Introduction to Neural Networks

Learning like a human

Yordan Darakchiev

Technical Trainer

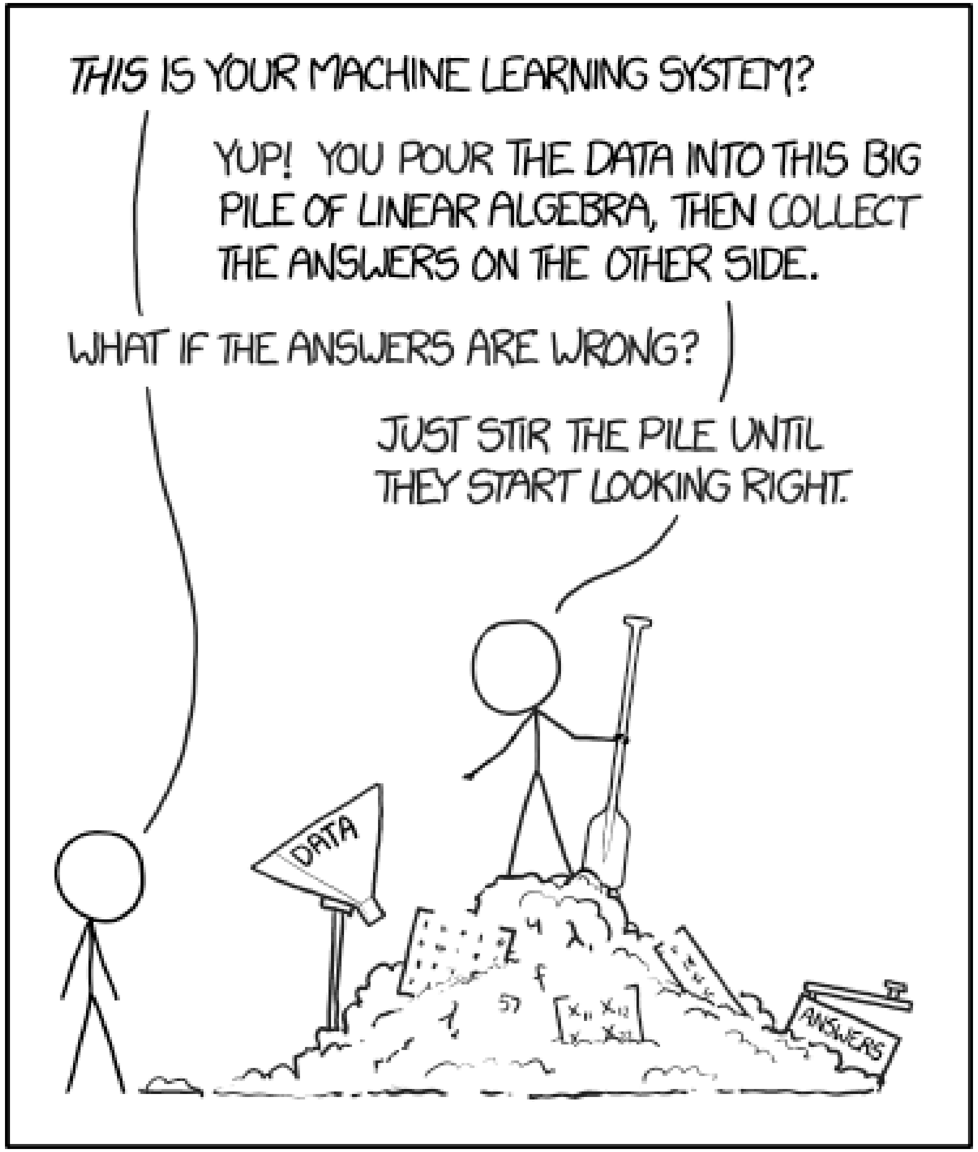
iordan93@gmail.com

sli.do
#MachineLearning

Table of Contents

- Neural networks
  - Overview
  - Problem statement
- Pros and cons
- Perceptron
- Feed-forward NNs (multi-layer perceptrons)
  - Training
  - Applications for classification

Before We Start…



Neural Networks

Combining simple algorithms to achieve glory

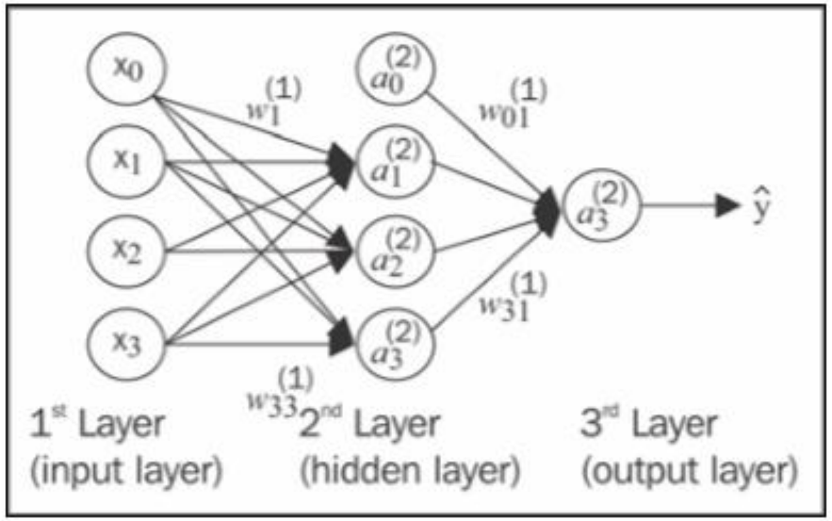
Neural Networks

- Neural networks try to mimic the way the human brain works
  - Series of interconnected artificial neurons (perceptrons)
  - Can do classification, regression, unsupervised learning, etc.
- Perceptrons were "invented" in the 1940s
  - Great development in the recent years
- "Deep learning" – ML algorithms using neural networks
- Cutting-edge applications
  - Machine translation
  - Speech recognition and generation
  - Image recognition
  - Game playing, etc.
- Some examples of deep learning applications

Neural Networks: Pros and Cons

- Can be used to model any datasets
  - Arbitrary dataset complexity
  - One type of algorithm can be used for many applications
- Do not provide any interpretability
  - The classification boundaries are hard to interpret
  - The model is mostly "black box"
  - NNs are not probabilistic (we can't get a confidence metric)
  - "The Dark Secrets at the Heart of AI"
    - Solutions: trying to explain decisions, combining with other algorithms, etc.
- Can be slow
  - Other models usually train a lot faster, even if we use special hardware
- **NNs are not a substitute for understanding the problem deeply**

Neural Network Architecture



- Neural network layout
  - Input(s) (+ bias unit)
  - "Hidden layers" (+ bias units)
  - Output(s)
- Each "node" is a perceptron
- Each arrow carries 0 or 1, and is assigned a weight
- The layers are fully connected
  - There are no connections within layers
- More than 1 hidden layer → "deep learning" (deep NN)
- How many layers? How many units per layer?
  - We don't know :(⇒ hyperparameter tuning

Neural Network Learning

- The type of NN we look at is called a "feed-forward NN"
  - Data flows only forward, there are no "back-links"
- Learning algorithm:
  - Forward propagation / backpropagation
  - Using the data, propagate the patterns from input to output
  - Based on the output, calculate the error (using a cost function)
  - Backpropagate the error (using derivatives), update the model
- We get the "final" weights after repeating the process for several epochs
- The math is a bit ugly
  - You can read an explanation here

Neural Network Learning (2)

- Classification: just use one-hot encoding
  - MLP = multi-layer perceptron

```
from sklearn.neural_network import MLPClassifier
```

- Regression: no activation function at the output layer

```
from sklearn.neural_network import MLPRegressor
```

- Regularization: parameter `alpha`
  - Increasing = less overfitting
  - A [visual comparison](#) of regularization parameters
- Tips
  - A neural network is very sensitive to feature scaling
    - [0; 1], [-1; 1] or Z
    - Use a scaler, e.g. `StandardScaler`
  - Use fine-tuning to optimize `alpha`
    - Usually in the range `10.0 ** -np.arange(1, 7)`

Example: Classifying Handwritten Digits

- Obtain the MNIST dataset of handwritten digits
  - This is a famous dataset for learning and comparing neural networks
  - Each data point represents a 28 x 28 image of a digit (0 – 9)
- Train a simple NN on the MNIST dataset
  - Choose a reasonable number of layers and units per layer, e.g. {3, 3}
- Test, score and evaluate the classification performance
  - E.g., accuracy, precision, recall, F1, confusion matrix, ROC curve
- * Try several other architectures (e.g., more layers, more units per layer, different structure, e.g., 2 + 3 + 2 units, etc.)
- * Compare the results with (an)other classifier(s), e.g., SVM

Neural Network Implementation

Achieving glory…
just a little bit harder

Conventions

- Try to vectorize where possible
  - Hundreds to thousands of times faster
- Always use 2-dimensional matrices
  - Matrix: `[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]`
  - Row vector: `[[1, 10, 100]]`
  - Column vector: `[[10], [100], [1000]]`
  - Scalar: can be `[[42]]` or just the number
- Python broadcasting will turn any vector to a matrix where needed
  - If a dimension has size 1 (e.g., 3×1), it will be copied
  - `[[2, 3], [4, 5] + [[-4, 2]]`
⇒ `[[2, 3], [4, 5] + [[-4, 2], [-4, 2]]`

Review: Logistic Regression

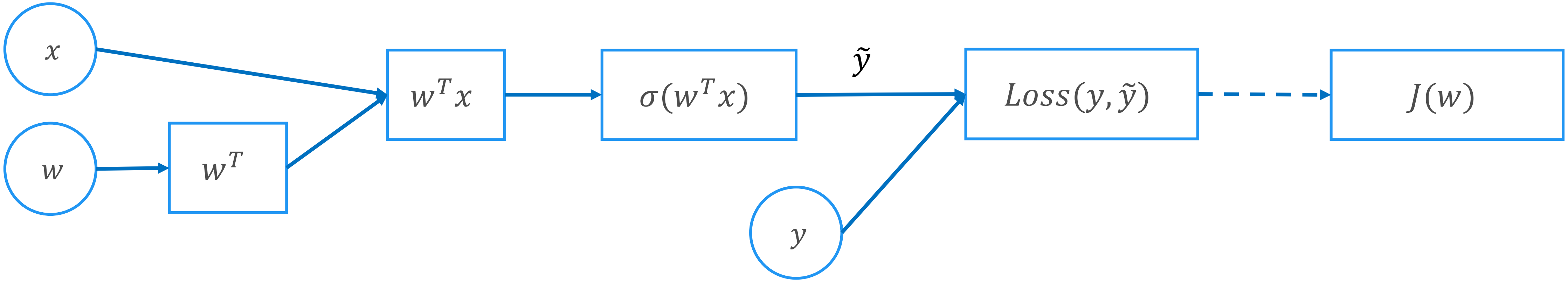
- The main NN unit (perceptron) does exactly this
- Input: $x = [x_1, x_2, \dots, x_m]^T$, $x^{(1)}, x^{(2)}, \dots, x^{(n)}$; output $p \in [0; 1]$
- Objective: Maximize the probability of the class given the input
  - Simplest possible: linear combination $w_0 + w_1 x_1 + \cdots + w_m x_m$
  - Convert this to be $[0; 1]$: $\tilde{y} = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(w_0 + w_1 x_1 + \cdots + w_m x_m)}}$
- Input augmentation
  - $w_0 = w_0 . 1$
    - $x = [1, x] = [1, x_1, x_2, \dots, x_m]^T$
    - Also: $w = [w_0, w_1, \dots, w_m]^T$
    - $\Rightarrow w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = [w_1 \quad w_2 \quad \dots \quad w_m] \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_m \end{bmatrix} \equiv w^T x$

Review: Logistic Regression (2)

- Objective function: $\tilde{y} = \sigma(w^T x)$
  - Represents the probability the class is 1 given x: $p(y = 1|x)$
- Loss function
  - If $y = 1$, $p(y|x) = \tilde{y}$; if $y = 0$, $p(y|x) = 1 - \tilde{y}$
  - Combined loss (we can check that): $p(y|x) = \tilde{y}^y (1 - \tilde{y})^{1-y}$
  - Log both sides:
    - $\ln p(y|x) = y \ln(\tilde{y}) + (1 - y) \ln(1 - \tilde{y})$
    - We want to maximize the probability, so the loss should be $\boldsymbol{-\ln p(y|x)}$
- Total cost function: The average of all losses (on all examples)
  - $J(w) = -\frac{1}{n} \sum_{i=1}^{n} \left(\left(y^{(i)} \ln\left(\tilde{y}^{(i)}\right) + \left(1 - y^{(i)}\right) \ln\left(1 - \tilde{y}^{(i)}\right) \right) \right)$
  - This is called **categorical cross-entropy** and is widely used in machine learning

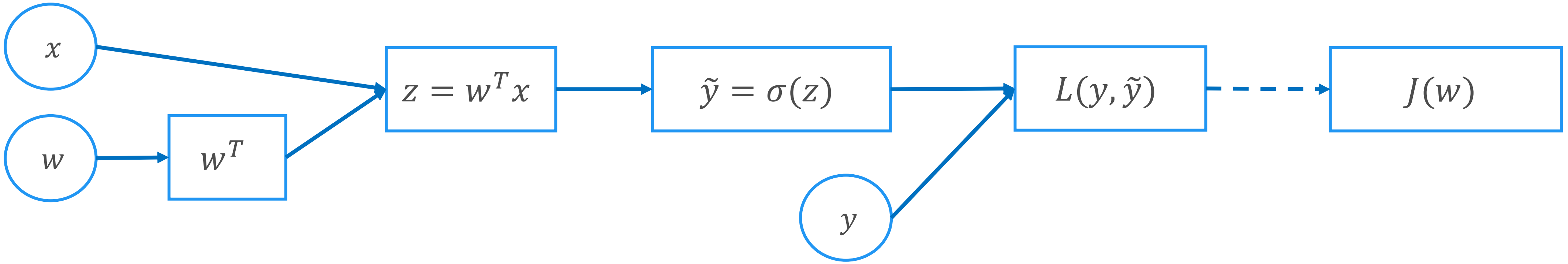
Computation Graphs

- Overview
  - A useful representation of computation sequences
  - Good not only for visualization
  - Almost every compiler / interpreter has some implementation
- Example: logistic regression



- Why is a graph so useful?
  - We need to know the derivatives of the last quantity
    - To compute them, we just need to go back

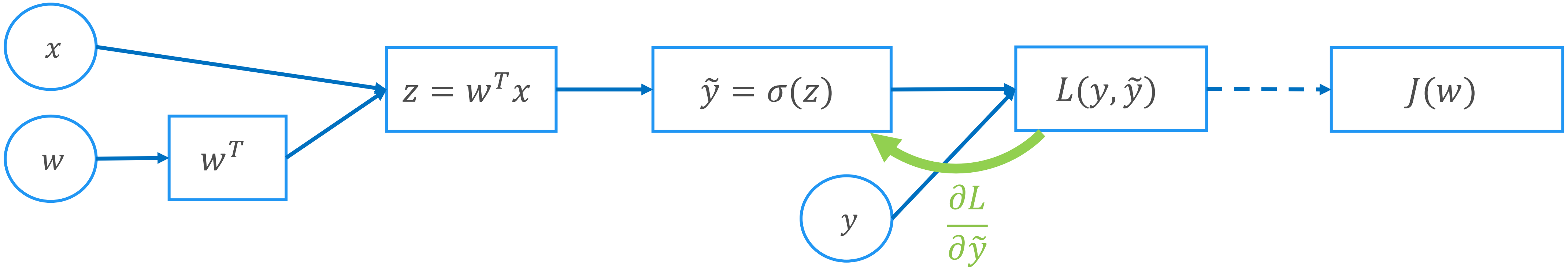
Gradients on Computational Graphs



- Now that we've computed J, we need to perform gradient descent
  - i.e. we need the gradient (derivatives) of J w.r.t. its input variables
  - $J = J(w; x, y)$
  - We don't like to change the data (x, y)
    - $\Rightarrow$ We're only interested in $\frac{\partial J}{\partial w}$
    - In case of many weights:

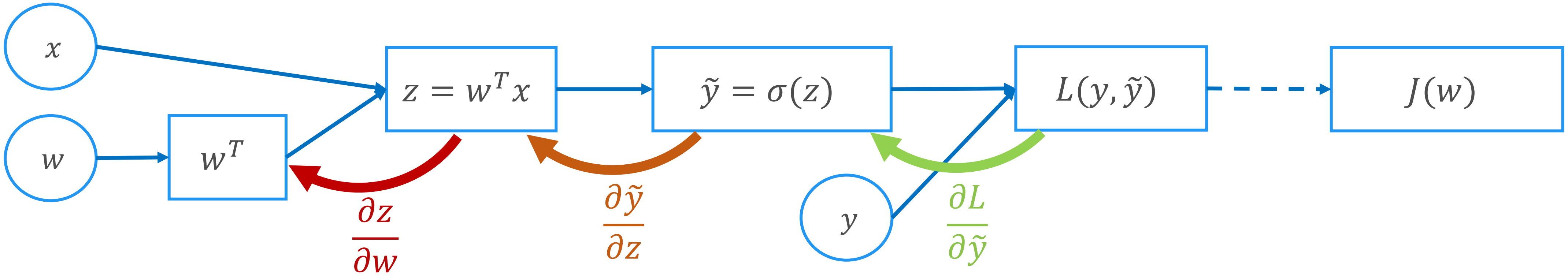
$$\nabla_w J = \left[\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m}\right]^T$$

Gradients on Computational Graphs (2)



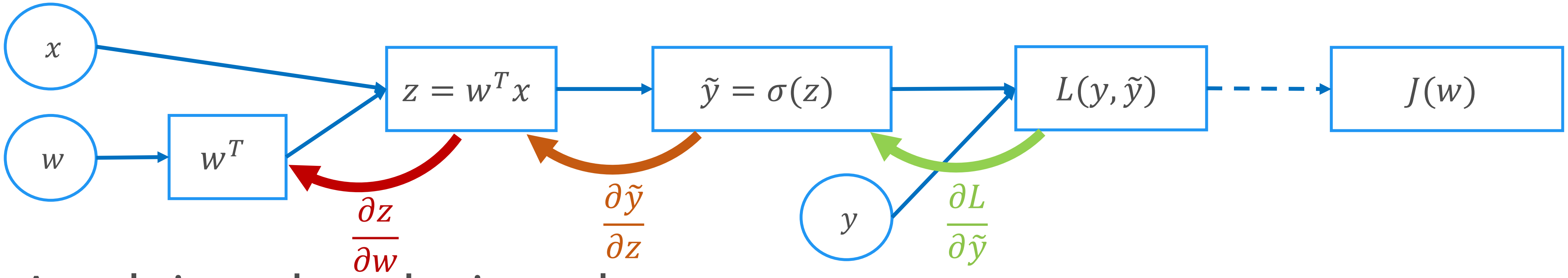
- Solution: Chain rule
  - For the function $f(g(x))$, $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$
- $J = -\frac{1}{n}\sum L^{(i)}$
- $L(y, \tilde{y}) = y \ln(\tilde{y}) + (1 - y) \ln(1 - \tilde{y})$
- $\frac{\partial L}{\partial \tilde{y}} = \frac{y}{\tilde{y}} - \frac{1-y}{1-\tilde{y}}$

Gradients on Computational Graphs (3)



- $\tilde{y}(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
- $\frac{d\tilde{y}}{dz} = \sigma(z)\big(1 - \sigma(z)\big) = \tilde{y}(1 - \tilde{y})$
  - Detailed derivation
- $z(w) = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$
- For any individual weight $w_k, k = 0, 1, 2, \ldots, m$:
  - $\frac{\partial z}{\partial w_k} = x_k$ (for simplicity: $\frac{\partial z}{\partial w} = x$)

Gradients on Computational Graphs (4)



- Applying the chain rule
- $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial z} \frac{\partial z}{\partial w} = \left(\frac{y}{\tilde{y}} - \frac{1-y}{1-\tilde{y}}\right)\left(\tilde{y}(1-\tilde{y})\right)(x) =$

$$= \frac{y(1-\tilde{y}) - \tilde{y}(1-y)}{\cancel{\tilde{y}(1-\tilde{y})}} \cancel{\tilde{y}(1-\tilde{y})} x =$$

$$= (y \cancel{- y\tilde{y}} - \tilde{y} \cancel{+ y\tilde{y}}) x =$$

$$= (y - \tilde{y}) x$$

- $\frac{\partial J}{\partial w} = -\frac{1}{n} \frac{\partial L}{\partial w} = \frac{1}{n} (\tilde{y} - y) x$

Putting It All Together

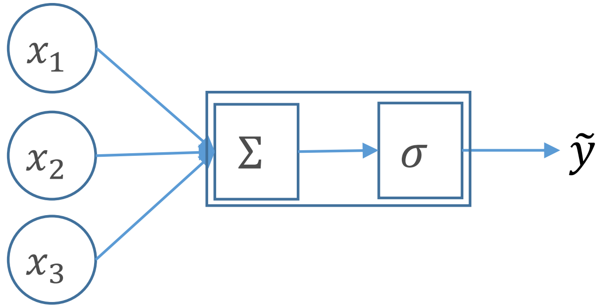
- Forward propagation (left to right)
  - $z = w^T x$
  - $\tilde{y} = \sigma(z)$
  - $L(y, \tilde{y}) = -y \ln(\tilde{y}) + (1 - y) \ln(1 - \tilde{y})$
  - $J = \frac{1}{n} \sum_i L^{(i)}$
- Backpropagation (computing gradients, right to left)
  - $\frac{\partial J}{\partial w} = \frac{1}{n} (\tilde{y} - y) x$
- Gradient updates
  - $w = w - \alpha \frac{\partial J}{\partial w}$

Generalization: Many Examples

- Just work in parallel
  - All training examples at once
  - You can check that the matrix multiplication works out exactly
- $x = [1, x_0, \dots, x_m]^T \Rightarrow X = \{x^{(i)}\}_{m \times n} = [x^{(1)}, x^{(2)}, x^{(3)} \dots, x^{(n)}]$
- $z \Rightarrow Z = [z^{(1)}, z^{(2)}, z^{(3)} \dots, z^{(n)}] = \sigma(w^T X)$
- Warning
  - X contains all variables in **rows**
  - Keep this in mind, it's different than what we're used to seeing
  - This makes computations easier
    - Otherwise, we need too many transpositions and indexing magic

Perceptron

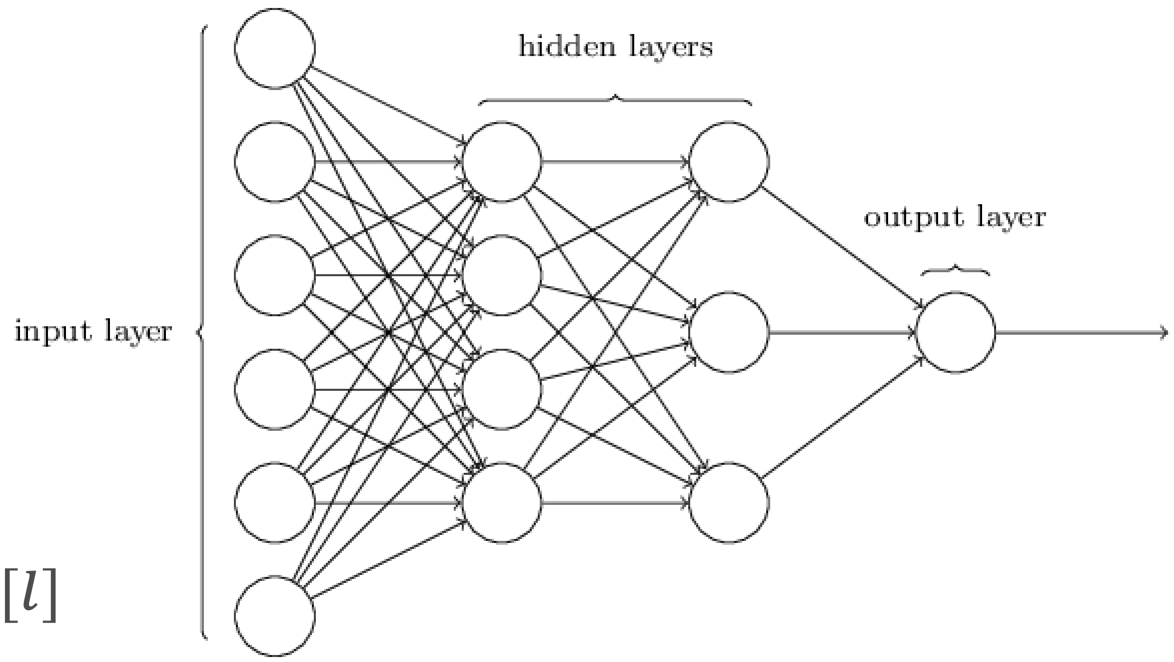
- Logistic regression (forward) at a glance



- In NN terminology, this is called a perceptron
  - The main NN unit
  - The result $Z = w^T X$ is called **activation**
  - $\sigma(Z)$ is called **activation function**
    - May be something else than sigmoid
    - Usually, we use other activations in the "middle" and sigmoid at the output layer
  - Many of those form a neural network layer

Neural Network

- Layers
  - Input layer
  - Hidden layers
  - Output layer
- Each layer has some number of perceptrons: $n^{[l]}$
- The perceptrons in one layer are **fully connected to the next**
- There are **no connections within a layer**



Neural Network Implementation

- For each layer, compute several instances of regression with the chosen activation function
  - Sigmoid in this case
- Vectorize for the entire layer
  - I.e., compute all logistic regressions at once
  - Don't forget to augment the input with bias terms
  - Using our convention
    - Each layer l has $m^{[l-1]}$ inputs (+ 1 bias term), $m^{[0]} = m + 1$
    - Each layer l has $m^{[l]}$ outputs
    - Therefore, each weight matrix will be W $\{m^{[l]} \times m^{[l-1]} + 1\}$
- Don't initialize W with zeroes!
- Don't forget the activation function!

Neural Network Implementation (2)

- Define layer sizes: $[m^{[0]} = m, m^{[1]}, m^{[2]}, \dots, m^{[L]}]$
- Initialize weights randomly: $[w^{[0]}, \dots, w^{[L]}]$, with dimensions $m^{[l]} \times m^{[l-1]} + 1$
- Forward (input activation $a^{[l-1]}$)
  - For each layer $l \in \{1, 2, \dots, L\}$
    - Augment the input activation so that it has dimensions $n \times m^{[l-1]} + 1$
    - Compute the linear combination $Z^{[l]} = w^{[l]} a^{[l-1]}$, cache it
    - Compute the activation $A^{[l]} = g(Z^{[l]})$
- Backward (input gradient $\frac{\partial L}{\partial Z^{[l]}}$)
  - For each layer $l \in \{1, 2, \dots, L\}$
    - Compute the gradients $\frac{\partial J}{\partial w^{[l]}} = \frac{1}{n} dZ^{[l]} \mathrm{A}^{[l-1]T}$, update $w^{[l]} = w^{[l]} - \alpha \frac{\partial J}{\partial w^{[l]}}$

Extensions

- There are a lot of things we can (and will) do
  - Add regularization
  - Use a better / different optimization algorithm
  - Tune hyperparameters
  - Deal with vanishing / exploding gradients
  - Reuse computations
  - …
- Regression
  - Omit the output activation, take the raw output
- Many classes / output values
  - Use many output neurons ($n^{[L]} > 1$)
- I recommend that you try to implement this yourself
  - It's quite hard but useful once you do it

Summary

- Neural networks
  - Overview
  - Problem statement
- Pros and cons
- Perceptron
- Feed-forward NNs (multi-layer perceptrons)
  - Training
  - Applications for classification

Questions?