CS/DS 541: Class 4

Jacob Whitehill

Whitening transformations

- Whitening transformations are a technique from "classical" ML rather than DL.
  - Time cost is $O(m^3)$, which for high-dimensional feature spaces is too large.
- However, whitening has inspired modern DL techniques such as **batch normalization** (Szegedy & Ioffe, 2015) (more to come later).

Whitening transformations

- Demos (gradient_descent, zscore).

Second-order methods for optimization

Second-order methods for optimization

- An alternative to changing the input features is to use an optimization procedure that considers the 2nd- (or even higher) order terms of the loss function.
- From the classical optimization literature, one of the most common method is Newton-Raphson (aka Newton's method).

Newton’s method

- When applicable, it offers faster convergence guarantees (quadratic rather than linear convergence).
- Newton’s method is an iterative method for finding the **roots** of a real-valued function *f*, i.e., **w** such that *f*(**w**)=0.
  - This is useful because we can use it to maximize/ minimize a function by finding the roots of the gradient.

Newton's method

- Let the 2nd-order Taylor expansion of *f* around **w**$^{(k)}$ be:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

  where **H** is the Hessian of *f* evaluated at **w**$^{(k)}$.

Newton’s method

- To minimize *f*, we find the root of the gradient of *f*’s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w} + {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w}^{(k)}\right)$$

Newton's method

- To minimize *f*, we find the root of the gradient of *f*'s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w} + {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w}^{(k)}\right)$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)}$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$0 = \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$\mathbf{H}\mathbf{w} = \mathbf{H}\mathbf{w}^{(k)} - \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

Newton's method

- To minimize *f*, we find the root of the gradient of *f*'s Taylor expansion:

$$
\begin{aligned}
f(\mathbf{w}) &\approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)}) \\
\nabla_{\mathbf{w}} f(\mathbf{w}) &\approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}}\left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w} + {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w}^{(k)}\right) \\
&\quad \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} \\
&\quad \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)} \\
0 &= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)} \\
\mathbf{H}\mathbf{w} &= \mathbf{H}\mathbf{w}^{(k)} - \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) \\
\mathbf{w}^{(k+1)} &= \mathbf{w}^{(k)} - \mathbf{H}^{-1}\nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})
\end{aligned}
$$

Newton’s method

- Note that, compared to gradient descent, the update rule in Newton’s method replaces the step size ε with the Hessian evaluated at $\mathbf{w}^{(k)}$:

  - Gradient descent:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

  - Newton’s method:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \mathbf{H}^{-1} \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

Newton's method

- Newton's method requires computation of **H**.
    - For high-dimensional feature spaces, **H** is huge, i.e., $O(m^3)$.
- Hence, Newton's method in its pure form is impractical for DL.
- However, it has inspired modern DL optimization methods such as the **Adam** optimizer (Kingma & Ba 2014) (more to come later).

Logistic regression

Regression vs. classification

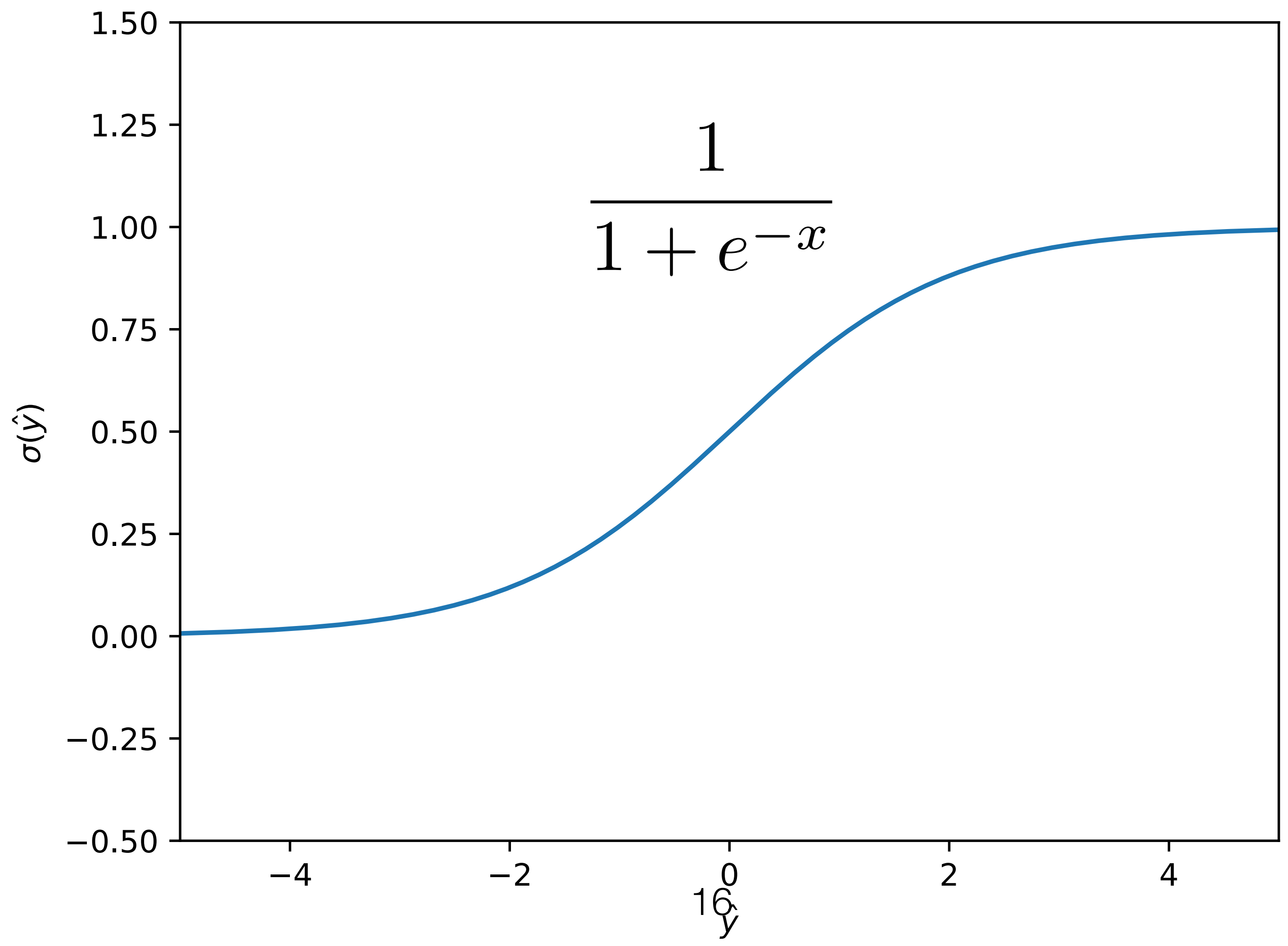
- Recall the two main supervised learning cases.
    - **Regression**: predict any real number.
    - **Classification**: choose from a finite set (e.g., {0, 1}).
- So far, we have talked only about the first case.

Binary classification

- The simplest classification problem consists of just 2 classes (binary classification), i.e., $y \epsilon \{ 0, 1 \}$.
- One of the simplest and most common classification techniques is **logistic regression**.
- Logistic regression is similar to linear regression but also uses a sigmoidal "squashing" function to ensure that $\hat{y} \epsilon (0, 1)$.

Sigmoid: a "squashing" function

- A sigmoid function is an "s"-shaped, monotonically increasing and bounded function.
- Here is the **logistic sigmoid** function σ:



Logistic sigmoid

- The logistic sigmoid function σ has some nice properties:
  - σ(-*z*) = 1 - σ(*z*)

$$\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}} \\
1-\sigma(z) &= 1-\frac{1}{1+e^{-z}} \\
&= \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} \\
&= \frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1/e^{-z}+1} \\
&= \frac{1}{1+e^{z}} \\
&= \sigma(-z)
\end{aligned}$$

Logistic sigmoid

- The logistic sigmoid function σ has some nice properties:
    - σ'(*z*) = σ(*z*)(1 - σ(*z*))

$$\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}} \\
\frac{\partial\sigma}{\partial z} = \sigma'(z) &= -\frac{1}{(1+e^{-z})^2}(e^{-z}\times(-1)) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{e^{-z}}{1+e^{-z}} \times \frac{1}{1+e^{-z}} \\
&= \frac{1}{1/e^{-z}+1} \times \frac{1}{1+e^{-z}} \\
&= \frac{1}{1+e^{z}} \times \frac{1}{1+e^{-z}} \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}$$

Logistic regression

- With logistic regression, our predictions are defined as:

$$\hat{y} = \sigma\left(\mathbf{x}^\top \mathbf{w}\right)$$

- Hence, they are forced to be in (0,1).
- For classification, we can interpret the real-valued outputs as probabilities that express how confident we are in a prediction, e.g.:
    - $\hat{y}$=0.95: very confident that a face contains a smile.
    - $\hat{y}$=0.58: not very confident that a face contains a smile.

Logistic regression

- How to train? Unlike linear regression, logistic regression has no analytical (closed-form) solution.
  - We can use (stochastic) gradient descent instead.
  - We have to apply the **chain-rule of differentiation** to handle the sigmoid function.

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.
- For simplicity, we'll consider just a single example:

$$\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&=
\end{aligned}$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}}\left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \quad \left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)
\end{aligned}$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.
- For simplicity, we'll consider just a single example:

$$\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right) \sigma(\mathbf{x}^\top \mathbf{w}) \left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right)
\end{aligned}$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.
- For simplicity, we'll consider just a single example:

$$\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y}-y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top\mathbf{w})-y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}}\left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top\mathbf{w})-y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top\mathbf{w})-y\right)\sigma(\mathbf{x}^\top\mathbf{w})\left(1-\sigma(\mathbf{x}^\top\mathbf{w})\right)
\end{aligned}$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\mathrm{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y}-y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top\mathbf{w})-y\right)^2 \\
\nabla_{\mathbf{w}} f_{\mathrm{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}}\left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top\mathbf{w})-y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top\mathbf{w})-y\right)\sigma(\mathbf{x}^\top\mathbf{w})\left(1-\sigma(\mathbf{x}^\top\mathbf{w})\right) \\
&= \mathbf{x}\,(\hat{y}-y)\,\hat{y}\,(1-\hat{y})
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right) \sigma(\mathbf{x}^\top \mathbf{w}) \left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right) \\
&= \mathbf{x}(\hat{y} - y)\, \hat{y}(1 - \hat{y})
\end{aligned}$$

Notice the extra multiplicative terms compared to the gradient for *linear* regression: x($\hat{y}$ - y)

Attenuated gradient

- What if the weights **w** are initially chosen badly, so that *ŷ* is very close to 1, even though *y* = 0 (or vice-versa)?
    - Then *ŷ*(1 - *ŷ*) is close to 0.
- In this case, the gradient:

$$\nabla_{\mathbf{w}} f_{\mathrm{MSE}}(\mathbf{w}) = \mathbf{x}\left(\hat{y} - y\right)\hat{y}\left(1 - \hat{y}\right)$$

will be very close to 0.

- If the gradient is 0, then no learning will occur!

Different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.
- One particularly well-suited cost function uses logarithms.
- Logarithms and the logistic sigmoid interact well:

$$\frac{\partial}{\partial \mathbf{w}} \left[\log \sigma(\mathbf{x}^\top \mathbf{w}) \right] =$$

Different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.

- One particularly well-suited cost function uses logarithms.

- Logarithms and the logistic sigmoid interact well:

$$\frac{\partial}{\partial \mathbf{w}}\left[\log \sigma(\mathbf{x}^\top \mathbf{w})\right] = \mathbf{x}\frac{1}{\sigma(\mathbf{x}^\top \mathbf{w})}\sigma(\mathbf{x}^\top \mathbf{w})\left(1-\sigma(\mathbf{x}^\top \mathbf{w})\right)$$

Different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.

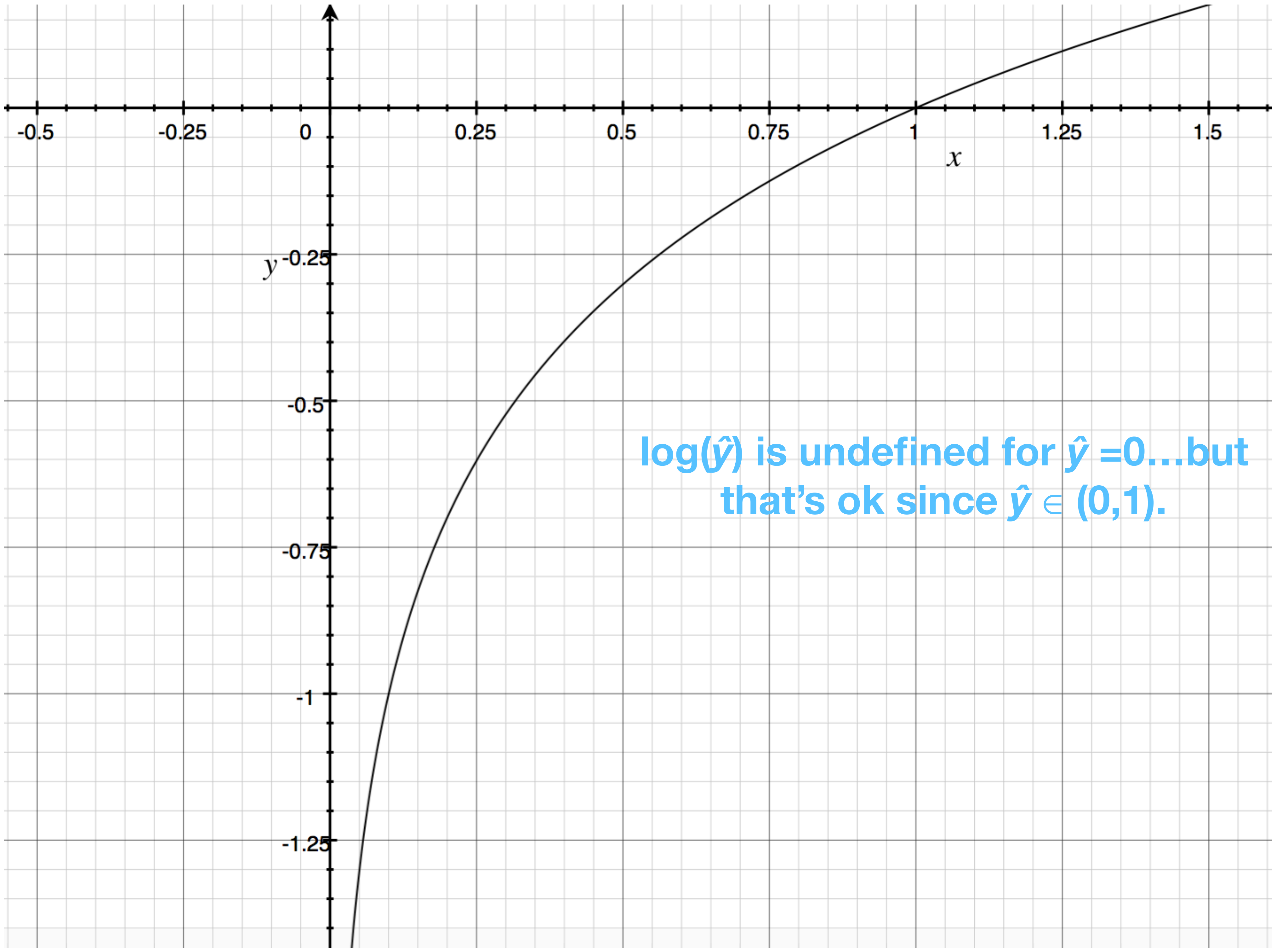
- One particularly well-suited cost function uses logarithms.

- Logarithms and the logistic sigmoid interact well:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}}\left[\log \sigma(\mathbf{x}^\top \mathbf{w})\right] &= \mathbf{x}\frac{1}{\sigma(\mathbf{x}^\top \mathbf{w})}\sigma(\mathbf{x}^\top \mathbf{w})\left(1-\sigma(\mathbf{x}^\top \mathbf{w})\right)\\ &= \mathbf{x}\left(1-\sigma(\mathbf{x}^\top \mathbf{w})\right)\end{aligned}$$

The gradient of log(σ) is surprisingly simple.

Logarithm function



log($\hat{y}$) is undefined for $\hat{y}$ =0…but that's ok since $\hat{y} \in (0,1)$.

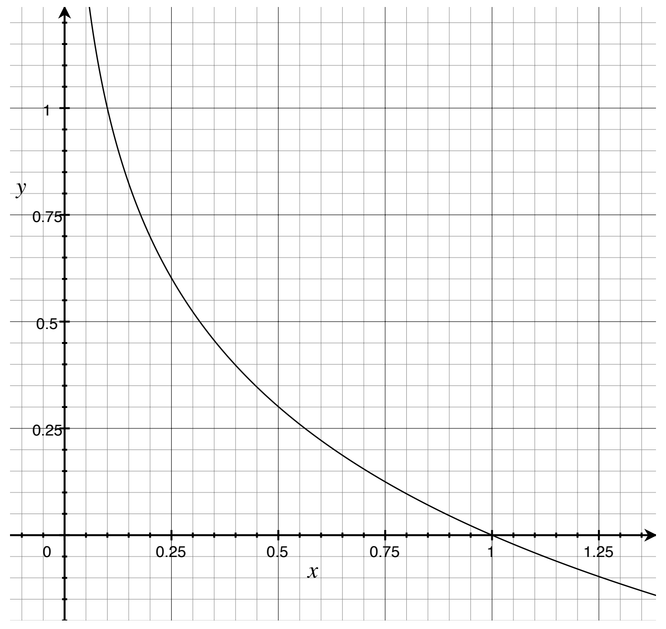
Log loss

- We want to assign a large loss when *y*=1 but *ŷ*=0

- We typically use the **log-loss** for logistic regression:

$$-y \log \hat{y}$$

The *y* or (1-*y*) "selects" which term in the expression is active, based on the ground-truth label.



-log(*ŷ*)

Jacob Whitehill, WPI

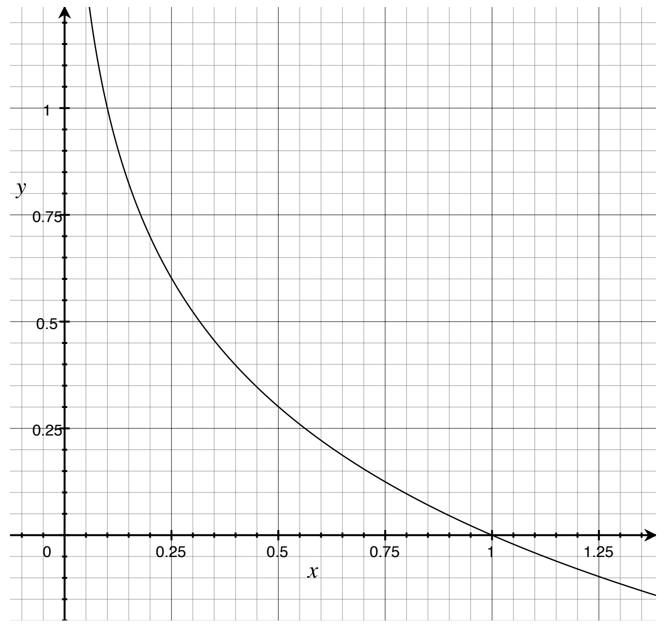
Log loss

- We want to assign a large loss when y=1 but $\hat{y}$=0, and for y=0 but $\hat{y}$=1.

- We typically use the **log-loss** for logistic regression:

$$-y \log \hat{y} - (1-y) \log(1-\hat{y})$$

The *y* or (1-*y*) "selects" which term in the expression is active, based on the ground-truth label.



-log($\hat{y}$)

Gradient descent for logistic regression with log-loss

$$\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) \quad = \quad \nabla_{\mathbf{w}} \left[- \left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right]$$

Gradient descent for logistic regression with log-loss

$$\begin{aligned}\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[- \left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right] \\ &= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1 - \sigma(\mathbf{x}^\top \mathbf{w})) \right)\end{aligned}$$

Gradient descent for logistic regression with log-loss

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[- \left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right] \\
&= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1-\sigma(\mathbf{x}^\top \mathbf{w})) \right) \\
&= -\left(y \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{\sigma(\mathbf{x}^\top \mathbf{w})} - (1-y) \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{1-\sigma(\mathbf{x}^\top \mathbf{w})} \right)
\end{aligned}
$$

Gradient descent for logistic regression with log-loss

$$\begin{aligned}
\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[-\left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right] \\
&= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1-\sigma(\mathbf{x}^\top \mathbf{w})) \right) \\
&= -\left(y \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{\sigma(\mathbf{x}^\top \mathbf{w})} - (1-y) \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{1-\sigma(\mathbf{x}^\top \mathbf{w})} \right) \\
&= -\left(y\mathbf{x}(1-\sigma(\mathbf{x}^\top \mathbf{w})) - (1-y)\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= -\mathbf{x} \left(y - y\sigma(\mathbf{x}^\top \mathbf{w}) - \sigma(\mathbf{x}^\top \mathbf{w}) + y\sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= -\mathbf{x} \left(y - \sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= \mathbf{x}(\hat{y} - y)
\end{aligned}$$

Same as for linear regression!

Linear regression versus logistic regression

	Linear regression	**Logistic regression**
Primary use	Regression	Classification
Prediction ($\hat{y}$)	$\hat{y} = \mathbf{x}^\top \mathbf{w}$	$\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w})$
Cost/Loss	f_{MSE}	$f_{\log}$
Gradient	$\mathbf{x}(\hat{y} - y)$	$\mathbf{x}(\hat{y} - y)$

- Logistic regression is used primarily for *classification* even though it's called "regression".
- Logistic regression is an instance of a **generalized linear model** — a linear model combined with a **link function** (e.g., logistic sigmoid).
  - In DL, link functions are typically called **activation functions.**

Softmax regression (aka multinomial logistic regression)

Multi-class classification

- So far we have talked about classifying only 2 classes (e.g., smile versus non-smile).
  - This is sometimes called **binary classification.**
- But there are many settings in which multiple (>2) classes exist, e.g., emotion recognition, hand-written digit recognition:

6 classes (fear, anger, sadness, happiness, disgust, surprise)

10 classes (0-9)

Classification versus regression

- Note that, even though the hand-written digit recognition (“MNIST”) problem has classes called “0” , “1”, …, “9”, there is no sense of “distance” between the classes.
  - Misclassifying a 1 as a 2 is just as “bad” as misclassifying a 1 as a 9.

Multi-class classification

- It turns out that logistic regression can easily be extended to support an arbitrary number ($\geq$2) of classes.
  - The multi-class case is called **softmax regression** or sometimes **multinomial logistic regression.**
- How to represent the ground-truth y and prediction $\hat{y}$?
  - Instead of just a scalar y, we will use a vector $\mathbf{y}$.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.
- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \left[\begin{array}{c} 1 \\ 0 \end{array}\right]$$

$$\mathbf{y}^{(2)} = \left[\begin{array}{c} 0 \\ 1 \end{array}\right]$$

$$\mathbf{y}^{(3)} = \left[\begin{array}{c} 1 \\ 0 \end{array}\right]$$

- Exactly 1 coordinate of each **y** is 1; the others are 0.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} \boxed{1} \\ 0 \end{bmatrix} \quad \mathbf{y}^{(2)} = \begin{bmatrix} \boxed{0} \\ 1 \end{bmatrix} \quad \mathbf{y}^{(3)} = \begin{bmatrix} \boxed{1} \\ 0 \end{bmatrix}$$

This "slot" is for class 0.

- This is called a **one-hot encoding** of the class label.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \left[\begin{array}{c} 1 \\ 0 \end{array} \right]$$

$$\mathbf{y}^{(2)} = \left[\begin{array}{c} 0 \\ 1 \end{array} \right]$$

$$\mathbf{y}^{(3)} = \left[\begin{array}{c} 1 \\ 0 \end{array} \right]$$

This "slot" is for class 1.

- This is called a **one-hot encoding** of the class label.

Example: 2 classes

- The machine's predictions **ŷ** about each example's label are also **probabilistic.**

- They could consist of:

$$\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.93 \\ 0.07 \end{bmatrix} \longleftarrow \text{Machine's "belief" that the label is 0.}$$

$$\hat{\mathbf{y}}^{(2)} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}$$

$$\hat{\mathbf{y}}^{(3)} = \begin{bmatrix} 0.99 \\ 0.01 \end{bmatrix}$$

- Each coordinate of ***ŷ*** is a probability.

Example: 2 classes

- The machine's predictions **ŷ** about each example's label are also **probabilistic**.
- They could consist of:

$$\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.93 \\ 0.07 \end{bmatrix} \longleftarrow \text{Machine's “belief” that the label is 1.}$$

$$\hat{\mathbf{y}}^{(2)} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}$$

$$\hat{\mathbf{y}}^{(3)} = \begin{bmatrix} 0.99 \\ 0.01 \end{bmatrix}$$

- The sum of the coordinates in each **ŷ** is 1.

Softmax activation function

- Logistic regression outputs a *scalar* label $\hat{y}$ representing the probability that the label is 1.
    - We needed just a single weight vector $\mathbf{w}$, so that $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w})$.

Softmax activation function

- Logistic regression outputs a *scalar* label $\hat{y}$ representing the probability that the label is 1.
  - We needed just a single weight vector $\mathbf{w}$, so that $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w})$.
- Softmax regression outputs a *c-vector* representing the probabilities that the label is k=1, …, c.
  - We need c different vectors of weights $\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}$.
  - Weight vector $\mathbf{w}^{(k)}$ computes how much input $\mathbf{x}$ "agrees" with class k.

Softmax activation function

- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

$$z_2 = \mathbf{x}^\top \mathbf{w}^{(2)}$$

$$\vdots$$

$$z_c = \mathbf{x}^\top \mathbf{w}^{(c)}$$

I will refer to the z's as "pre-activation scores".

Softmax activation function

- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$
$$z_2 = \mathbf{x}^\top \mathbf{w}^{(2)}$$
$$\vdots$$
$$z_c = \mathbf{x}^\top \mathbf{w}^{(c)}$$

- We then **normalize** across all *c* classes so that:
  1. Each output $\hat{y}_k$ is non-negative.
  2. The sum of $\hat{y}_k$ over all *c* classes is 1.

Normalization of the $\hat{y}_k$

1. To enforce non-negativity, we can exponentiate each z_k:

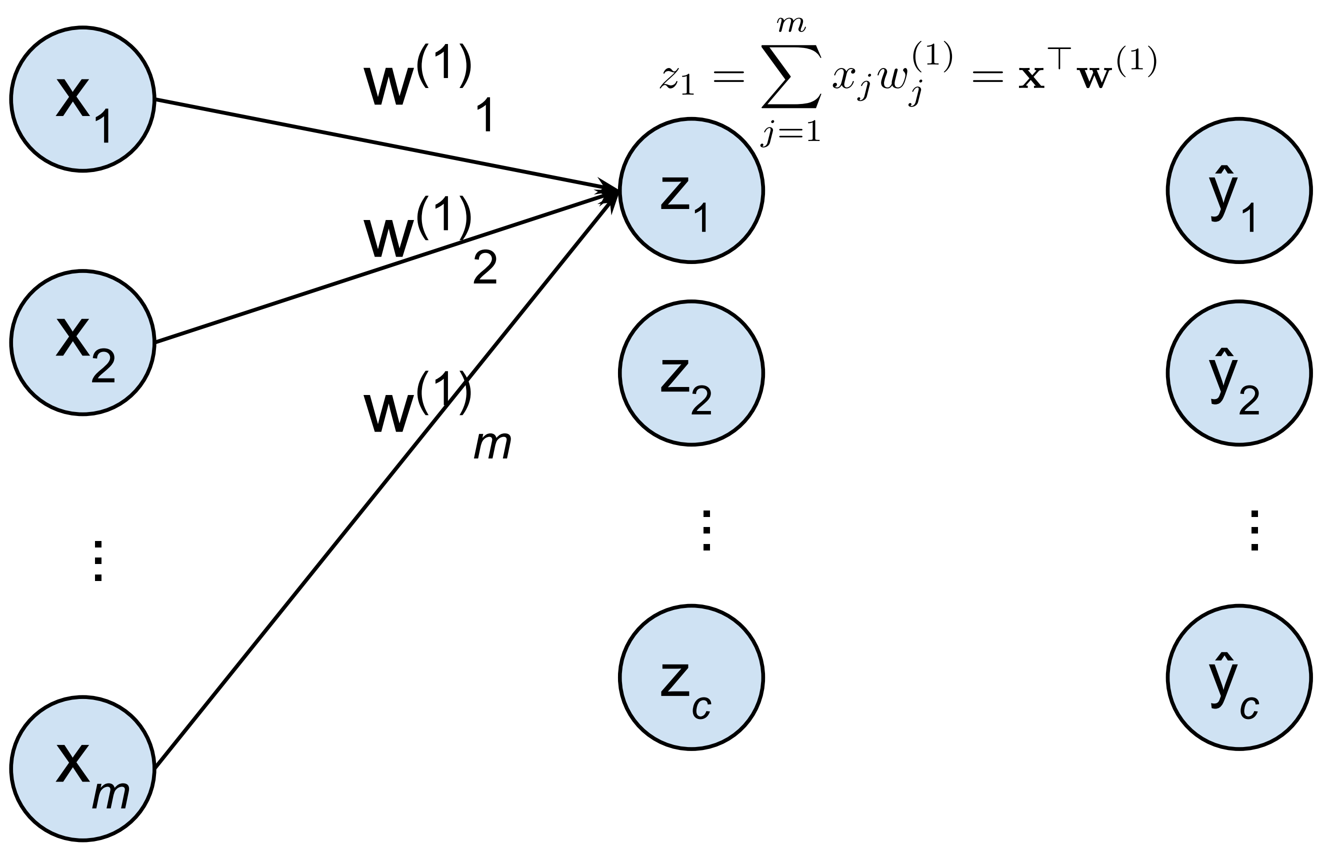
$$\hat{y}_k = \exp(z_k)$$

Normalization of the $\hat{y}_k$

2. To enforce that the $\hat{y}_k$ sum to 1, we can divide each entry by the sum:

$$\hat{y}_k = \frac{\exp(z_k)}{\sum_{k'=1}^{c} \exp(z_{k'})}$$

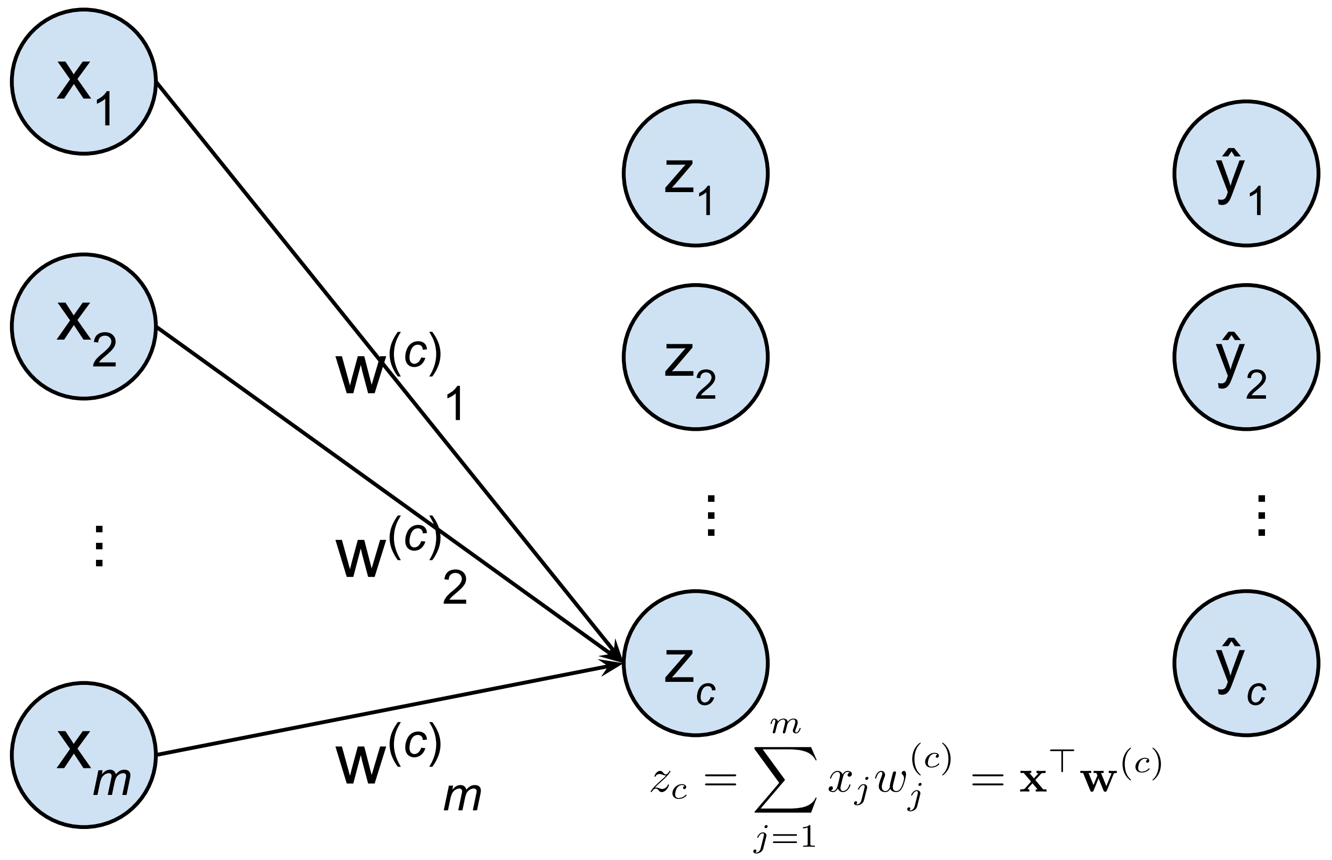
Softmax regression diagram



- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

Softmax regression diagram



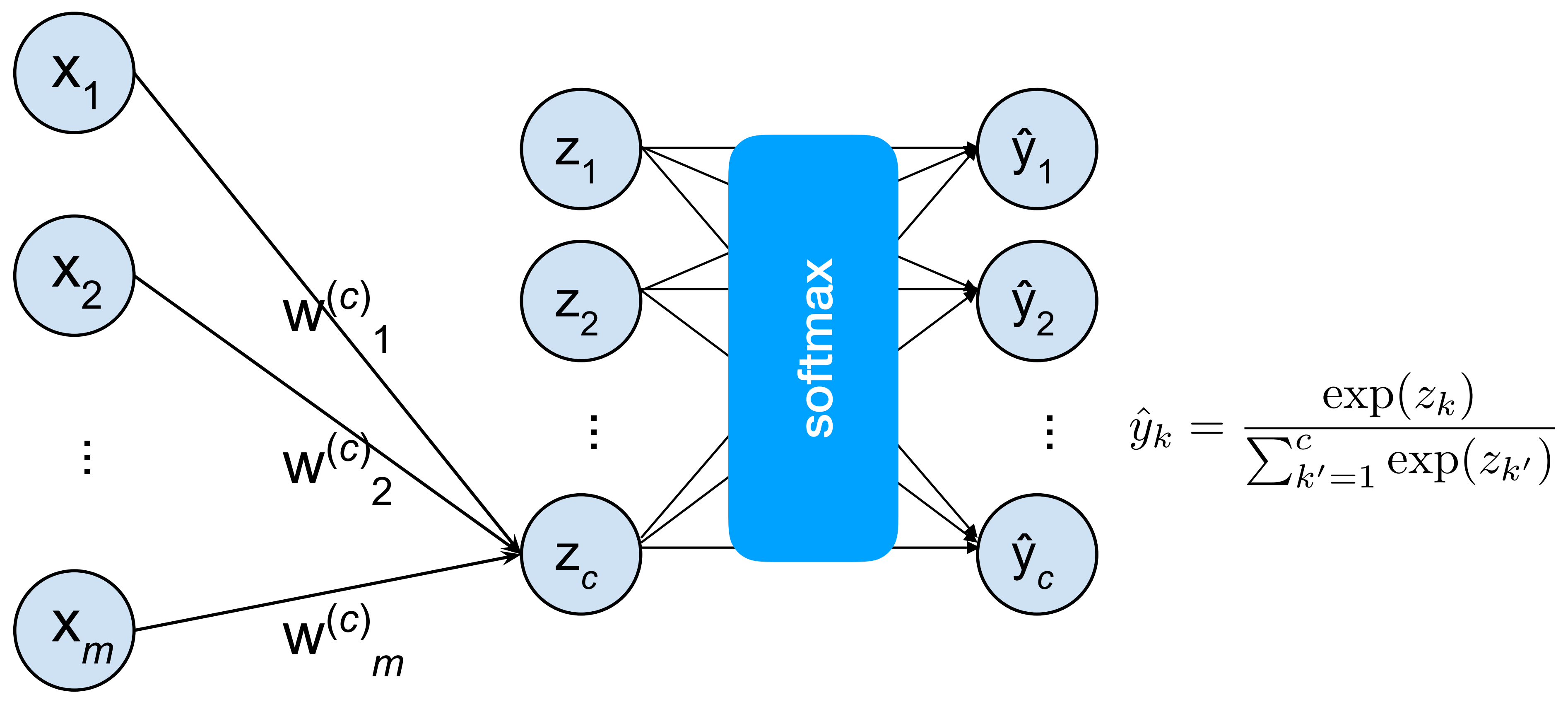
- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

$$\vdots$$

$$z_c = \mathbf{x}^\top \mathbf{w}^{(c)}$$

Softmax regression diagram



- We then **normalize** across all *c* classes.

$$\hat{y}_k = P(y = k \mid \mathbf{x}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) = \frac{\exp(z_k)}{\sum_{k'=1}^c \exp(z_{k'})}$$

Cross-entropy loss

- We need a loss function that can support *c*≥2 classes.
- We will use the **cross-entropy** (CE) loss:

$$f_{\mathrm{CE}} = -\sum_{i=1}^{n}\sum_{k=1}^{c} y_k^{(i)} \log \hat{y}_k^{(i)}$$

- Note that the CE loss subsumes the log-loss for *c*=2.

Cross-entropy loss

- The origin of the cross-entropy function is in coding & information theory.

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\begin{aligned}\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\ &= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})\end{aligned}$$

Conditional independence

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\begin{aligned}\text{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\ &= -\log \prod_{i=1}^{n} \boxed{P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})}\end{aligned}$$

Our NN's estimate of the probability that x belongs to a particular class.

E.g., if $\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$**,** $\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.72 \\ 0.28 \end{bmatrix}$

then this probability is 0.72.

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\text{NLL} = -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})$$

$$= -\log \prod_{i=1}^{n} \boxed{P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})}$$

Our NN's estimate of the probability that x belongs to a particular class.

E.g., if $\mathbf{y}^{(1)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.72 \\ 0.28 \end{bmatrix}$

then this probability is 0.28.

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\begin{aligned}
\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)}\right)^{\left({y_k}^{(i)}\right)}
\end{aligned}$$

For only one value of *k* is the exponent 1. Otherwise it is 0.

Example: $(0.72)^1(0.28)^0$

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$
\begin{aligned}
\text{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)}\right)^{\left(y_k{}^{(i)}\right)}
\end{aligned}
$$

For only one value of *k* is the exponent 1. Otherwise it is 0.

Example: $(0.72)^0(0.28)^1$

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

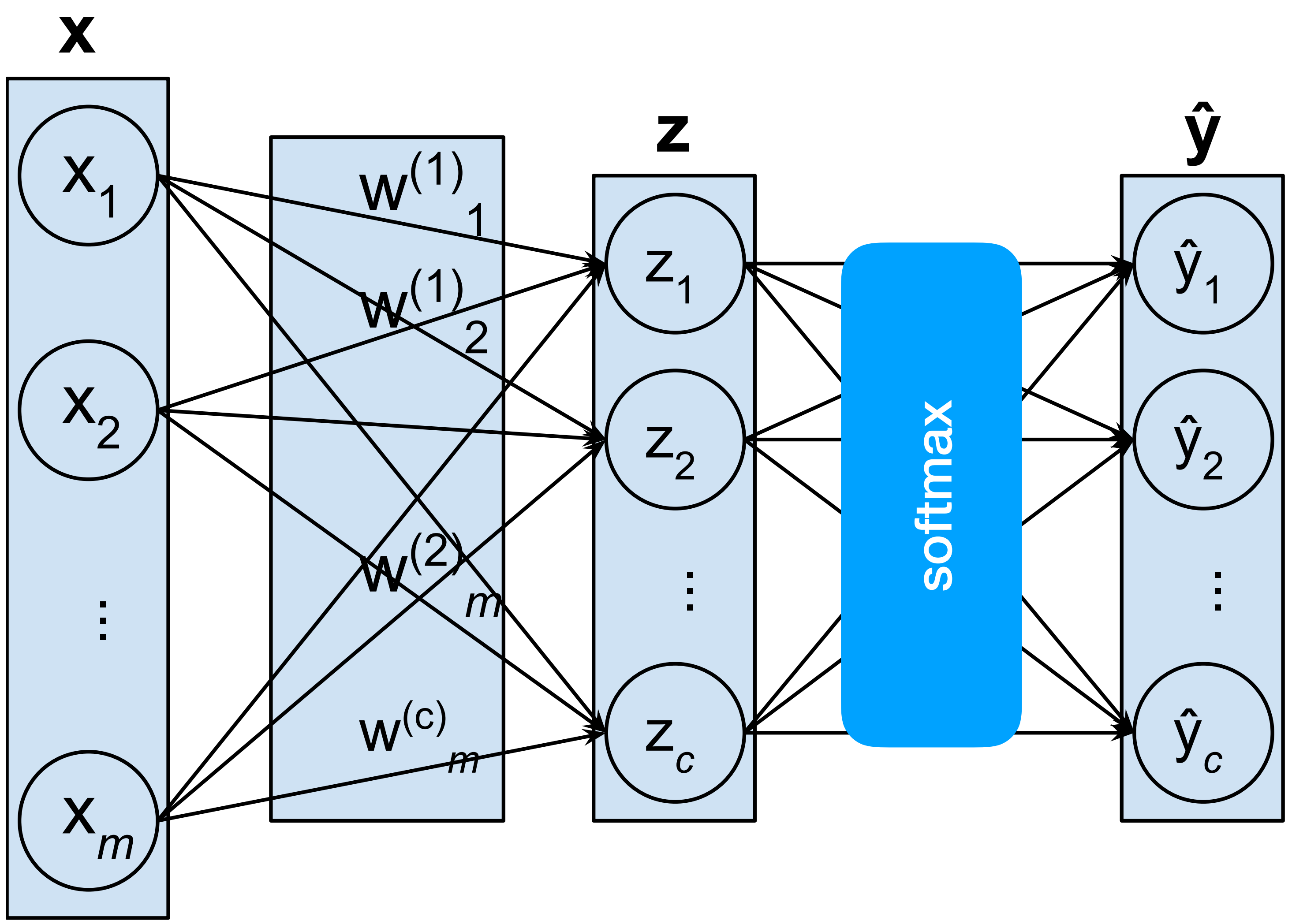
$$\begin{aligned}
\text{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)}\right)^{\left(y_k{}^{(i)}\right)} \\
&= -\sum_{i=1}^{n} \sum_{k=1}^{c} y_k^{(i)} \log \hat{y}_k^{(i)}
\end{aligned}$$

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

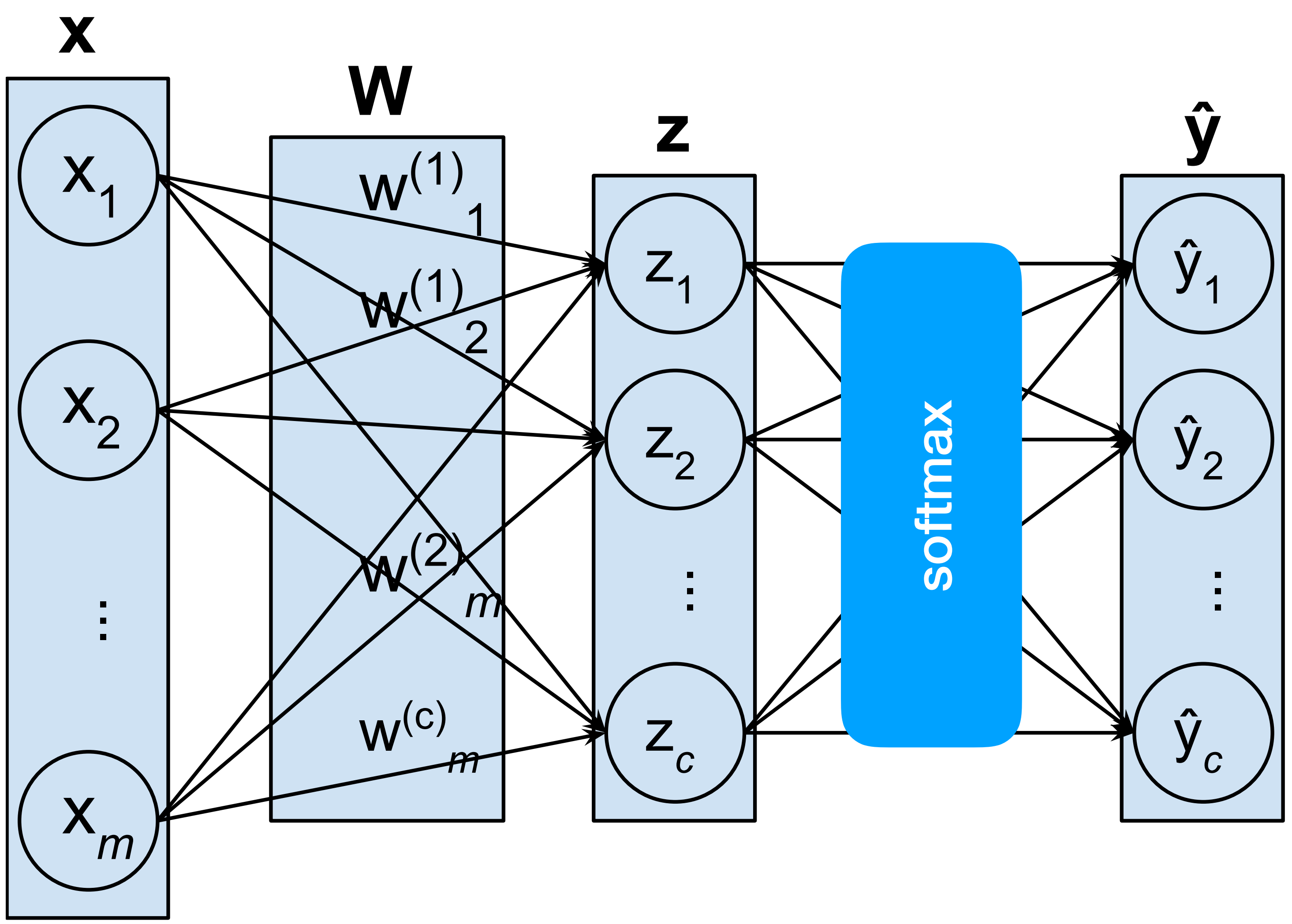
$$
\begin{aligned}
\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)}\right)^{\left(y_k{}^{(i)}\right)} \\
&= -\sum_{i=1}^{n} \sum_{k=1}^{c} y_k^{(i)} \log \hat{y}_k^{(i)} \\
&= f_{\mathrm{CE}}
\end{aligned}
$$

Softmax regression: vectorization



- We can represent each **layer** as a vector (**x**, **z**, **ŷ**).

Softmax regression: vectorization



- We can represent the collection of all c weight vectors $\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}$ as an $(m \times c)$ matrix $\mathbf{W}$.

Softmax regression: vectorization

- By vectorizing, we can compute the pre-activation scores for all *n* examples in one-fell-swoop as:

$$\mathbf{Z} = \mathbf{X}^\top \mathbf{W}$$

Softmax regression: vectorization

- By vectorizing, we can compute the pre-activation scores for all *n* examples in one-fell-swoop as:

$$\mathbf{Z} = \mathbf{X}^\top \mathbf{W}$$

- With numpy, we can call **`np.exp`** to exponentiate every element of **`Z`**.

- We can then use **`np.sum`** and **`/`** (element-wise division) to compute the softmax.

Gradient descent for softmax regression

- With softmax regression, we need to conduct gradient descent on all *c* of the weights vectors.
- As usual, let's just consider the gradient of the cross-entropy loss for a single example **x**.
- We will compute the gradient w.r.t. each weight vector $\mathbf{w}^{(k)}$ separately (where $k = 1, \ldots, c$).

Gradient descent for softmax regression

- Gradient for each weight vector $\mathbf{w}^{(k)}$:

$$\nabla_{\mathbf{w}^{(k)}} f_{\mathrm{CE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{W}) = \mathbf{x}(\hat{y}_k - y_k)$$

- This is the same expression (for each *k*) as for linear regression and logistic regression!

- We can vectorize this to compute all *c* gradients over all *n* examples…

Gradient descent for softmax regression

- Let **Y** and **Ŷ** both be *n* x *c* matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \dots & y_c^{(n)} \end{bmatrix}$$

One-hot encoded vector of class labels for example 1.

Gradient descent for softmax regression

- Let **Y** and **Ŷ** both be *n* x *c* matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \dots & y_c^{(n)} \end{bmatrix}$$

One-hot encoded vector of class labels for example *n*.

Gradient descent for softmax regression

- Let **Y** and **Ŷ** both be *n* x *c* matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \dots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \dots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \dots & \hat{y}_c^{(n)} \end{bmatrix}$$

The machine's estimates of the *c* class probabilities for example *n*.

Gradient descent for softmax regression

- Let **Y** and **Ŷ** both be *n* x *c* matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \dots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \dots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \dots & \hat{y}_c^{(n)} \end{bmatrix}$$

- Then we can compute all c gradient vectors as:

$$\nabla_{\mathbf{W}} f_{\mathrm{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}; \mathbf{W}) = \frac{1}{n}\mathbf{X}(\hat{\mathbf{Y}} - \mathbf{Y})$$

Gradient descent for softmax regression

- Let **Y** and **Ŷ** both be *n* x *c* matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \dots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \dots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \dots & \hat{y}_c^{(n)} \end{bmatrix}$$

- Then we can compute all c gradient vectors as:

$$\nabla_{\mathbf{W}} f_{\mathrm{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}; \mathbf{W}) = \frac{1}{n} \mathbf{X}(\hat{\mathbf{Y}} - \mathbf{Y})$$

How far the guesses are from ground-truth.

Gradient descent for softmax regression

- Let **Y** and **Ŷ** both be *n* x *c* matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \dots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \dots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \dots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \dots & \hat{y}_c^{(n)} \end{bmatrix}$$

- Then we can compute all c gradient vectors as:

$$\nabla_{\mathbf{W}} f_{\text{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}; \mathbf{W}) = \frac{1}{n}\mathbf{X}(\hat{\mathbf{Y}} - \mathbf{Y})$$

The input features (e.g., pixel values).

Bias term

- Like in linear regression, softmax regression also benefits from the use of a bias term.
- Instead of a scalar *b*, we have a bias vector **b** with *c* dimensions (one for each class).
- You will derive the gradient update for **b** as part of homework 3.

Softmax regression demo

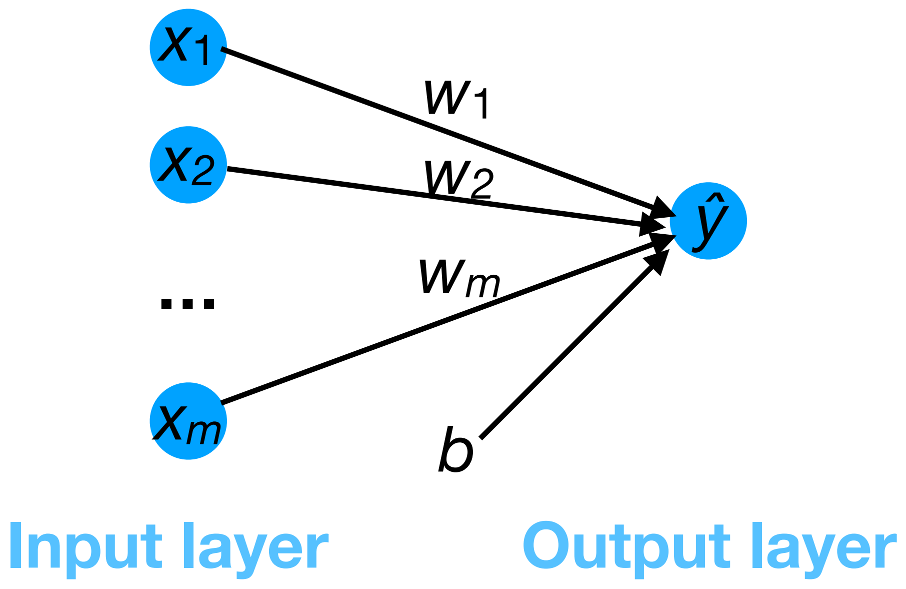
- In HW3, you will apply softmax regression to train a **handwriting recognition system** that can recognize all 10 digits (0-9).

- You will use the popular MNIST dataset consisting of 60K training examples and 10K testing examples:

Review: shallow prediction models

Review

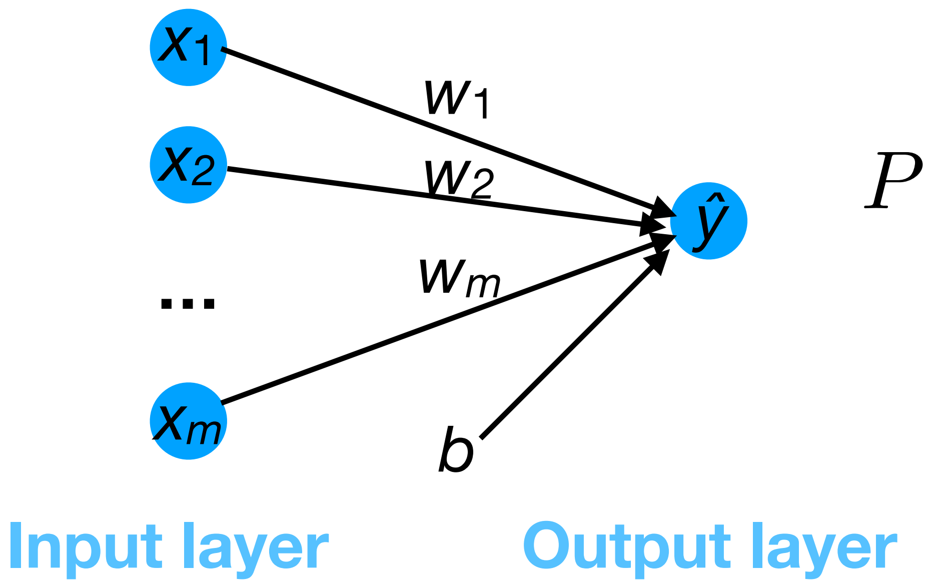
- Linear regression (2-layer NN with linear activation)
  - MSE formulation



$$\hat{y} = \mathbf{x}^\top \mathbf{w} + b$$

Review

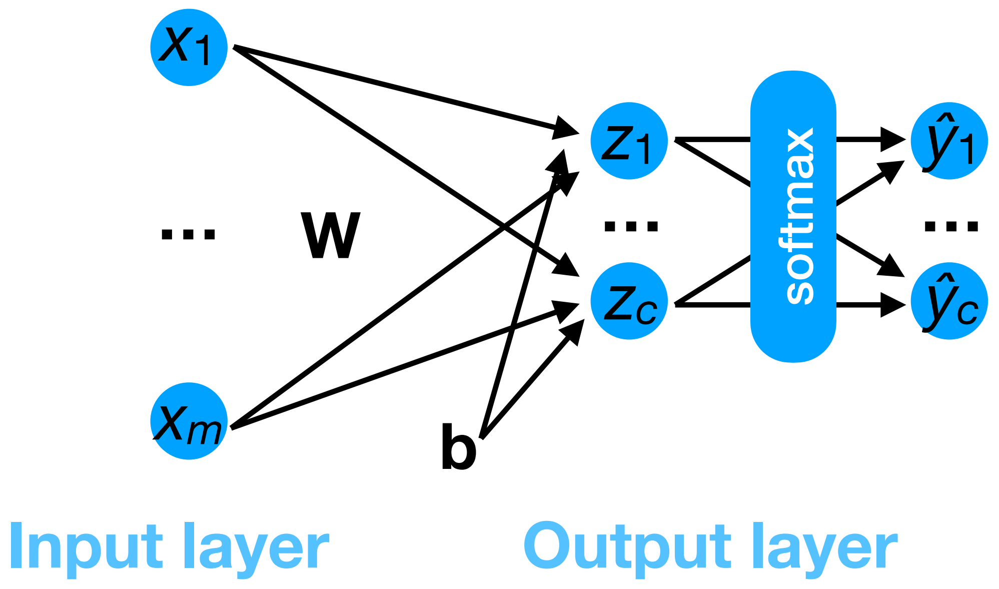
- Linear regression (2-layer NN with linear activation)
  - Probabilistic (MLE) formulation



$$P(y \mid \mathbf{x}, \mathbf{w}, b) = \mathcal{N}(y; \mathbf{x}^\top \mathbf{w} + b, \sigma^2)$$
$$\hat{y} = \mathbb{E}[y \mid \mathbf{x}, \mathbf{w}]$$

Review

- Softmax regression (2-layer NN with softmax activation)
  - Probabilistic (MLE) formulation

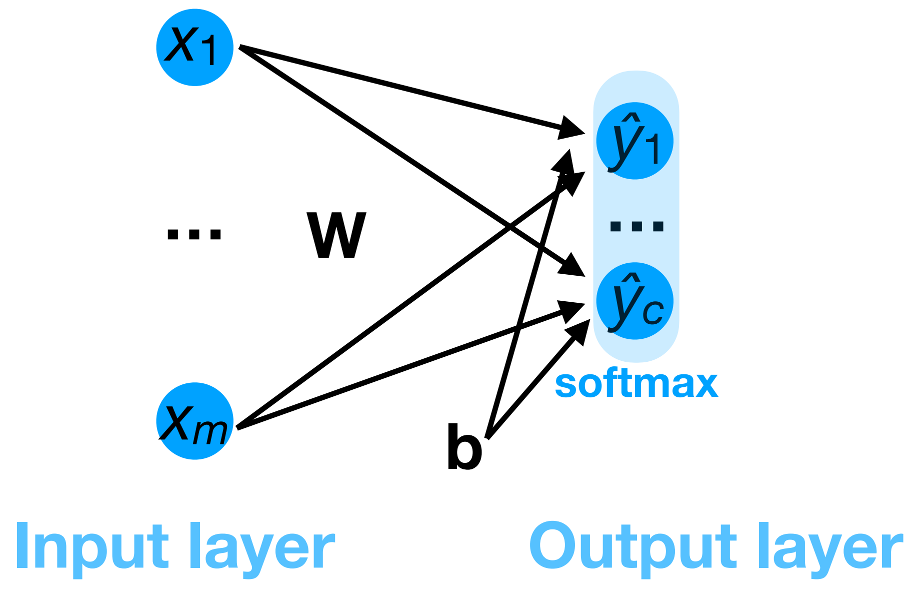


$$\hat{y}_k = P(y_k \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}$$

$$z_k = \mathbf{x}^\top \mathbf{w}^{(k)} + b_k$$

Review

- Softmax regression (2-layer NN with softmax activation)
  - Probabilistic (MLE) formulation



$$\hat{y}_k = P(y_k \mid \mathbf{x}, \mathbf{W}) = \frac{\exp(\mathbf{x}^\top \mathbf{w}^{(k)} + b_k)}{\sum_{k'} \exp(\mathbf{x}^\top \mathbf{w}^{(k')} + b_{k'})}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{x}^\top \mathbf{W} + \mathbf{b}^\top)$$

Shallow models

- Before diving into deeper models, we will examine one more shallow model.
- Instead of predicting a target value *y* from an input vector **x**, we will instead try to **generate** novel input vectors.
- One way to achieve this is using a latent variable model (LVM).

Latent variable models

Latent variable models

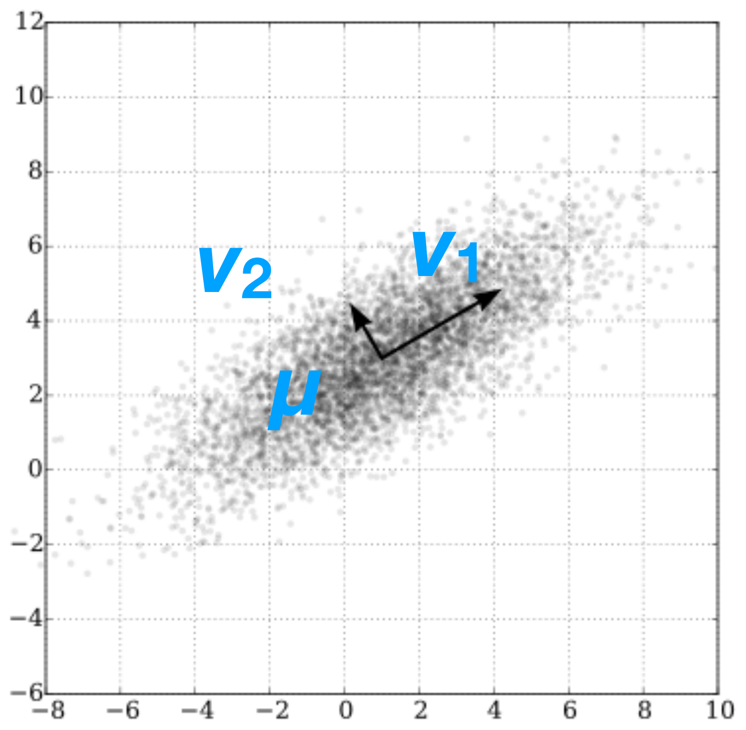
- Latent variable models (LVM) are **unsupervised** ML models.
- They posit that the observed data $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{n}$ can be explained by a set of low-dimensional **latent** (unobserved) variables $\{\mathbf{h}^{(i)}\}_{i=1}^{n}$.
- Examples:
  - MNIST:
  - GENKI4K:

Latent variable models

- Latent variable models (LVM) are **unsupervised** ML models.
- They posit that the observed data $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^n$ can be explained by a set of low-dimensional **latent** (unobserved) variables $\{\mathbf{h}^{(i)}\}_{i=1}^n$.
- Examples:
    - MNIST: each image is “generated” by the digit (0-9) and a thickness (0-5) latent “code”.
    - GENKI4K: each face image is “generated” by the gender (0-1) and smile intensity (0-1).

Latent variable models

- Arguably the simplest LVM is **principal component analysis** (PCA).
- Recall that, for a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{n}$ with mean $\boldsymbol{\mu}$, PCA defines the set of *k* principal directions along which the variance of the projected data is maximized:



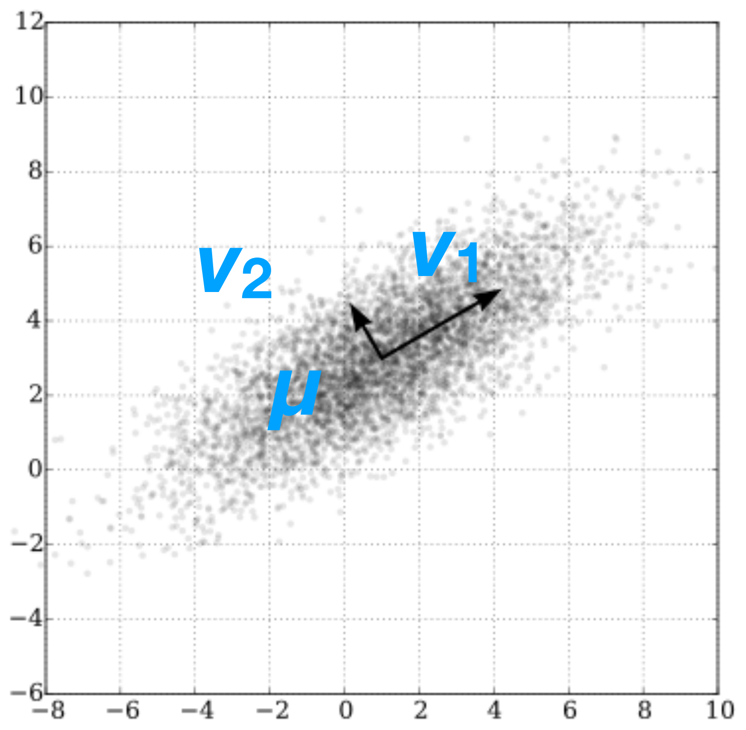
https://en.wikipedia.org/wiki/Principal_component_analysis

Latent variable models

- Arguably the simplest LVM is **principal component analysis** (PCA).

- Recall that, for a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{n}$ with mean $\boldsymbol{\mu}$, PCA defines the set of *k* principal directions along which the variance of the projected data is maximized:

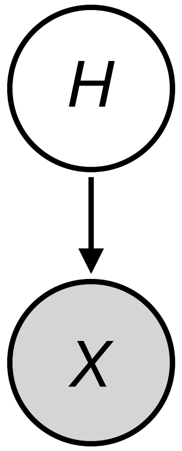
- Vector $\mathbf{v}^{(k)}$ has the same direction as the eigenvector of the auto-covariance matrix of $\mathcal{D}$ with the *k*th largest associated eigenvalue.



https://en.wikipedia.org/wiki/Principal_component_analysis

Principal component analysis (PCA)

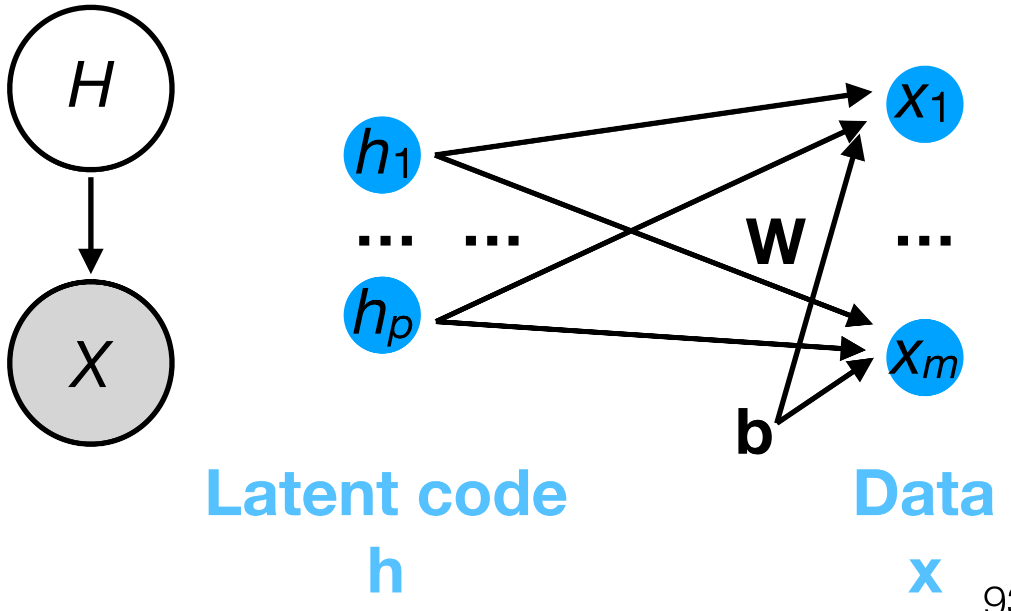
- However, PCA also has a **probabilistic** interpretation as an LVM.
- Let us assume that every data point $\mathbf{x} \in \mathbb{R}^m$ is actually generated from a lower-dimensional latent variable (or **code**) $\mathbf{h} \in \mathbb{R}^p$, where $p < m$.



Principal component analysis (PCA)

- In particular, we assume that each **x** is approximately linear in **h**, i.e.:

$$\mathbf{x} \approx \mathbf{W}\mathbf{h} + \mathbf{b}$$



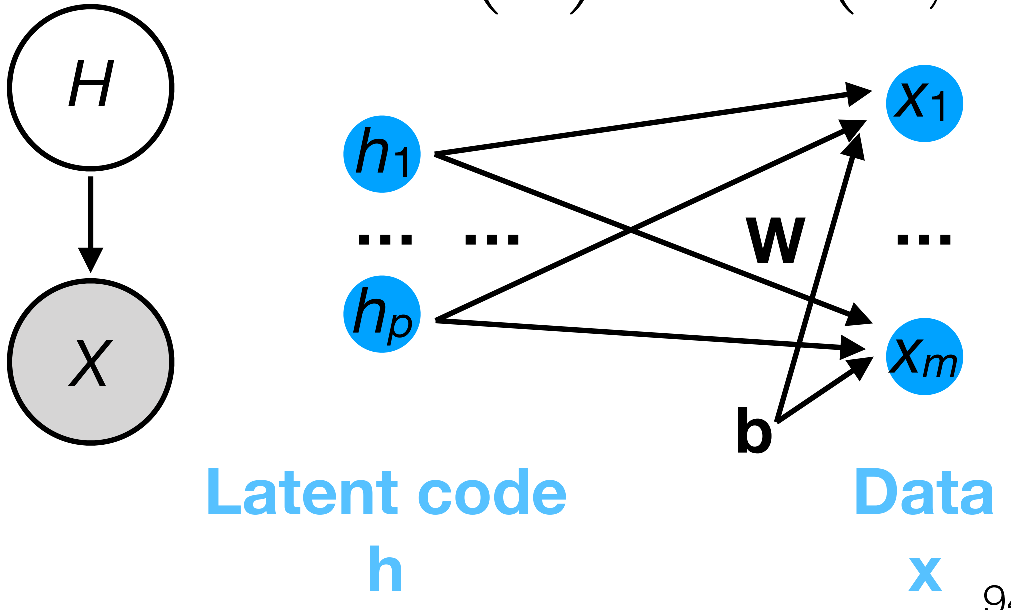
Principal component analysis (PCA)

- In particular, we assume that each **x** is approximately linear in **h**, i.e.:

$$\mathbf{x} \approx \mathbf{W}\mathbf{h} + \mathbf{b}$$

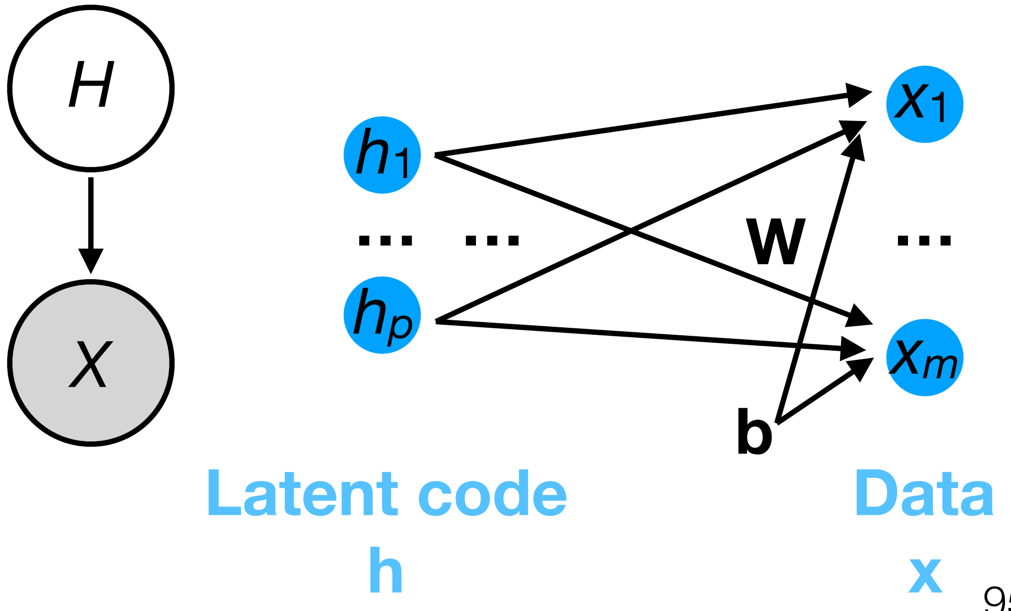
- We can represent this probabilistically as:

$$P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) = \mathcal{N}(\mathbf{x}; \mathbf{W}\mathbf{h} + \mathbf{b}, \sigma^2\mathbf{I})$$
$$P(\mathbf{h}) = \mathcal{N}(\mathbf{h}; \mathbf{0}, \mathbf{I})$$



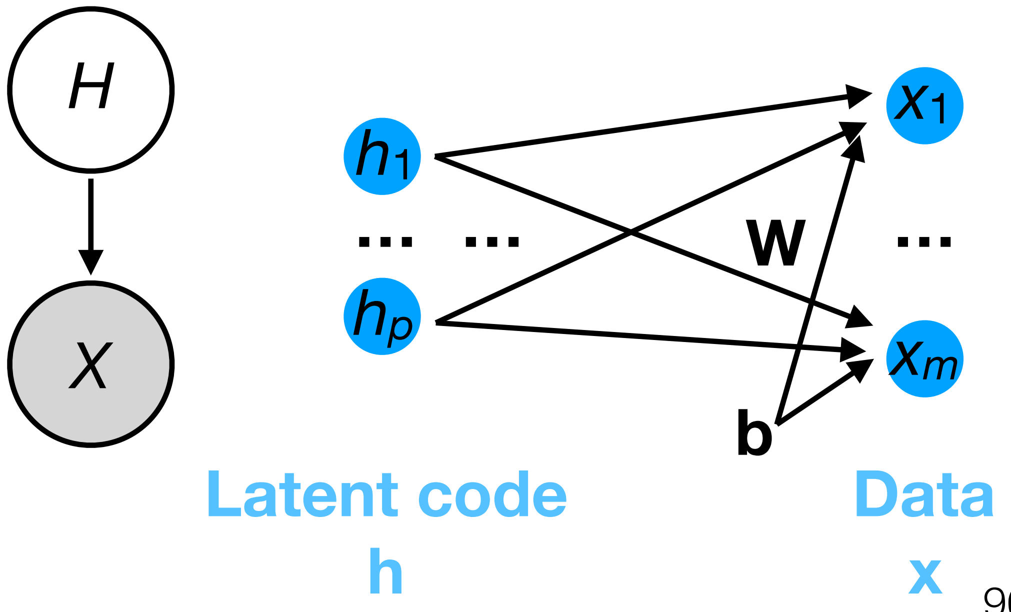
Principal component analysis (PCA)

- Given a set of training data $\mathcal{D}$, we can optimize **W** and **b** using maximum-likelihood estimation (MLE).



Principal component analysis (PCA)

- The MLE for **b** is the mean of the data $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{n}$.
- The MLE for **W** is the same set of principal eigenvectors from standard PCA (but weighted by their eigenvalues).
  - The weights ensure that *P*(**h**) is **0**-mean with **I**-covariance.

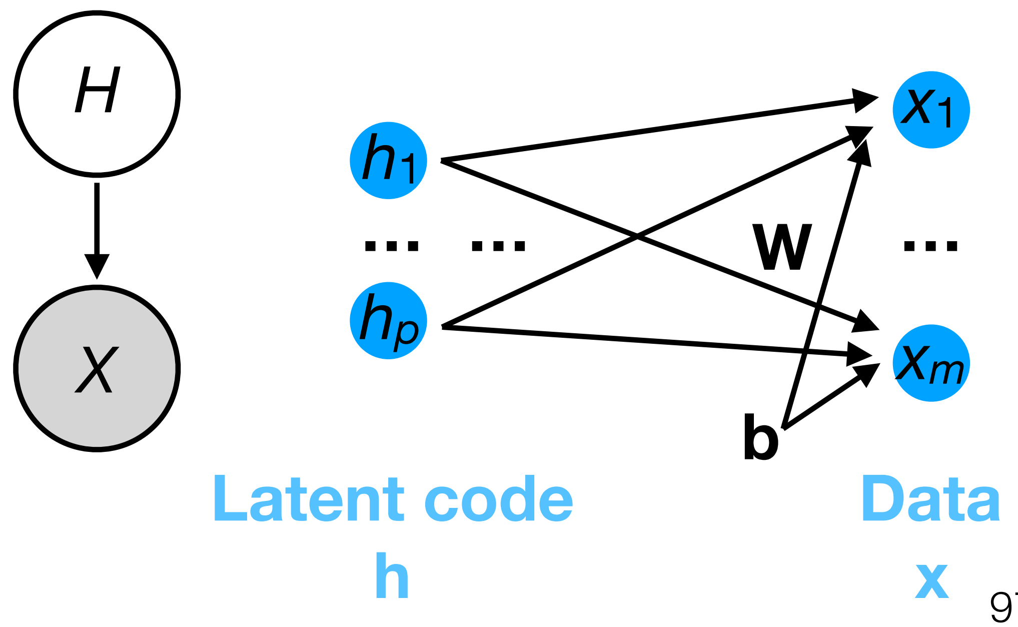


Principal component analysis (PCA)

- This probabilistic PCA model allows us to **generate** novel examples:

1. Sample from the prior distribution over **h**.

$$P(\mathbf{h}) = \mathcal{N}(\mathbf{h}; \mathbf{0}, \mathbf{I})$$



Principal component analysis (PCA)

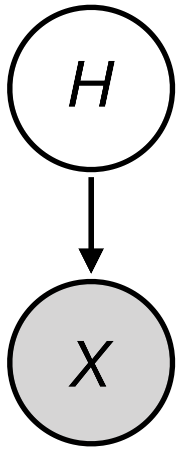
- This probabilistic PCA model allows us to **generate** novel examples:

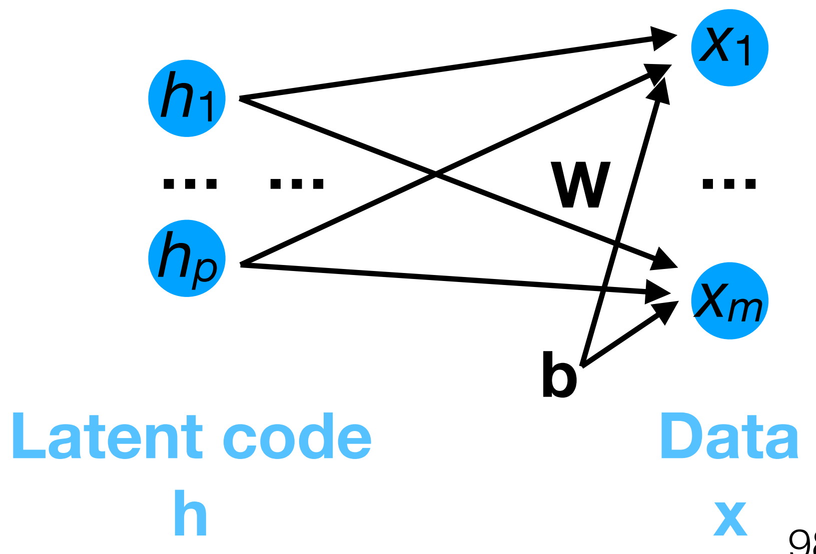
1. Sample from the prior distribution over **h**.

$$P(\mathbf{h}) = \mathcal{N}(\mathbf{h}; \mathbf{0}, \mathbf{I})$$

2. Sample from the conditional distribution of **x** | **h**.

$$P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) = \mathcal{N}(\mathbf{x}; \mathbf{W}\mathbf{h} + \mathbf{b}, \sigma^2 \mathbf{I})$$





Principal component analysis (PCA)

- Demo of probabilistic PCA on MNIST.
- Demo of probabilistic PCA on GENKI4K.

Principal component analysis (PCA)

- Demo of probabilistic PCA on MNIST.
- Demo of probabilistic PCA on GENKI4K.
- Overall, the results are not very good.
    - Problem: the generative model is **linear** — easy to train and sample, but too simplistic.
    - Later on, we will see more powerful deep latent variable models (variational auto-encoders (VAE)) and other generative neural networks (generative-adversarial networks (GAN)).

Principal component analysis (PCA)

- What if you want to know the latent code **h** for a given **x**, i.e., *P*(**h** | **x**)? We can use Bayes' rule:

$$P(\mathbf{h} \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) = \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{P(\mathbf{x} \mid \mathbf{W}, \mathbf{b})}$$

Principal component analysis (PCA)

- What if you want to know the latent code **h** for a given **x,** i.e., *P*(**h** | **x**)? We can use Bayes' rule:

$$
\begin{aligned}
P(\mathbf{h} \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) &= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{P(\mathbf{x} \mid \mathbf{W}, \mathbf{b})} \\
&= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{\int d\mathbf{h} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h})} \\
&= \frac{1}{Z(\mathbf{x})} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})
\end{aligned}
$$

Principal component analysis (PCA)

- What if you want to know the latent code **h** for a given **x**, i.e., *P*(**h** | **x**)? We can use Bayes' rule:

$$
\begin{aligned}
P(\mathbf{h} \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) &= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{P(\mathbf{x} \mid \mathbf{W}, \mathbf{b})} \\
&= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{\int d\mathbf{h} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h})} \\
&= \frac{1}{Z(\mathbf{x})} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})
\end{aligned}
$$

Here, *Z* is called the normalization constant.

Principal component analysis (PCA)

- In general, the integral in *Z* is **intractable**. However, for PCA it is a Gaussian and can be computed in closed form.

$$
\begin{aligned}
P(\mathbf{h} \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) &= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{P(\mathbf{x} \mid \mathbf{W}, \mathbf{b})} \\
&= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{\int d\mathbf{h} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h})} \\
&= \frac{1}{Z(\mathbf{x})} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})
\end{aligned}
$$

Here, *Z* is called the normalization constant.

Principal component analysis (PCA)

- For more complex LVMs, we can use techniques such as **variational inference** to compute *Z* approximately.

$$
\begin{aligned}
P(\mathbf{h} \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) &= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{P(\mathbf{x} \mid \mathbf{W}, \mathbf{b})} \\
&= \frac{P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})}{\int d\mathbf{h} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h})} \\
&= \frac{1}{Z(\mathbf{x})} P(\mathbf{x} \mid \mathbf{h}, \mathbf{W}, \mathbf{b}) P(\mathbf{h} \mid \mathbf{W}, \mathbf{b})
\end{aligned}
$$

Here, *Z* is called the normalization constant.