Deep Feedforward Networks

Anupam Datta

CMU

Spring 2018

Story so far

- Image classification problem
- Linear models
  - Score function
  - Loss function
  - Learning

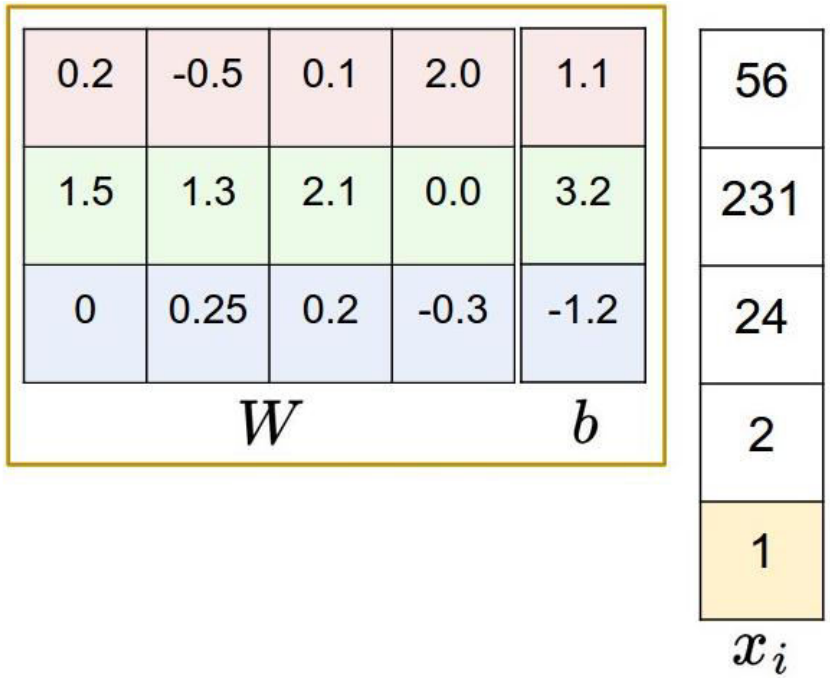
- Learning as optimization
  - Gradient descent (batch, mini-batch, stochastic)
  - Second-order methods (Newton’s method)
  - Backpropagation

Today

- From linear score functions to neural networks
  - Practical design choices
  - (Some) justification of design choices

Recall: Linear score function

$$f(x_i, W) = Wx_i$$

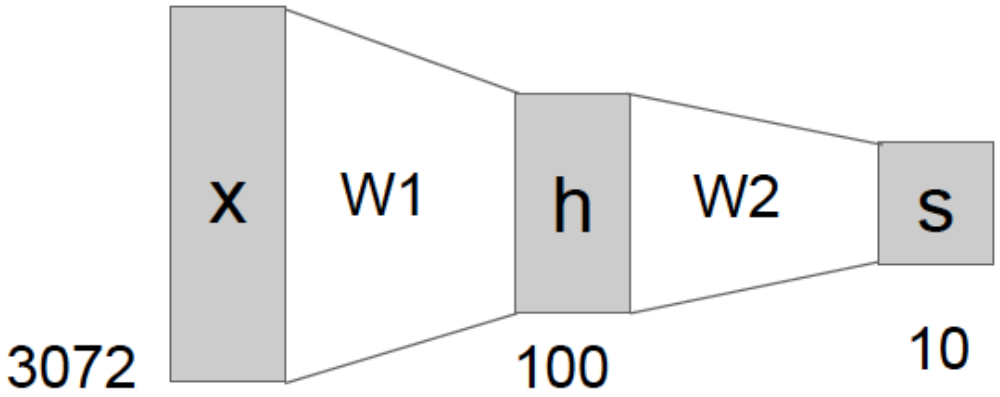


For CIFAR:

W: 10 x 3072
x: 3072 x 1
10 class scores

2-Layer neural network

$$s = W_2 \max(0, W_1 x)$$



For CIFAR:

W1: 100 x 3072
W2: 10 x 100
x: 3072 x 1
10 class scores

- Iterated construction: linear function followed by non-linear function
- Training network: learn W1, W2 using stochastic gradient descent; use backpropagation to compute gradients

Topic outline

- Setting up the architecture
- Setting up the data and the loss
- Learning and evaluation

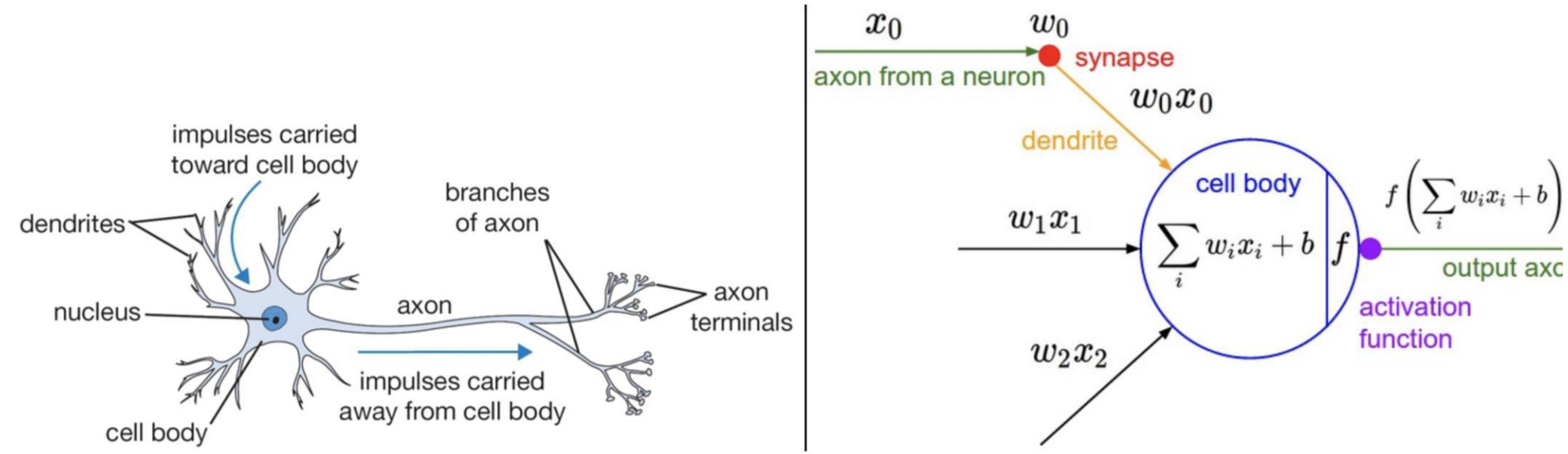
Neural network architecture

Outline

- Modeling one neuron
  - Biological motivation and connections
  - Single neuron as a linear classifier
  - Commonly used activation functions

- Neural Network architectures
  - Layer-wise organization
  - Example feed-forward computation
  - Representational power
  - Setting number of layers and their sizes

Biological motivation



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

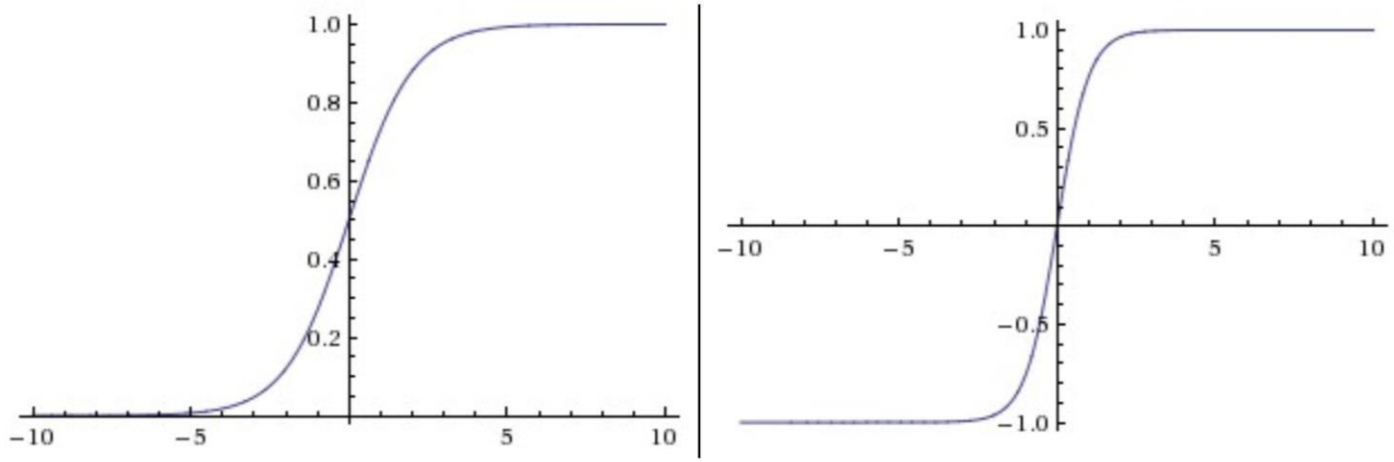
Imperfect analogy!

Single neuron as a linear classifier

- Binary softmax classifier
  - Interpret $\sigma(\sum_i w_i x_i + b)$ to be the probability of one of the classes $P(y_i = 1 \mid x_i; w)$
  - Set threshold at 0.5

- Binary SVM classifier
  - Attach a max-margin hinge loss to the output of the neuron

Commonly used activation functions

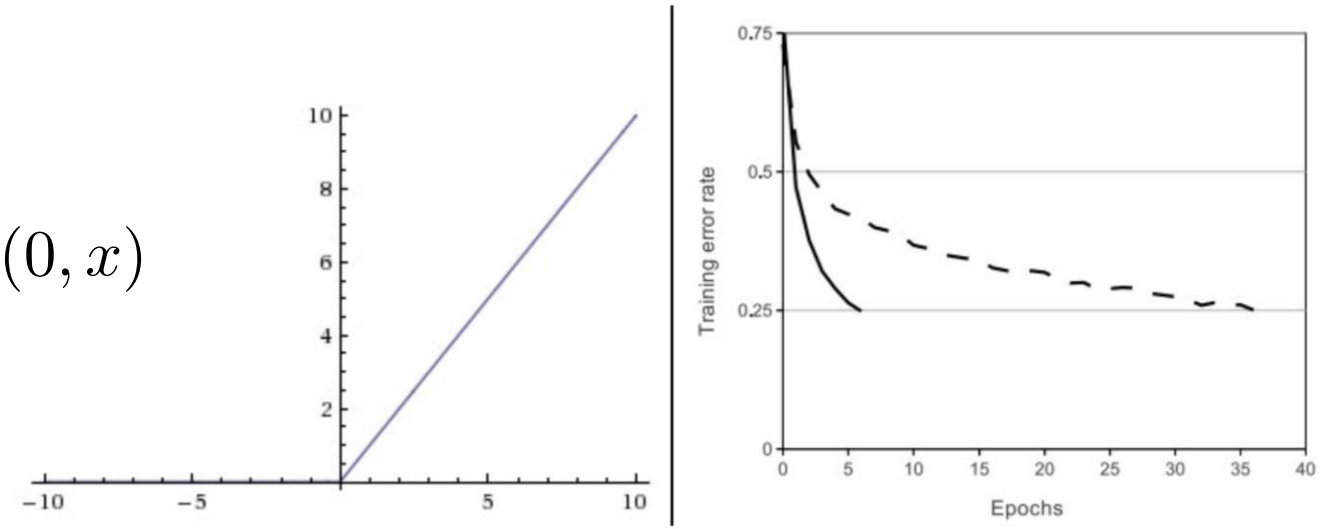


Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

- Sigmoid weaknesses:
  - saturate and kill gradients
  - outputs not zero-centered
- Tanh outputs are zero-centered

Commonly used activation functions

$$f(x) = \max(0, x)$$



Left: Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. **Right:** A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

- ReLU is often used in modern deep networks
  - Linear, non-saturating form speeds up convergence of stochastic gradient descent; efficient to compute (threshold operation)
  - If learning rate is high, then ReLU units can die i.e., never activate during subsequent training

Commonly used activation functions

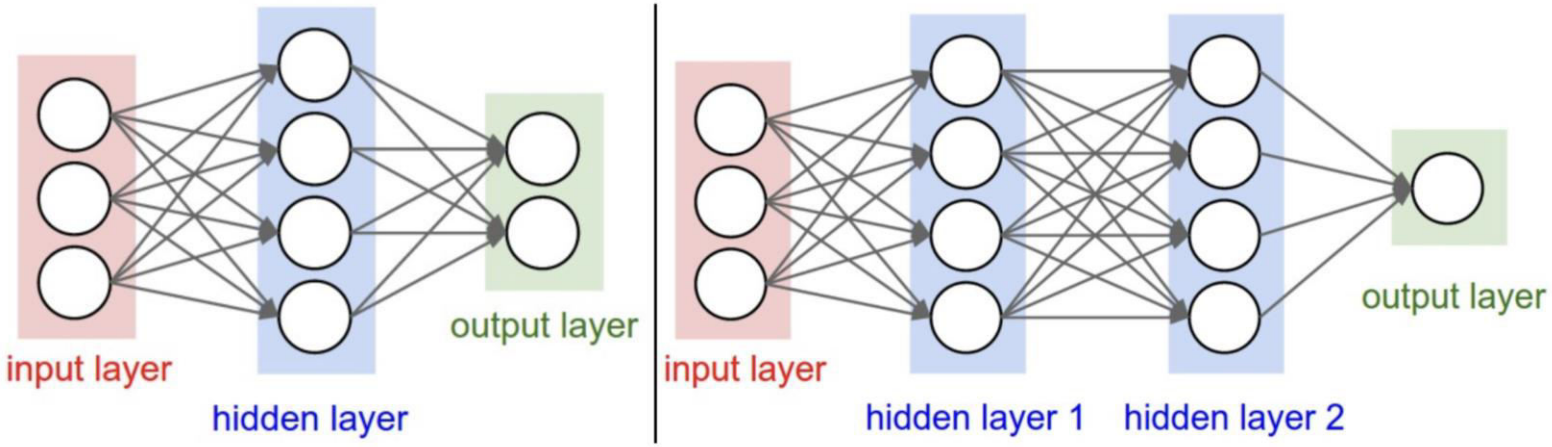
- Leaky ReLU
  - Function has small negative slope when x < 0 to avoid dying

$$f(x) = \mathcal{I}(x < 0)(\alpha x) + \mathcal{I}(x >= 0)(x)$$

- Maxout
  - Generalizes ReLU and Leaky ReLU; advantages of both but more parameters

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Neural network architectures



- Neural network as a directed acyclic graph
- Examples above: 2-layer NN and 3-layer NN
- Fully connected layer

Example feedforward computation

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

- Repeated matrix multiplications interwoven with activation function
- x could hold a batch of training data evaluated in parallel
- Output layer neurons do not go through non-linear activation function

Representational power

Neural Networks with at least one hidden layer are *universal approximators:*

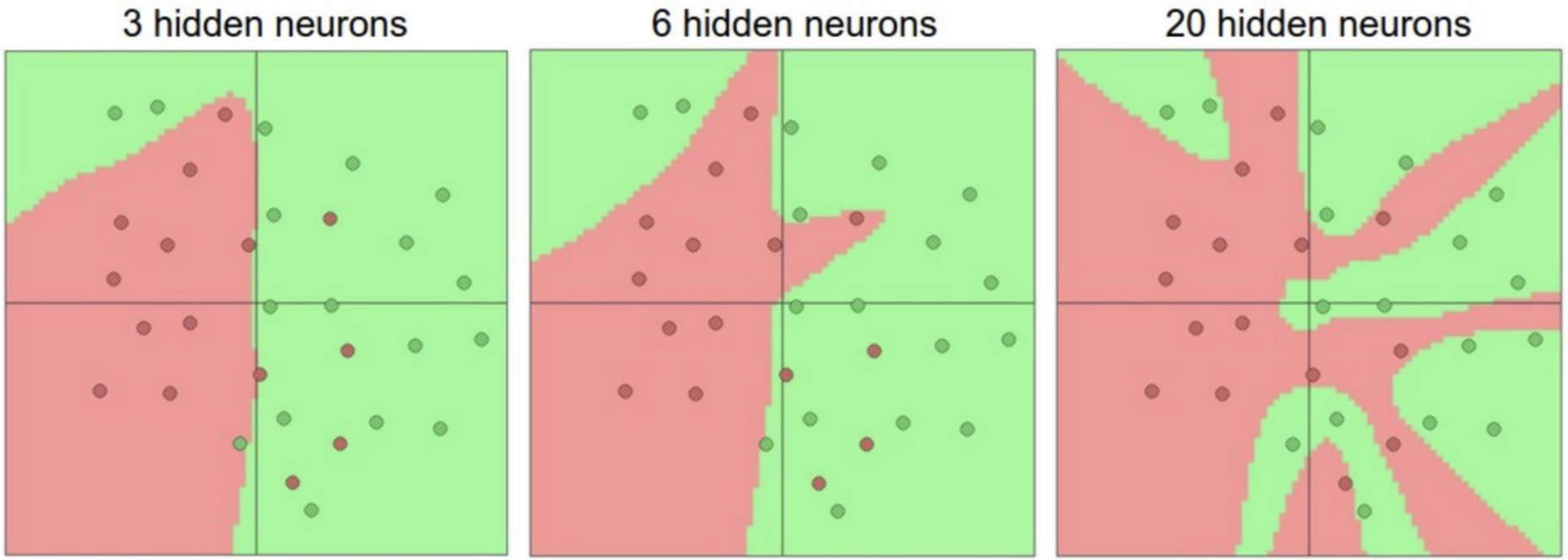
Given any continuous function $f(x)$ and some $\epsilon>0$, there exists a Neural Network $g(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, |f(x)-g(x)|<\epsilon \forall x$

Representational power

- Neural Networks work well in practice because they compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice, and are also easy to learn using our optimization algorithms (e.g. gradient descent).

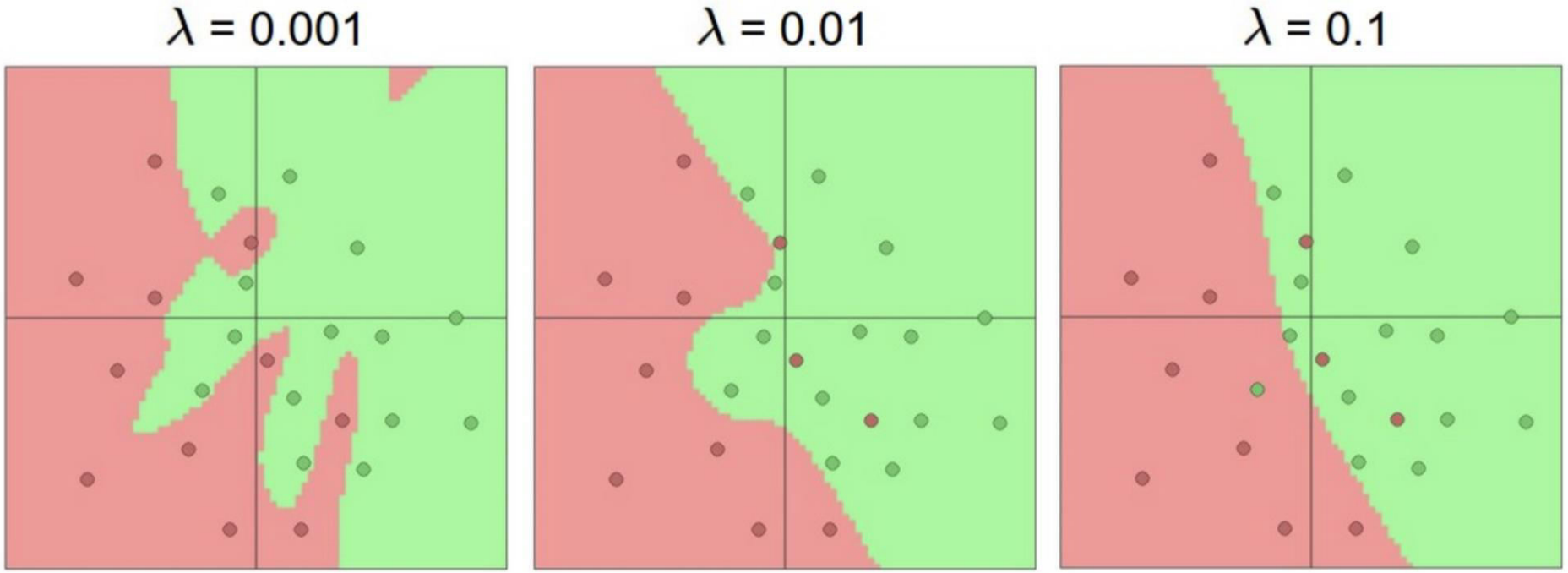
- The fact that deeper networks (with multiple hidden layers) can work better than a single-hidden-layer networks is an empirical observation, despite the fact that their representational power is equal.

Setting number of layers and their sizes



With more neurons, we have greater representation power but possibly more overfitting

Setting number of layers and their sizes



Train large network; control overfitting with regularization

The Loss Surfaces of Multilayer Networks

Setting up the data and the model

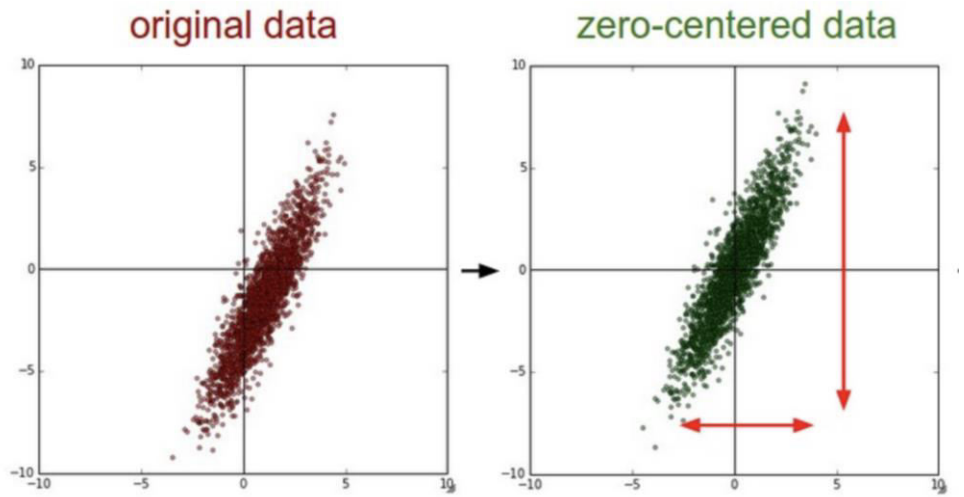
Outline

- Setting up the data and the model
  - Data Preprocessing
  - Weight Initialization
  - Regularization

- Loss functions

Data preprocessing

- Mean subtraction
  - Subtract the mean across every individual *feature* in the data

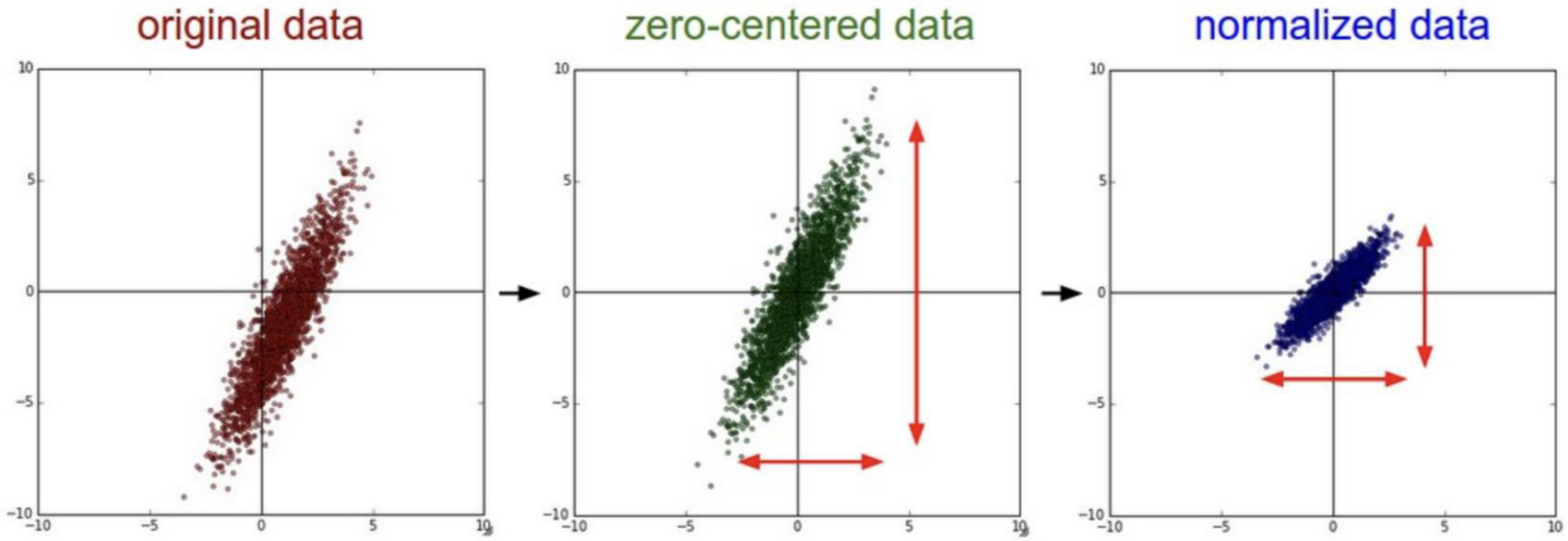



```
X -= np.mean(X, axis = 0)
```

Data matrix X, where we will assume that X is of size [N x D]
(N is the number of data, D is their dimensionality)

Data preprocessing

- Normalization
  - Divide each zero-centered feature by its standard deviation
  - Bringing data dimensions to same scale helps SGD converge




```
X /= np.std(X, axis = 0)
```

Outline

- Setting up the data and the model
  - Data Preprocessing
  - Weight Initialization
  - Regularization

- Loss functions

Weight initialization

- First attempt
  - Initialize all weights to 0
  - Not a good idea
    - Every neuron computes the same output => every neuron computes the same gradients and undergoes the same parameter updates

Weight initialization

- Important to introduce asymmetry
  - Idea: Initialize weights to independent small random numbers

```
W = 0.01* np.random.randn(D,H)
```

where `randn` samples from a zero mean, unit standard deviation gaussian.

  - Issue: Distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs

Weight initialization

- Recommended practice for initializing weights of neurons in NNs with ReLU units

```
w = np.random.randn(n) * sqrt(2.0/n)
```

where `n` is the number of its inputs

  - Every neuron's weight vector is sampled from a multi-dimensional gaussian normalized by its variance

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Weight initialization (under simplifying assumptions)

- Initialize weights of NN as follows

```
w = np.random.randn(n) * sqrt(1.0/n)
```

where `n` is the number of its inputs

  - Every neuron's weight vector is sampled from a multi-dimensional gaussian normalized by its variance

Based on: Understanding the difficulty of training deep feedforward neural networks

Weight initialization (under simplifying assumptions)

$$s = \sum_i^n w_i x_i$$

$$\mathrm{Var}(s) = \mathrm{Var}(\sum_i^n w_i x_i)$$

Want: Var(s) = Var(x)
Need: Var(w) = 1/n

Simplifying assumptions

Zero mean

Identically distributed

Var(aX) = a^2Var(X)
So, draw w from unit Gaussian and scale by 1/sqrt(n)

Weight initialization (under simplifying assumptions)

$$s = \sum_i^n w_i x_i$$

$$\mathrm{Var}(s) = \mathrm{Var}(\sum_i^n w_i x_i)$$

$$= \sum_i^n \mathrm{Var}(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

Want: Var(s) = Var(x)
Need: Var(w) = 1/n

Simplifying assumptions

Zero mean

$$= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

Identically distributed

$$= (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

Var(aX) = a^2Var(X)
So, draw w from unit Gaussian and scale by 1/sqrt(n)

Bias initialization

- Initialize biases to 0

Outline

- Setting up the data and the model
  - Data Preprocessing
  - Weight Initialization
  - Regularization

- Loss functions

Recall: loss function

$$L = \underbrace{\frac{1}{N}\sum_{i} L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

Regularization

- L2 regularization

For every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength

- Encourages the network to use all of its inputs a little rather than some of its inputs a lot
- During gradient descent parameter update, every weight is decayed linearly toward zero

Regularization

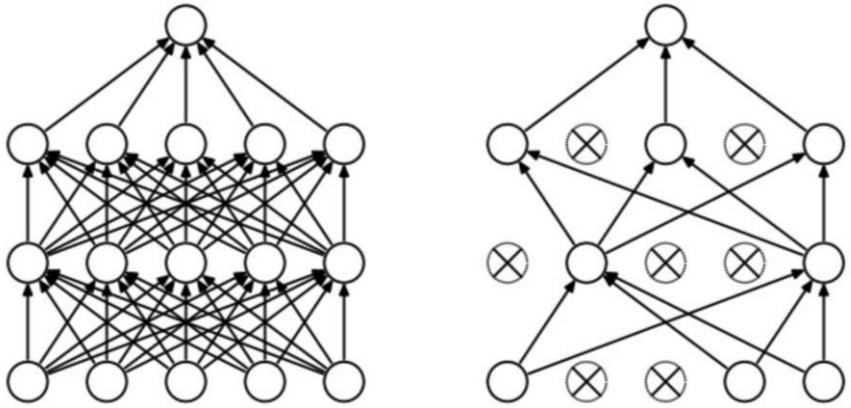
- L1 regularization

For every weight w in the network, we add the term $\lambda|w|$ to the objective, where λ is the regularization strength

- Encourages the network to use some of its inputs a lot (i.e. sparse weight vectors)
- If explicit feature selection is not a goal, L2 regularization usually performs better than L1 regularization

Regularization

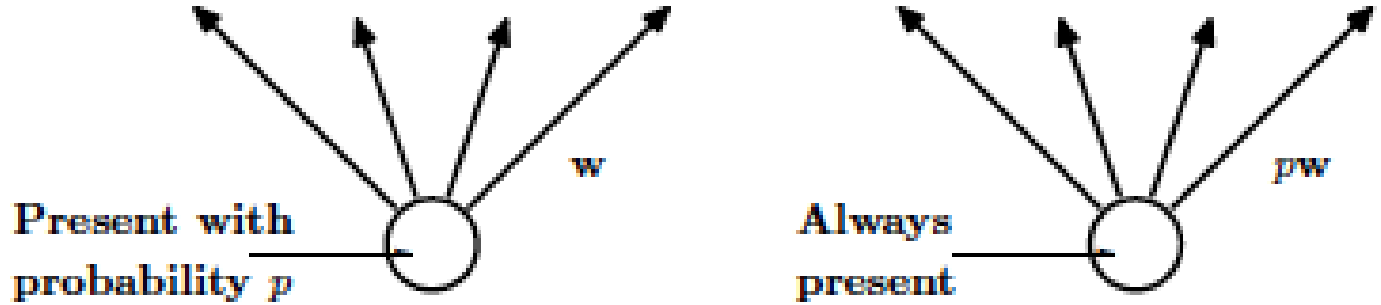
- Dropout
  - Sample a neural network within the full network and only update its parameters
  - Typically hidden units retained with p = 0.5, input units with p close to 1



(a) Standard Neural Net (b) After applying dropout.

[Dropout: A Simple Way to Prevent Neural Networks from Overfitting]

Regularization: Dropout



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Regularization: Dropout

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Regularization: Inverted Dropout

```
"""
Inverted Dropout: Recommended implementation example.
We drop and scale at train time and don't do anything at test time.
"""

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Regularization

In practice

It is most common to use a single, global L2 regularization strength that is cross-validated. It is also common to combine this with dropout applied after all layers. The value of p=0.5 is a reasonable default, but this can be tuned on validation data.

Outline

- Setting up the data and the model
  - Data Preprocessing
  - Weight Initialization
  - Regularization

- Loss functions

Loss functions for classification

Data loss

$$L = \frac{1}{N} \sum_i L_i$$

- SVM loss

$$L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$$

- Cross-entropy loss

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

Learning and evaluation

Outline

- Gradient checks
- Monitoring the learning process
- Parameter updates
- Hyperparameter Optimization
- Evaluation
  - Model Ensembles

Gradient checks

- SGD uses gradients that we computed analytically using calculus
- Issue: How do we check that we did not make errors?
- Compare *analytic gradient* implementation to the *numerical gradient*

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

$$\frac{f(x+h) - f(x)}{h} \text{ where } h \approx 10^{-5}$$

Gradient checks: tip

- Use centered formula

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

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

$$\frac{f(x+h) - f(x)}{h} \text{ where } h \approx 10^{-5}$$

$$\frac{f(x+h) - f(x-h)}{2h} \text{ where } h \approx 10^{-5}$$

Use Taylor expansion of f(x+h) and f(x−h): first formula has an error on order of O(h), while the second formula only has error terms on order of O(h^2)

Gradient checks:tip

- Use relative error
  - Analytical gradient: f'_a
  - Numerical gradient: f'_n

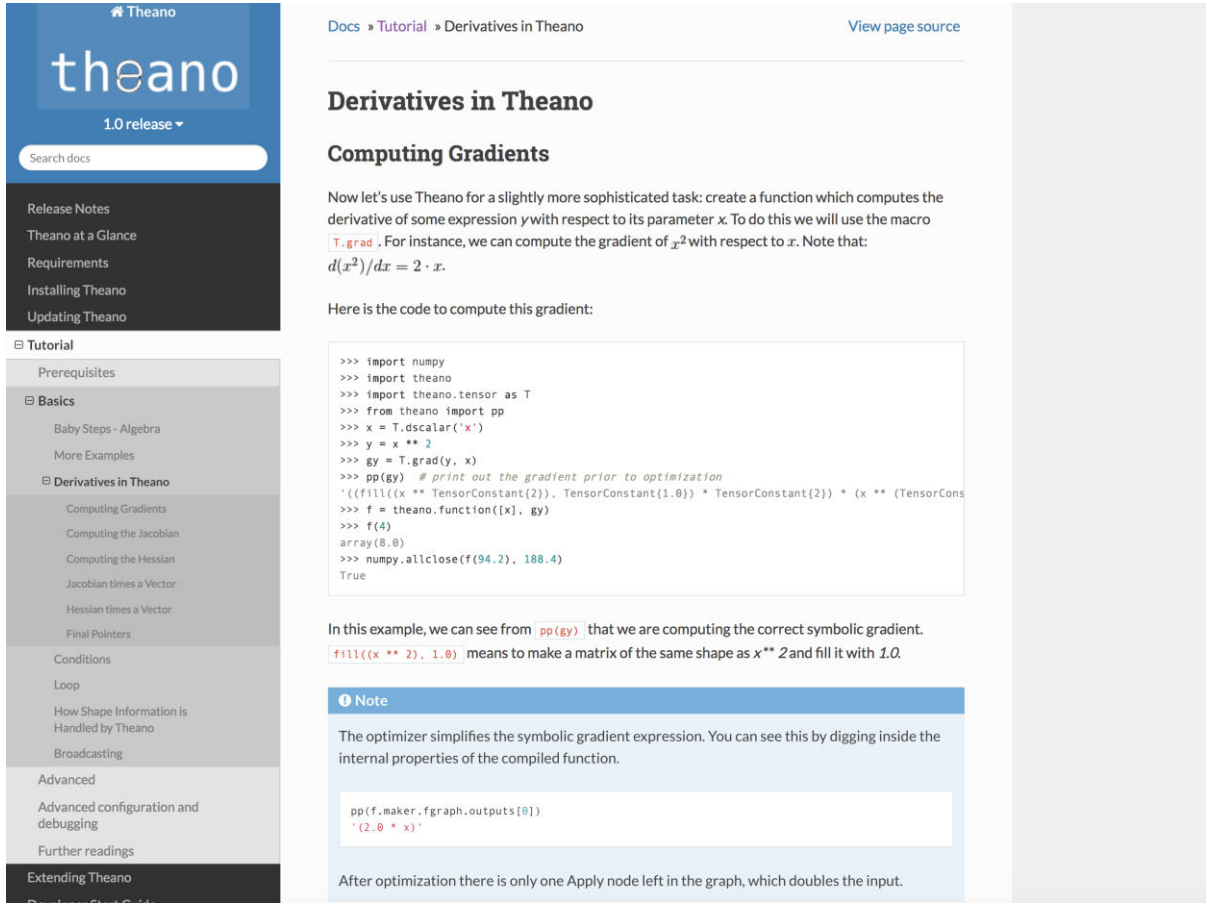
$$|f'_a - f'_n|$$

$$\frac{|f'_a - f'_n|}{\max(|f'_a|, |f'_n|)}$$

- In practice:
  - relative error > 1e-2 usually means the gradient is probably wrong
  - 1e-2 > relative error > 1e-4 should make you feel uncomfortable
  - 1e-4 > relative error is usually okay for objectives with kinks. But if there are no kinks (e.g. use of tanh nonlinearities and softmax), then 1e-4 is too high.
  - 1e-7 and less you should be happy.

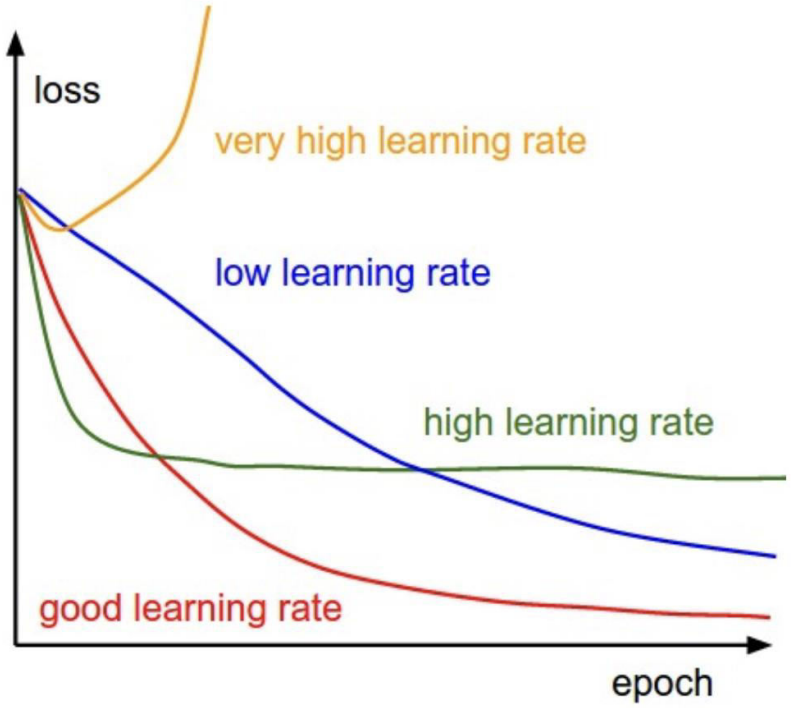
Gradient computation

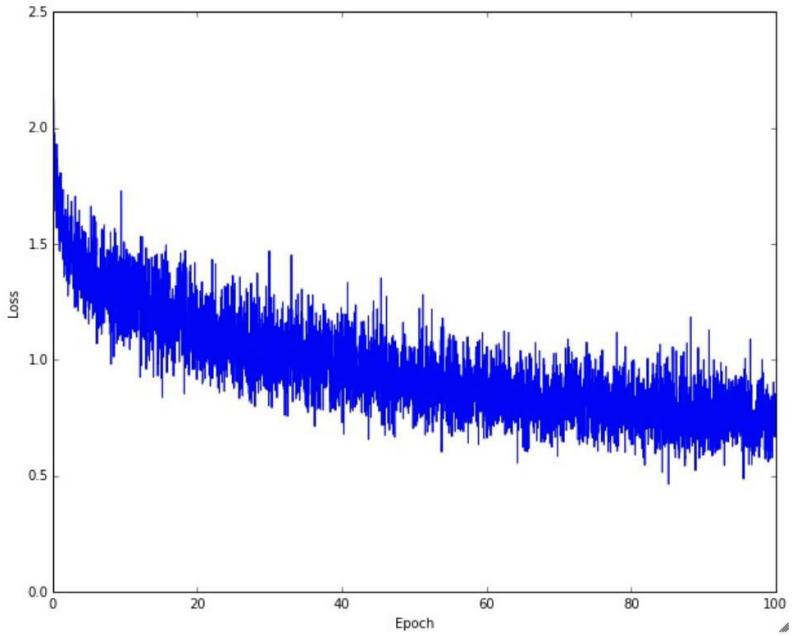
- Symbolic (analytical) differentiation available in deep learning libraries



Monitoring during learning

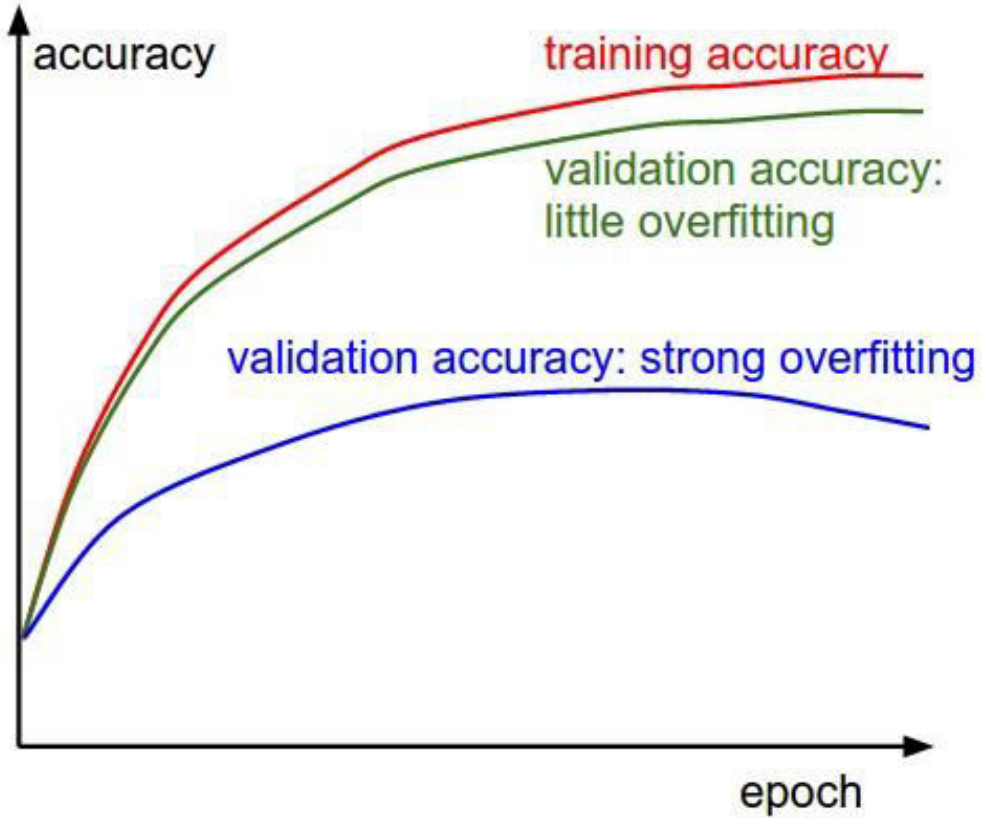
- Loss function





Monitoring during learning

- Train/Val accuracy



Monitoring during learning

- Ratio of updates:weights

A rough heuristic is that this ratio should be somewhere around 1e-3.

If it is lower than this then the learning rate might be too low. If it is higher then the learning rate is likely too high.

Monitoring during learning

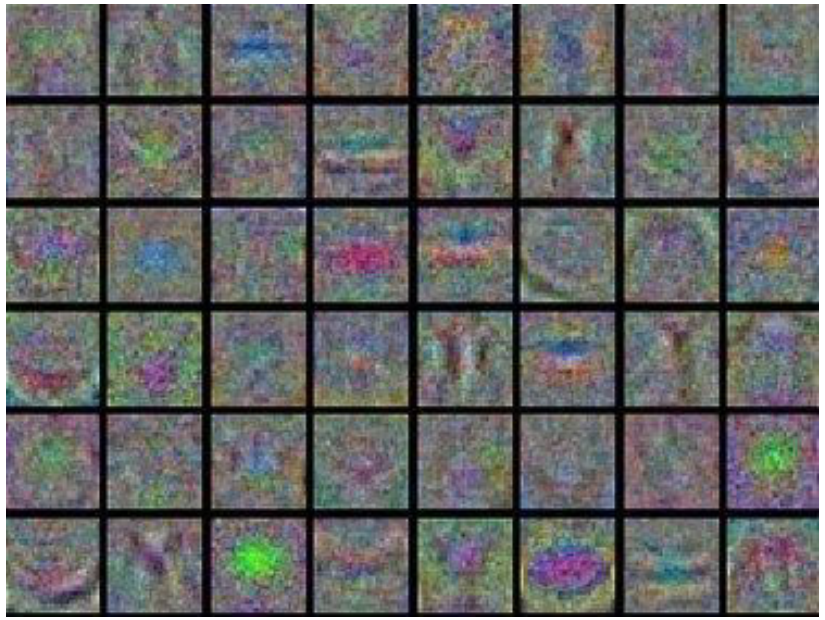
- Activation / Gradient distributions per layer
  - plot activation/gradient histograms for all layers of the network
  - not a good sign to see any strange distributions
  - e.g. with tanh neurons we would like to see a distribution of neuron activations between the full range of [-1,1], instead of seeing all neurons outputting zero, or all neurons being completely saturated at either -1 or 1.

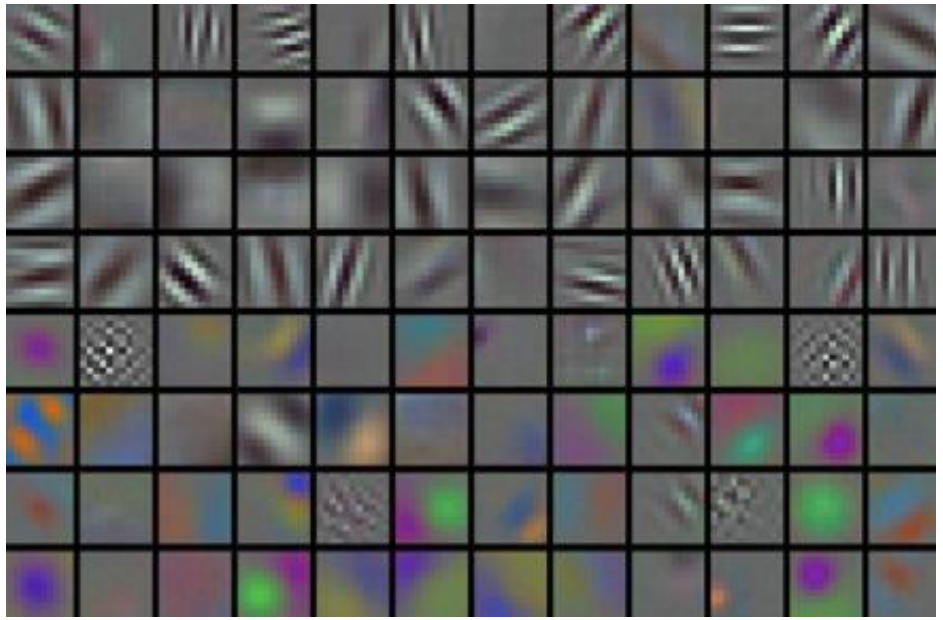
Annealing the learning rate

- Step decay
  - Reduce the learning rate by some factor every few epochs.
  - Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model.
  - One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.

Monitoring during learning

- First-layer Visualizations





Parameter updates

- Vanilla SGD

```
# Vanilla update
x += - learning_rate * dx
```

where learning_rate is a hyperparameter - a fixed constant.

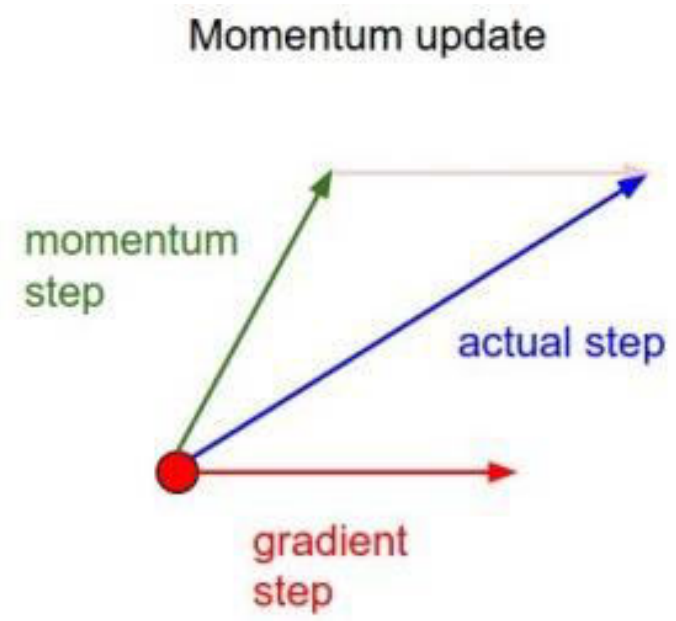
Parameter updates

- Momentum update

Viscous drag

"Gravity" along slope

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```



A typical setting is to start with momentum of about 0.5 and anneal it to 0.99 or so over multiple epochs (cf. learning rate is decreased over time)

Second-order methods

- Newton's method does not scale (earlier lecture)
  - Computing inverse Hessian explicitly is too expensive
- Quasi-newton method L-BFGS works quite well
  - Iteratively build up limited memory approximation of Hessian

Dean et al. Large Scale Distributed Deep Networks

Acknowledgment

Based in part on material from Stanford CS231n
http://cs231n.github.io/