CSCI567 Machine Learning (Spring 2021)

Sirisha Rambhatla

University of Southern California

Feb 12, 2021

Outline

Outline

Logistics

- Sign-up with your group members for the project!

Outline

1. Logistics
2. Review of last lecture
3. Neural Nets

Linear Discriminant Analysis

The main bottleneck is *not knowing* $\mathcal{P}(X = \boldsymbol{x}|y = c)$

$$\mathcal{P}(y = c|X = \boldsymbol{x}) = \frac{\mathcal{P}(X = \boldsymbol{x}|y = c)\mathcal{P}(y = c)}{\mathcal{P}(X = \boldsymbol{x})}.$$

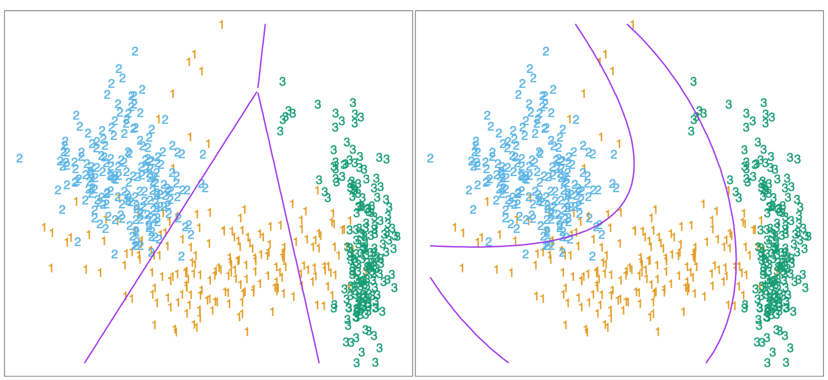
LDA makes two simplifying assumptions:

- Let $\mathcal{P}(X = \boldsymbol{x}|y = c) \sim \mathcal{N}(\boldsymbol{\mu}_c, \Sigma_c)$, and
- Let all class covariances be the same i.e. $\Sigma_c = \Sigma$ for all $c \in [C]$

If so, the decision boundary (for binary classification) is given by

$$\mathcal{P}(y = 0|X = \boldsymbol{x}) = \mathcal{P}(y = 1|X = \boldsymbol{x})$$
$$\mathcal{P}(X = \boldsymbol{x}|y = 0)\mathcal{P}(y = 0) = \mathcal{P}(X = \boldsymbol{x}|y = 1)\mathcal{P}(y = 1)$$

What do the decision boundaries look like?

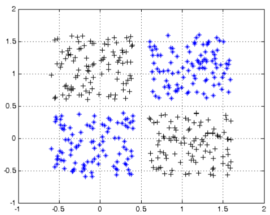


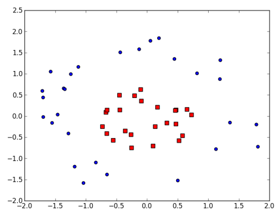
The decision boundaries are a quadratic when Σ's are not the same, this is known as *Quadratic Discriminant Analysis*!

Outline

1. Logistics
2. Review of last lecture
3. Neural Nets
   - Definition
   - Backpropagation
   - Preventing overfitting

Linear models are not always adequate





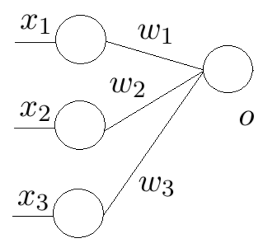
We can use a nonlinear mapping as discussed:

$$\boldsymbol{\phi}(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^{\mathsf{D}} \rightarrow \boldsymbol{z} \in \mathbb{R}^{\mathsf{M}}$$

But what kind of nonlinear mapping ϕ should be used? Can we actually learn this nonlinear mapping?

THE most popular nonlinear models nowadays: **neural nets**

Linear model as a one-layer neural net



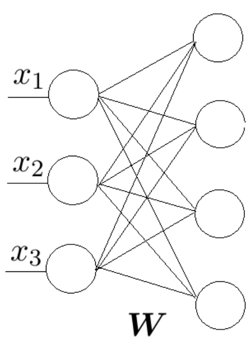
$o = h(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$

$h(a) = a$ for linear model

To create non-linearity, can use

- Rectified Linear Unit (**ReLU**): $h(a) = \max\{0, a\}$
- sigmoid function: $h(a) = \frac{1}{1+e^{-a}}$
- TanH: $h(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- many more

More output nodes

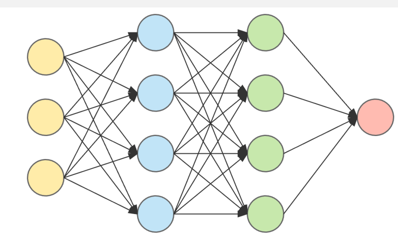


$\boldsymbol{o} = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$

$\boldsymbol{W} \in \mathbb{R}^{4\times3}$, $\boldsymbol{h} : \mathbb{R}^4 \rightarrow \mathbb{R}^4$ so $\boldsymbol{h}(\boldsymbol{a}) = (h_1(a_1), h_2(a_2), h_3(a_3), h_4(a_4))$

Can think of this as a nonlinear basis: $\boldsymbol{\Phi}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{W}\boldsymbol{x})$

More layers



Becomes a network:

- each node is called a **neuron**
- $\boldsymbol{h}$ is called the **activation function**
  - can use $h(a) = 1$ for one neuron in each layer to *incorporate bias term*
  - output neuron can use $h(a) = a$
- #layers refers to #hidden_layers (plus 1 or 2 for input/output layers)
- **deep** neural nets can have many layers and *millions* of parameters
- this is a **feedforward, fully connected** neural net, there are many variants

How powerful are neural nets?

Universal approximation theorem (Cybenko, 89; Hornik, 91):

A feedforward neural net with a single hidden layer can approximate any continuous functions.

It might need a huge number of neurons though, and *depth helps!*

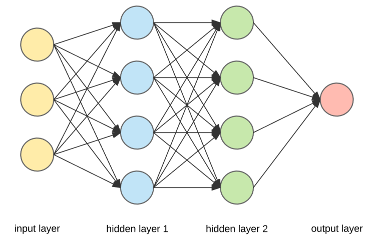
Designing network architecture is important and very complicated

- for feedforward network, need to decide number of hidden layers, number of neurons at each layer, activation functions, etc.

Math formulation

An L-layer neural net can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_{\mathsf{L}}\left(\boldsymbol{W}_L \boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}_{L-1} \cdots \boldsymbol{h}_1\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right)$$



To ease notation, for a given input $\boldsymbol{x}$, define recursively

$$\boldsymbol{o}_0 = \boldsymbol{x}, \qquad \boldsymbol{a}_\ell = \boldsymbol{W}_\ell \boldsymbol{o}_{\ell-1}, \qquad \boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell) \qquad\qquad (\ell = 1, \ldots, \mathsf{L})$$

where

- $\boldsymbol{W}_\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights between layer $\ell - 1$ and ℓ
- $\mathsf{D}_0 = \mathsf{D}, \mathsf{D}_1, \ldots, \mathsf{D}_{\mathsf{L}}$ are numbers of neurons at each layer
- $\boldsymbol{a}_\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}_\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output to layer ℓ
- $\boldsymbol{h}_\ell : \mathbb{R}^{\mathsf{D}_\ell} \to \mathbb{R}^{\mathsf{D}_\ell}$ is activation functions at layer ℓ

Learning the model

No matter how complicated the model is, our goal is the same: minimize

$$\mathcal{E}(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \frac{1}{N} \sum_{n=1}^{\mathsf{N}} \mathcal{E}_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}})$$

where

$$\mathcal{E}_n(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}) = \begin{cases} \|\boldsymbol{f}(\boldsymbol{x}_n) - \boldsymbol{y}_n\|_2^2 & \text{for regression} \\ \ln\left(1 + \sum_{k \neq y_n} e^{f(\boldsymbol{x}_n)_k - f(\boldsymbol{x}_n)_{y_n}}\right) & \text{for classification} \end{cases}$$

How to optimize such a complicated function?

Same thing: apply **SGD**! even if the model is *nonconvex*.

What is the gradient of this complicated function?

Chain rule is the only secret:

- for a composite function $f(g(w))$

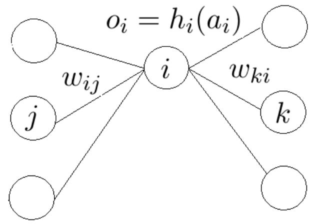
$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial w}$$

- for a composite function $f(g_1(w), \ldots, g_d(w))$

$$\frac{\partial f}{\partial w} = \sum_{i=1}^{d} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial w}$$

  the simplest example $f(g_1(w), g_2(w)) = g_1(w) g_2(w)$

Computing the derivative



Drop the subscript ℓ for layer for simplicity.

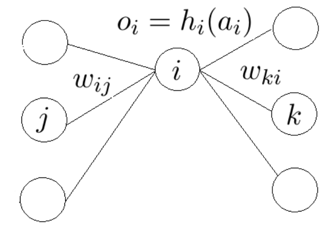
Find the **derivative of $\mathcal{E}_n$ w.r.t. to w_{ij}**

$$\frac{\partial \mathcal{E}_n}{\partial w_{ij}} = \frac{\partial \mathcal{E}_n}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial \mathcal{E}_n}{\partial a_i}\frac{\partial (w_{ij}o_j)}{\partial w_{ij}} = \frac{\partial \mathcal{E}_n}{\partial a_i} o_j$$

$$\frac{\partial \mathcal{E}_n}{\partial a_i} = \frac{\partial \mathcal{E}_n}{\partial o_i}\frac{\partial o_i}{\partial a_i} = \left(\sum_k \frac{\partial \mathcal{E}_n}{\partial a_k}\frac{\partial a_k}{\partial o_i}\right) h_i'(a_i) = \left(\sum_k \frac{\partial \mathcal{E}_n}{\partial a_k} w_{ki}\right) h_i'(a_i)$$

Computing the derivative

Adding the subscript for layer:



$$\frac{\partial \mathcal{E}_n}{\partial w_{\ell,ij}} = \frac{\partial \mathcal{E}_n}{\partial a_{\ell,i}} o_{\ell-1,j}$$

$$\frac{\partial \mathcal{E}_n}{\partial a_{\ell,i}} = \left(\sum_k \frac{\partial \mathcal{E}_n}{\partial a_{\ell+1,k}} w_{\ell+1,ki}\right) h_{\ell,i}'(a_{\ell,i})$$

For the last layer, for square loss

$$\frac{\partial \mathcal{E}_n}{\partial a_{\mathsf{L},i}} = \frac{\partial (h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i})^2}{\partial a_{\mathsf{L},i}} = 2(h_{\mathsf{L},i}(a_{\mathsf{L},i}) - y_{n,i})h_{\mathsf{L},i}'(a_{\mathsf{L},i})$$

Computing the derivative

Using **matrix notation** greatly simplifies presentation and implementation:

$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}_\ell} = \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}_\ell}\boldsymbol{o}_{\ell-1}^{\mathrm{T}}$$

$$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}_\ell} = \begin{cases} \left(\boldsymbol{W}_{\ell+1}^{\mathrm{T}} \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}_{\ell+1}}\right) \circ \boldsymbol{h}_\ell'(\boldsymbol{a}_\ell) & \text{if } \ell < \mathsf{L} \\ 2(\boldsymbol{h}_{\mathsf{L}}(\boldsymbol{a}_{\mathsf{L}}) - \boldsymbol{y}_n) \circ \boldsymbol{h}_{\mathsf{L}}'(\boldsymbol{a}_{\mathsf{L}}) & \text{else} \end{cases}$$

where $\boldsymbol{v}_1 \circ \boldsymbol{v}_2 = (v_{11}v_{21}, \cdots, v_{1\mathsf{D}}v_{2\mathsf{D}})$ is the element-wise product (a.k.a. Hadamard product).

Verify yourself!

Putting everything into SGD

The **backpropagation** algorithm (**Backprop**)

Initialize $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{\mathsf{L}}$. Repeat:

1. randomly pick one data point $n \in [\mathsf{N}]$
2. **forward propagation**: for each layer $\ell = 1, \ldots, \mathsf{L}$
   - compute $\boldsymbol{a}_\ell = \boldsymbol{W}_\ell \boldsymbol{o}_{\ell-1}$ and $\boldsymbol{o}_\ell = \boldsymbol{h}_\ell(\boldsymbol{a}_\ell)$ $\quad(\boldsymbol{o}_0 = \boldsymbol{x}_n)$
3. **backward propagation**: for each $\ell = L, \ldots, 1$
   - compute
   $$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}_\ell} = \begin{cases} \left(\boldsymbol{W}_{\ell+1}^{\mathrm{T}} \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}_{\ell+1}}\right) \circ \boldsymbol{h}_\ell'(\boldsymbol{a}_\ell) & \text{if } \ell < \mathsf{L} \\ 2(\boldsymbol{h}_{\mathsf{L}}(\boldsymbol{a}_{\mathsf{L}}) - \boldsymbol{y}_n) \circ \boldsymbol{h}_{\mathsf{L}}'(\boldsymbol{a}_{\mathsf{L}}) & \text{else} \end{cases}$$
   - update weights
   $$\boldsymbol{W}_\ell \leftarrow \boldsymbol{W}_\ell - \eta \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}_\ell} = \boldsymbol{W}_\ell - \eta \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}_\ell}\boldsymbol{o}_{\ell-1}^{\mathrm{T}}$$

Think about how to do the last two steps properly!

More tricks to optimize neural nets

Many variants based on backprop

- SGD with **minibatch**: randomly sample a batch of examples to form a stochastic gradient
- SGD with **momentum**
- $\cdots$

SGD with momentum

Initialize $\boldsymbol{w}_0$ and **velocity** $\boldsymbol{v} = \boldsymbol{0}$

For $t = 1, 2, \ldots$

- form a stochastic gradient $\boldsymbol{g}_t$
- update velocity $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \eta\boldsymbol{g}_t$ for some discount factor $\alpha \in (0, 1)$
- update weight $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} + \boldsymbol{v}$

Updates for first few rounds:

- $\boldsymbol{w}_1 = \boldsymbol{w}_0 - \eta\boldsymbol{g}_1$
- $\boldsymbol{w}_2 = \boldsymbol{w}_1 - \alpha\eta\boldsymbol{g}_1 - \eta\boldsymbol{g}_2$
- $\boldsymbol{w}_3 = \boldsymbol{w}_2 - \alpha^2\eta\boldsymbol{g}_1 - \alpha\eta\boldsymbol{g}_2 - \eta\boldsymbol{g}_3$
- $\cdots$

Overfitting

Overfitting is very likely since the models are too powerful.

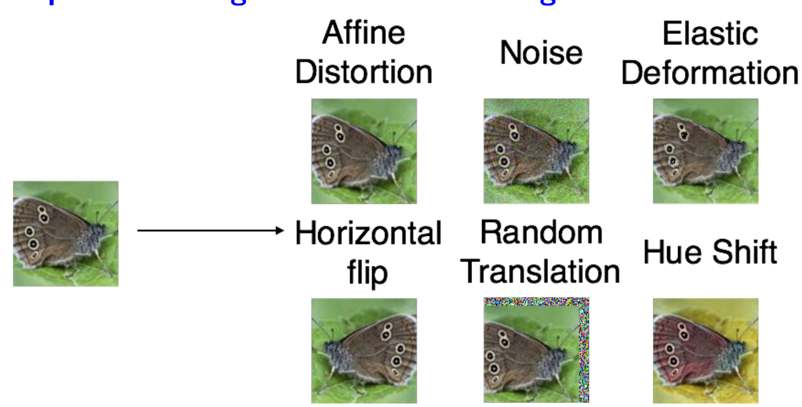
Methods to overcome overfitting:

- data augmentation
- regularization
- dropout
- early stopping
- $\cdots$

Data augmentation

Data: the more the better. How do we get more data?

Exploit prior knowledge to add more training data



Regularization

L2 regularization: minimize

$$\mathcal{E}'(\boldsymbol{W}_1, \dots, \boldsymbol{W}_\mathsf{L}) = \mathcal{E}(\boldsymbol{W}_1, \dots, \boldsymbol{W}_\mathsf{L}) + \lambda \sum_{\ell=1}^{\mathsf{L}} \|\boldsymbol{W}_\ell\|_2^2$$

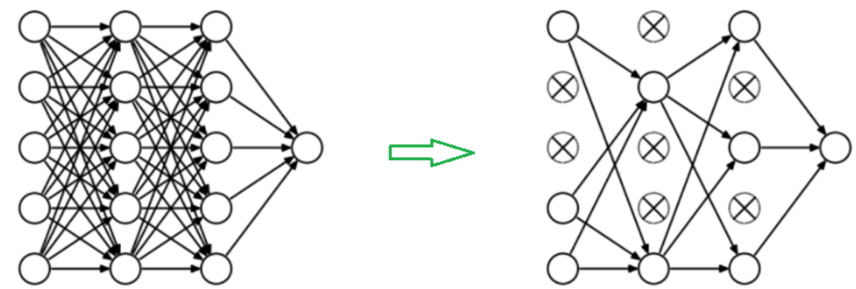
Simple change to the gradient:

$$\frac{\partial \mathcal{E}'}{\partial w_{ij}} = \frac{\partial \mathcal{E}}{\partial w_{ij}} + 2\lambda w_{ij}$$

Introduce *weight decaying effect*

Dropout

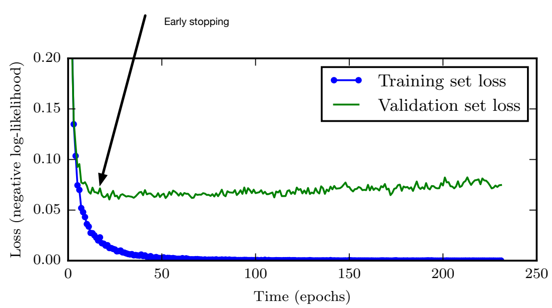
Randomly delete neurons during training



Very effective, makes training faster as well

Early stopping

Stop training when the performance on validation set stops improving



Conclusions for neural nets

Deep neural networks

- are hugely popular, achieving *best performance* on many problems
- do need *a lot of data* to work well
- take *a lot of time* to train (need GPUs for massive parallel computing)
- take some work to select architecture and hyperparameters
- are still not well understood in theory