ENGR-E 533

Deep Learning Systems

Module 01

Shallow Neural Network Basics

Minje Kim

Department of Intelligent Systems Engineering

Email: minje@indiana.edu

Website: http://minjekim.com

Research Group: http://saige.sice.indiana.edu

Meeting Request: http://doodle.com/minje

INDIANA UNIVERSITY

SCHOOL OF INFORMATICS, COMPUTING, AND ENGINEERING

The Last Layer

- Function approximation

- Don't worry, we're going to cover the first layer, too
  - And the layers in the middle
- But, why the last layer first?
  - Everything we do with neural networks is about approximating a function

The output of the function $\in \mathbb{R}^{K\times 1}$ $\boldsymbol{y} = \mathcal{F}(\boldsymbol{x})$ Your observed data sample $\in \mathbb{R}^{D\times 1}$

The mapping function you want to know (but you can never know its exact form)

The predicted output $\hat{\boldsymbol{y}} = \mathcal{G}(\boldsymbol{x}\ ; \mathbb{W}\ )$ Parameters

The estimate of the mapping function

Error function of your choice

- The objective function? $\arg\min_{\mathcal{G}} \mathcal{D}(\boldsymbol{y}||\hat{\boldsymbol{y}})$
  - What's wrong with it?
- It's easier to work with a parametric function $\arg\min_{\mathbb{W}} \mathcal{D}\big(\boldsymbol{y}||\mathcal{G}(\boldsymbol{x};\mathbb{W})\big)$
- The actual optimization is done to reduce the sum of all error network $\arg\min_{\mathbb{W}} \sum_t \mathcal{D}\big(\boldsymbol{Y}_{:,t})||\mathcal{G}(\boldsymbol{X}_{:,t};\mathbb{W})\big)$

The Last Layer

- Function approximation

- Today we focus on the function approximation
  - Forget about neural networks just for now
- What I want to show you is
  - We can solve this potentially nonlinear function approximation problem linearly
    - In a certain condition
    - We'll transform the raw input data into feature space
- What kind of functions?
  - Basically any kind of functions
    - Scalar-to-scalar mapping
    - Vector-to-scalar mapping
      - Vector-to-[bounded scalar] mapping
    - Vector-to-vector mapping
      - Vector-to-[bounded vector] mapping
- Hold on.. what about differentiation, optimization, backpropagation, etc?
  - Forget about them
  - For now I want to give you an intuition that things are about template matching

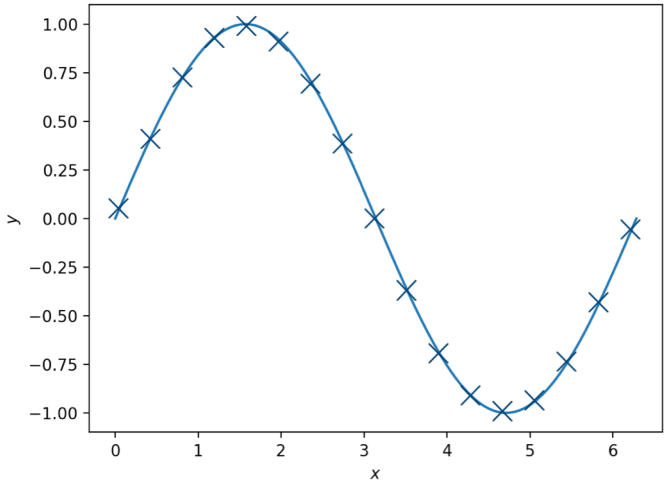
The Last Layer

- Case study #1: scalar-to-scalar mapping, a linear approximation

- Suppose the mapping function between scalar variables x and y is defined by

$$y = \mathcal{F}(x) = \sin(x)$$

  - And we don't know it



- What happens if we try to approximate it linearly?

$$\hat{y}_t = \mathcal{G}(x_t; \mathbb{W}) = a_1 x_t + a_0$$

$$\mathbb{W} = \boldsymbol{a} = [a_1, a_0]^\top$$

$$\hat{y}_t = [a_1, a_0][x_t, 1]^\top$$

- This relationship should hold for all pairs of input and output samples

$$[\hat{y}_0 \ \hat{y}_1 \ \hat{y}_2 \ \cdots \ \hat{y_{T-1}}] = [a_1 \ a_0] \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{T-1} \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix}$$

$$\Leftrightarrow \hat{\boldsymbol{y}}^\top = \boldsymbol{a}^\top \boldsymbol{X}$$

The Last Layer

- Case study #1: scalar-to-scalar mapping, a linear approximation

$$\hat{\boldsymbol{y}}^\top = \boldsymbol{a}^\top \boldsymbol{X}$$

- What are we going to do with this?

$$\arg\min_{\mathbb{W}=\boldsymbol{a}} \sum_t \mathcal{D}(\hat{y}_t||y_t) = \arg\min_{\boldsymbol{a}} \sum_t (\hat{y}_t - y_t)^2 = (\boldsymbol{y}-\hat{\boldsymbol{y}})^\top(\boldsymbol{y}-\hat{\boldsymbol{y}})$$

$$= \arg\min_{\boldsymbol{a}} (\boldsymbol{y}^\top - \boldsymbol{a}^\top\boldsymbol{X})(\boldsymbol{y}^\top - \boldsymbol{a}^\top\boldsymbol{X})^\top = \arg\min_{\boldsymbol{a}} (\boldsymbol{y}^\top - \boldsymbol{a}^\top\boldsymbol{X})(\boldsymbol{y} - \boldsymbol{X}^\top\boldsymbol{a})$$

$$= \arg\min_{\boldsymbol{a}} (\boldsymbol{y}^\top\boldsymbol{y} + \boldsymbol{a}^\top\boldsymbol{X}\boldsymbol{X}^\top\boldsymbol{a} - 2\boldsymbol{y}^\top\boldsymbol{X}^\top\boldsymbol{a})$$

- Then what?

$$\frac{\partial \boldsymbol{y}^\top\boldsymbol{y} + \boldsymbol{a}^\top\boldsymbol{X}\boldsymbol{X}^\top\boldsymbol{a} - 2\boldsymbol{y}^\top\boldsymbol{X}^\top\boldsymbol{a}}{\partial \boldsymbol{a}} = 2\boldsymbol{X}\boldsymbol{X}^\top\boldsymbol{a} - 2\boldsymbol{X}\boldsymbol{y}^\top = 0$$

$$\boldsymbol{a} = (\boldsymbol{X}\boldsymbol{X}^\top)^{-1}\boldsymbol{X}\boldsymbol{y}^\top$$

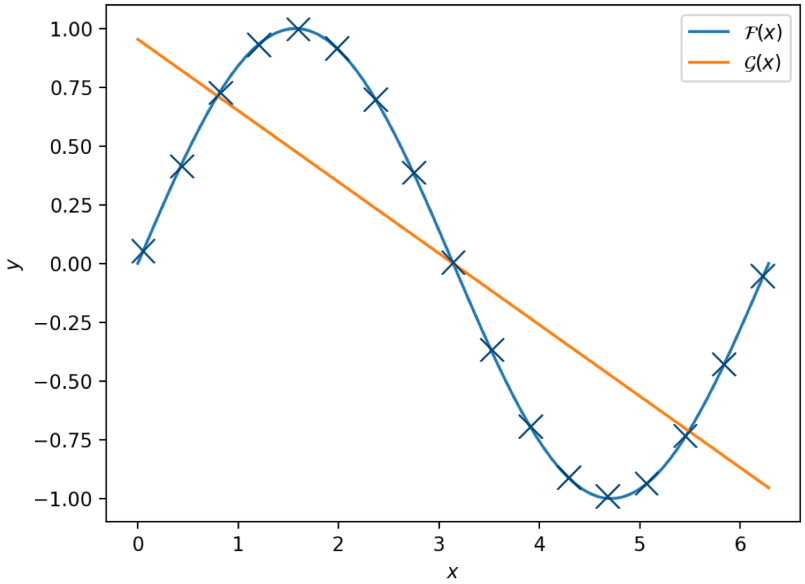
  - How can I be sure that this is the solution?
    - The objective function is quadratic

- We've got a closed form solution for this linear model
  - We don't need an iterative optimization algorithm

The Last Layer

- Case study #1: scalar-to-scalar mapping, a linear approximation

- Life is not that simple..

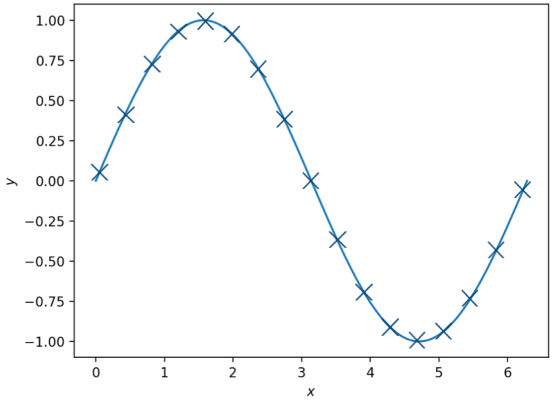


$$\mathcal{G}(x_t) = -0.30x_t + 0.95$$

The Last Layer

- Case study #2: scalar-to-scalar mapping using nonlinear feature transform

- It's meaningless to model a nonlinear function linearly
- Let's think of a nonlinear transformation of the input
  - So that the mapping between the features and the output can be linear
  - Any idea?
    - There are many different ways..
- I've got a feeling that this function looks like a cubic function
  - Because I was lucky to be able to visualize the data points
  - So, let's try a cubic transformation



$$[\hat{y_0}\ \hat{y_1}\ \hat{y_2}\ \cdots\ \hat{y_{T-1}}] = [a_3\ a_2\ a_1\ a_0] \begin{bmatrix} x_0^3 & x_1^3 & x_2^3 & \cdots & x_{T-1}^3 \\ x_0^2 & x_1^2 & x_2^2 & \cdots & x_{T-1}^2 \\ x_0 & x_1 & x_2 & \cdots & x_{T-1} \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix}$$

$$\Leftrightarrow \hat{\boldsymbol{y}}^\top = \boldsymbol{a}^\top \boldsymbol{X}$$

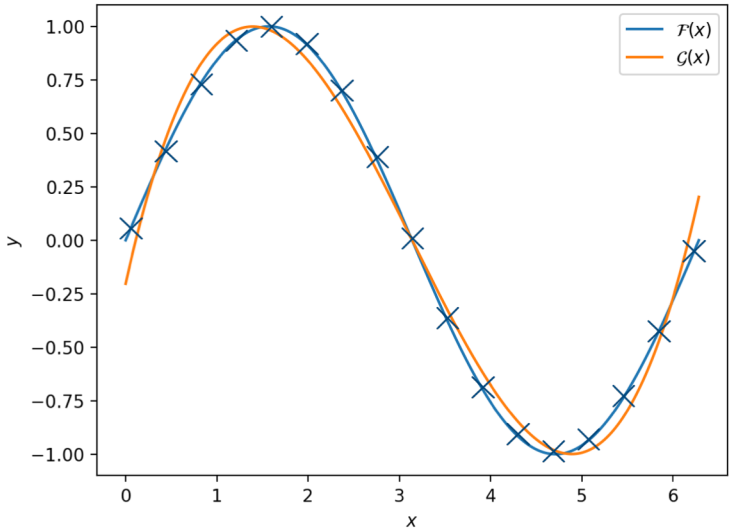
- How do we solve this?
  - Same as before

$$\boldsymbol{a} = (\boldsymbol{X}\boldsymbol{X}^\top)^{-1}\boldsymbol{X}\boldsymbol{y}^\top$$

The Last Layer

- Case study #2: scalar-to-scalar mapping using nonlinear feature transform

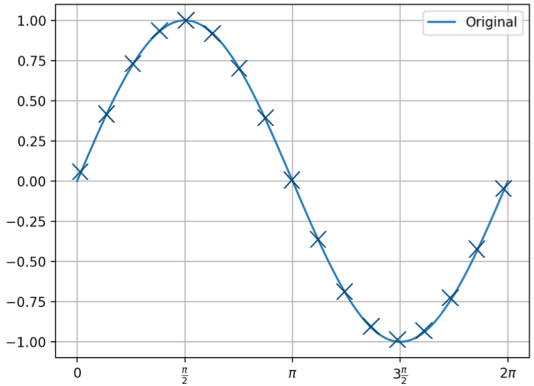
- So far so good..
- Do you see any problem in this approach?
  - You should know the answer if you took MLSP
- You don't always know the mapping function
  - So, I was lucky to know that a cubic function will work
- You should ask me a question at this point
  - "Is there any systematic way that can model *any* function?"
  - But hold on that question for now
- In the procedure we just saw
  - We transformed the 1D data samples into 3D features
  - Then the mapping procedure became a linear function

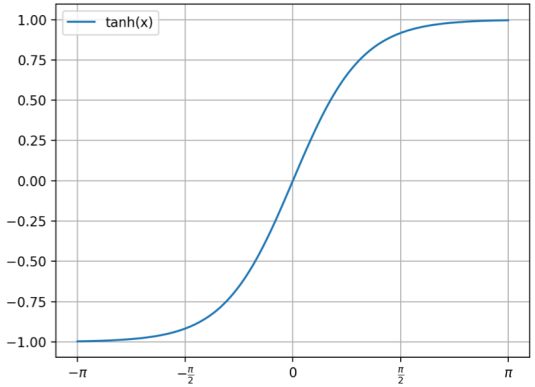


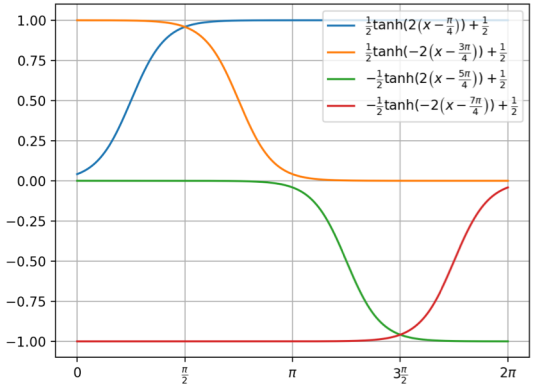
The Last Layer

- Case study #3: scalar-to-scalar mapping using *universal* nonlinear feature transform

- Another approach
- This time we're going to use another kind of transform
- Some observations
  - I see a hill and a valley
  - The hill: centered around $\frac{\pi}{2}$
  - The valley: centered around $\frac{3\pi}{2}$
- Let's create them
  - I will use sigmoid functions
  - I rescale and shift them around



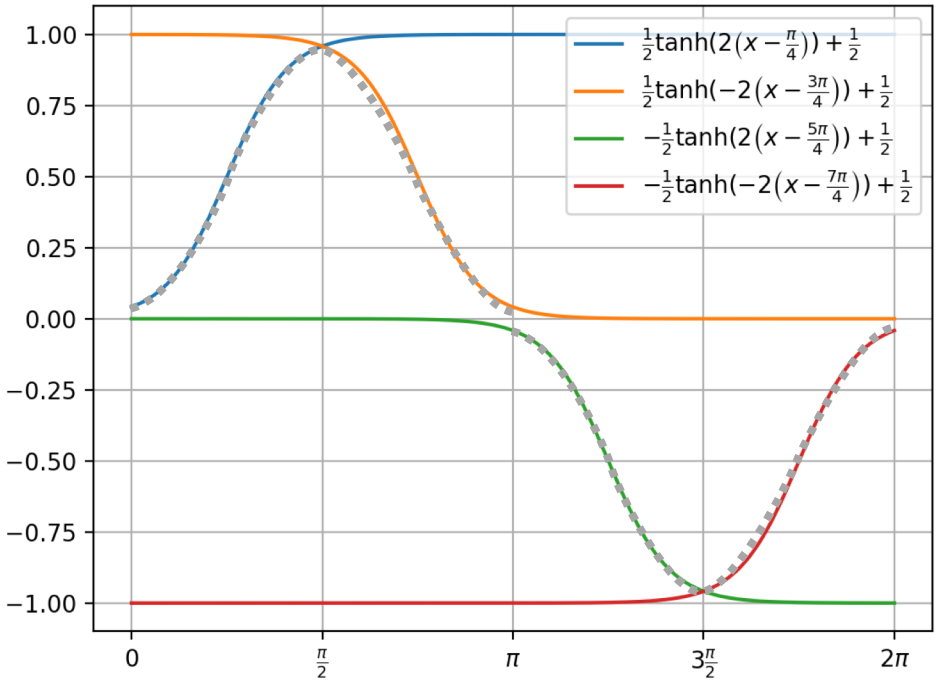




The Last Layer

- Case study #3: scalar-to-scalar mapping using *universal* nonlinear feature transform

○ If we can somehow combine the “features,” we might recover the original sine function



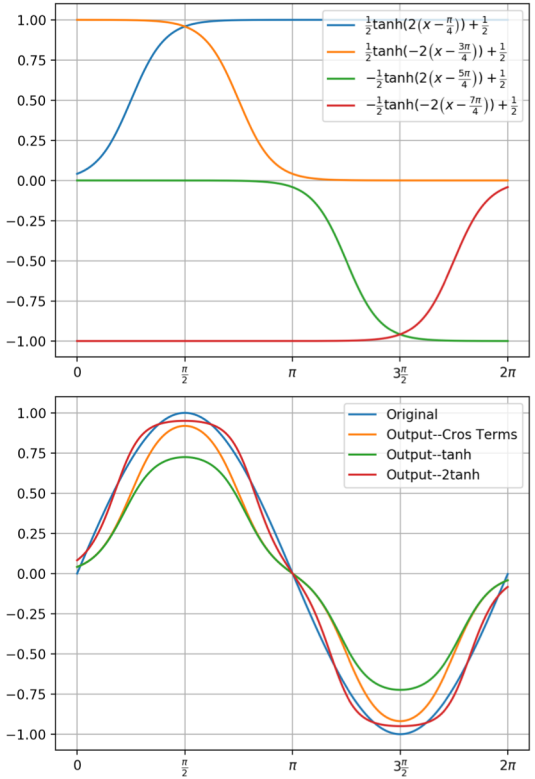
□ How?

The Last Layer

- Case study #3: scalar-to-scalar mapping using *universal* nonlinear feature transform

- Let's define our features first

$$h_1 = \frac{1}{2}\tanh(2\left(x - \frac{\pi}{4}\right)) + \frac{1}{2}$$
$$h_2 = \frac{1}{2}\tanh(-2\left(x - \frac{3\pi}{4}\right)) + \frac{1}{2}$$
$$h_3 = -\frac{1}{2}\tanh(2\left(x - \frac{5\pi}{4}\right)) + \frac{1}{2}$$
$$h_4 = -\frac{1}{2}\tanh(-2\left(x - \frac{7\pi}{4}\right)) + \frac{1}{2}$$



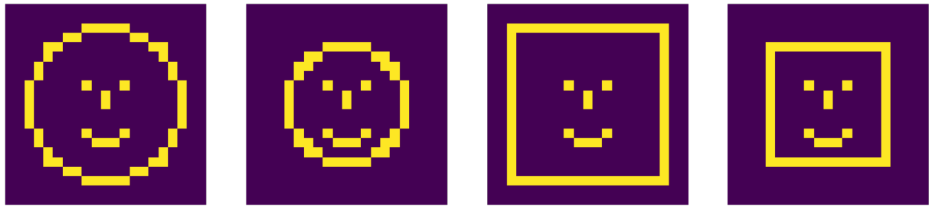
- Can you think of a way to form the sine function?
- $\hat{y} = h_1 h_2 - h_3 h_4$
- $\hat{y} = \tanh(h_1 + h_2 - 1) + \tanh(h_3 + h_4 + 1)$
- $\hat{y} = \tanh(2(h_1 + h_2 - 1)) + \tanh(2(h_3 + h_4 + 1))$
- We can combine a bunch of tanh functions to create valleys and hills!
- How do we find out those coefficients?
  - That's why we need optimization

The Last Layer

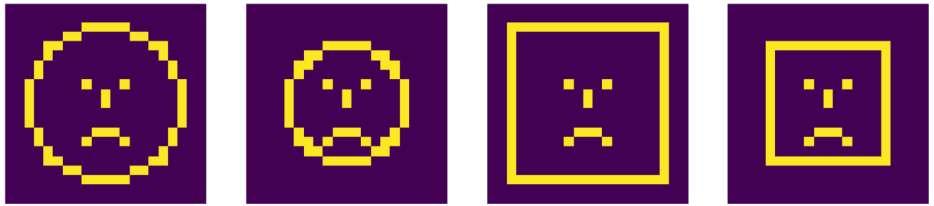
- Case study #4: vector-to-scalar mapping for binary classification

- Roughly speaking, classification can be seen as template matching
  - To see if there's a particular pattern in the example
- Can you figure out what kind of templates are there in the following images?

Class A



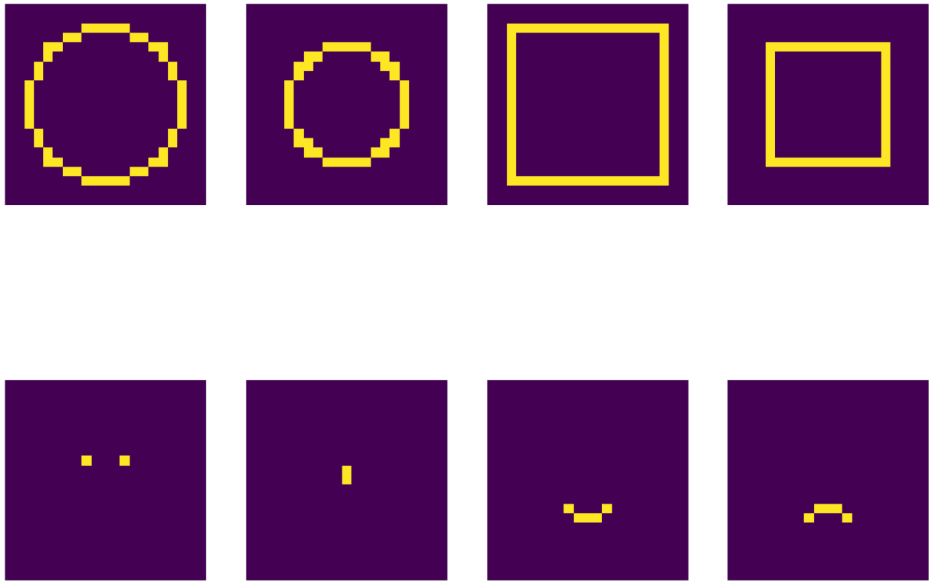
Class B



The Last Layer

- Case study #4: vector-to-scalar mapping for binary classification

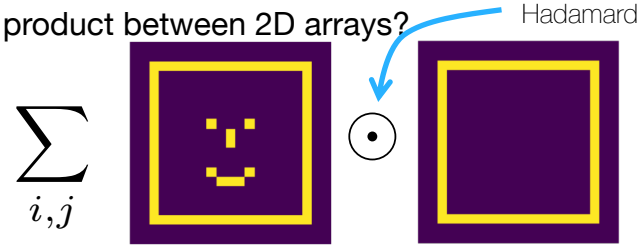
- I feel that there are eight different templates (and features drawn from them)
  - Which one is important for the classification problem?



The Last Layer

- Case study #4: vector-to-scalar mapping for binary classification

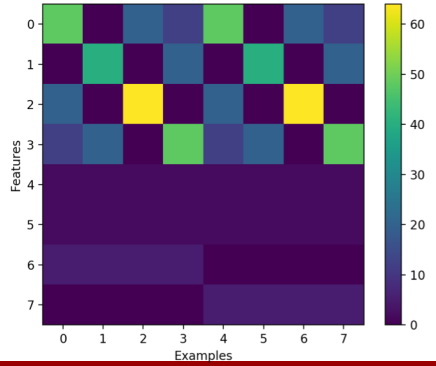
- Let's check out which one is important
- We do dot product between the template and the image sample
  - Dot product between 2D arrays?



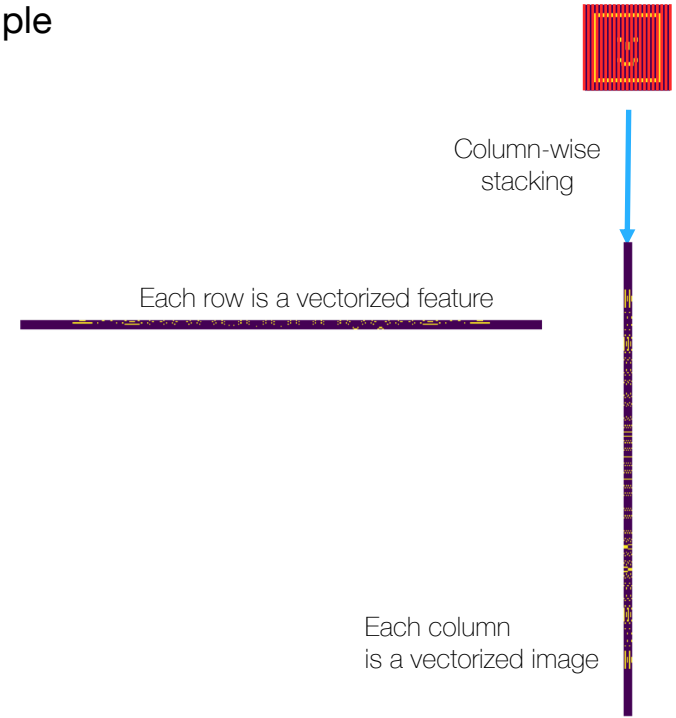
$$\sum_{i,j}$$

  - Vectorize and inner product

$$\boldsymbol{H} = \boldsymbol{W}^\top \boldsymbol{X}$$



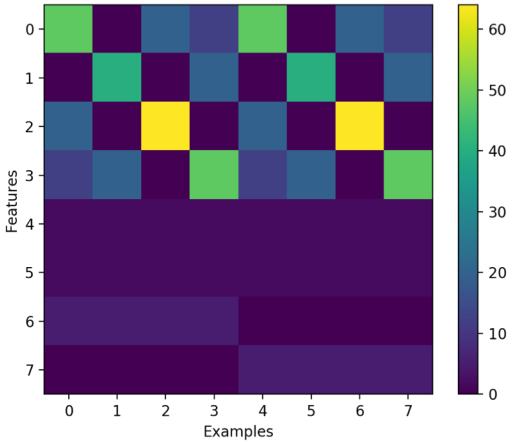
=



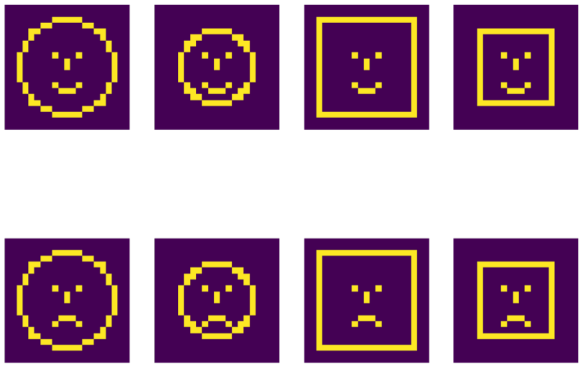
The Last Layer

- Case study #4: vector-to-scalar mapping for binary classification

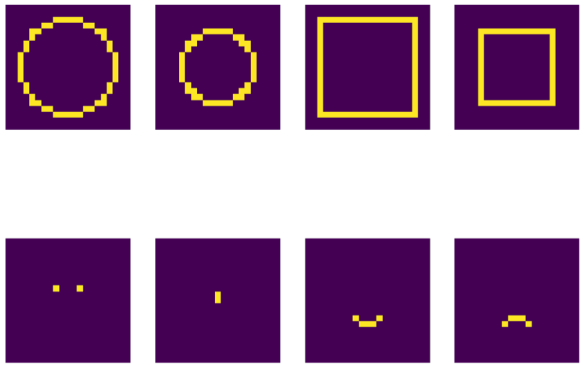
- What does this mean?



How many features were there originally?



Eight different templates form eight features

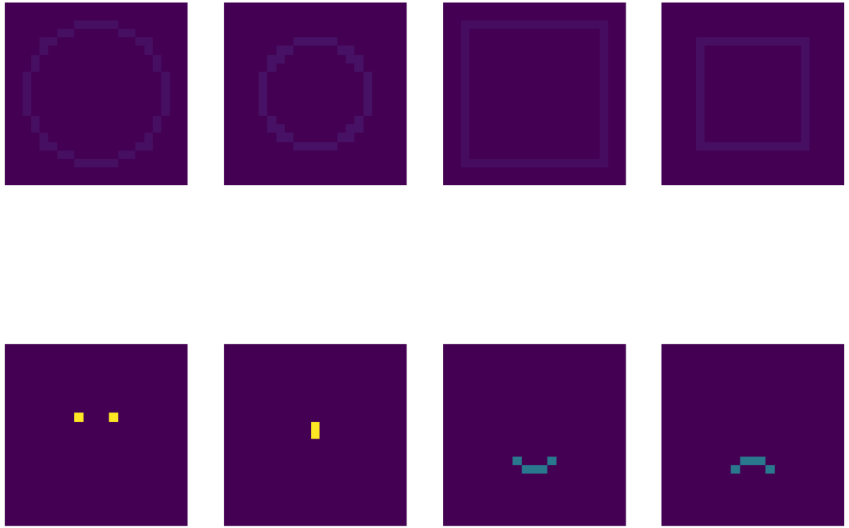


The Last Layer

- Case study #4: vector-to-scalar mapping for binary classification

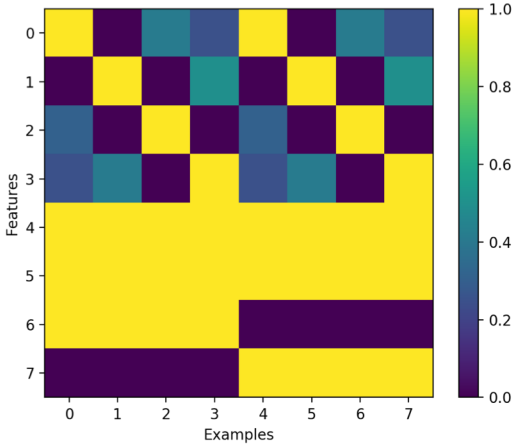
- I finally have a feeling that 6th and 7th features are important
  - But their activations are not strong enough
  - Why? How do we improve it?
- I normalized the templates so that $\boldsymbol{W}_{:,i}$ sums to one

$\tilde{\boldsymbol{W}}$



The Last Layer

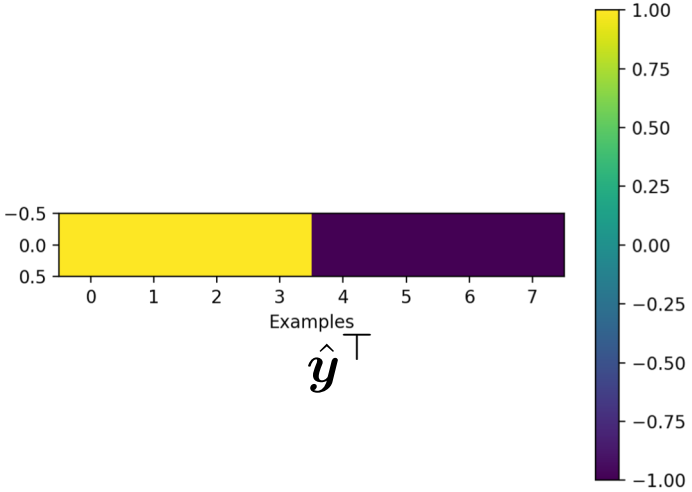
- Case study #4: vector-to-scalar mapping for binary classification



$$\boldsymbol{H} = \tilde{\boldsymbol{W}}^\top \boldsymbol{X}$$

- Then, how do I make a decision?
  - I'll figure out the way to focus only on those important ones
- One way: scalar output $\boldsymbol{w}^{(2)} = [0, 0, 0, 0, 0, 0, 1, -1]^\top$

$$\hat{\boldsymbol{y}}^\top = \boldsymbol{w}^{(2)\top} \boldsymbol{H}$$



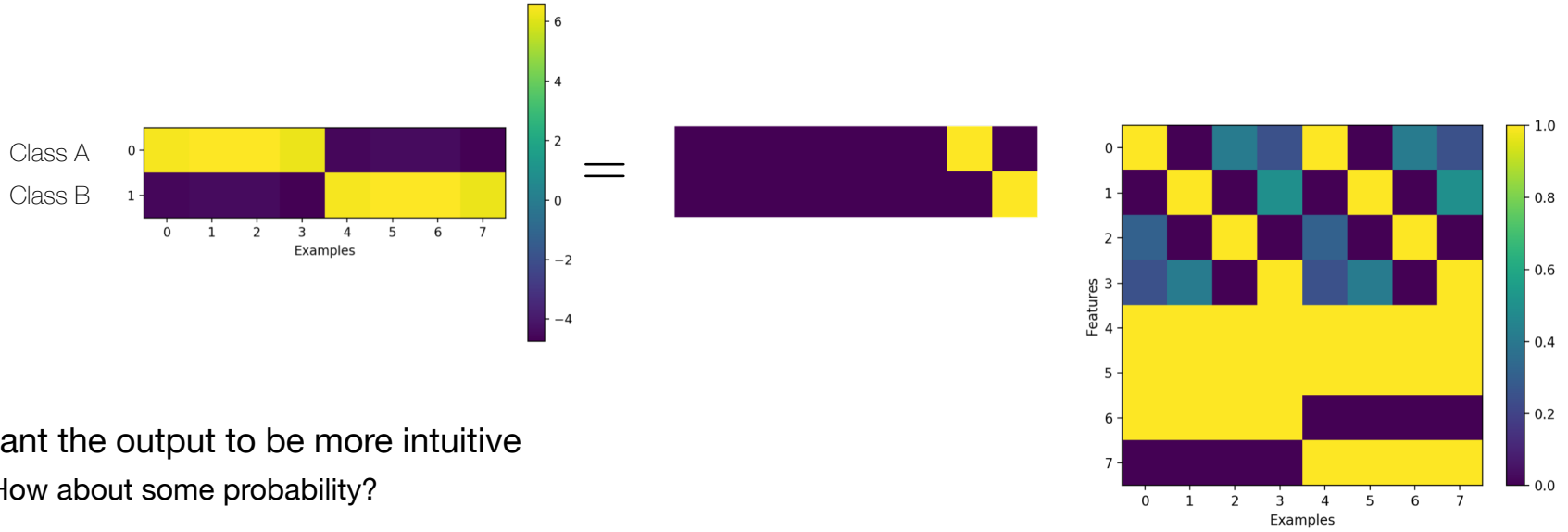
The Last Layer

- Case study #5: vector-to-vector mapping for multiclass classification

- As for the binary classification there's another way to encode the output

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & 10 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & 10 \end{bmatrix}$$

$$\hat{\boldsymbol{Y}} = \boldsymbol{W}^{(2)}\boldsymbol{H}$$



- I want the output to be more intuitive
  - How about some probability?

The Last Layer

- Case study #5: vector-to-vector mapping for multiclass classification

- Logistic regression
  - We want the prediction to be probability vectors

$$P(\text{Class A}|\boldsymbol{X}_{:,t}) \neq \hat{\boldsymbol{Y}}_{0,t} = \boldsymbol{W}^{(2)}_{0,:}\boldsymbol{H}_{:,t}$$

$$P(\text{Class B}|\boldsymbol{X}_{:,t}) \neq \hat{\boldsymbol{Y}}_{1,t} = \boldsymbol{W}^{(2)}_{1,:}\boldsymbol{H}_{:,t}$$

Why not? Can be negative

$$P(\text{Class A}|\boldsymbol{X}_{:,t}) \neq \exp(\hat{\boldsymbol{Y}}_{0,t}) = \exp(\boldsymbol{W}^{(2)}_{0,:}\boldsymbol{H}_{:,t})$$

$$P(\text{Class B}|\boldsymbol{X}_{:,t}) \neq \exp(\hat{\boldsymbol{Y}}_{1,t}) = \exp(\boldsymbol{W}^{(2)}_{1,:}\boldsymbol{H}_{:,t})$$

Why not? Can be larger than one

$$P(\text{Class A}|\boldsymbol{X}_{:,t}) = \frac{\exp(\hat{\boldsymbol{Y}}_{0,t})}{\exp(\hat{\boldsymbol{Y}}_{0,t}) + \exp(\hat{\boldsymbol{Y}}_{1,t})}$$

$$P(\text{Class B}|\boldsymbol{X}_{:,t}) = \frac{\exp(\hat{\boldsymbol{Y}}_{1,t})}{\exp(\hat{\boldsymbol{Y}}_{0,t}) + \exp(\hat{\boldsymbol{Y}}_{1,t})}$$

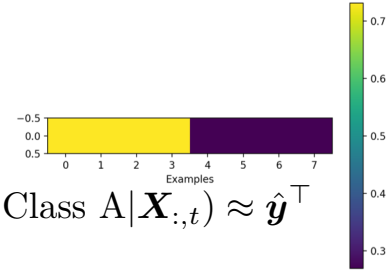
- $\boldsymbol{W}^{(2)}$ is overparameterized and we can simplify it

$$\frac{\exp(\hat{\boldsymbol{Y}}_{0,t})}{\exp(\hat{\boldsymbol{Y}}_{0,t}) + \exp(\hat{\boldsymbol{Y}}_{1,t})} = \frac{1}{1 + \exp(\hat{\boldsymbol{Y}}_{1,t} - \hat{\boldsymbol{Y}}_{0,t})} = \frac{1}{1 + \exp((\boldsymbol{W}^{(2)}_{1,:} - \boldsymbol{W}^{(2)}_{0,:})\boldsymbol{H}_{:,t})} = \frac{1}{1 + \exp(\boldsymbol{w}^{(2)\top}\boldsymbol{H}_{:,t})}$$

← Logistic Function

$$\frac{\exp(\hat{\boldsymbol{Y}}_{1,t})}{\exp(\hat{\boldsymbol{Y}}_{0,t}) + \exp(\hat{\boldsymbol{Y}}_{1,t})} = \frac{\exp(\hat{\boldsymbol{Y}}_{1,t} - \hat{\boldsymbol{Y}}_{0,t}))}{1 + \exp(\hat{\boldsymbol{Y}}_{1,t} - \hat{\boldsymbol{Y}}_{0,t}))} = \frac{\exp((\boldsymbol{W}^{(2)}_{1,:} - \boldsymbol{W}^{(2)}_{0,:})\boldsymbol{H}_{:,t})}{1 + \exp((\boldsymbol{W}^{(2)}_{1,:} - \boldsymbol{W}^{(2)}_{0,:})\boldsymbol{H}_{:,t})} = \frac{\exp(\boldsymbol{w}^{(2)\top}\boldsymbol{H}_{:,t})}{1 + \exp(\boldsymbol{w}^{(2)\top}\boldsymbol{H}_{:,t})}$$

$$\boldsymbol{w}^{(2)} = \boldsymbol{W}^{(2)}_{1,:} - \boldsymbol{W}^{(2)}_{0,:}$$



$$P(\text{Class A}|\boldsymbol{X}_{:,t}) \approx \hat{\boldsymbol{y}}^{\top}$$

- This gives us a hint about multiclass classification

The Last Layer

- Case study #5: vector-to-vector mapping for multiclass classification

- Softmax regression
  - Again, we want the output to be a probabilistic vector

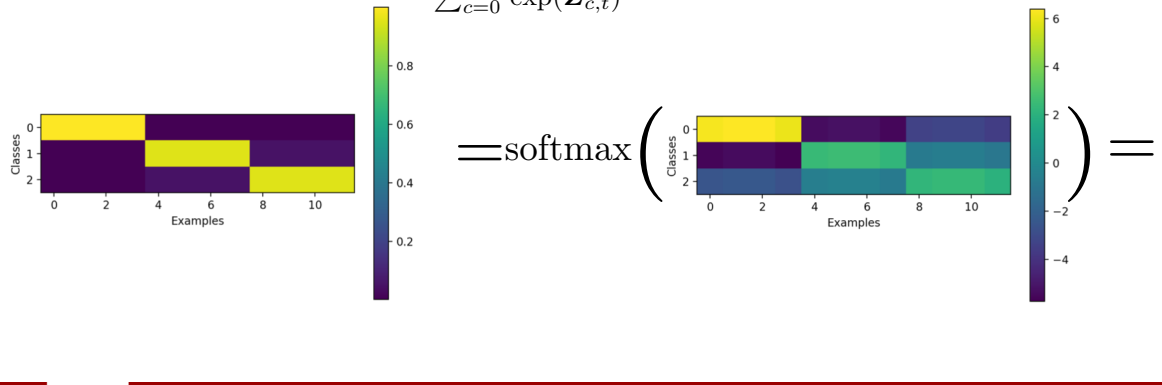
$$\boldsymbol{Z}_{c,t} = \boldsymbol{W}^{(2)}_{c,:}\boldsymbol{H}_{:,t}$$

$$P(\text{Class } \#0|\boldsymbol{X}_{:,t}) = \frac{\exp(\boldsymbol{Z}_{0,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

$$P(\text{Class } \#1|\boldsymbol{X}_{:,t}) = \frac{\exp(\boldsymbol{Z}_{1,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

$$\cdots$$

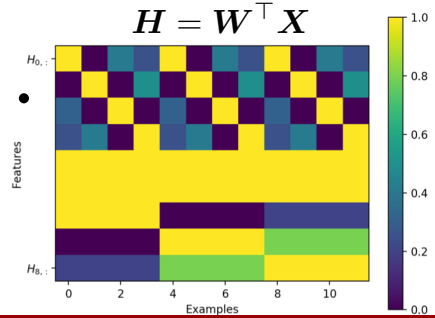
$$P(\text{Class } \#\mathcal{C}-1|\boldsymbol{X}_{:,t}) = \frac{\exp(\boldsymbol{Z}_{\mathcal{C}-1,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$



$$= \boldsymbol{W}^{(2)}$$

New features

$$\boldsymbol{H} = \boldsymbol{W}^{\top}\boldsymbol{X}$$



The Last Layer

- Take-home messages

- If you know good features things are easier
  - For regression, you can combine hills and valleys to approximate any funky shape
  - For classification, good features can be created from template matching
    - If you know a good set of templates for the classification

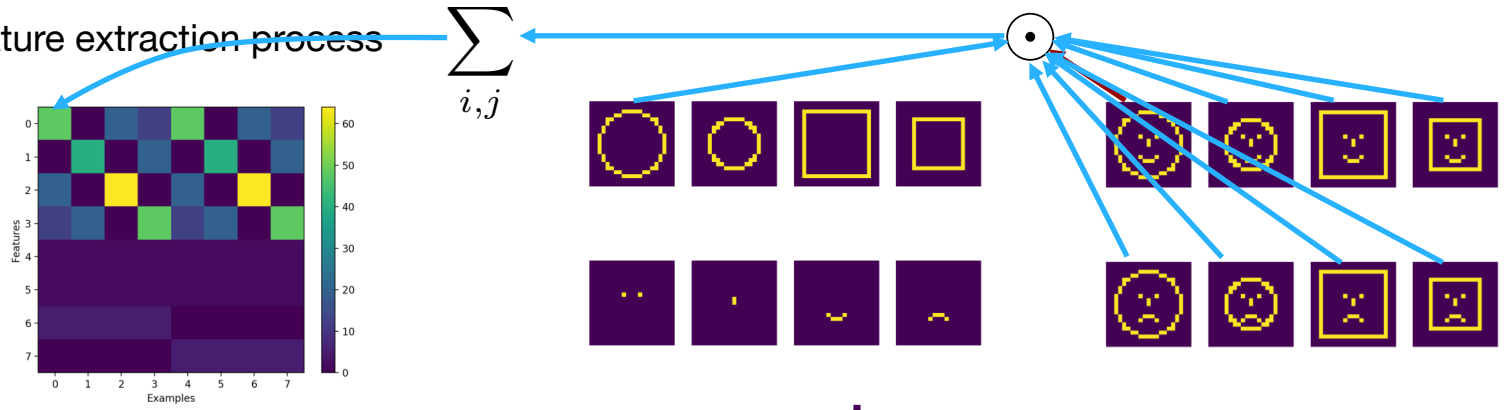
- Some details
  - Sigmoid functions are useful and versatile to create hills and valleys
  - Softmax and logistic functions are useful for the posterior-probability-like output variables

- Potential questions
  - How do we find out those features?
    - Note that I manually found them for this lecture

Baby Optimization

- The closed form solution

- Recall the feature extraction process



- Or,

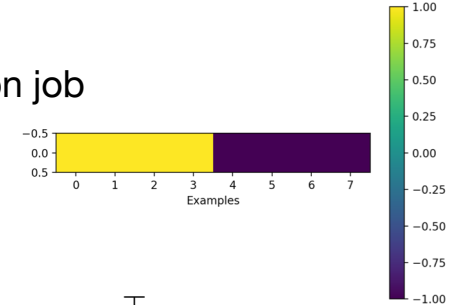
$$\boldsymbol{H} = \boldsymbol{W}^\top \boldsymbol{X}$$

Baby Optimization

- The closed form solution

- Having $\boldsymbol{H}$ as the new input we can find another set of weights for the classification job
- Recall that I magically manually found out them

$$\boldsymbol{w}^{(2)} = [0,0,0,0,0,0,1,-1]^\top \qquad \hat{\boldsymbol{y}}^\top = {\boldsymbol{w}^{(2)}}^\top \boldsymbol{H}$$



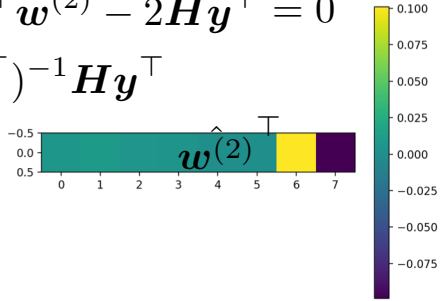
- If you choose to do optimization we have the pseudo inversed-based solution

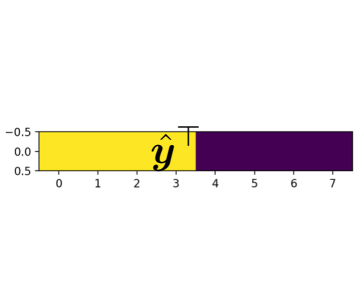
$$\arg\min_{\boldsymbol{w}^{(2)}} \sum_t \mathcal{D}(y_t||\hat{y}_t) = \arg\min_{\boldsymbol{w}^{(2)}} \sum_t (y_t - \hat{y}_t)^2 = \arg\min_{\boldsymbol{w}^{(2)}} (\boldsymbol{y}^\top - {\boldsymbol{w}^{(2)}}^\top \boldsymbol{H})(\boldsymbol{y}^\top - {\boldsymbol{w}^{(2)}}^\top \boldsymbol{H})^\top$$

$$= \arg\min_{\boldsymbol{w}^{(2)}} (\boldsymbol{y}^\top \boldsymbol{y} + {\boldsymbol{w}^{(2)}}^\top \boldsymbol{H}\boldsymbol{H}^\top \boldsymbol{w}^{(2)} - 2\boldsymbol{y}^\top \boldsymbol{H}^\top \boldsymbol{w}^{(2)})$$

$$\frac{\partial \boldsymbol{y}^\top \boldsymbol{y} + {\boldsymbol{w}^{(2)}}^\top \boldsymbol{H}\boldsymbol{H}^\top \boldsymbol{w}^{(2)} - 2\boldsymbol{y}^\top \boldsymbol{H}^\top \boldsymbol{w}^{(2)}}{\partial \boldsymbol{w}^{(2)}} = 2\boldsymbol{H}\boldsymbol{H}^\top \boldsymbol{w}^{(2)} - 2\boldsymbol{H}\boldsymbol{y}^\top = 0$$

$$\boldsymbol{w}^{(2)} = (\boldsymbol{H}\boldsymbol{H}^\top)^{-1}\boldsymbol{H}\boldsymbol{y}^\top$$





- I define my target output

$$\boldsymbol{y} = [1,1,1,1,0,0,0,0]^\top$$

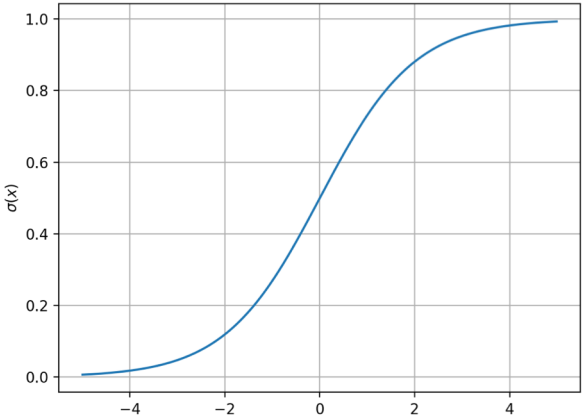
Baby Optimization

- Gradient descent: logistic regression

- Was it too easy?
  - Let me make it a little more difficult
- What I'm concerned is the range of $\hat{\boldsymbol{y}}$, the least mean squared error solution

$$\hat{\boldsymbol{y}}^\top = \boldsymbol{w}^{(2)^\top} \boldsymbol{H} \qquad \hat{\boldsymbol{y}} = \left((\boldsymbol{H}\boldsymbol{H}^\top)^{-1} \boldsymbol{H}\boldsymbol{y}^\top \right)^\top \boldsymbol{H}$$

  - There's no guarantee that $\hat{\boldsymbol{y}}$ is between 0 and 1
- Why BETWEEN 0 and 1?
  - We care about the probability $P(\text{Class A} | \boldsymbol{X}_{:,t})$
  - If you're absolutely sure, then this value will be 1 or 0 as in the target variable $\boldsymbol{y}$
  - In other words, your training samples are with absolutely sure class labels
  - While for your test samples you need to predict the labels by using a posterior probability of the label given the data point
- Any idea?
  - Hint: You Already Learned This (YALT)
  - Logistic function!



Baby Optimization

- Gradient descent: logistic regression

- So, we wrap the output values with the logistic sigmoid function

$$\boldsymbol{z}^\top = \boldsymbol{w}^{(2)^\top} \boldsymbol{H} \qquad \hat{\boldsymbol{y}}^\top = \sigma\left(\boldsymbol{z}^\top\right)$$

Note that the sigmoid function works element-wise

- Closed form solution?

$$\arg\min_{\boldsymbol{w}^{(2)}} \sum_t (y_t - \hat{y}_t)^2 = \arg\min_{\boldsymbol{w}^{(2)}} \left(\boldsymbol{y}^\top \boldsymbol{y} + \sigma\left(\boldsymbol{w}^{(2)^\top} \boldsymbol{H}\right) \sigma\left(\boldsymbol{H}^\top \boldsymbol{w}^{(2)}\right) - 2\boldsymbol{y}^\top \sigma\left(\boldsymbol{H}^\top \boldsymbol{w}^{(2)}\right)\right)$$

  - Maybe not a quadratic function anymore

- What does this mean (I mean 'no closed form solution')?
  - There can be many stationary points
  - Impossible to find the global optimum
  - Welcome to the machine learning world!

- Instead, we do numerical optimization
  - We start from randomly initialized parameters
  - Check the error using the current parameters
  - Find the negative gradient direction that reduce the error
  - Update the parameters using the negative gradient direction

Baby Optimization

- Gradient descent: logistic regression

- i=0
  1. Initialize parameters with small random numbers
  2. Calculate the output using all training samples (i.e. input and target pairs)

$$\boldsymbol{z}^\top = \boldsymbol{w}^{(2)^\top} \boldsymbol{H} \qquad \hat{\boldsymbol{y}}^\top = \sigma\left(\boldsymbol{z}^\top\right)$$

  3. Calculate the error (cost)

$$\mathcal{E} = \sum_t (y_t - \hat{y}_t)^2$$

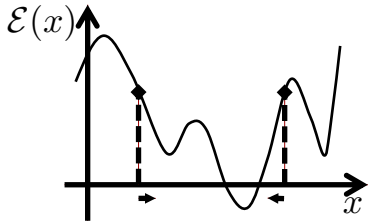
  4. Update the parameters

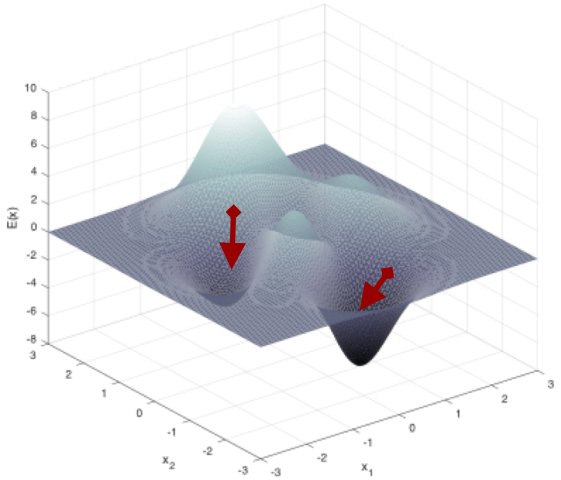
$$\boldsymbol{w}^{(2)} \leftarrow \boldsymbol{w}^{(2)} - \rho \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}^{(2)}}$$

Learning rate: how much to move?

Gradient: in which direction?

- i>0
  - Repeat 2-4
- You should ask two questions
  - What is ρ?
    - Learning rate
  - How do we calculate $\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}^{(2)}}$?
    - It depends on the problem
    - But basically by doing differentiation





Baby Optimization

- Gradient descent: logistic regression

$$z_t = \boldsymbol{w}^{(2)\top} \boldsymbol{H}_{:,t}$$

$$\hat{y}_t = \sigma(z_t)$$

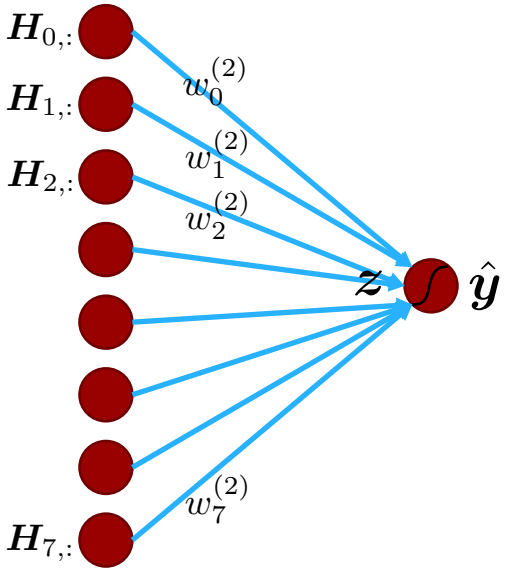
$$\mathcal{E} = \sum_t (y_t - \hat{y}_t)^2$$

- So, let's do the differentiation
- The chain rule

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}^{(2)}} = \sum_t \frac{\partial \mathcal{E}}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial z_t} \frac{\partial z_t}{\partial \boldsymbol{w}^{(2)}} = \sum_t 2(\hat{y}_t - y_t)\, \sigma'(z_t)\, \boldsymbol{H}_{:,t}$$

$$= \boldsymbol{H}\underbrace{\left((2\hat{\boldsymbol{y}} - 2\boldsymbol{y}) \odot \sigma'(\boldsymbol{z})\right)}_{\text{BP error}}$$

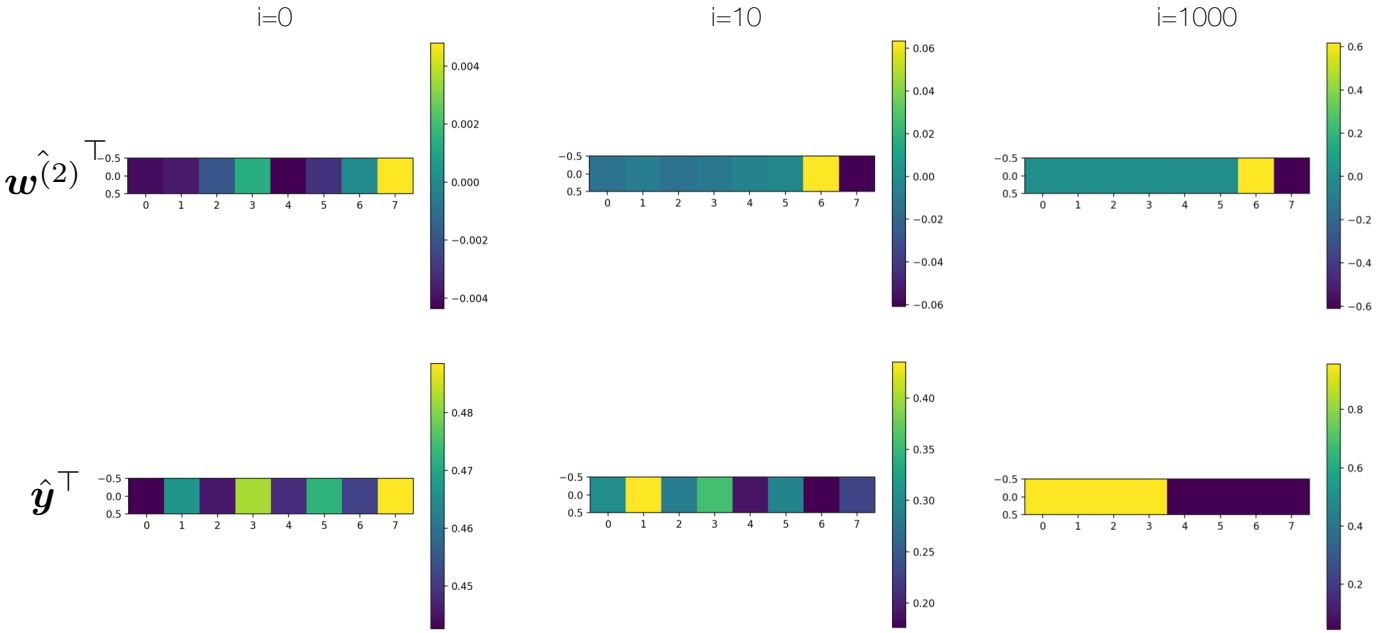
- The gradient direction for the weights is defined by
  - The inner product of the data matrix and the **backpropagation error**
- Backpropagation error?
  - Partial derivative of the total cost w.r.t. the input to a node
  - In this case, $\dfrac{\partial \mathcal{E}}{\partial z_t}$

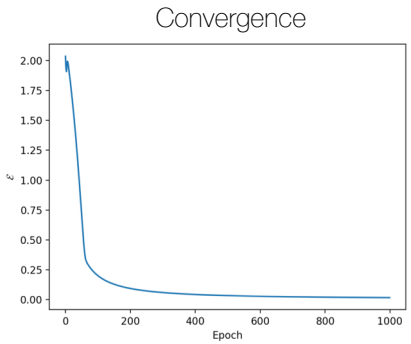


Baby Optimization

- Gradient descent: logistic regression

- Parameters and predictions over epochs





Baby Optimization

- Gradient descent: softmax

$$P(\text{Class } \#0|\boldsymbol{X}_{:,t}) = \frac{\exp(\boldsymbol{Z}_{0,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

$$P(\text{Class } \#1|\boldsymbol{X}_{:,t}) = \frac{\exp(\boldsymbol{Z}_{1,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

$$\cdots$$

$$P(\text{Class } \#\mathcal{C}-1|\boldsymbol{X}_{:,t}) = \frac{\exp(\boldsymbol{Z}_{\mathcal{C}-1,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

Summation over **c** makes differentiation difficult

- Softmax regression is special
  - So is logistic regression
  - The output vector sums to one and nonnegative
    - A probabilistic distribution over classes
- Sum of squared error might not be the best
- Then, why not using a more suitable error metric?
  - Cross entropy $-\sum_i q(x_i)\log p(x_i)$

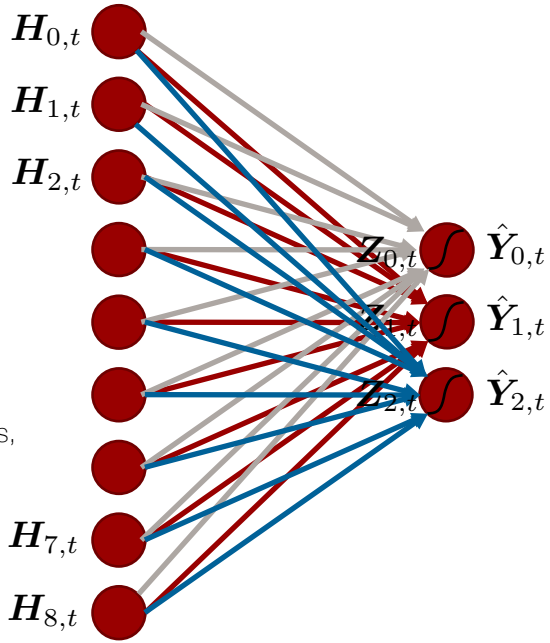
$$\mathcal{E} = -\sum_t\sum_c \boldsymbol{Y}_{c,t}\log\hat{\boldsymbol{Y}}_{c,t}$$

- Eventually, we want a partial differentiation of $\mathcal{E}$, but it's not that simple

$$\frac{\partial\mathcal{E}}{\partial\boldsymbol{W}^{(2)}_{j,:}} = \sum_t\sum_c \frac{\partial\mathcal{E}}{\partial\hat{\boldsymbol{Y}}_{c,t}}\frac{\partial\hat{\boldsymbol{Y}}_{c,t}}{\partial\boldsymbol{Z}_{j,t}}\frac{\partial\boldsymbol{Z}_{j,t}}{\partial\boldsymbol{W}^{(2)}_{j,:}}$$

Note: **j** and **c** are the same index for the three output units, but **j** is for the input to the units and **c** is for the output. For example, $\mathbf{W}^{(2)}_{0,:}$ is for the gray arrows

$$\frac{\partial\mathcal{E}}{\partial\hat{\boldsymbol{Y}}_{c,t}} = -\frac{\boldsymbol{Y}_{c,t}}{\hat{\boldsymbol{Y}}_{c,t}} \qquad \frac{\partial\hat{\boldsymbol{Y}}_{c,t}}{\partial\boldsymbol{Z}_{j,t}} = ? \qquad \frac{\partial\boldsymbol{Z}_{j,t}}{\partial\boldsymbol{W}^{(2)}_{j,:}} = \boldsymbol{H}^{\top}_{:,t}$$



Baby Optimization

- Gradient descent: softmax

$$P(\text{Class } \#0|\boldsymbol{X}_{:,t}) = \hat{\boldsymbol{Y}}_{0,t} = \frac{\exp(\boldsymbol{Z}_{0,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

$$P(\text{Class } \#1|\boldsymbol{X}_{:,t}) = \hat{\boldsymbol{Y}}_{1,t} = \frac{\exp(\boldsymbol{Z}_{1,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

$$\dots$$

$$P(\text{Class } \#\mathcal{C}-1|\boldsymbol{X}_{:,t}) = \hat{\boldsymbol{Y}}_{\mathcal{C}-1,t} = \frac{\exp(\boldsymbol{Z}_{\mathcal{C}-1,t})}{\sum_{c=0}^{\mathcal{C}-1}\exp(\boldsymbol{Z}_{c,t})}$$

- Any $\hat{\boldsymbol{Y}}_{c,t}$ involves all $\boldsymbol{Z}_{j,t}$ $\forall j$

$$\frac{\partial \hat{\boldsymbol{Y}}_{c,t}}{\partial \boldsymbol{Z}_{j,t}} = \frac{\exp(\boldsymbol{Z}_{c,t})\left(\sum_i \exp(\boldsymbol{Z}_{i,t})\right) - \left(\exp(\boldsymbol{Z}_{c,t})\right)^2}{\left(\sum_i \exp(\boldsymbol{Z}_{i,t})\right)^2} \quad \text{if } j = c$$

$$= \hat{\boldsymbol{Y}}_{c,t}(1 - \hat{\boldsymbol{Y}}_{c,t}) = \hat{\boldsymbol{Y}}_{j,t}(1 - \hat{\boldsymbol{Y}}_{j,t})$$

$$\frac{\partial \hat{\boldsymbol{Y}}_{c,t}}{\partial \boldsymbol{Z}_{j,t}} = -\frac{\exp(\boldsymbol{Z}_{c,t})\exp(\boldsymbol{Z}_{j,t})}{\left(\sum_i \exp(\boldsymbol{Z}_{i,t})\right)^2} \quad \text{if } j \neq c$$

$$= -\hat{\boldsymbol{Y}}_{c,t}\hat{\boldsymbol{Y}}_{j,t}$$

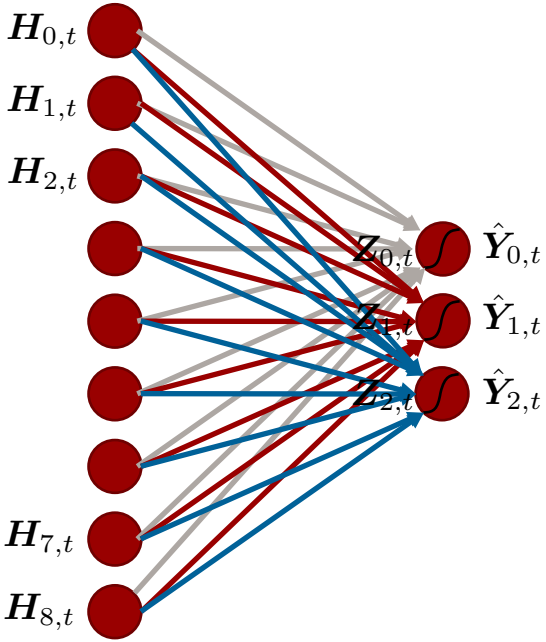
$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(2)}_{j,:}} = \sum_t \sum_c \frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{Y}}_{c,t}} \frac{\partial \hat{\boldsymbol{Y}}_{c,t}}{\partial \boldsymbol{Z}_{j,t}} \frac{\partial \boldsymbol{Z}_{j,t}}{\partial \boldsymbol{W}^{(2)}_{j,:}}$$

$$= \sum_t \left(-\frac{\boldsymbol{Y}_{j,t}}{\hat{\boldsymbol{Y}}_{j,t}} \hat{\boldsymbol{Y}}_{j,t}(1 - \hat{\boldsymbol{Y}}_{j,t}) + \sum_{c \neq j} \frac{\boldsymbol{Y}_{c,t}}{\hat{\boldsymbol{Y}}_{c,t}} \hat{\boldsymbol{Y}}_{c,t}\hat{\boldsymbol{Y}}_{j,t} \right) \boldsymbol{H}^\top_{:,t}$$

$$= \sum_t \left(-\boldsymbol{Y}_{j,t}(1 - \hat{\boldsymbol{Y}}_{j,t}) + \sum_{c \neq j} \boldsymbol{Y}_{c,t}\hat{\boldsymbol{Y}}_{j,t} \right) \boldsymbol{H}^\top_{:,t}$$

$$= \sum_t \left(-\boldsymbol{Y}_{j,t} + \sum_c \boldsymbol{Y}_{c,t}\hat{\boldsymbol{Y}}_{j,t} \right) \boldsymbol{H}^\top_{:,t} = (\hat{\boldsymbol{Y}}_{j,:} - \boldsymbol{Y}_{j,:})\boldsymbol{H}^\top$$

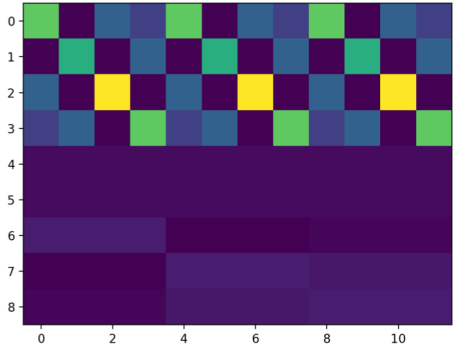
$$\boxed{\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(2)}} = (\hat{\boldsymbol{Y}} - \boldsymbol{Y})\boldsymbol{H}^\top}$$



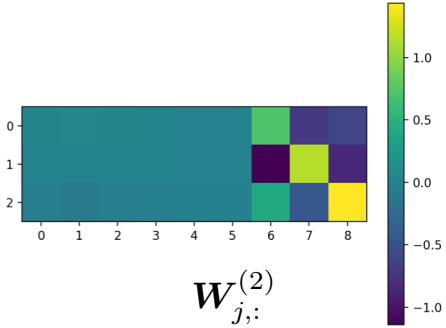
Baby Optimization

- Gradient descent: softmax

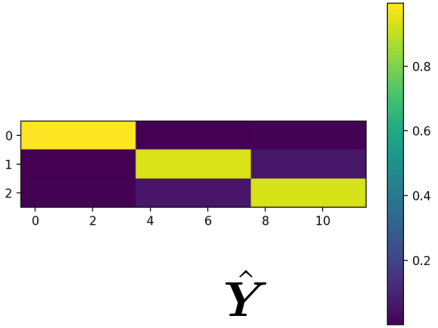
- Parameter estimates: the three-class case



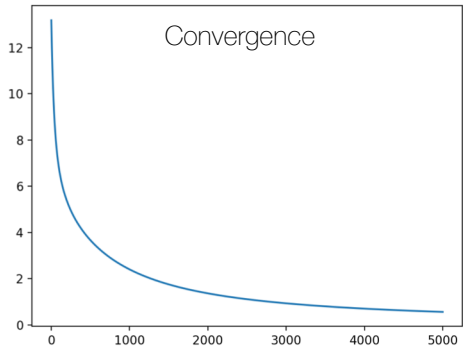
$$\boldsymbol{H} = \boldsymbol{W}^\top \boldsymbol{X}$$



$$\boldsymbol{W}^{(2)}_{j,:}$$



$$\hat{\boldsymbol{Y}}$$



Take-home Messages

- Parameter estimation for nonlinear models involve complicated optimization procedure
  - No global optimum
  - Differentiation
  - Learning rate
  - Gradient directions
  - Initialization
  - Computational cost
- In the last layer the choice of the activation function and cost function matters
  - You want to transform the output variable so that
    - Its dynamic range matches the target variable
  - The cost function needs to be chosen accordingly

Deep Learning Frameworks

- GPU computing

- A little bit about GPU computing
  - GPUs are different from CPUs in terms of their number of cores
    - K80 has 2496 cores
    - A better choice for a task with many simple sub-tasks, such as matrix multiplication
    - Each core is much less powerful than a usual CPU core
- Common packages you might need
  - Tensorflow or PyTorch
    - You may want to use some even higher lever wrappers like Keras, but I wouldn't cover them during the class
  - Jupyter
    - A nice interactive development framework
    - Google Colab comes with something similar to this

Deep Learning Frameworks

- Baby TensorFlow

- **Computational graphs**
  - That consist of tensors and operations on them
- Building and running the computational graphs are separate
  - You define the graph first
    - This procedure is to represent what you want to do symbolically
    - It doesn't do any computation
  - Running the graph
    - Visit the graph nodes as required by the session's **run** method
    - Actually does the computation
- What for?
  - One obvious reason is to make your life easier by doing differentiation for you
    - So, it does the differentiation on the symbolic representations
  - And then actually calculate the gradient when you run the graph

Deep Learning Frameworks

- Baby TensorFlow

- Three different kinds of **Tensors**
  - `tf.Variable`: something TF can change
  - `tf.placeholder`: something you can change
  - `tf.constant`: something nobody doesn't change once initialized
- For example,
  - You want to train a classifier from a very large training dataset

This will be a very large data matrix

$$\boldsymbol{Y} \approx \mathcal{G}(\boldsymbol{X}; \mathbb{W})$$

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{(2)}} = (\hat{\boldsymbol{Y}} - \boldsymbol{Y})\boldsymbol{H}^{\top}$$

A heavy matrix operation, maybe too large for the main memory

  - So, you want to do something called **Stochastic** Gradient Descent (SGD), where
    - You sample a data point and calculate the gradient by using it
    - Or, you divide the entire data matrix into smaller pieces, called **minibatches** and calculate gradients for each of them
  - Then,
    - `tf.Variable`: $\boldsymbol{W}^{(2)}$ ← You can set TF to update them during the run (of course you can initialize them as you want)
    - `tf.placeholder`: $\boldsymbol{H}_{:,t}, \boldsymbol{Y}$ ← You need to feed this SGD samples or minibatches at every iteration
    - `tf.constant`: vector of ones for bias ← Once initialized nobody cares about this

Deep Learning Frameworks

- Baby TensorFlow

- Running the graph: if everything is symbolic, when do we actually do the operation?
  - In TF, there's a concept called **session** where all the variables and operations are actually taken care of
  - `tf.session.run(fetches, feed_dict=None, ...)`
    - `fetches`: The graph element you want to run
    - `feed_dict`: You can assign some values to the `placeholder` tensors
- Chain reaction
  - Running a graph element means running all the operations and evaluating tensors that are necessary for the fetched one
- For example,
  - You are interested in the training cost $\mathcal{E} = -\sum_t \sum_c \boldsymbol{Y}_{c,t} \log \hat{\boldsymbol{Y}}_{c,t}$
  - You define this cost as a graph element
  - If you do `sess.run()` on it, you need to know $\hat{\boldsymbol{Y}}$, the prediction of the class labels
  - `sess.run()` traverses the graph to get this done
- `tf.gradients(cost, variables)`
  - Does the differentiation w.r.t. the variables
  - Most of the time you'll use a wrapper that calls this function internally

Deep Learning Frameworks

- Baby PyTorch

- Difference between TF and PyTorch
- The look
  - TF: you construct the graph first and then run the graph
  - PT: it does construct the graph but things are more seamless
    - Therefore the source code is more similar to the regular numpy codes
- The tensors
  - TF: there are three different types of tensors
  - PT: a tensor is basically an N-dimensional array that has nothing to do with deep learning
    - But you can specify a tensor in the GPU memory if you want
    - `torch.autograd.Variable` defines a wrapper that turns a tensor into a PT variable
      - With which you can do all the cool things like automatic gradient calculation
- GPU computing
  - TF: if the same operation has two kernels for both CPU and GPU computing, GPU version gets the priority
  - PT: there are some predefined GPU data types and a way to convert tensors
    - e.g. `torch.FloatTensor` versus `torch.cuda.FloatTensor`
    - `cuda()`: copies the tensor in the GPU memory

Deep Learning Frameworks

- Baby PyTorch

- The gradients
  - TF: `tf.gradients(cost, variables)` differentiates the cost function w.r.t. the variable during the graph construction
    - The procedure is symbolic, so are the derivatives
  - PT: all variables have their own `.grad` component that holds the actual gradient values
    - It's called "autograd" so things are still automatic, but treated differently
    - When `.backward()` method is called, the node is differentiated w.r.t. the leaf nodes in a recursive way
    - To do so, the forward pass records the input to the node
    - The backward pass calculate the gradients of all the intermediate nodes on the way back

Deep Learning Frameworks

- TF versus PT

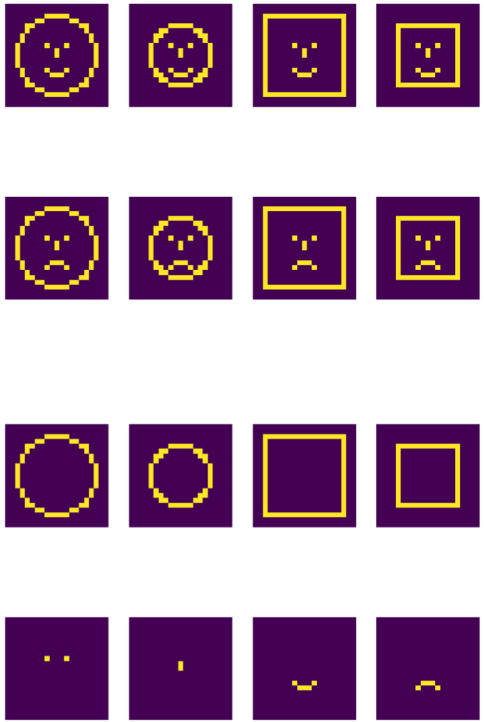
- TensorFlow
  - Google
  - Seems to have a larger user community
  - Said to be better in building a serious project
- PyTorch
  - Facebook and the other supporting organizations
  - Rapidly evolving
  - Easier to quickly see the proof of concept

I do have my own preference, but for this course choose whatever you prefer

The First Layer

- Unsupervised feature learning

- Before deep learning, a common practice was to extract features and then learn a supervised model
  - Source separation
    - Convert the time domain audio signals into matrices by using Short-Time Fourier Transform (STFT) and then learn dictionaries
  - Object recognition
    - Extract a bunch of different features (e.g. HoG, SIFT, etc) and then build a classifier
  - Sentiment analysis
    - Preprocess the text, learn topics, and then build a classifier
  - Speech recognition
    - Extract Mel-Frequency Cepstrum Coefficients (MFCC) and then learn Hidden Markov Models
- YALT
  - In our baby facial expression recognition problem we manually extracted the features first
  - And then built a softmax classifier



The First Layer

- Unsupervised feature learning

- Today we will learn how to systematically learn those features from raw data
  - We call this procedure **unsupervised feature learning**
  - Don't worry, we're not getting away from neural networks
- Why unsupervised?
  - Supervision here means human intervention to solve the problem
    - e.g. When you train a classifier, you need a bunch of pairs: a data sample (facial image) and its label (happy or sad)
    - You let the model know how you think of the problem
    - Mathematically, this can be done by either learning the mapping function between the data sample and its label

      $$\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}) \qquad \hat{\boldsymbol{y}} = \mathcal{G}(\boldsymbol{x}\ ; \mathbb{W})$$

    - Or, finding the parameters that maximize the *a posteriori* probability

      $$P(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\Theta})$$

  - Unsupervised learning
    - You don't have the human intervention, or the labels
    - Then, what are we learning?
    - A model that best describes the data

The First Layer

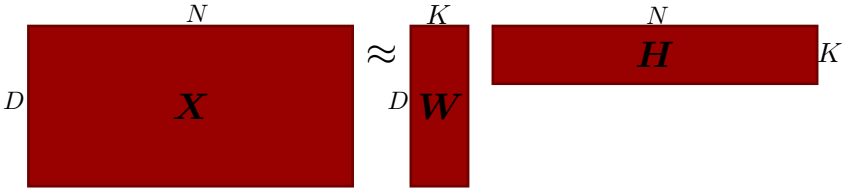
- Unsupervised feature learning

- Modeling the data
  - If we use a probabilistic distribution it's to find out the maximum likelihood solution

    $P(\boldsymbol{x}; \boldsymbol{\Theta})$
  - Often, this can be viewed as a latent variable analysis with K latent variables, too

The data matrix → $\boldsymbol{X} \approx \boldsymbol{W}\boldsymbol{H}$ ← The activation of the latent variable

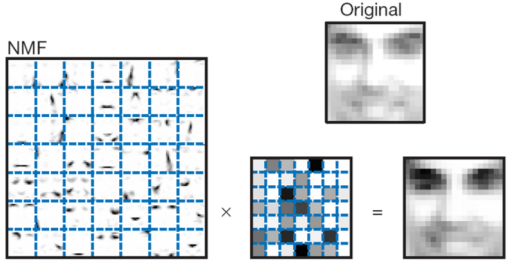
The representative vectors of the latent variable



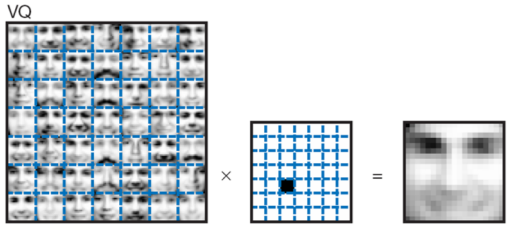
- For example
  - If you find the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\top$, that will correspond to $\boldsymbol{W}$
    - $\boldsymbol{W}^\top\boldsymbol{W} = \boldsymbol{I}$ and, therefore, $\boldsymbol{W}^\top\boldsymbol{X} = \boldsymbol{H}$. Also, rows of $\boldsymbol{H}$ are ordered in their variances
    - Principal Component Analysis
    - $\boldsymbol{H}$ is a set of lower dimensional features that can still describe the original distribution
  - If $\boldsymbol{X}_{:,n} \sim \sum_k \boldsymbol{H}_{k,n}\mathcal{N}(\boldsymbol{W}_{:,k}, \boldsymbol{\Sigma}_{:,:,k})$ and $\boldsymbol{H}_{:,n}$ is a one hot vector
    - Gaussian Mixture Model (a.k.a. Vector Quantization)
  - If you find $\boldsymbol{W}$ and $\boldsymbol{H}$ that minimize the approximation error, but are nonnegative at the same time
    - Nonnegative Matrix Factorization
    - $\boldsymbol{H}$ gives you the parts-based representation

The First Layer

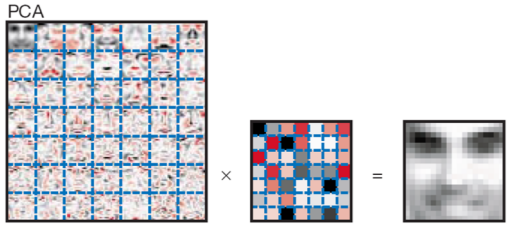
- Unsupervised feature learning



NMF estimates parts-based representations, something like Lego blocks.
Reconstruction is a linear combination of them, but subtraction is not allowed.



VQ finds a bunch of cluster means.
Reconstruction is choosing the most similar mean.



PCA finds the holistic eigenfaces.
From the important one down to the subtle ones.

Lee&Seung Nature 1999, Lee&Seung NIPS 2001

The First Layer

- Unsupervised feature learning

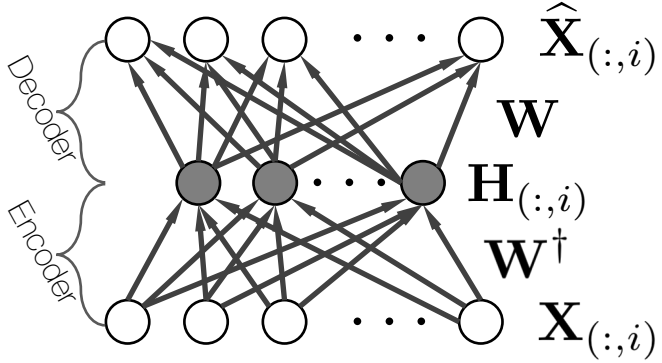
- There are so many different ways to solve the linear approximation $\boldsymbol{X} \approx \boldsymbol{W}\boldsymbol{H}$
- So, eventually it's all about how to constrain the problem

$$\arg\min_{\boldsymbol{W},\boldsymbol{H}} \mathcal{D}(\boldsymbol{X}||\boldsymbol{W}\boldsymbol{H}) + \lambda_{\boldsymbol{W}} f(\boldsymbol{W}) + \lambda_{\boldsymbol{H}} g(\boldsymbol{H})$$

  - For clustering, you want a super sparse column vectors in $\boldsymbol{H}$ so that the data sample is one of the cluster means, deviated accordingly
  - For PCA, you want the after-projection-samples are with maximal variances
  - For NMF, you constrain $\boldsymbol{W}, \boldsymbol{H}$ to be nonnegative (element-wise)
  - …
- It is convenient to assume that there exists some kind of pseudo inverse of $\boldsymbol{W}$
  - For PCA, $\boldsymbol{W}^{\dagger} = \boldsymbol{W}^{\top}$
  - For NMF, $\boldsymbol{W}^{\dagger} = \boldsymbol{W}^{\top}$ depending on the choice of the error function $\boldsymbol{H} \leftarrow \boldsymbol{H} \odot \frac{\boldsymbol{W}^{\top}\boldsymbol{X}}{\boldsymbol{W}^{\top}\boldsymbol{W}\boldsymbol{H}}$
- If you assume this pseudo inverse of the basis vectors, you can think of the projection as a way to convert your data into features $\boldsymbol{W}^{\dagger}\boldsymbol{X} \approx \boldsymbol{W}^{\dagger}\boldsymbol{W}\boldsymbol{H} \approx \boldsymbol{H}$

The First Layer

- Autoencoder: neural network-based unsupervised learning



$$\underset{\boldsymbol{W},\boldsymbol{W}^{\dagger},\boldsymbol{H}}{\arg\min}\ \mathcal{D}(\boldsymbol{X}||\boldsymbol{W}\boldsymbol{W}^{\dagger}\boldsymbol{X}) + \lambda_{\boldsymbol{W}} f(\boldsymbol{W}) + \lambda_{\boldsymbol{W}^{\dagger}} f(\boldsymbol{W}^{\dagger}) + \lambda_{\boldsymbol{H}} g(\boldsymbol{H})$$

- Once the objective function is setup in this way, you can estimate the parameters using SGD
  - TF or PT will take care of this part using automatic gradient calculation

The First Layer

- Autoencoder: neural network-based unsupervised learning

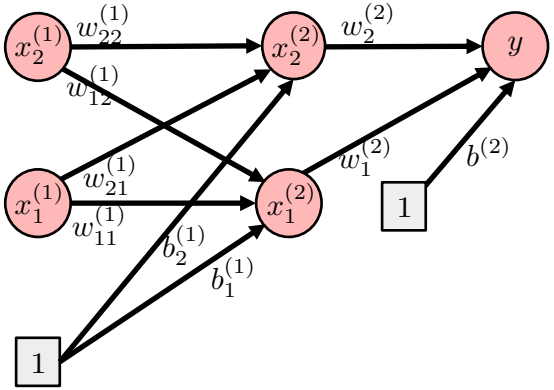
- Demo
  - Sparse coding
  - NMF

The First Layer

- Nonlinearity in neural networks

- So far I haven't said anything about nonlinearity in the lower layers (at least officially)
- Why do we need it?
  - Because stacked linear models form yet another linear model
- Stacked linear models

$$x_1^{(2)} = w_{11}^{(1)} x_1^{(1)} + w_{12}^{(1)} x_2^{(1)} + b_1^{(1)}$$
$$x_2^{(2)} = w_{21}^{(1)} x_1^{(1)} + w_{22}^{(1)} x_2^{(1)} + b_2^{(1)}$$
$$y = w_1^{(2)} x_1^{(2)} + w_2^{(2)} x_2^{(2)} + b^{(2)}$$



- Form another linear model

$$\begin{aligned} y &= w_1^{(2)} w_{11}^{(1)} x_1^{(1)} + w_1^{(2)} w_{12}^{(1)} x_2^{(1)} + w_1^{(2)} b_1^{(1)} \\ &\quad + w_2^{(2)} w_{21}^{(1)} x_1^{(1)} + w_2^{(2)} w_{22}^{(1)} x_2^{(1)} + w_2^{(2)} b_2^{(1)} + b^{(2)} \\ &= \left(w_1^{(2)} w_{11}^{(1)} + w_2^{(2)} w_{21}^{(1)}\right) x_1^{(1)} + \left(w_1^{(2)} w_{12}^{(1)} + w_2^{(2)} w_{22}^{(1)}\right) x_2^{(1)} + w_1^{(2)} b_1^{(1)} + w_2^{(2)} b_2^{(1)} + b^{(2)} \end{aligned}$$

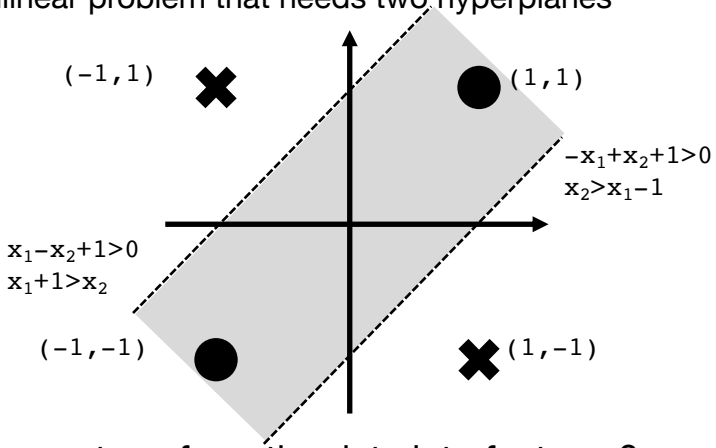
  - Or

$$\begin{aligned} y &= \boldsymbol{w}^{(2)} (\boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)} \\ &= \boldsymbol{w}^{(2)} \boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{w}^{(2)} \boldsymbol{b}^{(1)} + b^{(2)} \end{aligned}$$

The First Layer

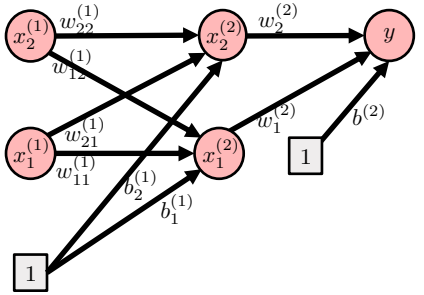
- Nonlinearity in neural networks

- What can we do then?
- The XOR problem
  - A nonlinear problem that needs two hyperplanes

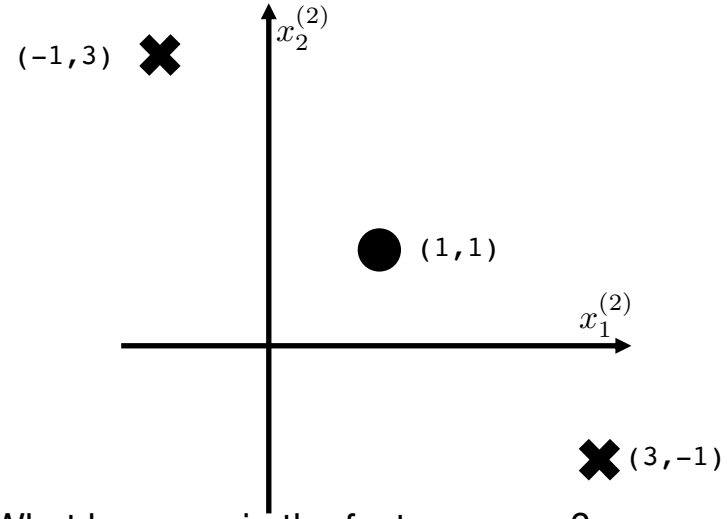


- How do you transform the data into features?
  - It must have something to do with hyperplanes

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} +1 & -1 & 1 \\ -1 & +1 & 1 \end{bmatrix}$$



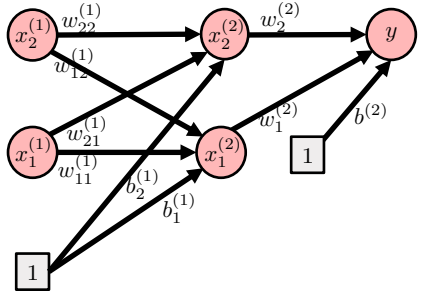
- The new feature space



- What happens in the feature space?
  - If it were not for the help from the nonlinearity
  - Still NOT linearly solvable!

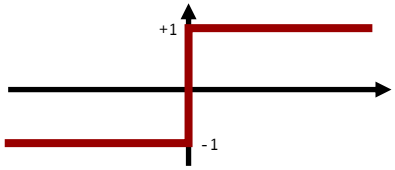
The First Layer

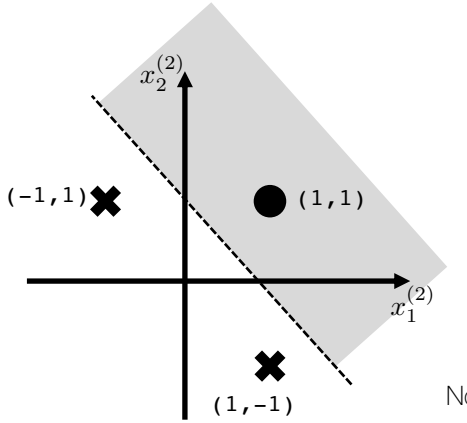
- Nonlinearity in neural networks



- With a sign function

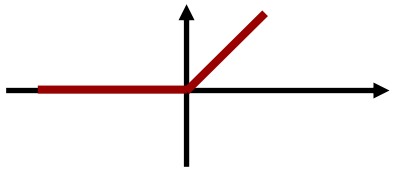
$$x_i^{(2)} = \text{sign}(\boldsymbol{W}_{i,:}^{(1)}\boldsymbol{x} + b_i^{(1)})$$

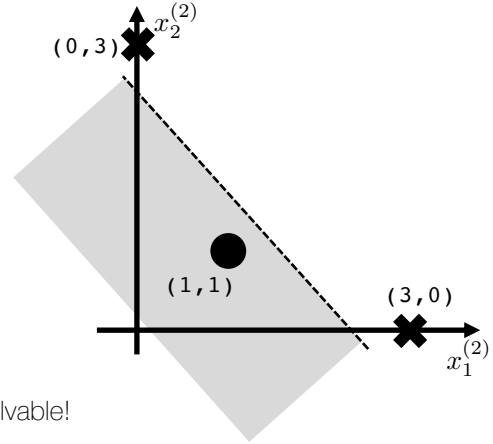


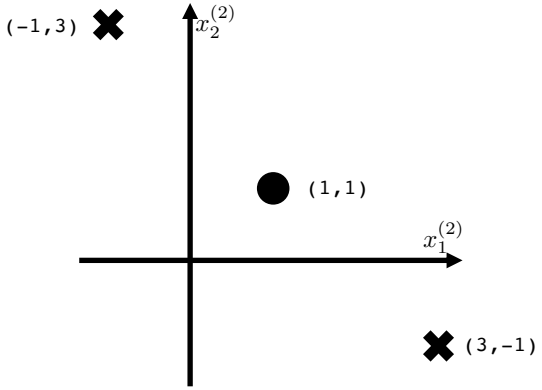


- With a Rectified Linear Unit (ReLU)

$$x_i^{(2)} = \max(\boldsymbol{W}_{i,:}^{(1)}\boldsymbol{x} + b_i^{(1)}, 0)$$





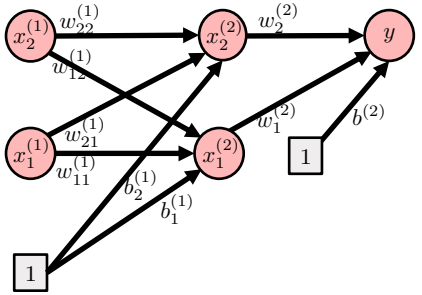


Now they are linearly solvable!

New hyperplane corresponds to the last layer weights

The First Layer

- Nonlinearity in autoencoders



- You can replace the last layer binary output with the input vector
  - Classifier turns into an AE
- Suppose you can estimate these weights of the XOR network by using BP
- Can you also estimate these weights with the AE setup?
- You just saw that nonlinearity can help produce better features
- AE might benefit from it to

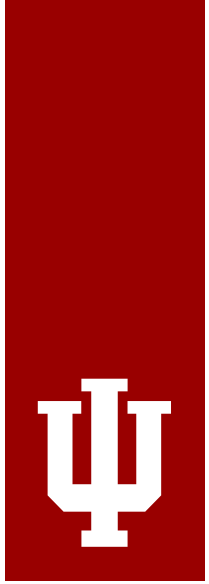
$$\underset{\boldsymbol{W},\boldsymbol{W}^\dagger,\boldsymbol{H}}{\arg\min}\ \mathcal{D}(\boldsymbol{X}||\boldsymbol{W}\sigma(\boldsymbol{W}^\dagger\boldsymbol{X})) + \lambda_{\boldsymbol{W}} f(\boldsymbol{W}) + \lambda_{\boldsymbol{W}^\dagger} f(\boldsymbol{W}^\dagger) + \lambda_{\boldsymbol{H}} g(\boldsymbol{H})$$

- But it's not clear if the encoder weights are going to be actually create useful features for the following classification
- We will cover this part in the next time

The First Layer

- The pipeline for supervised learning

- Feature engineering
  - Tries to come up with a set of features that best describe the data and the problem
  - There can be many different ways (as you've seen)
  - You never know the quality until you actually test out those features for your problem
- Supervised learning (classification/regression)
  - Tries to come up with the best model to best predict the output variable
  - Even if you use a very nice set of features, nonlinearity can be still involved (e.g. kernel methods for SVM)
  - If you don't like the performance, you never know what to blame
    - It could be because of either bad feature engineering or classifier
- A holistic approach
  - In a multilayer perceptron, you have the first layer dedicated to the feature extraction
  - And the last layer as your supervised learning part
  - What if you just optimize the weights of both layers all together?
    - Then, you can skip the feature engineering part
  - Does this work? When does this approach not work? Why?



Thank You!