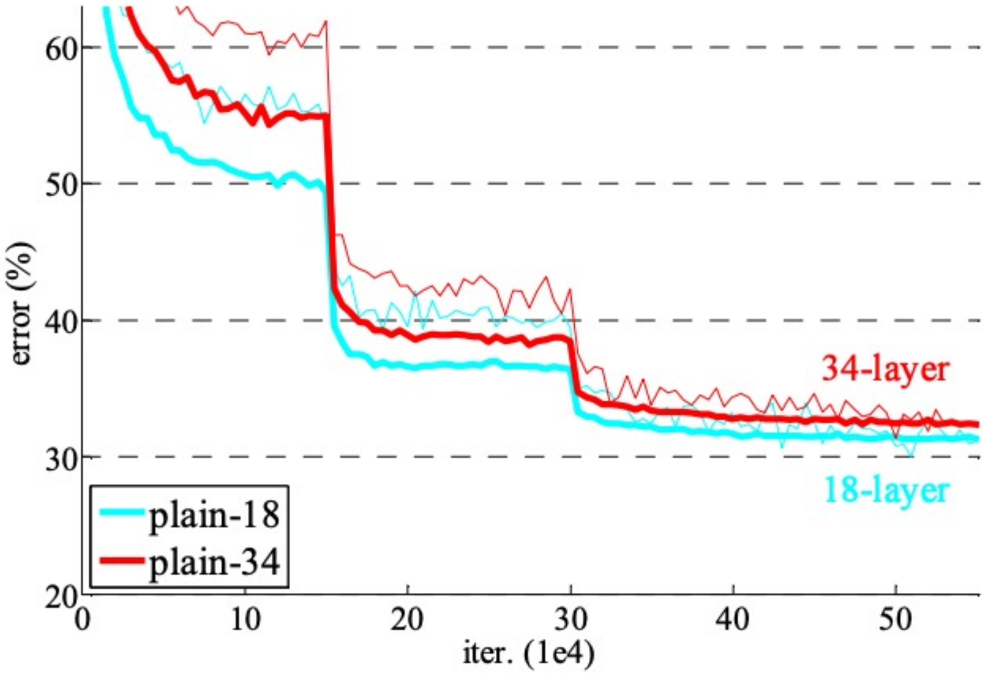
- Learning rate vs. training error:



Goodfellow et al, Deep Learning Book, 2019

Learning Rate

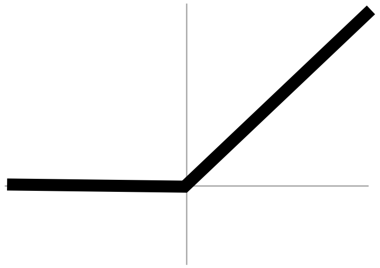
- **Schedules:** Reducing the learning rate every time the validation loss stagnates can be very effective for training



He et al, Residual Networks, 2015

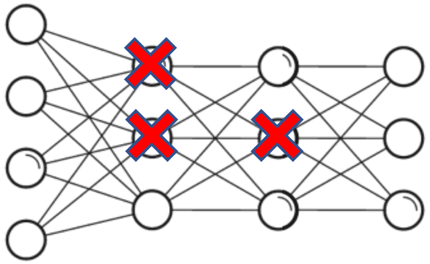
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

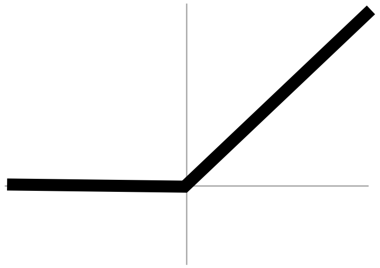


Dropout

Managing Training

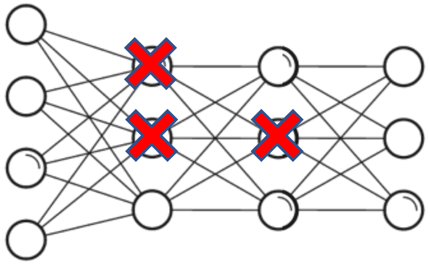
Neural Network Tips & Tricks

Optimization



Activation Functions

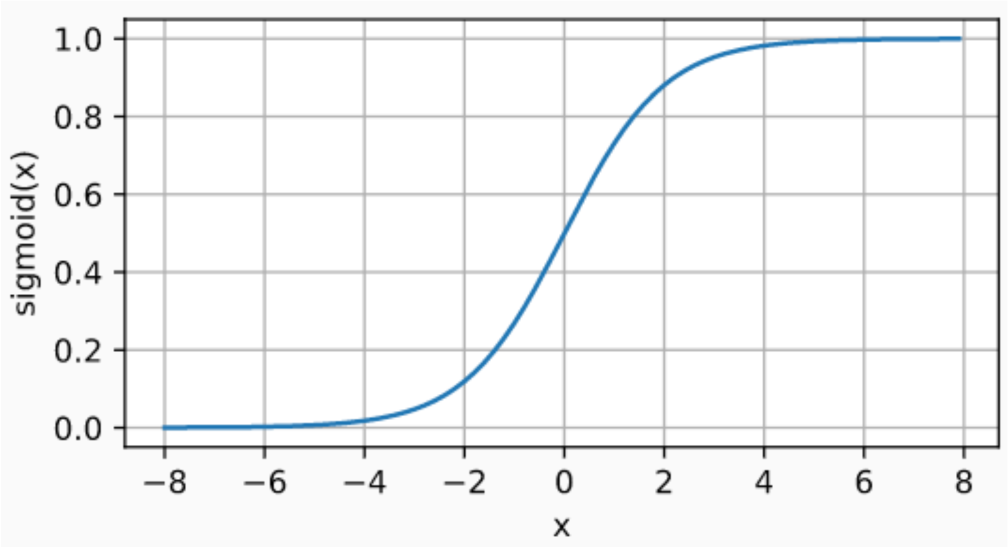
Managing Weights



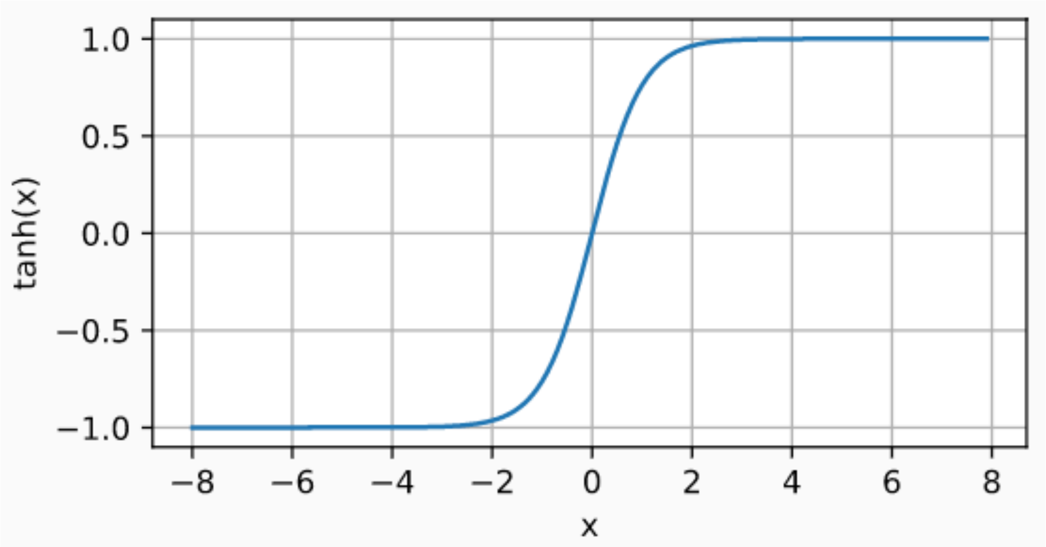
Dropout

Managing Training

Historical Activation Functions



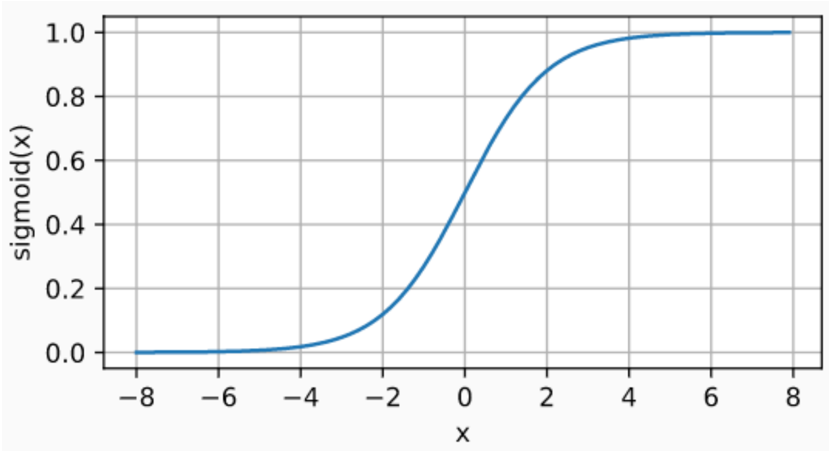
sigmoid



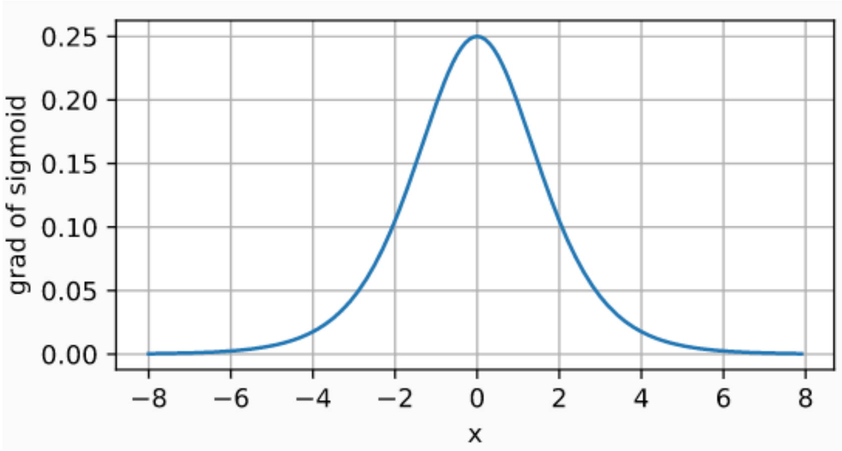
tanh

Vanishing Gradient Problem

- The gradient of the sigmoid function is often nearly zero
- **Recall:** In backpropagation, gradients are products of $\partial_z g\left(z^{(j)}\right)$
- **Quickly multiply to zero!**
  - Early layers update very slowly



sigmoid



sigmoid gradient

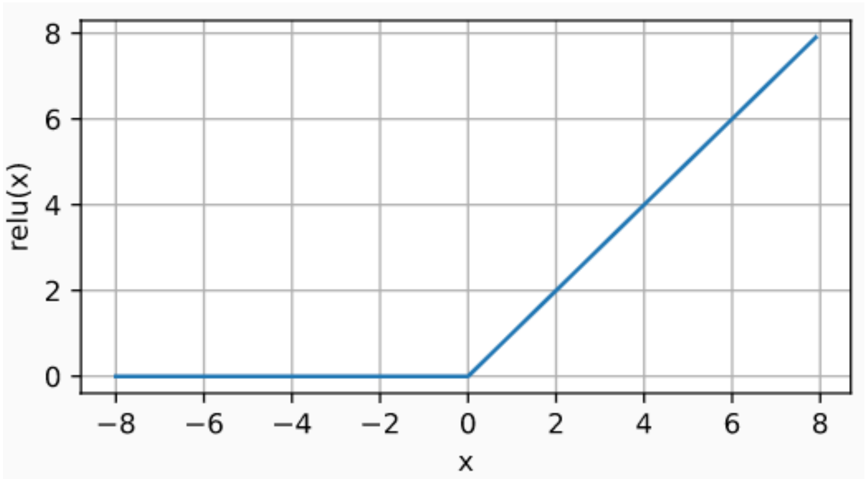
ReLU Activation

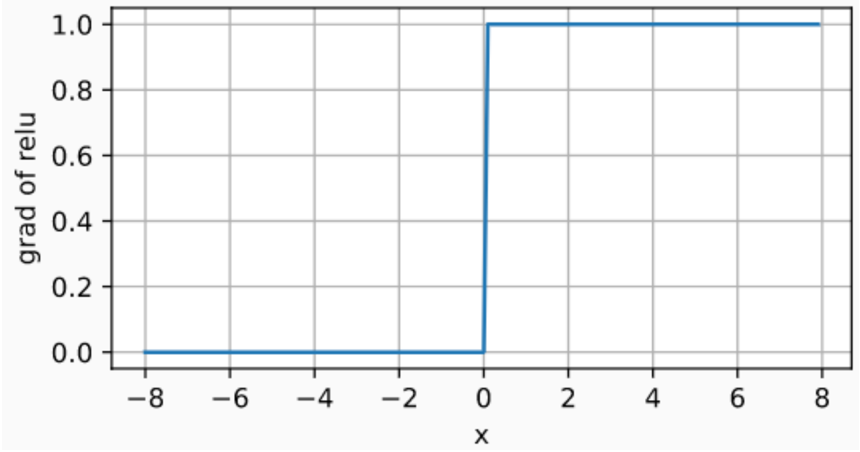
- Activation function

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

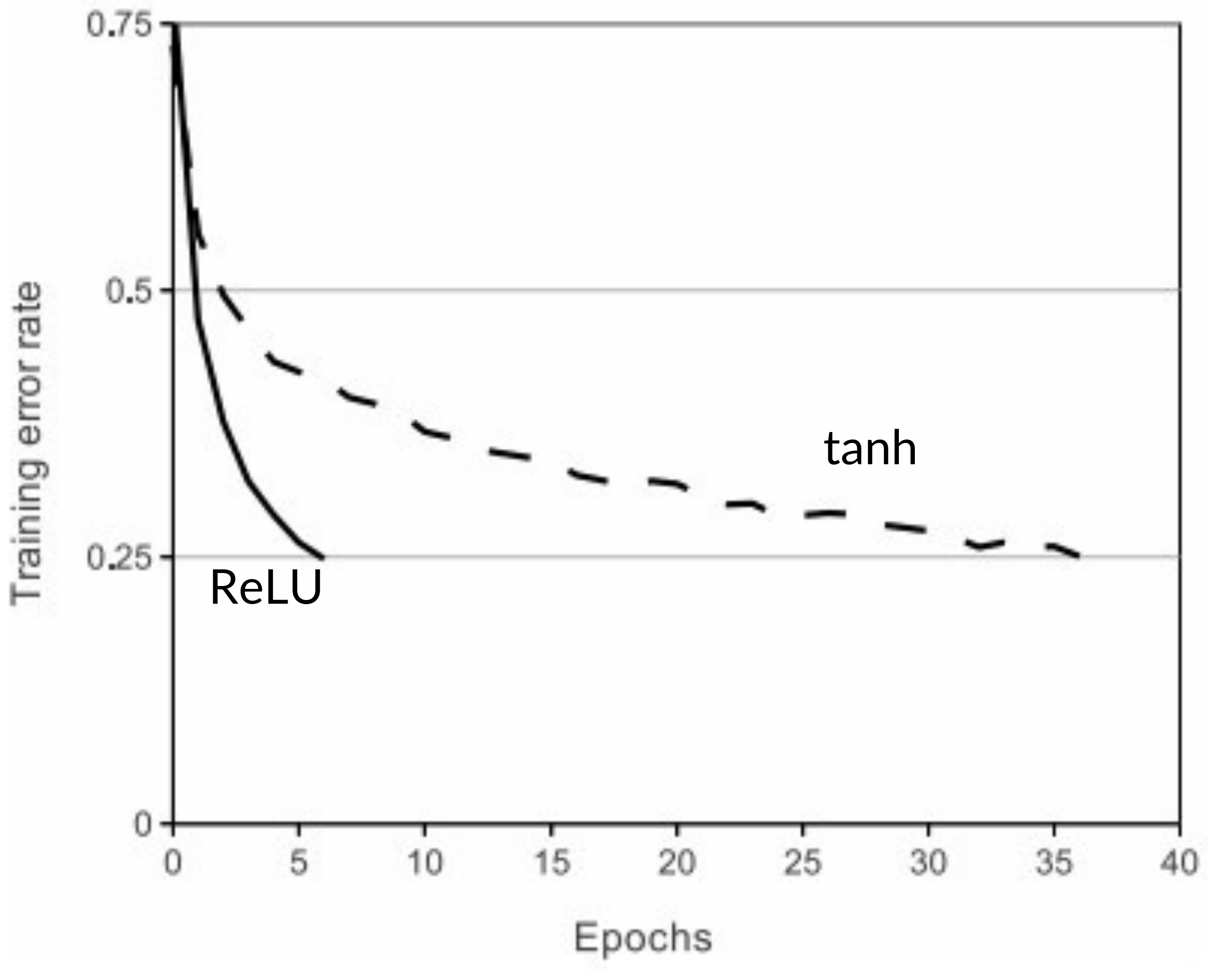
- Gradient now positive on the entire region $z \geq 0$

- Significant performance gains for deep neural networks

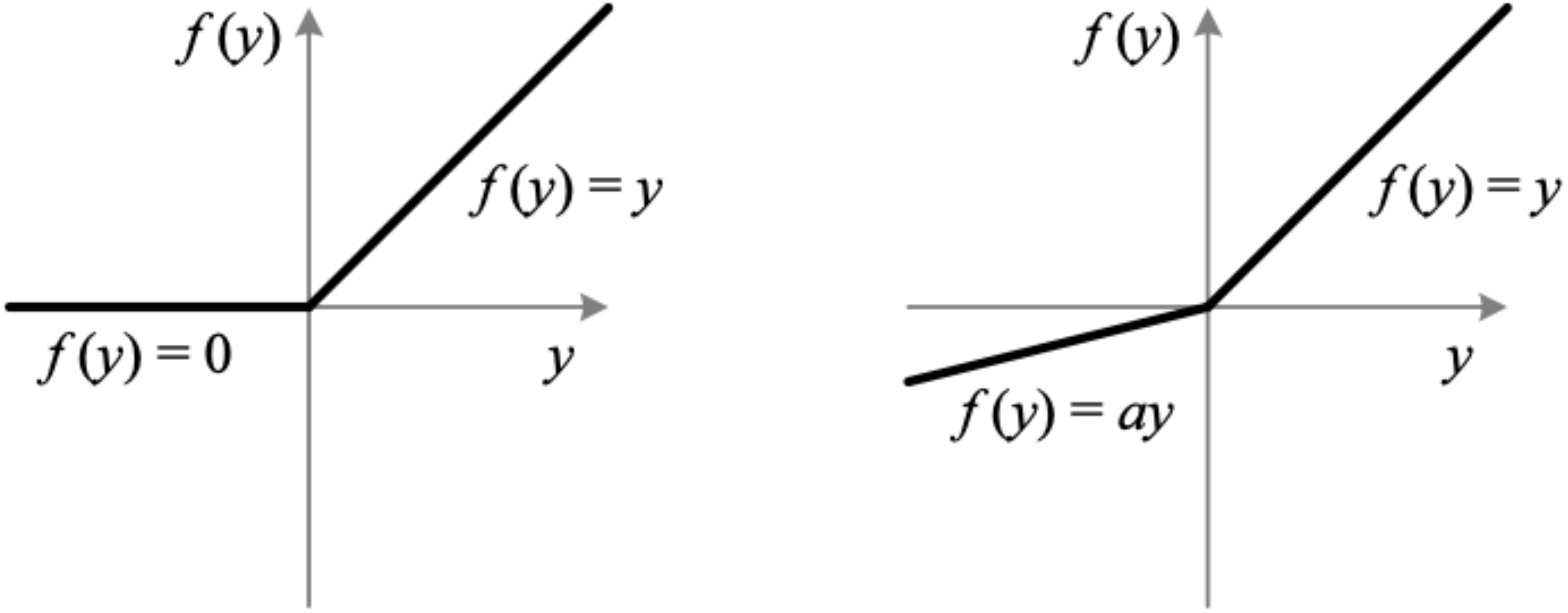




ReLU Activation



PRReLU Activation

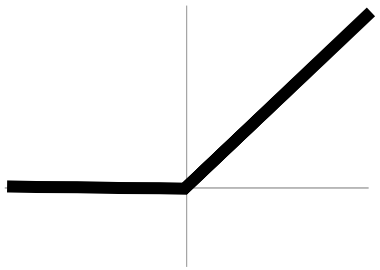


Activation Functions

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

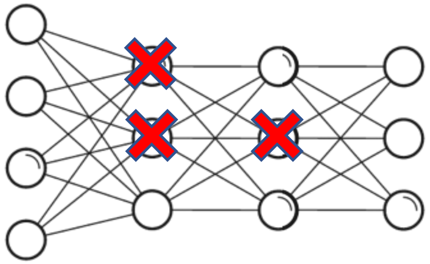
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

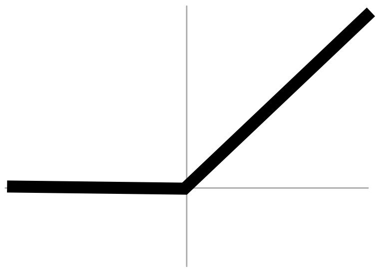


Dropout

Managing Training

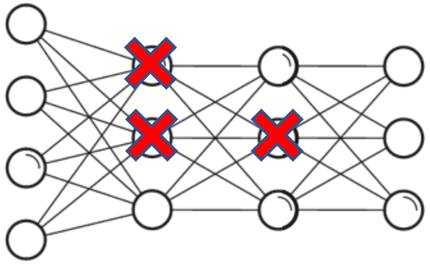
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

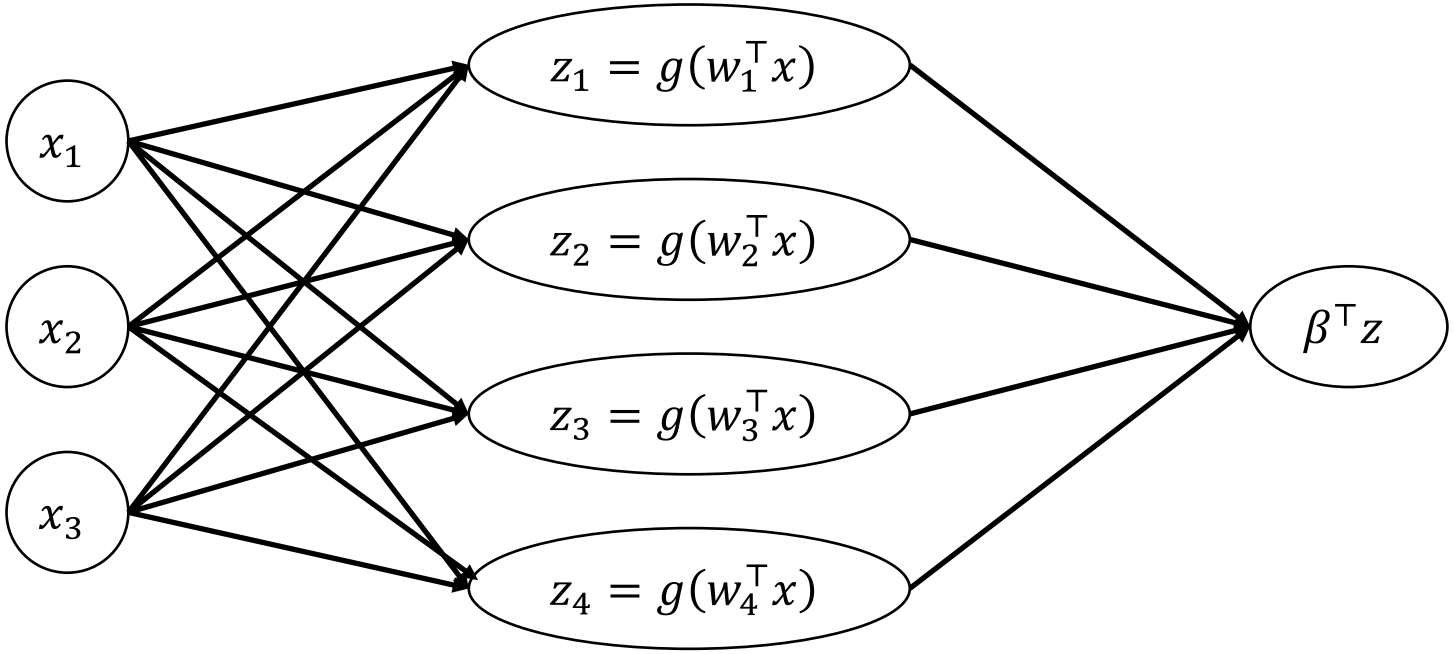


Dropout

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 $W_j \sim N\left(0, \frac{2}{d_{\text{in}}}\right)$

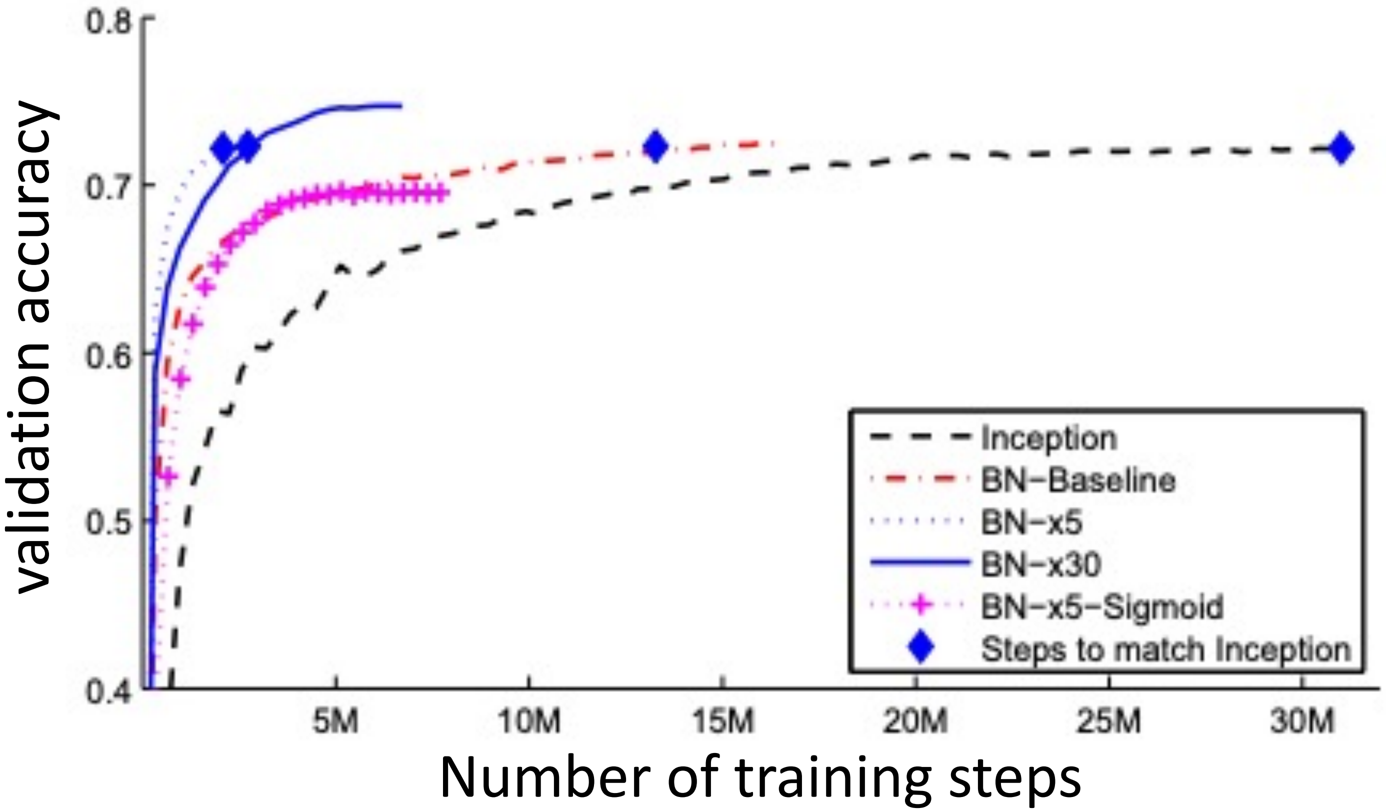
- **Xavier initialization**
  - For tanh activations, use $W_j \sim N\left(0, \frac{1}{d_{\text{in}}+d_{\text{out}}}\right)$ (d_{out} is output dimension)

Batch Normalization

- **Problem**
  - During learning, the distribution of inputs to each layer are shifting (since the layers below are also updating)
  - This “covariate shift’’ slows down learning

- **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 $z^{(j)}$ (this is differentiable!)
  - **Note:** Needs nontrivial mini-batches or will divide by zero
  - Apply after every layer (before or after activation; after can work better)

Batch Normalization

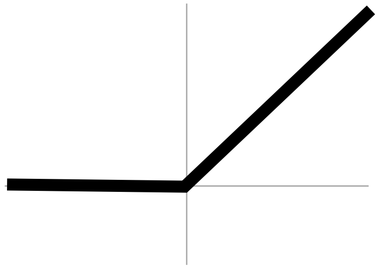


Regularization

- 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 $\|W_j\|_F = \sum_{i=1}^{k} \sum_{i'=1}^{h} W_{i,i'}^2$

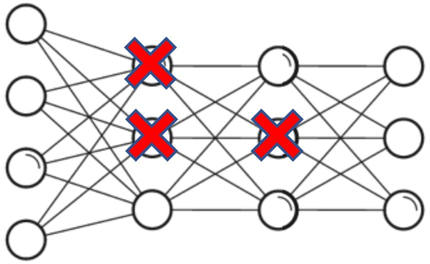
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

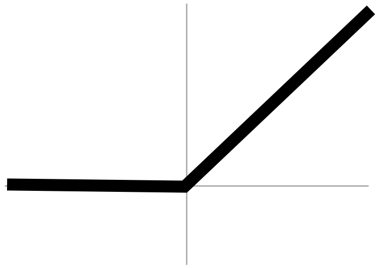


Dropout

Managing Training

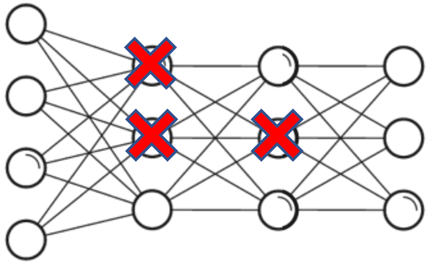
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights



Dropout

Managing Training

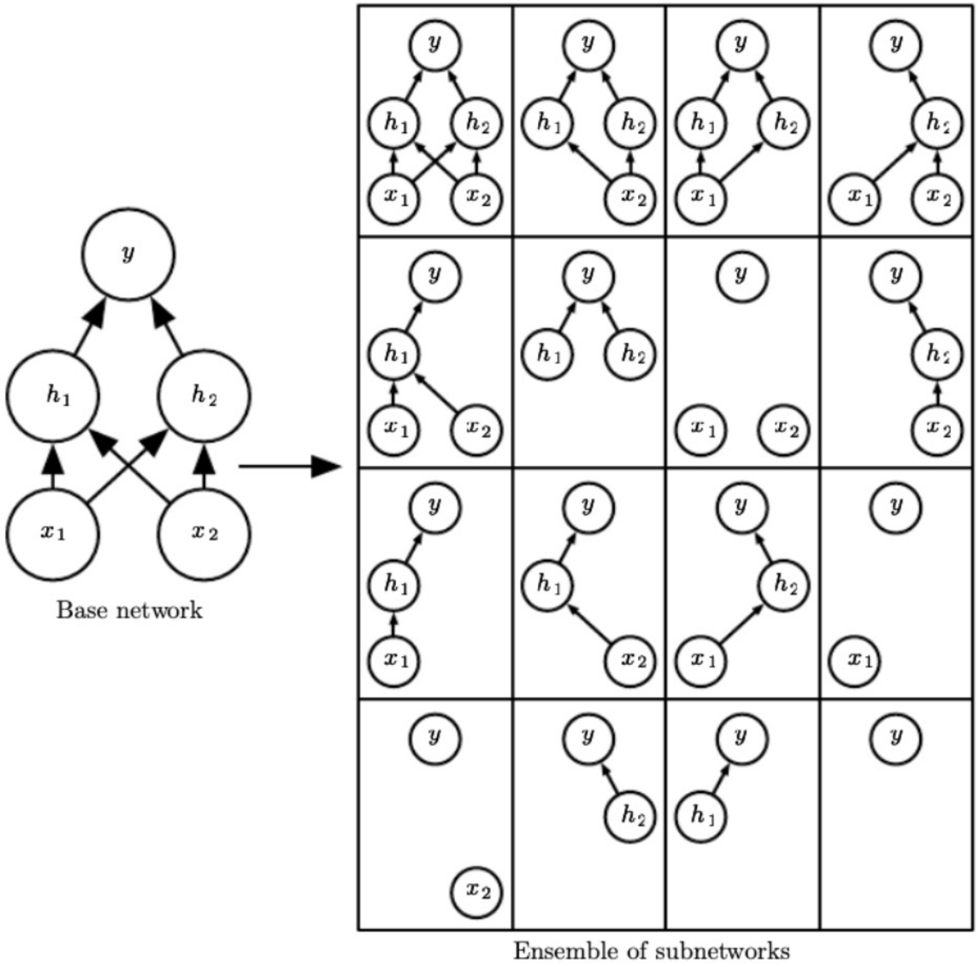
Dropout

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

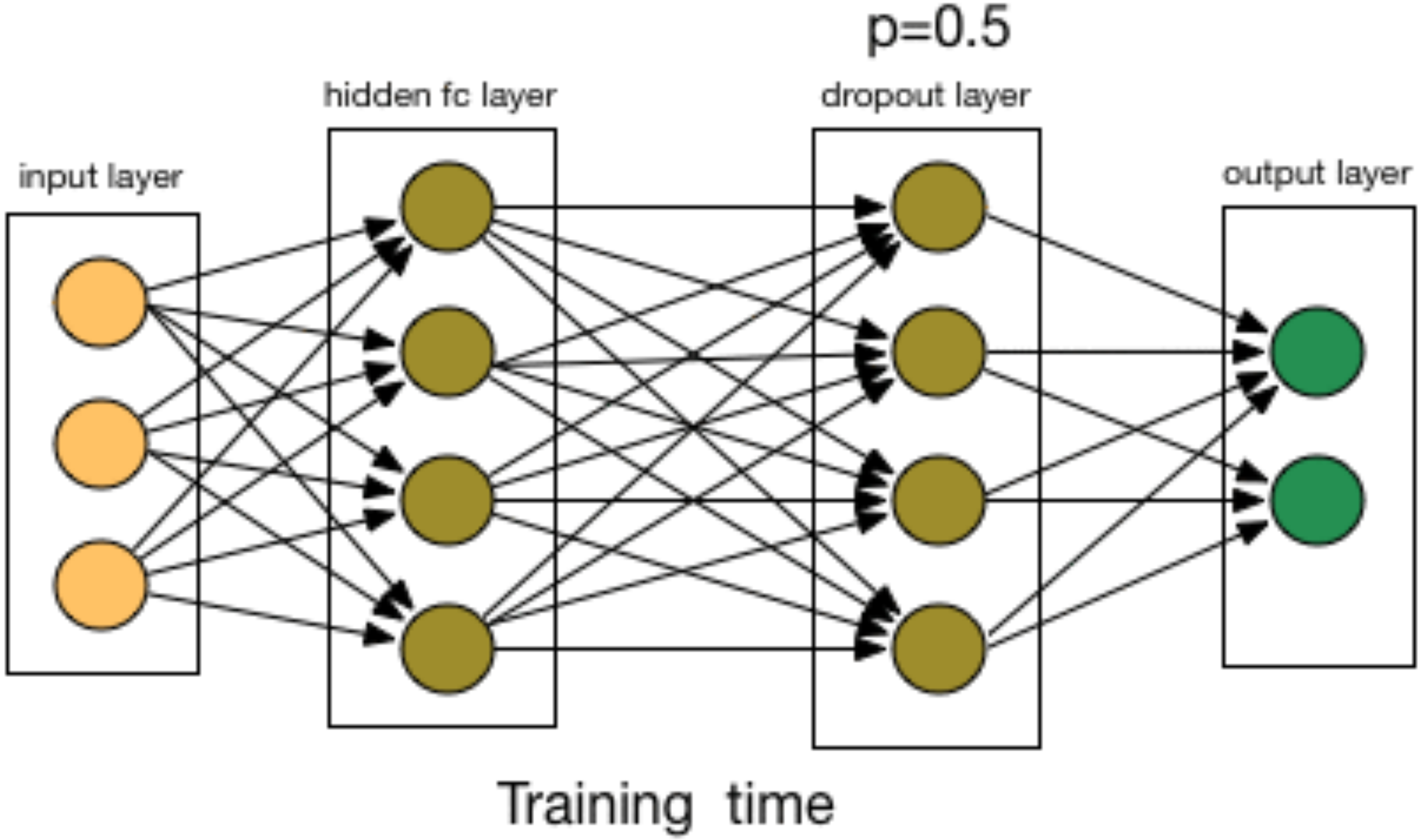
- Implemented as its own layer

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

- 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

Dropout

- **Intuition:** A form of 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

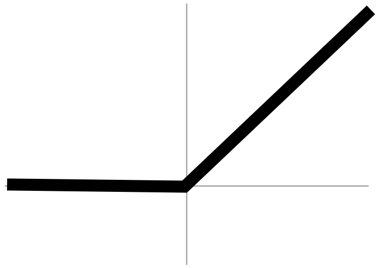
- **Connection to ensembles**
  - Each training iteration is training a slightly different network, selected at random out of $2^{\#\mathrm{neurons}}$ networks!
  - Since the networks share weights, training one network updates others

Dropout at Test Time

- **Naïve strategy:** Stop dropping neurons
  - **Problem:** Not the distribution the layer was trained on (covariate shift)!
- **Naïve strategy:** Average across all possible predictions
  - **Problem:** There are $2^{\#\text{neurons}}$ possible realizations of the randomness
- **Solution:** Turn off dropout but divide the outgoing weights by 2
  - 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

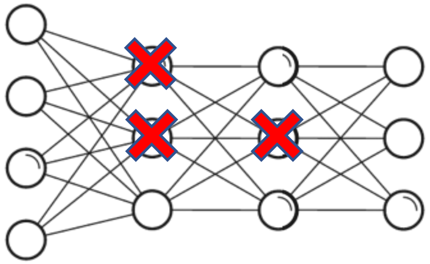
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

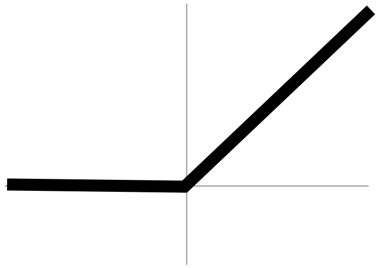


Dropout

Managing Training

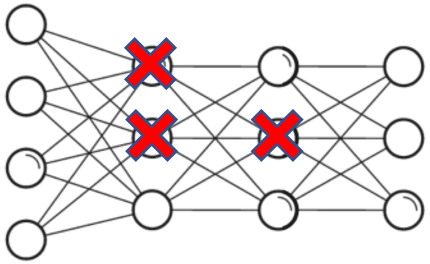
Neural Network Tips & Tricks

Optimization



Activation Functions

Managing Weights

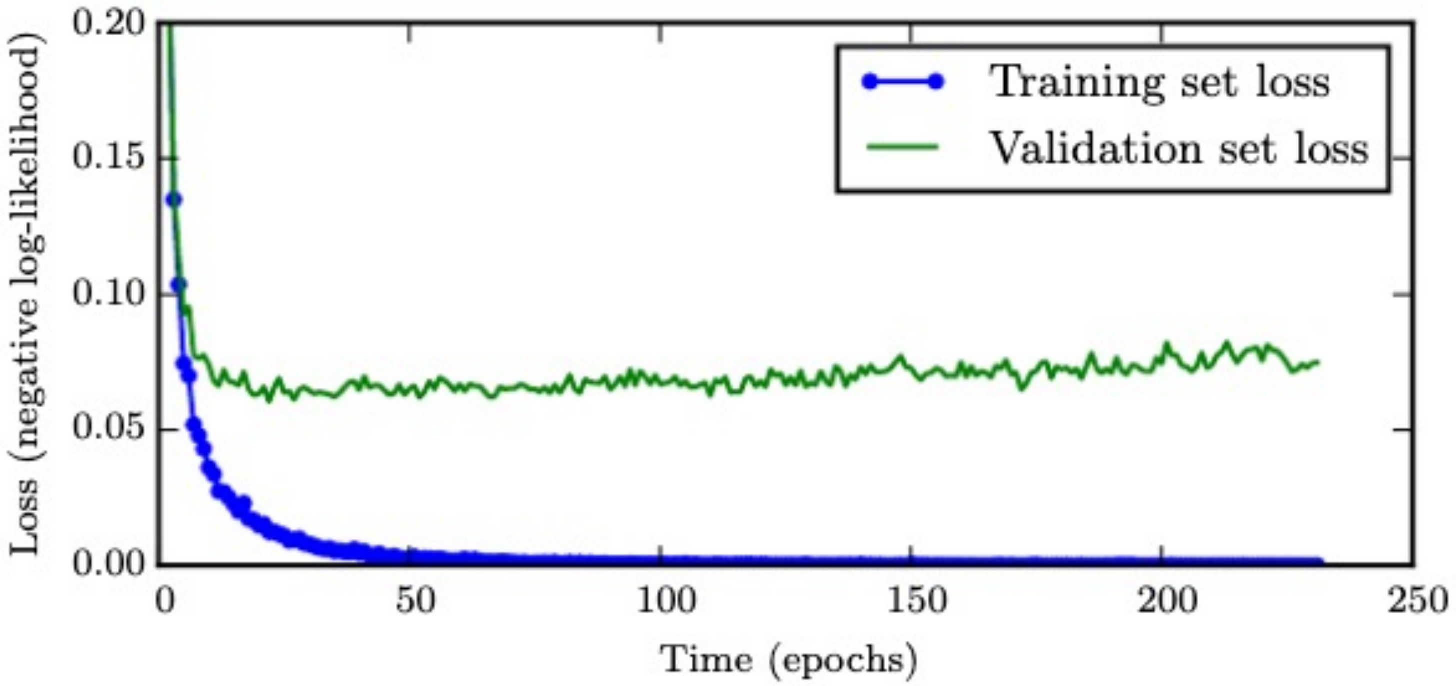


Dropout

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



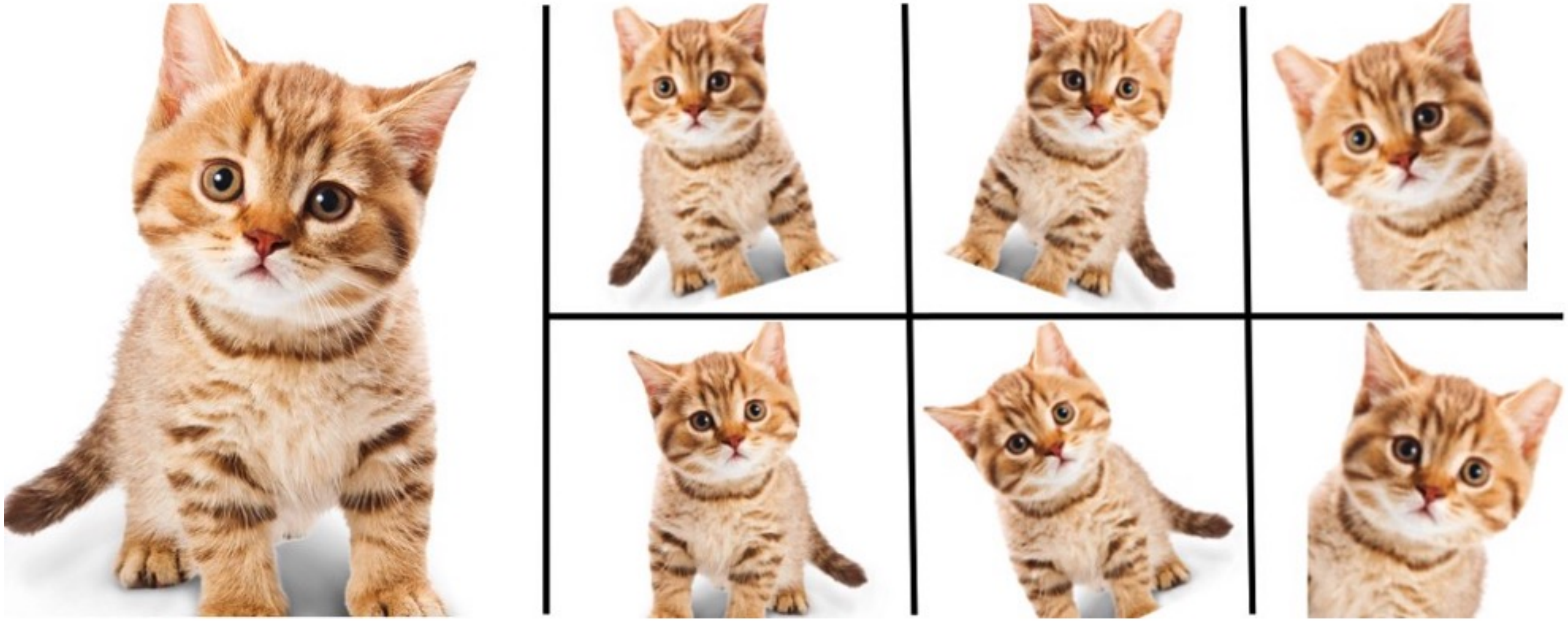
Data Augmentation

- **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

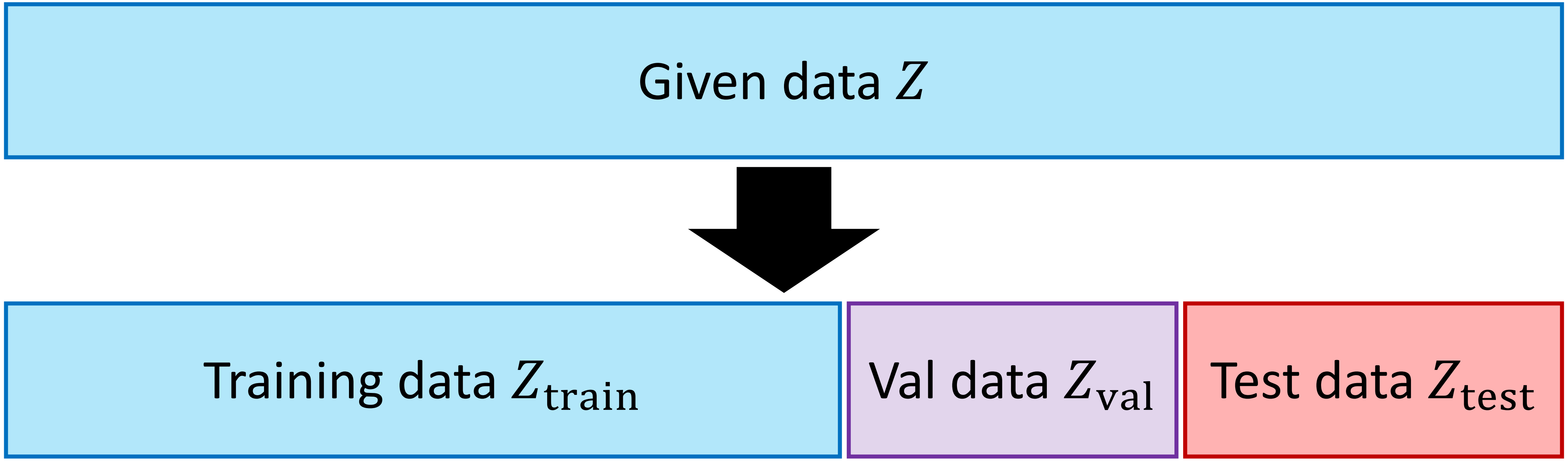
- **Recap**
- **Neural network tips and tricks**
- **Hyperparameter tuning**
- **Implementation**

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

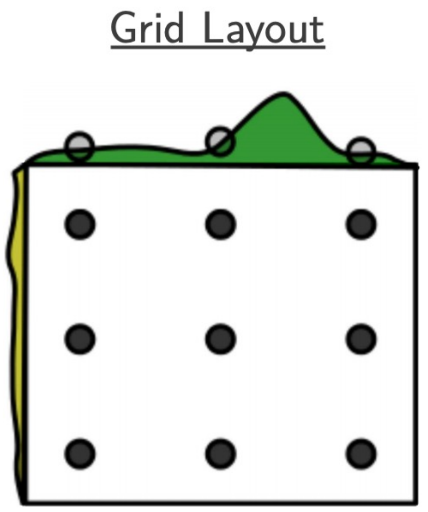


Hyperparameter Optimization Tips

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

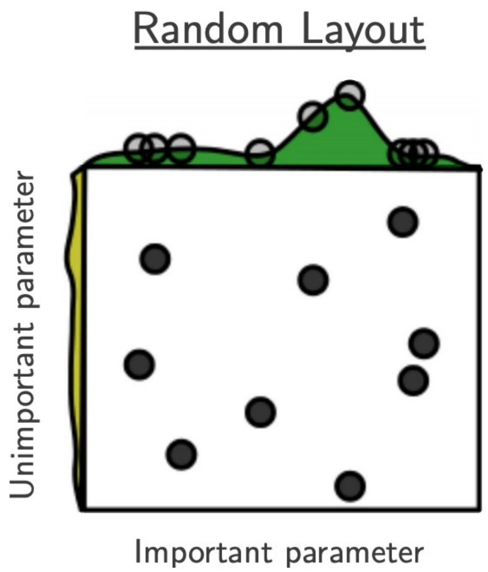
- **What about multiple hyperparameters?**
  - For 2 or 3 hyperparameters, do a systematic "grid search"



[Bergstra & Bengio, JMLR 2012]

Hyperparameter Optimization Tips

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

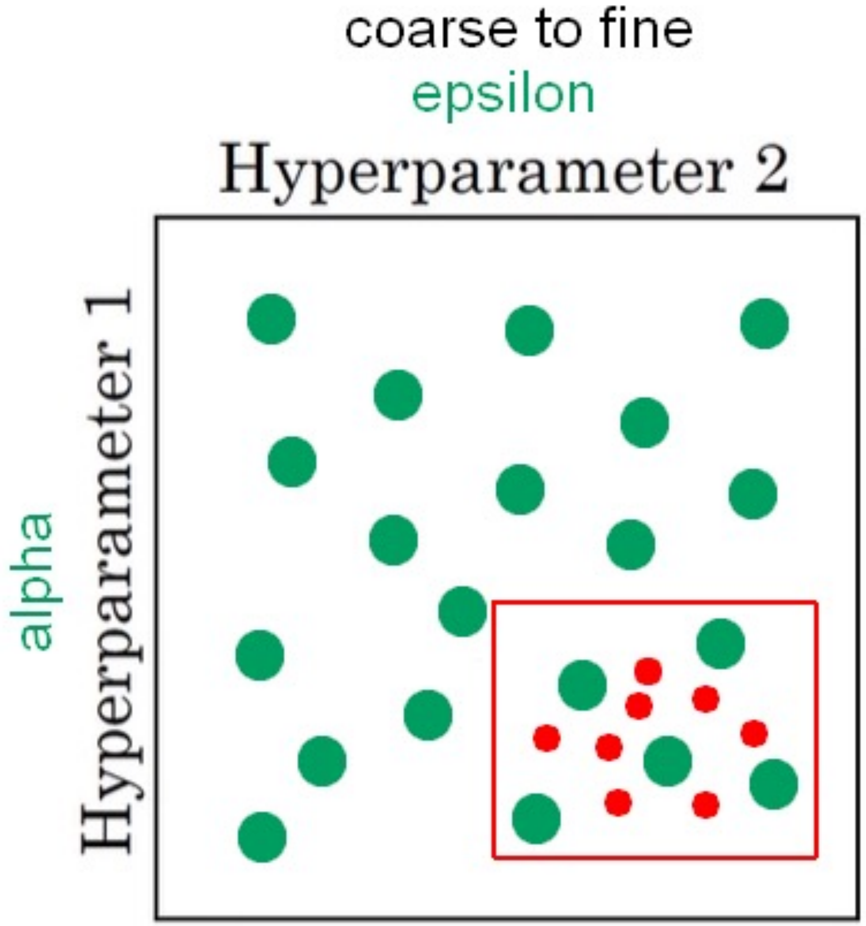


[Bergstra & Bengio, JMLR 2012]

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/

More Practical Tips

- **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

- **Additional 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

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

Common parent class: nn.Module

Constructor: Defining layers of the network

```python
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
```

Forward propagation

What about backward propagation?

Pytorch

- Open source packages have helped democratize deep learning

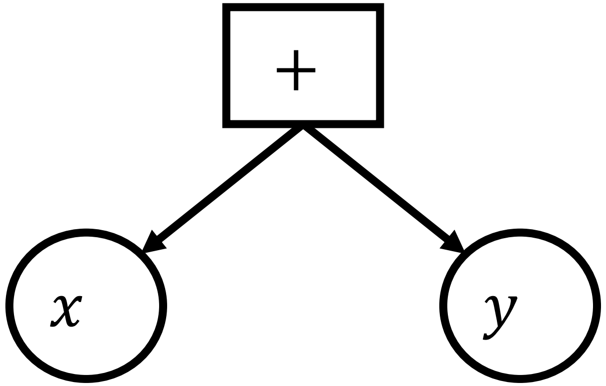
- Backpropagation implemented for all neural network architectures
  - Most modern libraries, including Tensorflow, Mxnet, Caffe, Pytorch, and Jax
  - Only need gradients of new layers

- **Basic Idea:** Provide model family as sequence of functions $[f_1, \ldots, f_m]$
  - What about more general compositions?
  - **Solution:** Composition of functions can be represented as trees (but typically called graphs)!

Computation Graphs

- The **tensor** datatype represents a **computation graph**
    - **Not just a numpy array!**
    - Instead, performing the computation produces a numpy array

- **Example:**
    - Suppose x is tensor that evaluates to $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
    - Suppose y is a tensor evaluates to $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
    - Then, $x + y$ is a tensor that evaluates to $\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$

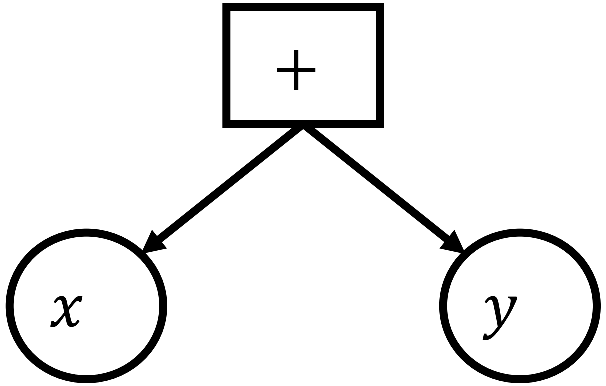


Toy Implementation of Computation Graphs

```
class Constant(tensor):
    def __init__(self, val):
        self.val = val
    def backpropagate(self):
        ...

class Add(tensor):
    def __init__(self, t1, t2):
        self.t1 = t1
        self.t2 = t2
        self.val = self.t1.val + self.t2.val
    def backpropagate(self):
        ...
```

```
x = Constant(np.array([[1, 0], [0, 1]])
y = Constant(np.array([[1, 1], [1, 0]])
z = x + y  # z has type Add
```

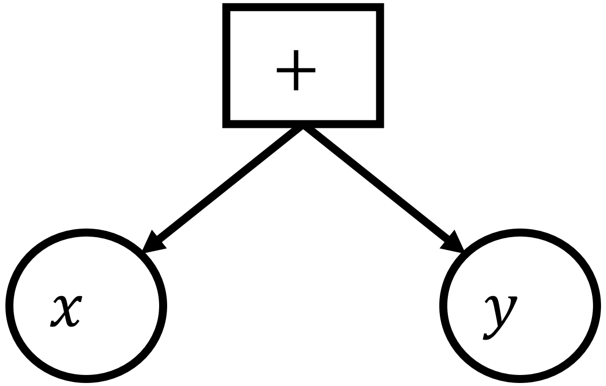


Toy Implementation of Computation Graphs

```
class Constant(tensor):
      def __init__(self, val):
            self.val = val
      def backpropagate(self):
            ...
```

```
x = Constant(np.array([[1, 0], [0, 1]])
y = Constant(np.array([[1, 1], [1, 0]])
z = x + x + y  # Z has type Add
```

```
class Add(tensor):
      def __init__(self, t1, t2):
            self.t1 = t1
            self.t2 = t2
            self.val = self.t1.val + self.t2.val
      def backpropagate(self):
            ...
```



Computation Graphs

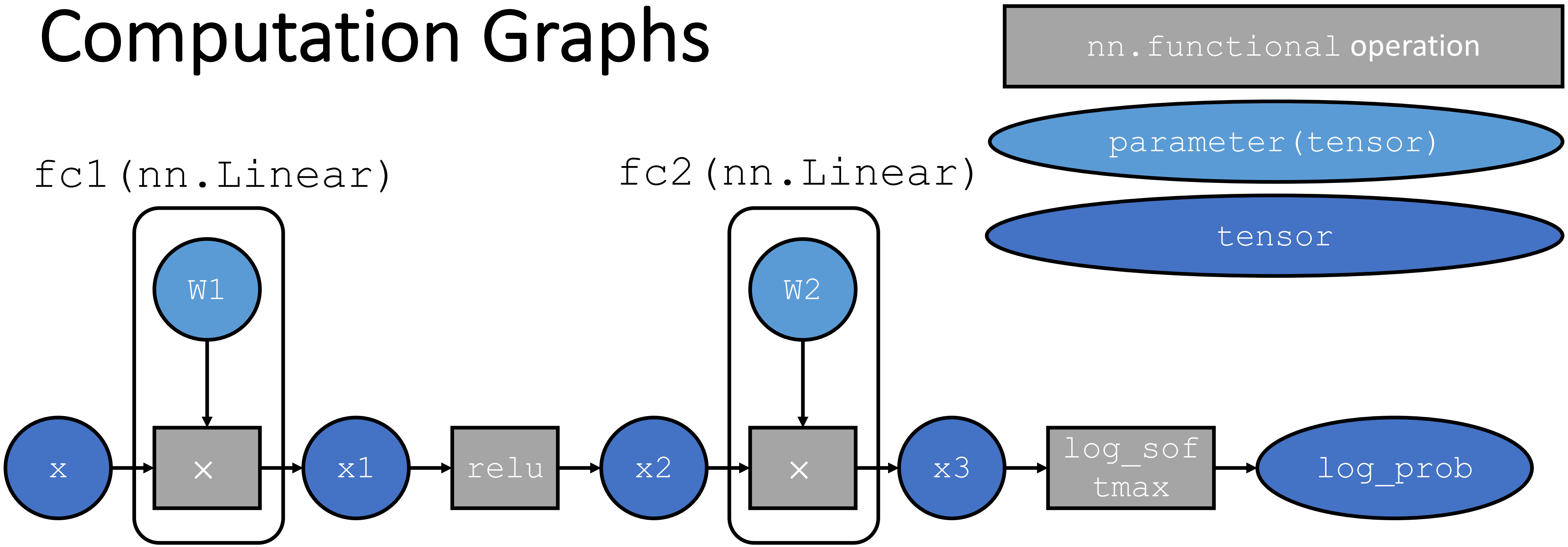
- Layers are implemented as tensors
  - **Examples:** addition, multiplication, ReLU, sigmoid, softmax, matrix multiplication/linear layers, MSE, logistic NLL, concatenation, etc.
  - You can also implement your own by providing forward pass and derivatives

- Tensors can be composed together to form neural networks

Computation Graphs

- **Forward propagation:** Values are evaluated as they are constructed
- **Backpropagation:** Automatically compute derivative of scalar with respect to all parameters based on derivatives of layers
  - `x.backwards()`
  - Does not perform any gradient updates!

Computation Graphs




```python
    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
```

Pytorch Training Loop

```
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

Zero out all old gradients

Runs forward pass model.forward(data)

Loss computation

Backpropagation

Gradient step

Pytorch Training Loop

```python
def main():
    torch.manual_seed(1)
    device = torch.device("cuda")
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=64, shuffle=True)

    model = Net().to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)
    sc                                       e=1, gamma=0.9)
    for epoch in range(1, 15):
        train(model, device, train_loader, optimizer, epoch)
        scheduler.step()
```

Load dataset

Loop over epochs (full passes over data)

Minibatch SGD for one epoch

Update base learning rate

Pytorch Model

- To use your model (once it has been trained):

```
label = model(input)
```