Introduction to Deep Neural Networks (1)

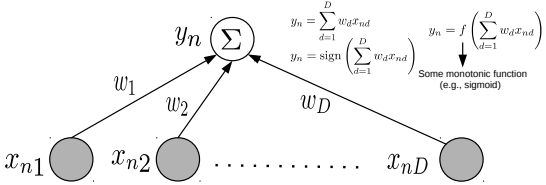
Piyush Rai

Introduction to Machine Learning (CS771A)

October 23, 2018

Linear Models (and their limitations..)

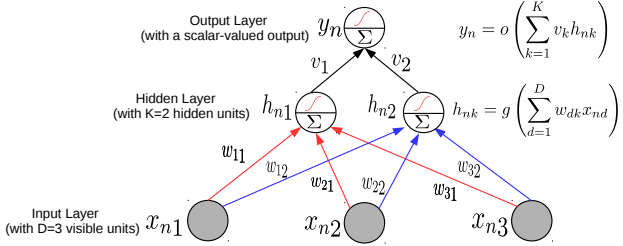
- Linear models: Output produced by taking a linear combination of input features



- Already seen several examples (linear regression, logistic regression, linear SVM, multi-output linear regression, softmax regression, and several others)
- This basic architecture is classically also known as the "Perceptron" (not to be confused with the Perceptron "algorithm", which learns a linear classification model)
- This simple model can't however learn complex functions (e.g., nonlinear decision boundaries)
- We have already seen a way of handling nonlinearities: Kernel Methods (invented in the 90s)
- Something existed in the pre-kernel methods era, too..

Neural Networks a.k.a. Multi-layer Perceptron (MLP)

- Became very popular in early 80s and went to slumber in late 80s (people didn't know how to train them well), but woke up again in the late 2000s (now we do). Rechristened as "Deep Learning"
- An MLP consists of an input layer, an output layer, and one or more hidden layers
- A very simple MLP with input layer with $D = 3$ nodes and single hidden layer with $K = 2$ nodes



- Each node (a.k.a. unit) in the hidden layer computes a nonlinear transform of inputs it receives
- Hidden layer nodes act as "features" in the final layer (a linear model) to produce the output
- The overall effect is a nonlinear mapping from inputs to outputs (justification later)

Illustration: A Neural Network with One Hidden Layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

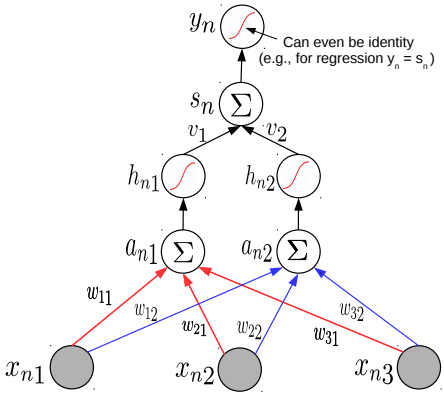
- Nonlinear activation applied on each pre-activation

$$h_{nk} = g(a_{nk})$$

- A linear model applied on the new "features" $\boldsymbol{h}_n$

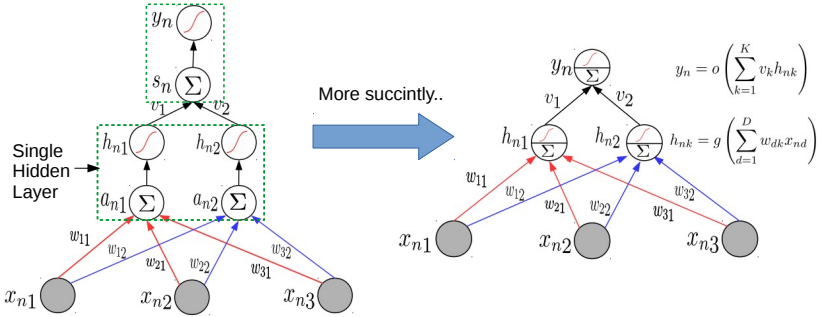
$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

- Finally, the output is produced as $y_n = o(s_n)$
- Unknowns of the model ($\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K$ and $\boldsymbol{v}$) learned by minimizing a loss $\mathcal{L}(\mathbf{W}, \boldsymbol{v}) = \sum_{n=1}^{N} \ell(y_n, o(s_n))$, e.g., squared, logistic, softmax, etc (depending on the output)



Neural Networks: A Basic Pictorial Representation

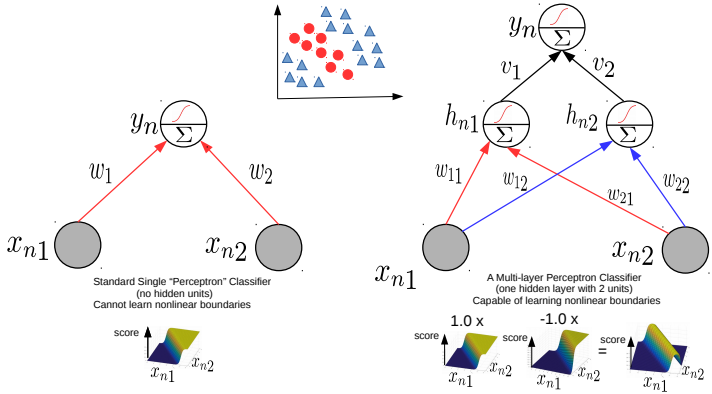
- Note: Hidden layer pre-activations a_{nk} and post-activations h_{nk} will be shown together for brevity



- We will only show h_{nk} to denote the value computed by the k-th hidden unit
- Likewise, for the output layer, we will directly show the final output y_n
- Each node in hidden/output layers computes a linear trans. of its inputs + applies a nonlinearity
- Different layers can use different types of activations (output layer may have none)

MLPs Can Learn Nonlinear Functions: A Justification

- An MLP can be seen as a composition of multiple linear models combined nonlinearly
- Let's look at a simple example of 2-dim inputs that are linearly not separable

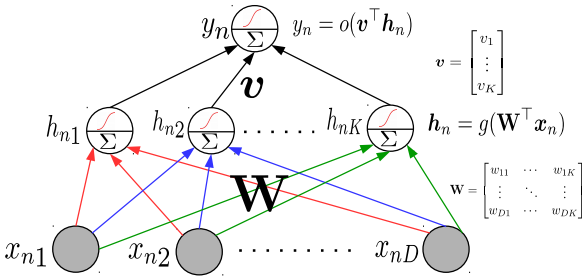


- MLP with a single, sufficiently wide hidden layer can approximate any function (Hornick, 1991)

Examples of some NN/MLP architectures

Neural Networks with One Hidden Layer

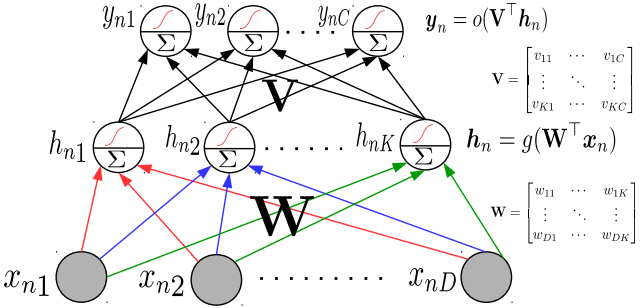
- Aleady saw the special case of a single layer NN ($D = 3$, $K = 2$)
- In general, an NN with D input units and a single hidden layer with K units



- Note: w_{dk} is the weight of edge between input layer node d and hidden layer node k
- $\mathbf{W} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K]$, with $\boldsymbol{w}_k$ being the weights incident on k-th hidden unit
- Each $\boldsymbol{w}_k$ acts as a "feature detector" or "filter" (and there are K such filters in above NN)

Neural Networks with One Hidden Layer and Multiple Outputs

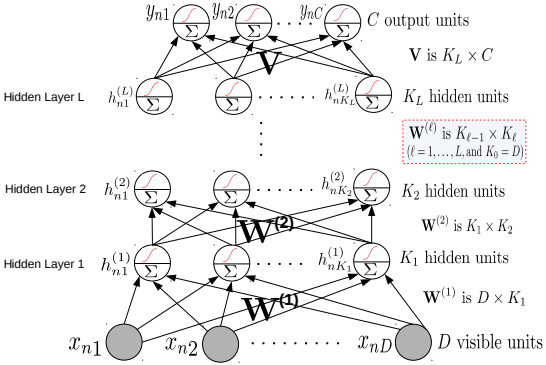
- Very common in multi-class or multi-output/multi-label learning problems
- An NN with D input units, single hidden layers with K units, and multiple outputs



- Similar to multi-output regression or softmax regression on $\boldsymbol{h}_n$ as features

Neural Networks: Multiple Hidden Layers and Multiple Outputs

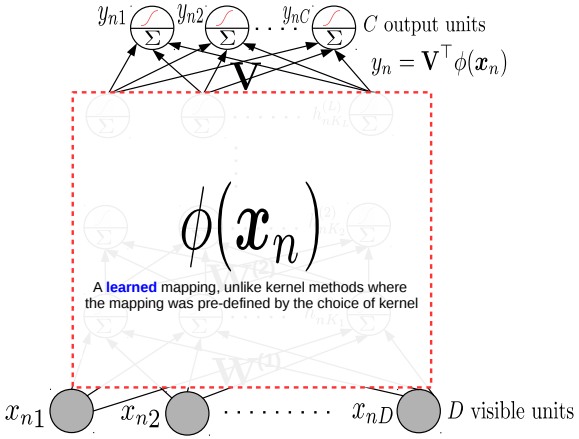
- An NN with D input units, multiple hidden layers, and multiple outputs



- Note: This (and also the previous simpler ones) is called a fully-connected feedforward network
- Fully connected: All pairs of units between adjacent layers are connected to each other
- Feedforward: No backward connections. Also, only nodes in adjacent layers connected

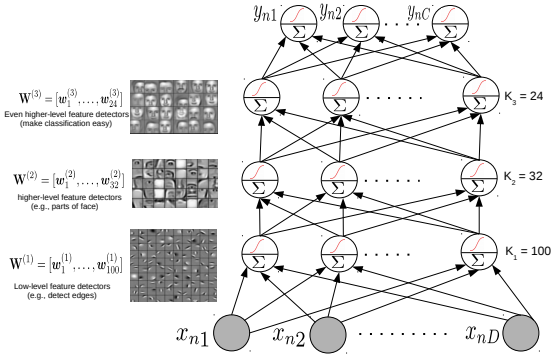
Neural Networks are Feature Learners!

- An NN (single/multiple hidden layers) tries to learn features that can predict the output well



Neural Networks: The Features Learned..

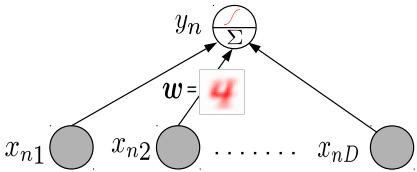
- Deep neural networks are good at detecting features at multiple layers of abstraction
- The connection weights between layers can be thought of as feature detectors or "filters"



- Lowest layer weights detect generic features, higher level weights detect more specific features
- Features learned in one layer are composed of features learned in the layer below

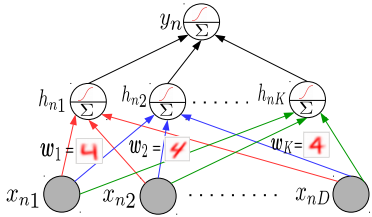
Why Are Neural Network Learned Features Helpful?

- A single layer model will learn an "average" feature detector



- Can't capture subtle variations in the inputs

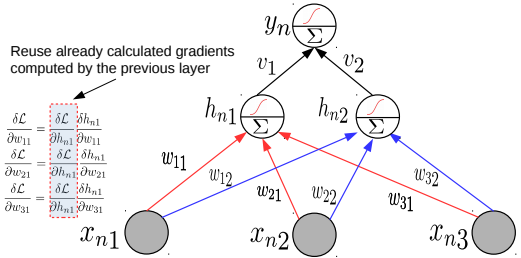
- An MLP can learn multiple feature detectors (even with a single hidden layer)



- Therefore even a single hidden layer helps capture subtle variations in the inputs

Learning Neural Networks via Backpropagation

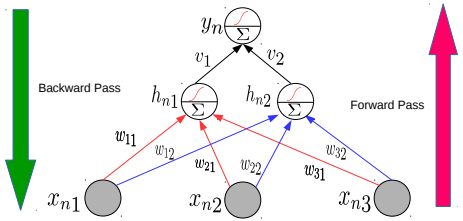
- Backpropagation = Gradient descent using chain rule of derivatives
- (Current) Ideal way for learning deep neural networks
- Chain rule of derivatives: Example, if $y = f_1(x)$ and $x = f_2(z)$ then $\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z}$
- Since neural networks have a "recursive" architecture backprop is especially useful



- Basic idea: Start taking the derivatives from output layer and proceed backwards (hence the name)
- Using backprop in neural nets enables us to reuse previous computations efficiently

Learning Neural Networks via Backpropagation

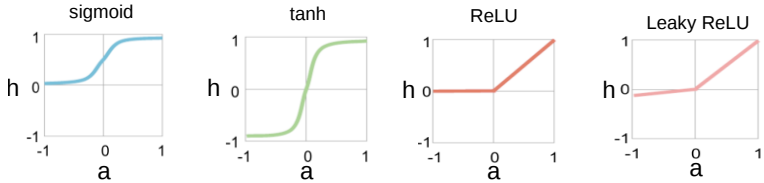
- Backprop iterates between a forward pass and a backward pass



- Forward pass computes the errors $\boldsymbol{e}_n$ using the current parameters
- Backward pass computes the gradients and updates the parameters, starting from the parameters at the top layer and then moving backwards
- Implementing backprop by hand may be very cumbersome for complex, very deep NNs
- Good News: Many software frameworks (e.g., Tensorflow and PyTorch) already implement backprop so you don't need to do it by hand (compute derivatives using chain rule, etc)

Activation Functions

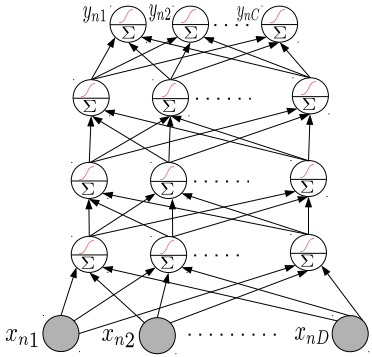
- Some common activation functions



- Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$
- tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$
- ReLU (Rectified Linear Unit): $h = \max(0, a)$
- Leaky ReLU: $h = \max(\beta a, a)$ where β is a small postive number
- Several others, e.g., Softplus $h = \log(1 + \exp(a))$, exponential ReLU, maxout, etc.
- Sigmoid, tanh can have issues during backprop (saturating gradients, non-centered)
- ReLU/leaky ReLU currently one of the most popular (also cheap to compute)

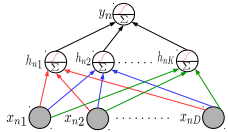
Neural Networks: Some Aspects..

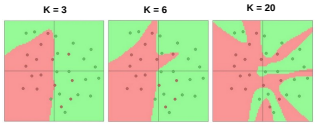
- Much of the magic lies in the hidden layer(s)
- As we've seen, hidden layers learn and detect good features
- However, we need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornick's universal function approximator theorem)?
  - Complex network (several, very wide hidden layers) or simple network (few, moderately wide hidden layers)?
  - Aren't deep neural network prone to overfitting (since they contain a huge number of parameters)?



Representational Power of Hidden Layers

- Consider an NN with a single hidden layer

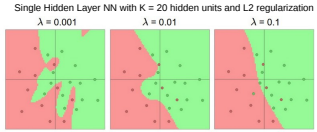




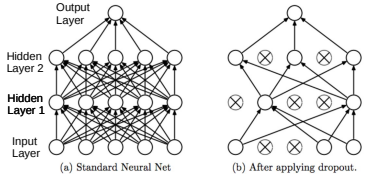
- Recall that each hidden unit "adds" a function to the overall function
- Increasing the number of hidden units will learn more and more complex function
- Very large K seems to overfit. Should we instead prefer small K?
- No! It is better to use large K and regularize well. Here is a reason/justification:
  - Simple NN with small K will have a few local optima, some of which may be bad
  - Complex NN with large K will have many local optimal, all equally good
  - Note: The above interesting behavior of NN has some theoretical justifications (won't discuss here)
- We can also use multiple hidden layers (each sufficiently large) and regularize well

(Preventing) Overfitting in Neural Networks

- Complex single/multiple hidden layer NN can overfit
- Many ways to avoid overfitting, such as
  - Standard regularization on the weights, such as ℓ_2, ℓ_1, etc (ℓ_2 reg. is also called weight decay)

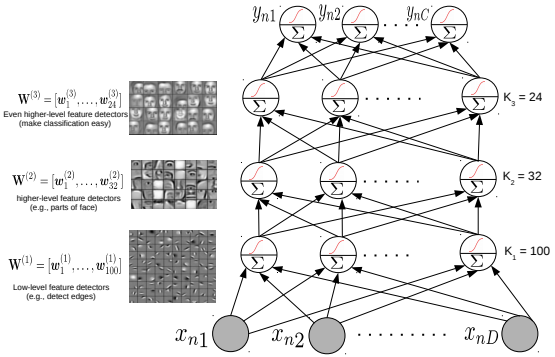


  - Early stopping (traditionally used): Stop when validation error starts increasing
  - Dropout: Randomly remove units (with some probability $p \in (0, 1)$) during training



Wide or Deep?

- While very wide single hidden layer can approx. any function, often we prefer many hidden layers



- Higher layers help learn more directly useful/interpretable features (also useful for compressing data using a small number of features)

Kernel Methods vs Deep Neural Nets

- Recall the prediction rule for a kernel method (e.g., kernel SVM)

$$y = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

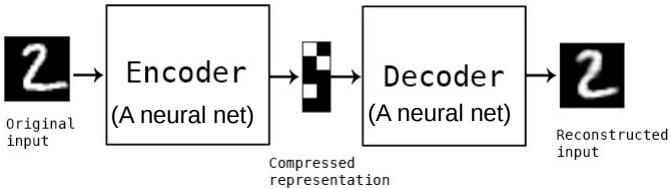
- This is analogous to a single hidden layer NN with fixed/pre-defined hidden nodes $\{k(\boldsymbol{x}_n, \boldsymbol{x})\}_{n=1}^{N}$ and output layer weights $\{\alpha_n\}_{n=1}^{N}$
- The prediction rule for a deep neural network

$$y = \sum_{k=1}^{K} v_k h_k$$

- Here, the h_k's are learned from data (possibly after multiple layers of nonlinear transformations)
- Both kernel methods and deep NNs be seen as using nonlinear basis functions for making predictions. Kernel methods use fixed basis functions (defined by the kernel) whereas NN learns the basis functions adaptively from data

Deep Neural Nets for Unsupervised Learning

- Can use neural nets for dimensionality reduction
- A popular approach is to use autoencoders
- Autoencoder: Compress the input and uncompress to reconstruct
- An encoder (a neural net) does compression and a decoder (a neural net) does decompression



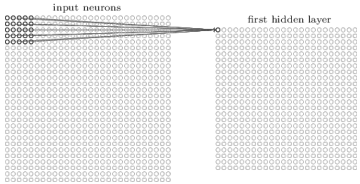
- In an NN based autoencoder, the output layer is the same as the input!

Deep Neural Nets: Some Comments

- Highly effective in learning good feature representations from data in an "end-to-end" manner
- The objective functions of these models are highly non-convex
  - Lots of recent work on non-convex opt., so non-convexity doesn't scare us (that much) anymore
- Training these models is computationally very expensive
  - But GPUs can help to speed up many of the computations
- Training these models can be tricky, especially a proper initialization
  - Several ways to intelligently initialize these models (e.g., unsupervised layer-wise pre-training)
- Deep learning models can also be probabilistic and generative (will look at some of it in next class)

Next Class

- Other types of deep neural networks, e.g.,
  - Convolutional Neural Networks (especially suited for images); not "fully connected"



  - Neural networks for sequence data (e.g., text)
- Some optimization methods especially popular for neural networks
- An overview of other recent advances in deep learning