

# Deep learning

Unpacking Transformers, LLMs and image generation

## Session 2

# Syllabus

Session	Lecture	TP
1	Intro to DL Gradient descent and backprop	Intro to micrograd
2	DL fundamentals I <ul style="list-style-type: none"><li>• Backprop</li><li>• Loss functions</li><li>• Neural Probabilistic Language Model (Bengio 2003)</li></ul>	Bigram model and MLP for next-character prediction
3	DL fundamentals II <ul style="list-style-type: none"><li>• Activation function</li><li>• Regularization</li><li>• Initialization</li><li>• Residual networks</li><li>• Normalization</li></ul> Recurrent Neural Networks	Backprop ninja MLP in pytorch (*) add batchnorm to TP2 (*) MLP for MNIST (Fleuret 3)
4	Attention and Transformers	GPT from scratch (*) Fleuret practical 5 and 6
5	DL for computer vision: convnets, <u>unets</u>	Convnets for Fashion MNIST (*) Fleuret practical 4
6	VAE & Diffusion models	1D diffusion model 2D diffusion model (*) train on GPU  <b>Quiz</b>

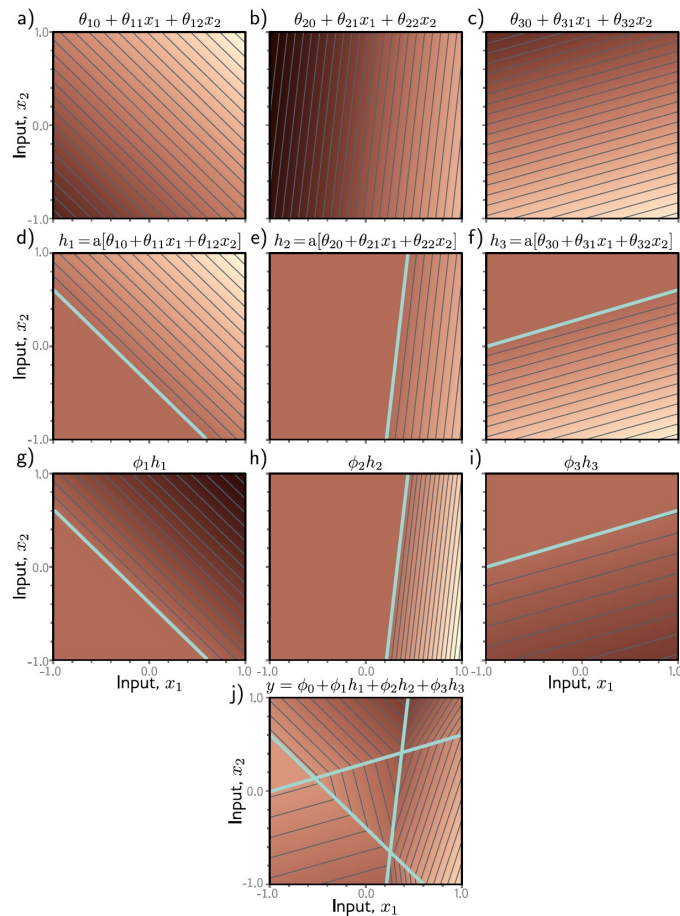
There is no reason for deep learning to work

---

**Training:** Finding the global optimum of an arbitrary non-convex function is NP-hard (Murty & Khabadi, 1987).

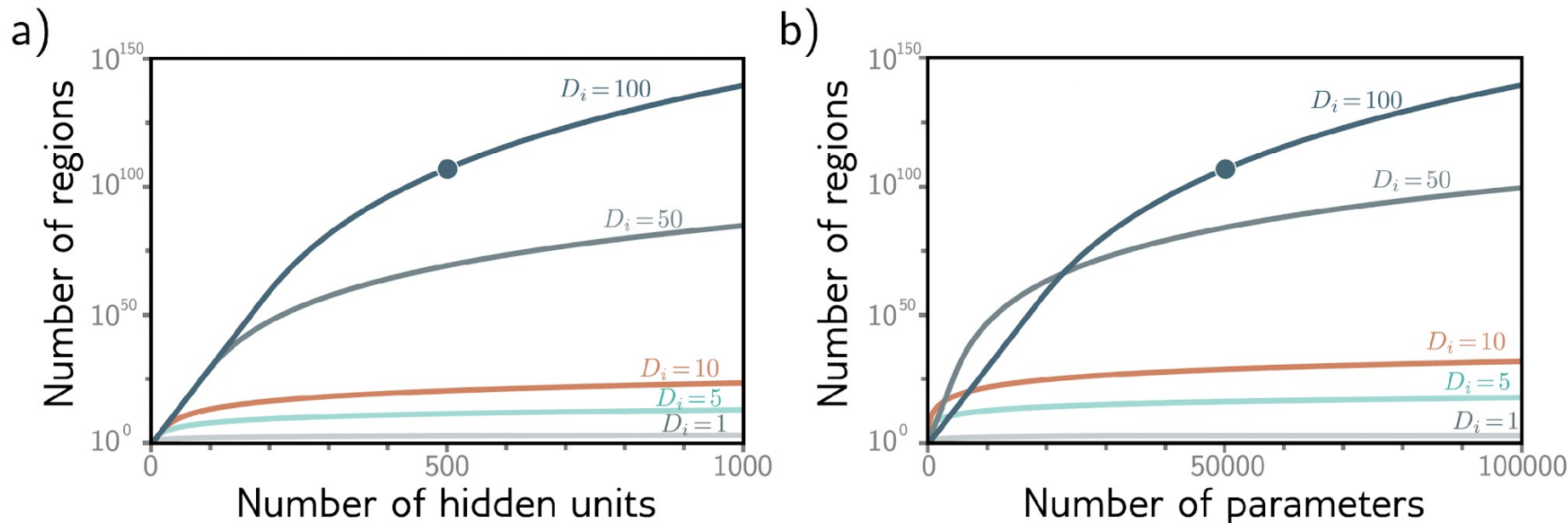
**Generalization:** deep networks generate way more regions than training samples.

# Neural networks generate large number of regions



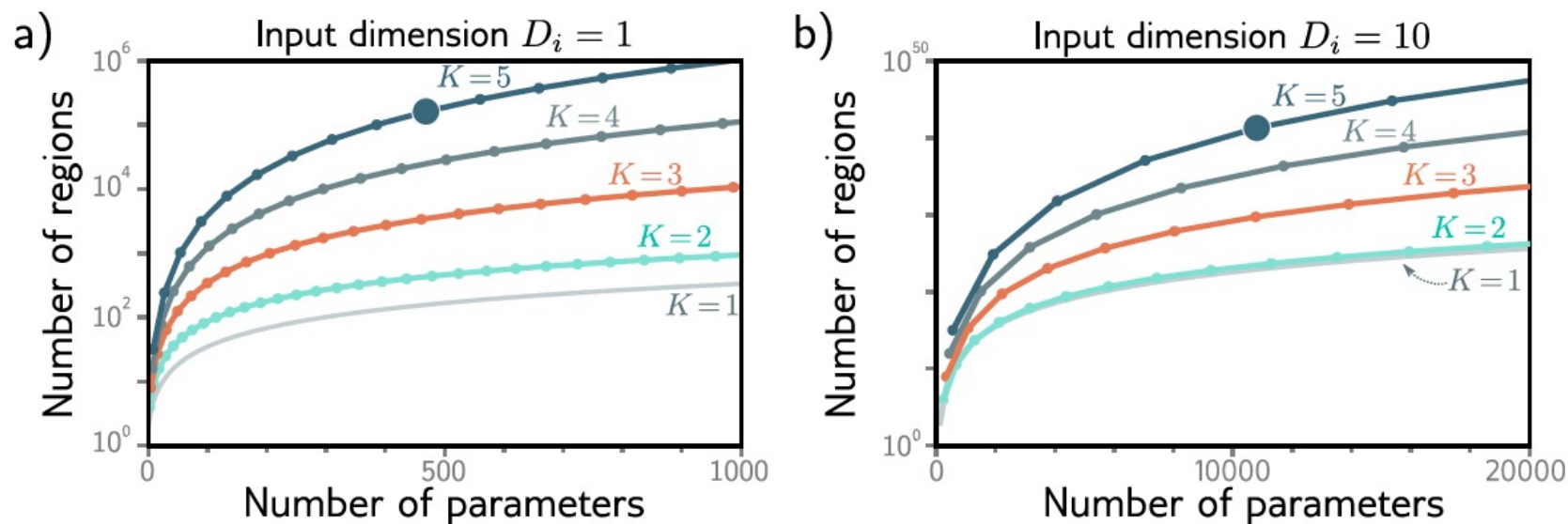
A neural network generate a number of linear sub-regions.

# Shallow networks generate large number of regions



**Figure 3.9** Linear regions vs. hidden units. a) Maximum possible regions as a function of the number of hidden units for five different input dimensions  $D_i = \{1, 5, 10, 50, 100\}$ . The number of regions increases rapidly in high dimensions; with  $D = 500$  units and input size  $D_i = 100$ , there can be greater than  $10^{107}$  regions (solid circle). b) The same data are plotted as a function of the number of parameters. The solid circle represents the same model as in panel (a) with  $D = 500$  hidden units. This network has 51,001 parameters and would be considered very small by modern standards.

## Deep networks generate **even more** of regions / parameter count



**Figure 4.7** The maximum number of linear regions for neural networks increases rapidly with the network depth. a) Network with  $D_i = 1$  input. Each curve represents a fixed number of hidden layers  $K$ , as we vary the number of hidden units  $D$  per layer. For a fixed parameter budget (horizontal position), deeper networks produce more linear regions than shallower ones. A network with  $K = 5$  layers and  $D = 10$  hidden units per layer has 471 parameters (highlighted point) and can produce 161,051 regions. b) Network with  $D_i = 10$  inputs. Each subsequent point along a curve represents ten hidden units. Here, a model with  $K = 5$  layers and  $D = 50$  hidden units per layer has 10,801 parameters (highlighted point) and can create more than  $10^{40}$  linear regions.

## Deep networks generate **even more** of regions / parameter count

Deep neural networks create much more complex functions for a fixed parameter budget.

With 1D input, 1D output,  $D$  hidden units:

	# regions	# parameters
Shallow network	$D + 1$	$3D + 1$
Deep network ( $K$ layers)	$(D + 1)^K$	$3D + 1 + (K - 1)D(D + 1)$



# Let's venture into the variations of a deep networks

---

Network architecture and inductive bias

Loss function

Activation function

Regularization

Initialization

Residual networks

Batch norm, layer norm



# Inductive bias

---

Set of assumptions made by the model about the relationship between input data and output data.

Examples:

- Minimum features
- Maximum margin (SVM)
- Minimum cross-validation error
- Neural net architecture (convnet, transformer)

# Do networks have to be deep?

---

Empirical evidence: shallow networks don't work as well as deeper ones.

Intuition:

1. Deep networks can represent more complex functions with the parameter count
2. Deep networks are easier to train
3. Deep network impose better inductive bias

## The challenges of depth

---

- Vanishing/exploding gradients
- Shattered gradients

In short, depth is required but comes with challenges that need to be addressed.

# Let's venture into the variations of a deep networks

---

Network architecture and inductive bias

**Loss function**

Activation function

Regularization

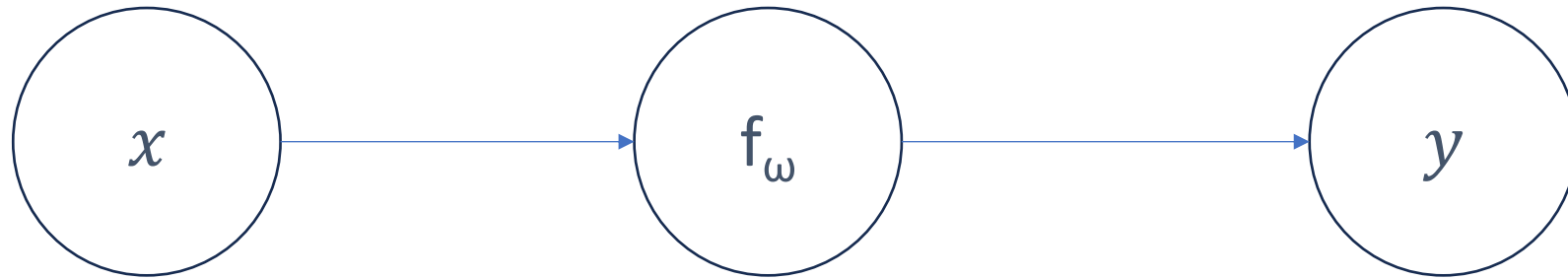
Initialization

Residual networks

Batch norm, layer norm

# Loss functions

---



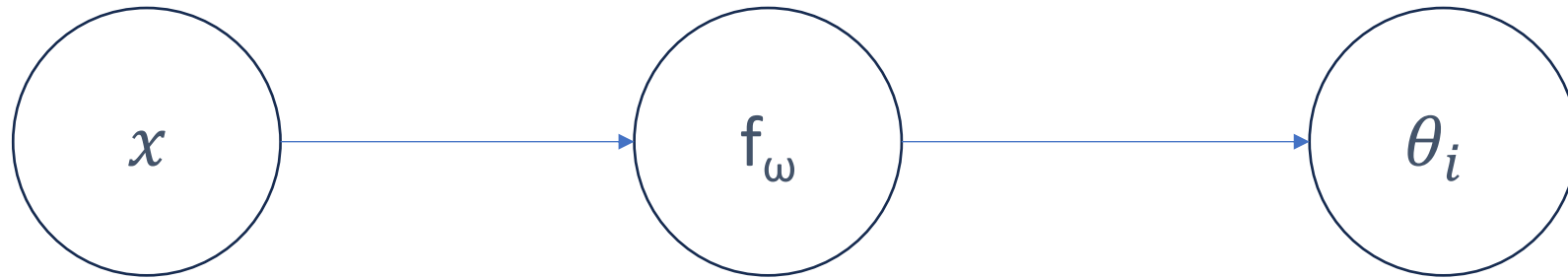
$$\ell = (y - \hat{y})^2$$

*ground-truth*

A blue arrow originates from the text 'ground-truth' and points upwards and to the left, terminating at the  $y$  term in the equation  $\ell = (y - \hat{y})^2$ .

# Loss functions

---

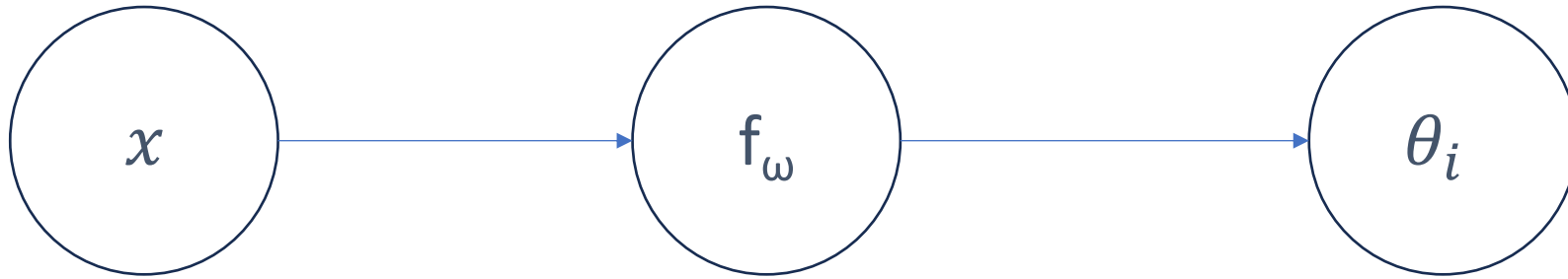


$$f_\omega(x_i) = \theta_i$$

*Parameters of a distribution*

# Loss functions

---



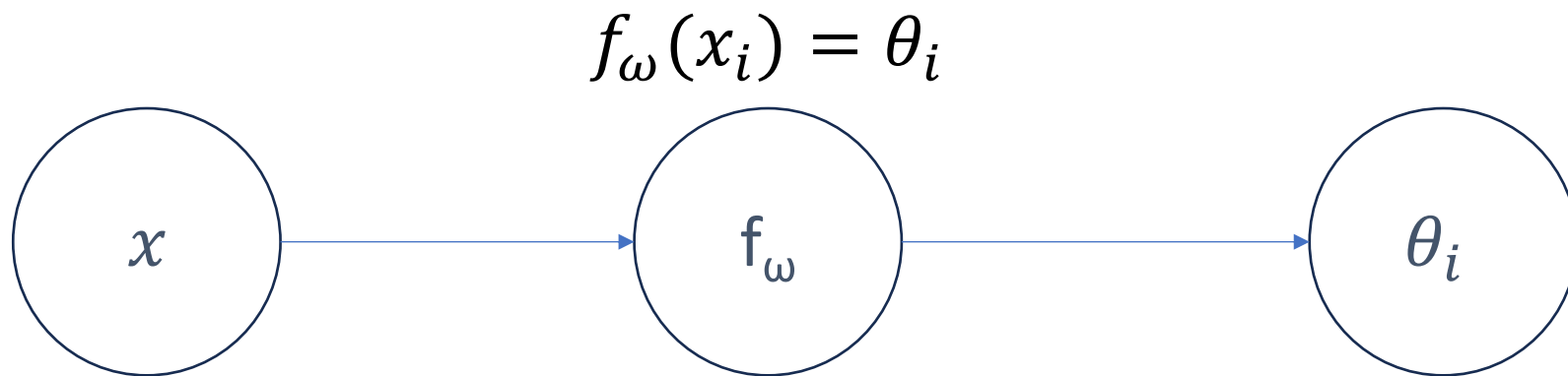
$$f_{\omega}(x_i) = \theta_i$$

The distribution is chosen based on the domain.

The model computes the optimal  $\theta_i$  given the data.



# Loss functions

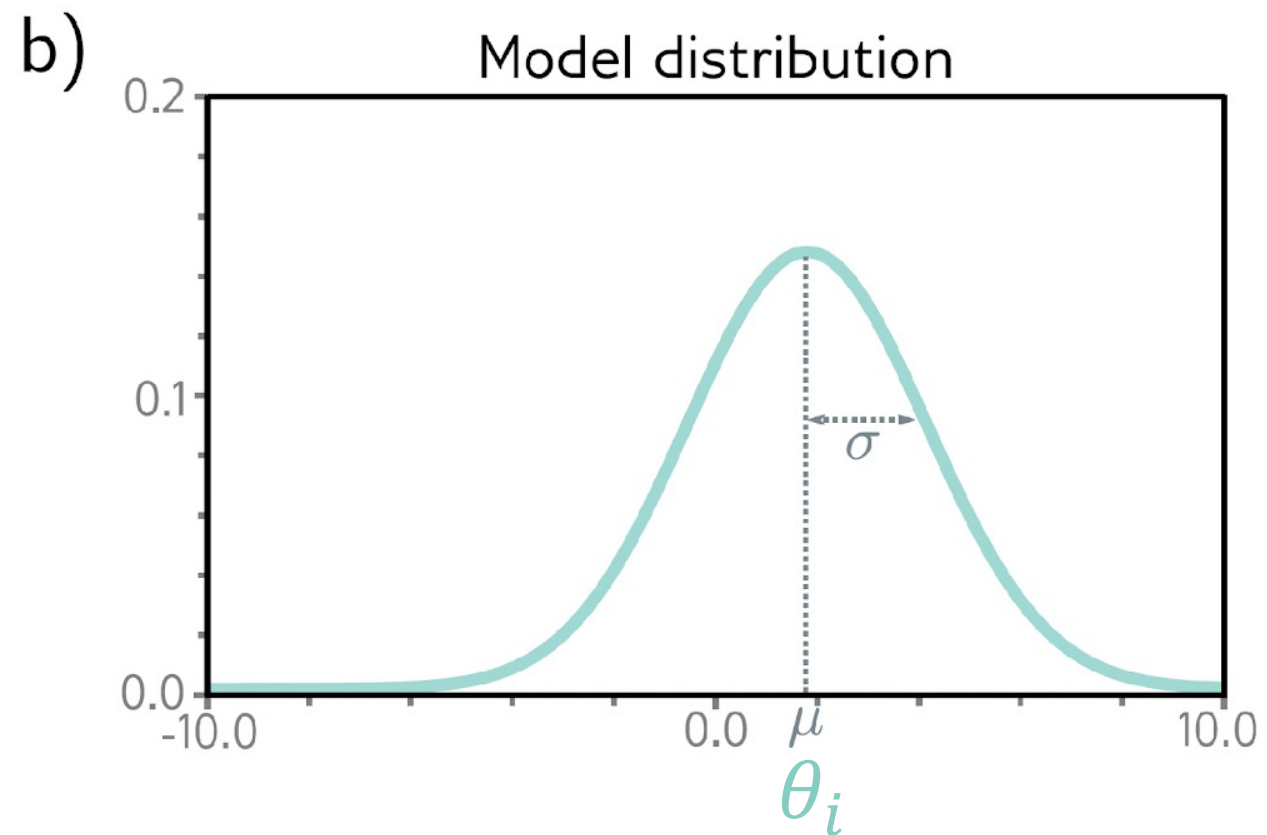
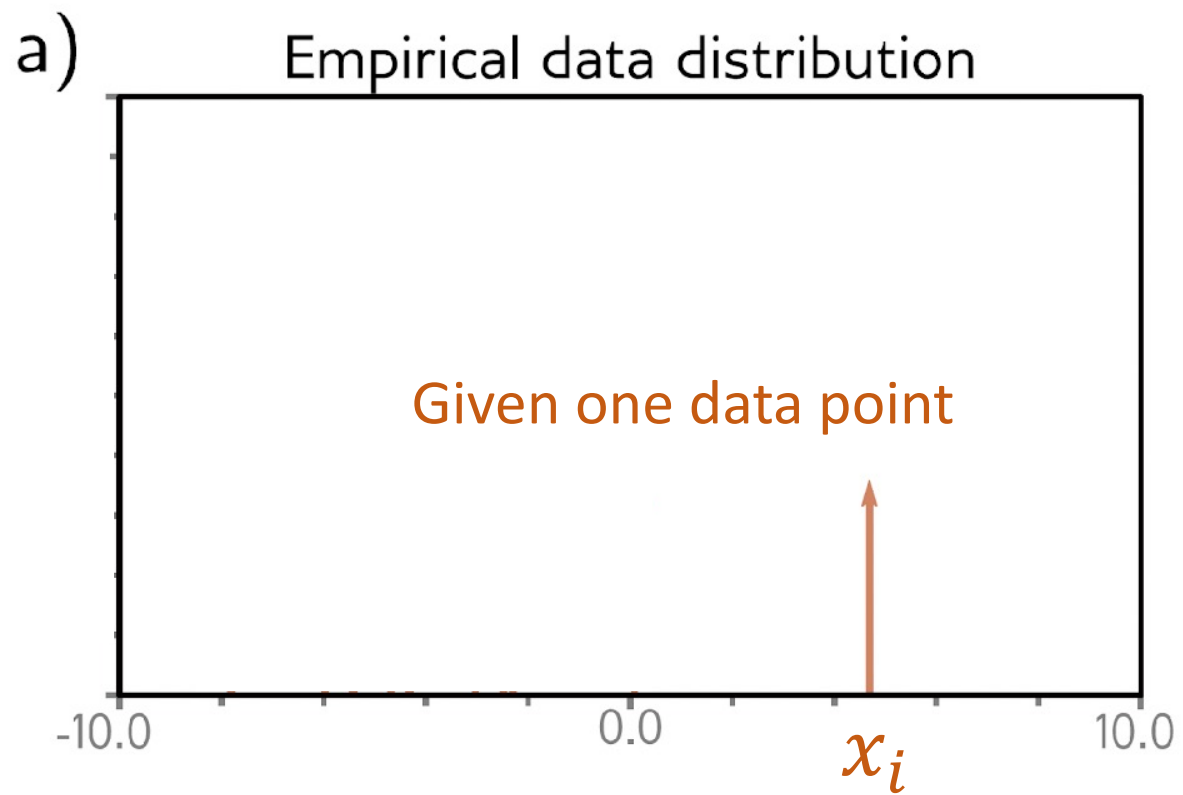


Example: univariate regression

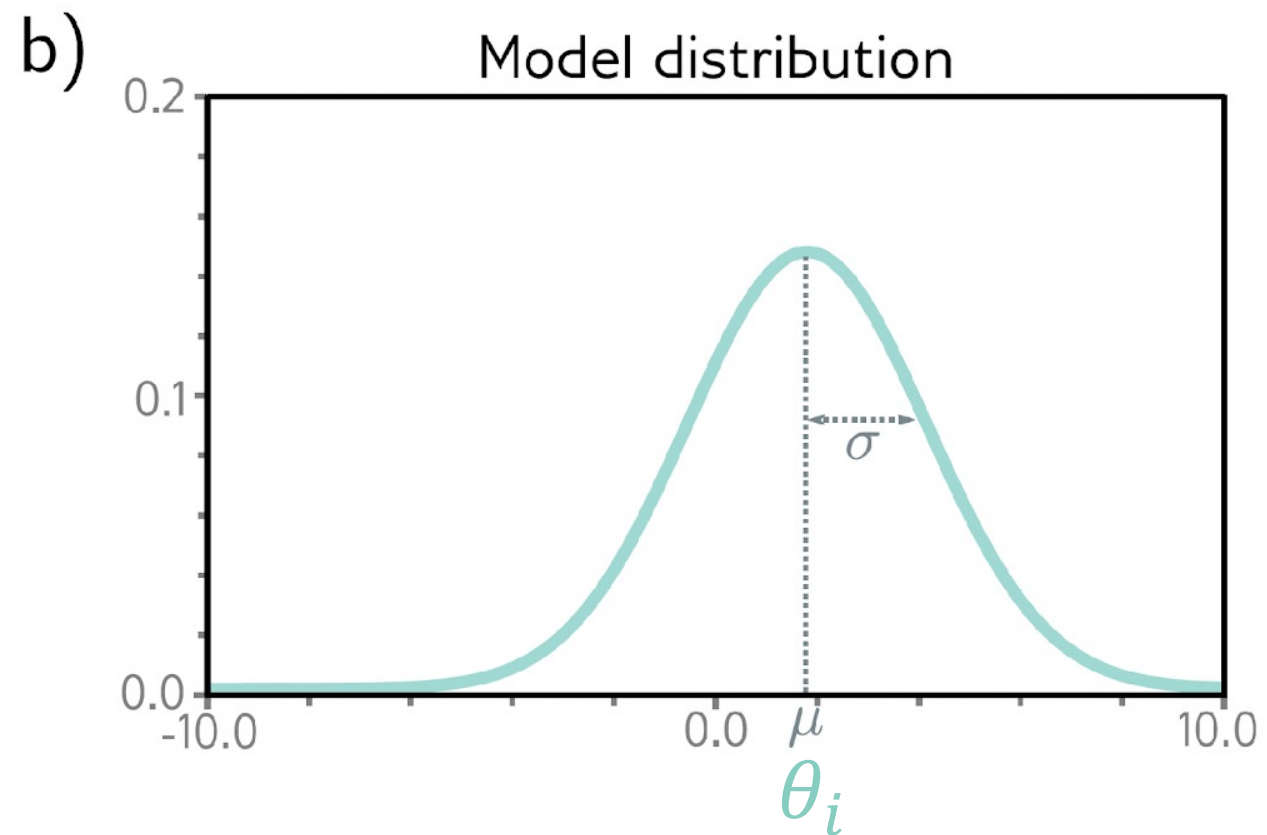
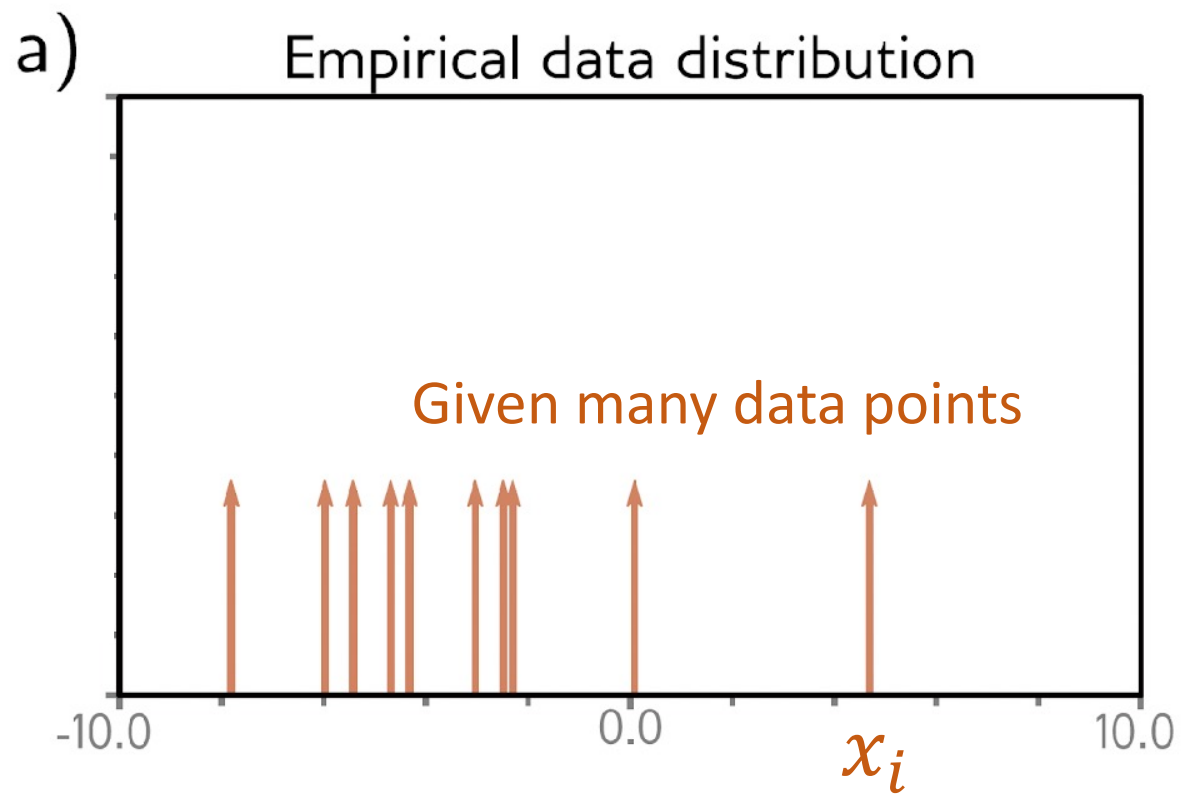
$$\Pr(y|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

$\theta$

The diagram shows the Gaussian distribution formula. Below the formula, the Greek letter  $\theta$  is written. A vertical blue arrow points from  $\theta$  up to the  $\mu$  in the formula, indicating that  $\theta$  represents the mean parameter.



Predict the corresponding distribution parameter



Predict the corresponding distribution parameters

$$f_{\omega}(x_i) = \theta_i$$

# Loss functions

---

$$\hat{\omega} = \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | x_i) \right]$$

$$= \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | \theta_i) \right]$$

$$= \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | f_{\omega}(x_i)) \right]$$

# Loss functions

$$\hat{\omega} = \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | x_i) \right]$$

$\Pr(y_1, y_2, \dots, y_N | x_1, x_2, \dots, x_N)$   
*Data is assumed i.i.d*

$$= \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | \theta_i) \right]$$

$$= \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | f_{\omega}(x_i)) \right]$$

# Loss functions

$$\hat{\omega} = \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | x_i) \right]$$

$$= \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | \theta_i) \right]$$

$$= \operatorname{argmax}_{\omega} \left[ \prod_{i=1}^N \Pr(y_i | f_{\omega}(x_i)) \right]$$

$$= \operatorname{argmax}_{\omega} \left[ \sum_{i=1}^N \log[\Pr(y_i | f_{\omega}(x_i))] \right]$$

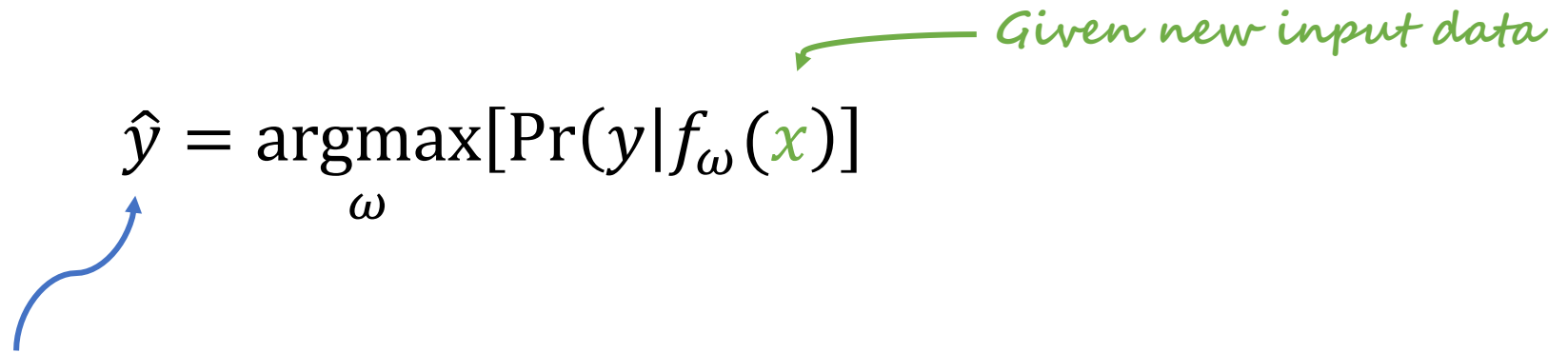
$$= \operatorname{argmin}_{\omega} \left[ - \sum_{i=1}^N \log[\Pr(y_i | f_{\omega}(x_i))] \right] \quad \text{Negative log likelihood (NLL)}$$

# Inference

---

$$\hat{y} = \underset{\omega}{\operatorname{argmax}} [\operatorname{Pr}(y | f_{\omega}(x))]$$

*Given new input data*



*Optimal choice: maximum of the distribution*

*Or sample from the distribution!*



# Loss functions

In the case of the univariate regression, the NLL is equivalent to least squares.

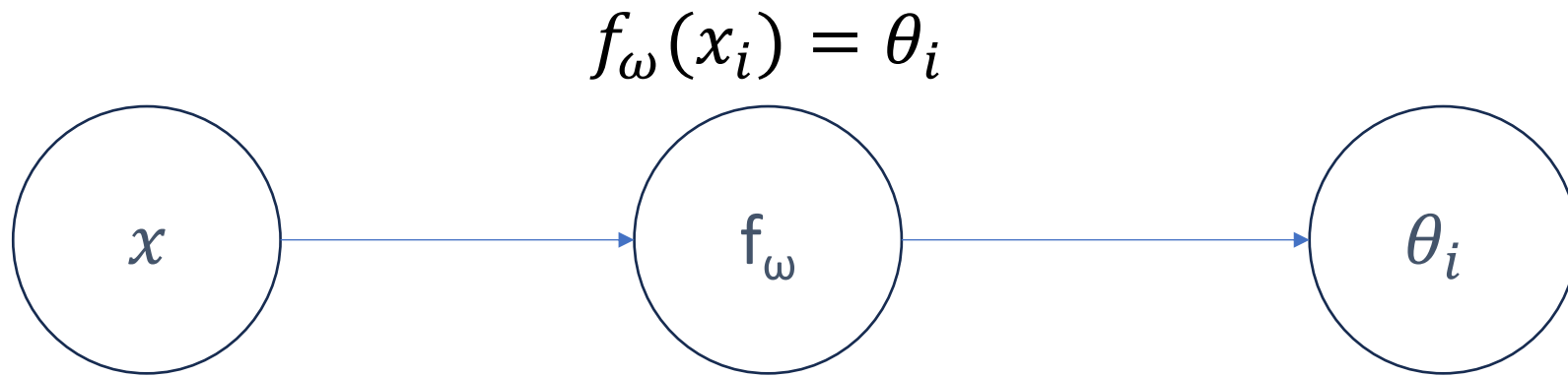
$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \sum_{i=1}^N \log[\operatorname{Pr}(y_i | f_{\omega}(x_i))] \right] \quad \text{Negative log likelihood (NLL)}$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \sum_{i=1}^N \log \left[ \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \mu)^2}{2\sigma^2}} \right] \right]$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \sum_{i=1}^N \log \left[ \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - f_{\omega}(x_i))^2}{2\sigma^2}} \right] \right]$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ \sum_{i=1}^N (y_i - f_{\omega}(x_i))^2 \right] \quad \text{least squares}$$

# Loss functions



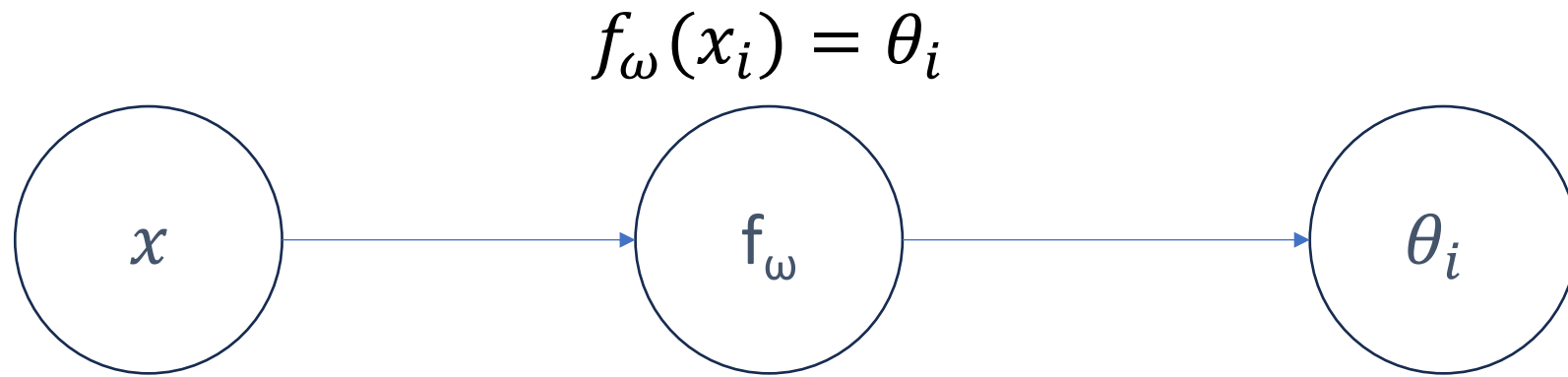
Example: binary classification

$$\Pr(y|\lambda) = \begin{cases} 1 - \lambda & y = 0 \\ \lambda & y = 1 \end{cases}$$

$\theta$

An upward-pointing arrow connects the symbol  $\theta$  to the parameter  $\lambda$  in the probability expression above.

# Loss functions



Example: binary classification

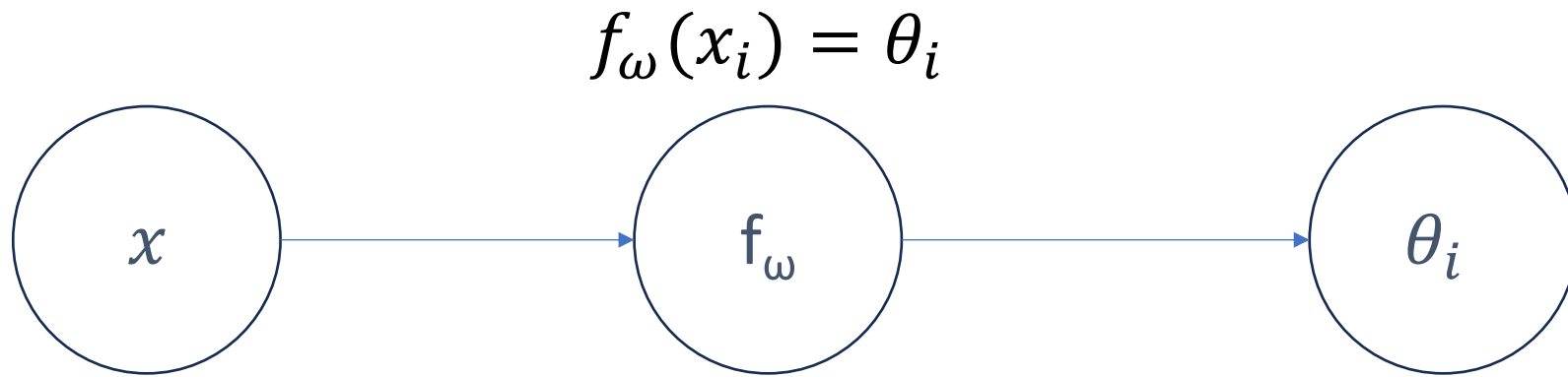
$$\Pr(y|\lambda) = \begin{cases} 1 - \lambda & y = 0 \\ \lambda & y = 1 \end{cases}$$

*NLL*

$$\ell = \sum_{i=1}^N -(1 - y_i) \log[1 - \sigma(f_{\omega}(x_i))] - y_i \log[\sigma(f_{\omega}(x_i))]$$

*$\sigma$ : sigmoid function*

# Loss functions

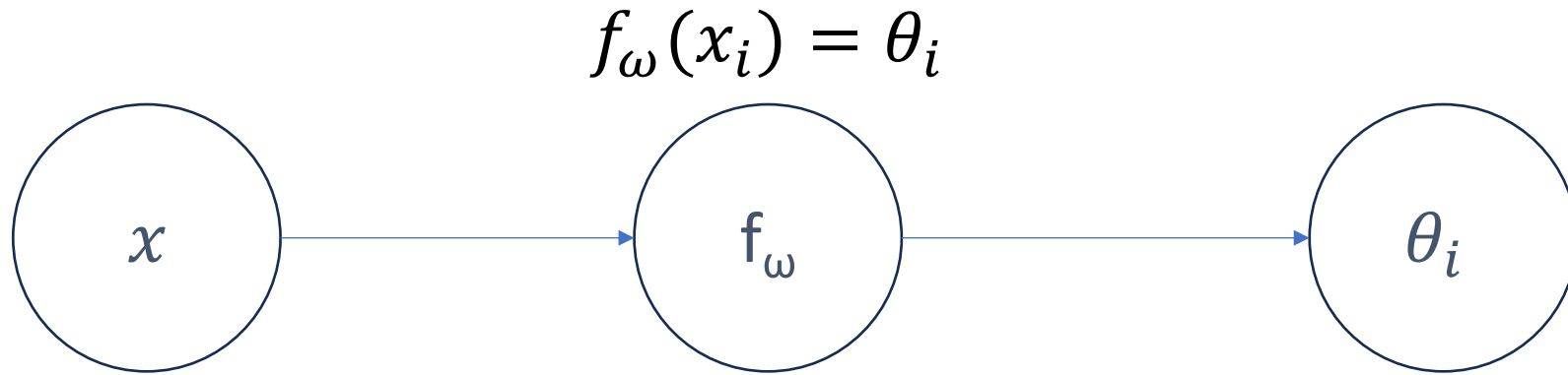


Example: multiclass classification

$$\Pr(y = k) = \lambda_k \qquad \sum \lambda_k = 1 \qquad 0 < \lambda_k < 1$$

$$\Pr((y = k|x)) = \text{softmax}_k[f_{\omega}(x)] \qquad \text{softmax}(\mathbf{z}) = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

# Loss functions



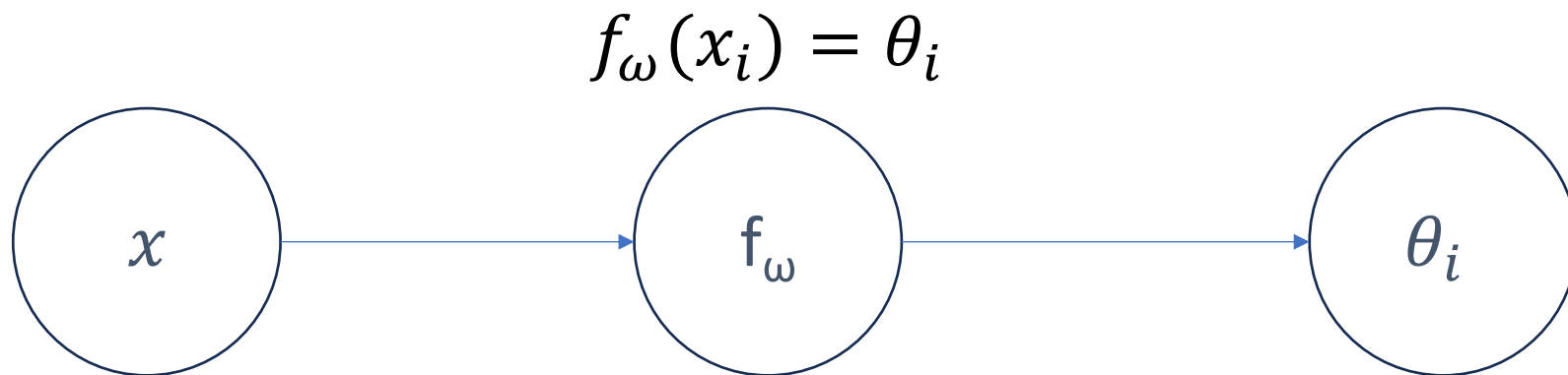
Example: multiclass classification

$$\Pr(y = k) = \lambda_k \quad \sum \lambda_k = 1$$

*NLL*

$$\ell = - \sum_{i=1}^N \log \left[ \text{softmax}_{y_i} [f_{\omega}(x_i)] \right]$$

# Loss functions



Example: multiclass classification

$$\Pr(y = k) = \lambda_k \quad \sum \lambda_k = 1$$

NLL

$$\ell = - \sum_{i=1}^N \log \left[ \text{softmax}_{y_i} [f_{\omega}(x_i)] \right]$$

*Wait, can I differentiate softmax?*

*Yes, and you will do it by hand in TP3!*

# Loss functions

---

$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}} \left[ - \sum_{i=1}^N \log[\operatorname{Pr}(y_i | f_{\omega}(x_i))] \right] \quad \text{Negative log likelihood (NLL)}$$

is equivalent to the cross-entropy loss



# Loss functions

---

Given two distributions  $q(z)$  and  $p(z)$ , the distance between the two distributions can be computed with:

$$D_{KL}(q|p) = \int_{-\infty}^{\infty} q(z) \log(q(z)) dz - \int_{-\infty}^{\infty} q(z) \log(p(z)) dz$$

Given an empirical distribution  $q(y)$  and a model distribution  $Pr(y|\omega)$ , we want to minimize the KL divergence:

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ \int_{-\infty}^{\infty} q(y) \log(q(y)) dy - \iint_{-\infty}^{\infty} q(y) \log[Pr(y|\omega)] dy \right]$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \iint_{-\infty}^{\infty} q(y) \log[Pr(y|\omega)] dy \right]$$

# Loss functions

Given two distributions  $q(z)$  and  $p(z)$ , the distance between the two distributions can be computed with:

$$D_{KL}(q|p) = \int_{-\infty}^{\infty} q(z) \log(q(z)) dz - \int_{-\infty}^{\infty} q(z) \log(p(z)) dz$$

Given an empirical distribution  $q(y)$  and a model distribution  $Pr(y|\omega)$ , we want to minimize the KL divergence:

*entropy of  $q$*   *divergence between  $q$  and  $p$*  

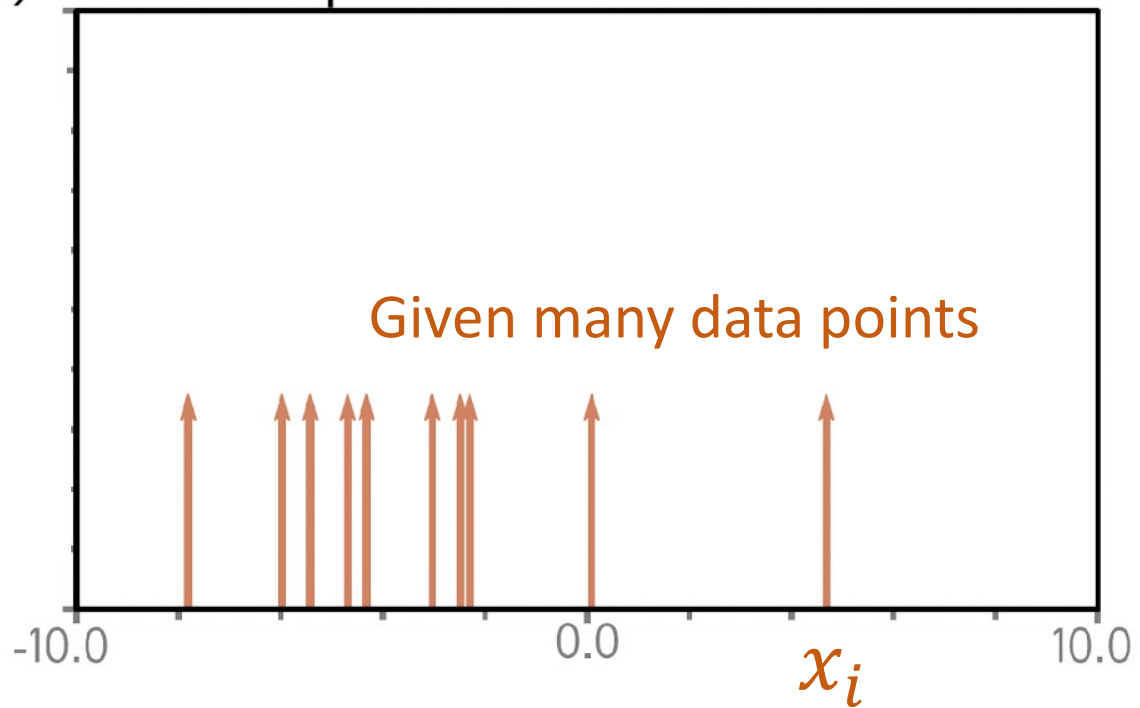
$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ \int_{-\infty}^{\infty} q(y) \log(q(y)) dy - \iint_{-\infty}^{\infty} q(y) \log[Pr(y|\omega)] dy \right]$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \iint_{-\infty}^{\infty} q(y) \log[Pr(y|\omega)] dy \right]$$

a)

## Empirical data distribution

Given many data points



$$q(y) = \frac{1}{N} \sum_{i=1}^N \delta[y - y_i]$$

 *$\delta$ : dirac*

# Loss functions

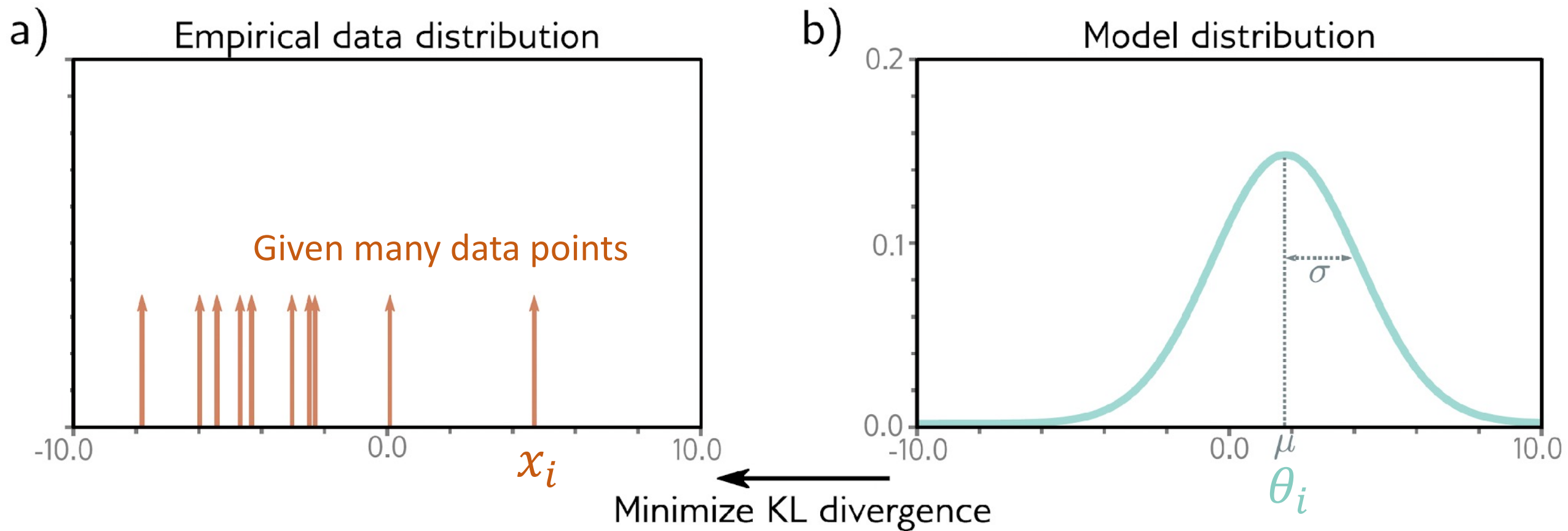
Given an empirical distribution  $q(y)$  and a model distribution  $Pr(y|\omega)$ , we want to minimize the KL divergence:

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \iint_{-\infty}^{\infty} q(y) \log[Pr(y|\omega)] dy \right] \quad \text{cross-entry loss}$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \iint_{-\infty}^{\infty} \left( \frac{1}{N} \sum_{i=1}^N \delta[y - y_i] \right) \log[Pr(y|\omega)] dy \right]$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \frac{1}{N} \sum_{i=1}^N \log[Pr(y_i|\omega)] \right]$$

$$\hat{\omega} = \operatorname{argmin}_{\omega} \left[ - \sum_{i=1}^N \log[Pr(y_i|\omega)] \right] \quad \text{NLL}$$



**Figure 5.12** Cross-entropy method. a) Empirical distribution of training samples (arrows denote Dirac delta functions). b) Model distribution (a normal distribution with parameters  $\theta = \mu, \sigma^2$ ). In the cross-entropy approach, we minimize the distance (KL divergence) between these two distributions as a function of the model parameters  $\theta$ .

# Loss functions

---

Definition of **cross-entropy loss** of distribution  $p$  relative to distribution  $q$  over the set  $\mathcal{X}$ :

$$H(p, q) = -E_p[\log q]$$

where  $E_p[\cdot]$  is the expected value operator with respect to distribution  $p$ .

In the continuous case:

$$H(p, q) = - \int_{\mathcal{X}} P(x) \log Q(x) dx$$

In the discrete case:

$$H(p, q) = - \sum_{x \in \mathcal{X}} p(x) \log q(x)$$

## TP2: makemore

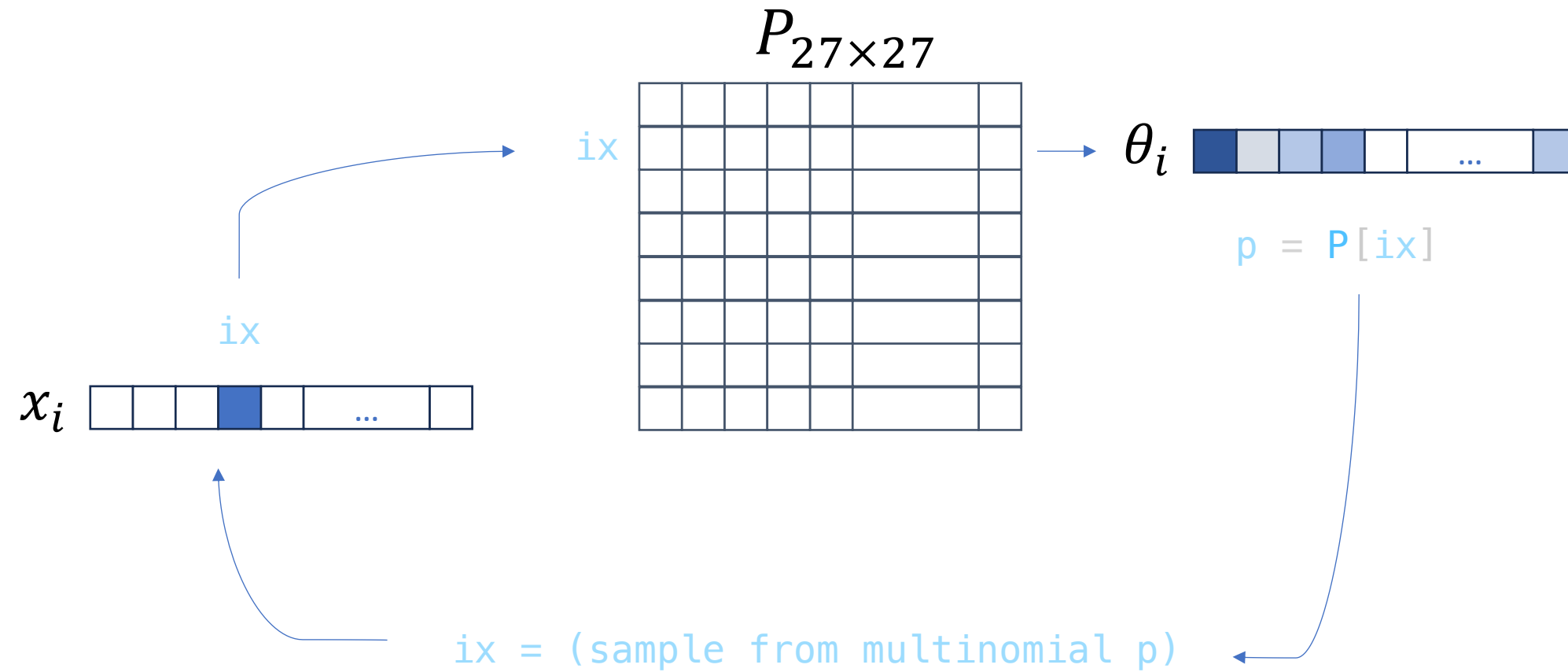
---

Goal: Given a bunch of names, generate more “name-like” words.

1. Build a simple bigram model for next-character prediction
2. Build the same bigram model using the NLL loss
3. Implement a better model: [[Bengio et al., 2003](#)]



## Step 1: bigram “by hand”



## Step 1: bigram “by hand”

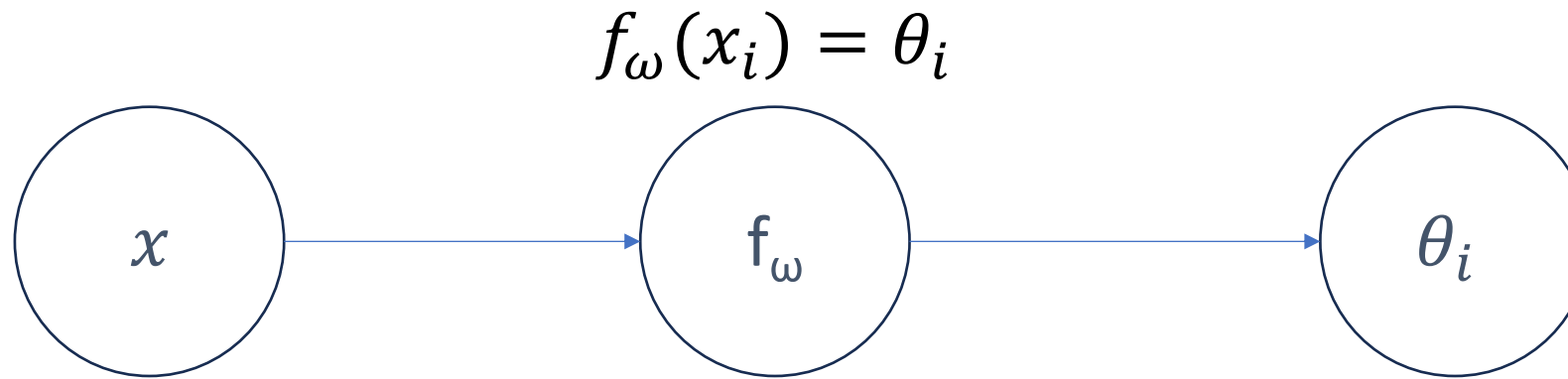
---

The dot character (.) marks the beginning and end of a word.

When sampling, you need to stop when you hit that special character.

How to initialize a torch matrix of size 27x27 containing floats?

## Step 2: bigram as a learnable matrix



$$x_i = [0, 0, 0, 1, 0, \dots, 0]$$

*One-hot encoding of letter 'd'*

$\omega$  is a matrix  $W_{27 \times 27}$  such that

$\theta_i = \text{softmax}(x \cdot W)$  is a  $N \times 27$  vector representing the distribution of the next character for each sample

## Step 2: bigram as a learnable matrix

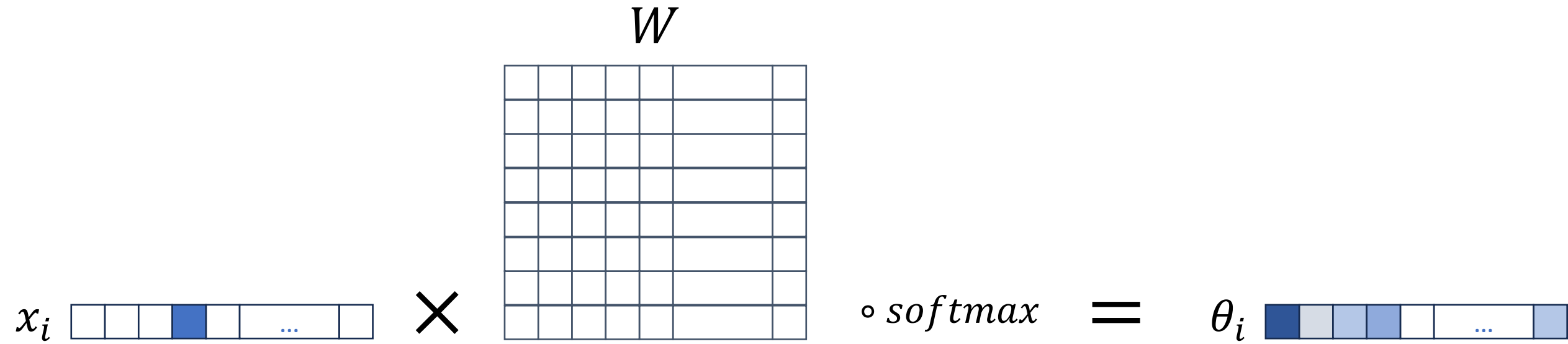
---

$x_i$ 

--	--	--	--	--	--	--	--

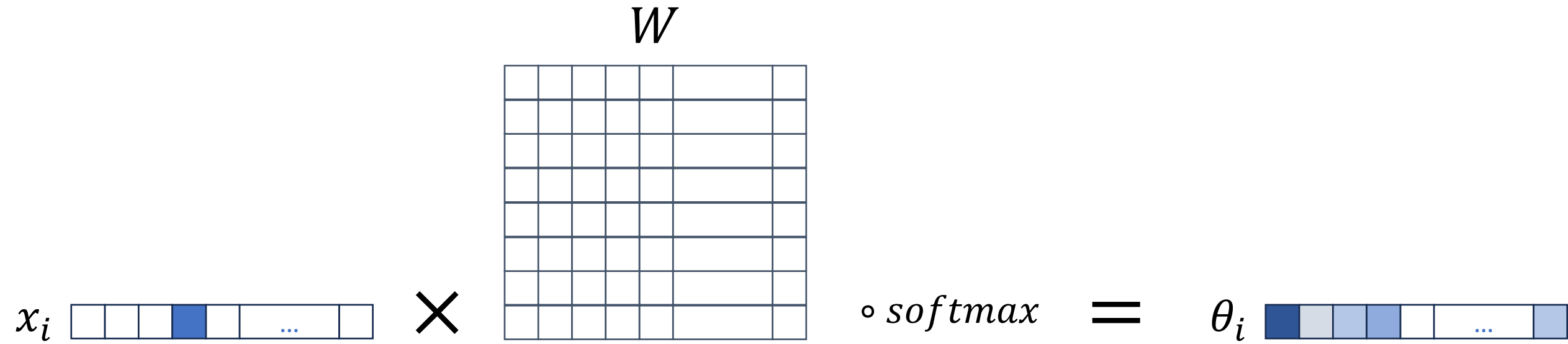
*One-hot encoding of letter 'd'*

## Step 2: bigram as a learnable matrix



*One-hot encoding of letter 'd'*

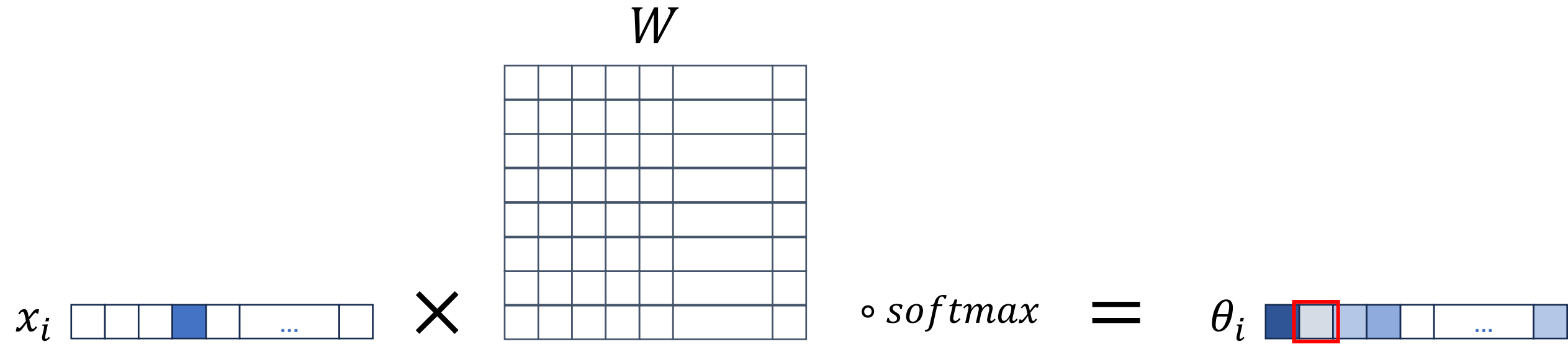
## Step 2: bigram as a learnable matrix



We want to minimize the KL divergence or NLL

$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}} \left[ - \sum_{i=1}^N \log[Pr(y_i|\omega)] \right] \quad \text{NLL}$$

## Step 2: bigram as a learnable matrix



We want to minimize the KL divergence or NLL

$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}} \left[ - \sum_{i=1}^N \log \boxed{Pr(y_i | \omega)} \right] \quad \text{NLL}$$

## Step 2: bigram as a learnable matrix



$x_i$



$\theta_i$



*Loss = mean of log of the red values*



## Step 2: bigram as a learnable matrix

---

```
#forward pass
xenc = ??? # encode xs with F.one_hot
logits = ??? # multiply by W
counts = ??? # softmax
probs = ??? #
loss = ??? # sum of logs of probs
```

## A few tips...

---

```
import torch.nn.functional as F
```

$x \cdot W$  is written as `x @ W`

One-hot encoding: `F.one_hot(x, num_classes=...).float()`

For inference `z.multinomial()`

Normalizing a matrix  $W_{27 \times 27}$  by row requires the `keepdim` parameter somewhere...

## A few tips...

---

```
>>> a = torch.randn((7,7))
>>> a
tensor([[ 1.2555,  0.6821,  0.9131, -0.7238,  0.5636, -2.8689, -0.4744],
        [ 2.1393, -0.8737,  2.4039,  0.0056,  0.6169, -0.2245, -0.2242],
        [ 0.1821, -0.4250, -0.1115, -0.3568, -2.2182,  0.9574,  1.9415],
        [-0.2646,  1.7013, -2.7297,  0.3786, -1.7883,  0.8484, -0.1894],
        [-0.5430, -0.2352,  0.4820, -0.0737,  0.8632,  0.1648,  1.1864],
        [ 1.3596, -0.6411,  2.9097,  0.9422, -0.0167, -0.1453, -0.6059],
        [-0.4946,  0.2705,  0.5348, -1.8176, -1.3861, -1.0276, -1.0050]])
>>> a.sum(axis=1)
tensor([-0.6527,  3.8434, -0.0306, -2.0437,  1.8446,  3.8025, -4.9256])
>>> a.sum(axis=1, keepdim=True)
tensor([[ -0.6527],
        [ 3.8434],
        [-0.0306],
        [-2.0437],
        [ 1.8446],
        [ 3.8025],
        [-4.9256]])
```

## Step 3: A Neural Probabilistic Language Model

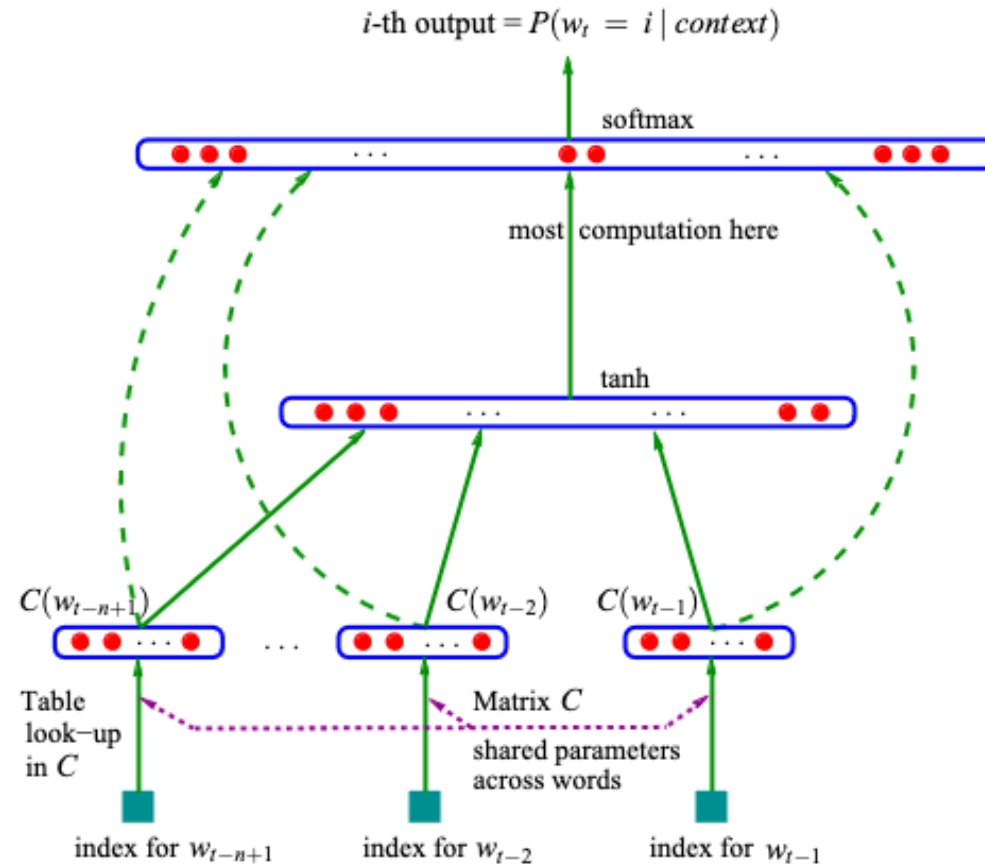
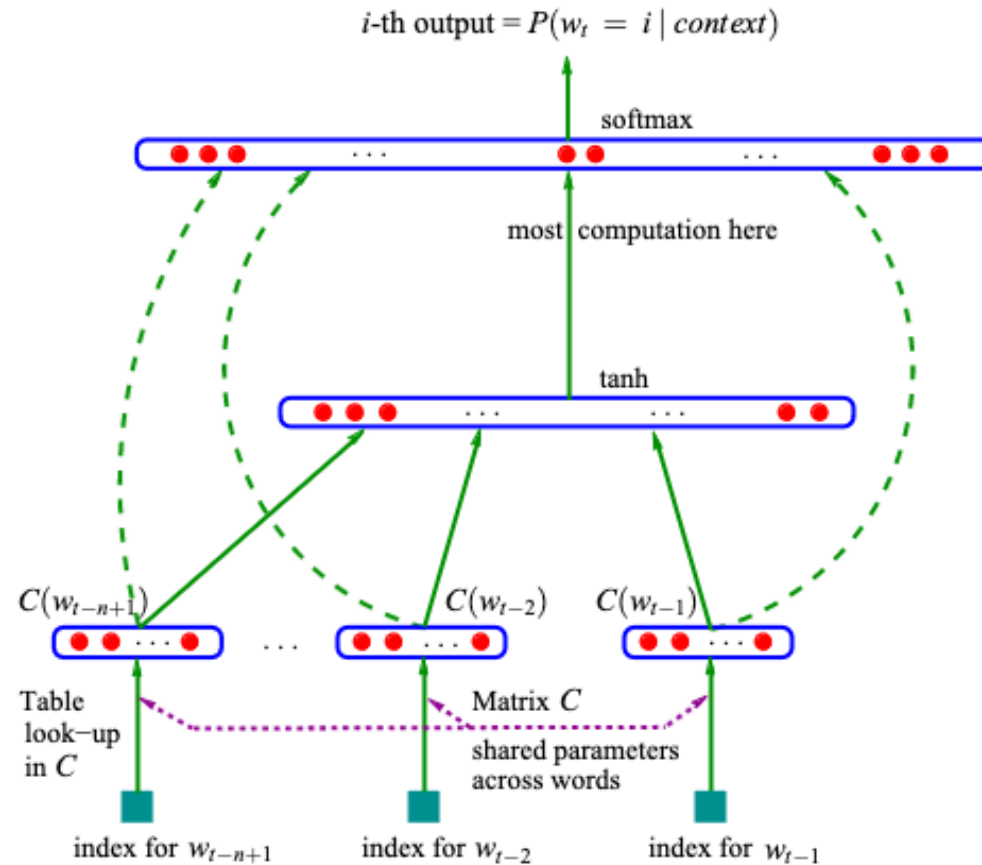


Figure 1: Neural architecture:  $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$  where  $g$  is the neural network and  $C(i)$  is the  $i$ -th word feature vector.

# Step 3: A Neural Probabilistic Language Model

```
X_train = tensor([
  [ 0, 0, 0],
  [ 0, 0, 5],
  [ 0, 5, 2], ...,
  [25, 1, 14],
  [ 1, 14, 14],
  [14, 14, 9]])
```

$C_{27 \times 10}$  = dictionary



$$\text{logits} = \tanh(W_2 \cdot h + b_2)$$

$$h = \tanh(W_1 \cdot \text{emb} + b_1)$$

$\text{emb} = C[X\_train[ix]]$   
 $\text{emb} = \text{emb.view(...)}$

Figure 1: Neural architecture:  $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$  where  $g$  is the neural network and  $C(i)$  is the  $i$ -th word feature vector.