Deep Learning

What is deep learning?

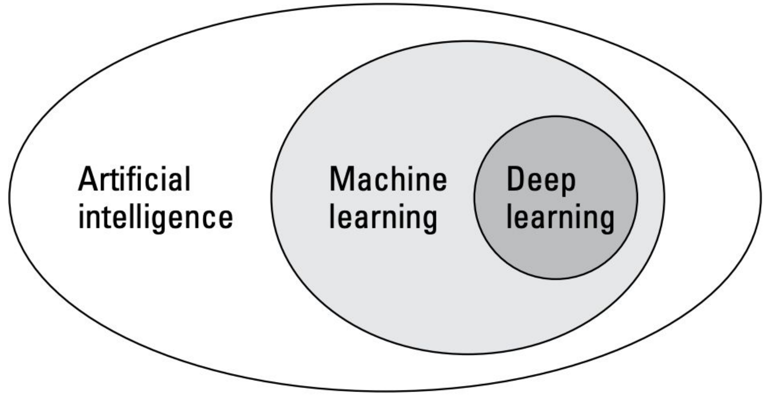


FIGURE 1-1: Deep learning, a subset of a subset of AI.

Artificial Intelligence (AI): system that mimics human intelligence, including rule-based reasoning and problem-solving
Machine Learning (ML): system that learns patterns from data instead of being explicitly programmed
Deep Learning (DL): a system that uses deep neural networks to automatically learn complex patterns

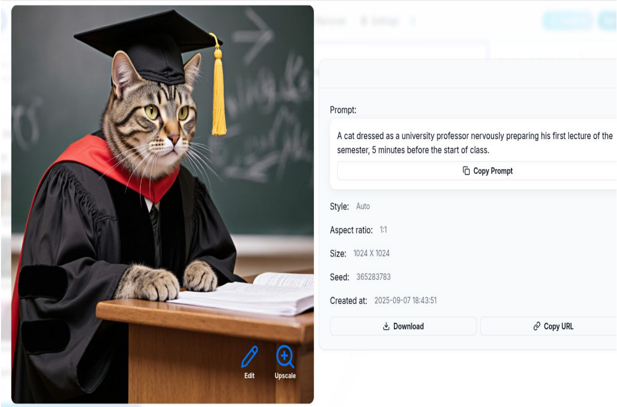
Examples of Deep Learning (DL) models

AlexNet (Krizhevsky et al., 2012)

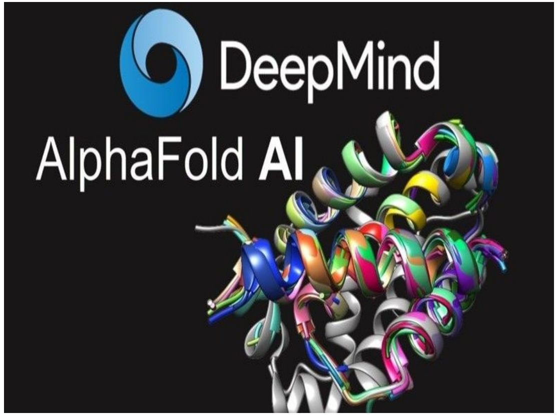
AlphaGo (Silver et al., 2016)

StyleGAN (Karras et al., 2018)

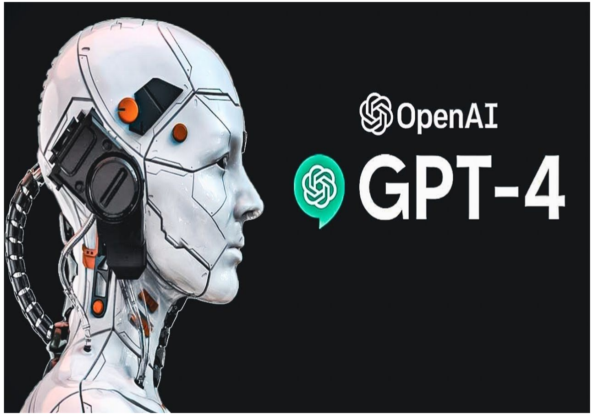
Examples of Deep Learning (DL) models



Stable Diffusion (Rombach et al., 2022)



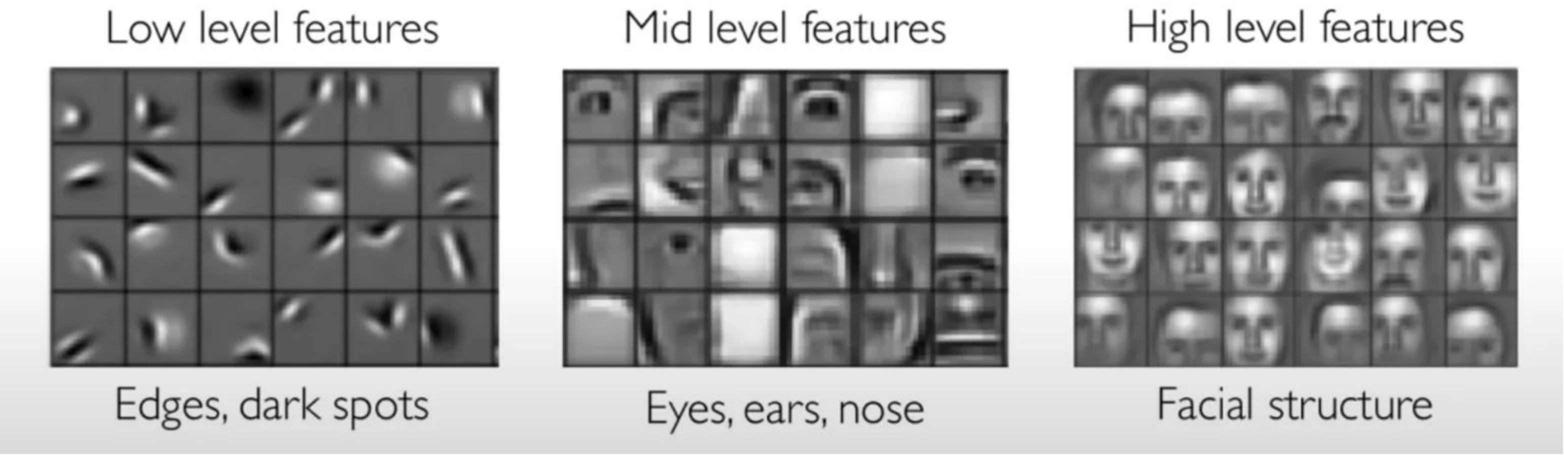
AlphaFold 2 (Jumper et al., 2021)



GPT-4 (OpenAI et al, 2022)

Why study deep learning?

Hand engineered features are time consuming and not scalable in practice
Can we learn the **underlying features** directly from data?



Why there was a boom?

1952 — Stochastic Gradient Descent

1958 — Perceptron
- Learnable Weights

1986 — Backpropagation
- Multi-Layer Perceptron

1989 — Deep Convolutional NN
- Digit Recognition

I. Big Data
- Large Datasets
- Easier Collection & Storage

II. Hardware
- Graphical Processing Units (GPUs)
- Massively Parallelizable

III. Software
- Improved Techniques
- New Models & Toolboxes

Why there was a boom?

Aim of the course

The purpose of this course is to acquaint you with the world of neural networks:

- What type of problems one can solve using ML/DL?
- What methodology should be chosen to solve the problem?
- How to implement the chosen methodology?

Team

Eduard Vladimirov

Education

Bachelor: MIPT with *honours*
Master: MIPT with *honours*

Avito Academy of Analysts

Deep Learning Systems

Experience

Analyst

Data Scientist

Projects

Diffusion Model from scratch

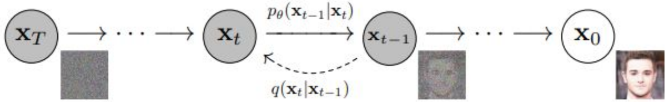


Figure 2: The directed graphical model considered in this work.

Nikita Kiselev

Experience

SBER AI — Deep Learning Researcher

Education

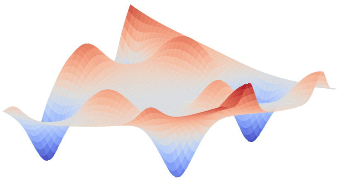
- Bachelor: MIPT with honours

Past Experience

INNOPOLIS UNIVERSITY — Mathematician-Programmer

Technician

Projects



[Loss Landscape Convergence & Sample Size Determination]

[Kandinsky: text-to-image & text-to-video & image-to-video]

Daniil Dorin

Experience

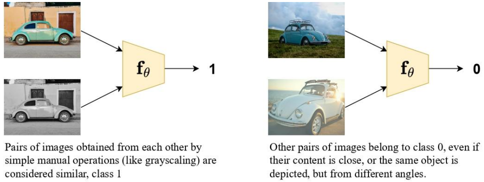
АНТИПЛАГИАТ

CV Researcher-Developer

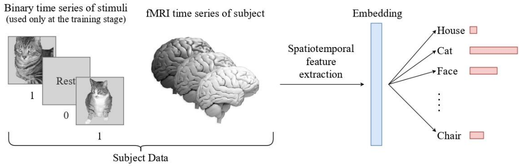
Education

- Bachelor: MIPT with honours

Projects



Pairwise Image Matching for Plagiarism Detection



Enhancing fMRI Data Decoding with Spatiotemporal Characteristics in Limited Dataset

Past Experience

Technician

Sergey Firsov

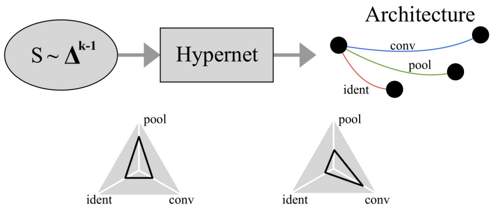
Experience

MIPT & Junior CV engineer

СБЕР СТРАХОВАНИЕ Middle Data Science

Education

- Bachelor: MIPT



[Neural architecture search with target hardware control]

Course

Course programme

Course consists of 14 lectures & seminars:

- Lecture will take one and a half and is focused on theory.
- Then we have a break and continue with a seminar which is focused on practical details of the theory.

Discussed topics:

- Basics of deep learning (3 classes)
- NLP (2 classes)
- Computer Vision (2 classes)
- RL (2 classes)
- Generative models (2 classes)
- Other topics (3 classes)

Final mark = 0.3 x Exam + 0.7 x Homework

Homework

- Homework consists of 6 programming assignments

- You will need GPU to complete your programming homework. Therefore, we recommend that you understand Google Colab and Kaggle.

- Possibly we have some bugs in code :(
Don’t hesitate to contact us. If you let us know after the deadline, we can’t help you.

- Bug fixes and homework proposition (fully implemented in code) are encourages and are rewarded!

Exam

The exam is taken at the end of the course on the topics covered. It consists of answering a question on the theoretical minimum and the main ticket. It is obligatory to pass the exam to get positive mark for the course.

Collaboration policy

All submitted content should be your own content, written yourself!

However, you may (in fact are encouraged to) discuss the homework with others in the chat:

- This creates some room for undue copying, but please obey the reasonable person principle: discuss as you see fit, but don’t simply share answers

You may use snippets of code from sources like Stack Overflow, as long as you cite these properly (put a link to the source)

Lecture 1: Multi-layer perceptron

Reminder: Logistic Regression

Consider some binary classification task:

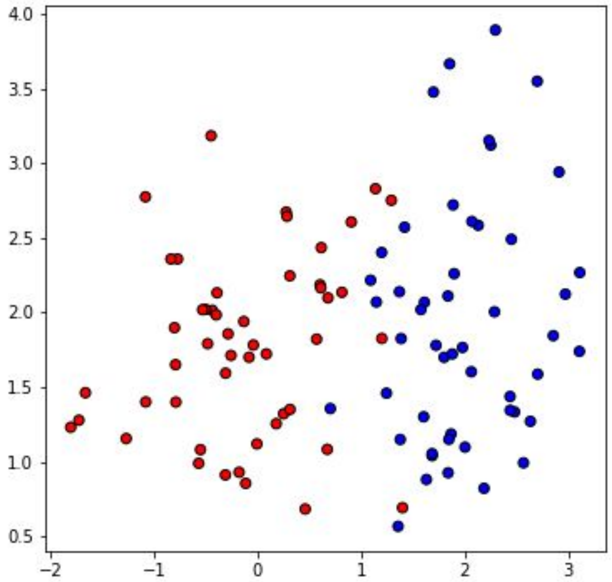
$$\{x_i, y_i\}_{i=1}^n, \ x_i \in R^d, y_i \in \{-1, 1\};$$

We want to train logistic regression models

$$p(y = 1|w, x) = \frac{1}{1 + \exp(-w^\top x)}$$

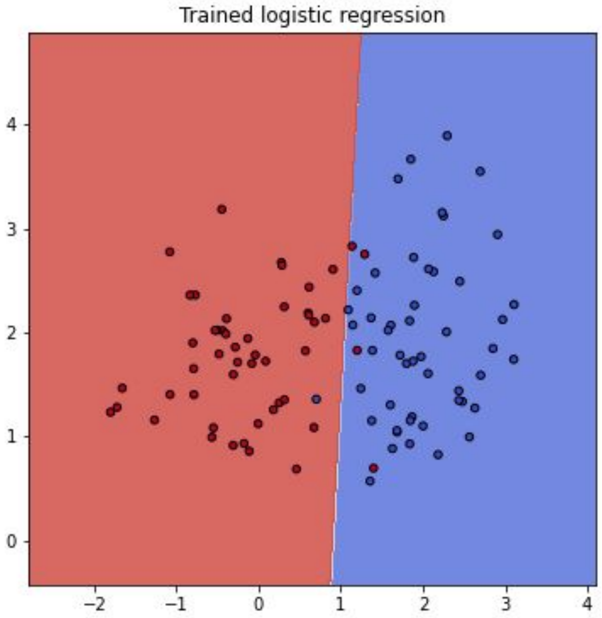
Loss function

$$-\log p(y|W, X) = -\sum_{i=1}^{N} \log p(y_i|W, x_i) \rightarrow min_W$$



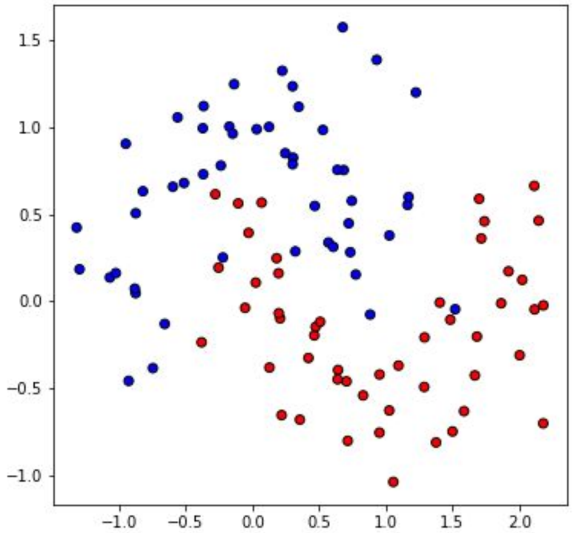
https://mlu-explain.github.io/logistic-regression/

Result

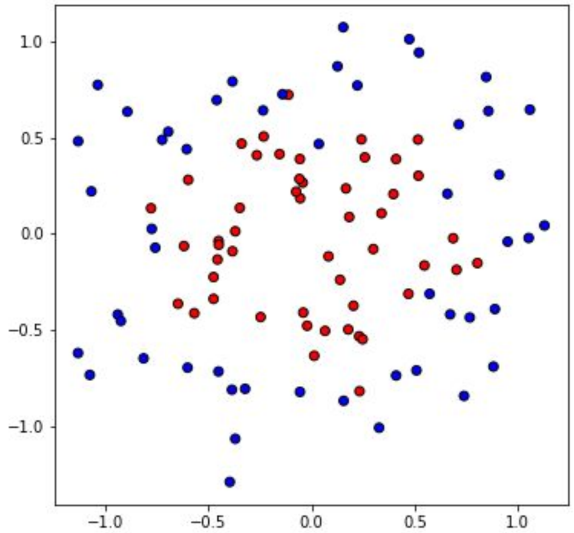


Logistic regression can solve the classification problem with a high accuracy

What does happen when our data has non-linear relationships in our data?



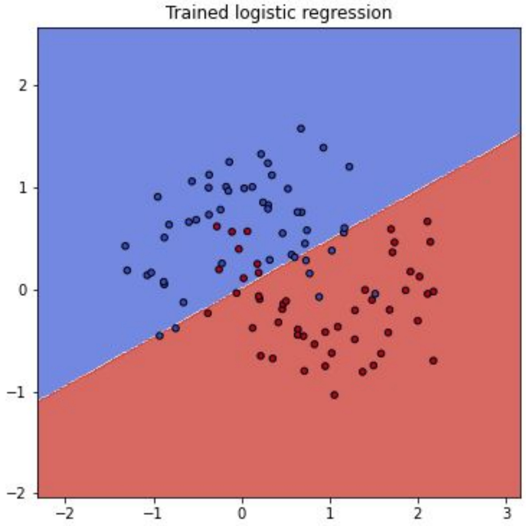
Case 1



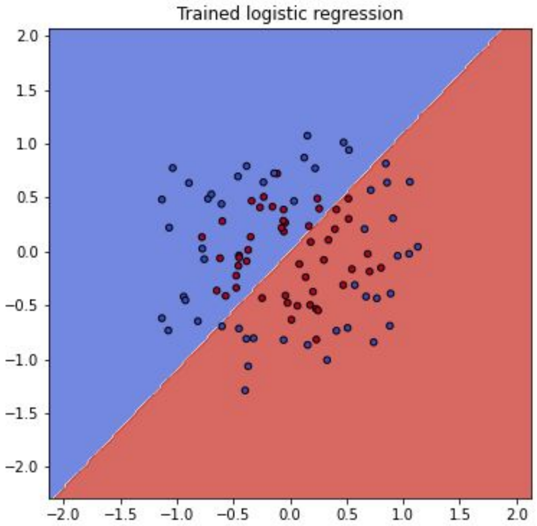
Case 2

In this case logistic regression fails :(

- Moreover, we can hardly do anything about…

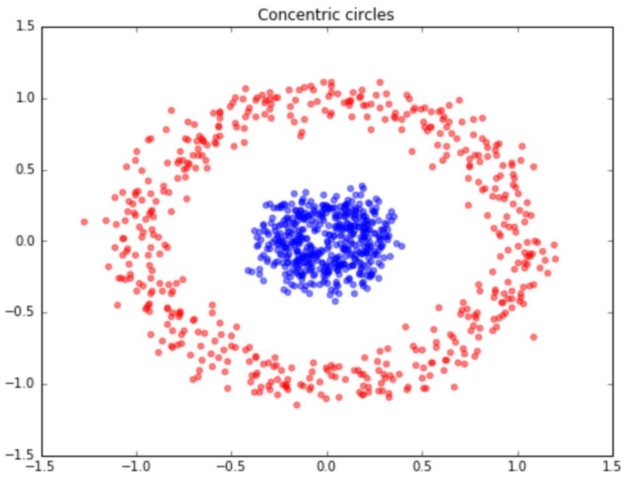


Case 1



Case 2

Some reasons perceptron a.k.a logistic regression fails



- Data can’t be splitted in linear way

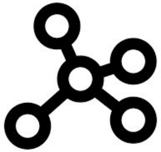
- Features itself are non-informative. But combination does!

https://thiscatdoesnotexist.com

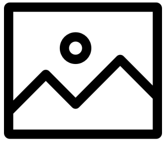
We see failure cases of logistic regression:

- To be honest, it’s just regular cases of real-world data

Audio



Graphs



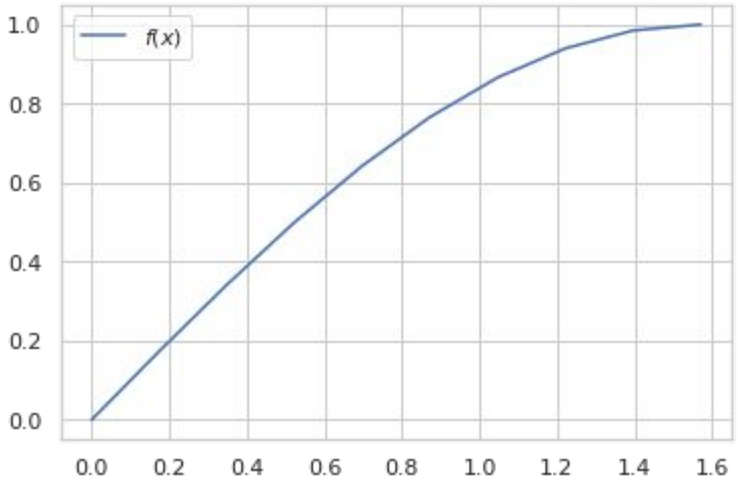
Image

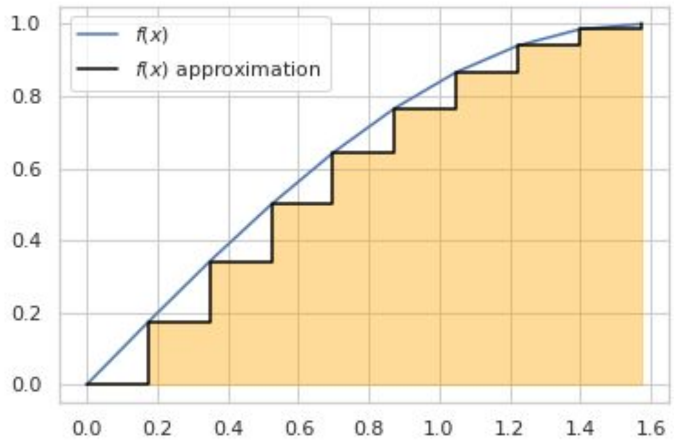
So, we should figure out some solution to overcome linear models problems:

- As always, let’s go to the basics

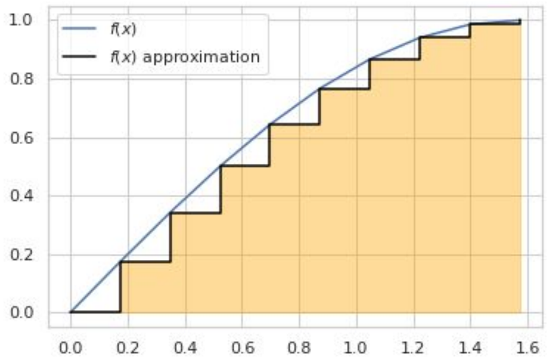
Reminder: Calculus II term

How can we approximate arbitrary continuous function?





We can use approximation with constant thresholds!



Mathematically, it can be written in the following way

- We can approximate the function with the following sum:

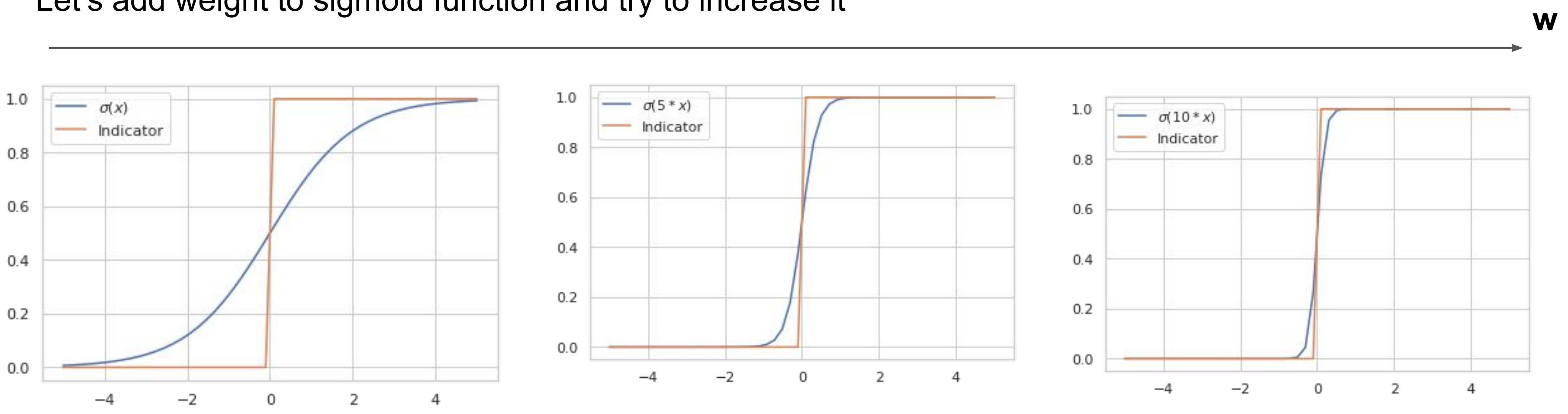
$$f(x) \approx \sum_{i=1}^{N} f(a_i)\underbrace{[x \in [a_i, b_i]]}_{\text{It is not differentiable...}}$$

How can we approximate the following sum even further?

What about sigmoid?

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

Let's add weight to sigmoid function and try to increase it



With the increase of w the sigmoid becoming very close to indicator function!

How to approximate interval indicator?

We have learnt on how to approximate the following function: $[x \geqslant 0]$

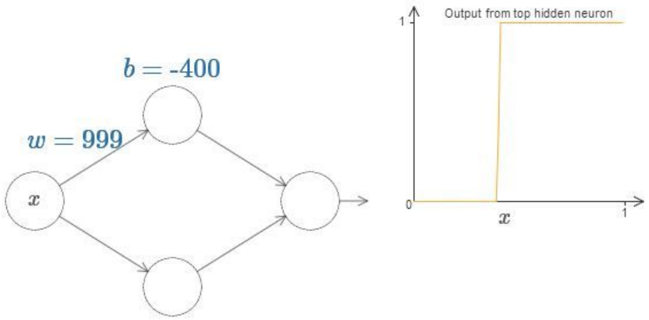
But in reality we need to approximate the following: $f(x) \approx \sum_{i=1}^{N} f(a_i)[x \in [a_i, b_i]]$

<u>We can do a simple trick!</u>

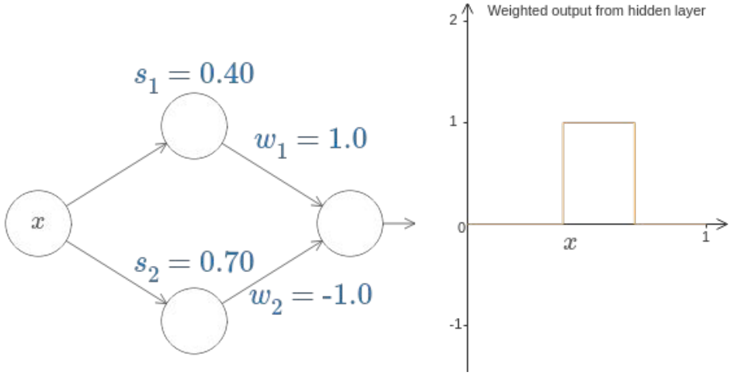
$$[x \in [a_i, b_i]] = [[x \geqslant a_i] - [x \geqslant b_i] > 0.5]$$

Summary!

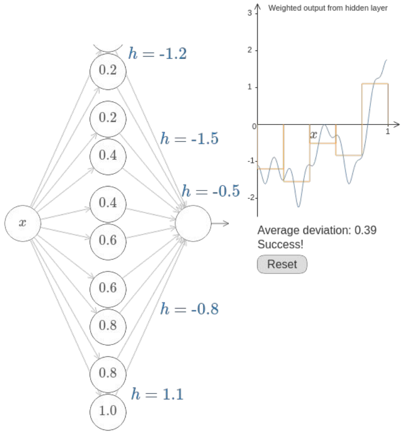
(1) Approximation of Heaviside function



(2) Approximation of Indicator function



3) Approximation of any continuous function



Universal approximation theorem

A sum of the form:

$$f(\mathbf{x}) = \sum_j w_j \sigma\left(b_j + \mathbf{v}_j \cdot \mathbf{x}\right)$$

can approximate any continuous function to any accuracy, but it <u>might require large number of hidden neurons.</u>

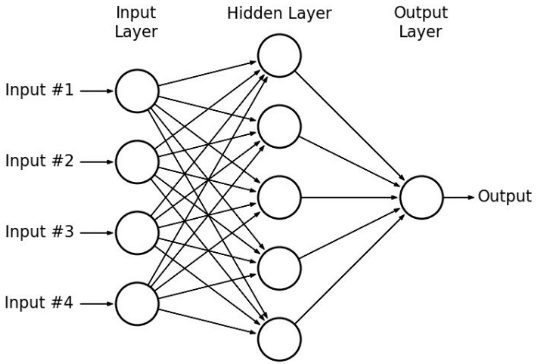
https://www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/

Activation functions

Okay, we should do the following:

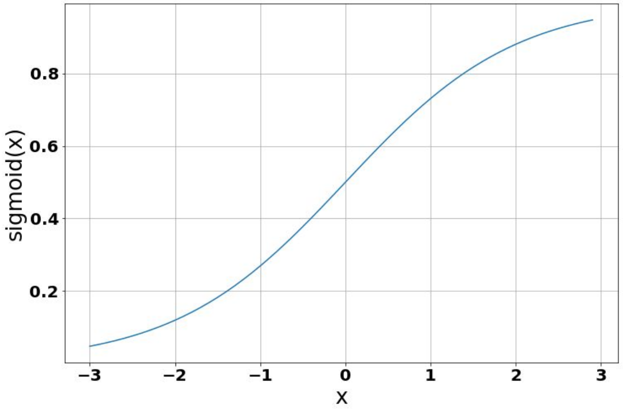
$$h_i = \sigma\left(\sum_{j=1}^{n_{\text{inputs}}} w_{ij}x_j + b_i\right), \quad i = 1, 2, \ldots, n_{\text{hidden}}$$

$$y_k = \sigma\left(\sum_{i=1}^{n_{\text{hidden}}} w'_{ki}h_i + b'k\right), \quad k = 1, 2, \ldots, n_{\text{outputs}}$$

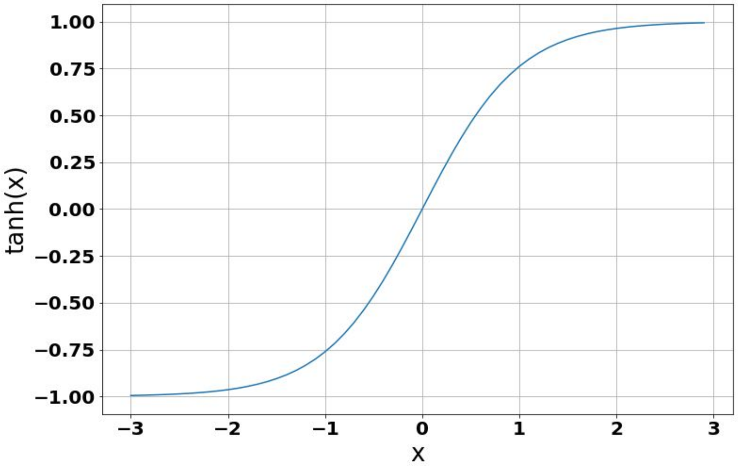


- Can we use Identity transformation instead of sigmoid?
- Can we use other functions?

Activation functions

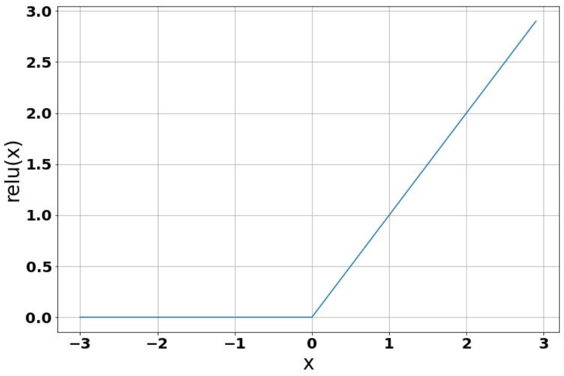


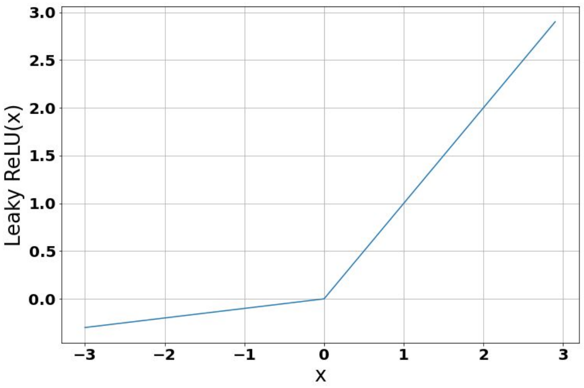
Sigmoid(x)

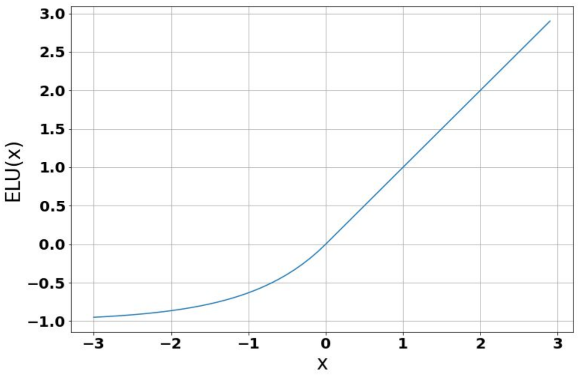


Tanh(x)

Activation functions





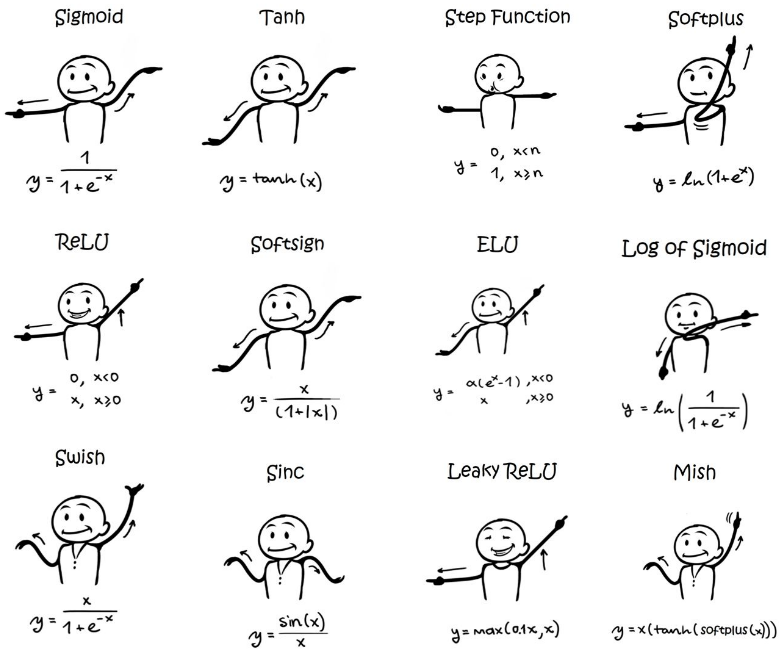


$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases}$$

$$LeakyReLU(z) = \begin{cases} x & x > 0 \\ \alpha x & x <= 0 \end{cases}$$

$$ELU(x) = \begin{cases} x & x > 0 \\ \alpha(e^{\frac{x}{\alpha}} - 1) & x \leqslant 0 \end{cases}$$

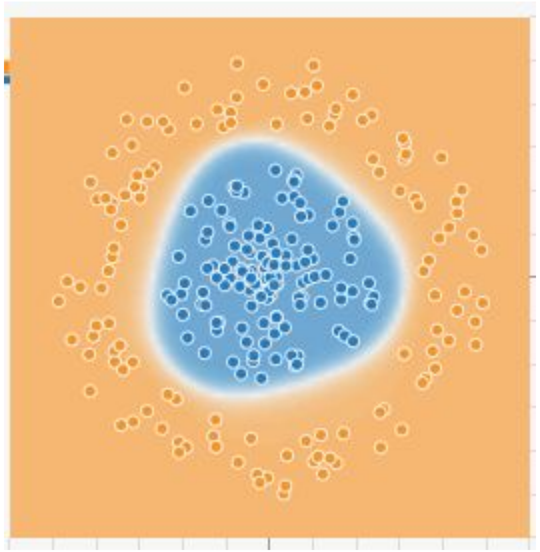
Activation functions



https://sefiks.com/2020/02/02/dance-moves-of-deep-learning-activation-functions/

https://www.vincentsitzmann.com/siren/

Conclusion



- To solve on more complex data, where there is complex relationship between features we should use neural networks.

- A simple example of neural networks isa composition of several perceptron, a.k.a multilayer perceptron

- Multilayer perceptron is a very expressive model - even **one hidden layer is enough to approximate any continuous function.**

https://playground.tensorflow.org/

Gradient calculation

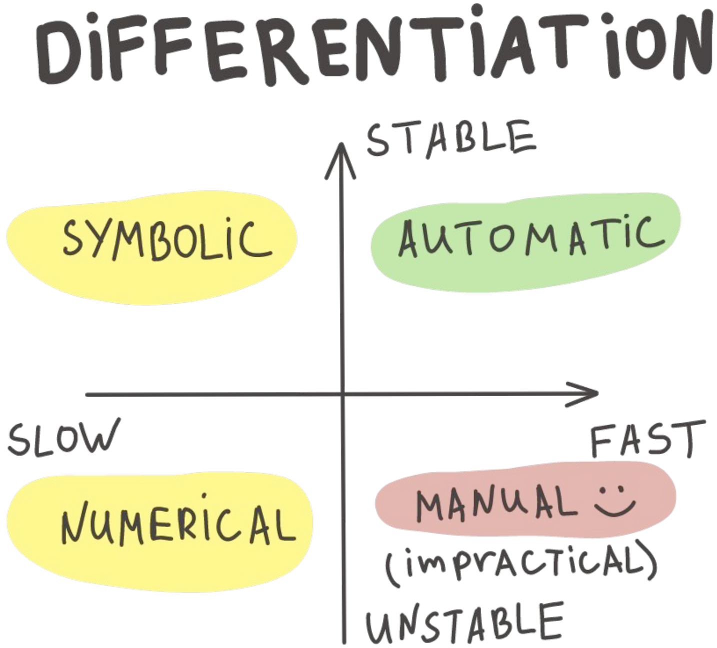
Why do we need gradient calculation?

The workhorse of optimization is gradient descent algorithm:

- Deep learning is not an exception!

We should figure out on how to calculate the gradient:

- Let’s recall our memories about differentiation!



There are four ways on how to do differentiation!

Symbolic differentiation

Write down the formulas, derive the gradient by sum, product and chain rules

$$\frac{\partial(f(\theta)+g(\theta))}{\partial\theta} = \frac{\partial f(\theta)}{\partial\theta} + \frac{\partial g(\theta)}{\partial\theta} \qquad \frac{\partial(f(\theta)g(\theta))}{\partial\theta} = \mathrm{g}(\theta)\frac{\partial f(\theta)}{\partial\theta} + \mathrm{f}(\theta)\frac{\partial g(\theta)}{\partial\theta} \qquad \frac{\partial f(g(\theta))}{\partial\theta} = \frac{\partial f(g(\theta))}{\partial g(\theta)}\frac{\partial g(\theta)}{\partial\theta}$$

Naively do so can result in wasted computations

Example: $f(\theta) = \prod_{i=1}^{n} \theta_i$ $\qquad \frac{f(\theta)}{\partial\theta_k} = \prod_{j \neq k}^{n} \theta_j$

Cost $n(n-2)$ multiplies to compute all partial gradients

Numerical

Calculation of the gradient by definition:

$$\frac{\partial f}{\partial x_i} = \lim_{\varepsilon \to 0} \frac{f\left(x + \varepsilon e_i\right) - f(x)}{\varepsilon} \approx \frac{f\left(x + \varepsilon e_i\right) - f(x)}{\varepsilon}$$

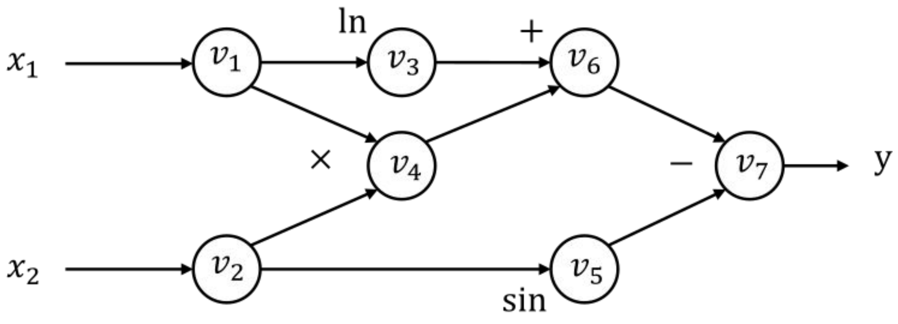
Problems

- For large x dimension we should do a lot of function calls
- Numerically instability, optimal **eps** is equal to square root of machine precision

Hint: One can use numerical differentiation for checking automatic differentiation

Computational graph

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$



Each node represent an (intermediate) value in the computation. Edges present input output relations.

Forward evaluation trace

$$
\begin{aligned}
v_1 &= x_1 = 2 \\
v_2 &= x_2 = 5 \\
v_3 &= \ln v_1 = \ln 2 = 0.693 \\
v_4 &= v_1 \times v_2 = 10 \\
v_5 &= \sin v_2 = \sin 5 = -0.959 \\
v_6 &= v_3 + v_4 = 10.693 \\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
y &= v_7 = 11.652
\end{aligned}
$$

Multi-class classification setting

Let's consider a *k-class classification setting,* where we have

- Training data: $x^{(i)} \in \mathbb{R}^n, y^{(i)} \in \{1, \ldots, k\}$ for $i = 1, \ldots m$
- $n =$ dimensionality of the input data
- $k =$ number of different classes / labels
- $m =$ number of points in the training set

Example: classification of 28x28 MNIST digits

- $n = 28 \cdot 28 = 784$
- $k = 10$
- $m = 60{,}000$

Linear hypothesis function

Our hypothesis function maps inputs $x \in \mathbb{R}^n$ to k-dimensional vectors

$$h: \mathbb{R}^n \to \mathbb{R}^k$$

where $h_i(x)$ indicates some measure of "belief" in how much likely the label is to be class i (i.e., "most likely" prediction is coordinate i with largest $h_i(x)$).

A **linear hypothesis function** uses a *linear* operator (i.e. matrix multiplication) for this transformation

$$h_\theta(x) = \theta^T x$$

for *parameters* $\theta \in \mathbb{R}^{n \times k}$

Matrix batch notation

Often more convenient (and this is how you want to code things for efficiency) to write the data and operations in *matrix batch* form

$$X \in \mathbb{R}^{m \times n} = \begin{bmatrix} - \; {x^{(1)}}^T \; - \\ \vdots \\ - \; {x^{(m)}}^T \; - \end{bmatrix}, \qquad y \in \{1, \ldots, k\}^m = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Then the linear hypothesis applied to this batch can be written as

$$h_\theta(X) = \begin{bmatrix} - \; h_\theta(x^{(1)})^T \; - \\ \vdots \\ - h_\theta(x^m)^T \; - \end{bmatrix} = \begin{bmatrix} - \; {x^{(1)}}^T \theta \; - \\ \vdots \\ - \; {x^{(m)}}^T \theta \; - \end{bmatrix} = X\theta$$

Loss function #1: classification error

The simplest loss function to use in classification is just the classification error, i.e., whether the classifier makes a mistake a or not

$$\ell_{err}(h(x), y) = \begin{cases} 0 & \text{if } \operatorname{argmax}_i h_i(x) = y \\ 1 & \text{otherwise} \end{cases}$$

We typically use this loss function to assess the *quality* of classifiers

Unfortunately, the error is a bad loss function to use for *optimization*, i.e., selecting the best parameters, because it is not differentiable

Loss function #2: softmax / cross-entropy loss

Let's convert the hypothesis function to a "probability" by exponentiating and normalizing its entries (to make them all positive and sum to one)

$$z_i = p(\text{label} = i) = \frac{\exp\big(h_i(x)\big)}{\sum_{j=1}^{k} \exp\big(h_j(x)\big)} \equiv \text{normalize}(\exp(h(x)))$$

Then let's define a loss to be the (negative) log probability of the true class: this is called *softmax* or *cross-entropy* loss

$$\ell_{ce}(h(x), y) = \underbrace{-\log}_{\text{NLL}} \underbrace{p(\text{label} = y)}_{\text{Softmax}} = -h_y(x) + \log \sum_{j=1}^{k} \exp\big(h_j(x)\big)$$

$\begin{bmatrix} \vdots \\ h_i(x) \\ \vdots \end{bmatrix} \quad y = i$

The gradient of the softmax objective

So, how do we compute the gradient for the softmax objective?

$$\nabla_\theta \ell_{ce}(\theta^T x, y) = ?$$

Let's start by deriving the gradient of the softmax loss itself: for vector $h \in \mathbb{R}^k$

$$\frac{\partial \ell_{ce}(h,y)}{\partial h_i} = \frac{\partial}{\partial h_i}\left(-h_y + \log \sum_{j=1}^{k} \exp h_j\right)$$

$$= -1\{i = y\} + \frac{\exp h_i}{\sum_{j=1}^{k} \exp h_j} \quad z$$

e_y

So, in vector form: $\nabla_h \ell_{ce}(h,y) = z - e_y$, where $z = \text{normalize}(\exp(\text{h}))$

$\mathbb{R}^k$ $\quad \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $y=0$ $\quad y=1 \to \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Neural networks in machine learning

Consider the following optimization problem:

$$\underset{\theta}{\text{minimize}} \frac{1}{m} \sum_{i=1}^{m} \ell_{ce}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

Requires computing the gradients $\nabla_\theta \ell_{ce}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$

Let's work through the derivation of the gradients for a simple two-layer network, written in batch matrix form, i.e.

$$\nabla_{\{W_1, W_2\}} \ell_{ce}\left(\sigma\left(X W_1\right) W_2, y\right)$$

The next slides are taken from https://dlsyscourse.org/lectures/

The gradient(s) of a two-layer network

$$\frac{\partial \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right)}{\partial W_2} = \frac{\partial \ell_{ce}(\sigma(XW_1)W_2, y)}{\partial \sigma(XW_1)W_2} \cdot \frac{\partial \sigma(XW_1)W_2}{\partial W_2}$$

$I_y \in \mathbb{R}^{m \times k}$

$k = 2$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{array}{c} y \\ 1 \\ 0 \\ 0 \\ 1 \end{array}$$

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{c} y \\ 2 \\ 1 \\ 0 \\ 2 \end{array}$$

The gradient(s) of a two-layer network

$\mathbb{R} \to \mathbb{R}$

$\begin{bmatrix} -0.2 & 0.5 \\ \vdots & \\ \vdots & \end{bmatrix} \to \begin{bmatrix} 0 & 0.5 \\ \vdots & \\ \vdots & \end{bmatrix}$

$\in \mathbb{R}^{d \times k}$

$$\frac{\partial \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right)}{\partial W_2} = \frac{\partial \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right)}{\partial \sigma\left(XW_1\right)W_2} \cdot \frac{\partial \sigma\left(XW_1\right)W_2}{\partial W_2}$$

$X \in \mathbb{R}^{m \times n}$

$W_1 \in \mathbb{R}^{n \times d}$

$W_2 \in \mathbb{R}^{d \times k}$

$W_2 = W_2 - \alpha \frac{\partial \ell_{ce}}{\partial W_2}$

$$= \underbrace{\left(S - I_y\right)}_{m \times k} \cdot \left(\sigma\left(\overset{m \times d}{XW_1}\right)\right)$$

$$\left[S = \text{normalize}\left(\exp\left(\underbrace{\sigma\left(\overset{m \times d}{XW_1}\right)}_{m \times d} W_2\right)\right)\right]$$

so (matching sizes) the gradient is

$\begin{bmatrix} 0 & 0.2 \\ \vdots & \end{bmatrix} \to \begin{bmatrix} 1 & e^{0.2} \end{bmatrix} \to \begin{bmatrix} \frac{1}{1+e^{0.2}} & \frac{e^{0.2}}{1+e^{0.2}} \end{bmatrix}$

$m \times k \quad m \times d \quad m \times k$

$$\nabla_{W_2} \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right) = \overbrace{\sigma\left(XW_1\right)^T}^{d \times m} \overbrace{\left(S - I_y\right)}^{m \times k} \in \mathbb{R}^{d \times k}$$

The gradient(s) of a two-layer network

Deep breath and let's do the gradient

$$\frac{\partial \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right)}{\partial W_1} =$$

The gradient(s) of a two-layer network

$X \in \mathbb{R}^{m \times n}$

$W_1 \in \mathbb{R}^{n \times d}$

$W_2 \in \mathbb{R}^{d \times k}$

Deep breath and let's do the gradient

$$\underbrace{\frac{\partial \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right)}{\partial W_1}}_{n \times d} = \frac{\partial \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right)}{\partial \sigma\left(XW_1\right)W_2} \cdot \frac{\partial \sigma\left(XW_1\right)W_2}{\partial \sigma\left(XW_1\right)} \cdot \frac{\partial \sigma\left(XW_1\right)}{\partial XW_1} \cdot \frac{\partial XW_1}{\partial W_1}$$

$$= \underbrace{\left(S - I_y\right)}_{m \times k} \cdot \underbrace{\left(W_2\right)}_{d \times k} \cdot \underbrace{\left(\sigma'\left(XW_1\right)\right)}_{m \times d} \cdot \underbrace{\left(X\right)}_{m \times n}$$

and so the gradient is

$$\nabla_{W_1} \ell_{ce}\left(\sigma\left(XW_1\right)W_2, y\right) = \underbrace{X^T}_{n \times m} \left(\underbrace{\left(S - I_y\right)}_{m \times k} \underbrace{W_2^T}_{k \times d} \circ \underbrace{\sigma'\left(XW_1\right)}_{m \times d}\right)$$

Backpropagation "in general"

There *is* a method to this madness … consider our fully-connected network:

$$Z_{i+1} = \sigma_i(Z_i W_i), \qquad i = 1, \ldots, L$$

Then (now being a bit terse with notation)

$$\frac{\partial \ell(Z_{L+1}, y)}{\partial W_i} = \underbrace{\left[\frac{\partial \ell}{\partial Z_{L+1}} \cdot \frac{\partial Z_{L+1}}{\partial Z_L} \cdot \frac{\partial Z_{L-1}}{\partial Z_{L-2}} \cdot \ldots \cdot \frac{\partial Z_{i+2}}{\partial Z_{i+1}}\right]}_{G_{i+1} = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_{i+1}}} \cdot \frac{\partial Z_{i+1}}{\partial W_i}$$

$\frac{\partial \ell}{\partial Z_{i+1}}$

Then we have a simple "backward" iteration to compute the G_i's

$\frac{\partial \ell}{\partial Z_i}$

$$G_i = G_{i+1} \cdot \frac{\partial Z_{i+1}}{\partial Z_i} = G_{i+1} \cdot \frac{\partial \sigma_i(Z_i W_i)}{\partial Z_i W_i} \cdot \frac{\partial Z_i W_i}{\partial Z_i} = G_{i+1} \cdot \left[\sigma'(Z_i W_i) \cdot W_i\right]$$

Computing the real gradients

To convert these quantities to "real" gradients, consider matrix sizes

$$G_i = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_i} = \nabla_{Z_i} \ell(Z_{L+1}, y) \in \mathbb{R}^{m \times n_i}$$

so with "real" matrix operations

$$G_i = G_{i+1} \cdot \sigma'(Z_i W_i) \cdot W_i = \left(G_{i+1} \circ \sigma'(Z_i W_i)\right) W_i^T$$

Similar formula for actual parameter gradients $\nabla_{W_i} \ell(Z_{L+1}, y) \in \mathbb{R}^{n_i \times n_{i+1}}$

$$\frac{\partial \ell(Z_{L+1}, y)}{\partial W_i} = G_{i+1} \cdot \frac{\partial \sigma_i(Z_i W_i)}{\partial Z_i W_i} \cdot \frac{\partial Z_i W_i}{\partial W_i} = G_{i+1} \cdot \sigma'(Z_i W_i) \cdot Z_i$$

$$\Longrightarrow \nabla_{W_i} \ell(Z_{L+1}, y) = Z_i^T \left(G_{i+1} \circ \sigma'(Z_i W_i)\right)$$

Backpropagation

Forward: $X = Z_1 \xrightarrow[W_1]{} Z_2 \xrightarrow[W_2]{} \cdots \xrightarrow[W_{L-1}]{} Z_L \xrightarrow[W_L]{} Z_{L+1}$

Backward $G_1 \xleftarrow[\nabla_{W_1}\ell]{} G_2 \xleftarrow[\nabla_{W_2}\ell]{} \cdots \xleftarrow[\nabla_{W_{L-1}}\ell]{} G_L \xleftarrow[\nabla_{W_L}\ell]{} G_{L+1}$

Putting it all together, we can efficiently compute *all* the gradients we need for a neural network by following the procedure below

$\nabla_{W_i} \ell(Z_{k+1}, y)$

1. Initialize: $Z_1 = X$
   Iterate: $Z_{i+1} = \sigma_i(Z_i W_i), \quad i = 1, \ldots, L$ } Forward pass

2. Initialize: $G_{L+1} = \underline{\nabla_{Z_{L+1}} \ell(Z_{L+1}, y)} = S - I_y$
   Iterate: $\underline{G_i} = \left(G_{i+1} \circ \sigma_i'(Z_i W_i)\right) W_i^T, \quad i = L, \ldots, 1$ } Backward pass

$\frac{\partial \ell}{\partial Z_i}$

And we can compute all the needed gradients along the way

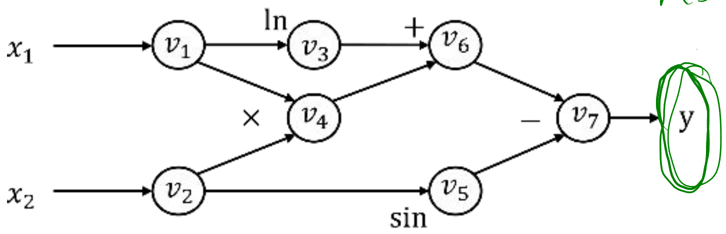
$\frac{\partial \ell}{\partial W_i}$

$$\underline{\nabla_{W_i}} \ell(Z_{k+1}, y) = \underline{Z_i^T \left(G_{i+1} \circ \sigma_i'(Z_i W_i)\right)}$$

"Backpropagation" is just chain rule + intelligent caching of intermediate results

Forward Automatic mode differentiation

$y = \mathrm{f}(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$ ← Noce



Define $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$

We can then compute the $\dot{v}_i$ iteratively in the forward topological order of the computational graph

Forward evaluation trace

$$
\begin{aligned}
v_1 &= x_1 = 2 \\
v_2 &= x_2 = 5 \\
v_3 &= \ln v_1 = \ln 2 = 0.693 \\
v_4 &= v_1 \times v_2 = 10 \\
v_5 &= \sin v_2 = \sin 5 = -0.959 \\
v_6 &= v_3 + v_4 = 10.693 \\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
y &= v_7 = 11.652
\end{aligned}
$$

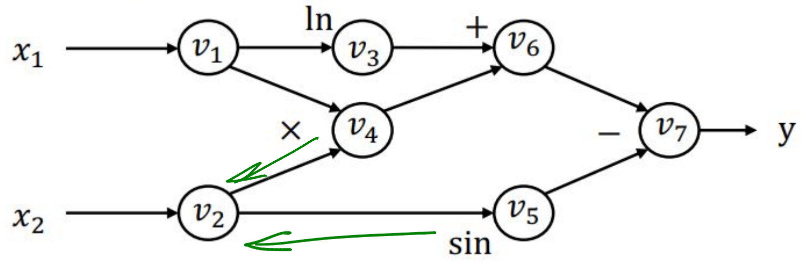
Forward AD trace

$$
\begin{aligned}
\dot{v}_1 &= 1 \\
\dot{v}_2 &= 0 \\
\dot{v}_3 &= \dot{v}_1 / v_1 = 0.5 \\
\dot{v}_4 &= \dot{v}_1 v_2 + \dot{v}_2 v_1 = 1 \times 5 + 0 \times 2 = 5 \\
\dot{v}_5 &= \dot{v}_2 \cos v_2 = 0 \times \cos 5 = 0 \\
\dot{v}_6 &= \dot{v}_3 + \dot{v}_4 = 0.5 + 5 = 5.5 \\
\dot{v}_7 &= \dot{v}_6 - \dot{v}_5 = 5.5 - 0 = 5.5
\end{aligned}
$$

Now we have $\frac{\partial y}{\partial x_1} = \dot{v}_7 = 5.5$

Reverse mode AP

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$



Forward evaluation trace

$$v_1 = x_1 = 2$$
$$v_2 = x_2 = 5$$
$$v_3 = \ln v_1 = \ln 2 = 0.693$$
$$v_4 = v_1 \times v_2 = 10$$
$$v_5 = \sin v_2 = \sin 5 = -0.959$$
$$v_6 = v_3 + v_4 = 10.693$$
$$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$$
$$y = v_7 = 11.652$$

$$\overline{v_2} = \frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_2} + \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_2} = \overline{v_5} \cdot \cos v_2 + \overline{v_4} \cdot v_1$$

Define adjoint $\boxed{\overline{v_i} = \frac{\partial y}{\partial v_i}}$

We can then compute the $\overline{v_i}$ iteratively in the **reverse** topological order of the computational graph

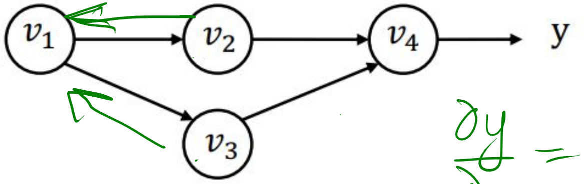
Reverse AD evaluation trace

$$\overline{v_7} = \frac{\partial y}{\partial v_7} = 1$$
$$\overline{v_6} = \overline{v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7} \times 1 = 1$$
$$\overline{v_5} = \overline{v_7}\frac{\partial v_7}{\partial v_5} = \overline{v_7} \times (-1) = -1$$
$$\overline{v_4} = \overline{v_6}\frac{\partial v_6}{\partial v_4} = \overline{v_6} \times 1 = 1$$
$$\overline{v_3} = \overline{v_6}\frac{\partial v_6}{\partial v_3} = \overline{v_6} \times 1 = 1$$
$$\overline{v_2} = \overline{v_5}\frac{\partial v_5}{\partial v_2} + \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_5} \times \cos v_2 + \overline{v_4} \times v_1 = -0.284 + 2 = 1.716$$
$$\overline{v_1} = \overline{v_4}\frac{\partial v_4}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_4} \times v_2 + \overline{v_3}\frac{1}{v_1} = 5 + \frac{1}{2} = 5.5$$

$$\overline{v_6} = \frac{\partial y}{\partial v_6} = \frac{\partial y}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_6} = \overline{v_7} \cdot 1 = 1 \cdot 1 = 1.$$

Derivation for the multiple pathway case

v_1 is being used in multiple pathways (v_2 and v_3)



$$\frac{\partial y}{\partial u_1} = \frac{\partial f(v_2, v_3)}{\partial u_1}$$

y can be written in the form of $y = f(v_2, v_3)$

$$\overline{v_1} = \frac{\partial y}{\partial v_1} = \frac{\partial f(v_2,v_3)}{\partial v_2}\frac{\partial v_2}{\partial v_1} + \frac{\partial f(v_2,v_3)}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \underbrace{\overline{v_2}\frac{\partial v_2}{\partial v_1}}_{\overline{v_{1\to 2}}} + \underbrace{\overline{v_3}\frac{\partial v_3}{\partial v_1}}_{\overline{v_{1\to 3}}}$$

Define partial adjoint $\overline{v_{i\to j}} = \overline{v_j}\frac{\partial v_j}{\partial v_i}$ for each input output node pair i and j

$$\overline{v_1} = \overline{v_{1\to 2}} + \overline{v_{2\to 3}}.$$

$$\overline{v_i} = \sum_{j \in next(i)} \overline{v_{i\to j}}$$

We can compute partial adjoints separately then sum them together

Reverse mode AD algorithm

def gradient(out):

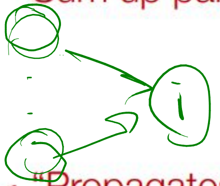
    node_to_grad = {out: [1]}

Dictionary that records a list of partial adjoints of each node

    for i in reverse_topo_order(out):

        $\overline{v_i} = \sum_j \overline{v_{i \to j}}$ = sum(node_to_grad[i])

Sum up partial adjoints



        for $k \in inputs(i)$:

            compute $\overline{v_{k \to i}} = \overline{v_i} \dfrac{\partial v_i}{\partial v_k}$

            append $\overline{v_{k \to i}}$ to node_to_grad[k]

"Propagates" partial adjoint to its input

    return adjoint of input $\overline{v_{input}}$

Reverse mode AD by extending computational graph

$e^{v_1} \cdot e^{v}$ $(e^x+1)e^x$

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
1   v̄_i = Σ_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
2.    compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
3     append v̄_{k→i} to node_to_grad[k]
  return adjoint of input v̄_input
```

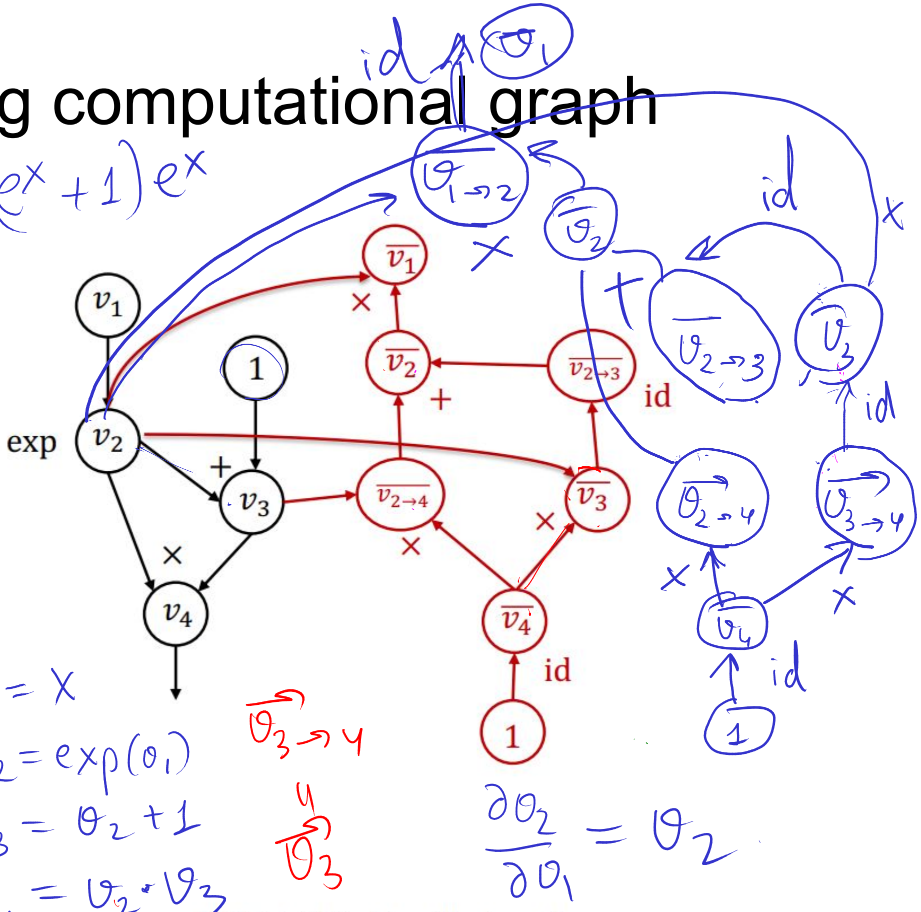
$i = 2$

```
node_to_grad: {
  1: [v̄_1]
  2: [v̄_{2→4}, v̄_{2→3}]
  3: [v̄_3]
  4: [v̄_4]
}
```

$\{4: [\bar{v}_4]\}$

$\{4: [\bar{v}_4]$
$3: [\bar{v}_{3\to4}]$
$2: [\bar{v}_{2\to4}, \bar{v}_{2\to3}]\}$

$\frac{\partial v_3}{\partial v_2}$



$v_1 = x$

$v_2 = \exp(v_1)$

$v_3 = v_2 + 1$

$v_4 = v_2 \cdot v_3$

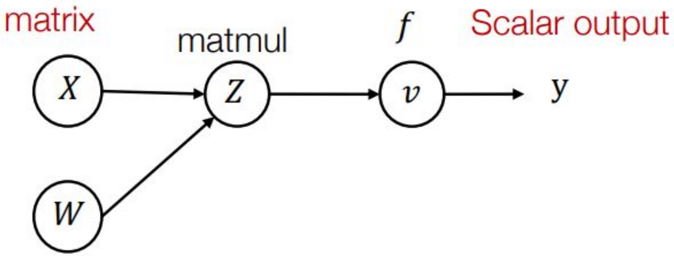
$\bar{v}_{3\to4}$

$\overset{4}{\bar{v}_3}$

$\frac{\partial v_2}{\partial v_1} = v_2$

NOTE: id is identity function

Reverse mode AD on tensors



Define adjoint for tensor values $\bar{Z} = \begin{bmatrix} \frac{\partial y}{\partial Z_{1,1}} & \cdots & \frac{\partial y}{\partial Z_{1,n}} \\ \cdots & \cdots & \cdots \\ \frac{\partial y}{\partial Z_{m,1}} & \cdots & \frac{\partial y}{\partial Z_{m,n}} \end{bmatrix}$

Forward evaluation trace

$$Z_{ij} = \sum_k X_{ik} W_{kj}$$
$$v = f(Z)$$

Reverse evaluation in scalar form

$$\overline{X_{i,k}} = \sum_j \frac{\partial Z_{i,j}}{\partial X_{i,k}} \bar{Z}_{i,j} = W_{k,j}\, \bar{Z}_{i,j}$$

Forward matrix form

$$Z = XW$$
$$v = f(Z)$$

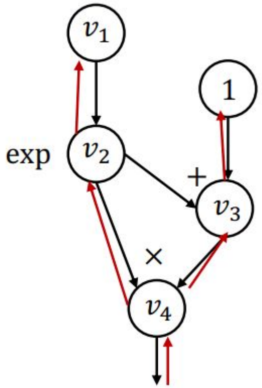
Reverse matrix form

$$\bar{X} = \bar{Z} W^T$$

Reverse mode AD vs Backprop

requires_grad { False / True

Backprop

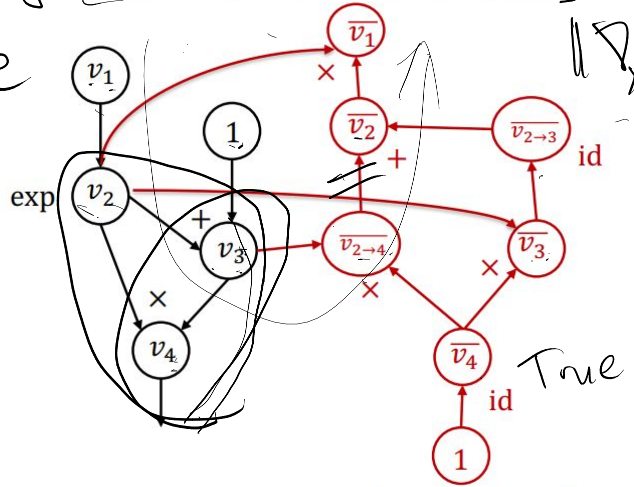


W_L.requires_grad [= True

Reverse mode AD by extending computational graph]

GAN

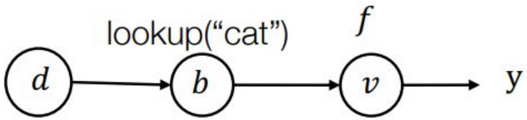
$\|\nabla_x \ell - 1\|$



- Run backward operations the same forward graph
- Used in first generation deep learning frameworks (caffe, cuda-convnet)

- Construct separate graph nodes for adjoints
- Used by modern deep learning frameworks

Reverse mode AD on data structures



Forward evaluation trace

d = {"cat": a_0, "dog": a_1}
$b = d$ ["cat"]
$v = f(b)$

Define adjoint data structure

$\bar{d}$ ={"cat":$\frac{\partial y}{\partial a_0}$, "dog":$\cancel{\frac{\partial y}{\partial a_1}}$} $\frac{\partial y}{\partial a_1}$.

Reverse evaluation

$\bar{b} = \frac{\partial v}{\partial b}\bar{v}$
$\bar{d}$ = {"cat": $\bar{b}$ }

type(d) == type($\bar{d}$).
d.shape == $\bar{d}$.shape.

Key take away: Define "adjoint value" usually in the same data type as the forward value and adjoint propagation rule. Then the sample algorithm works.

Do not need to support the general form in our framework, but we may support "tuple values"

Automatic differentiation: summary

- There is a numerical differentiation. It’s not effective, but you can use it as a simple check
- The classical algorithm is a backpropagation - it uses chain rule to calculate the gradient
- The modern algorithm is a reverse mode AD which construct computational graph for both forward and backward passes. It’s more efficient because of possible node optimization and flexibility (allows to calculate loss w.r.t gradient for free)
- Also, there is a forward AD, but it’s efficient only in a case of low number input and large numbers of outputs.

Homework preparation

Differentials

To understand on how to calculate the gradient, we should recall what differential is.

Вход \ Выход	Скаляр	Вектор	Матрица
Скаляр	$df(x) = f'(x)dx$ ($f'(x)$: скаляр; dx: скаляр)	–	–
Вектор	$df(x) = \langle \nabla f(x), dx \rangle$ ($\nabla f(x)$: вектор; dx: вектор)	$df(x) = J_f(x)dx$ ($J_f(x)$: матрица; dx: вектор)	–
Матрица	$df(X) = \langle \nabla f(X), dX \rangle$ ($\nabla f(X)$: матрица; dX: матрица)	–	–

Note: $\langle A, B \rangle = \sum_{i,j} A_{ij} B_{i,j} = tr(A^T B)$

Matrix-vector differentiation

Conversion Rules

$$dA = 0$$
$$d(\alpha X) = \alpha(dX)$$
$$d(AXB) = A(dX)B$$
$$d(X + Y) = dX + dY$$
$$d(X^T) = (dX)^T$$
$$d(XY) = (dX)Y + X(dY)$$
$$d\langle X, Y\rangle = \langle dX, Y\rangle + \langle X, dY\rangle$$
$$d\left(\frac{X}{\phi}\right) = \frac{\phi dX - (d\phi)X}{\phi^2}$$

Standard derivatives table

$$d\langle A, X\rangle = \langle A, dX\rangle$$
$$d\langle Ax, x\rangle = \langle (A + A^T)x, dx\rangle$$
$$d\langle Ax, x\rangle = 2\langle Ax, dx\rangle \qquad (\textbf{если } A = A^T)$$
$$d(\mathrm{Det}(X)) = \mathrm{Det}(X)\langle X^{-T}, dX\rangle$$
$$d(X^{-1}) = -X^{-1}(dX)X^{-1}$$

Examples of calculating the differential of functions.

Example 1:

$$f(x) = (x^T A x)(x^T B x), \quad x \in \mathcal{R}^n, \;\; A, B \in \mathcal{R}^{n \times n} \;\; A = A^T, B = B^T$$

$$df = d(x^T A x)(x^T B x) + (x^T A x) d(x^T B x) =$$
$$= 2x^T A dx (x^T B x) + (x^T A x) 2x^T B dx$$

Examples of calculating the differential of functions.

Example 2:

$$f(x) = \|x\|_2^3, \quad x \in \mathcal{R}^n$$

$$df = \frac{3}{2}(x^T x)^{\frac{1}{2}} d(x^T x) = \frac{3}{2}(x^T x)^{\frac{1}{2}} 2x^T dx = 3\|x\|_2 x^T dx$$

Examples of calculating the differential of functions.

Example 3:

$$f(X) = det(AXB), \quad X \in \mathcal{R}^{n\times n}$$

$$df = det(AXB)tr((AXB)^{-1}d(AXB)) =$$

$$= det(AXB)tr((AXB)^{-1}AdXB)$$

Then the gradients of the functions from the examples.

Examples 1:

$$\begin{aligned}
df &= d(x^T Ax)(x^T Bx) + (x^T Ax)d(x^T Bx) = \\
&= 2x^T Adx(x^T Bx) + (x^T Ax)2x^T Bdx = \\
&= 2(x^T Bx)x^T Adx + 2(x^T Ax)x^T Bdx = \\
&= (2(x^T Bx)x^T A + 2(x^T Ax)x^T B)dx = \nabla f(x)^T dx
\end{aligned}$$

Then: $\nabla f(x) = 2((x^T Bx)xAx + 2(x^T Ax)Bx)$

Then the gradients of the functions from the examples.
Examples 2-3:

$$df = 3\|x\|_2 x^T dx = \nabla f(x)^T dx$$

Then:

$$\nabla f(x) = 3\|x\|_2 x$$

$$df = det(AXB)tr((AXB)^{-1}AdXB) =$$

$$= tr(det(AXB)B(AXB)^{-1}AdX) = tr(\nabla f(X)^T dX)$$

Then:

$$\nabla f(X) = det(AXB)A^T(AXB)^{-T}B^T$$

Hessian Calculation

$$f : U \to V$$

First differential: $df(x)[h]$

Second differential: $d^2 f(x)[h_1, h_2] = d(df(x)[h_1])(x)[h_2]$

If : $U = \mathcal{R}^n,\ V = \mathcal{R}$

Then: $d^2 f(x)[h_1, h_2] = h_1^T \nabla^2 f(x) h_2$

Show the calculation of the Hessian by an example

$$f(x) = \|x\|_2^3 \qquad\qquad df = 3\|x\|_2 x^T dx$$

$$\begin{aligned}
d^2 f(x)[dx_1, dx_2] &= d\left(3\|x\|_2 x^T dx_1\right) = \\
&= 3d((x^T x)^{\frac{1}{2}}) x^T dx_1 + 3(x^T x)^{\frac{1}{2}} d(x^T dx_1) = \\
&= 3\frac{1}{2}(x^T x)^{-\frac{1}{2}} 2x^T dx_2 x^T dx_1 + 3(x^T x)^{\frac{1}{2}} dx_2^T dx_1 = \\
&= dx_1^T x 3(x^T x)^{-\frac{1}{2}} x^T dx_2 + dx_1^T 3(x^T x)^{\frac{1}{2}} I dx_2
\end{aligned}$$

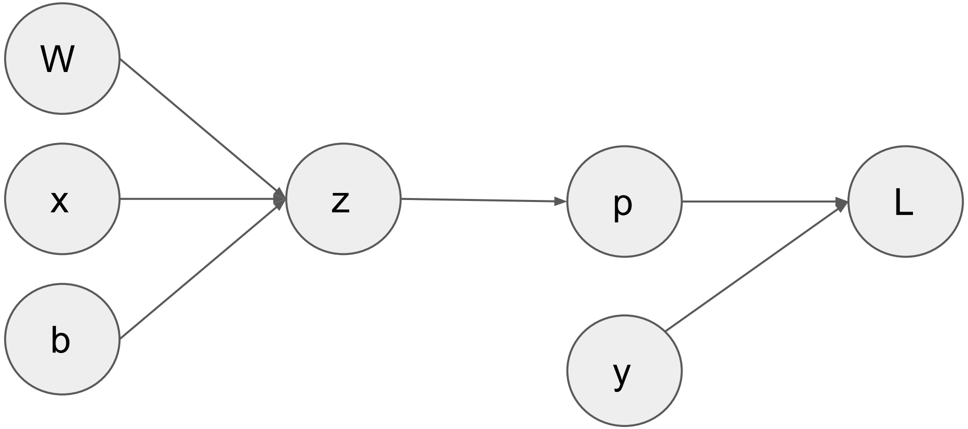
$$\boxed{\nabla^2 f(x) = 3\frac{xx^T}{\|x\|} + 3\|x\| I}$$

Gradient Calculation Examples: Logistic regression

$$z = Wx + b, \quad W \in \mathcal{R}^{K \times D}, \;\; b \in \mathcal{R}^{D} \qquad\qquad x \in \mathcal{R}^{D}, \quad y \in \{1, 2, \ldots, K\}$$

$$p = softmax(z) = \frac{\exp(z)}{\sum_j \exp(z_j)}$$

$$L(p, y) = -\log p_y = -\log p^T \mathbf{1}_y, \quad \mathbf{1}_y = [0, \ldots, 0, \overset{y}{1}, 0, \ldots, 0]$$



Gradient Calculation Examples: Logistic regression

$$dL = d(-\log p^T \mathbf{1}_y) = -\frac{1}{p^T \mathbf{1}_y} d(p^T \mathbf{1}_y) = -\frac{1}{p^T \mathbf{1}_y} \mathbf{1}_y d(p)$$

then:

$$\nabla_p L = -\frac{1}{p^T \mathbf{1}_y} \mathbf{1}_y$$

Next, we need to calculate how p and z are related:

$$p = \frac{\exp(z)}{\exp(z)^T \mathbf{1}}$$

Gradient Calculation Examples: Logistic regression

$$dp = \frac{d(\exp(z))\exp(z)^T\mathbf{1} - \exp(z)d(\exp(z)^T\mathbf{1})}{(\exp(z)^T\mathbf{1})^2} =$$

$$= \frac{diag(\exp(z))dz\exp(z)^T\mathbf{1} - \exp(z)\mathbf{1}^T diag(\exp(z))dz}{(\exp(z)^T\mathbf{1})^2}$$

$$dL = -\frac{1}{p^T\mathbf{1}_y}\mathbf{1}_y dp =$$

$$= -\frac{1}{p^T\mathbf{1}_y}\frac{1}{\exp(z)^T\mathbf{1}}\mathbf{1}_y^T diag(\exp(z))dz + \frac{1}{p^T\mathbf{1}_y}\frac{\mathbf{1}_y\exp(z)\mathbf{1}_y^T diag(\exp(z))dz}{(\exp(z)^T\mathbf{1})^2}$$

Gradient Calculation Examples: Logistic regression

$$\nabla_z L = -\frac{1}{p^T \mathbf{1}_y} \frac{1}{\exp(z)^T \mathbf{1}} \mathbf{1}_y^T diag(\exp(z)) + \frac{1}{p^T \mathbf{1}_y} \frac{\mathbf{1}_y \exp(z) \mathbf{1}_y^T diag(\exp(z))}{(\exp(z)^T \mathbf{1})^2}$$

$$z = Wx + b$$

$$dz = dWx + Wdx + db$$

$$dL = \nabla_z L^T dz = \nabla_z L^T (dWx + Wdx + db) = tr(x \nabla_z L^T dW) + \nabla_z L^T W dx + \nabla_z L^T db$$

Then: $\nabla_W L = \nabla_z L x^T$ $\quad \nabla_x L = W^T \nabla_z L$ $\quad \nabla_b L = \nabla_z L$

Recap

- Multi-layer perceptron
  - Activations
  - Properties

- Gradient calculation
  - How to calculate gradient?