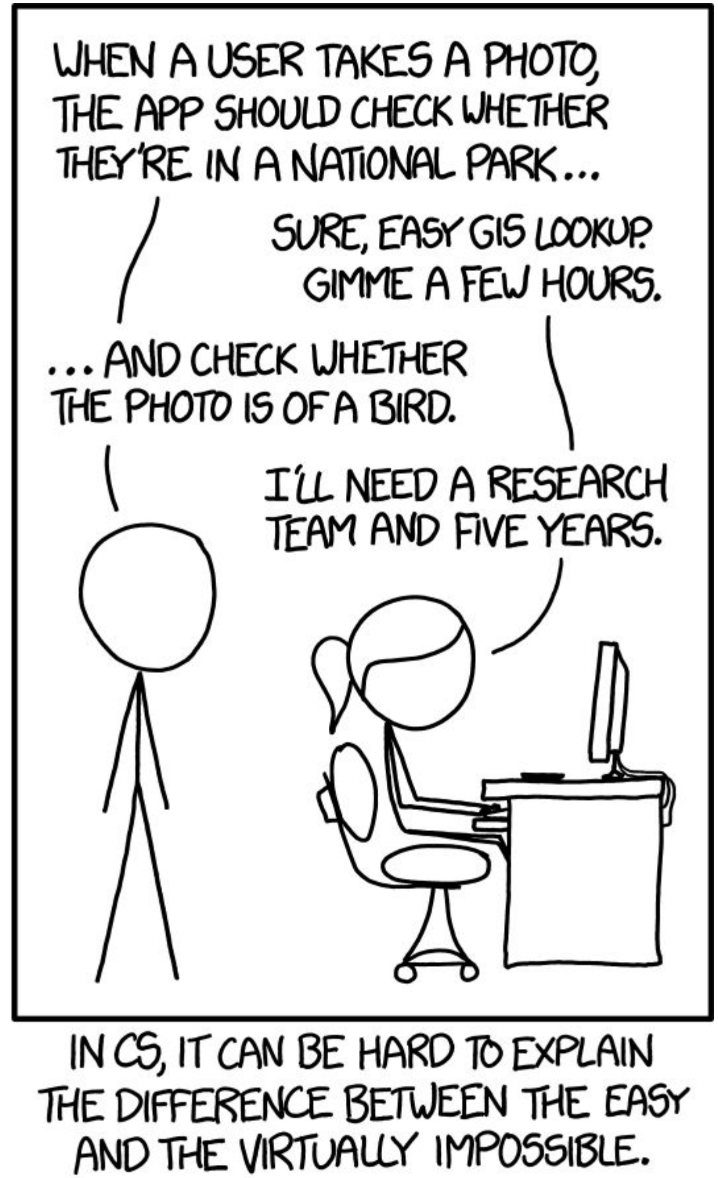
Intro to Artificial Neural Networks

Oscar Mañas
@oscmansan

talks

Outline

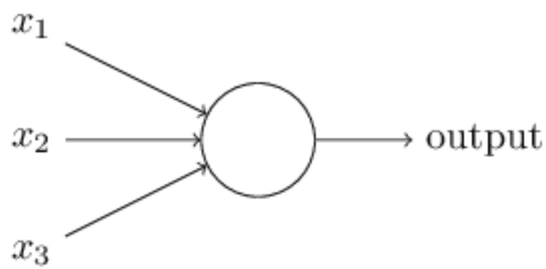
1. Perceptrons
2. Sigmoid Neurons
3. Linear vs Nonlinear Classification
4. Neural Networks Anatomy
5. Training and loss
    a. Reducing loss
    b. Gradient Descent
    c. Learning rate
    d. Stochastic Gradient Descent
    e. Backpropagation
6. Convolutional Neural Networks
7. Why GEMM is at the heart of Artificial Neural Networks?
8. References
9. Hands on



https://xkcd.com/1425/

Perceptrons

- A perceptron is a function that takes several binary inputs, *x1,x2,…*, and produces a single binary output:



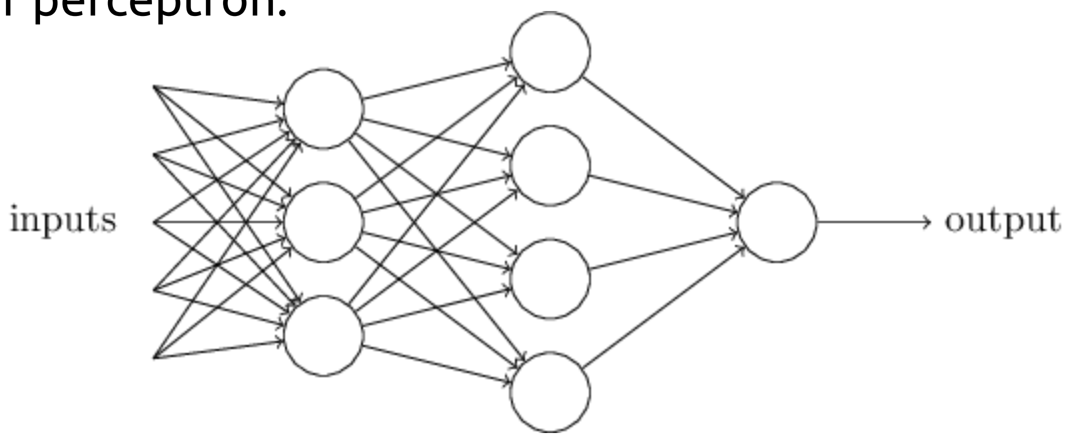
- Weights, *w1,w2,…*, are real numbers expressing the importance of the respective inputs to the output.

- The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold* value:

$$\begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Perceptrons

- Multi-layer perceptron:



- We can write $\sum_j w_j x_j$ as a dot product, $w \cdot x \equiv \sum_j w_j x_j$, and replace the threshold by a *bias*, $b \equiv -\text{threshold}$. The perceptron rule can be rewritten as:

$$\begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Sigmoid Neurons

- The sigmoid neuron has inputs, *x1,x2,…*. But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1.
- Just like a perceptron, the sigmoid neuron has weights for each input, *w1,w2,…*, and an overall bias, *b*.
- But the output is not 0 or 1. Instead, it's $\sigma(w \cdot x + b)$, where σ is called the *sigmoid function*, defined by:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$ (nonlinearity)

- Putting it all together, the output of a sigmoid neuron with inputs *x1,x2,…*, weights *w1,w2,…*, and bias *b* is:

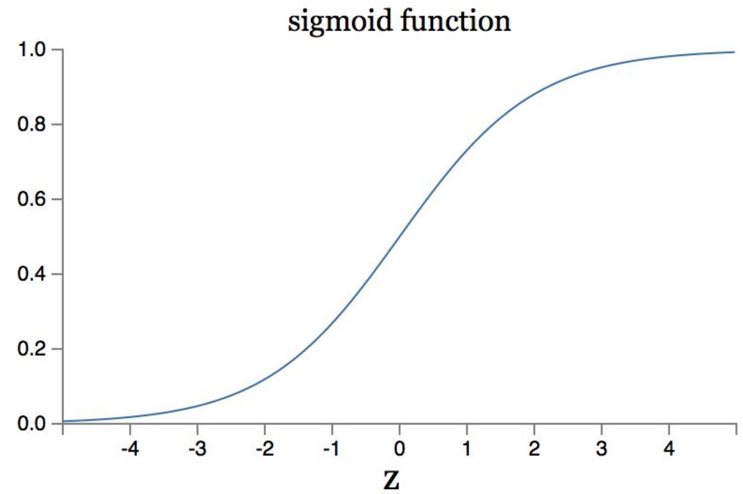
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}$$

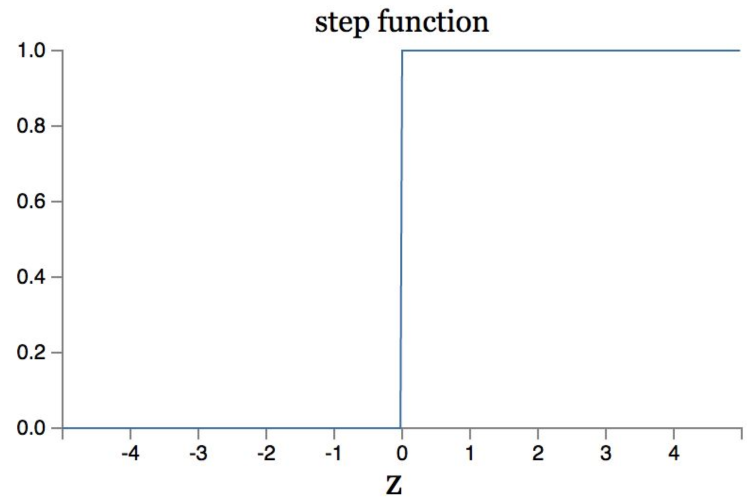
Sigmoid Neurons

- How does σ look?
- This shape is a smoothed out version of a *step function*.
- In fact, If σ had been a step function, then the **sigmoid neuron** would be a **perceptron**!

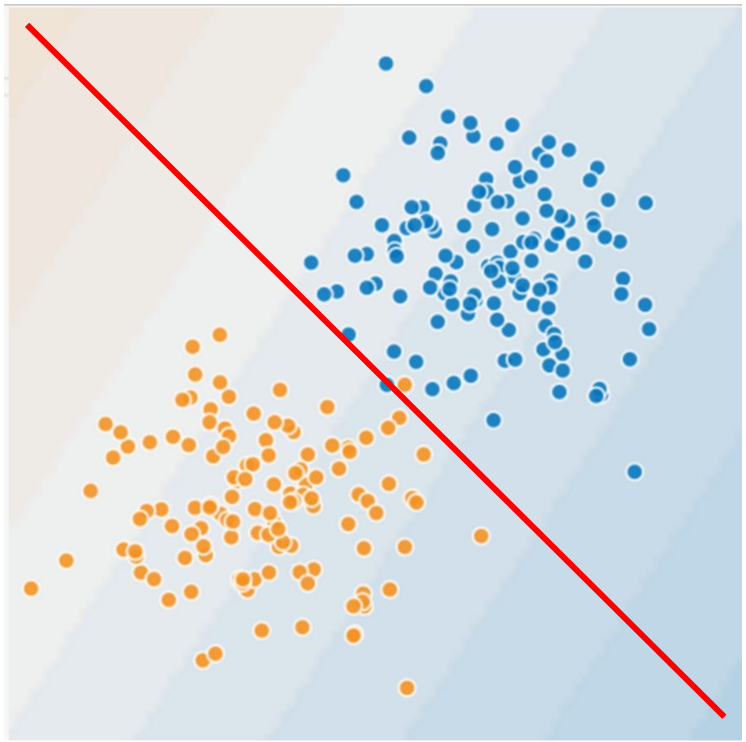
$$output = \sigma(w \cdot x + b)$$

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

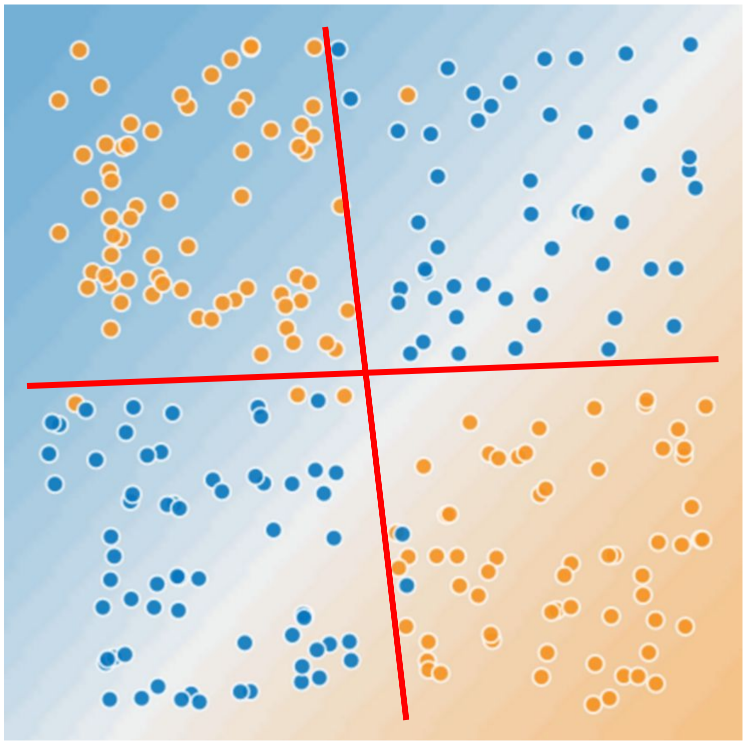




Linear vs Nonlinear Classification



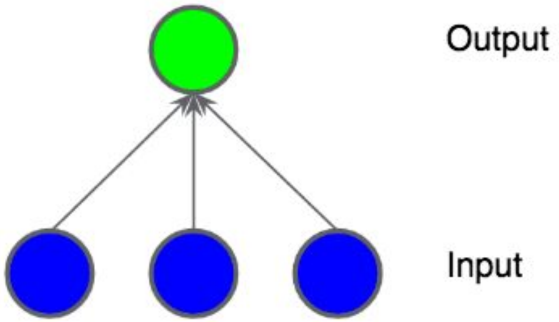
Linear classification problem



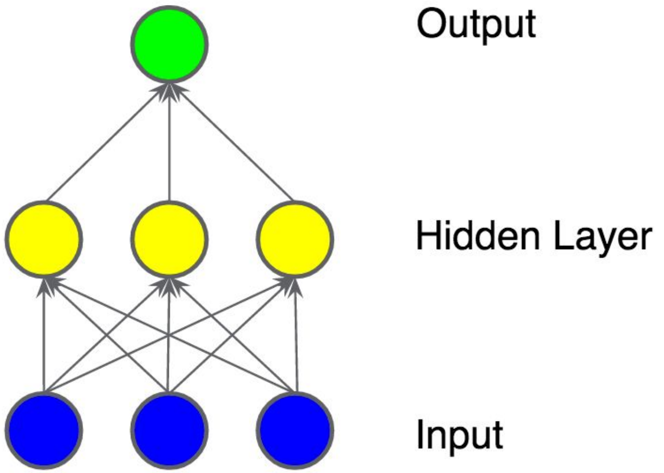
Nonlinear classification problem

Neural Networks Anatomy

- We can represent a linear model as a graph.

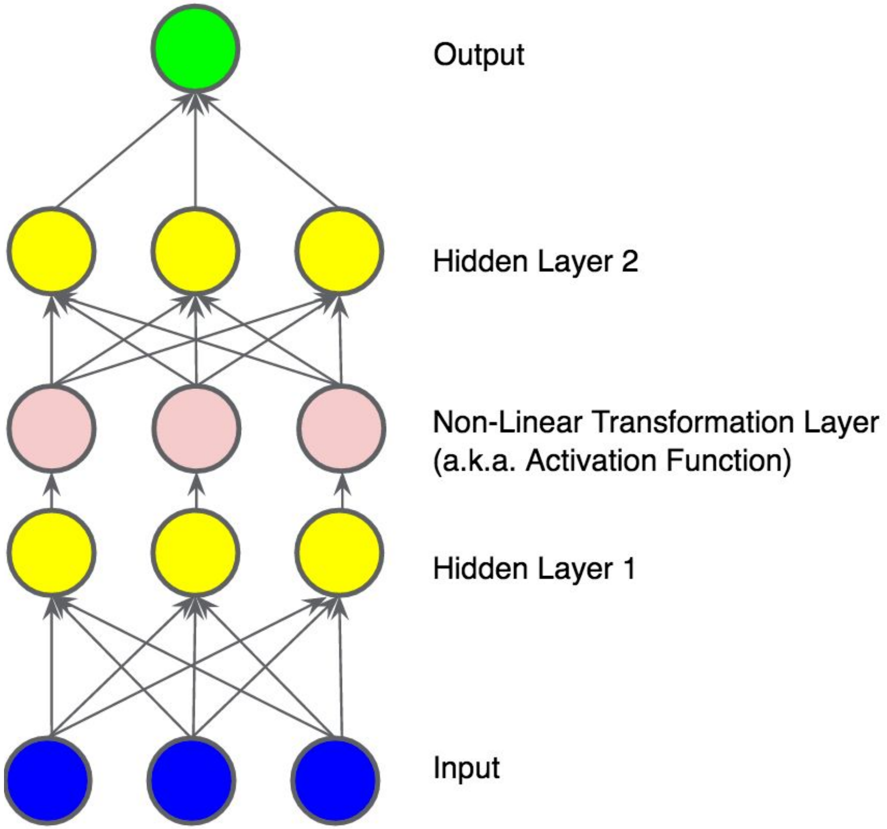


- We can add a "hidden layer" of intermediary values. This is still a linear model.



Neural Networks Anatomy

- To model a nonlinear problem, we can directly introduce a nonlinearity.
- We can pipe each hidden layer node through a nonlinear function.
- The *sigmoid function* is a common activation function.
- Stacking nonlinearities on nonlinearities lets us model very complicated relationships between the inputs and the predicted outputs.



Neural Networks Anatomy

- Summary: a Neural Network is a classifier model consisting of:
    - A set of nodes, analogous to neurons, organized in layers.
    - A set of weights representing the connections between each neural network layer and the layer beneath it.
    - A set of biases, one for each node.
    - An activation function that transforms the output of each node in a layer. Different layers may have different activation functions

Training and Loss

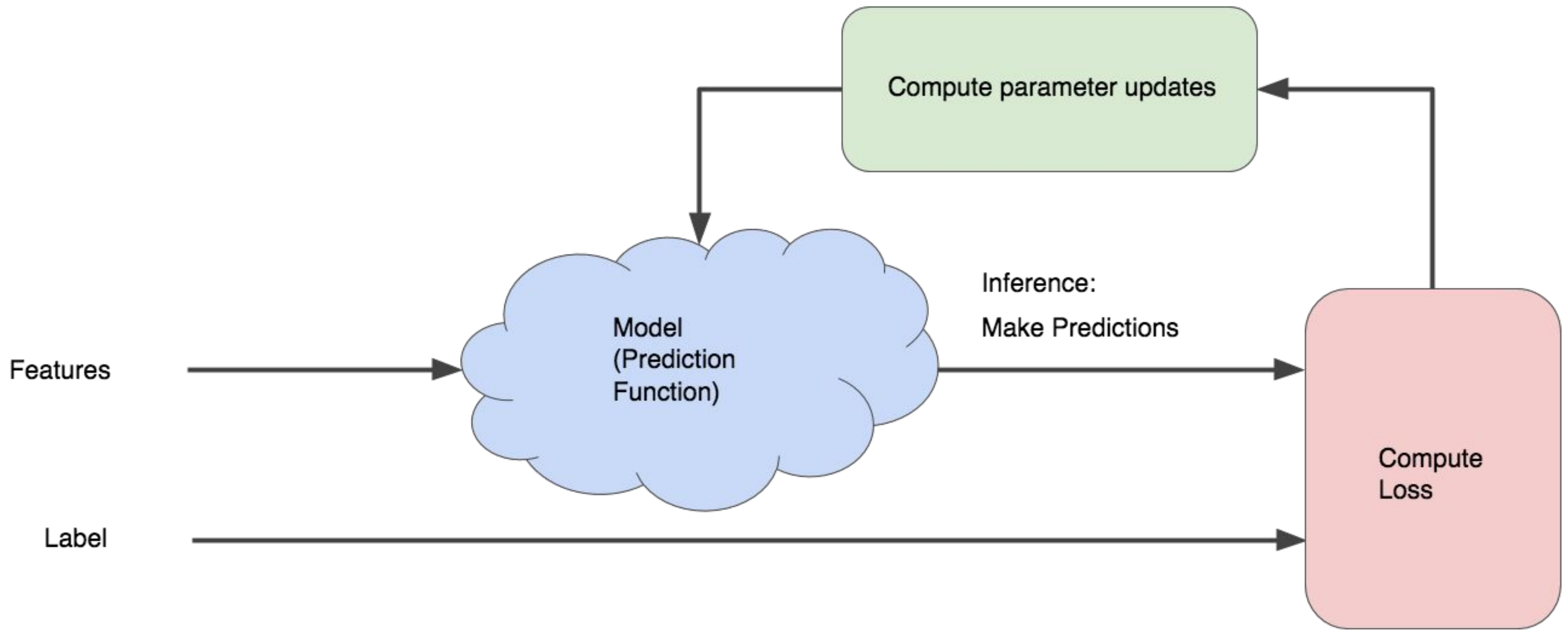
- **Training** a model simply means learning (determining) good values for all the weights and the bias from labeled examples.
- In **supervised learning**, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss.
- **Loss** is the penalty for a bad prediction. It is a number indicating how bad the model's prediction was on a single example (0 for a perfect prediction, greater than 0 otherwise).

$$Loss \equiv \frac{1}{2n} \sum_x \|y(x) - a(x))\|^2$$

- The goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples.

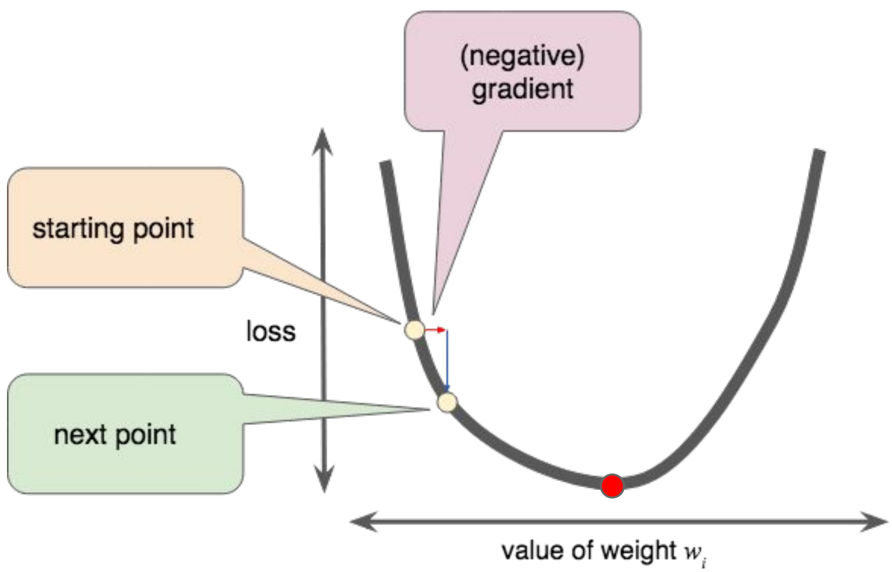
Reducing loss

- Machine learning algorithms use an iterative trial-and-error process to train a model:



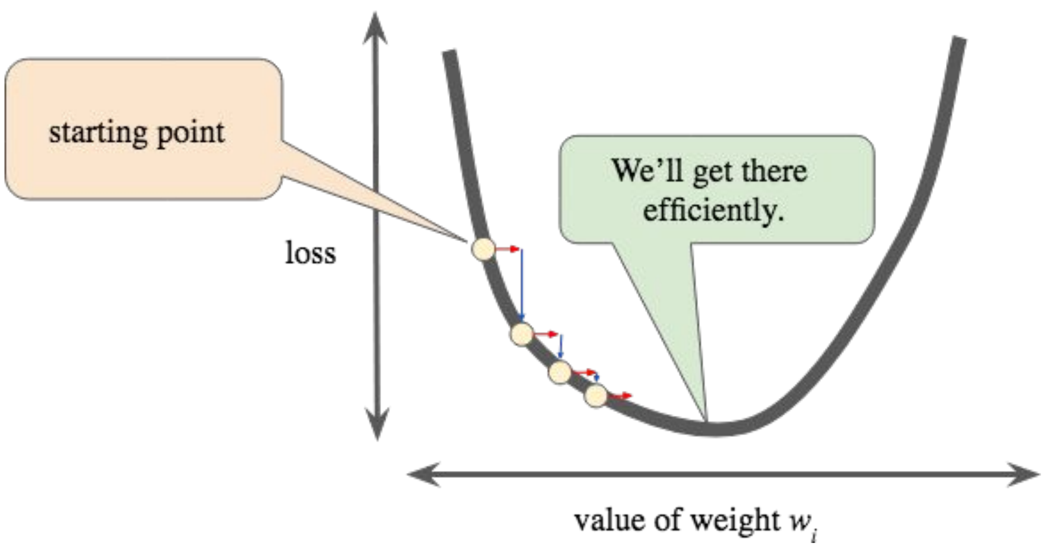
Gradient Descent

- **Gradient descent** is an optimization algorithm used to find a (local) minimum of the loss function, which is parametrized by the weights *w1,w2,…*.
- Algorithm:
  a. Pick a starting point (initialization of weights).
  b. Calculate the gradient (**how?**) of the loss with respect to the weights at the starting point.
  c. Take a step (**how big?**) in the direction of the negative gradient.
  d. Select next point (update weights).
  e. Repeat the process heading towards the minimum of the loss function.



Learning Rate

- Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** to determine the next point.
- Learning rate too small -> learning will take too long.
- Learning rate too large -> might overshoot the minimum.
- **Hyperparameters** are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate.



Stochastic Gradient Descent

- In gradient descent, a **batch** is the total number of examples you use to calculate the gradient in a single iteration.
- So far, we've assumed that the batch has been the entire data set -> a very large batch may cause even a single iteration to take a very long time to compute.
- By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one -> in **mini-batch Stochastic Gradient Descent (mini-batch SGD)**, a batch is typically between 10 and 1000 examples, chosen at random.
- **Stochastic Gradient Descent (SGD)** takes this idea to the extreme -> it uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy.

Backpropagation

- **Backpropagation** is the algorithm that allows us to compute the gradient of the loss function with respect to the learnable parameters (weights and biases) of the network: $\partial Loss_x / \partial w$, $\partial Loss_x / \partial b$

- Algorithm (high level):
  a. **Input**: set the corresponding activation a^0 for the input layer
  b. **Forward pass**: for each layer l = {1, 2, 3, …, L} compute $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$
  c. **Output error**: compute the vector $\delta_j^L = \frac{\partial Loss}{\partial a_j^L} \sigma'(z_j^L)$
  d. **Backpropagate the error**: for each layer l = {1, 2, 3, …, L} compute

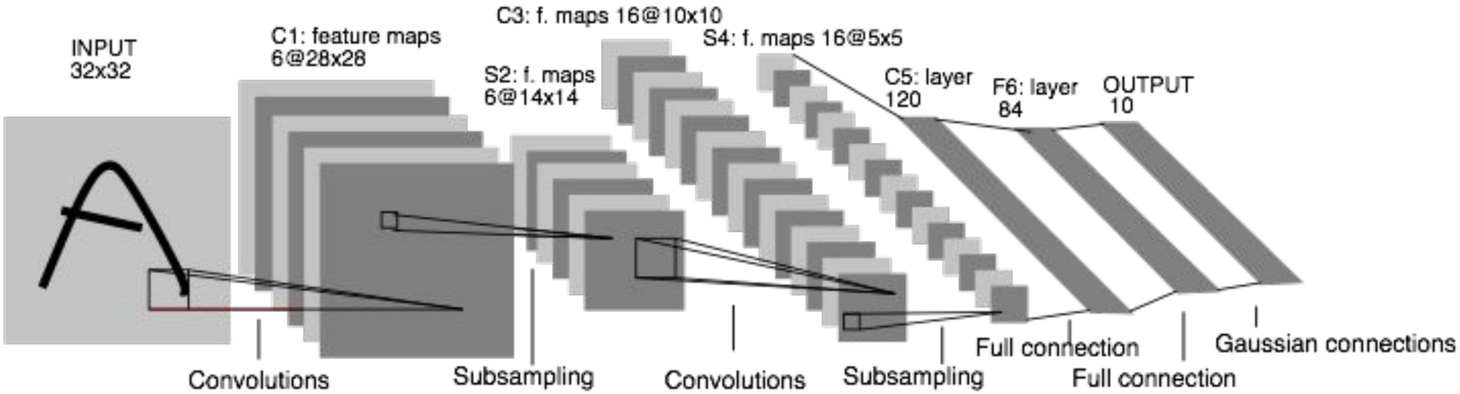
$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

  d. **Output**: the gradient of the loss function is given by

$$\frac{\partial Loss}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad \text{and} \quad \frac{\partial Loss}{\partial b_j^l} = \delta_j^l$$

Convolutional Neural Networks

- **Convolutional Neural Networks** are a type of feedforward networks that make use of the *convolution* operation.
- ConvNets make the explicit assumption that the inputs are images.
  - Instead of dense connections use convolutions (with shared weights).
- Convolutional filters are learnt -> no need for feature engineering.

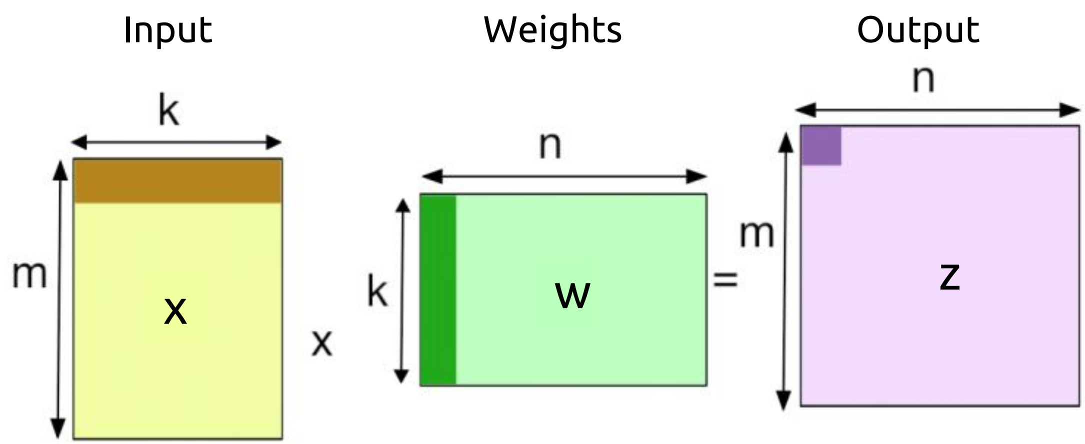


LeNet (1989): a layered model composed of convolution and subsampling operations followed by a classifier for handwritten digits.

Why GEMM is at the heart of Artificial Neural Networks?

- **GEMM** stands for GEneral Matrix to Matrix Multiplication.
- For a fully-connected layer:

$$z = \sum_j w_j x_j$$



- Similar story for convolutional layers (though less obvious).
- GEMM is a highly parallel operation, so it scales naturally to many cores.
    - This kind of workload is well-suited for a **GPU** architecture.

References

- Neural Networks and Deep Learning
- Machine Learning Crash Course
- DIY Deep Learning for Vision: a Hands-On Tutorial with Caffe
- Convolutional Neural Networks (CSE 455)
- Why GEMM is at the heart of deep learning
- Image Classification and Filter Visualization

Thanks!

Questions?

Hands on!

https://goo.gl/Ud2Pws