Data mining and machine learning

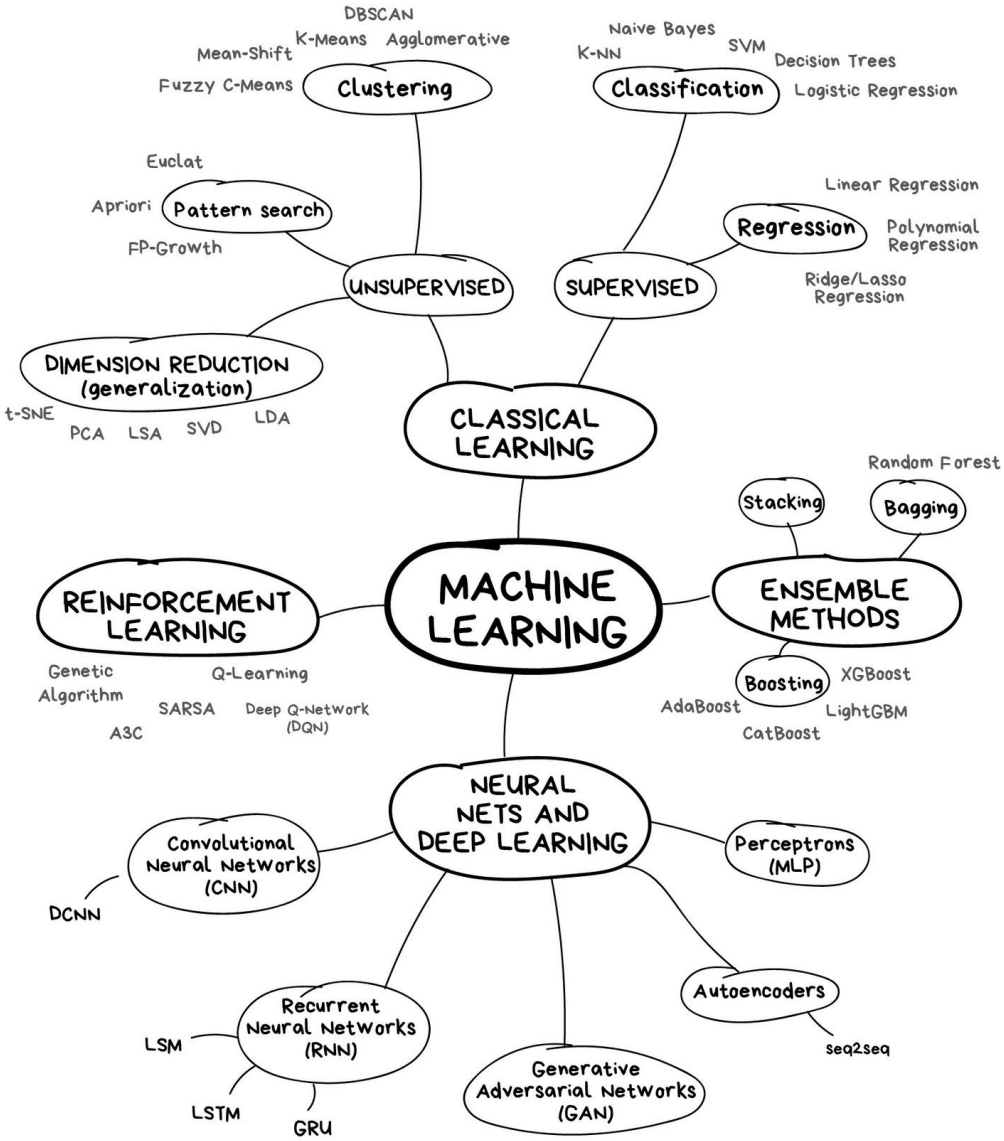
dsminingf17vm

Physics MSc course

09 - Neural networks

Pataki Bálint Ármin
ELTE, Physics of Complex Systems Department
2020.11.09.

Machine learning models - there is no single best one!

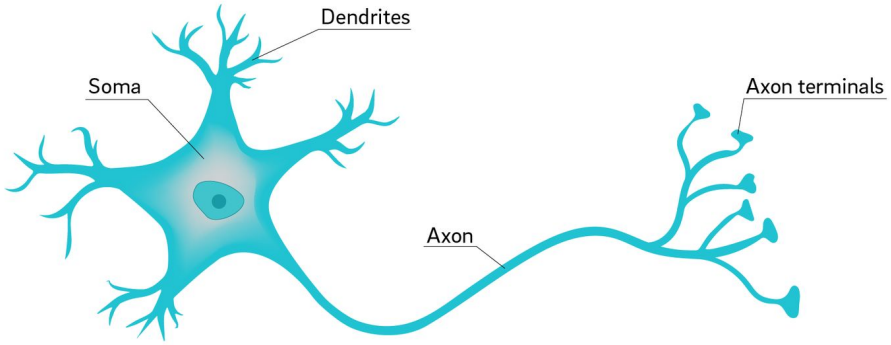


http://valyrics.vas3k.com/blog/machine_learning/

Motivation from biology - a neuron

Biology:

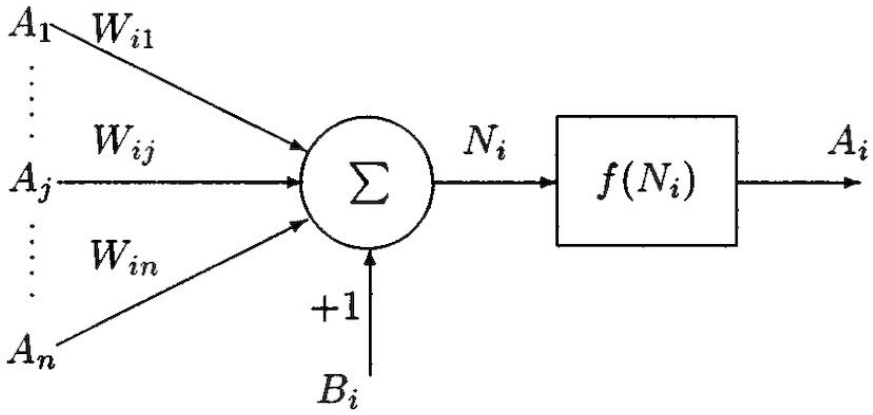
- Input from other neurons (dendrites)
- Summing them (soma)
- Based on the sum fire or not
- Pass the output (firing or not) on axon



David Baillot/ UC San Diego

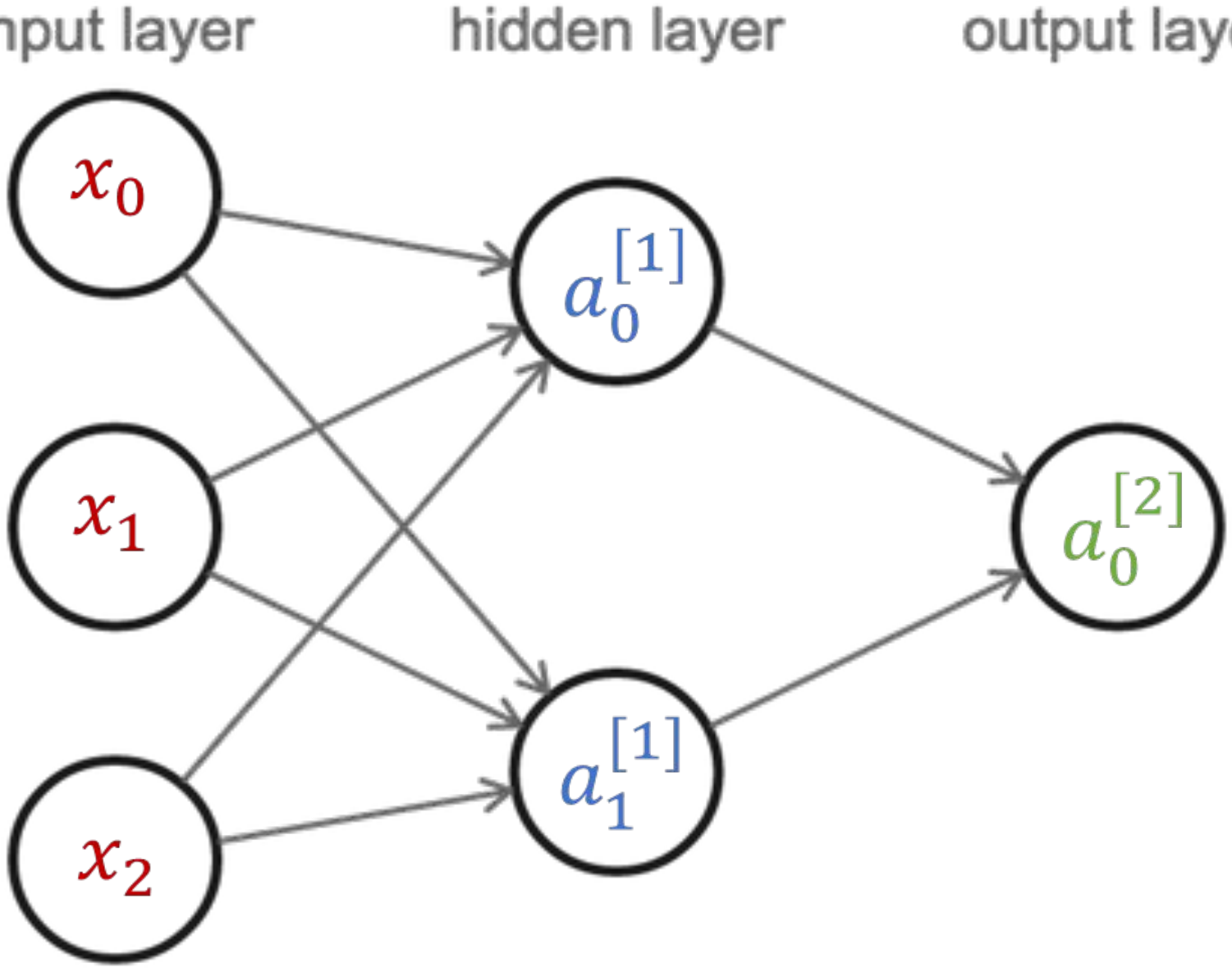
Model:

- Inputs are the activation form others (X)
- Weighted sum of inputs + bias (B)
- Activation function (f or g)
- Output = activation: A = g(w*x + b)

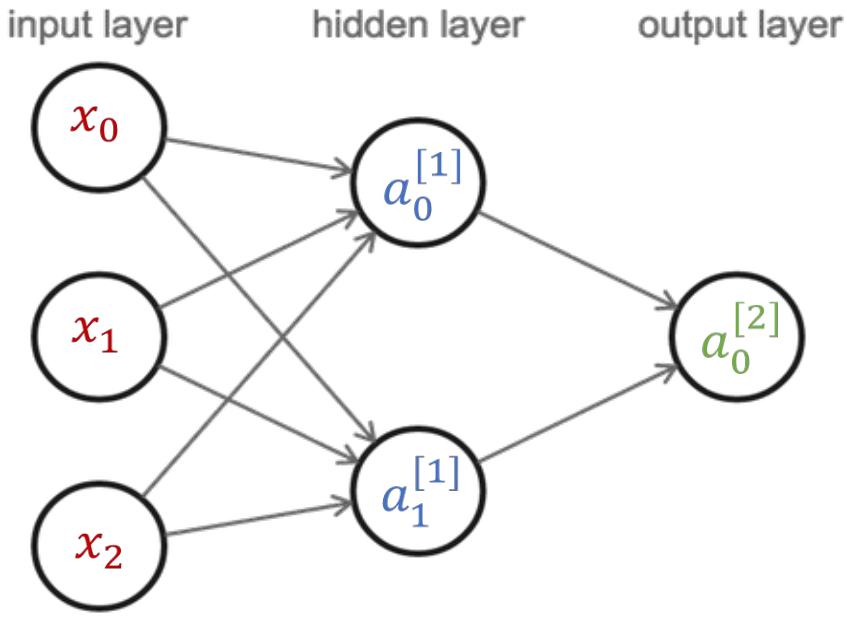


Yang et el, 2000. Constraint Satisfaction Adaptive Neural Network and Heuristics Combined Approaches for Generalized Job-Shop Scheduling

2-layer fully connected neural network - step-by-step



2-layer fully connected neural network - step-by-step

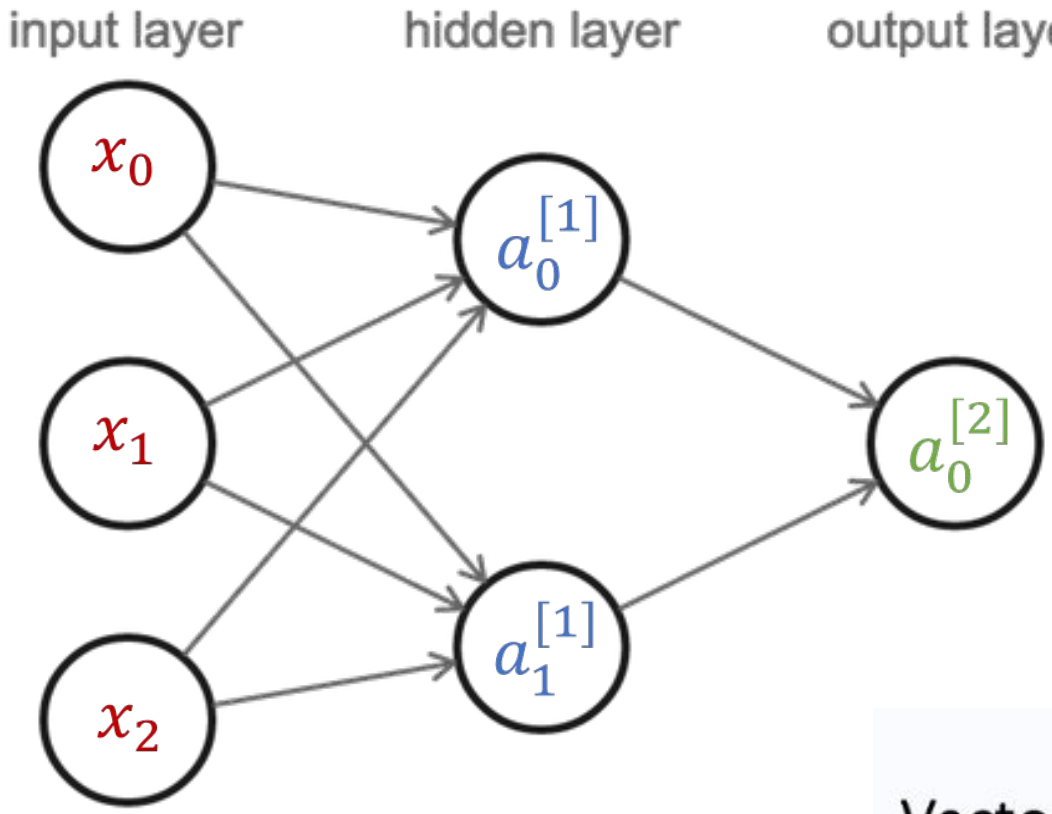


$$z_0^{[1]} = w_{0,0}^{[1]} x_0 + w_{0,1}^{[1]} x_1 + w_{0,2}^{[1]} x_2 + b_0^{[1]}$$
$$a_0^{[1]} = g\left(z_0^{[1]}\right)$$

$$z_1^{[1]} = w_{1,0}^{[1]} x_0 + w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_0^{[1]}$$
$$a_1^{[1]} = g\left(z_1^{[1]}\right)$$

$$z_0^{[2]} = w_{0,0}^{[2]} a_0 + w_{0,1}^{[2]} a_1 + b_0^{[2]}$$
$$a_0^{[2]} = g\left(z_0^{[2]}\right)$$

2-layer fully connected neural network - step-by-step



$$z_0^{[1]} = w_{0,0}^{[1]} x_0 + w_{0,1}^{[1]} x_1 + w_{0,2}^{[1]} x_2 + b_0^{[1]}$$
$$a_0^{[1]} = g\left(z_0^{[1]}\right)$$

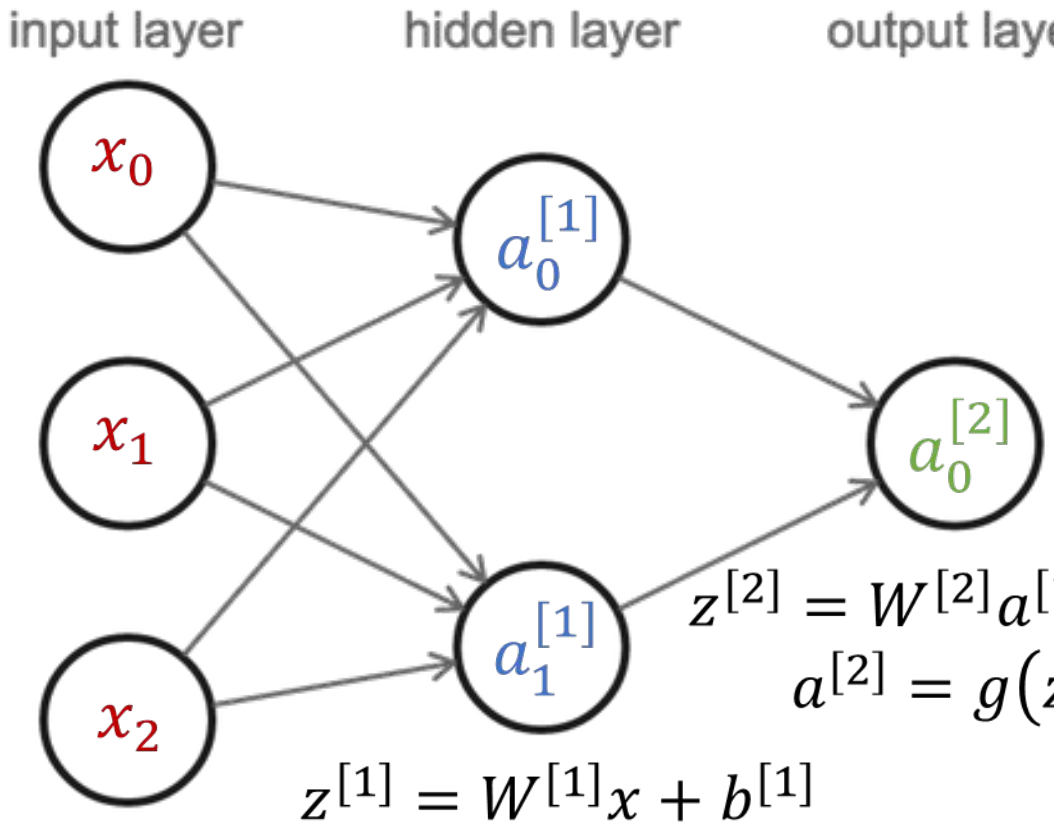
$$z_1^{[1]} = w_{1,0}^{[1]} x_0 + w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_0^{[1]}$$
$$a_1^{[1]} = g\left(z_1^{[1]}\right)$$

$$z_0^{[2]} = w_{0,0}^{[2]} a_0 + w_{0,1}^{[2]} a_1 + b_0^{[2]}$$
$$a_0^{[2]} = g\left(z_0^{[2]}\right)$$

Vectorization: $x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$, $a^{[1]} = \begin{bmatrix} a_0^{[1]} \\ a_1^{[1]} \end{bmatrix}$

$$W^{[1]} = \begin{bmatrix} w_{0,0}^{[1]} & w_{0,1}^{[1]} & w_{0,2}^{[1]} \\ w_{1,0}^{[1]} & w_{1,1}^{[1]} & w_{1,2}^{[1]} \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} b_0^{[1]} \\ b_1^{[1]} \end{bmatrix}$$

2-layer fully connected neural network - step-by-step



$$z_0^{[1]} = w_{0,0}^{[1]} x_0 + w_{0,1}^{[1]} x_1 + w_{0,2}^{[1]} x_2 + b_0^{[1]}$$
$$a_0^{[1]} = g\left(z_0^{[1]}\right)$$

$$z_1^{[1]} = w_{1,0}^{[1]} x_0 + w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_0^{[1]}$$
$$a_1^{[1]} = g\left(z_1^{[1]}\right)$$

$$z_0^{[2]} = w_{0,0}^{[2]} a_0 + w_{0,1}^{[2]} a_1 + b_0^{[2]}$$
$$a_0^{[2]} = g\left(z_0^{[2]}\right)$$

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g\left(z^{[2]}\right)$$

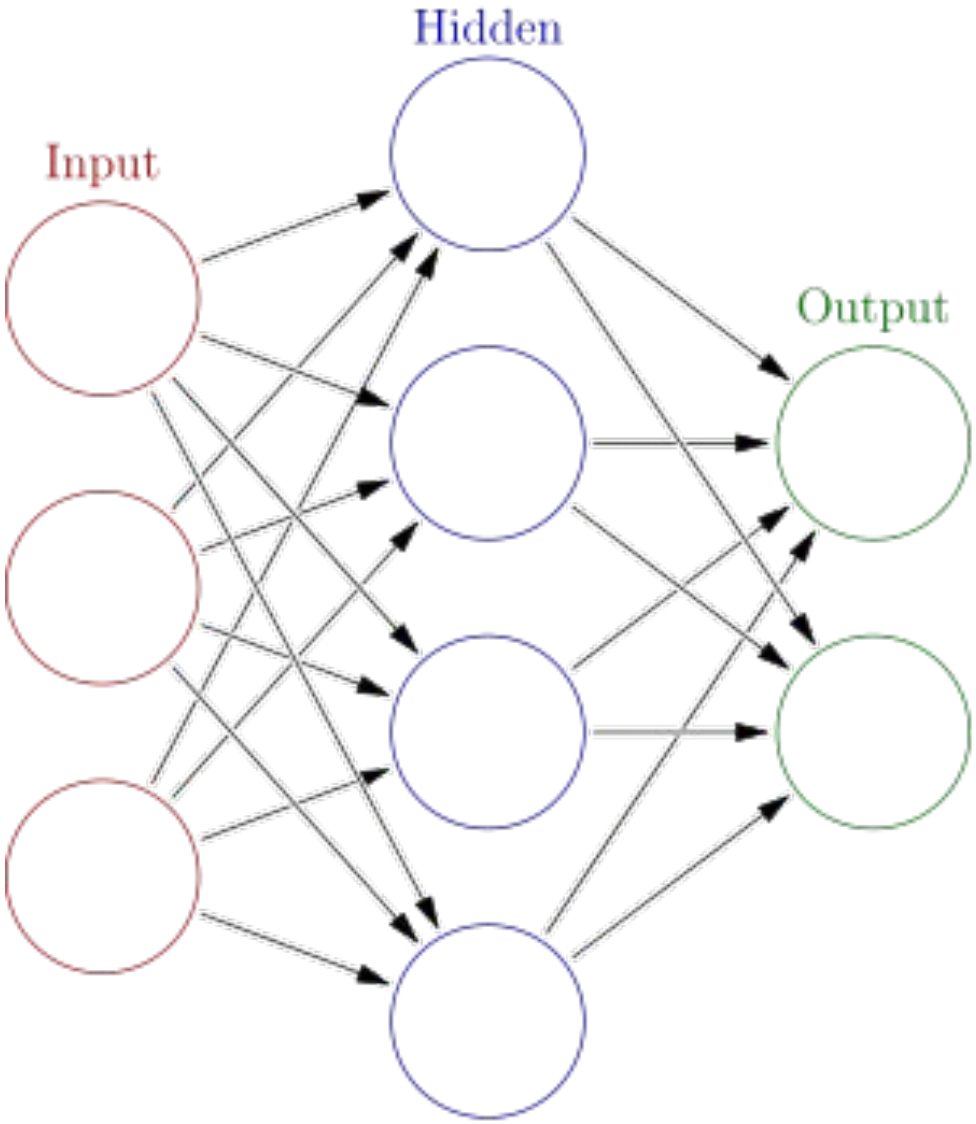
$$z^{[1]} = W^{[1]} x + b^{[1]}$$
$$a^{[1]} = g\left(z^{[1]}\right)$$

Vectorization: $x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$, $a^{[1]} = \begin{bmatrix} a_0^{[1]} \\ a_1^{[1]} \end{bmatrix}$

$$W^{[1]} = \begin{bmatrix} w_{0,0}^{[1]} & w_{0,1}^{[1]} & w_{0,2}^{[1]} \\ w_{1,0}^{[1]} & w_{1,1}^{[1]} & w_{1,2}^{[1]} \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} b_0^{[1]} \\ b_1^{[1]} \end{bmatrix}$$

One will not be enough - neural network

- Fully connected neural network
- Input: data data itself
- Output: the prediction
- Hidden: everything between



- Activation functions (details later)
  - ReLU
  - Softmax
  - Tanh
  - Sigmoid

- We need non-linear activation function!
  - **Why?**

L-layer fully connected neural network

$$z^{[1]} = W^{[1]}x + b^{[1]}, \quad W: n^{[1]} \times N, \quad b: n^{[1]} \times 1$$
$$a^{[1]} = g(z^{[1]})$$

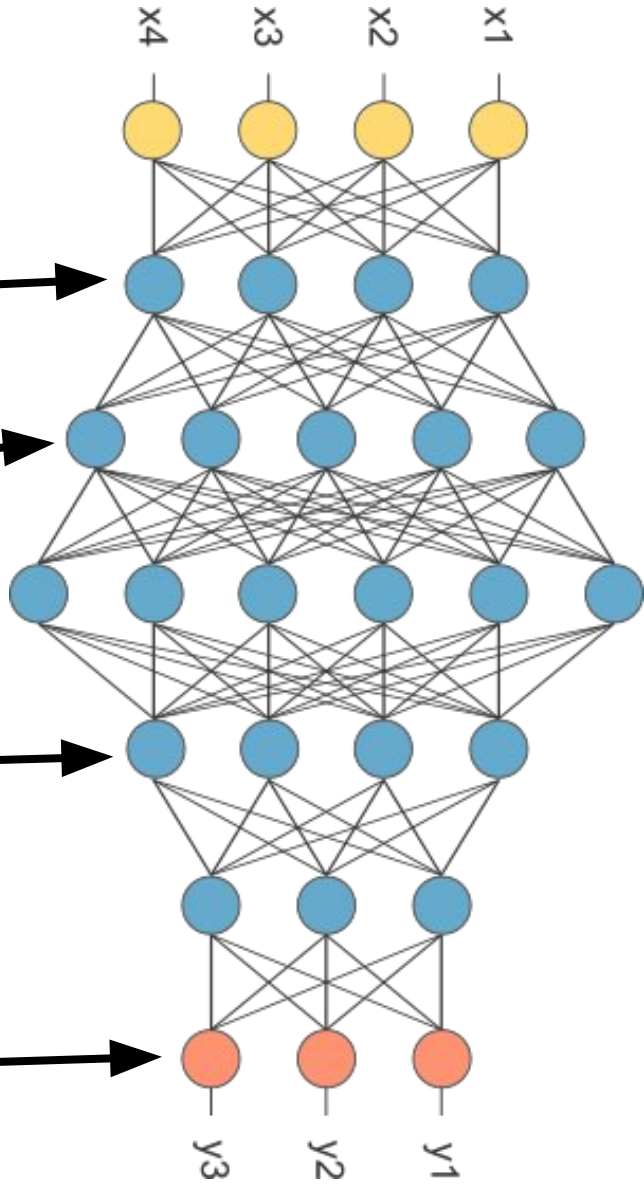
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, \quad W: n^{[2]} \times n^{[1]}, \quad b: n^{[2]} \times 1$$
$$a^{[2]} = g(z^{[2]})$$

...a few more layers

$$z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}, \quad W: n^{[i]} \times n^{[i-1]}, \quad b: n^{[i]} \times 1$$
$$a^{[i]} = g(z^{[i]})$$

...a few more layers

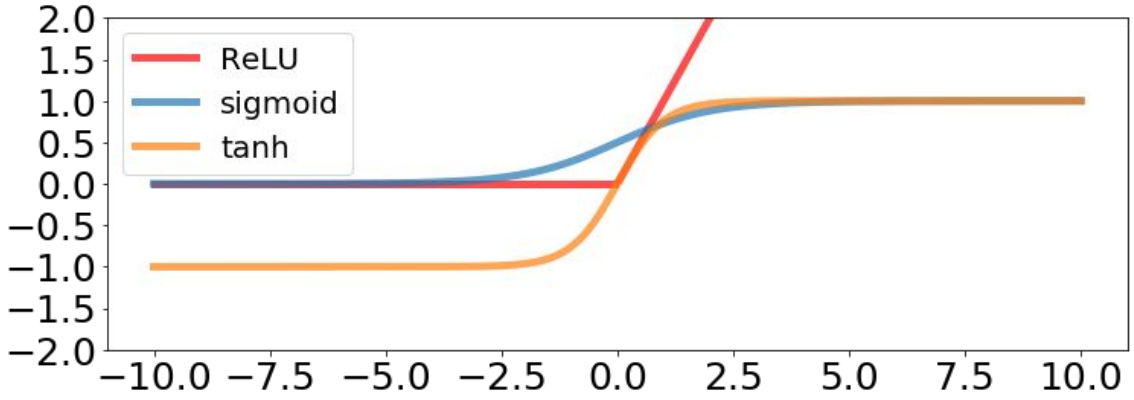
$$z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}, \quad W: n^{[L]} \times n^{[L-1]}, \quad b: n^{[L]} \times 1$$
$$y = a^{[L]} = g(z^{[L]})$$



Online neural network playground: https://playground.tensorflow.org/

Neural network details

- A fully connected neural network with one hidden layer
  - Can approximate any continuous function
  - If there are enough neurons in the hidden layer
- Visual support: http://neuralnetworksanddeeplearning.com/chap4.html



- Activation functions
  - Linear - g(z) = x
  - **ReLU (rectified linear unit) - g(z) = max(0, z) ← most popular**
  - Sigmoid - $g(z) = \frac{1}{1 + e^{-z}}$
  - Tanh - g(z) = tanh(z)
  - **Softmax ← for the last classification layer**
- Pros & cons

Softmax

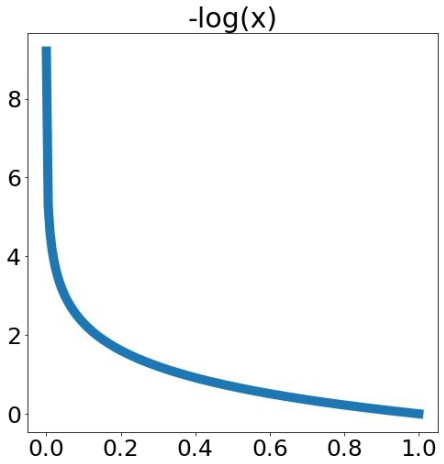
- For classification at the last layer we want probabilities
  - Probabilites > 0
  - Probabilities sum to 1 by definition
  - $z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$ does not know any of them
  - Softmax: $g(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} \rightarrow \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ e^{z_3} \\ e^{z_4} \end{bmatrix} \rightarrow \begin{bmatrix} \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} \\ \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} \\ \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} \\ \frac{e^{z_4}}{e^{z_1}+e^{z_2}+e^{z_3}+e^{z_4}} \end{bmatrix}$$

$$\begin{bmatrix} -0.7 \\ 1.2 \\ 2.4 \\ 0.6 \end{bmatrix} \rightarrow \begin{bmatrix} 0.50 \\ 3.32 \\ 11.02 \\ 1.82 \end{bmatrix} \rightarrow \begin{bmatrix} 0.03 \\ 0.20 \\ 0.66 \\ 0.11 \end{bmatrix} = \begin{bmatrix} p_{horse} \\ p_{cat} \\ p_{dog} \\ p_{human} \end{bmatrix}$$

Loss functions

Minimizing this function we expect to gain an accurate model.



- Measures how good we are, differentiable, continuous
- Classification labels are one-hot encoded ([0, 0, 0, 1])
- Regression
  - Mean absolute error (MAE)
    - More robust to outliers
  - Mean squared error (MSE)
    - Less robust to outliers
  - Log-cosh loss
    - like MAE when error is large, like MSE when error is small

- Classification
  - **Binary**
    - y_i is the actual class (0 or 1)
    - $\hat{y}_i$ is the predicted probability

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)$$

  - **Multi class** (eg dog vs car vs horse vs human)
    - y_{ic} is 1 if the sample *i* belongs to class *c*, 0 otherwise
    - $\hat{y_{ic}}$ is the predicted probability for sample *i* and class *c*
    - Sum of the predicted probabilities should be 1 for each sample!

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic} \cdot log(\hat{y_{ic}})$$

How to train the model?

- Want to minimize the loss wrt **w** and **b**.

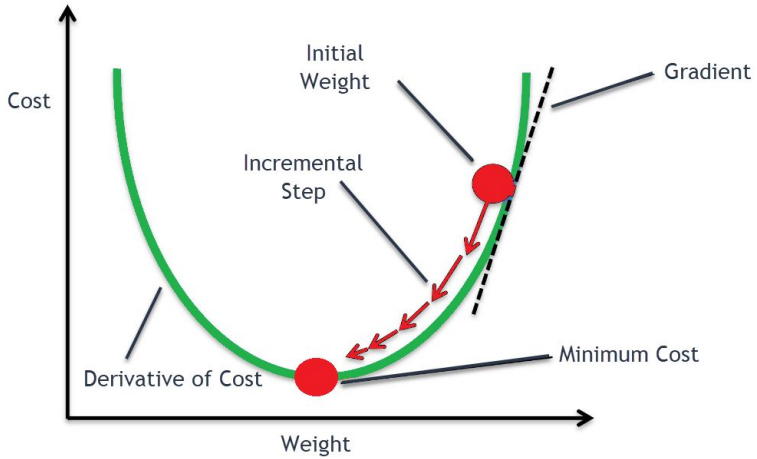
$$\underset{w,b}{\operatorname{argmin}} L(w, b)$$

- One solution $\frac{\partial L}{\partial w} = 0, \frac{\partial L}{\partial b} = 0$
  - We do not know L(w, b)
- Instead: gradient descent
  - Repeat for N times

$$w = w - \alpha \frac{\partial L}{\partial w}$$

$$b = b - \alpha \frac{\partial L}{\partial b}$$

α is the so-called learning rate



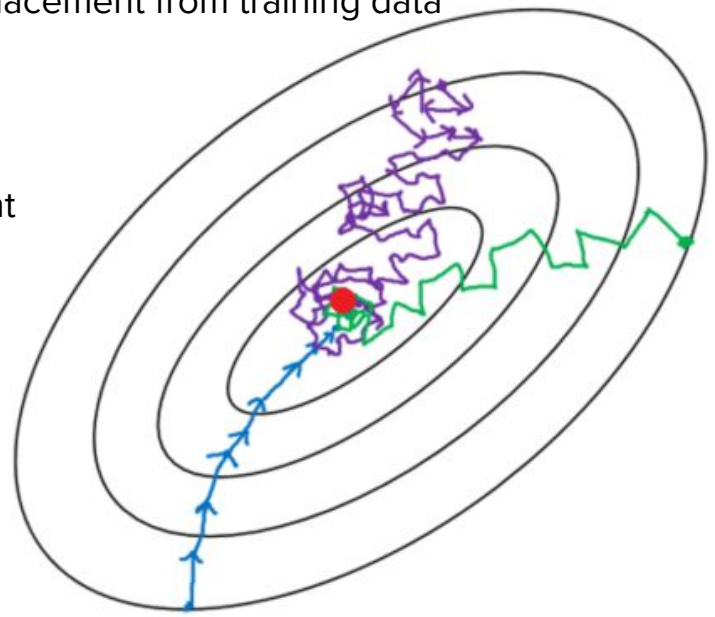
https://kraj3.com.np/blog/2019/06/introduction-to-gradient-descent-algorithm-and-its-variants/

How to train the model?

1. Randomly initialize weights
    a. Constant initialization is bad (symmetry breaking)
2. Calculate prediction for inputs (forward step)
3. Calculate loss based on predictions and the labels
    a. One-hot encoded labels (eg class 3 is $y_3 = [0, 0, 0, 1, 0]$, class 4 is $y_4 = [0, 0, 0, 0, 1]$)
4. **Calculate gradients of the loss wrt weights → details in 11th lecture**
5. Update weights accordingly
6. Goto 2

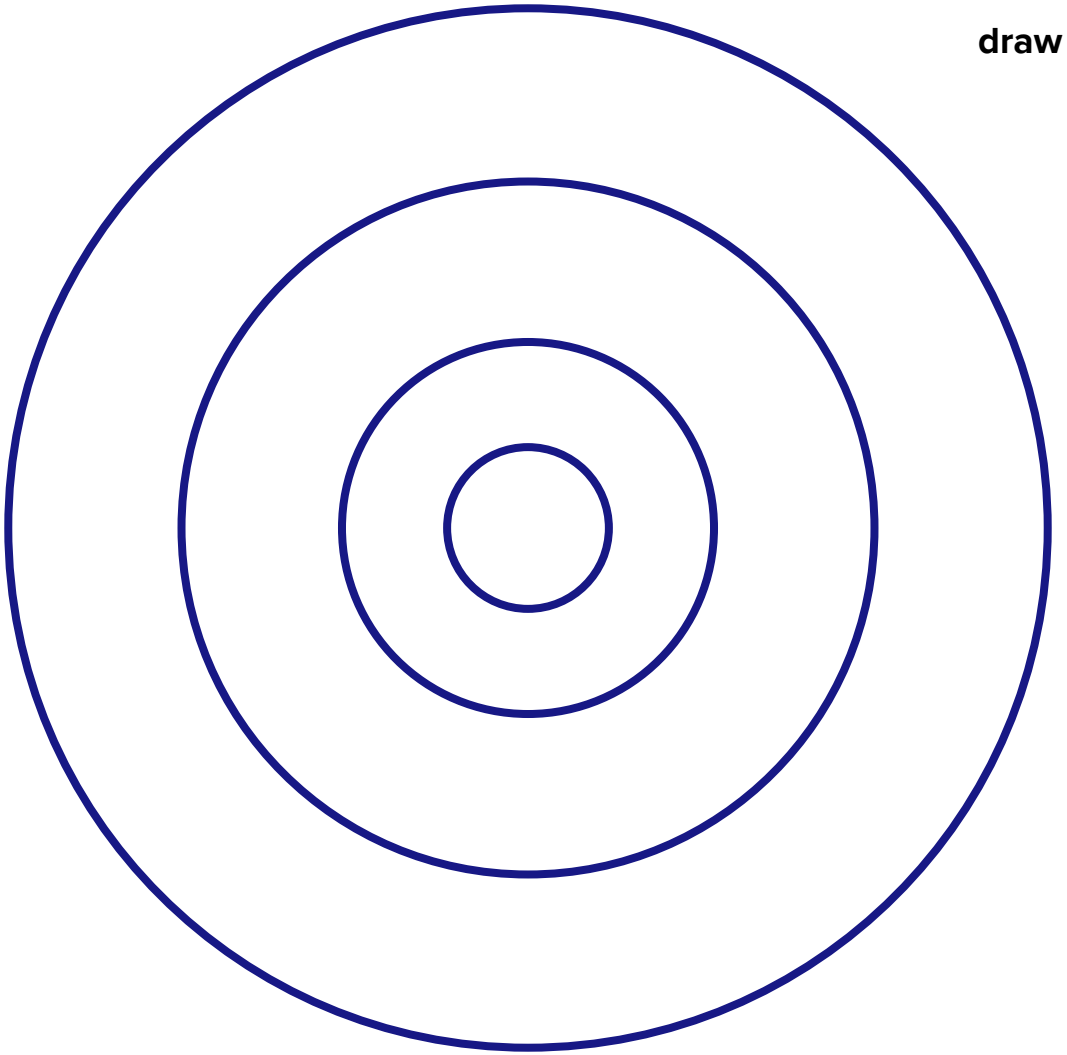
Stochastic gradient descent (SGD)

- We want to minimize L over the whole training set
- How frequent should we update weights?
  - Calculate gradients & update weight after iterating over all the data?
    - Way too slow, 1 update after iterating over all the data
  - Calculate gradients & update weights after each images?
    - Quick, but noisy
  - Something in-between
    - Sample batch_size samples randomly without replacement from training data
      - Calculate gradients for each separately
      - Average over them
      - Update the weight with this average gradient



https://suniljangirblog.wordpress.com/2018/12/13/variants-of-gradient-descent/

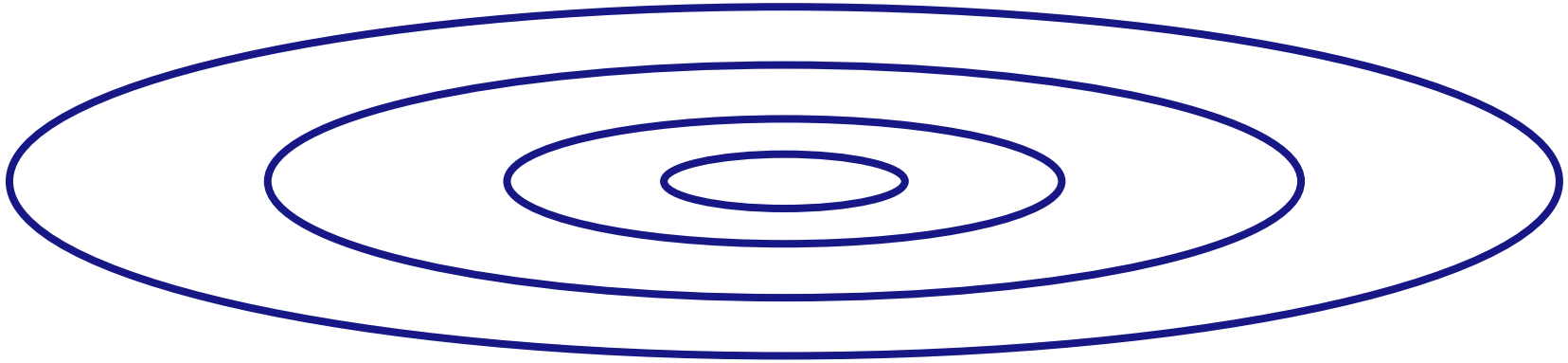
If the surface is symmetric



If the surface is asymmetric

Struggle, to many steps to reach minima

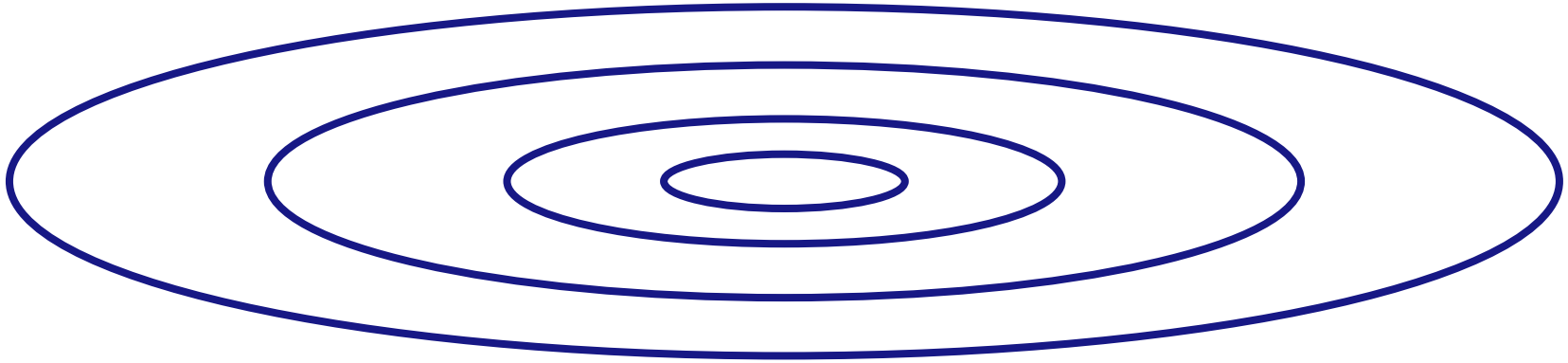
draw



If the surface is asymmetric

Something like this would be better

draw



Adam (adaptive momentum) optimizer

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

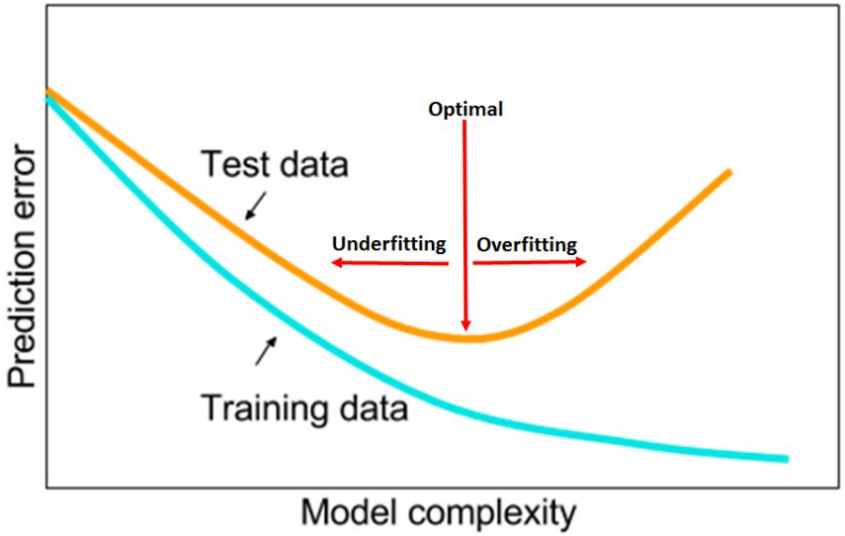
Kingma et al, ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION, 2015

Strategies, tips

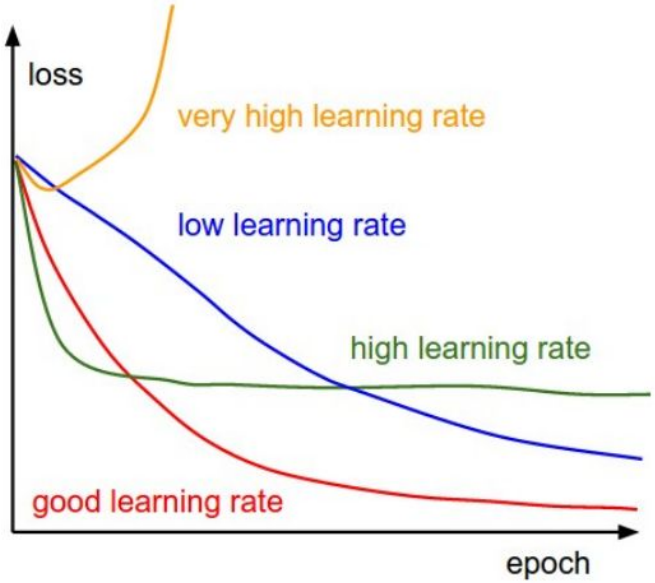
- Baby sitting vs brute-force hyperparameter tuning
  - Based on your hardware limitation
- Learning rate schedule
  - At the beginning larger learning rate
  - Later smaller
- Ensemble prediction
  - Train different networks (VGG16 & ResNet)
  - Average their prediction
- Visualization ➜ loss, accuracy etc...

learning rate

$$w = w - \alpha \frac{\partial L}{\partial w}$$

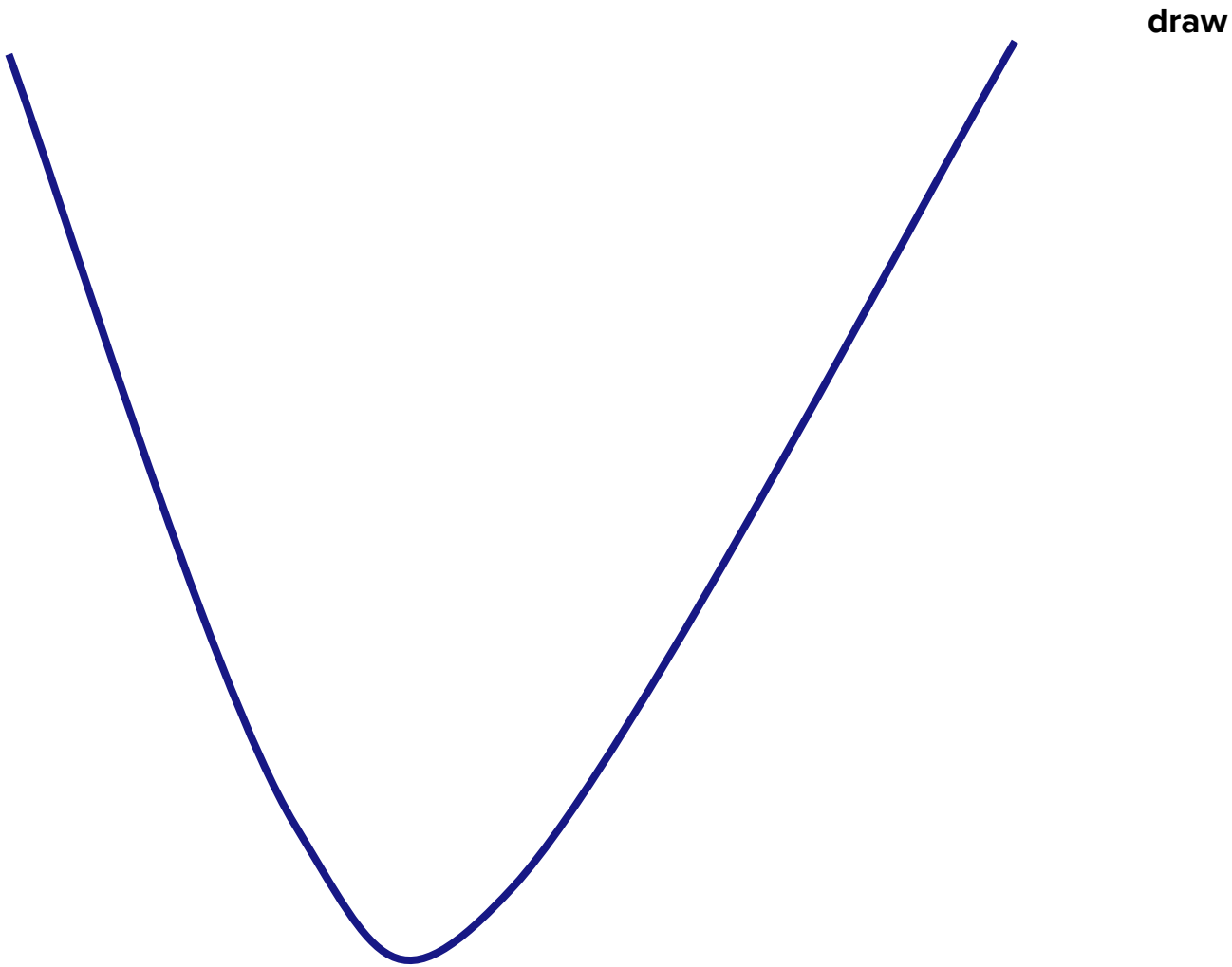


L.N. Smith, 2018

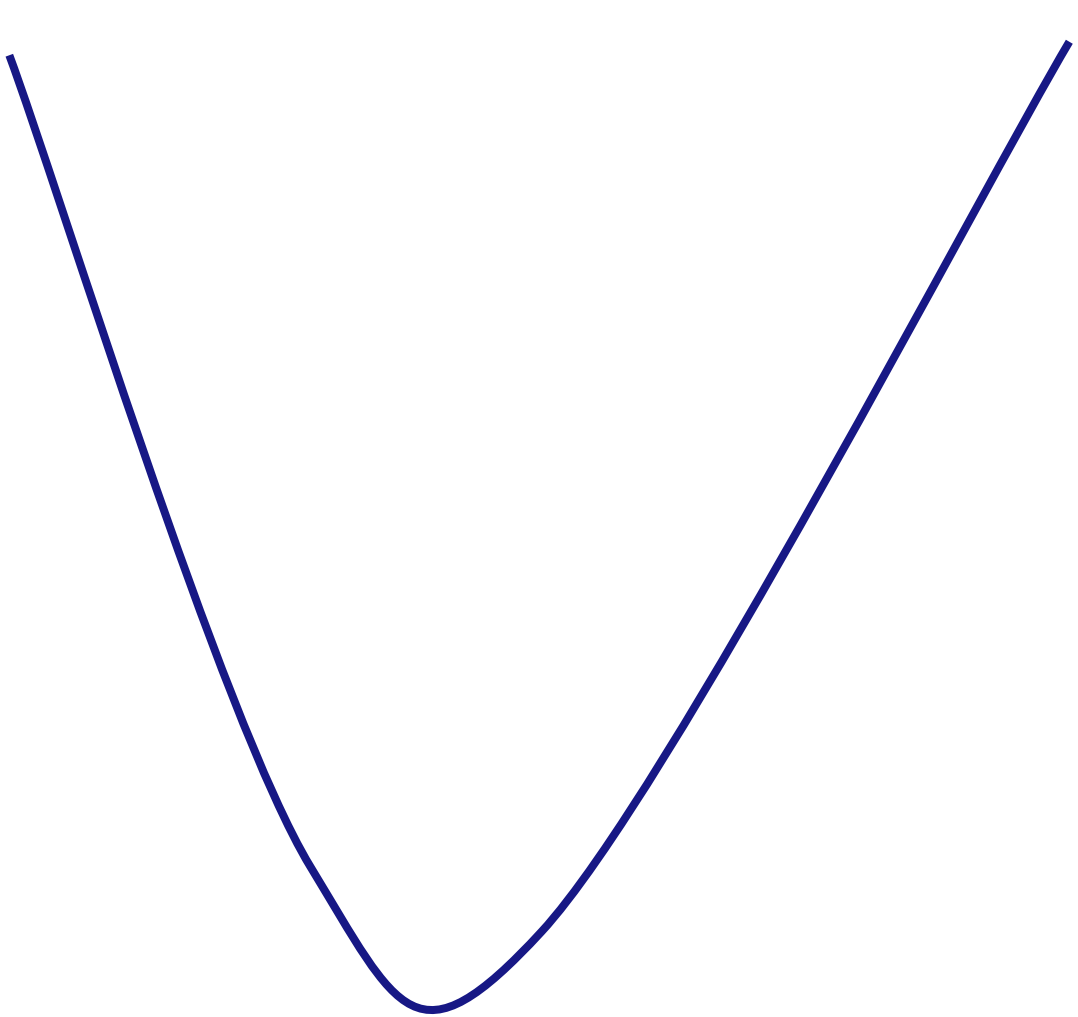


Effect of various learning rates on convergence (Img Credit: cs231n)

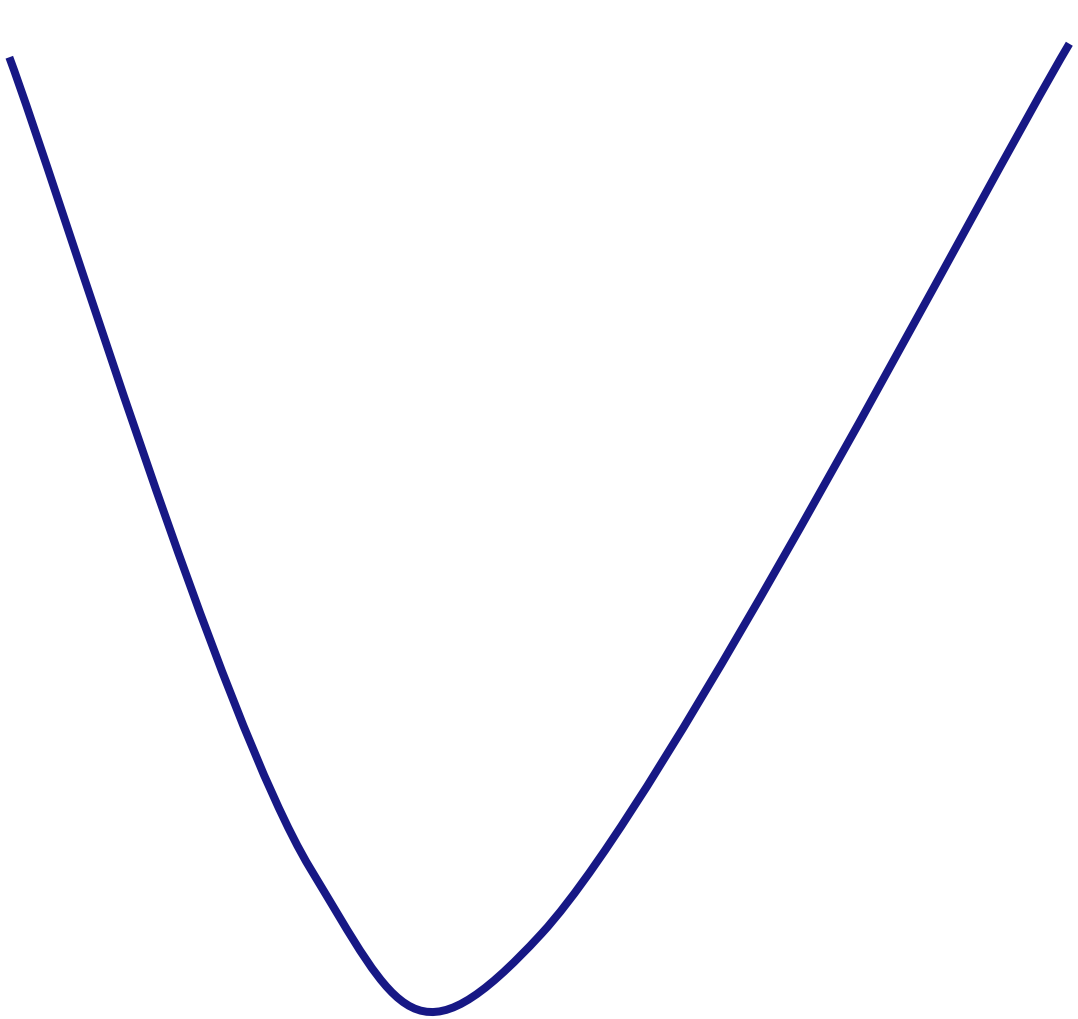
Learning rate - too large



Learning rate - too small

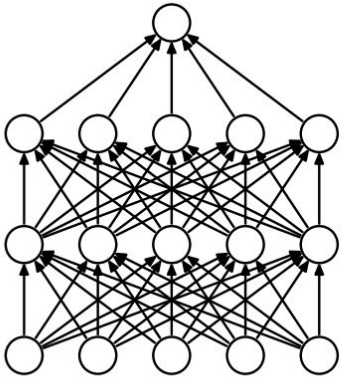


Learning rate - start larger & then smaller

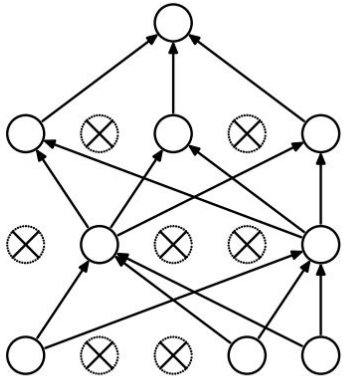


Dropout

- Complex model ➔ can overfit
  - Memorize train data
  - Perform badly on test data
  - Cannot train more, as it is perfect for train
- Dropout
  - During train ➔ randomly 0 out a few activation
  - During test ➔ turn off ➔ no removing of activation!
  - Can help close the train-test performance gap
- Idea
  - Network can not memorize the data easily
  - Need to stand on multiple legs
- Not so popular now (it was a few years ago)



(a) Standard Neural Net



(b) After applying dropout.

Srivastava et al, 2014:Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Assignment 9

On kooplex tensorflow 2 does not work, please work on Google Colab and then upload your solution to kooplex! Also, on Google Colab you can use GPUs

This week we will use the MNIST handwritten digits dataset! The weights.npy file is provided, which contains the weight vector for a traines fully connected neural network.

1 - 2. Implement fully connected neural network via using only numpy

In this task we need to implement a small fully connected neural network that can generate predictions for us if we provide the weights and the input data!

- implement the following function:

```
def pred_nn(weights, x_test):
 ...
 return predictions
```

- x_test has a shape of (N_samples, 784)
- predictions has a shape of (N_samples, 10)
- then function implements a fully connected neural network with the follwing layers:
  - 750 neuron, relu activation
  - 500 neuron, relu activation
  - 500 neuron, relu activation
  - 10 neuron, softmax activation
- weights is a numpy array of the weights
  - 1st element is a shape of (784, 750), 2nd is (750,), the bias
  - 3rd element is a shape of (750, 500), 4th is (500,) ... the rest matches the weight dimensions of the above-mentioned architecture
- use numpy's built-in vectorized operations, try not to write for loops!

An optimally implemented function runs < 1s for N_samples = 10.000

3. Same architecture via tensorflow/keras

- Implement the same architecture with tensorflow/keras as we did in 1-2).
- Load the provided weights for the neural network!

4-5. Compate performances

- load the MNIST dataset from the tensorflow/keras built-in dataset
- use the original train/test split!
- divide each pixel's value by 255 & reshape to have 1D input vector (784) instead of the 2D matrix (28x28)
  - eg for the test set you will have a (10000, 784) shaped vector
- generate prediction for the 10.000 test images with both methods!
- calculate the categorical cross-entropy loss and the accuracy for both methods! are they the same? (if not, it indicates a bug somewhere...) Hint: you should get ~97% accuracy
- show the confusion matrix of the predictions (predicted values vs actual labels)
- where does the model make mistakes?