

Text Classification 2 / Neural Network Basics

CSE 5525: Foundations of Speech and Language Processing
<https://shocheen.github.io/cse-5525-fall-2025/>



THE OHIO STATE UNIVERSITY

Sachin Kumar (kumar.1145@osu.edu)

Logistics

- HW1 is due one week from today.
 - Have you started working on it?
 - Have you read the instructions / explored the code or dataset?
 - Any questions?
- Final Projects
 - A 1-page proposal for the final project will be due early October.
 - Please start forming your teams (2-3 typically)
 - We will post example projects later this week, please talk to me or Harsh for advice on your project.

Recap: Text Classification

1. How do we evaluate our classifier f ?

- (Keyword for this section: ... evaluation)

2. How do we “digest” text into a form usable by a function?

- (Keywords for this section: features, feature extraction, feature selection, representations)

3. What kinds of strategies might we use to create our function f ?

- (Keyword for this section: models)



Recap: Text Classification



1. ~~How do we evaluate our classifier f ?~~

- (Keyword for this section: ... evaluation)

2. ~~How do we “digest” text into a form usable by a function?~~

- (Keywords for this section: features, feature extraction, feature selection, representations)

3. What kinds of strategies might we use to create our function f ?

- (Keyword for this section: models)

Binary classification in logistic regression

- Given a series of input/output pairs:
 - $(\mathbf{x}^{(i)}, y^{(i)})$
- For each observation $\mathbf{x}^{(i)}$
 - We represent $\mathbf{x}^{(i)}$ by a feature vector $\{x_1, x_2, \dots, x_n\}$
 - We compute an output: a predicted class $\hat{y}^{(i)} \in \{0, 1\}$

Features in logistic regression

- For feature $x_i \in \{x_1, x_2, \dots, x_n\}$, weight $w_i \in \{w_1, w_2, \dots, w_n\}$ tells us how important is x_i
 - x_i = "review contains 'awesome'": $w_i = +10$
 - x_j = "review contains horrible": $w_j = -10$
 - x_k = "review contains 'mediocre'": $w_k = -2$

How to do classification

- For each feature x_i , weight w_i tells us importance of x_i
 - (Plus we'll have a bias b)
 - We'll sum up all the weighted features and the bias

$$z = \left(\sum_{i=1}^n w_i x_i \right) + b$$

$$z = w \cdot x + b$$

If this sum is high, we say $y=1$; if low, then $y=0$

Formalizing “sum is high”

- We’d like a principled classifier that gives us a probability
- We want a model that can tell us:
 - $p(y=1|x; \theta)$
 - $p(y=0|x; \theta)$

The problem: z isn't a probability, it's just a number!

- z ranges from $-\infty$ to ∞

$$z = w \cdot x + b$$

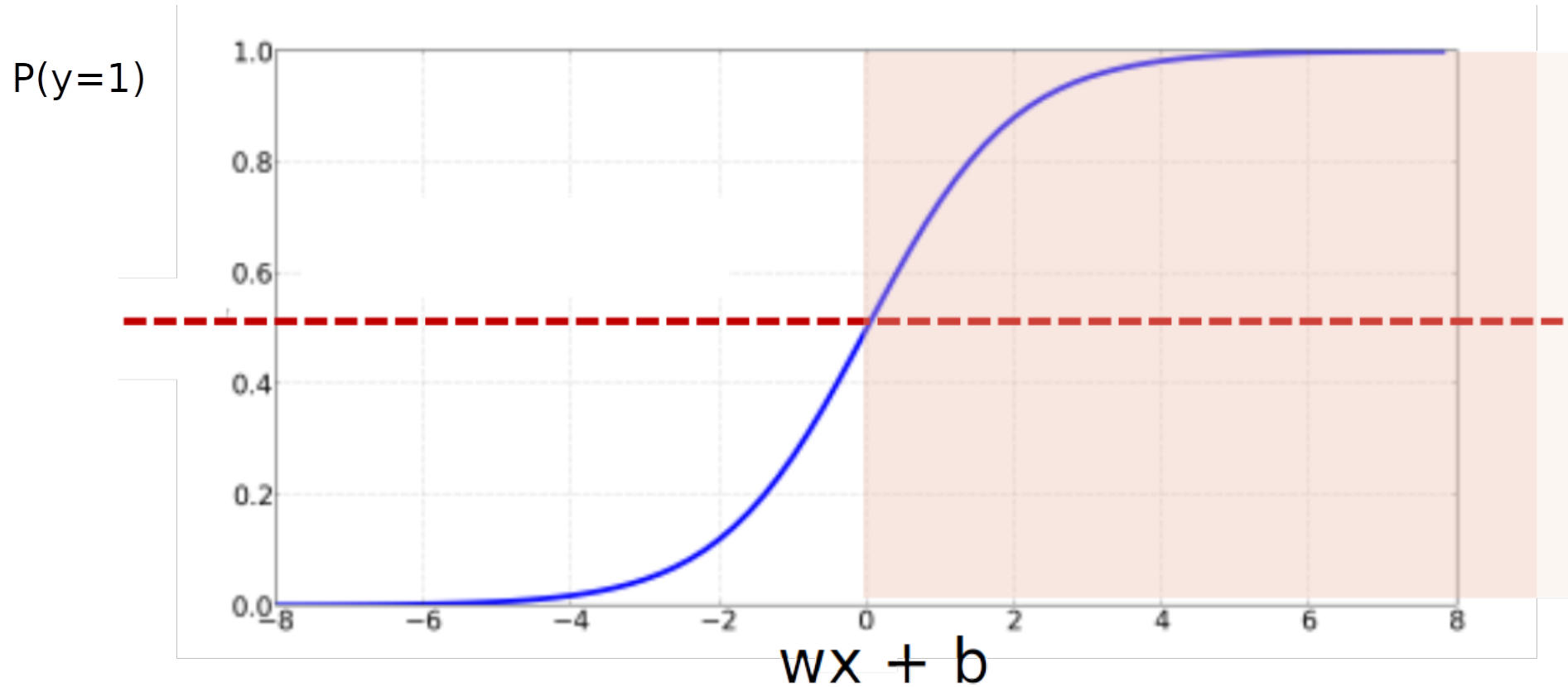
- **Solution:** use a function of z that goes from 0 to 1

“sigmoid” or
“logistic” function

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + \exp(-z)}$$

The probabilistic classifier

$$P(y = 1) = \sigma(w \cdot x + b)$$
$$= \frac{1}{1 + \exp(-(w \cdot x + b))}$$



Where do the weights (W) come from?

- Supervised classification:
 - At training time, we know the correct label y (either 0 or 1) for each x .
 - But what the system produces at inference time is an estimate \hat{y}
- We want to set w and b to minimize the **distance** between our estimate $\hat{y}^{(i)}$ and the true $y^{(i)}$
 - We need a distance estimator: a **loss function** or an objective function
 - We need an **optimization algorithm** to update w and b to minimize the loss

Learning components in LR

A **loss function**:

- **cross-entropy loss**

An **optimization algorithm**:

- **gradient descent**

Loss function: the distance between \hat{y} and y

We want to know how far is the classifier output $\hat{y} = \sigma(w \cdot x + b)$

from the true output: y [= either 0 or 1]

We'll call this difference: $L(\hat{y}, y)$ = how much \hat{y} differs from the true y

Training Objective: Maximize the Likelihood of the Training Data.

We choose the parameters \mathbf{w}, \mathbf{b} that maximize

- the probability (aka likelihood)
- of the true \mathbf{y} labels in the training data
- given the observations \mathbf{x}

Training Data

- Our training data (also known as the training corpus) is a list of input/output pairs:
 - $D = [(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})]$
 - Each $x^{(i)}$ is a document (or a paragraph or a sentence) --- piece of text. In general also called an observation.
 - Each $y^{(i)}$ is a label (0 or 1 in case of binary classification)

Deriving the objective for a single observation x

Goal: maximize likelihood of the correct label under the model

The predicted probability for class 1 is \hat{y} .

If the correct label is 1, then the likelihood is \hat{y} .

If the correct label is 0, then the likelihood is $1-\hat{y}$

We can express the likelihood from our classifier:

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Deriving the objective for a single observation x

Goal: maximize likelihood

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Noting:

if $y=1$, this simplifies to \hat{y}
if $y=0$, this simplifies to $1 - \hat{y}$

Deriving the objective for a single observation x

Goal: maximize likelihood

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Now take the log of both sides (mathematically handy)

Maximize:

$$\begin{aligned}\log p(y|x) &= \log [\hat{y}^y (1 - \hat{y})^{1-y}] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y})\end{aligned}$$

Deriving the objective for a single observation x

Goal: maximize likelihood

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Now take the log of both sides (mathematically handy)

Maximize:

$$\begin{aligned}\log p(y|x) &= \log [\hat{y}^y (1 - \hat{y})^{1-y}] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y})\end{aligned}$$

Whatever values maximize $\log p(y|x)$ will also maximize $p(y|x)$

Deriving the objective for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Maximize: $\log p(y|x) = \log [\hat{y}^y (1 - \hat{y})^{1-y}]$
 $= y \log \hat{y} + (1 - y) \log(1 - \hat{y})$

Now flip sign to turn this into a loss: something to minimize

Deriving the objective for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Maximize: $\log p(y|x) = \log [\hat{y}^y (1 - \hat{y})^{1-y}]$
 $= y \log \hat{y} + (1 - y) \log(1 - \hat{y})$

Now flip sign to turn this into a loss: something to minimize

Minimize:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Deriving the objective for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

Maximize: $\log p(y|x) = \log [\hat{y}^y (1 - \hat{y})^{1-y}]$
 $= y \log \hat{y} + (1 - y) \log(1 - \hat{y})$

Now flip sign to turn this into a **cross-entropy loss**: something to minimize

Minimize:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Deriving cross-entropy loss for a single observation x

Goal: maximize probability of the correct label $p(y|x)$

$$\begin{aligned}\text{Maximize: } \log p(y|x) &= \log [\hat{y}^y (1 - \hat{y})^{1-y}] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y})\end{aligned}$$

Now flip sign to turn this into a **cross-entropy loss**: something to minimize

Minimize: $L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$

Or, plug in definition of $\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$


$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))]$$

Our goal: minimize the loss

Let's make explicit that the loss function is parameterized by weights $\theta=(w,b)$

- And we'll represent \hat{y} as $f(x; \theta)$ to make the dependence on θ more obvious

We want the weights that minimize the loss, averaged over all examples:

$$\hat{\theta} = \operatorname{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^m L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)})$$


$L_{\text{CE}}(\hat{y}, y)$

Let's see if this works for our sentiment example

We want loss to be:

- smaller if the model estimate \hat{y} is close to correct
- bigger if model is confused

Let's first suppose the true label of this is $y=1$ (positive)

It's hokey . There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable ? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Let's see if this works for our sentiment example

True value is $y=1$ (positive). How well is our model doing?

$$\begin{aligned} p(+|x) = P(Y = 1|x) &= \sigma(w \cdot x + b) \\ &= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\ &= \sigma(.833) \\ &= 0.70 \end{aligned}$$

Pretty well!

Let's see if this works for our sentiment example

True value is $y=1$ (positive). How well is our model doing?

$$\begin{aligned} p(+|x) = P(Y = 1|x) &= \sigma(w \cdot x + b) \\ &= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\ &= \sigma(.833) \\ &= 0.70 \end{aligned}$$

Pretty well! What's the loss?

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log \sigma(\mathbf{w} \cdot \mathbf{x} + b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

Let's see if this works for our sentiment example

Suppose the true value instead was $y=0$ (negative).

$$\begin{aligned} p(-|x) = P(Y = 0|x) &= 1 - \sigma(w \cdot x + b) \\ &= 0.30 \end{aligned}$$

Let's see if this works for our sentiment example

Suppose the true value instead was $y=0$ (negative).

$$\begin{aligned} p(-|x) = P(Y = 0|x) &= 1 - \sigma(w \cdot x + b) \\ &= 0.30 \end{aligned}$$

What's the loss?

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -\log (.30) \\ &= 1.2 \end{aligned}$$

Let's see if this works for our sentiment example

The loss when the model was right (if true $y=1$)

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log \sigma(\mathbf{w} \cdot \mathbf{x} + b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

The loss when the model was wrong (if true $y=0$)

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -\log(.30) \\ &= 1.2 \end{aligned}$$

Sure enough, loss was bigger when model was wrong!

Learning components

A loss function:

- **cross-entropy loss**

An optimization algorithm:

- **gradient descent**

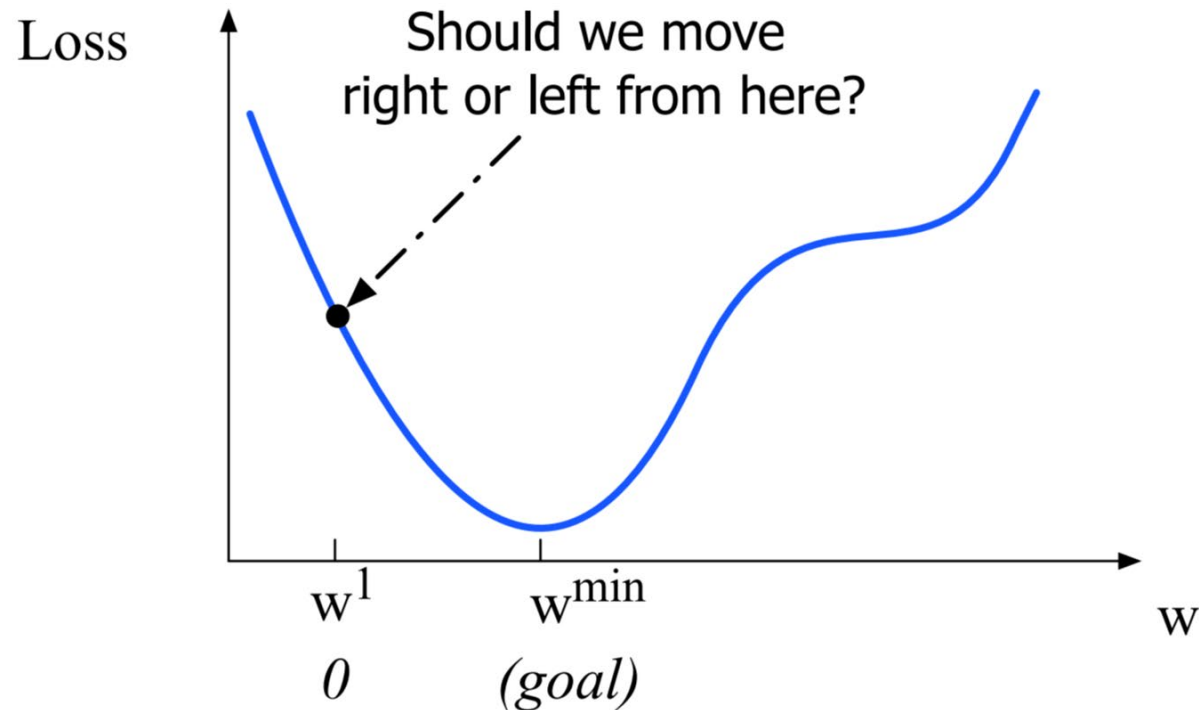
Gradient Descent

- Gradient Descent algorithm
 - is used to optimize the weights for a machine learning model

Let's first visualize for a single scalar w

Q: Given current w , should we make it bigger or smaller?

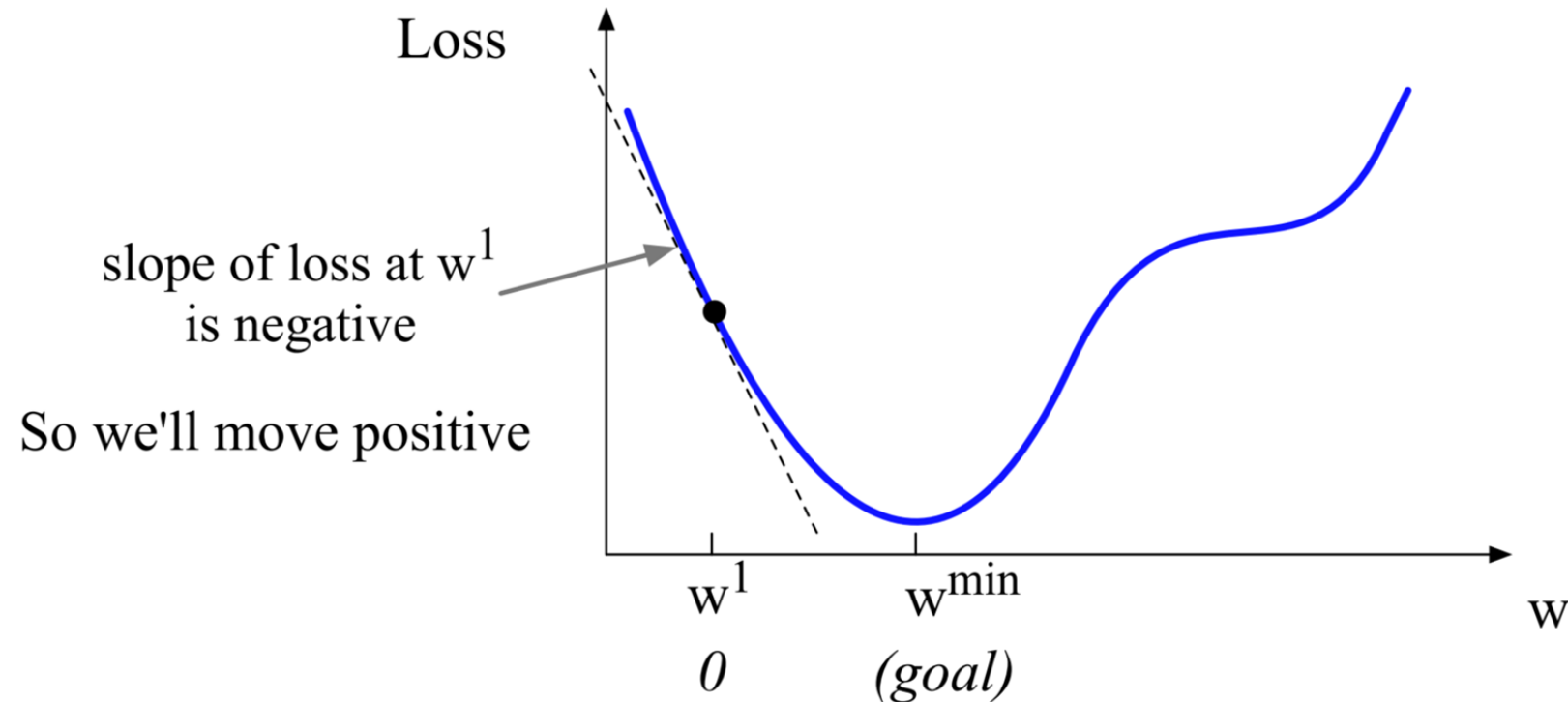
A: Move w in the reverse direction from the slope of the function



Let's first visualize for a single scalar w

Q: Given current w , should we make it bigger or smaller?

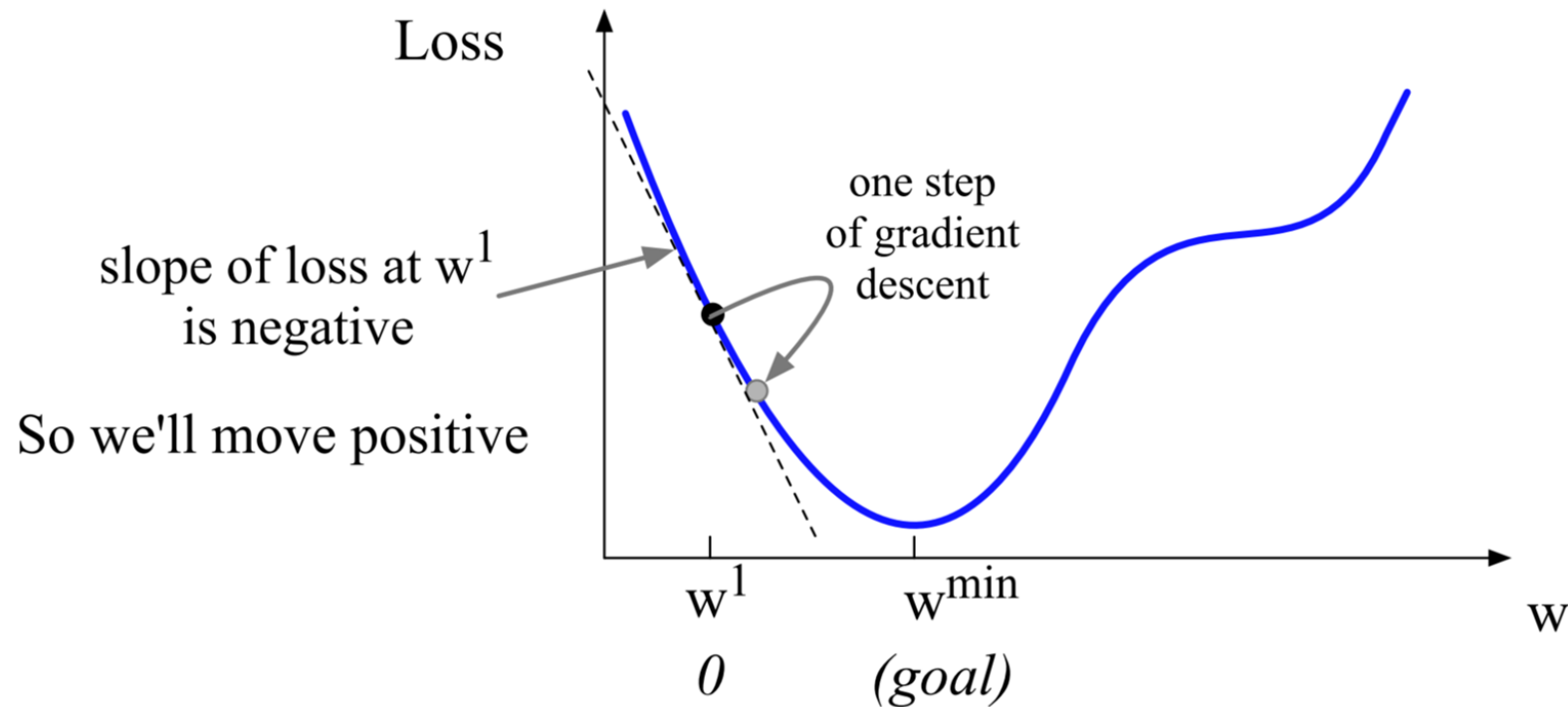
A: Move w in the reverse direction from the slope of the function



Let's first visualize for a single scalar w

Q: Given current w , should we make it bigger or smaller?

A: Move w in the reverse direction from the slope of the function



Our goal: minimize the loss

For logistic regression, loss function is **convex**

- A convex function has just one minimum
- Gradient descent starting from any point is guaranteed to find the minimum
 - (Loss for neural networks is non-convex)

Gradients

The **gradient** of a function
is a vector pointing in the direction of the greatest increase
in a function.

Gradient Descent: Find the gradient of the loss function at the
current point and move in the **opposite** direction.

How much do we move in that direction?

- The value of the gradient (slope in our example) $\frac{d}{dw}L(f(x;w),y)$
 - weighted by a learning rate η
- Higher learning rate means move w faster

$$w^{t+1} = w^t - \eta \frac{d}{dw}L(f(x;w),y)$$

Now let's consider N dimensions

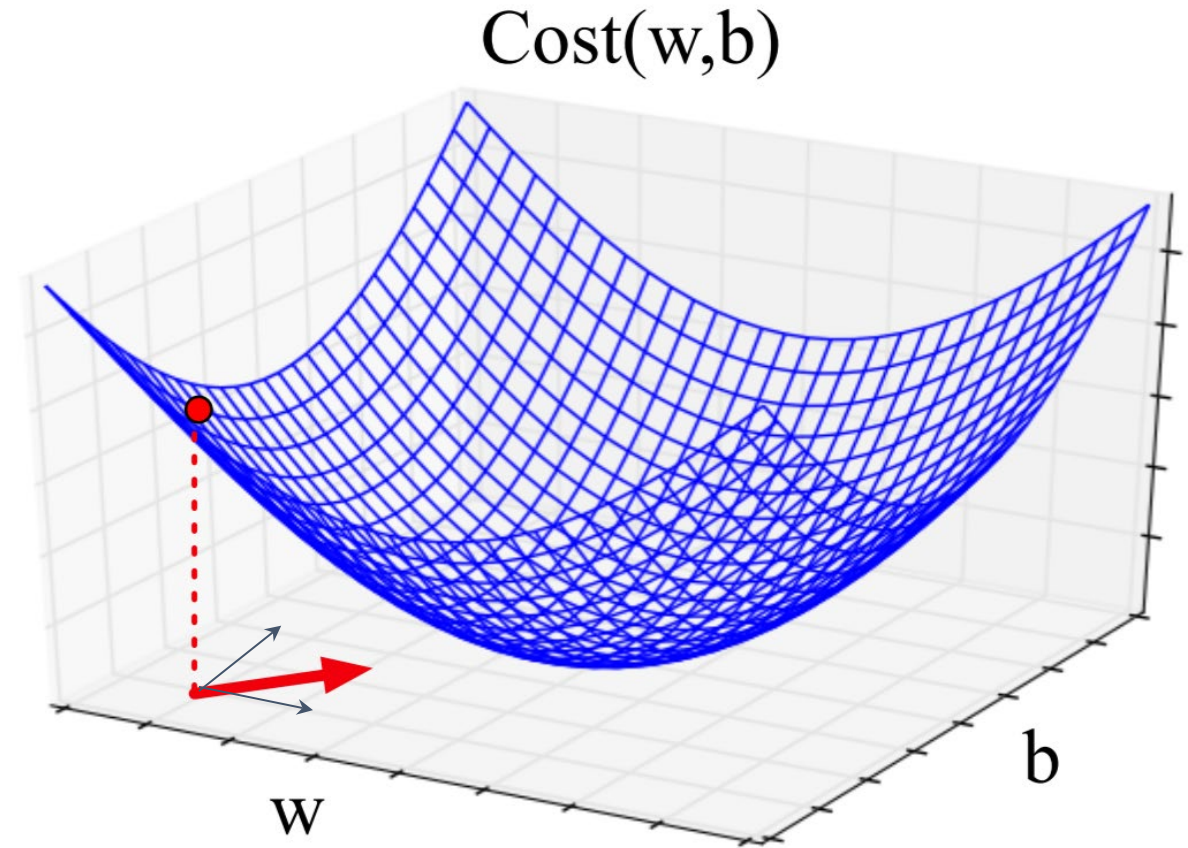
We want to know where in the N -dimensional space (of the N parameters that make up θ) we should move.

The gradient is just such a vector; it expresses the directional components of the sharpest slope along each of the N dimensions.

Imagine 2 dimensions, w and b

Visualizing the gradient vector
at the red point

It has two dimensions shown
in the x - y plane



Real gradients

Are much longer; lots and lots of weights

For each dimension w_i the gradient component i tells us the slope with respect to that variable.

- “How much would a small change in w_i influence the total loss function L ?”
- We express the slope as a partial derivative ∂

The gradient is then defined as a vector of these partials.

The gradient

We'll represent \hat{y} as $f(\mathbf{x}; \theta)$ to make the dependence on θ more obvious:

$$\nabla_{\theta} L(f(x; \theta), y) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x; \theta), y) \\ \frac{\partial}{\partial w_2} L(f(x; \theta), y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x; \theta), y) \end{bmatrix}$$

The final equation for updating θ based on the gradient is thus:

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x; \theta), y)$$

What are these partial derivatives for logistic regression?

The loss function

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log (1 - \sigma(w \cdot x + b))]$$

The elegant derivative of this function

$$\begin{aligned} \frac{\partial L_{\text{CE}}(\hat{y}, y)}{\partial w_j} &= [\sigma(w \cdot x + b) - y]x_j \\ &= (\hat{y} - y)\mathbf{x}_j \end{aligned}$$

Gradient Descent Algorithm: Summary

1. Given a dataset $D = [(x, y)]$, a model with weights $w = (w_1, \dots, w_n)$, and a loss function $L(D, w)$.
2. Initialize w randomly
3. Compute gradient of L , ∇L
Update $w \leftarrow w - \eta \nabla L$
4. Repeat 3
 - until **convergence**

Hyperparameters

The learning rate η is a **hyperparameter**

- too high: the learner will take big steps and overshoot
- too low: the learner will take too long

Hyperparameters:

- Briefly, a special kind of parameter for an ML model
- Instead of being learned by algorithm from supervision (like regular parameters), they are chosen by algorithm designer.

Mini-batch training

Gradient descent computes the loss over the entire dataset. That can be slow.

$$\hat{\theta} = \operatorname{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^m L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

More common to compute gradient over batches of training instances.

Mini-batch training: m random examples at every gradient step.
Also known as stochastic gradient descent

Overfitting

A model that perfectly match the training data has a problem.

It will also **overfit** to the data, modeling noise

- A random word that perfectly predicts y (it happens to only occur in one class) will get a very high weight.
- Failing to generalize to a test set without this word.

A good model should be able to **generalize**

Regularization

A solution for overfitting

Add a **regularization** term $R(\theta)$ to the loss function

$$\hat{\theta} = \operatorname{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^m L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)}) + \alpha R(\theta)$$

Idea: choose an $R(\theta)$ that penalizes large weights

- fitting the data well with lots of big weights not as good as fitting the data a little less well, with small weights

L2 regularization (ridge regression)

The sum of the squares of the weights

$$R(\theta) = \|\theta\|_2^2 = \sum_{j=1}^n \theta_j^2$$

L2 regularized objective function:

$$\hat{\theta} = \operatorname{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^m L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)}) + \alpha \sum_{j=1}^n \theta_j^2$$

Multinomial Logistic Regression

Classification into more than 2 classes.

If >2 classes we use **multinomial logistic regression**

= Softmax regression

= Multinomial logit

= (defunct names : Maximum entropy modeling or MaxEnt)

Multinomial Logistic Regression

In binary classification, we have a set of weights w , one corresponding to each feature.

In N-class classification, we define a set of weights for each class, w_y : n weights for each feature

Multinomial Logistic Regression

- Binary: convert the features to a score (real number), and apply sigmoid
 - Score: $z = w \cdot x + b$
 - Probability of $y=1$: $\sigma(z)$
- N-class: convert the features in N scores (also called logits):
 - Scores: $[w_1 \cdot x + b, w_2 \cdot x + b, \dots] = [z_1, z_2, \dots]$
 - Convert to probabilities using a “softmax” function – N-dimensional generalization of sigmoid.

Sigmoid \rightarrow softmax

$$\frac{1}{1+e^{-z}} \longrightarrow \frac{e^{z_j}}{\sum_{c=1}^k e^{z_c}}$$

Softmax gives you a vector (whose values sum up to 1)

The softmax function

- Turns a vector $\mathbf{z} = [z_1, z_2, \dots, z_k]$ of k arbitrary values (logits) into probabilities

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^k \exp(z_j)} \quad 1 \leq i \leq k$$

$$\sum_{i=1}^k e^{z_i}$$

- The
into

$$\text{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^k \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^k \exp(z_i)}, \dots, \frac{\exp(z_k)}{\sum_{i=1}^k \exp(z_i)} \right]$$

Summary

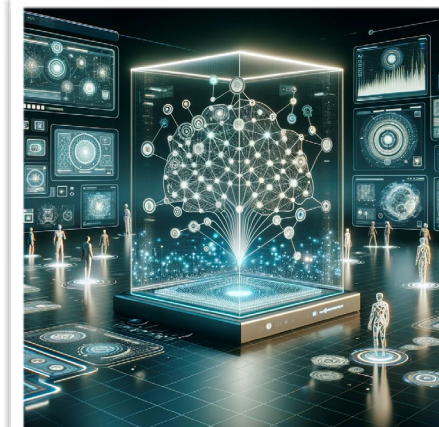
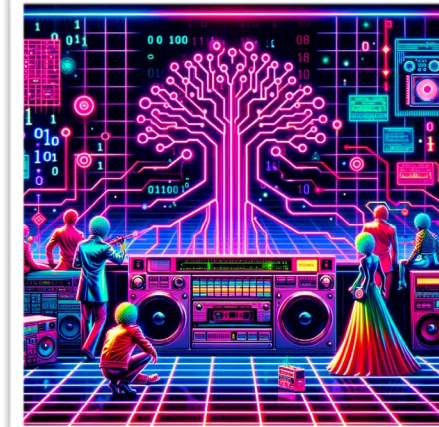
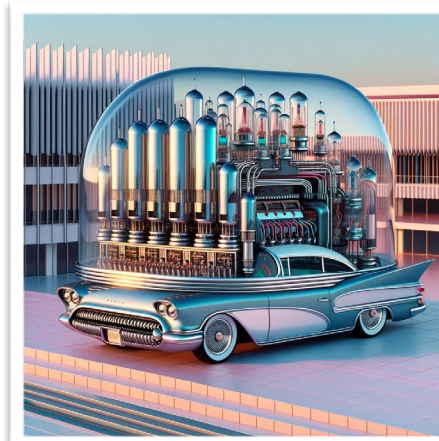
- Loss function to train a logistic regression classifier: Cross Entropy
- Optimization algorithm to find the weights of the classifier
 - Gradient descent (or stochastic gradient descent)
 - Can lead to overfitting, adding regularization helps.
- Logistic regression can be extended to multiple classes by
 - Creating weights for each class
 - using a softmax function instead of sigmoid.

Neural Nets Basics

Neural Networks

A Little Bit of History

- Neural network algorithms date to the 1980s, and design trace their origin to the 1950s
 - Originally inspired by early neuroscience
- Historically slow, complex, and unwieldy
- Now: term is abstract enough to encompass almost any model – but useful!
- Dramatic shift started around 2013-15 away from linear, convex (like logistic regression) to *neural networks* (non-linear architecture, non-convex)



Why neural networks?

- Linear models like logistic regression require hand-designing features.
 - Requires knowledge of the task, domain, language.
 - Time consuming
- Linear models assume the classes are linearly separable given the features.

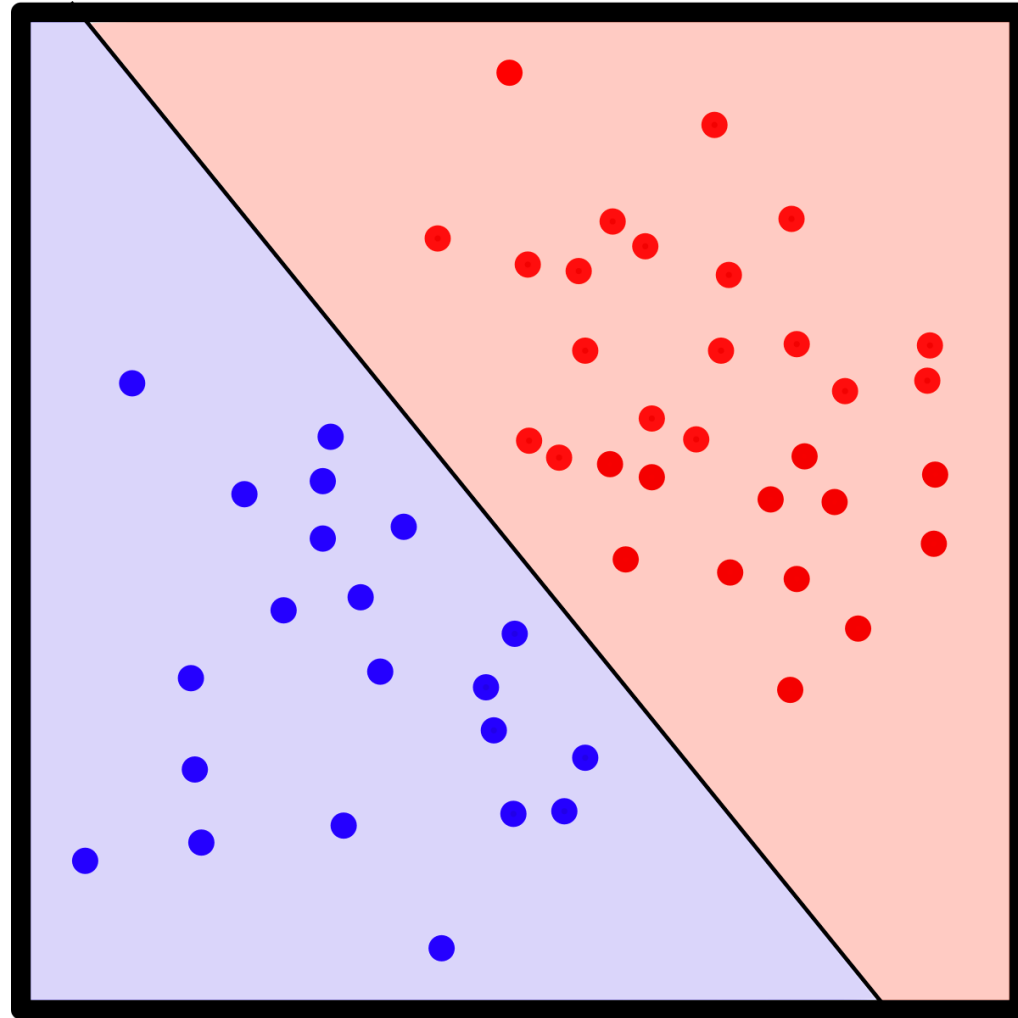
Neural Networks

The Promise

- Non-neural ML works well because of human-designed representations and input features
- ML becomes just optimizing weights
- **Representation learning** attempts to automatically learn good features and representations
- **Deep learning** attempts to learn multiple levels of representation of increasing complexity/abstraction

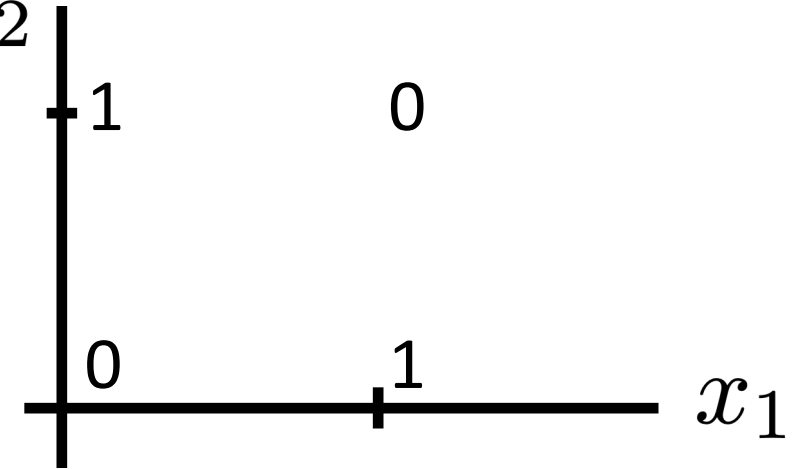


Linear models assume separability



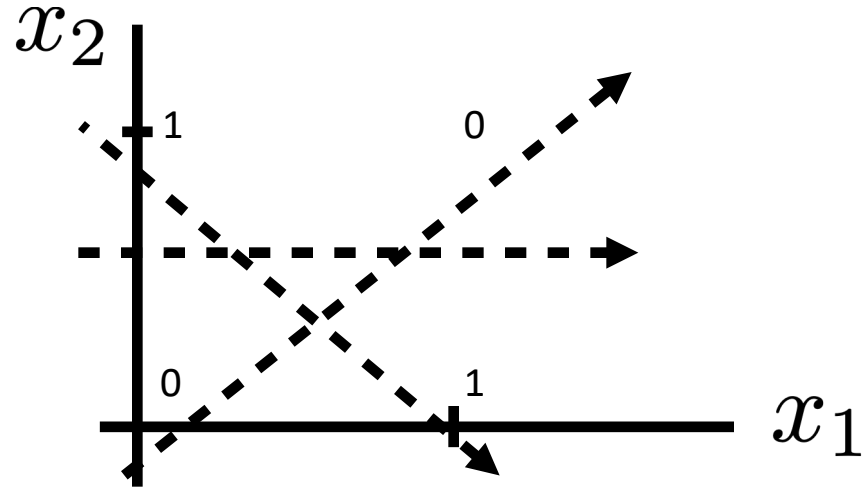
Neural Networks: XOR

- Let's see how we can use neural nets x_2 to learn a simple nonlinear function
- ▶ Inputs x_1, x_2
- ▶ Output y



x_1	x_2	$y = x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



$$y = a_1x_1 + a_2x_2$$

X

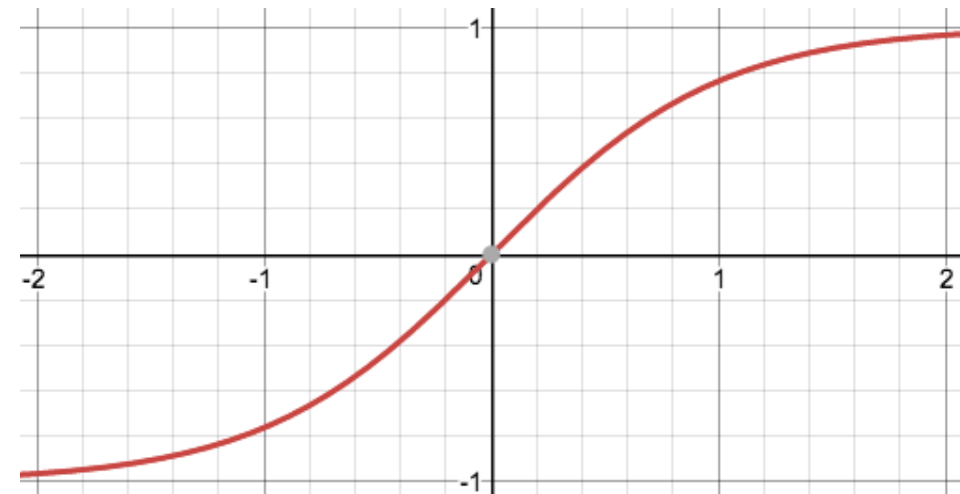
$$y = a_1x_1 + a_2x_2 + a_3 \tanh(x_1 + x_2)$$

✓

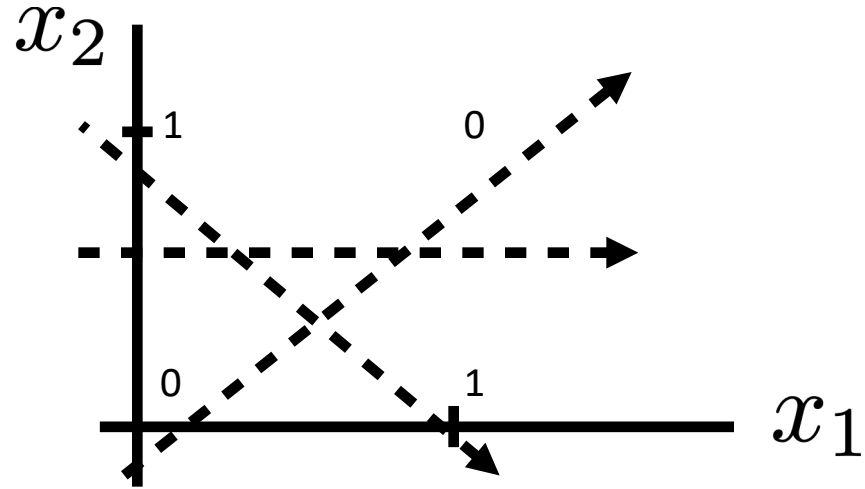
"or"

(looks like action
potential in neuron)

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



Neural Networks: XOR



x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

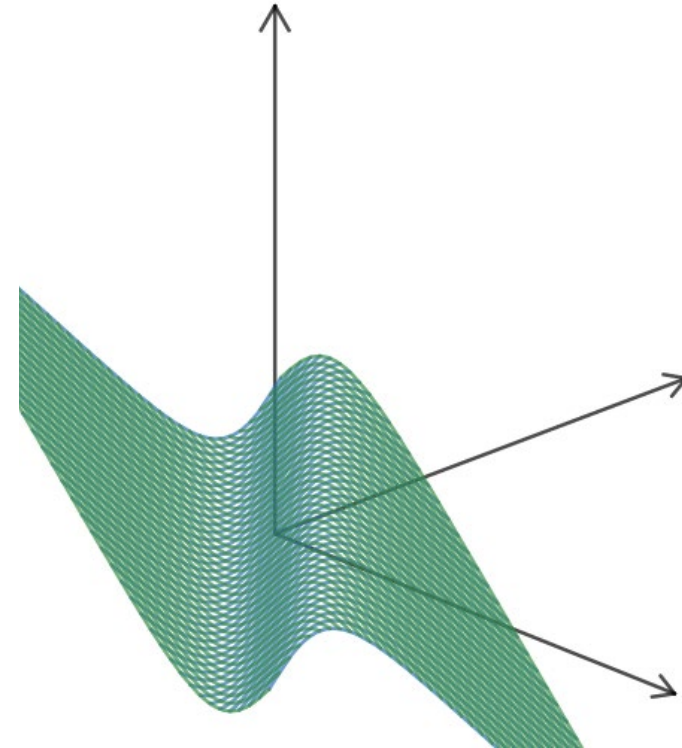
$$y = a_1 x_1 + a_2 x_2$$

✗

$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

✓

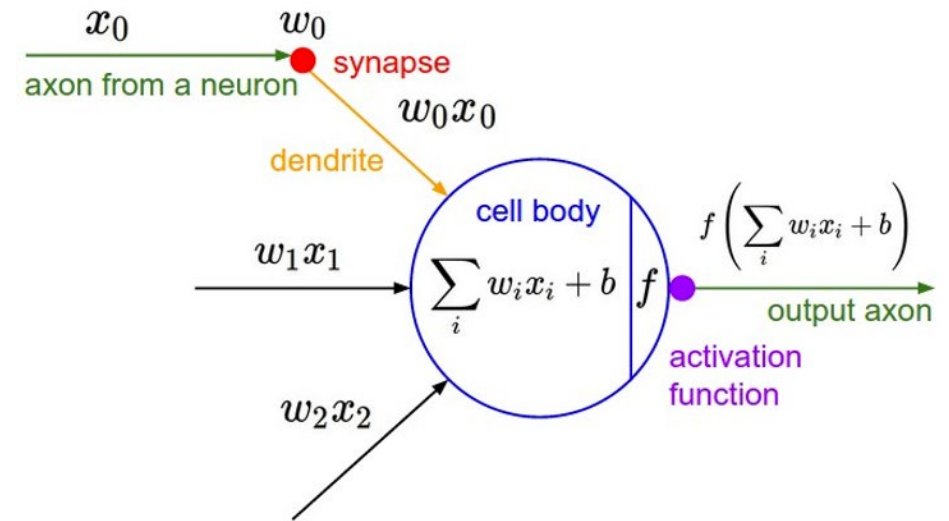
$$y = -x_1 - x_2 + 2 \tanh(x_1 + x_2)$$



Building Blocks

The Neuron

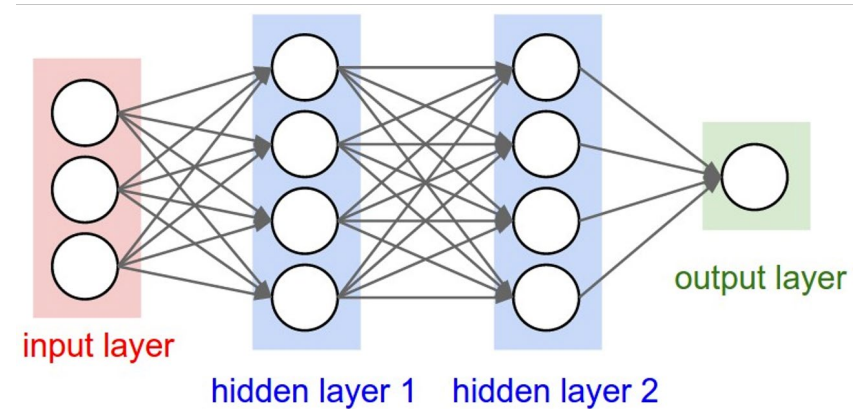
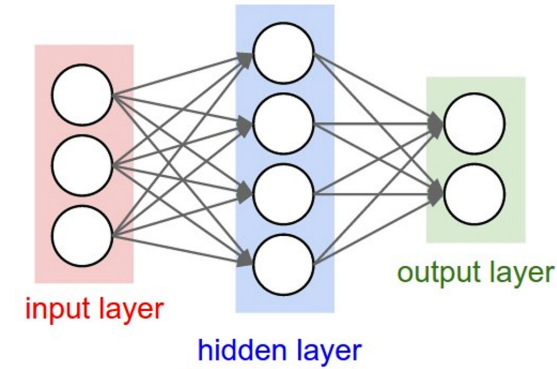
- Neural networks traditionally come with their own terminology baggage
 - Some of it is less common in more recent work
- Parameters:
 - Inputs: x_i
 - Weights: w_i and b
 - Activation function f
- If we drop the activation function, reminds you of something?



Building Blocks

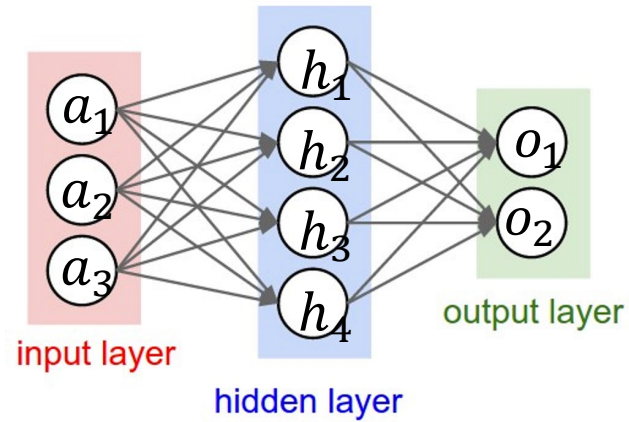
Hidden Layers

- It gets interesting when you connect and stack neurons
- This modularity is one of the greatest strengths of neural networks
- Input vs. hidden vs. output layers
- The activations of the hidden layers are the learned representation



Building Blocks

Matrix Notation

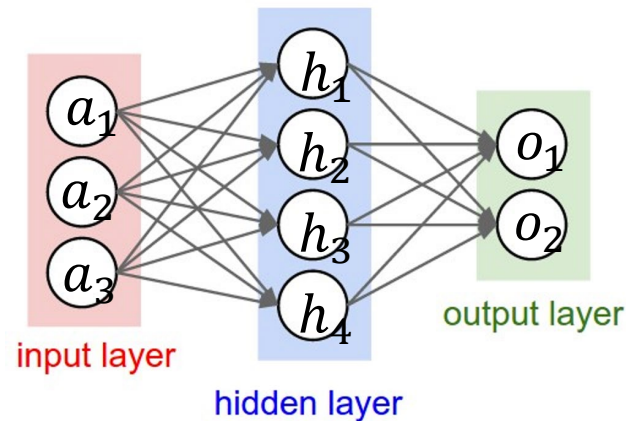


No activation/non-linearity function

Building Blocks

Matrix Notation

$$\begin{aligned}h_1 &= a_1W'_{11} + a_2W'_{21} + a_3W'_{31} + b'_1 \\h_2 &= a_1W'_{12} + a_2W'_{22} + a_3W'_{32} + b'_1 \\h_3 &= a_1W'_{13} + a_2W'_{23} + a_3W'_{33} + b'_1 \\h_4 &= a_1W'_{14} + a_2W'_{24} + a_3W'_{34} + b'_4\end{aligned}$$



$$\mathbf{h}_{4 \times 1} = \mathbf{W}'_{4 \times 3} \mathbf{a}_{3 \times 1} + \mathbf{b}'_{4 \times 1}$$

$$\mathbf{o}_{2 \times 1} = \mathbf{W}''_{2 \times 4} \mathbf{h}_{4 \times 1} + \mathbf{b}''_{2 \times 1}$$

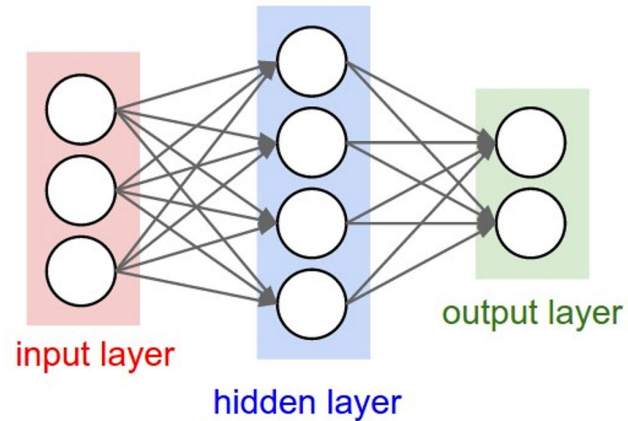
$$\begin{aligned}o_1 &= h_1W''_{11} + h_2W''_{21} + h_3W''_{31} + h_4W''_{41} + b''_1 \\o_2 &= h_1W''_{12} + h_2W''_{22} + h_3W''_{32} + h_4W''_{42} + b''_2\end{aligned}$$

Building Blocks

Activation Functions

Activation (non-linearity) function is an entry-wise function

$$f: \mathbb{R} \rightarrow \mathbb{R}$$



$$\begin{aligned}h_1 &= a_1 W'_{11} + a_2 W'_{21} + a_3 W'_{31} + b'_1 \\h_2 &= a_1 W'_{12} + a_2 W'_{22} + a_3 W'_{32} + b'_1 \\h_3 &= a_1 W'_{13} + a_2 W'_{23} + a_3 W'_{33} + b'_1 \\h_4 &= a_1 W'_{14} + a_2 W'_{24} + a_3 W'_{34} + b'_4\end{aligned}$$

$$\mathbf{h}_{4 \times 1} = f(\mathbf{W}'_{4 \times 3} \mathbf{a}_{3 \times 1} + \mathbf{b}'_{4 \times 1})$$

$$\mathbf{o}_{2 \times 1} = \mathbf{W}''_{2 \times 4} \mathbf{h}_{4 \times 1} + \mathbf{b}''_{2 \times 1}$$

Building Blocks

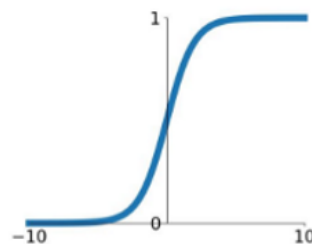
Activation Functions

Activation (non-linearity) function is an entry-wise function

$$f: \mathbb{R} \rightarrow \mathbb{R}$$

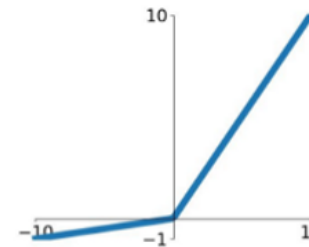
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



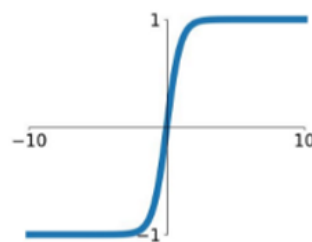
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

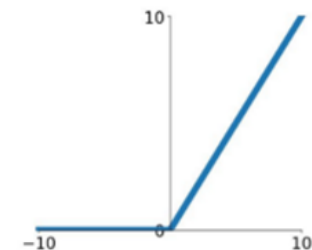


Maxout

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

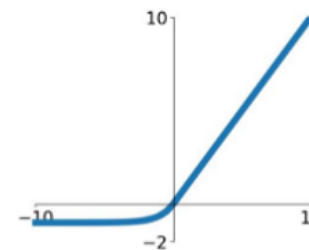
ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Why activation functions?

- What if we do not have activation functions

$$\mathbf{o} = \mathbf{W}''\mathbf{h} + \mathbf{b}''$$

$$\mathbf{o} = \mathbf{W}''(\mathbf{W}'\mathbf{a} + \mathbf{b}') + \mathbf{b}''$$

$$\mathbf{o} = \mathbf{W}''\mathbf{W}'\mathbf{a} + \mathbf{W}''\mathbf{b}' + \mathbf{b}''$$

Define $\mathbf{W}''' = \mathbf{W}''\mathbf{W}'$ and $\mathbf{b}''' = \mathbf{W}''\mathbf{b}' + \mathbf{b}''$

A multi-layer linear network is the same as a 1-layer network (with some caveats)

Deep Neural Networks

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\mathbf{z} = g(\mathbf{V}\mathbf{y} + \mathbf{c})$$

$$\mathbf{z} = g(\mathbf{V} \underbrace{g(\mathbf{W}\mathbf{x} + \mathbf{b})}_{\text{output of first layer}} + \mathbf{c})$$

Building Blocks of Neural NLP

One-hot Word Representations

- Neural networks take continuous vector inputs
- How can we represent text as continuous vectors?
- One-hot vectors

$$\begin{aligned} hotel &= [0 \quad 0 \quad 0 \quad \dots \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0] \\ conference &= [0 \quad 0 \quad 0 \quad \dots \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0] \end{aligned}$$

- Dimensionality: size of the vocabulary
 - Can be >10M for web-scale corpora
- Problems?

Building Blocks for Neural NLP

One-hot Word Representations

- One-hot vectors

$$\begin{aligned} \textit{hotel} &= [0 \quad 0 \quad 0 \quad \dots 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0] \\ \textit{conference} &= [0 \quad 0 \quad 0 \quad \dots 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0] \end{aligned}$$

- Problems?
 - Information sharing? “hotel” vs. “hotels”

Building Blocks

Word Embeddings

- Each word is represented using a dense low-dimensional vector
 - Low-dimensional \ll vocabulary size
- If trained well, similar words will have similar vectors
- How to train? What objective to maximize?
 - As part of task training (e.g., supervised training)
 - Pre-training (more on this later)

Training Neural Networks

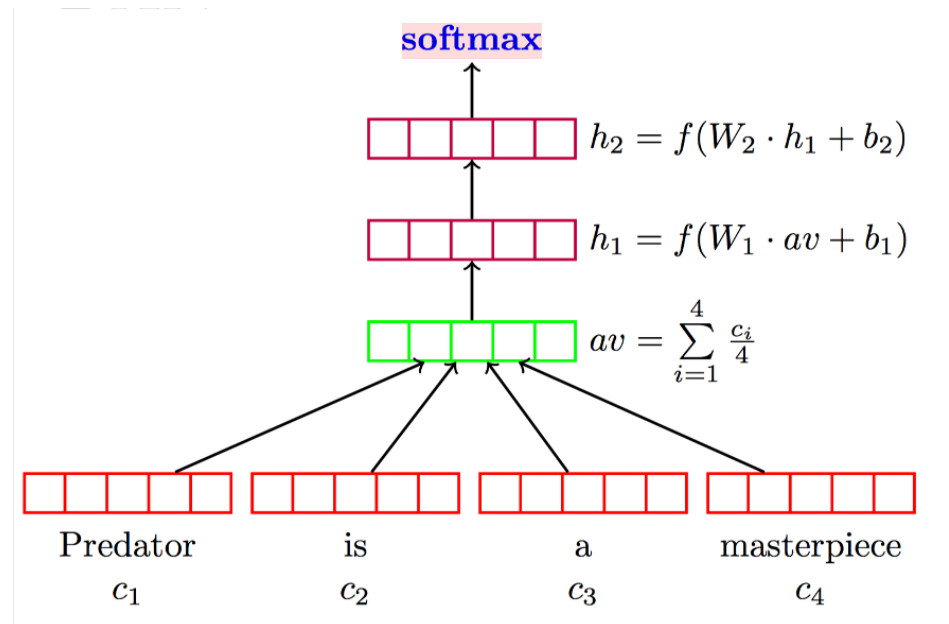
- No hidden layer → same as logistic regression (convex, guaranteed to converge)
- With hidden layers:
 - Latent units → not convex
 - What do we do?
 - Back-propagate the gradient
 - Based on the chain rule
 - About the same, but no guarantees

Neural Bag of Words

- One of the most basic neural models
- Example: sentiment classification
 - Input: text document
 - Classes: very positive, positive, neutral, negative, very negative
- We discussed doing this with a bag-of-words feature-based model
- What would be the neural equivalent?
 - Concatenate all vectors?
 - Problem: different documents → different input length
 - Instead: sum, average, etc.

Neural Bag of Words

Deep Averaging Networks (Iyyer et al. 2015)



IMDB Sentiment Analysis

BOW + linear model	88.23
NBOW DAN	89.4

Computation Graphs

- The descriptive language of deep learning models
- Functional description of the required computation
- Can be instantiated to do two types of computation:
 - Forward computation
 - Backward computation

expression:

x

graph:

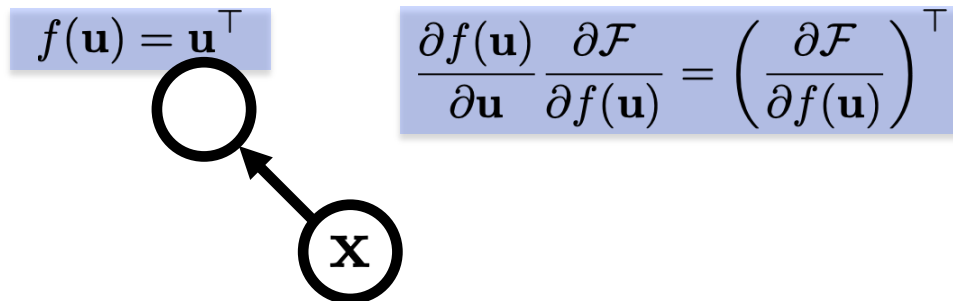
A **node** is a {tensor, matrix, vector, scalar} value

x

An **edge** represents a function argument (and also data dependency). They are just pointers to nodes.

A **node** with an incoming **edge** is a **function** of that edge's tail node.

A **node** knows how to compute its value and the *value of its derivative w.r.t each argument (edge) times a derivative of an arbitrary input* $\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})}$.

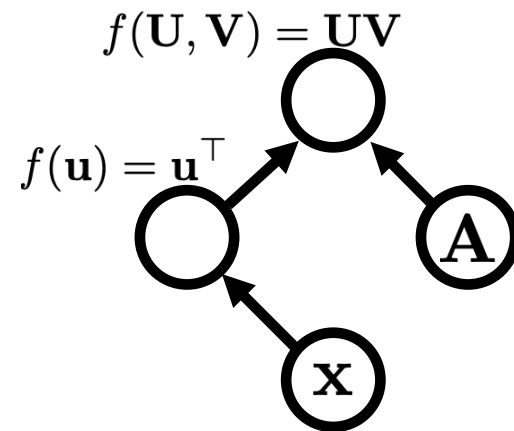


expression:

$$\mathbf{x}^\top \mathbf{A}$$

graph:

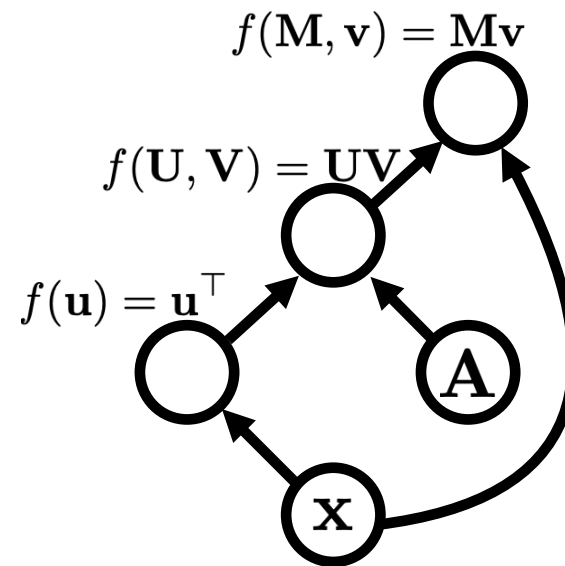
Functions can be nullary, unary,
binary, ... n -ary. Often they are unary or binary.



expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:

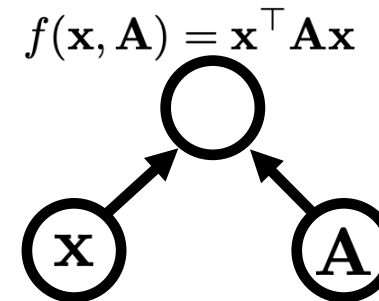
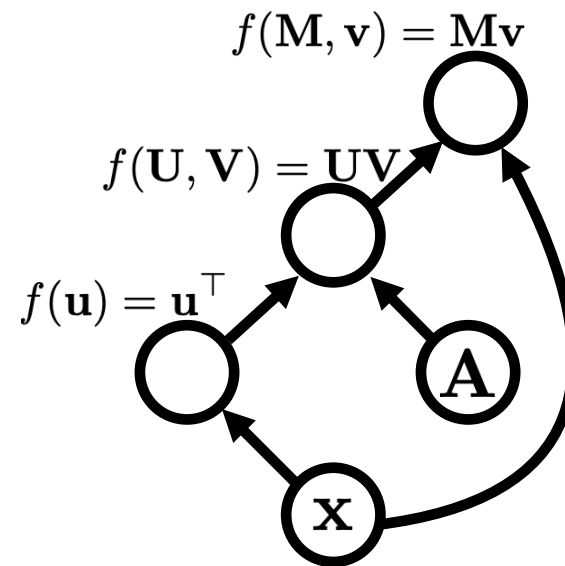


Computation graphs are directed and acyclic (usually)

expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:



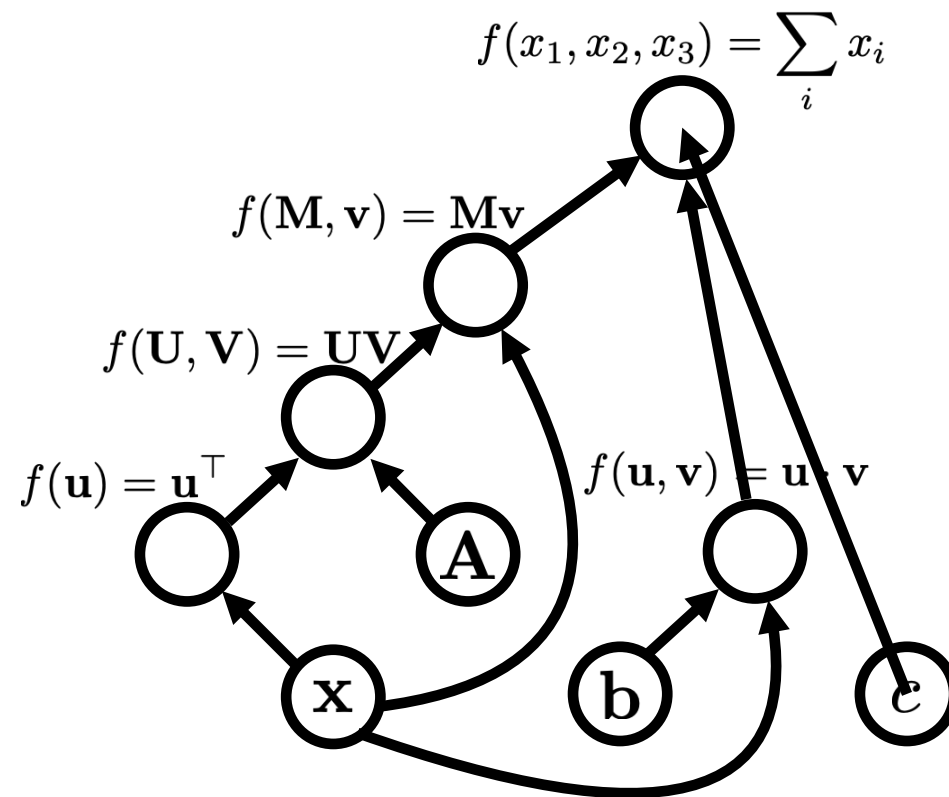
$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{x}} = (\mathbf{A}^\top + \mathbf{A})\mathbf{x}$$

$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{A}} = \mathbf{x}\mathbf{x}^\top$$

expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

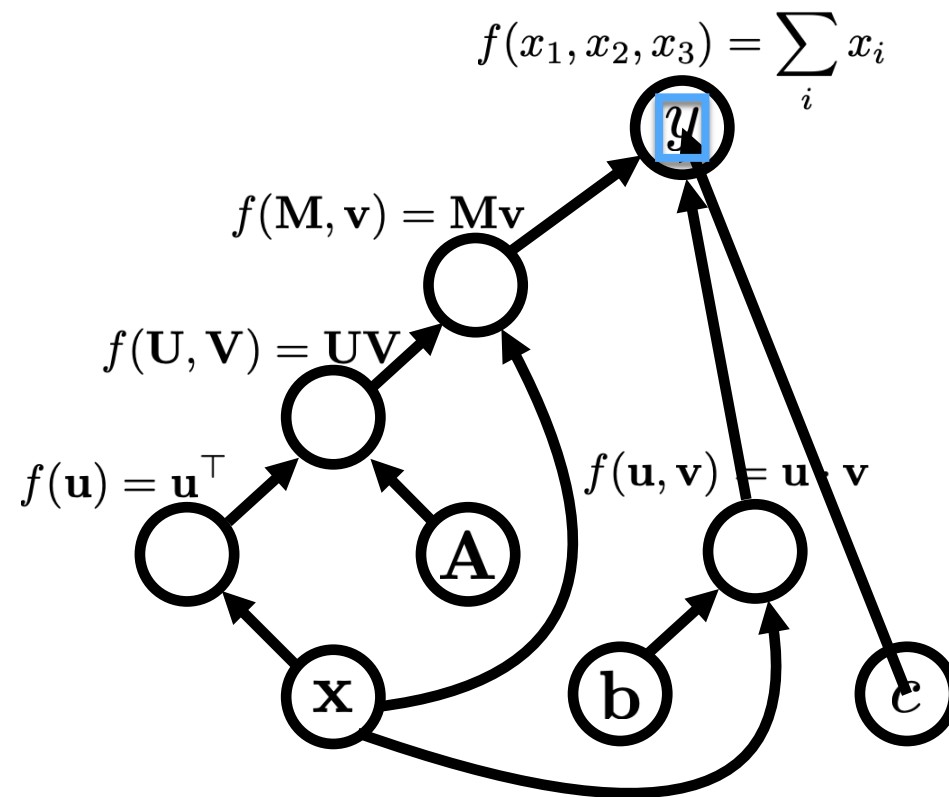
graph:



expression:

$$y = \mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:



variable names are just labelings of nodes.

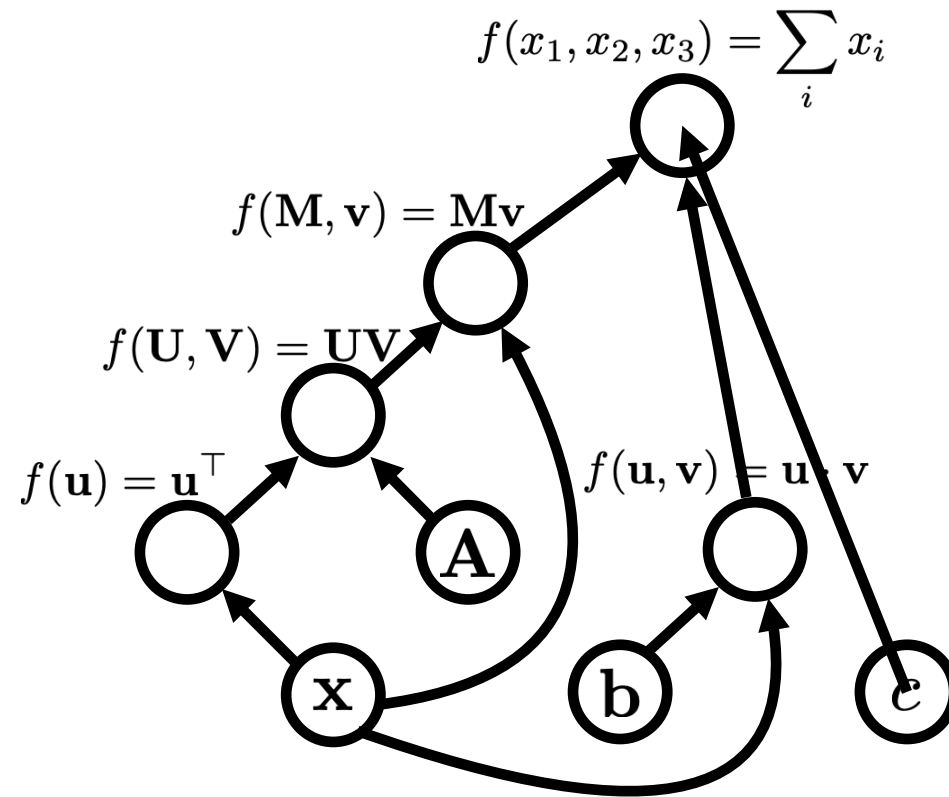
Computation Graphs

Algorithms

- Graph construction
- Forward propagation
 - Loop over nodes in topological order
 - Compute the value of the node given its inputs
 - *Given my inputs, make a prediction (or compute an "error" with respect to a "target output")*
- Backward propagation
 - Loop over the nodes in reverse topological order starting with a final goal node
 - Compute derivatives of final goal node value with respect to each edge's tail node
 - *How does the output change if I make a small change to the inputs?*

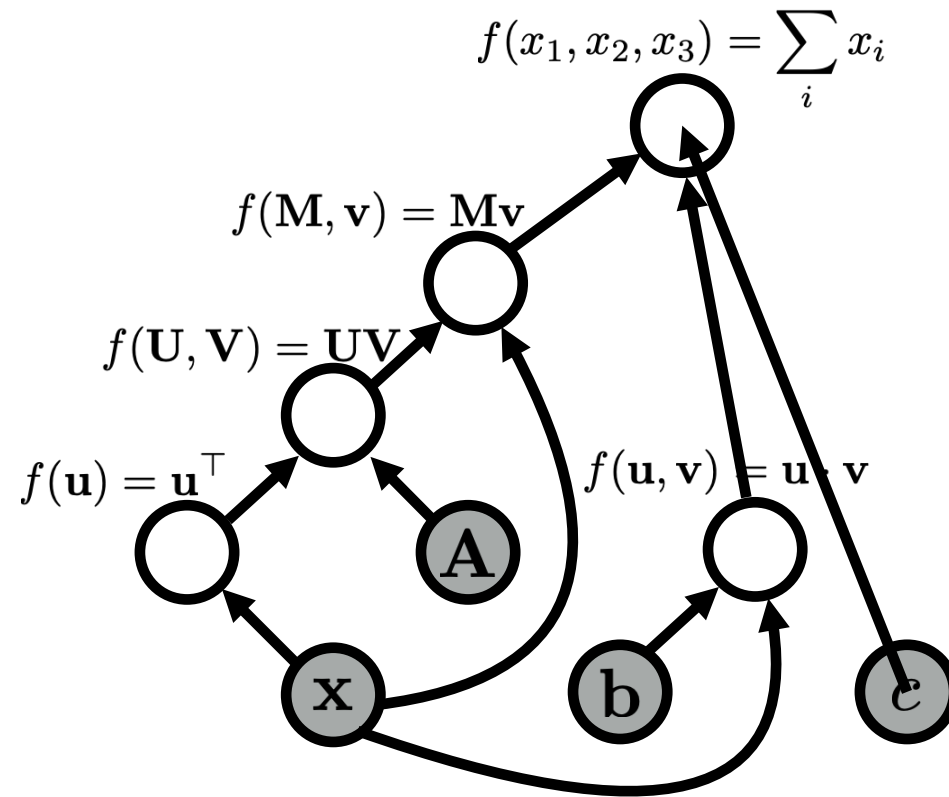
Forward Propagation

graph:



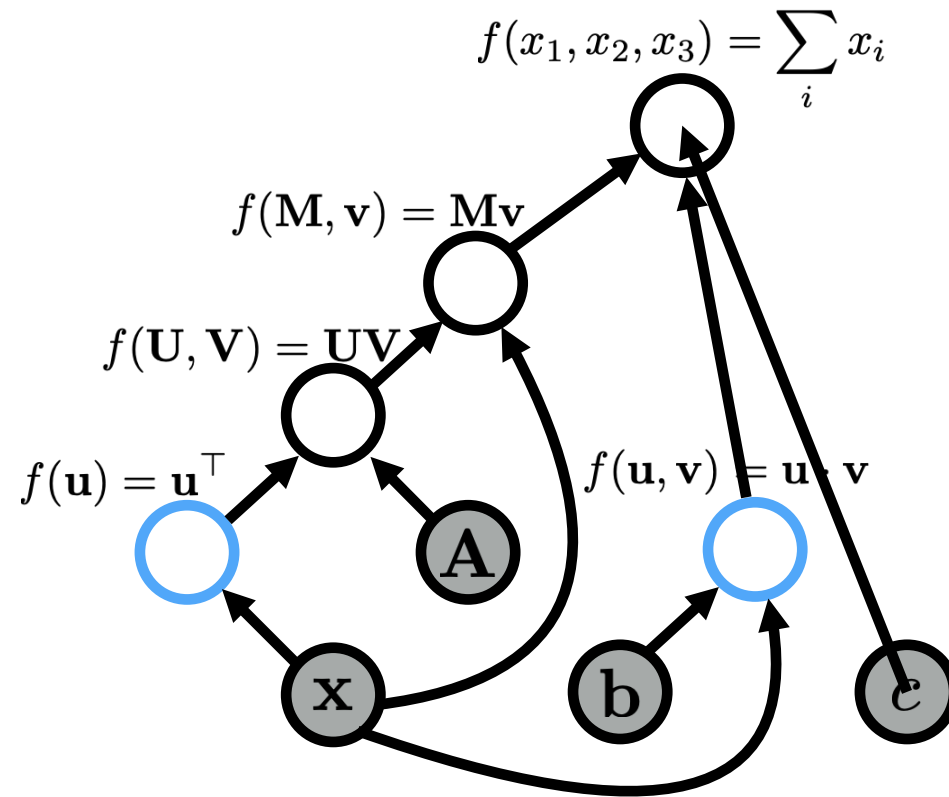
Forward Propagation

graph:



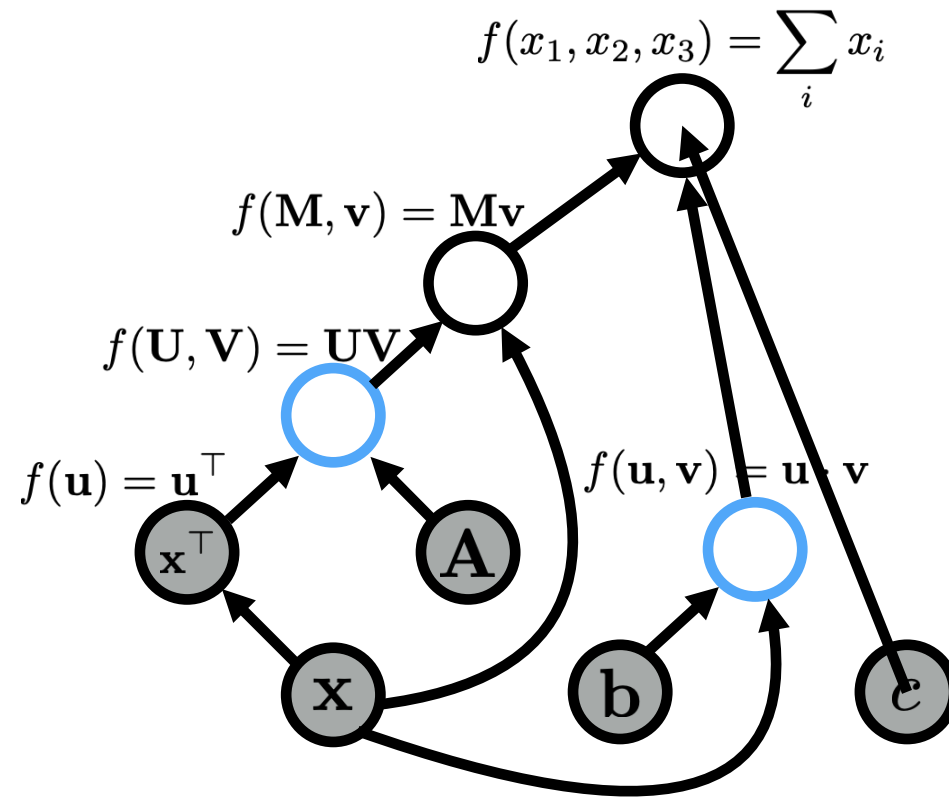
Forward Propagation

graph:



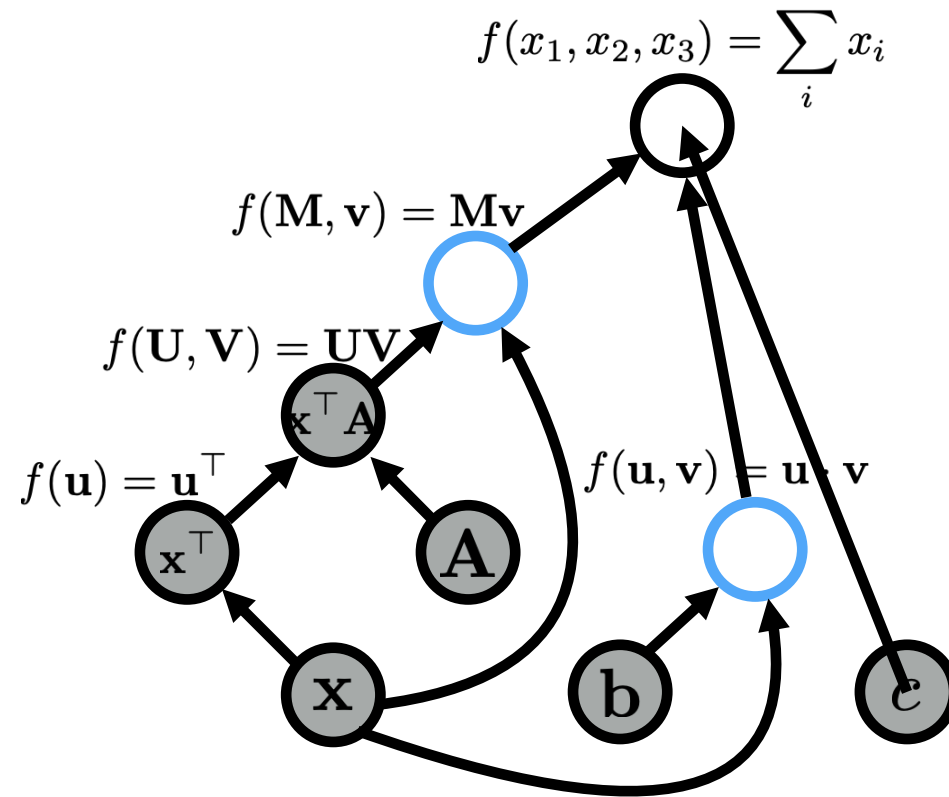
Forward Propagation

graph:



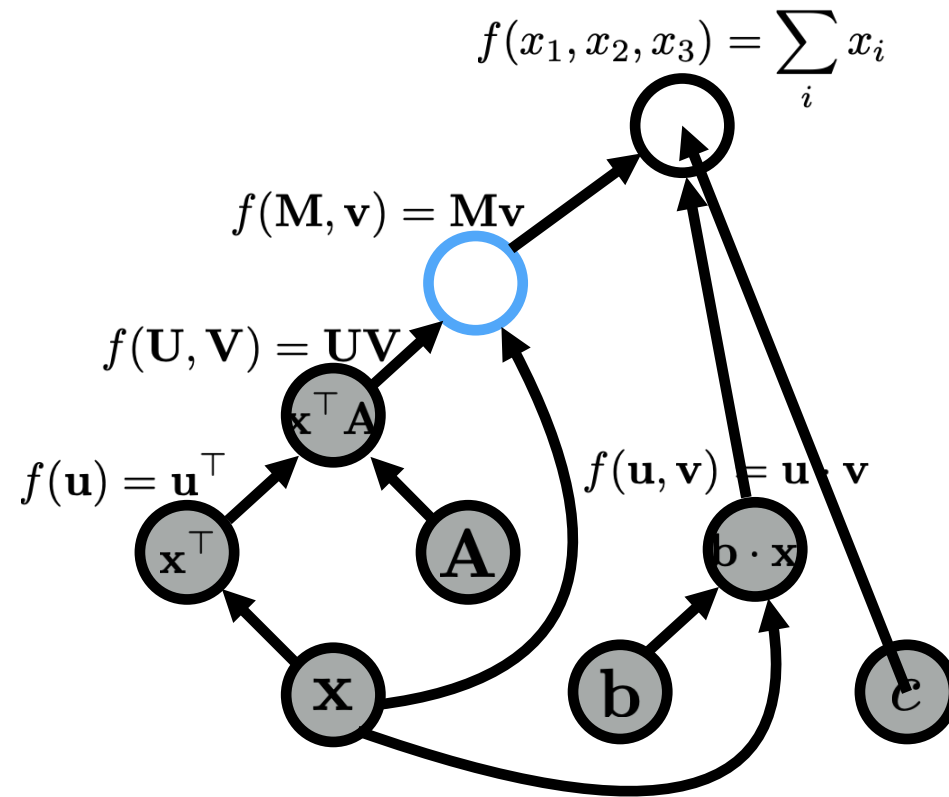
Forward Propagation

graph:



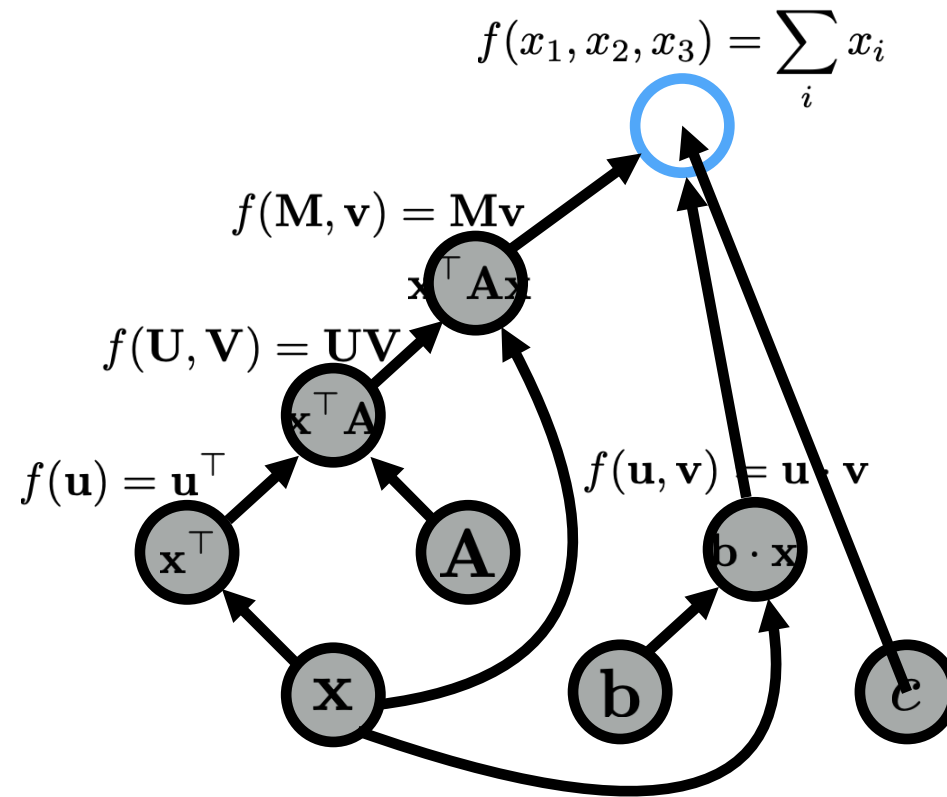
Forward Propagation

graph:



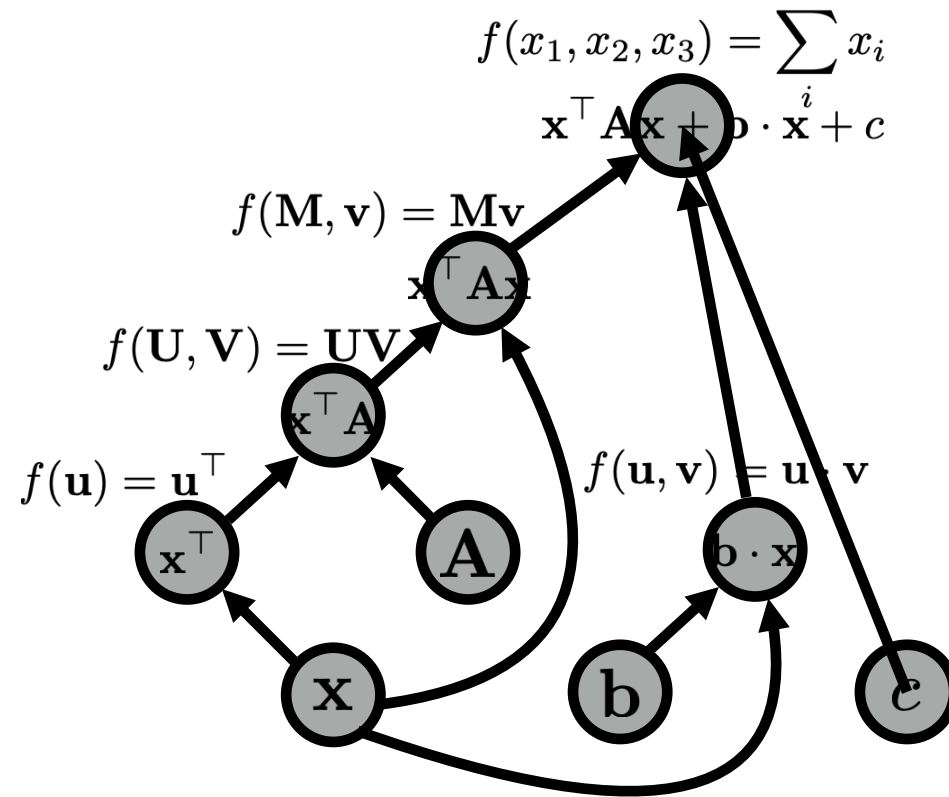
Forward Propagation

graph:



Forward Propagation

graph:



Constructing Graphs

Two Software Models

- Static declaration
 - Phase 1: define an architecture
(maybe with some primitive flow control like loops and conditionals)
 - Phase 2: run a bunch of data through it to train the model and/or make predictions
- Dynamic declaration (a.k.a define-by-run)
 - Graph is defined implicitly (e.g., using operator overloading) as the forward computation is executed
 - Graph is constructed dynamically
 - This allows incorporating conditionals and loops into the network definitions easily

Batching

- Two senses to processing your data in batch
 - Computing gradients for more than one example at a time to update parameters during learning
 - Processing examples together to utilize all available resources
- CPU: made of a small number of cores, so can handle some amount of work in parallel
- GPU: made of thousands of small cores, so can handle a lot of work in parallel
- Process multiple examples together to use all available cores

Batching

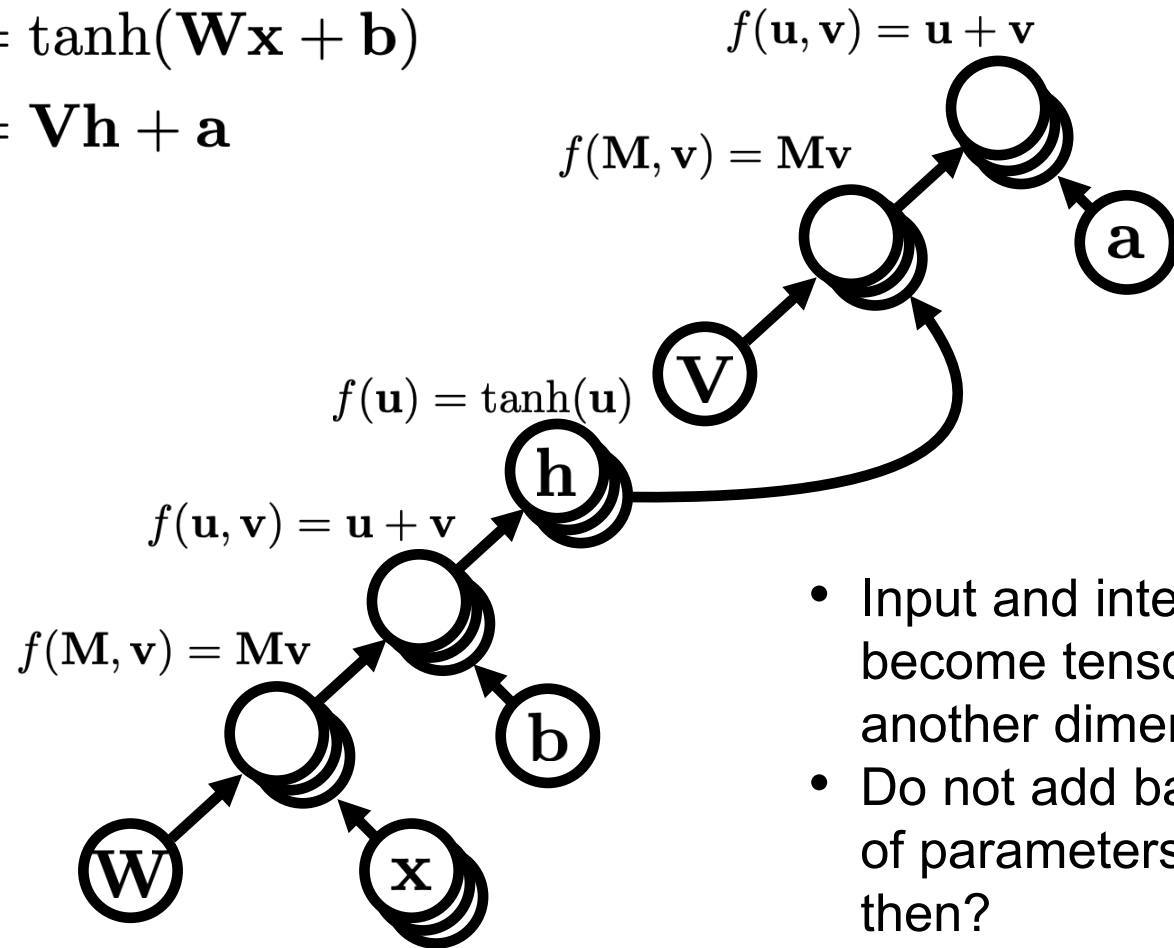
- Relatively easy when the network looks exactly the same for all examples
- More complex with language data: documents/sentences/words have different lengths
- Frameworks provide different methods to help common cases, but still require work on the developer side
- Key concept is broadcasting:
<https://pytorch.org/docs/stable/notes/broadcasting.html>

Batching

MLP Sketch

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



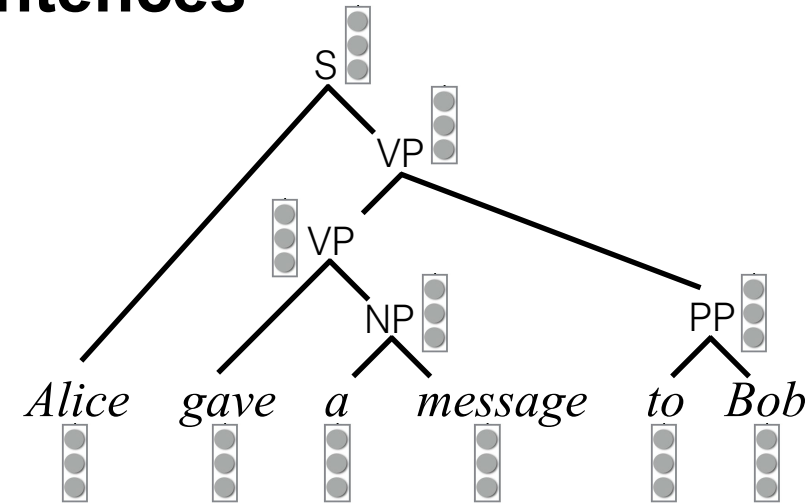
- Input and intermediate results become tensors — batch is another dimension!
- Do not add batch dimension of parameters! What happens then?

Batching

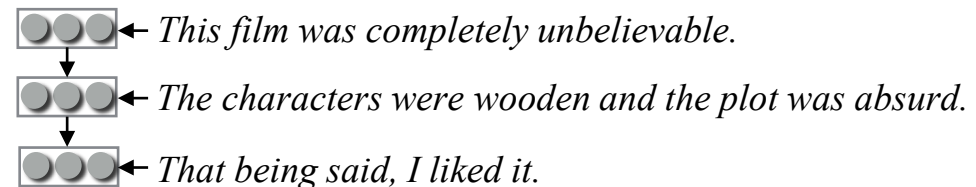
Complex Network Architectures

- Complex networks may include different parts with varying length (more about this later)
- In the extreme, it may be complex to batch complete examples this way
- But: you can still batch sub-parts across examples, so you alternate between batched and non-batched computations

Sentences



Documents



Backpropagation

But what about the gradient w.r.t. W_1 ?

Apply the chain rule

$$\frac{\partial \mathcal{L}(x, i^*)}{\partial W_{1i,j}} = \frac{\partial \mathcal{L}(x, i^*)}{\partial z} \cdot \frac{\partial z}{\partial W_{1i,j}}$$

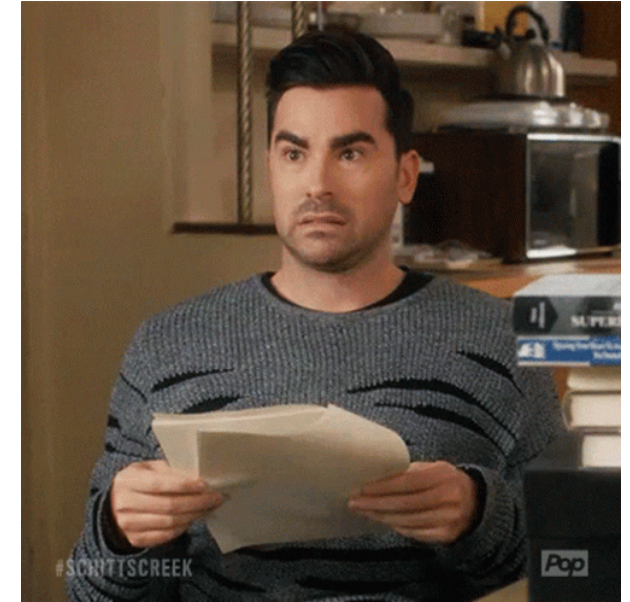
$$\frac{\partial z}{\partial W_{1i,j}} = \frac{\partial g(a)}{\partial a}$$

$$a = W_1 f(x)$$

Are we going to compute derivatives ourselves every time?

No, we will use frameworks that we will do them for us!

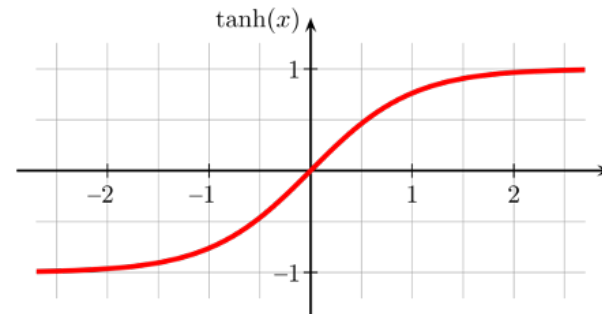
- [Deep Learning with PyTorch: A 60 Minute Blitz](#)



```
import torch
from torchvision.models import resnet18, ResNet18_Weights
model = resnet18(weights=ResNet18_Weights.DEFAULT)
data = torch.rand(1, 3, 64, 64)
labels = torch.rand(1, 1000)
prediction = model(data) # forward pass
loss = (prediction - labels).sum()
loss.backward() # backward pass; autograd calculates and stores the gradients for each model
parameter in the parameter's .grad attribute.
optim = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)
optim.step() #gradient descent; optimizer adjusts each parameter by its gradient stored in .grad
```

Neural Networks: Practical Tips

- Select network structure appropriate for the problem
 - Window vs. recurrent vs. recursive (will discuss throughout the semester)
- Parameter initialization
- Model is powerful enough?
 - If not, make it larger
 - Yes, so regularize, otherwise it will overfit
- Gradient checks to identify bugs
 - If you build from scratch
- Know your non-linearity function and its gradient
 - Example $\tanh(x)$
 - $\frac{\partial}{\partial x} \tanh(x) = 1 - \tanh^2(x)$



Practical Tips

Debugging

- Verify value of initial loss when using softmax
- Perfectly fit a single example, then mini-batch, then train
- If learning fails completely, maybe gradients stuck
 - Check learning rate
 - Verify parameter initialization
 - Change non-linearity functions

Practical Tips

Avoid Overfitting

- Very expressive models, can overfit easily
 - It will look great on the training data, but everything else will be terrible
- Some potential cures 🚑
 - Reduce model size (but not too much)
 - L1 and L2 regularization
 - Early stopping (e.g., patience)
 - Learning rate scheduling
 - Dropout (Hinton et al. 2012)
 - Randomly set 50% of inputs in each layer to 0