DataSci 420

lesson 8: neural networks

Seth Mottaghinejad

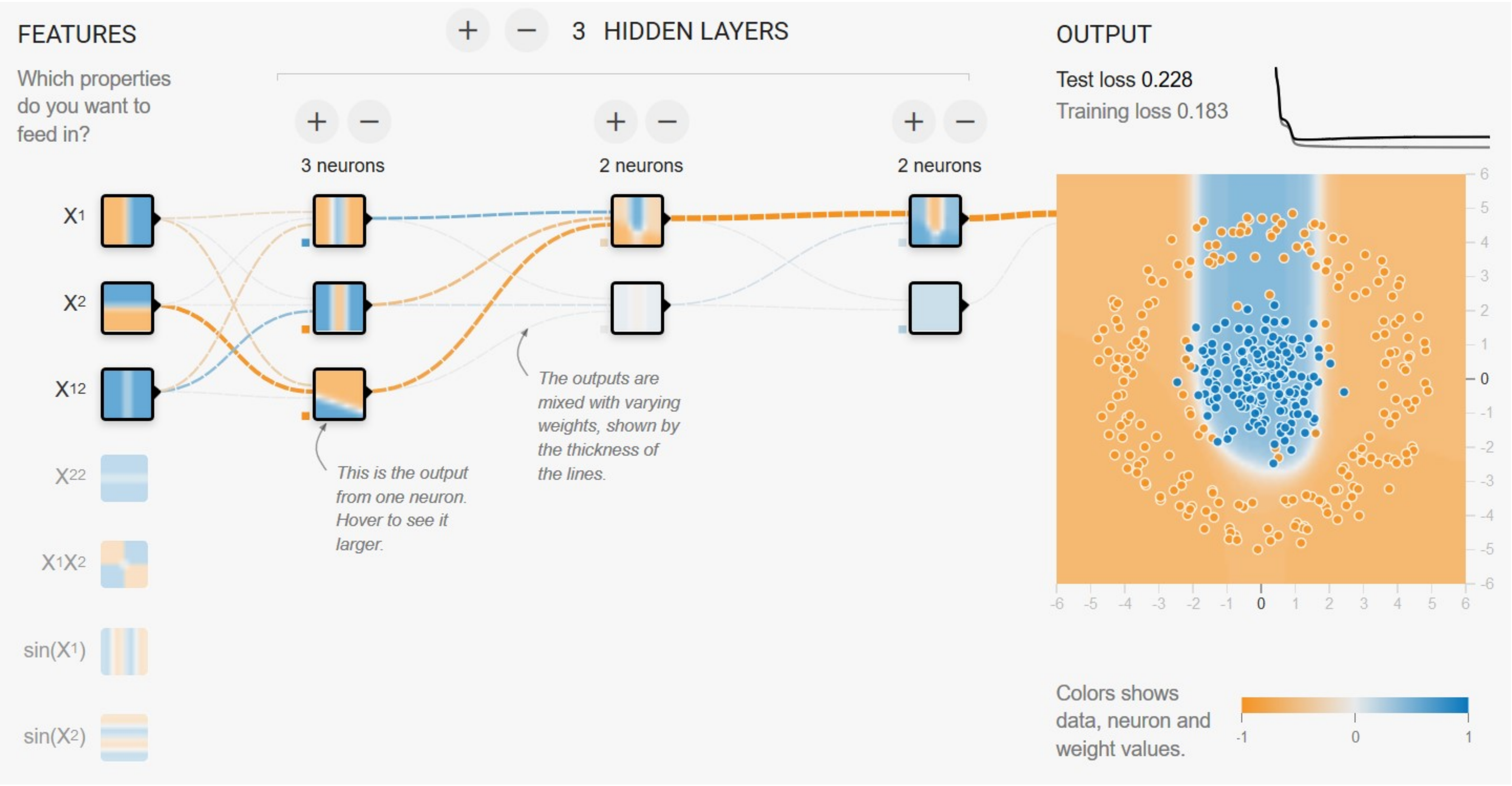
today's agenda

what makes neural networks special

- can **train gradually**: show training progress and stop training when it starts to overfit (otherwise overfitting is almost **inevitable**)
- can **update model parameters** with new data
- they have **feature engineering built in**, allowing for more abstract features in deeper layers
- very **data-hungry** (deep models need a lot of data to not overfit) and **computate-hungry** (**GPU** hardware such as Nvidia, CUDA software layer on top)

math behind neural networks

- **optimization**
  - **batch** gradient descent: precise but too slow
  - **mini-batch** (stochastic) gradient descent SGD: noisy but fast
  - momentum, RMSProp, Adam, etc: help converge faster
- **calculus**
  - **chain rule** and multivariate calculus
- **linear algebra** - basic matrix algebra for how to vectorize computation



source: playground.tensorflow.org

neural network terminology

- input / hidden / output **layers and neurons**
- the **weights** and **biases** are the model **parameters**
- **activation functions** are applied to the **weighted sum** at each layer to get **activations**
- each iteration applies one forward and backward pass using a **mini-batch**, once we exhaust data we have one **epoch**
- at the end of one **forward pass** we compute **loss**
- at the end of one **backward pass** weights and biases are adjusted

activation function

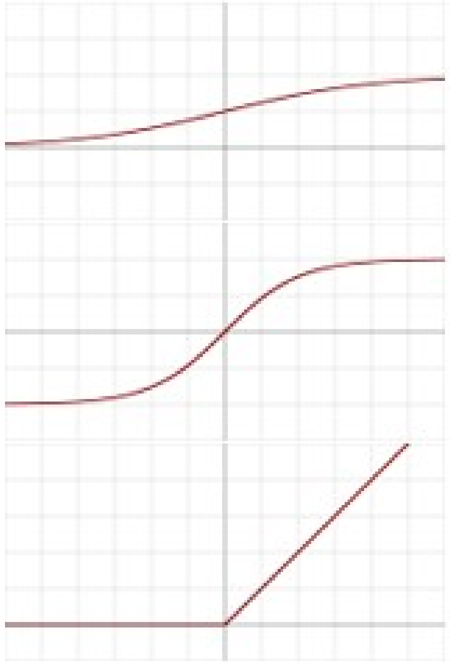
- functions that squash inputs to learn non-linearity
- **sigmoid:** $\sigma(x) \in (0, 1)$
- **tanh:** $\tanh(x) \in (-1, 1)$
- **ReLU:** zeros out negatives
- many other ones

source: www.wikipedia.org

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

$$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$



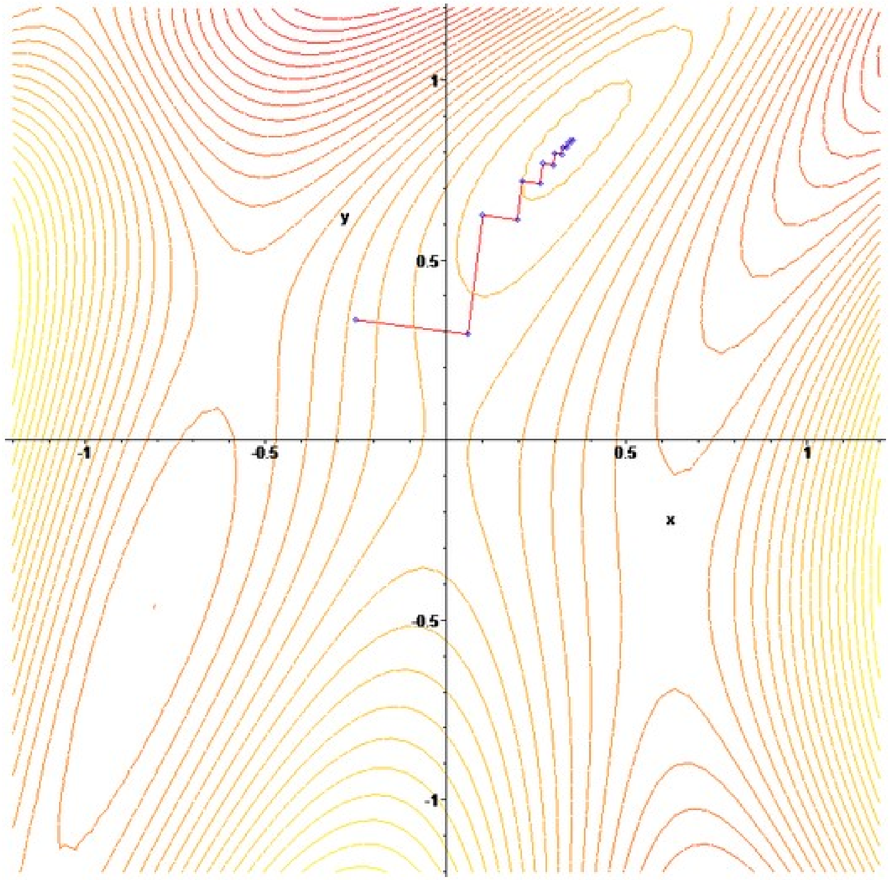
how a neural network trains

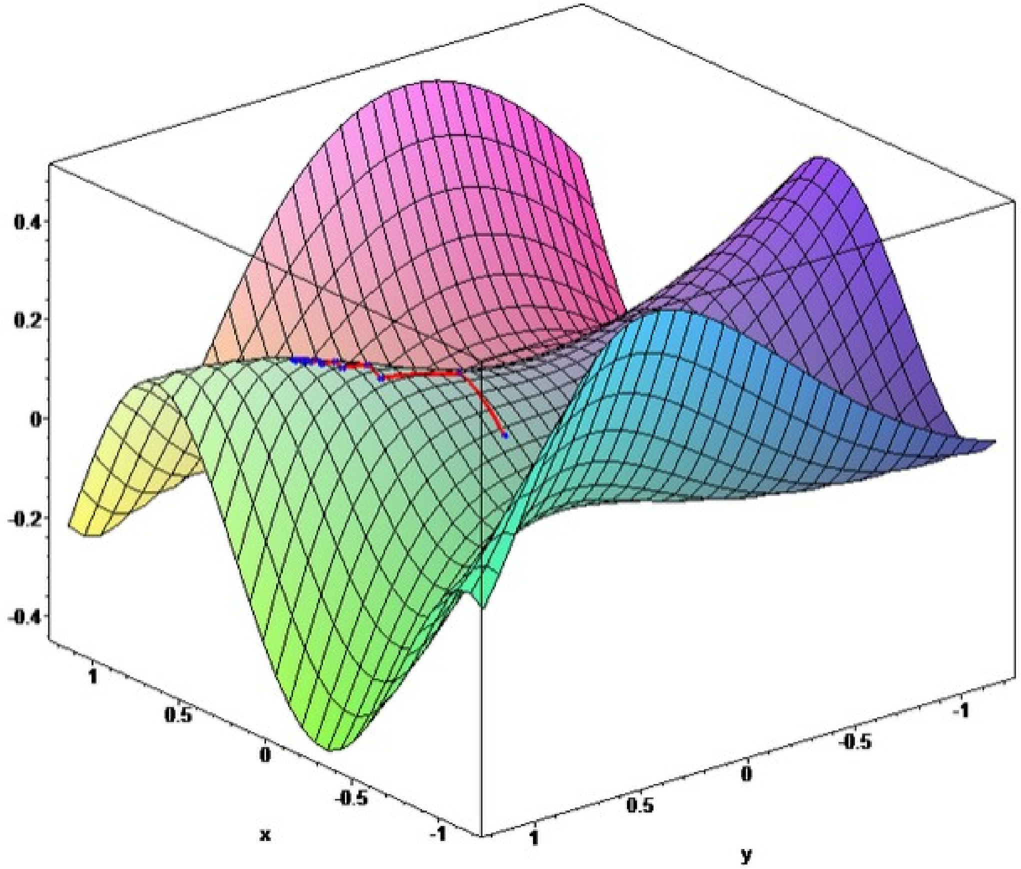
1. **initialize** weights and biases
2. take a **mini-batch** of data
    - make a **forward pass** (get predictions) and evaluate **loss** (error)
    - make a **backward pass** to get gradient of loss w.r.t. weights and biases at each layer
    - **update** weights and biases accordingly
3. repeat step 2 until you converge, meanwhile keep track of performance at every **epoch**

analogy: writing an essay

1. write a essay filled with **random words**
2. for a random **paragraph** of your essay
    - go see your teacher and read it to him to get his **feedback**
    - go back and find how you **incorporate** the feedback
    - **update** paragraph to reflect feedback
3. repeat step 2 until the feedback is just minor details, meanwhile keep track of improvement every time you exhaust all paragraphs (have read whole essay)

break time





source: www.wikipedia.org

backpropagation

- the optimization routine is to minimize the loss function (total error) w.r.t. the weights and biases
  - **gradient descent** will do it, but batch GD is too slow
  - **stochastic gradient descent** using **mini-batches** is much faster, but we still have to do this one parameter at a time
  - **backpropagation** implements SGD but in **one** backward pass
- with **arrays computations**, we can run BP efficiently
- with **tensors and GPUs**, we can speed up even more

prevent over-fitting

- **early-stopping:** if after a certain epoch performance (on validation data) starts to decline then stop training
- **drop-out:** for each iteration, zero out some weights (and biases), do the forward and backward pass, update remaining weights and biases, then repeat
- **regularization:** same as traditional ML
- **more training data:** data augmentation
- **hyper-parameter tuning:** simplify the **architecture** (fewer HLs, fewer units within each HL, etc.)

the end