ENSAE

Deep learning

Unpacking Transformers, LLMs and image generation

Session 2

Syllabus

onfido

Session	Lecture	TP
1	Intro to DL Neural networks 101	Intro to micrograd (Value class)
2	DL fundamentals • Backprop • Loss functions • Residual networks • Batch norm, LayerNorm • Initialization • Intro to Bengio et al 2003	Bigram model and MLP for next-character prediction (*) add batchnorm
3	Backprop is a leaky abstraction Attention Transformers	Backprop ninja (*) Fleuret practical 3 (MLP for MNIST)
4	Transformers in the real world (from scratch starting from a bigram)	mini-gpt (training on GPU) Find a good open-source LLM and build a notebook running it on a CPU (*) Fleuret practical 5 and 6
5	DL for computer vision: convnets, unets	Convnets for Fashion MNIST Fleuret practical 4
6	VAE & Diffusion models	1D diffusion model 2D diffusion model (*) train on GPU 1 QCM

Important notes

Notebooks are scored from 0 to 4.

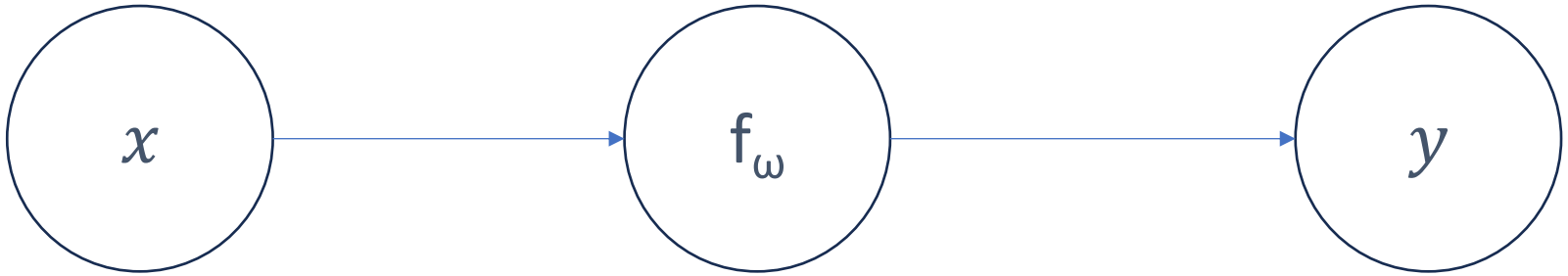
Always submit a notebook after class. Max score = 1/4 otherwise.

Exercises marked as (*) are required to get to 4. Max score = 2 otherwise.

No class on Feb 21st.

No class on March 27th –> last class on April 3rd. Be there!

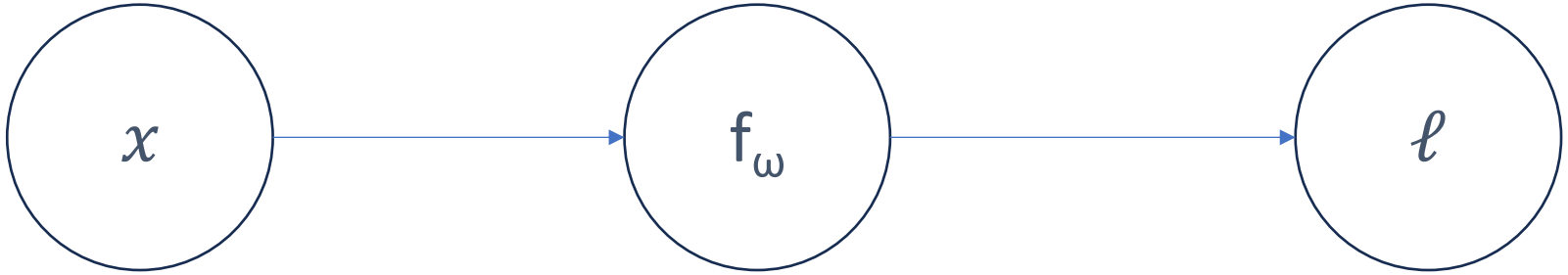
Backprop and gradient descent



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

ground-truth

Backprop and gradient descent



Goal: find ω that minimizes:

$$\ell = f_\omega(x)$$

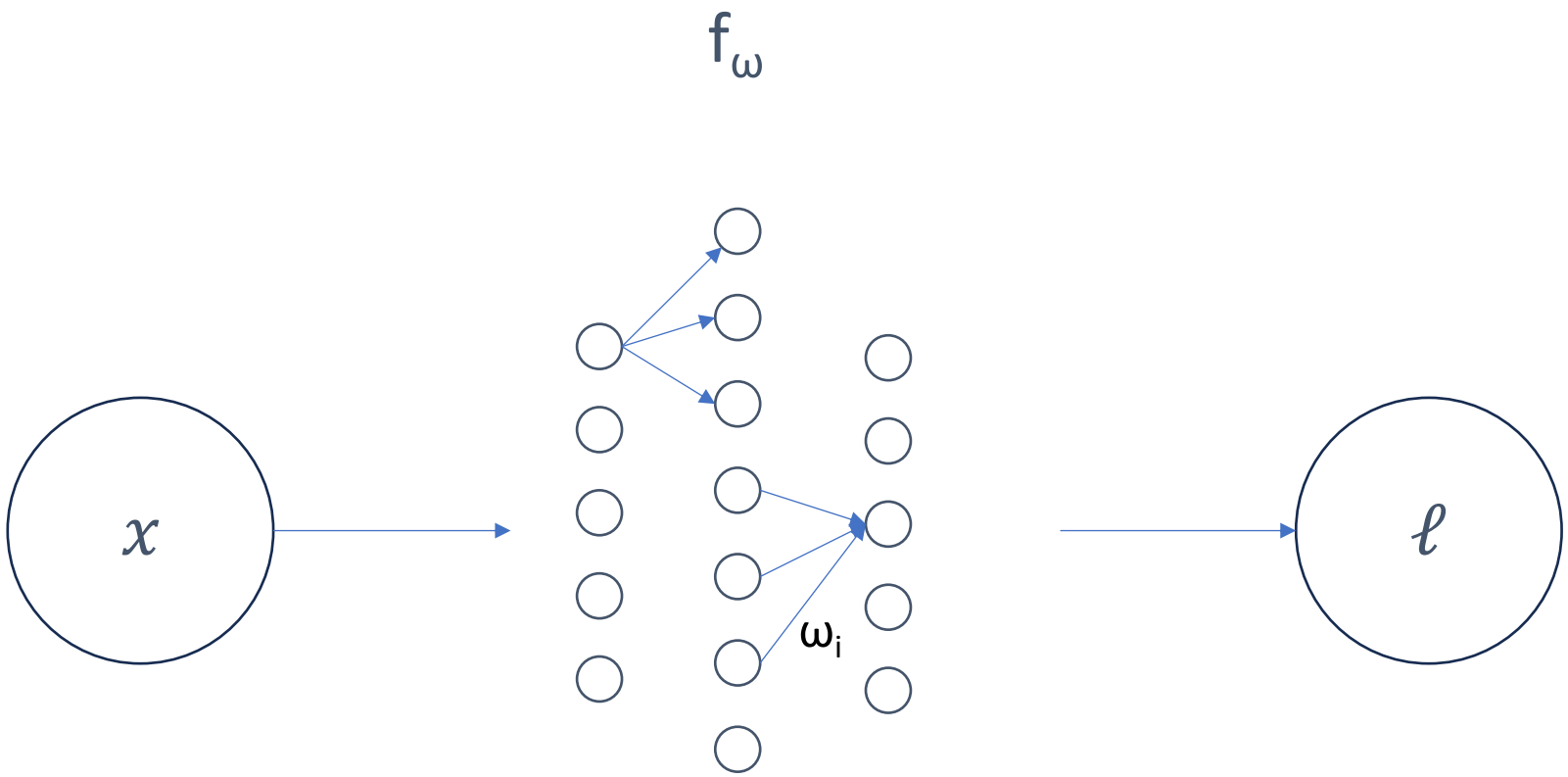
$$\ell = f_\omega(x)$$

Assuming $f_\omega(x)$ is smooth and differentiable w.r.t ω

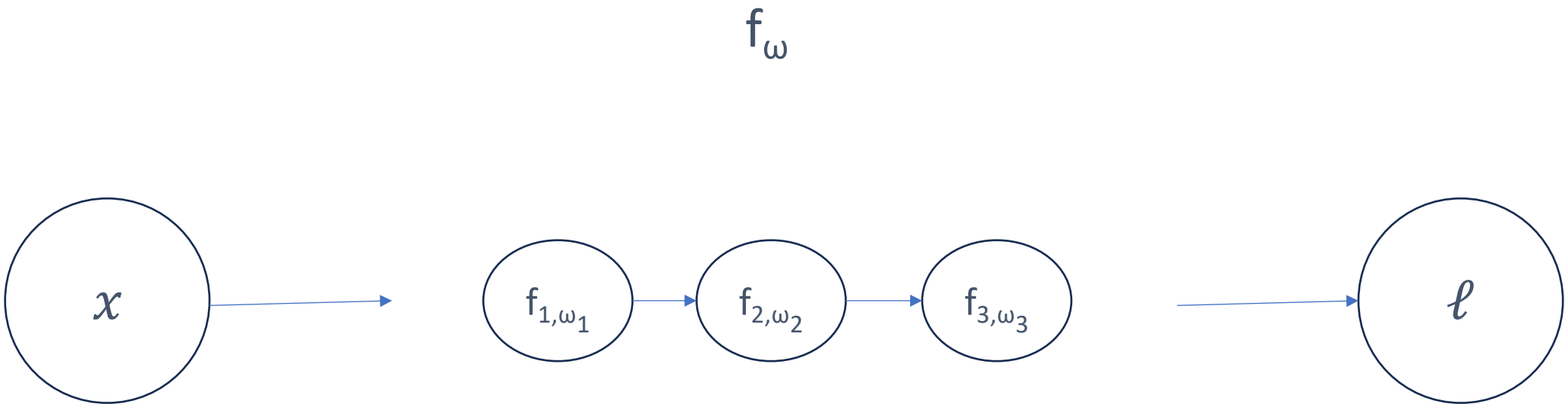
$\omega = \omega_0$
Repeat:
    Compute $\ell(x)$
    Compute ∇f_ω
    $\omega = \omega - \eta * \nabla f_\omega$

η: learning rate

Backprop and gradient descent



Backprop and gradient descent



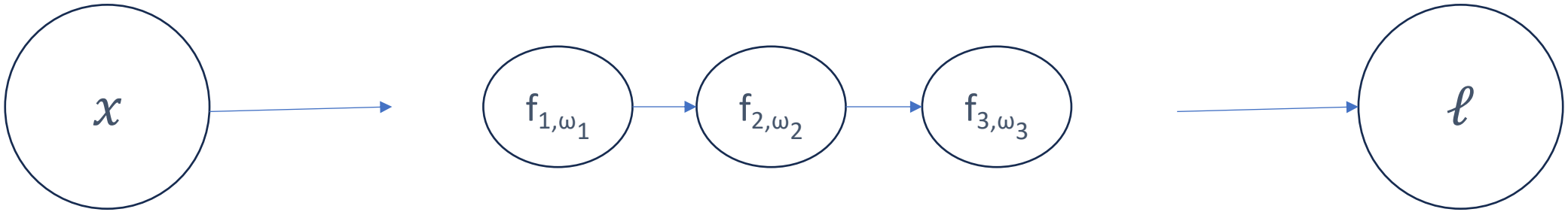
Chain rule and backprop

$$y = f\big(g(x)\big)$$

$$y' = g'(x) * f'\big(g(x)\big)$$

Chain rule and backprop

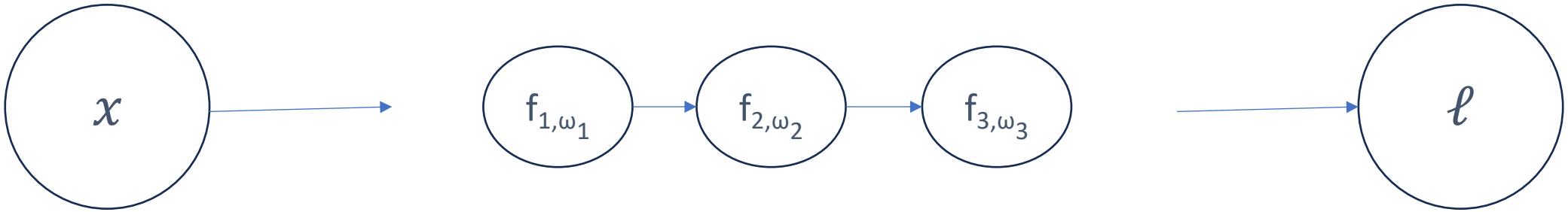
f_ω



$$\frac{\partial \ell}{\partial \omega_1} = \frac{\partial \ell}{\partial f_3} \frac{\partial f_3}{\partial f_2} \frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial \omega_1}$$

Chain rule and backprop

f_ω

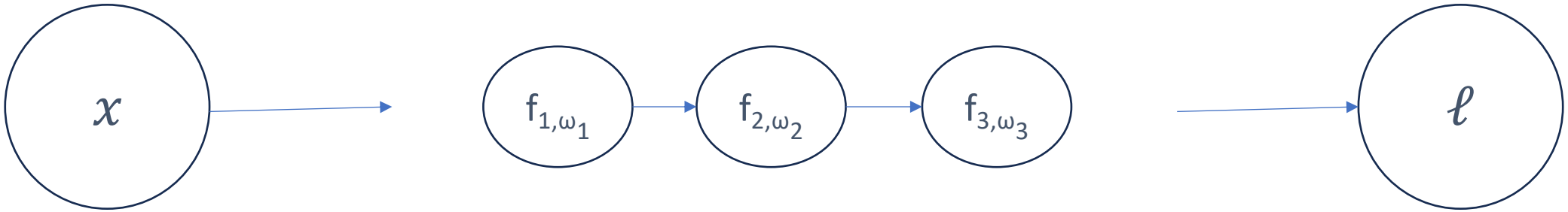


$$\frac{\partial \ell}{\partial \omega_1} = \frac{\partial \ell}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial \omega_1}$$

$$\frac{\partial \ell}{\partial \omega_2} = \frac{\partial \ell}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial \omega_2}$$

Chain rule and backprop

f_ω



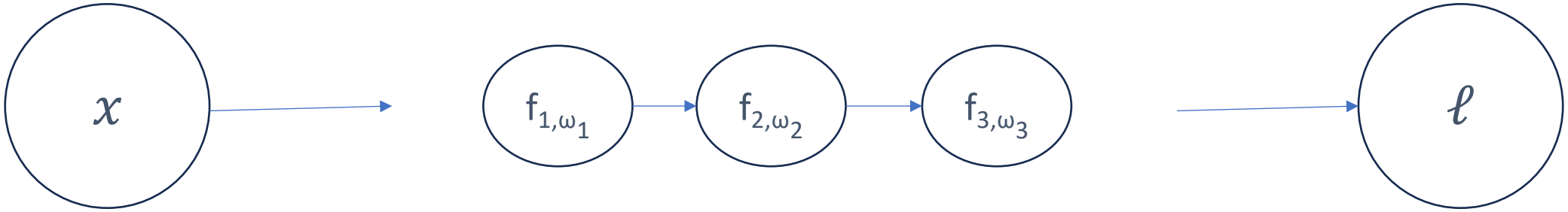
$$\frac{\partial \ell}{\partial \omega_1} = \frac{\partial \ell}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial \omega_1}$$

$$\frac{\partial \ell}{\partial \omega_2} = \frac{\partial \ell}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial \omega_2}$$

Chain rule and backprop

f_ω

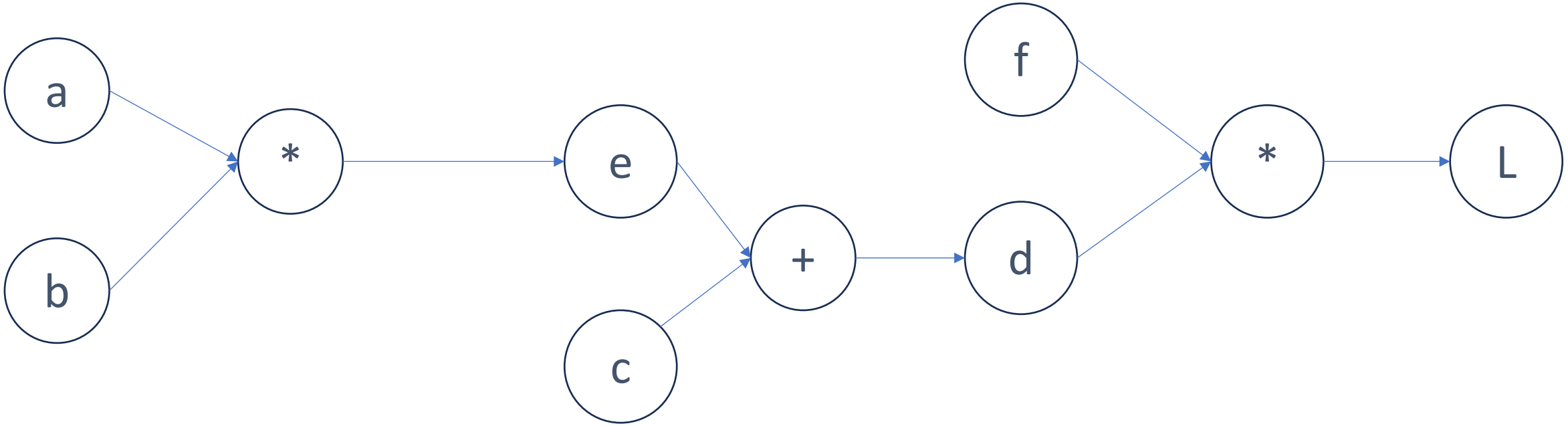
All subfunctions of a neural net are easy to derive



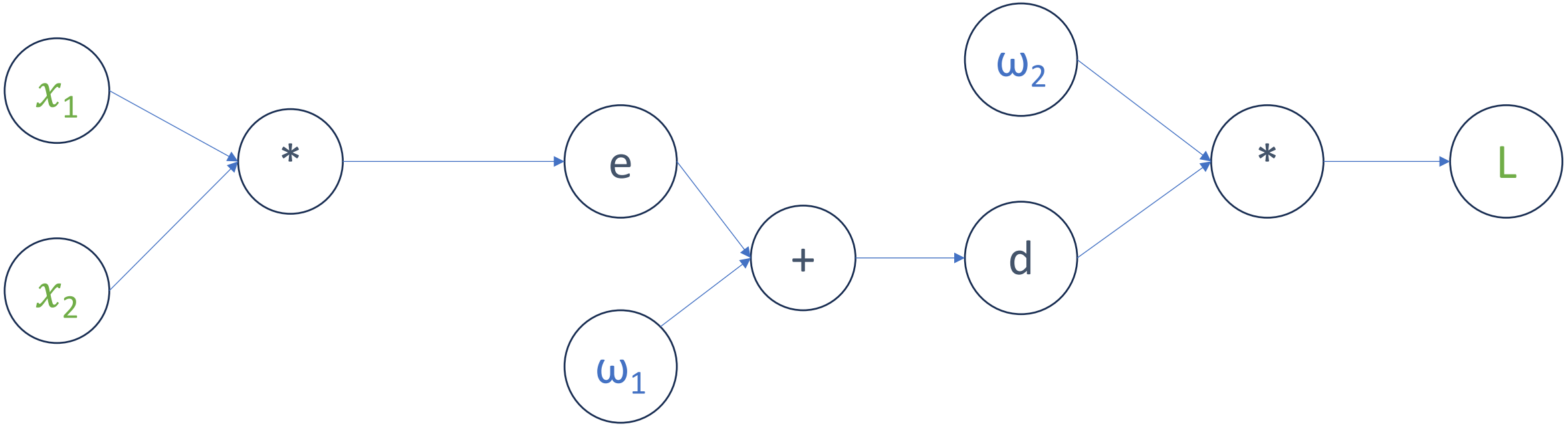
$$\frac{\partial \ell}{\partial \omega_1} = \frac{\partial \ell}{\partial f_3} \frac{\partial f_3}{\partial f_2} \frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial \omega_1}$$

$$\frac{\partial \ell}{\partial \omega_2} = \frac{\partial \ell}{\partial f_3} \frac{\partial f_3}{\partial f_2} \frac{\partial f_2}{\partial \omega_2}$$

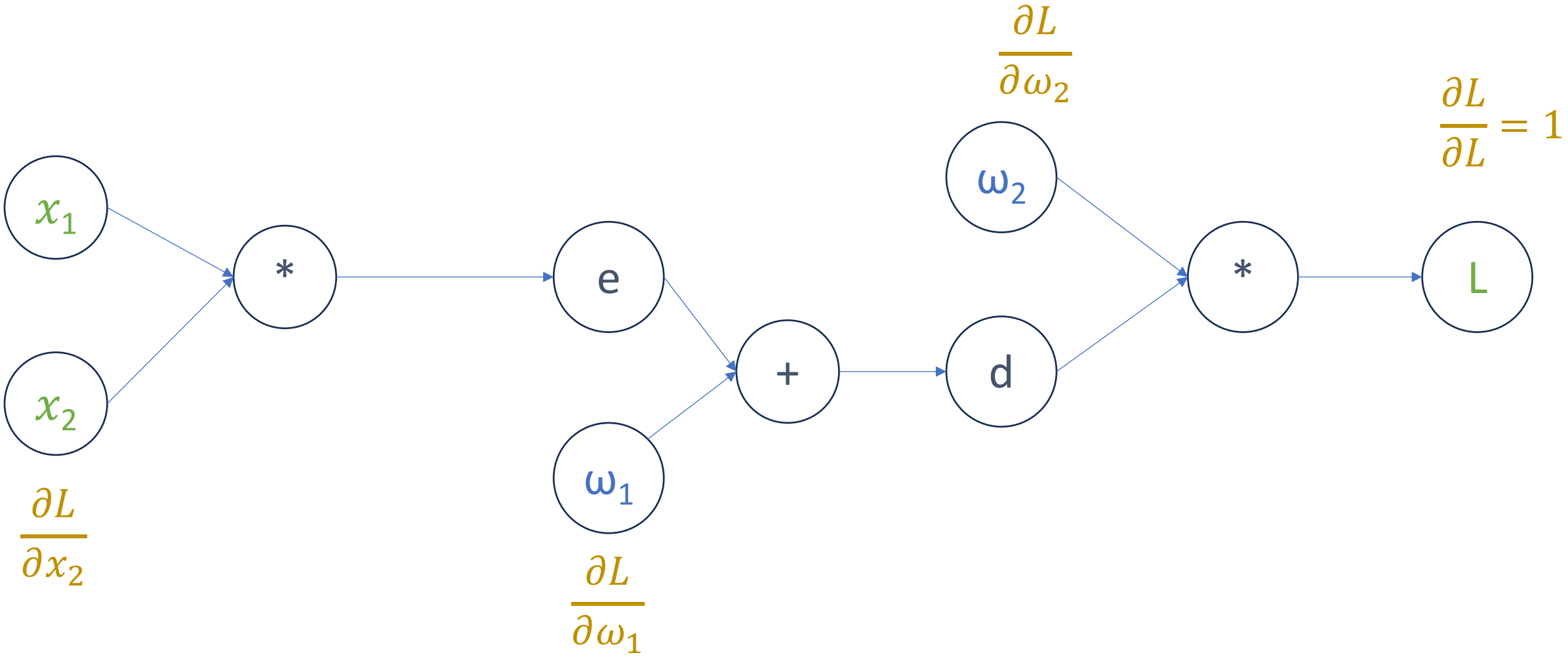
The lol() function



The lol() function



The lol() function

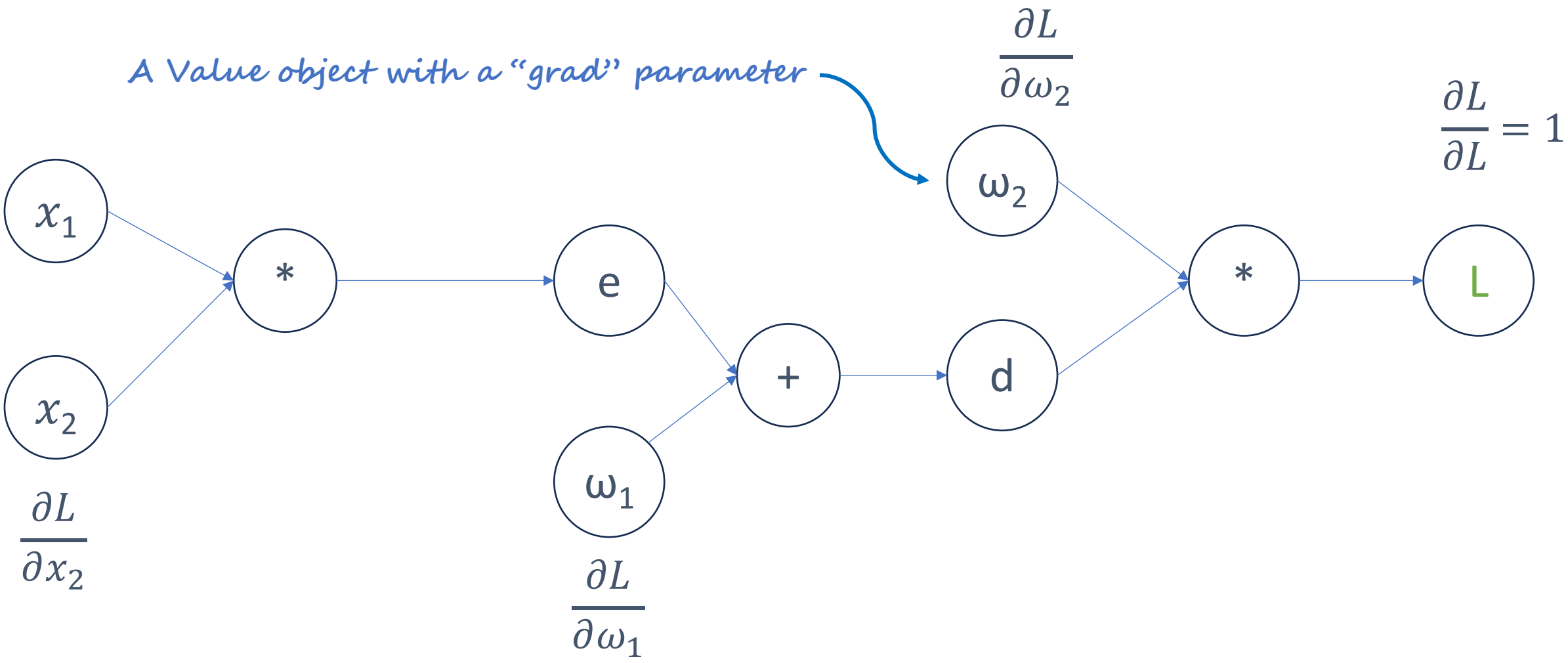


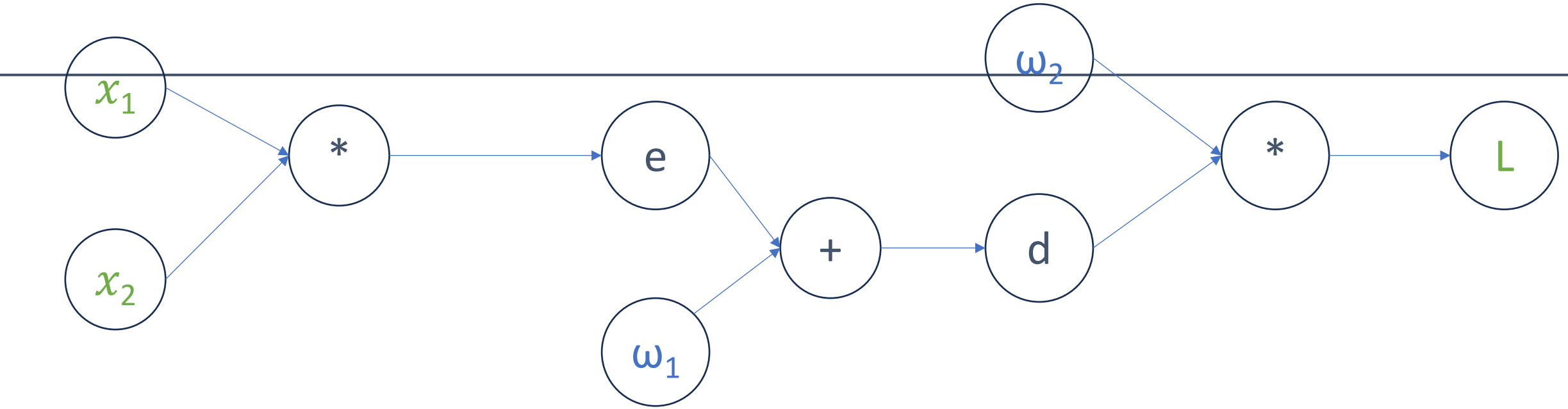
Gradient descent for the lol() function

$$\omega_1 = \omega_1 - \eta \frac{\partial L}{\partial \omega_1}$$

$$\omega_2 = \omega_2 - \eta \frac{\partial L}{\partial \omega_2}$$

The lol() function






```python
class Neuron:

  def __init__(self, nin):
    self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
    self.b = Value(random.uniform(-1,1))

  def __call__(self, x):
    # w * x + b
    act = [redacted]
    out = act.tanh()
    return out

  def parameters(self):
    return self.w + [self.b]
```

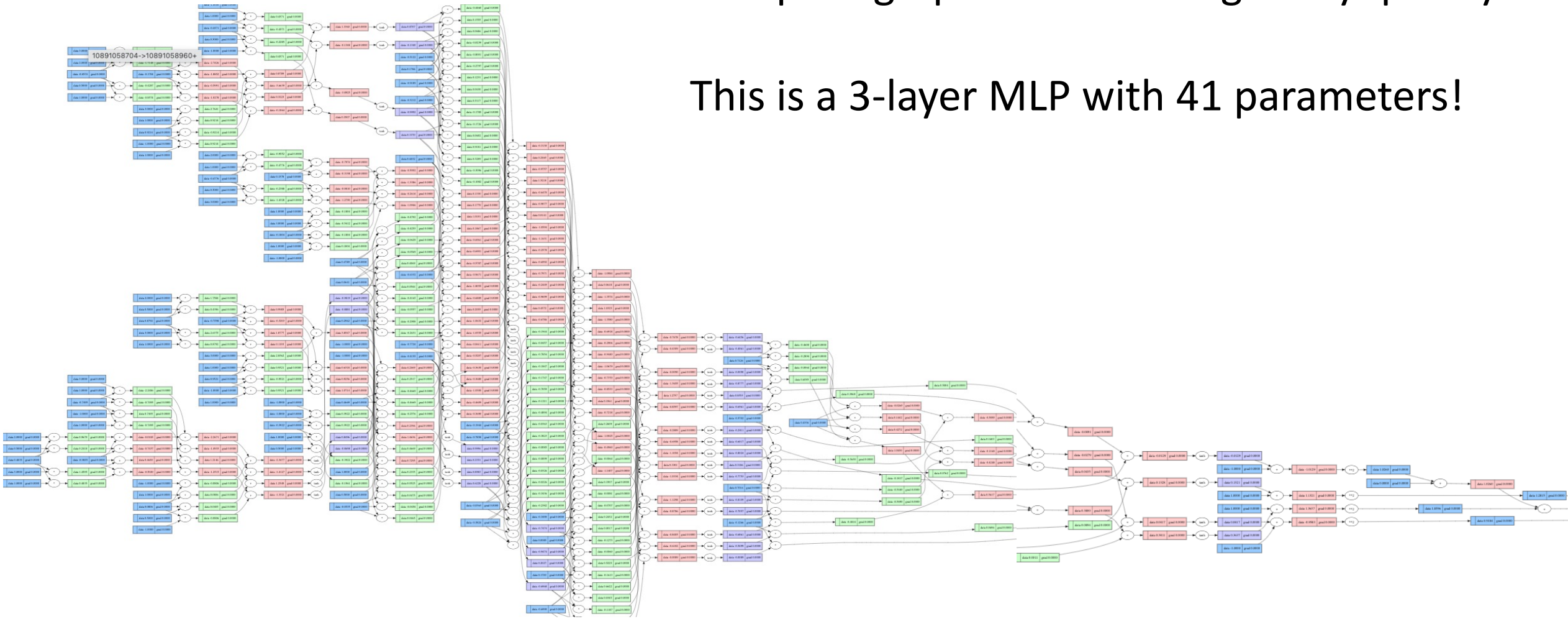
Mean square error loss

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

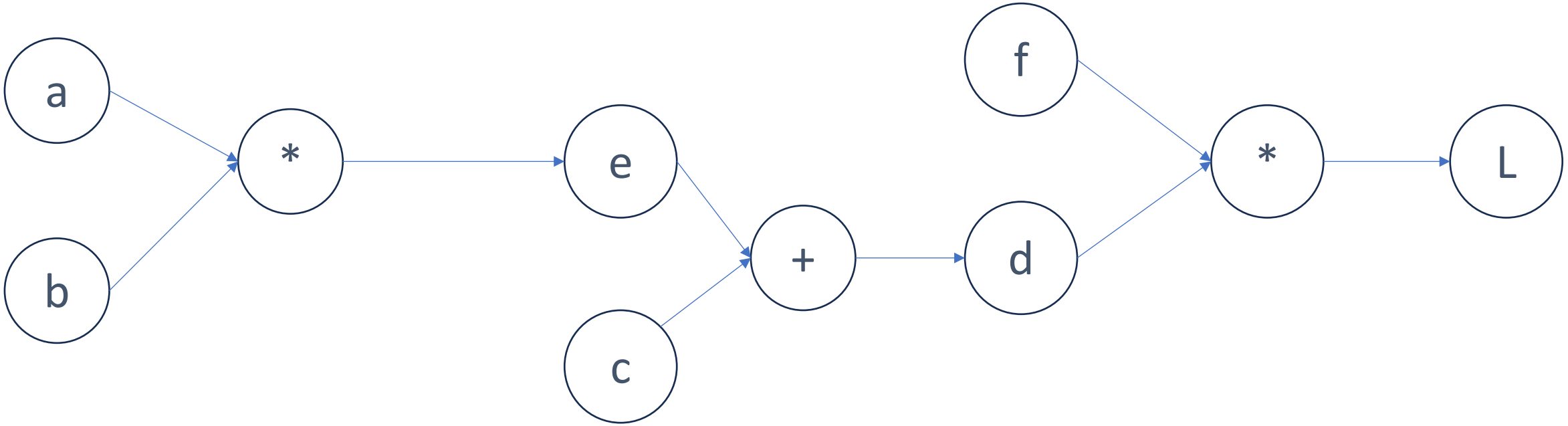
```
n = MLP(3, [4, 4, 1])

ypred = [n(x) for x in xs]

loss = sum([(a-b)**2 for (a,b) in zip(ypred, ys)])
```

Compute graphs become huge very quickly

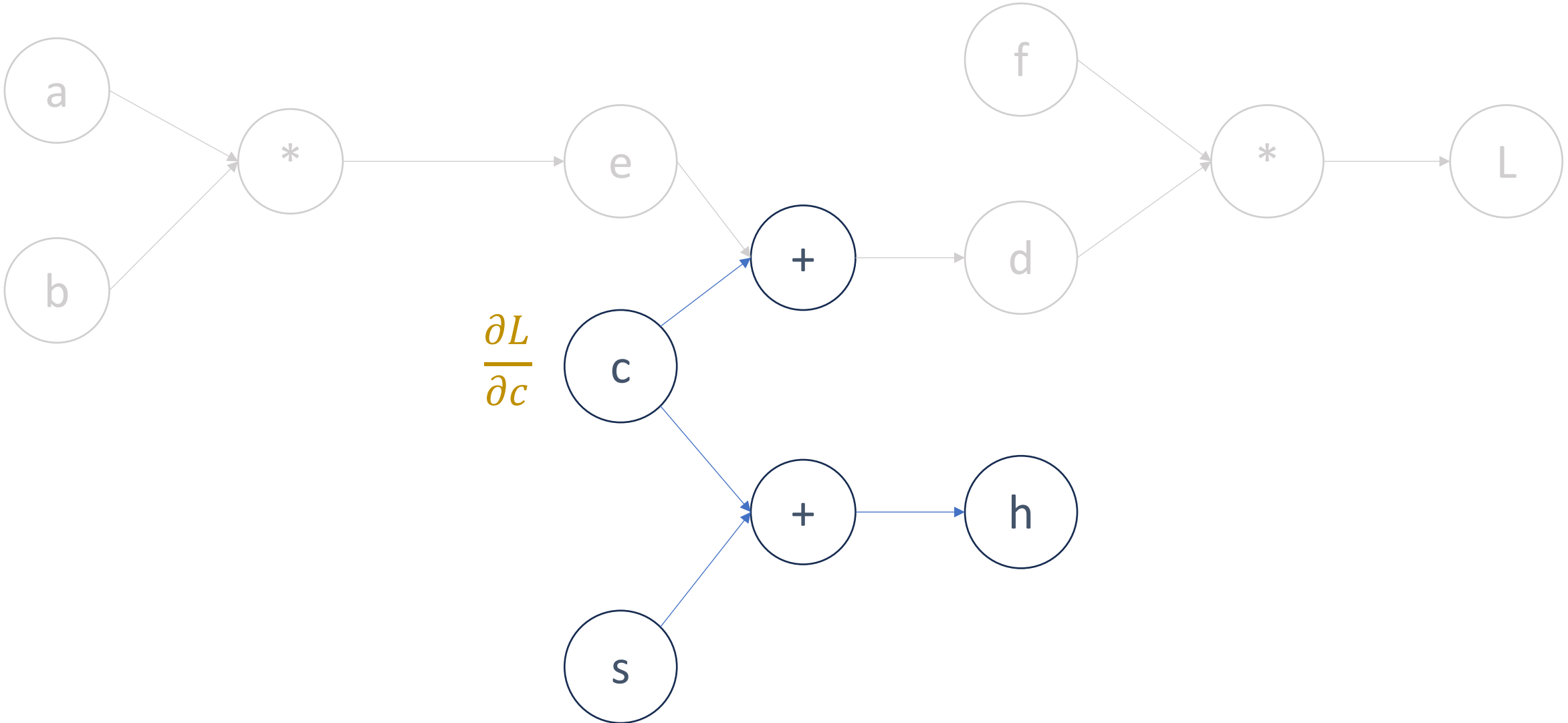
This is a 3-layer MLP with 41 parameters!



Why += in the gradient update?



Why += in the gradient update?



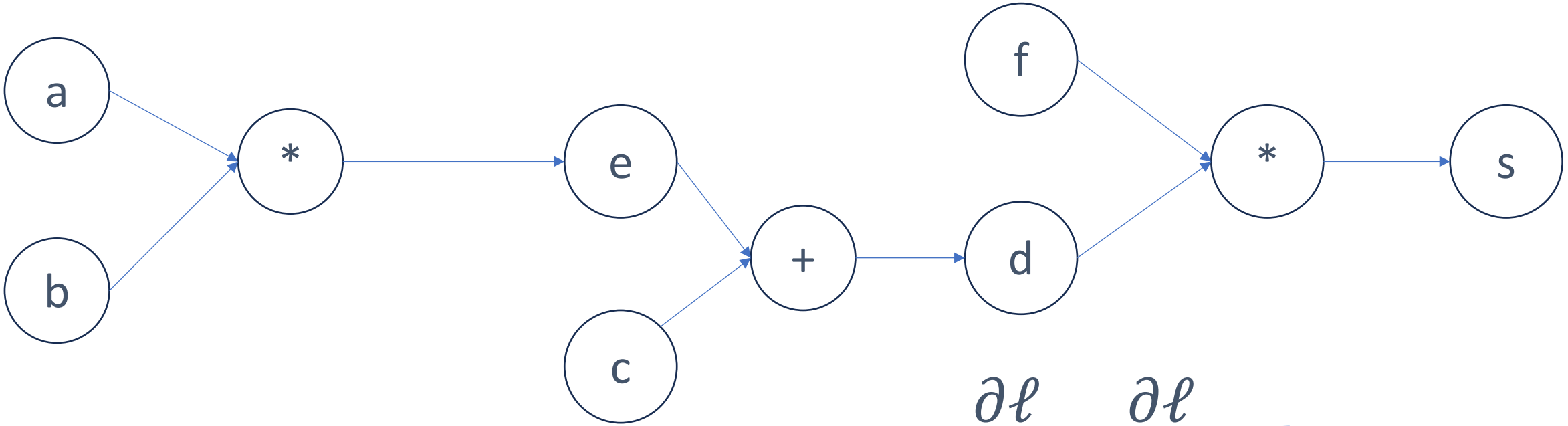
But then you need to zero your gradient at each iteration!

$\omega = \omega_0$

Repeat:

- Compute $\ell(x)$
- Reset $\nabla \ell_\omega$
- Compute $\nabla \ell_\omega$
- $\omega = \omega - \eta * \nabla \ell_\omega$

The forward pass is critical to update the values in the network



$$\frac{\partial \ell}{\partial f} = \frac{\partial \ell}{\partial s} * d$$

Stochastic gradient descent

Iterate over batches of samples instead of the whole dataset at once

1. Scales to large datasets (that don’t fit in memory)
2. Adds randomness to the process

Now you should be able to finish TP1!

There is no reason for deep learning to work

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

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

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

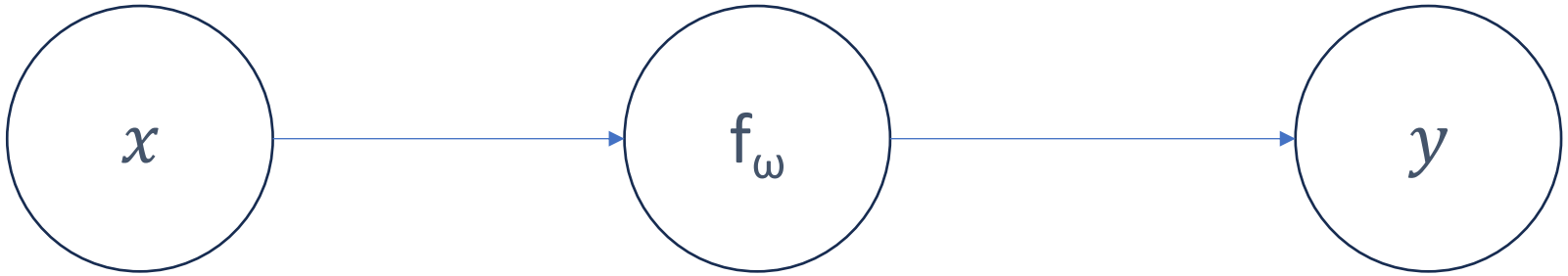
Regularization

Initialization

Residual networks

Batch norm, layer norm

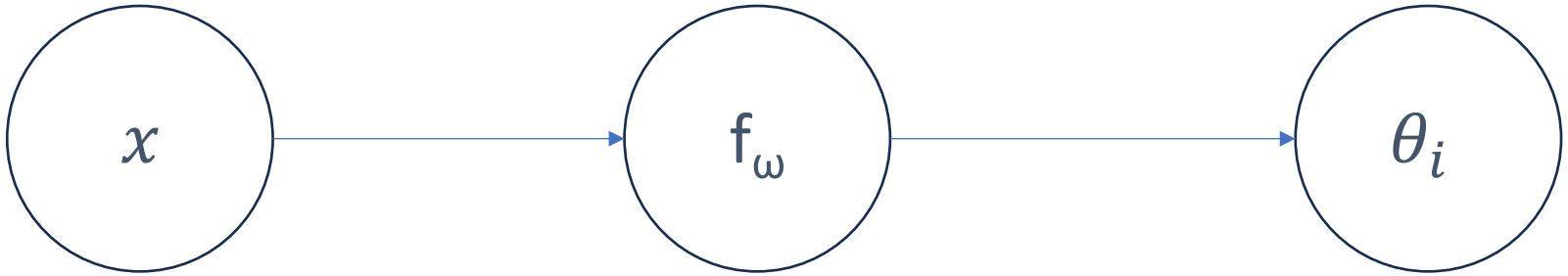
Loss functions



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

ground-truth

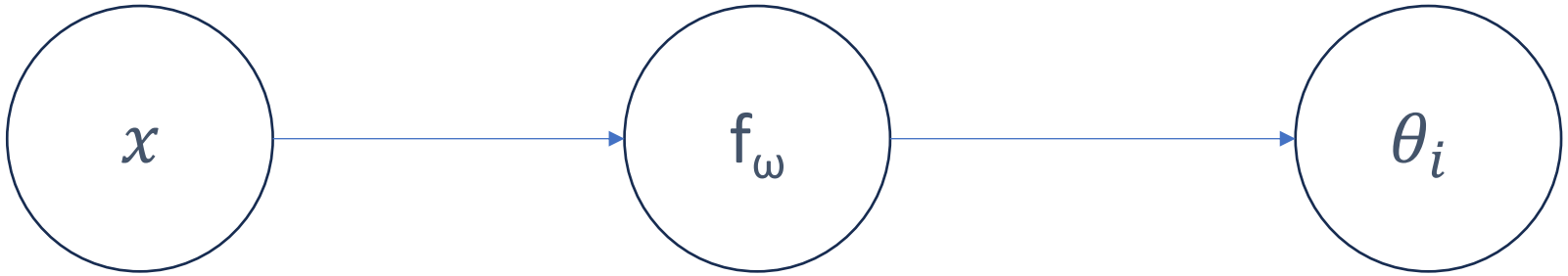
Loss functions



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

Parameters of a distribution

Loss functions



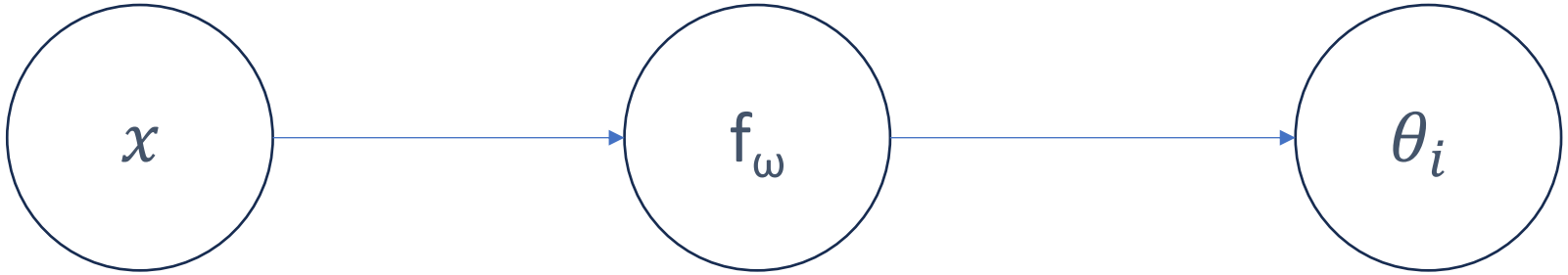
$$f_\omega(x_i) = \theta_i$$

The distribution is chosen based on the domain.

The model computes the optimal θ_i given the data.

Loss functions

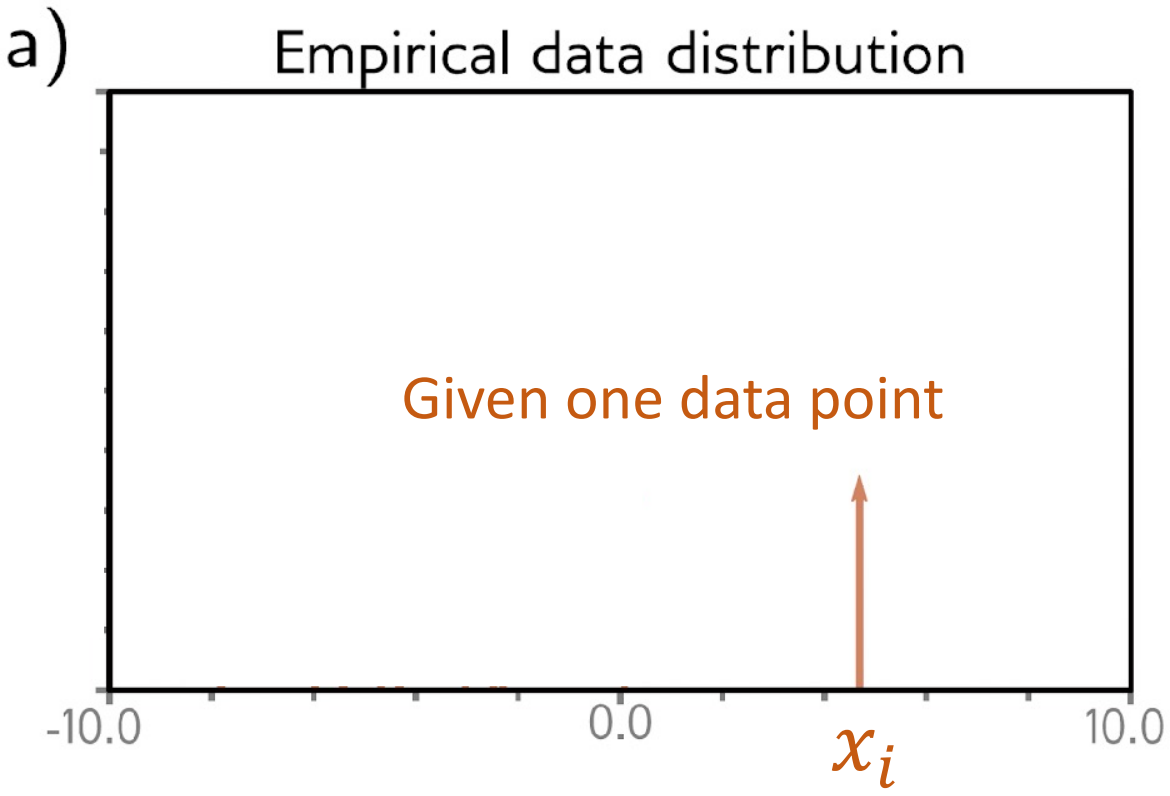
$$f_\omega(x_i) = \theta_i$$

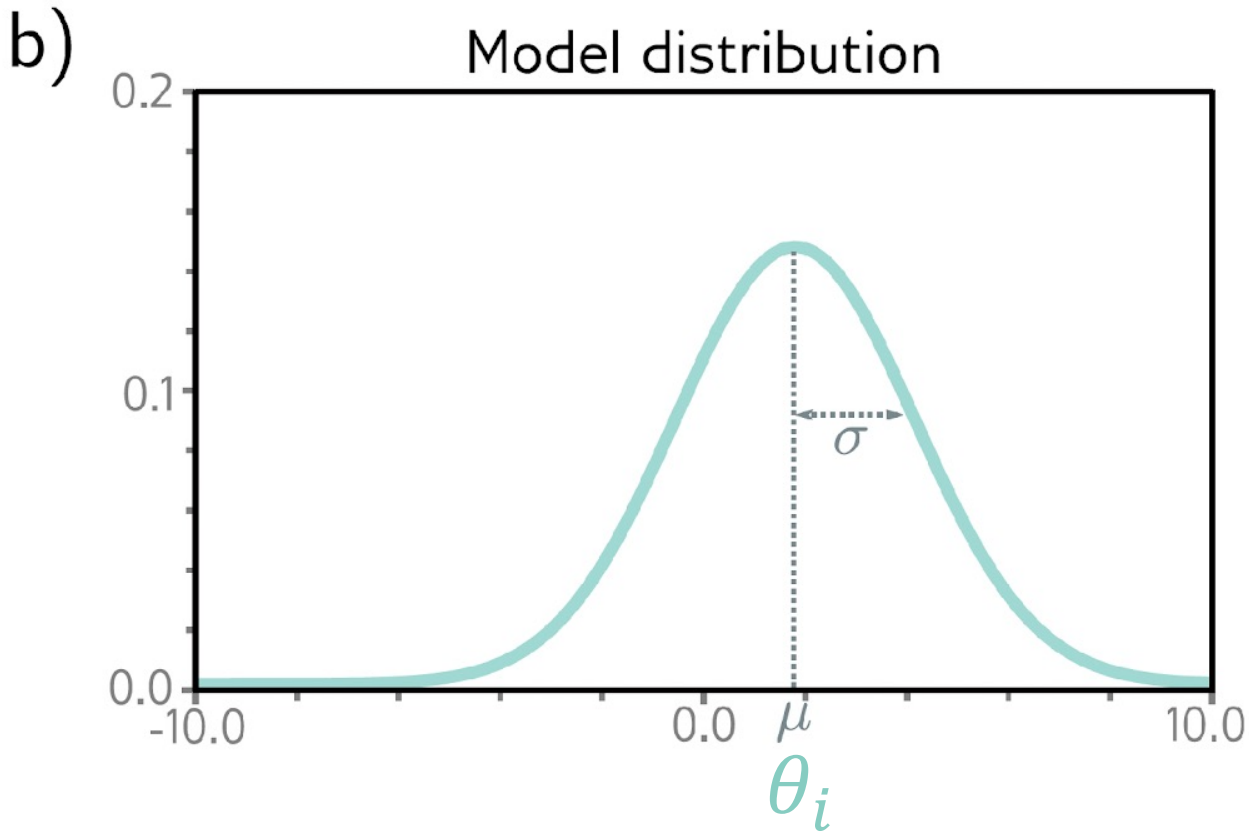


Example: univariate regression

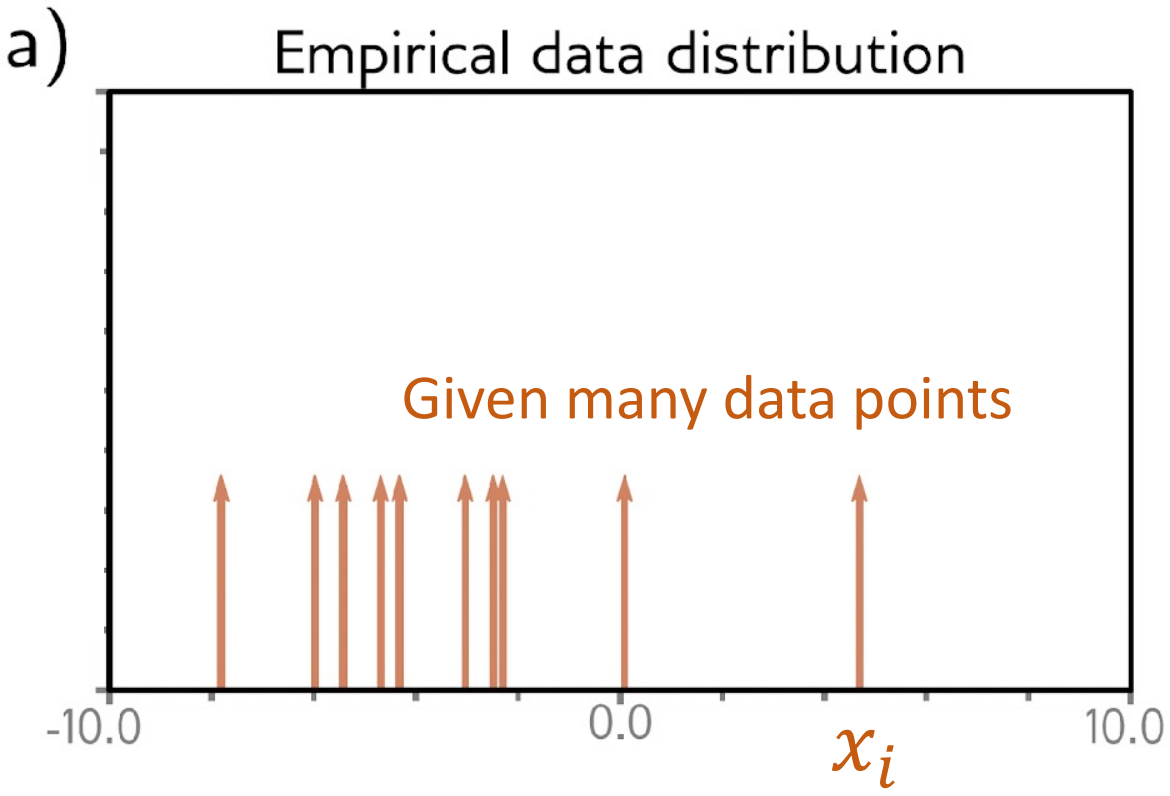
$$\Pr(y|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

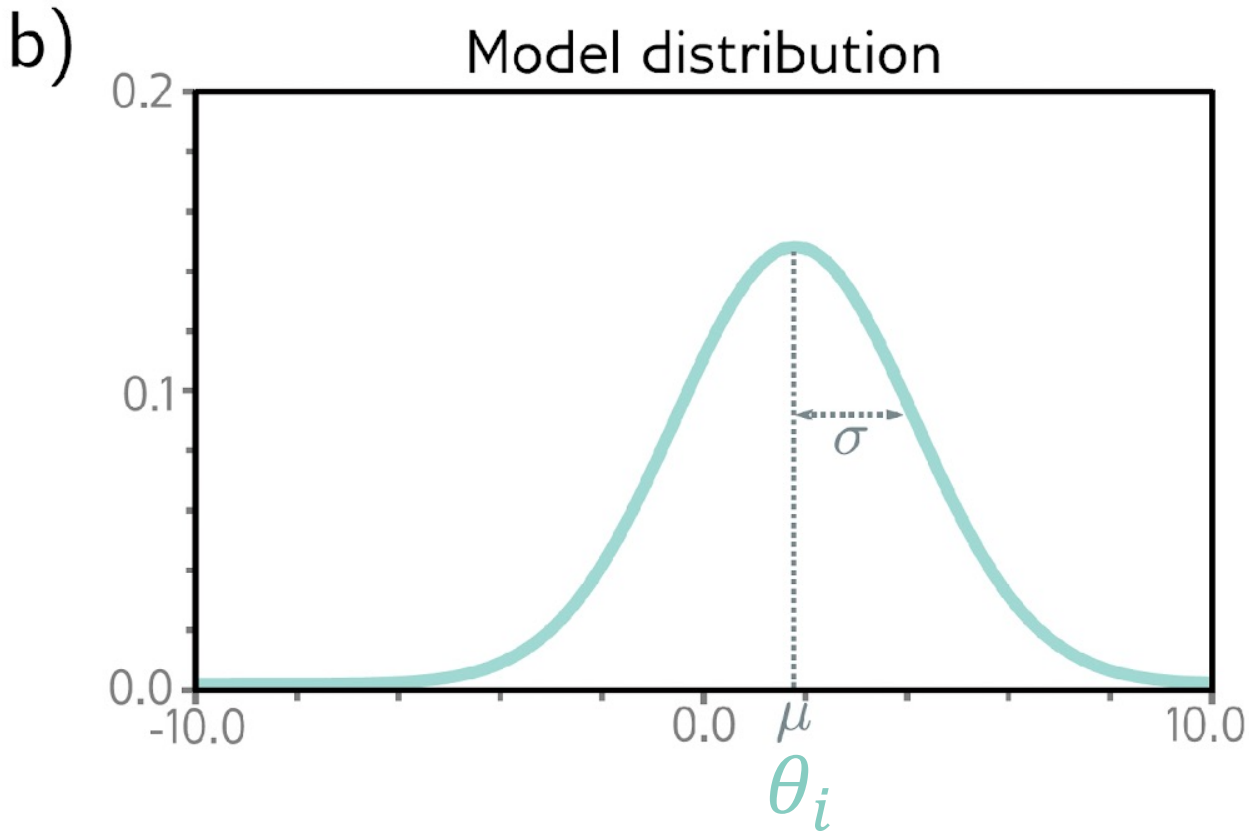
θ





Predict the corresponding distribution parameter





Predict the corresponding distribution parameters

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

Loss functions

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

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

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

Loss functions

$$\Pr(y_1, y_2, \ldots, y_N | x_1, x_2, \ldots, x_N)$$

Data is assumed i.i.d

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

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

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

Loss functions

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

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

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

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

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

Negative log likelihood (NLL)

Inference

Given new input data

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

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.

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

Negative log likelihood (NLL)

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

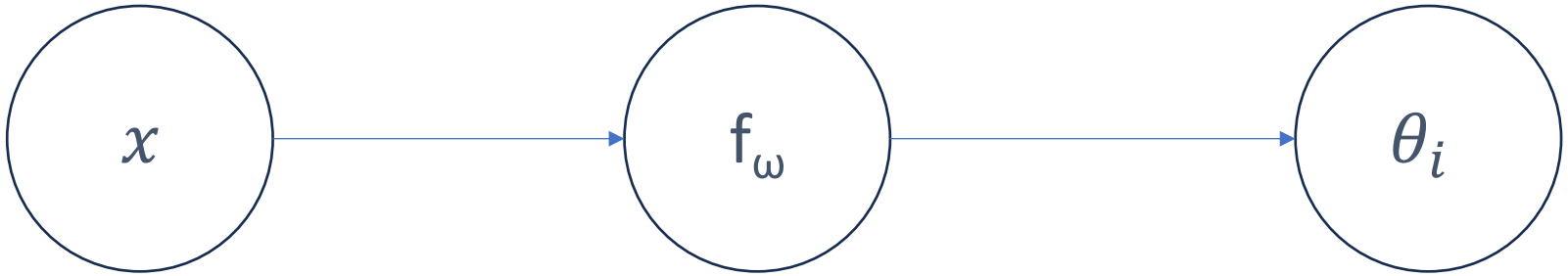
$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}} \left[-\sum_{i=1}^{N} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i-f_\omega(x_i))^2}{2\sigma^2}} \right] \right]$$

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

least squares

Loss functions

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



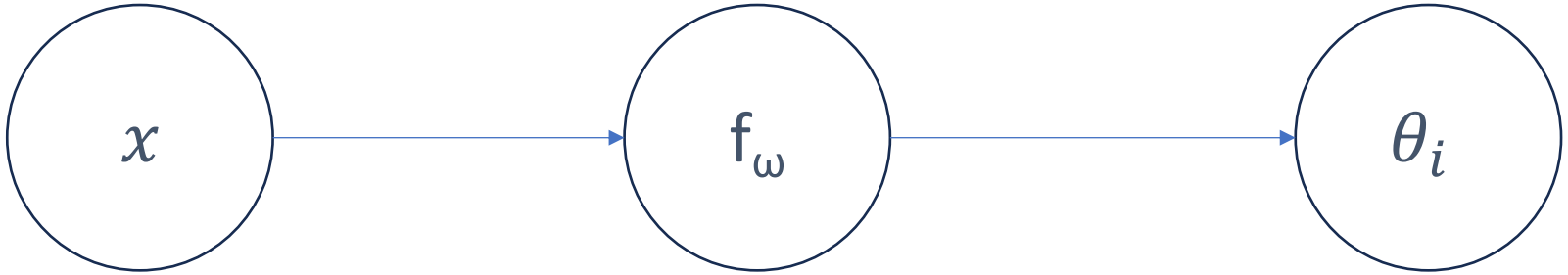
Example: binary classification

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

θ

Loss functions

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



Example: binary classification

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

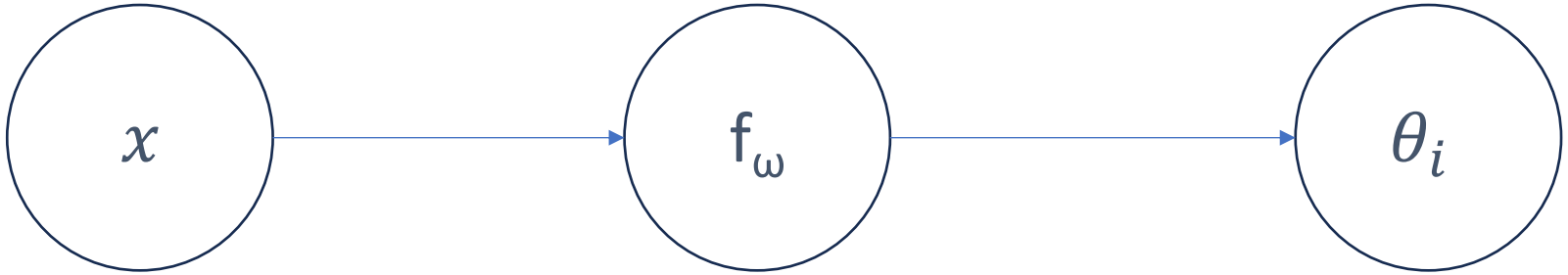
NLL

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

σ: sigmoid function

Loss functions

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



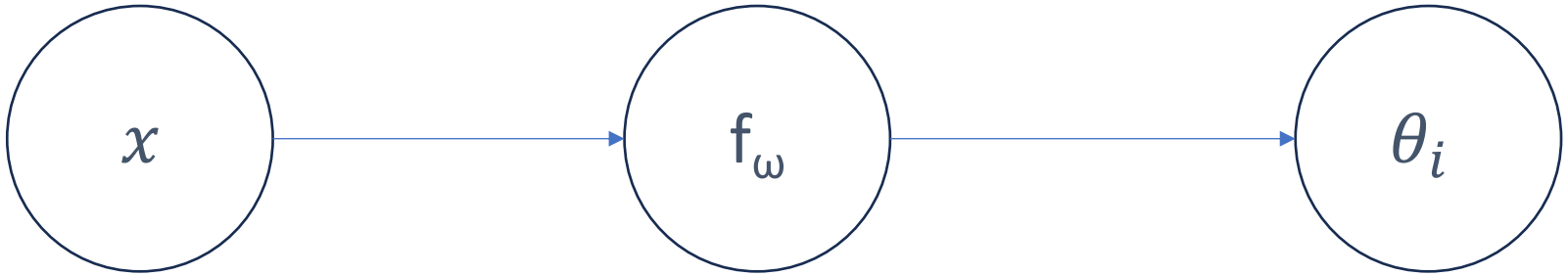
Example: multiclass classification

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

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

Loss functions

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



Example: multiclass classification

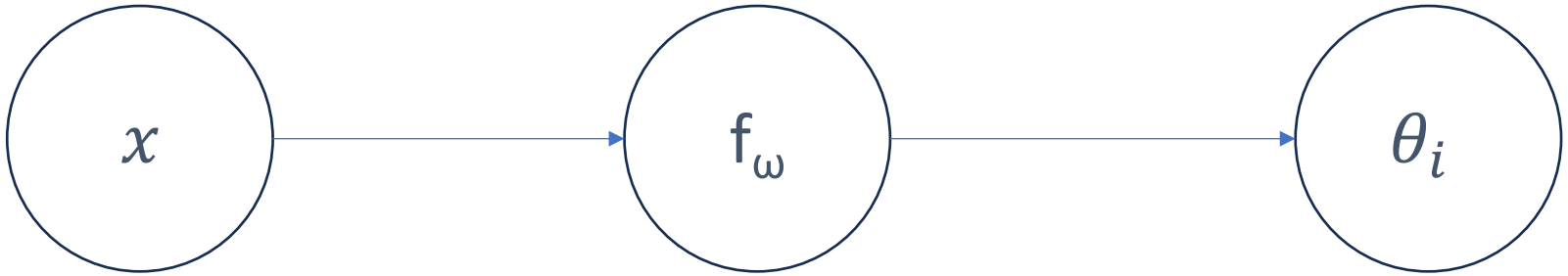
$$\Pr(y = k) = \lambda_k \qquad \sum \lambda_k = 1$$

NLL

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

Loss functions

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



Example: multiclass classification

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

Wait, can I differentiate softmax?

NLL

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

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

Loss functions

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

Negative log likelihood (NLL)

is equivalent to the cross-entropy loss

Loss functions

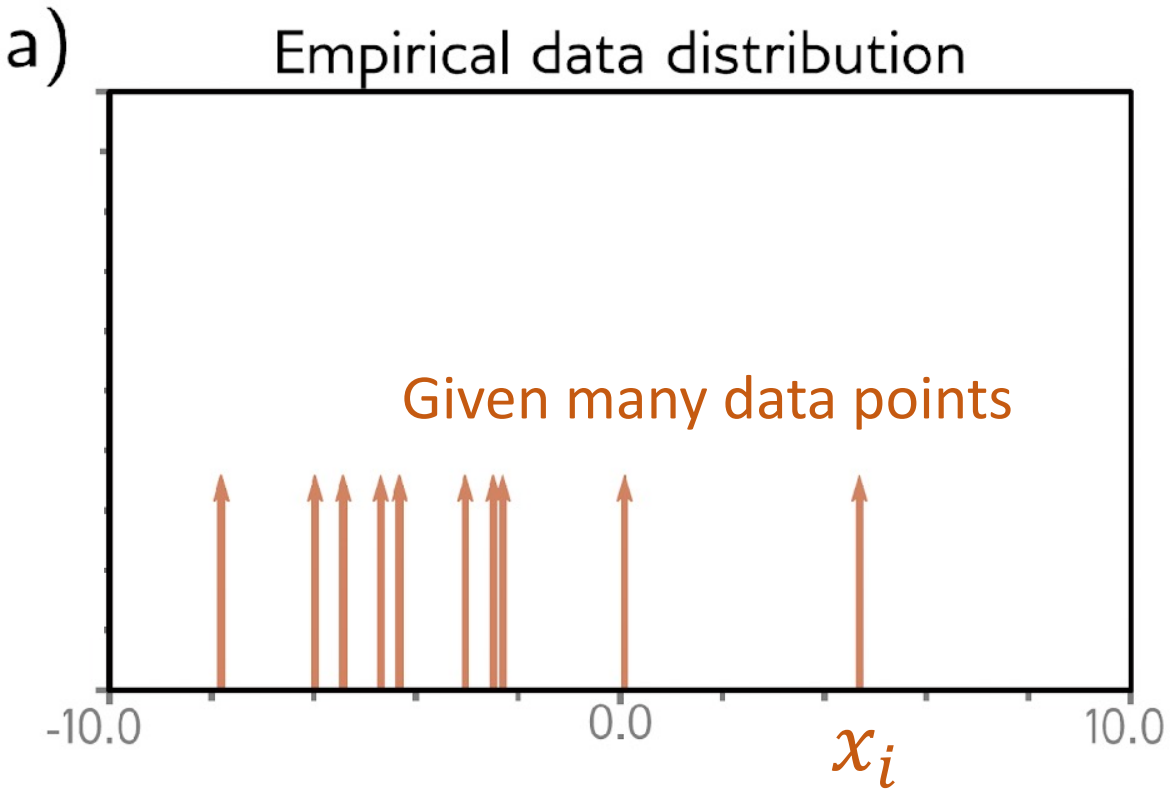
Given two distributions $q(z)$ and p(z), the distance between the two distributions can be computed with:

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

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

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

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



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

δ: dirac

Loss functions

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

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

cross-entry loss

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

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

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

NLL

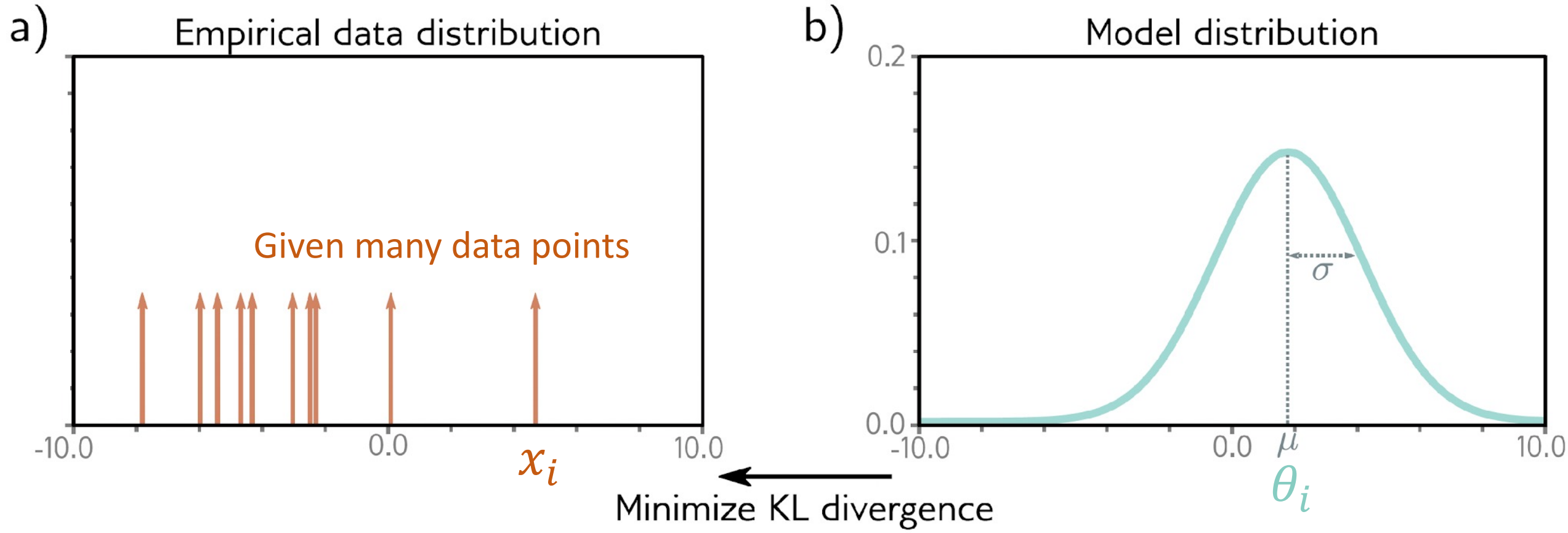
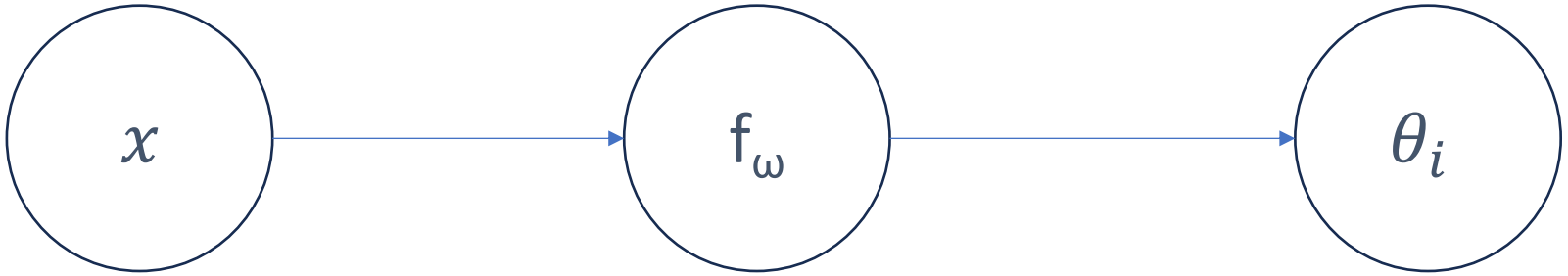


Figure 5.12 Cross-entropy method. a) Empirical distribution of training samples (arrows denote Dirac delta functions). b) Model distribution (a normal distribution with parameters $\boldsymbol{\theta} = \mu, \sigma^2$). In the cross-entropy approach, we minimize the distance (KL divergence) between these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

Example: predict the next character in a set of words (TP2)

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



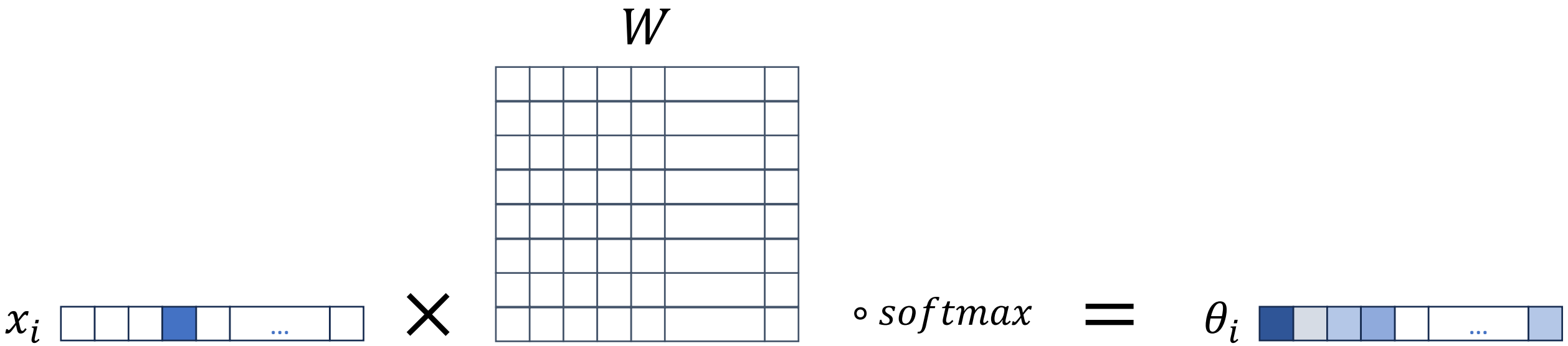
$$x_i = [0,0,0,1,0, \dots 0]$$

One-hot encoding of letter 'd'

ω is a matrix $W_{27\times27}$ such that

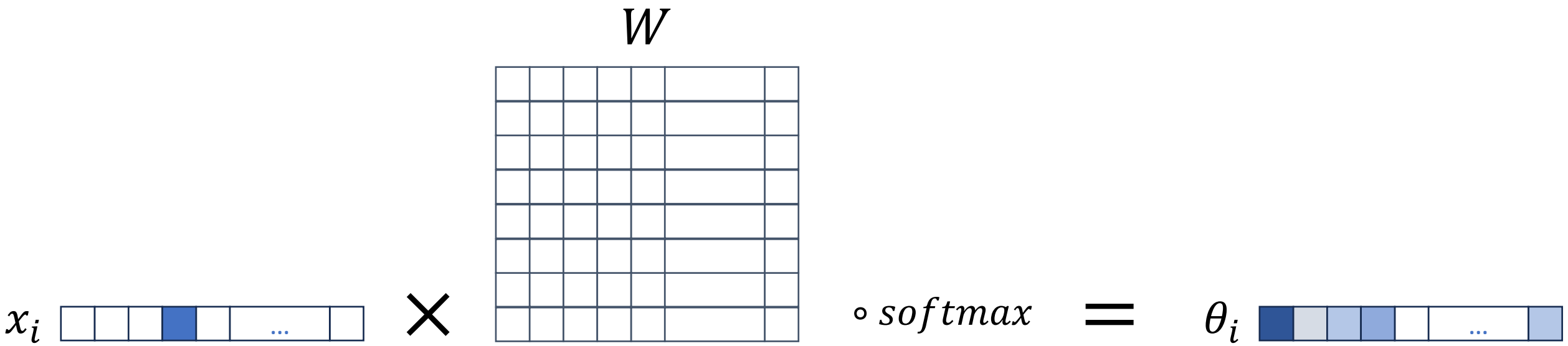
$\theta_i = softmax(x \cdot W)$ is a $N\times27$ vector representing the distribution of the next character for each sample

Example: predict the next character in a set of words



One-hot encoding of letter 'd'

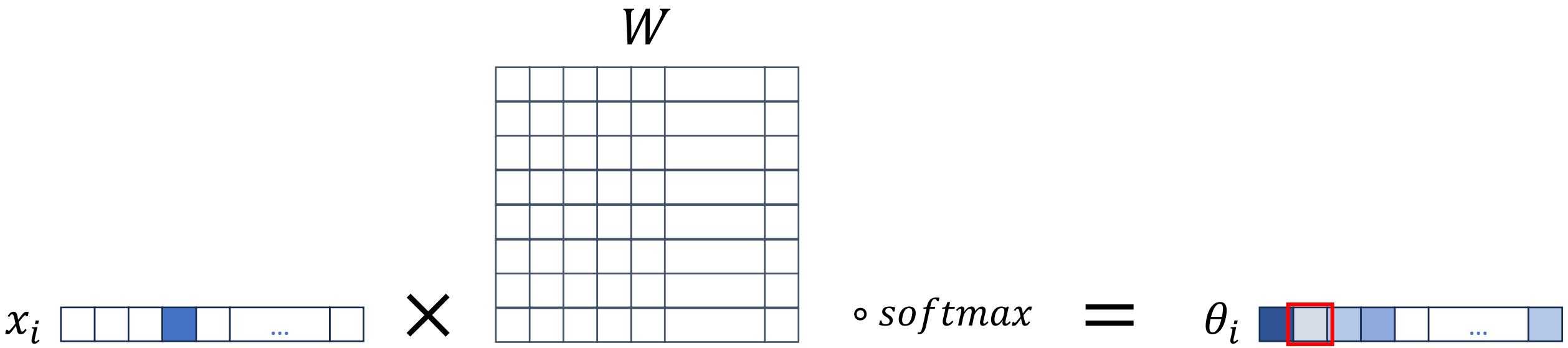
Example: predict the next character in a set of words



We want to minimize the KL divergence or NLL

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

Example: predict the next character in a set of words

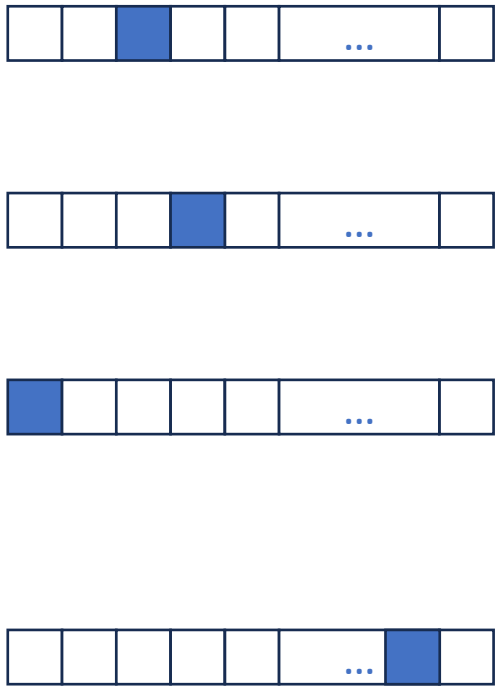


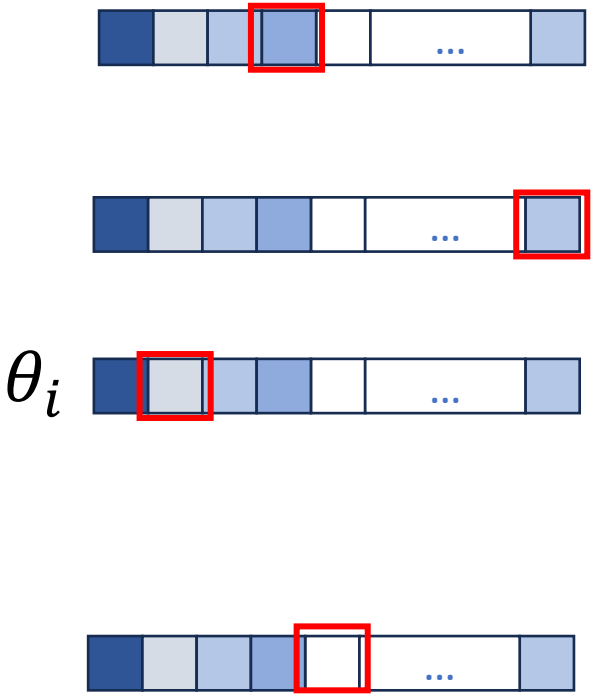
We want to minimize the KL divergence or NLL

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

NLL

Example: predict the next character in a set of words





Loss = mean of log of the red values

Example: predict the next character in a set of words

```
#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 $W_{27\times27}$ 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]])
```

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

Activation functions

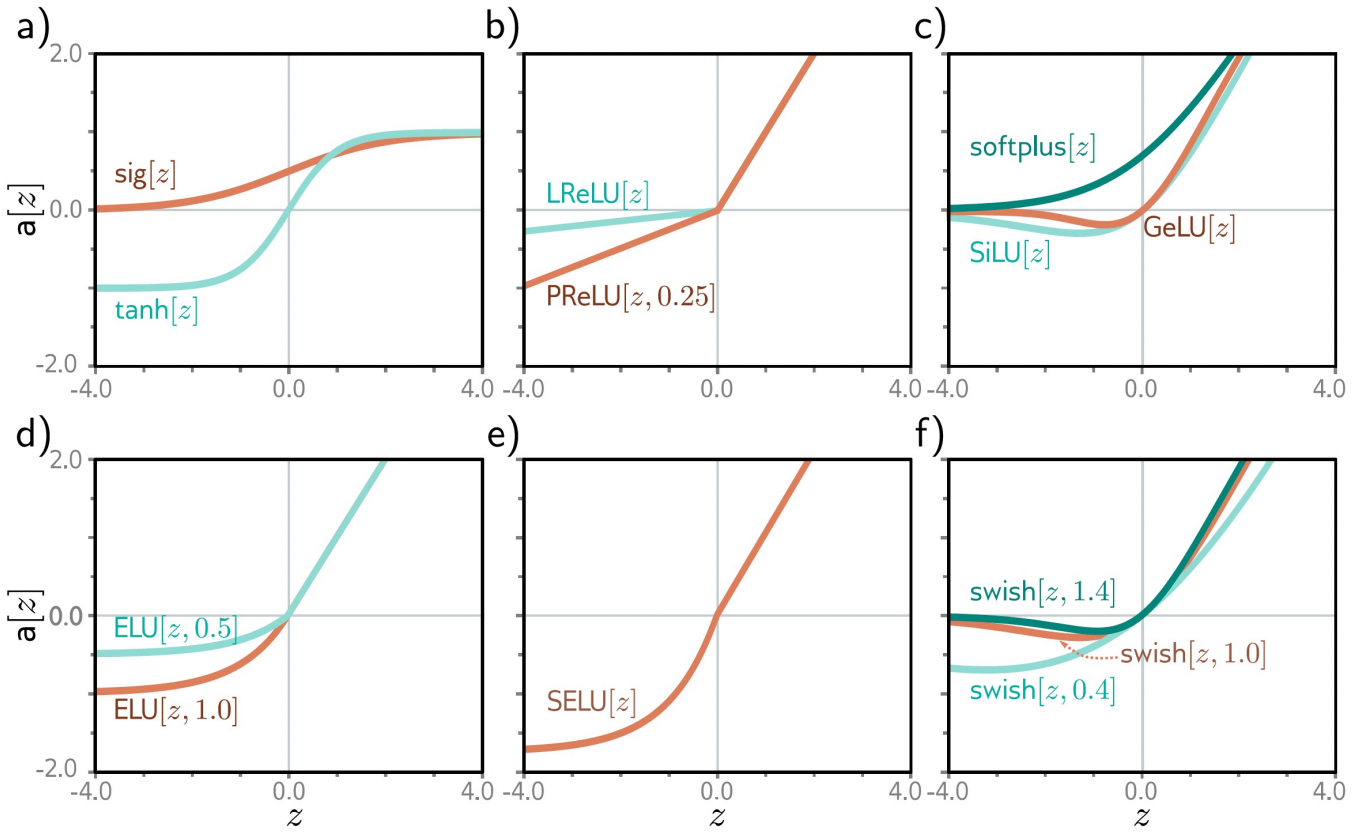


Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

Source: Understanding Deep Learning, S. Prince, 2023

Activation functions

Sigmoid: not blowing up activation

ReLU : not vanishing gradient

ReLU : More computationally efficient to compute than Sigmoid like functions since ReLU just needs to pick max(0,x) and not perform expensive exponential operations as in sigmoids

ReLU : In practice, networks with ReLU tend to show better convergence performance than sigmoid. (Krizhevsky et al.)

By default, use ReLU

ImageNet Classification with Deep Convolutional
Neural Networks, Krizhevsky, Sutskever and Hinton, NIPS 2012

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

Regularization

Regularization is about reducing the generalization gap between training and testing performance.

Implicit regularization is baked into SGD.

Explicit regularization is added through various methods (penalty term, data augmentation, dropout, etc.)

Explicit regularization: adding a penalty term to the loss function

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

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

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

$$\hat{\omega} = \underset{\omega}{\text{argmin}} \left[- \sum_{i=1}^{N} \ell_i[x_i, y_i] + \lambda \cdot g(\omega) \right]$$

penalty term

Explicit regularization: adding a penalty term to the loss function

$$\hat{\omega} = \underset{\omega}{\operatorname{argmin}} \left[-\sum_{i=1}^{N} \ell_i[x_i, y_i] + \lambda \cdot \sum \omega_j^2 \right]$$

L2 loss

Regularization

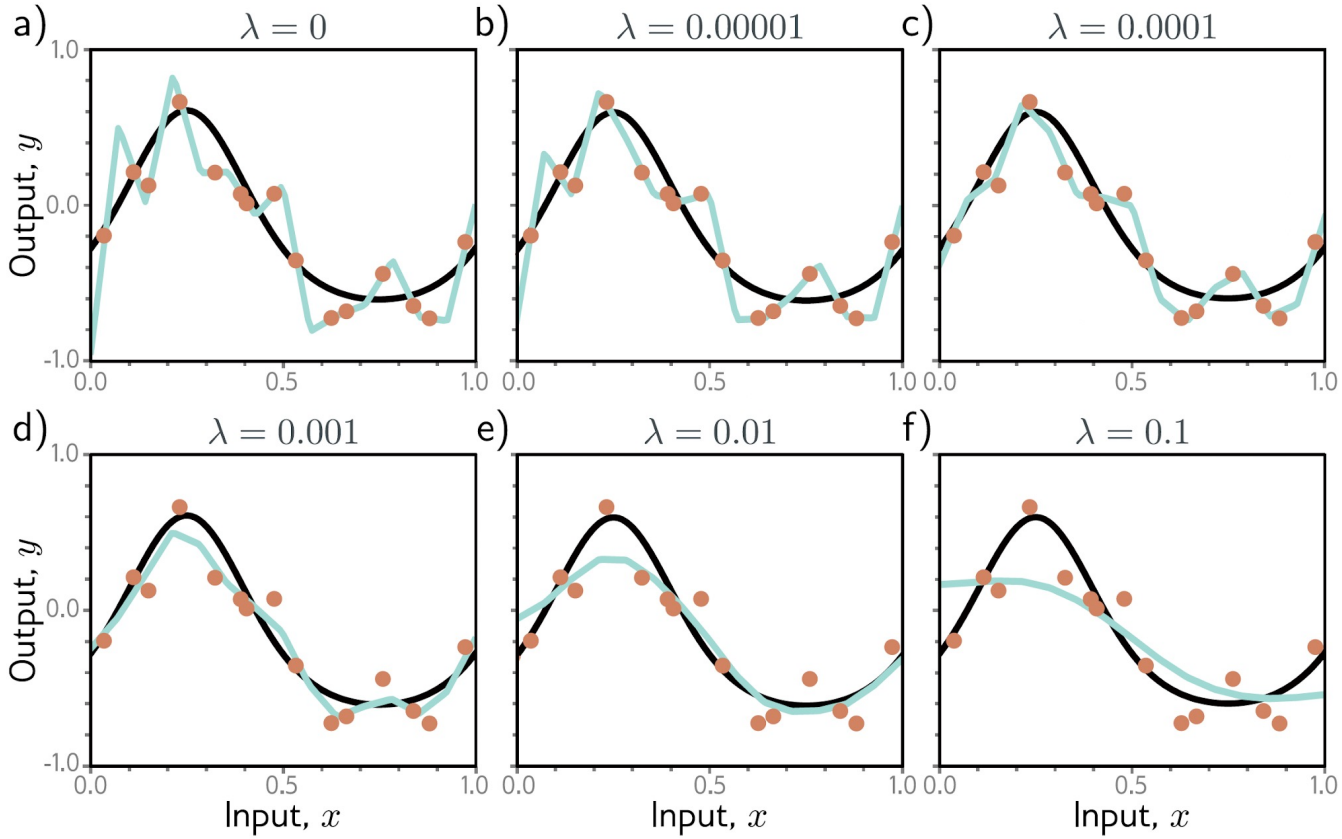


Figure 9.2 L2 regularization in simplified network (see figure 8.4). a–f) Fitted functions as we increase the regularization coefficient λ. The black curve is the true function, the orange circles are the noisy training data, and the cyan curve is the fitted model. For small λ (panels a–b), the fitted function passes exactly through the data points. For intermediate λ (panels c–d), the function is smoother and more similar to the ground truth. For large λ (panels e–f), the fitted function is smoother than the ground truth, so the fit is worse.

Source: Understanding Deep Learning, S. Prince, 2023

Implicit regularization due to gradient descent

$$\omega_{t+1} = \omega_t - \eta \cdot \frac{\partial L}{\partial \omega}$$

$$L_{GD}[\omega] = L[\omega] + \frac{\eta}{4}\left\|\frac{\partial L}{\partial \omega}\right\|^2$$

implicit penalty

Implicit regularization due to stochastic gradient descent

If we denote L the average loss overall all samples and L_B the average loss over all batches:

$$L_{SGD}[\omega] = L_{GD}[\omega] + \frac{\eta}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_B}{\partial \omega} - \frac{\partial L}{\partial \omega}\right\|^2$$

$$L_{SGD}[\omega] = L[\omega] + \frac{\eta}{4}\left\|\frac{\partial L}{\partial \omega}\right\|^2 + \frac{\eta}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_B}{\partial \omega} - \frac{\partial L}{\partial \omega}\right\|^2$$

Larger steps regularize more

Smaller batches regularize more

Larger learning rate regularize more

Smaller batches regularize more

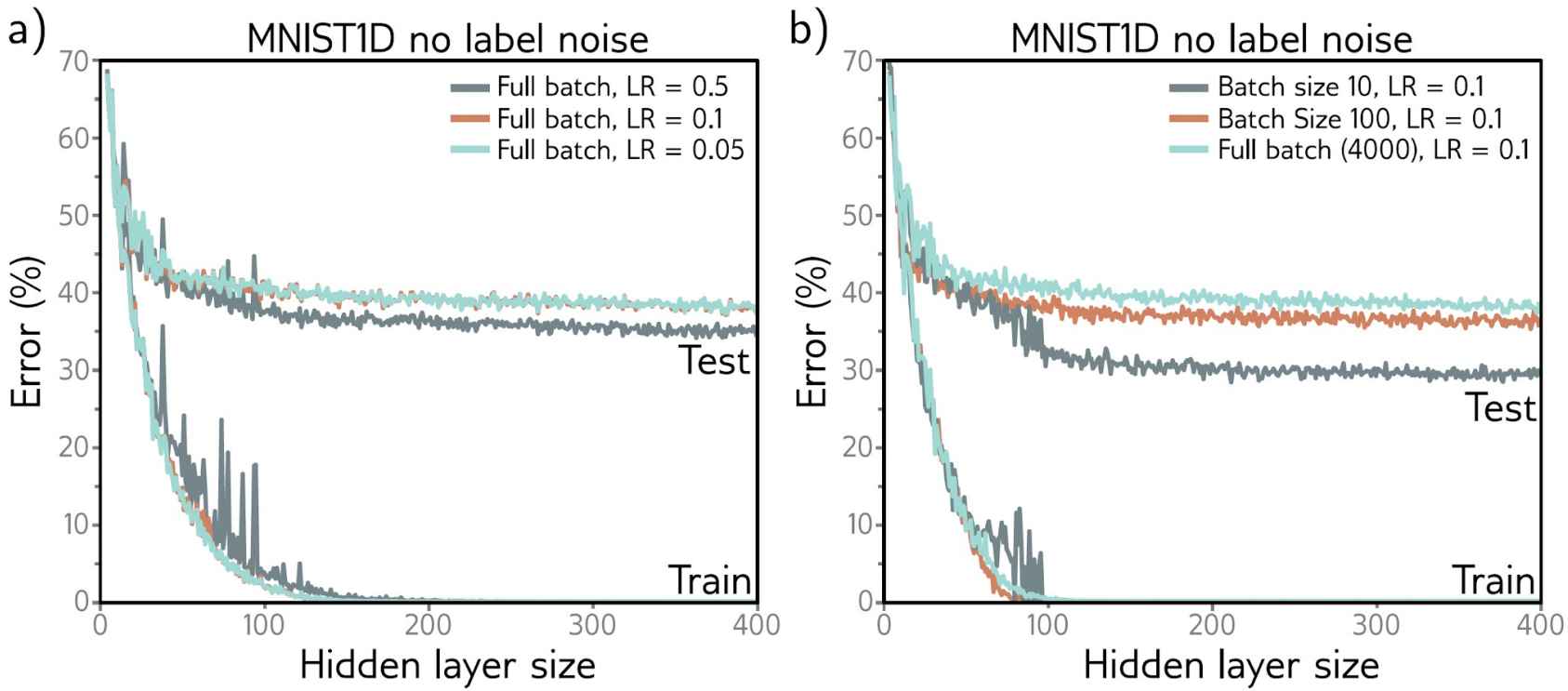


Figure 9.5 Effect of learning rate and batch size for 4000 training and 4000 test examples from MNIST-1D (see figure 8.1) for a neural network with two hidden layers. a) Performance is better for large learning rates than for intermediate or small ones. In each case, the number of iterations is 6000× the learning rate, so each solution has the opportunity to move the same distance. b) Performance is superior for smaller batch sizes. In each case, the number of iterations was chosen so that the training data were memorized at roughly the same model capacity.

Source: Understanding Deep Learning, S. Prince, 2023

Regularization

Heuristics to add regularization:

- Early stopping
- Ensembling
- Dropout
- Data augmentation

Regularization: summary

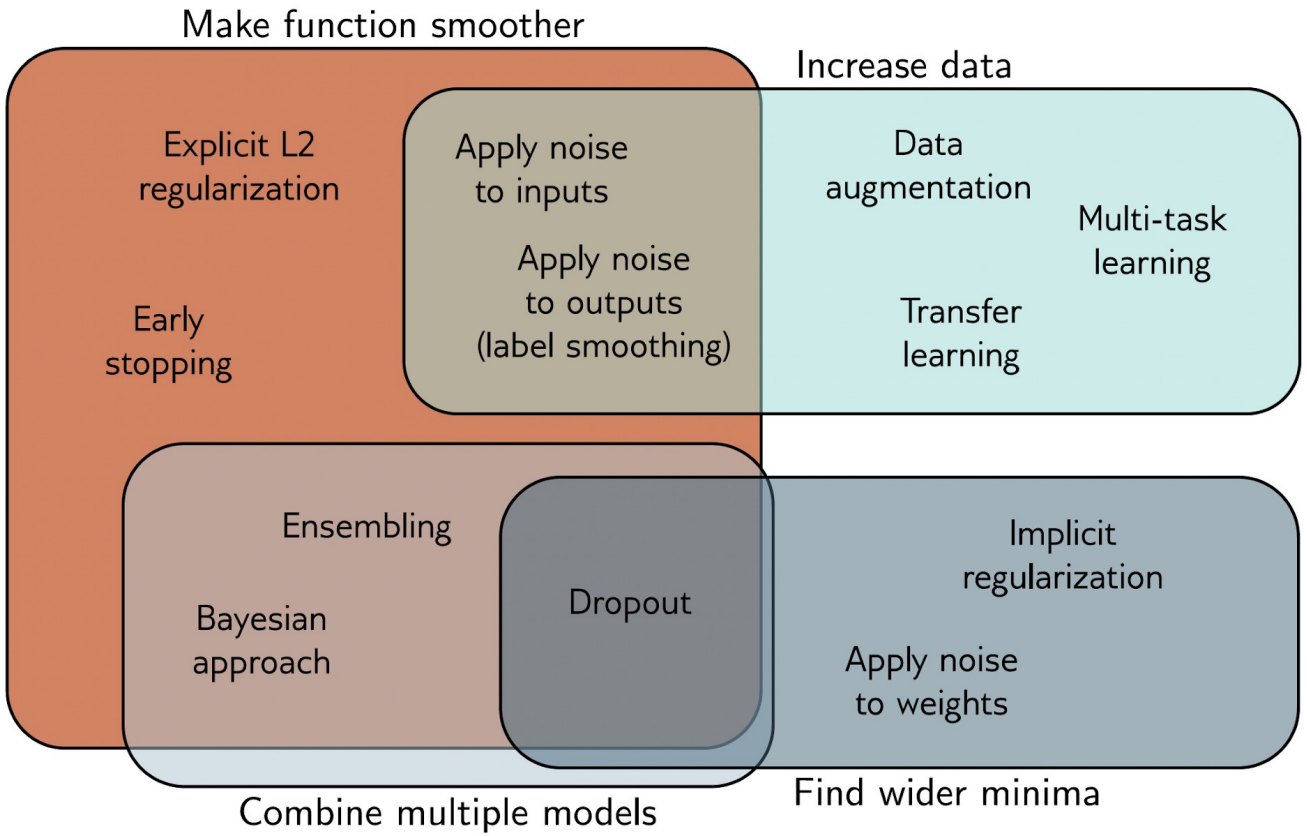


Figure 9.14 Regularization methods. The regularization methods discussed in this chapter aim to improve generalization by one of four mechanisms. Some methods aim to make the modeled function smoother. Other methods increase the effective amount of data. The third group of methods combine multiple models and hence mitigate against uncertainty in the fitting process. Finally, the fourth group of methods encourages the training process to converge to a wide minimum where small errors in the estimated parameters are less important (see also figure 20.11).

Source: Understanding Deep Learning, S. Prince, 2023

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

Initialization: forward pass

At each layer k, given weights $\boldsymbol{\Omega}$ with variance $\boldsymbol{\sigma_{\Omega}^2}$ and pre-activations $\boldsymbol{f}_k$:

$$\boldsymbol{f}_k = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot a[\boldsymbol{f}_{k-1}]$$

If $\boldsymbol{\sigma_{\Omega}^2}$ is too large → exploding gradients

If $\boldsymbol{\sigma_{\Omega}^2}$ is too small → vanishing gradients

Initialization: forward pass

At each layer k, given weights $\boldsymbol{\Omega}$ with variance $\boldsymbol{\sigma_{\Omega}^2}$ and pre-activations $\boldsymbol{f}_k$:

$$\boldsymbol{\sigma_{f_{k+1}}^2 = \frac{1}{2} D_{h_k} \sigma_{\Omega}^2 \sigma_{f_k}^2}$$

where $\boldsymbol{D_{h_k}}$ is the dimensionality of the input layer k.

Hence the optimal variance of the weights is:

$$\boldsymbol{\sigma_{\Omega}^2 = \frac{2}{D_{h_k}}}$$

He initialization

Initialization: backward pass

Similarly, the optimal variance of the weights for the backward pass is:

$$\sigma_{\Omega}^{2} = \frac{2}{D_{h_{k+1}}}$$

Overall, the optimal variance of the weights is:

$$\sigma_{\Omega}^{2} = \frac{4}{D_{h_k} + D_{h_{k+1}}}$$

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

Residual networks

Gradient descent assumes that the function is smooth.

Unfortunately, the loss becomes less and less smooth with more depth (shattered gradients).

Residual networks

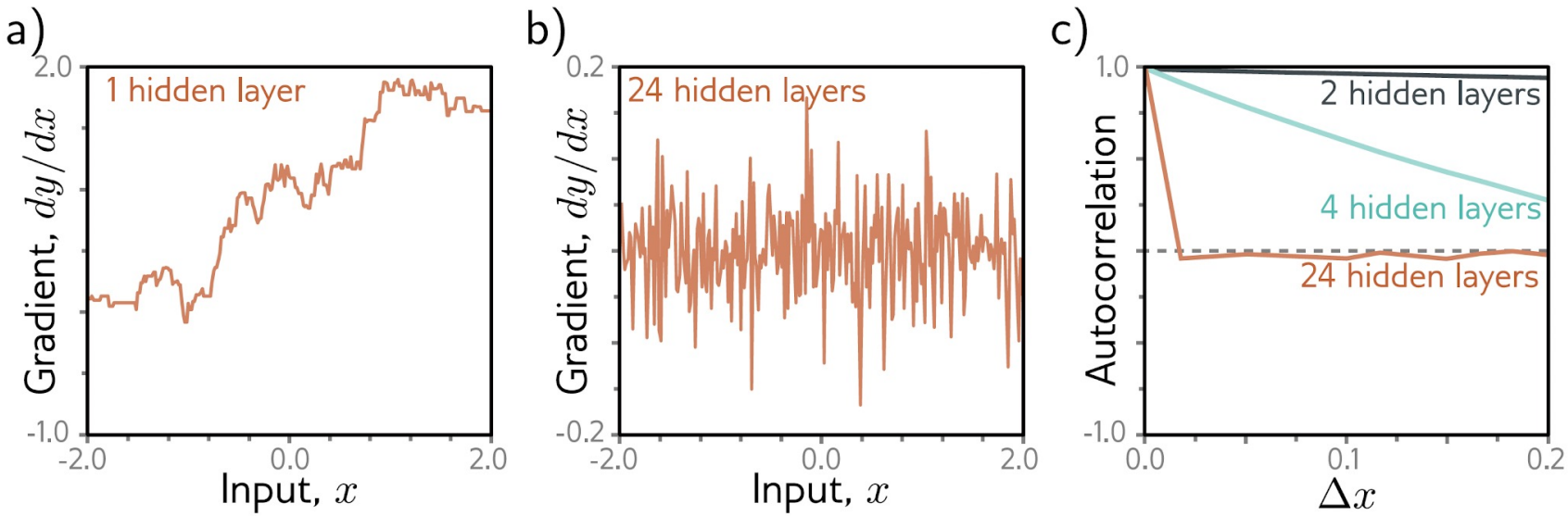


Figure 11.3 Shattered gradients. a) Consider a shallow network with 200 hidden units and Glorot initialization (He initialization without the factor of two) for both the weights and biases. The gradient $\partial y/\partial x$ of the scalar network output y with respect to the scalar input x changes relatively slowly as we change the input x. b) For a deep network with 24 layers and 200 hidden units per layer, this gradient changes very quickly and unpredictably. c) The autocorrelation function of the gradient shows that nearby gradients become unrelated (have autocorrelation close to zero) for deep networks. This *shattered gradients* phenomenon may explain why it is hard to train deep networks. Gradient descent algorithms rely on the loss surface being relatively smooth, so the gradients should be related before and after each update step. Adapted from Balduzzi et al. (2017).

Source: Understanding Deep Learning, S. Prince, 2023

Residual networks

More depth is not always better!

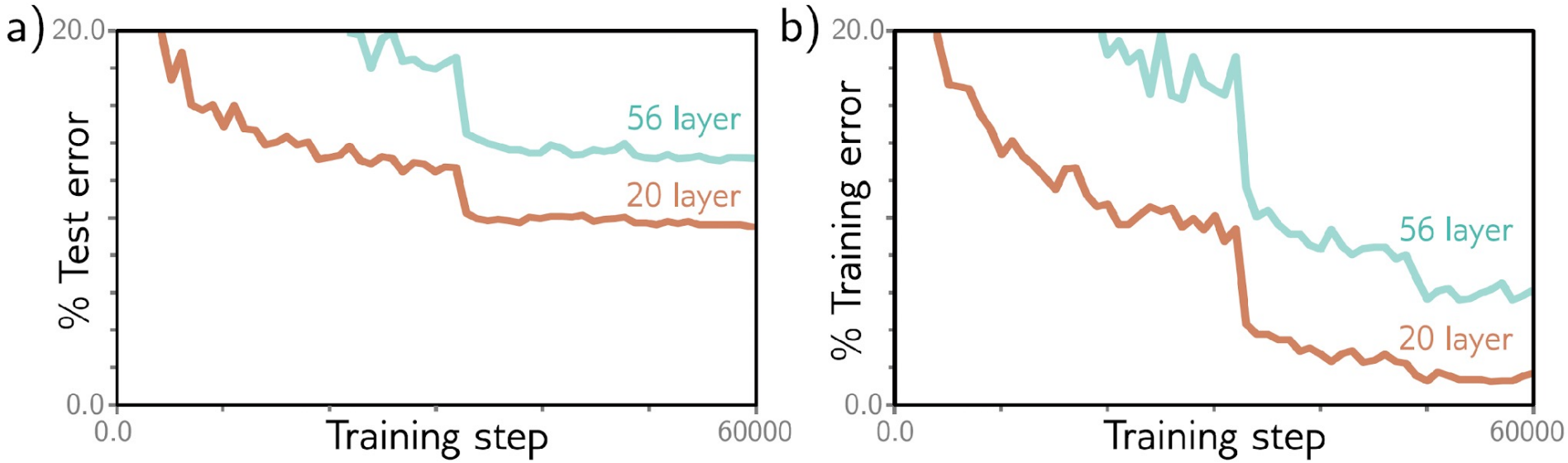


Figure 11.2 Decrease in performance when adding more convolutional layers. a) A 20-layer convolutional network outperforms a 56-layer neural network for image classification on the test set of the CIFAR-10 dataset (Krizhevsky & Hinton, 2009). b) This is also true for the training set, which suggests that the problem relates to training the original network rather than a failure to generalize to new data. Adapted from He et al. (2016a).

Source: Understanding Deep Learning, S. Prince, 2023

Residual networks

This can be addressed with skip connections.

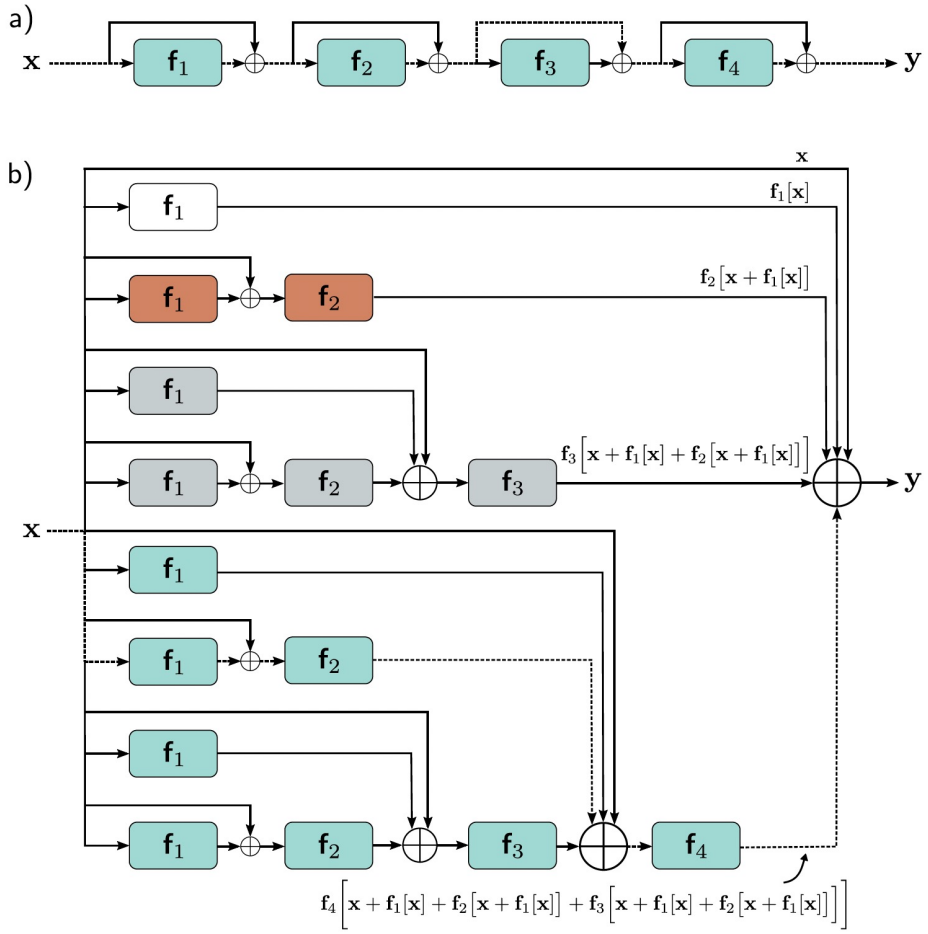


Figure 11.4 Residual connections. a) The output of each function $\mathbf{f}_k[\mathbf{x}, \boldsymbol{\phi}_k]$ is added back to its input, which is passed via a parallel computational path called a residual or skip connection. Hence, the function computes an additive change to the representation. b) Upon expanding (unraveling) the network equations, we find that the output is the sum of the input plus four smaller networks (depicted in white, orange, gray, and cyan, respectively, and corresponding to terms in equation 11.5); we can think of this as an ensemble of networks. Moreover, the output from the cyan network is itself a transformation $\mathbf{f}_4[\bullet, \boldsymbol{\phi}_4]$ of another ensemble, and so on. Alternatively, we can consider the network as a combination of 16 different paths through the computational graph. One example is the dashed path from input $\mathbf{x}$ to output $\mathbf{y}$, which is the same in panels (a) and (b).

Source: Understanding Deep Learning, S. Prince, 2023

Residual networks

This can be addressed with skip connections.

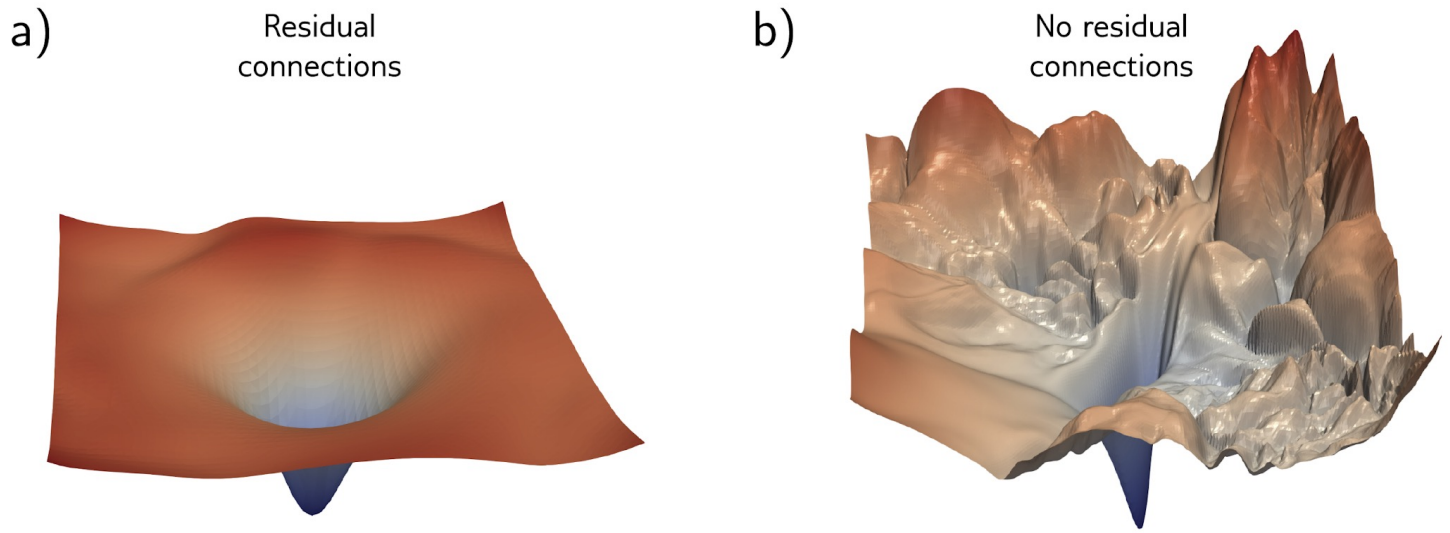


Figure 11.13 Visualizing neural network loss surfaces. Each plot shows the loss surface in two random directions in parameter space around the minimum found by SGD for an image classification task on the CIFAR-10 dataset. These directions are normalized to facilitate side-by-side comparison. a) Residual net with 56 layers. b) Results from the same network without skip connections. The surface is smoother with the skip connections. This facilitates learning and makes the final network performance more robust to minor errors in the parameters, so it will likely generalize better. Adapted from Li et al. (2018b).

Source: Understanding Deep Learning, S. Prince, 2023

Residual networks

But then we need to update the order of operations.

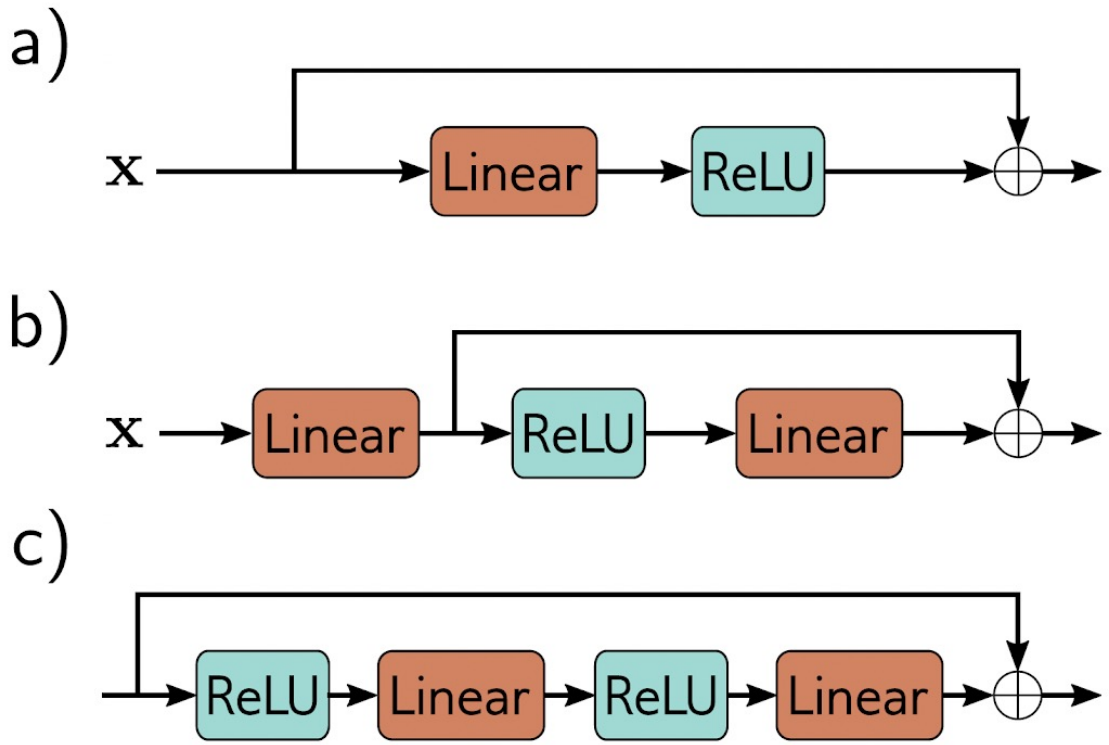


Figure 11.5 Order of operations in residual blocks. a) The usual order of linear transformation or convolution followed by a ReLU nonlinearity means that each residual block can only add non-negative quantities. b) With the reverse order, both positive and negative quantities can be added. However, we must add a linear transformation at the start of the network in case the input is all negative. c) In practice, it's common for a residual block to contain several network layers.

Source: Understanding Deep Learning, S. Prince, 2023

But then the variance doubles at every layer! -> Batch norm

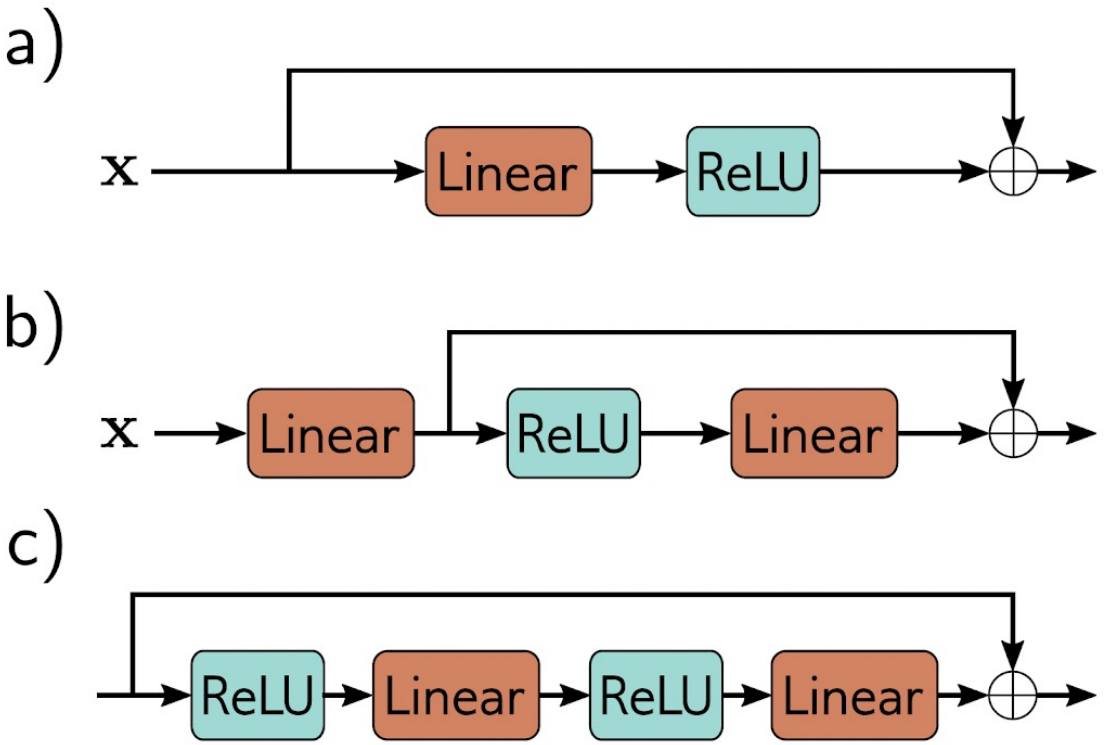


Figure 11.5 Order of operations in residual blocks. a) The usual order of linear transformation or convolution followed by a ReLU nonlinearity means that each residual block can only add non-negative quantities. b) With the reverse order, both positive and negative quantities can be added. However, we must add a linear transformation at the start of the network in case the input is all negative. c) In practice, it's common for a residual block to contain several network layers.

Source: Understanding Deep Learning, S. Prince, 2023

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

Batch norm

Batch Norm addresses the exploding gradient problem.

Alternative intuition:

We initialize the weights so that they have a nice distribution.

Why don’t we do this at each pass then? ☺

That’s Batch Norm!

Introduced by Ioff & Szegedy (2015)

Batch norm

Shift and scale each activation so that their mean and variance across the batch

become values that are learned during training.

Not constant values!

Batch norm

Compute m_B and s_B (mean and variance) over the batch during training

Normalize activations $h_i = \frac{h_i - m_h}{s_h + \epsilon}$

Scale and shift: $h_i = \gamma \cdot h_i + \delta$

γ and δ are learned during training, for each hidden unit (not each layer)

For K layers containing each D units, that's $2 * K * D$ extra parameters.

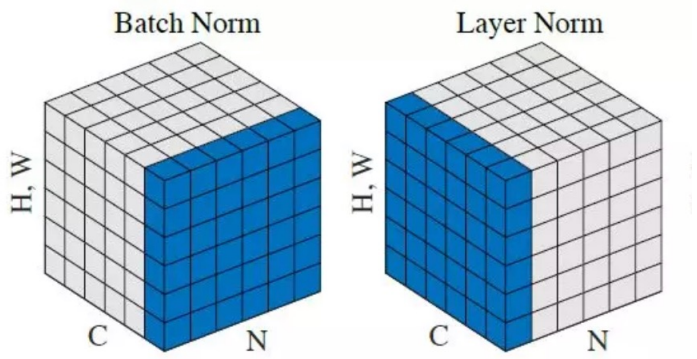
Batch norm

Downsides of Batch Norm

- Prone to bugs (need to freeze during inference)
- More parameters to learn
- Introduces dependencies between the training samples
- Needs to recompute the statistics on the whole dataset at testing time

Layer norm

Compute the layer normalization statistics over all the hidden units in the same layer.



All the hidden units in a layer share the same normalization terms
μ and σ, but different training cases have different normalization terms.

Introduced by Ba et al. in Layer Normalization

Conclusion

The choice of the loss function is critical.

Deep learning adds inductive bias.

Deep learning is not supposed to work (in theory).

Yet, it works thanks to:

- Activation function
- Regularization
- Initialization
- Residual networks
- Normalization

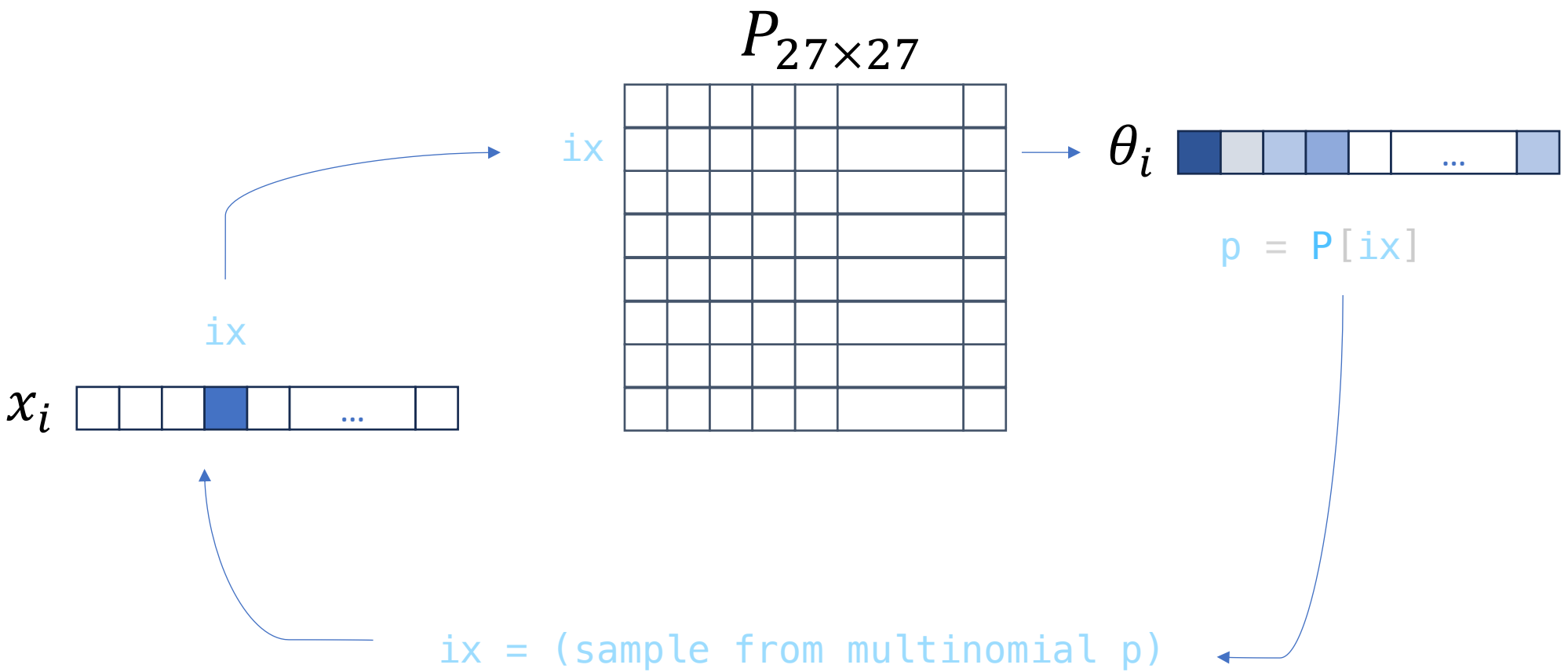
TP2: makemore

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]
4. (homework): add batchnorm to your network and report the results

Example: predict the next character in a set of words

Exercice 1: start super-simple



Example: predict the next character in a set of words

Exercice 1: start super-simple

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?

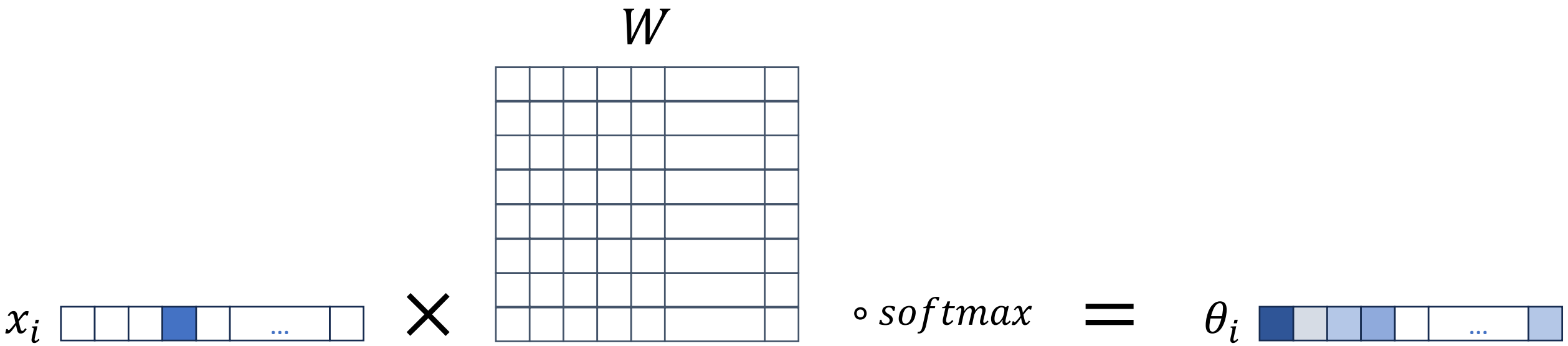
Example: predict the next character in a set of words

One-hot encoding of letter 'd'

x_i 

Example: predict the next character in a set of words

Exercice 2: minimize the NLL



We want to minimize the KL divergence or NLL

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

Example: predict the next character in a set of words

Exercice 4: Bengio et al, 2003

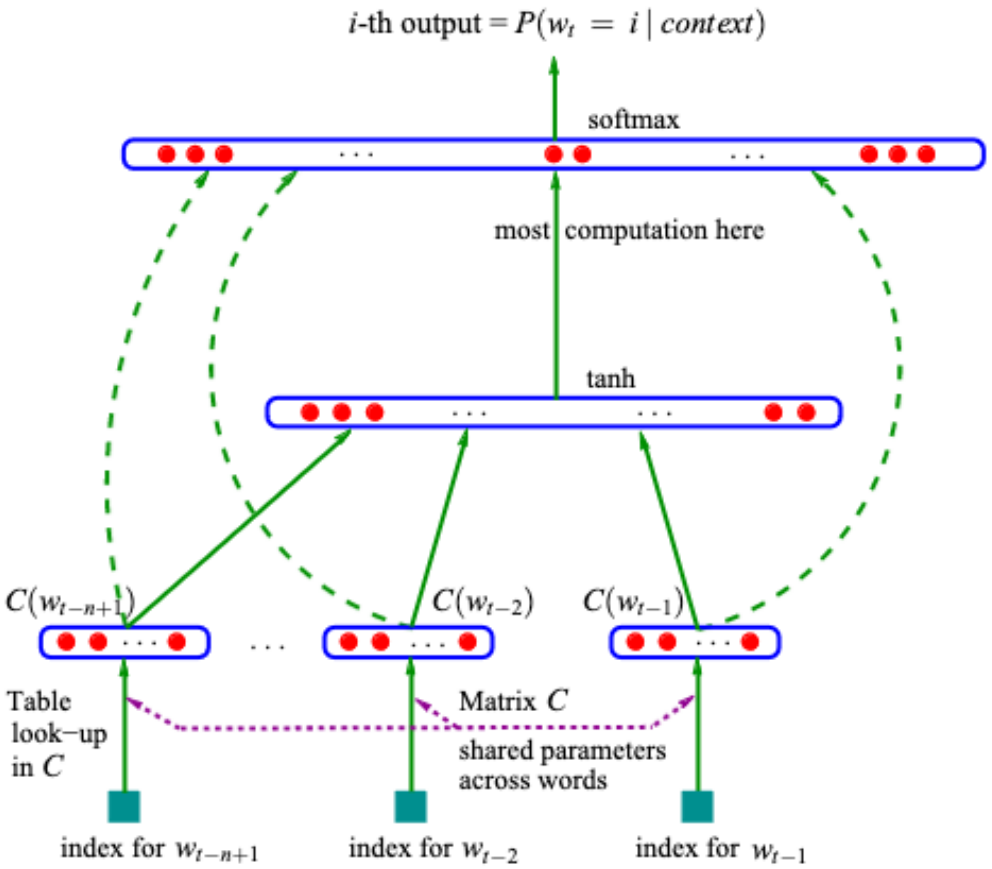


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

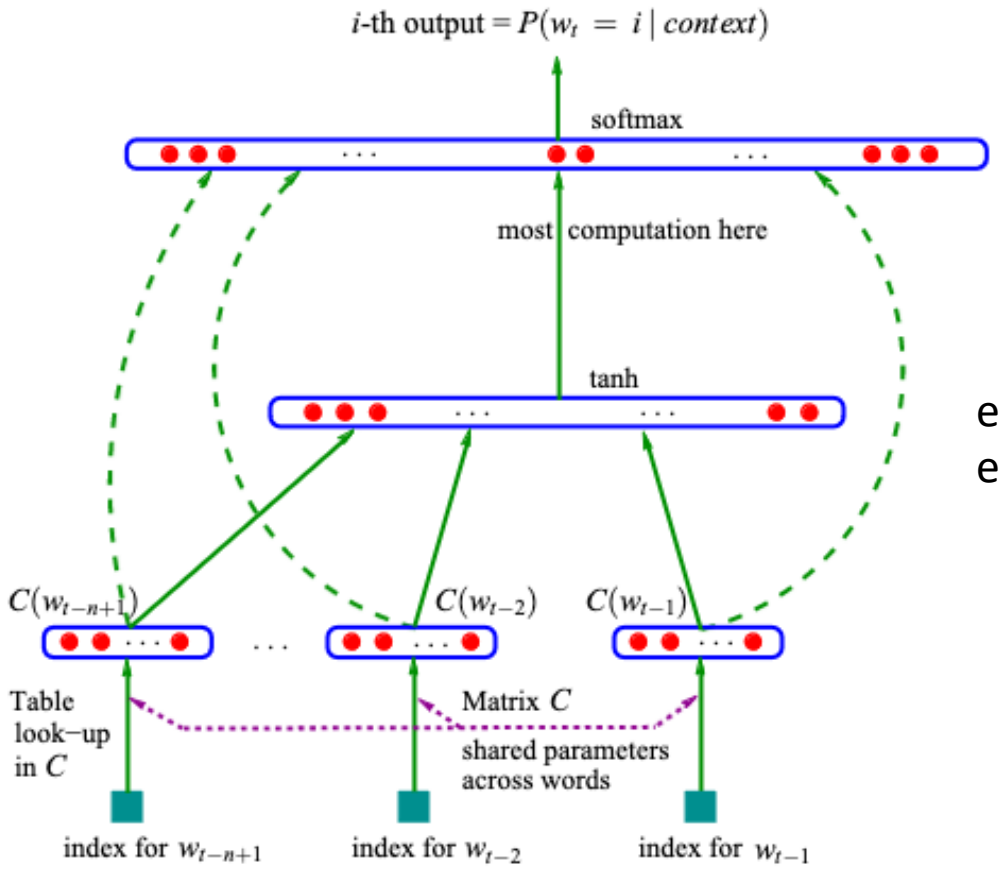
A Neural Probabilistic Language Model, Bengio et al, 2003

Example: predict the next character in a set of words

Exercice 4: Bengio et al, 2003

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

C_{27x10}= dictionary



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

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

emb = C[X_train[ix]]
emb = emb.view(…)

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

A Neural Probabilistic Language Model, Bengio et al, 2003

Batch norm: reminder

Compute m_B and s_B (mean and variance) over the batch during training

Normalize activations $h_i = \frac{h_i - m_h}{s_h + \epsilon}$

Scale and shift: $h_i = \gamma \cdot h_i + \delta$

γ and δ are learned during training, for each hidden unit (not each layer)

For K layers containing each D units, that's $2 * K * D$ extra parameters.

Batch norm: reminder

```
bngains = torch.???((1, emb_size))
bnbias = torch.???((1, emb_size))
bnmean_running = torch.???((1, emb_size))
bnstd_running = torch.???((1, emb_size))
```

Don’t forget to recompute the mean and variance over the entire dataset after training!