Deep Learning

Tutorial 4 – Multi-Layer Perceptrons I (student version)

Glenn Angrabeit
Autonomous Vision Group
University of Tübingen

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN





Tübingen AI Center
tuebingen.ai



ellis
European Laboratory for Learning and Intelligent Systems

Lecture Quiz

Question 1

A network classifies images into $K = 4$ mutually exclusive classes. Which output layer and loss function would you use for this learning problem?

(A) Softmax output, cross-entropy loss

(B) One sigmoid output per class, binary cross-entropy loss

(C) Linear output, squared error loss

(D) Softmax output, squared error loss

Question 2

When a neural network is trained to predict clean images from noisy versions of the same images with pixel-wise L_2 loss, then the network will learn to predict

(A) The denoised image, correct at every pixel

(B) The expected denoised image at the given noise level

(C) The mean image of the corresponding class label

(D) The noise vector

Question 3

All weights and biases of a network are initialized to the same constant (for example 0). What happens during training?

(A) All units of a layer compute the same output and receive the same gradient, so they stay identical

(B) The gradients explode

(C) The loss diverges

(D) Nothing special: constant and random initialization behave the same

Question 4

A layer has $D_{\text{in}} = 400$ inputs with zero mean and unit variance. With which standard deviation σ should we draw the weights so that the pre-activations $h = \sum_i a_i x_i$ also have unit variance?

(A) $\sigma = 1/20 = 0.05$

(B) $\sigma = 1/400$

(C) $\sigma = 1$

(D) $\sigma = 0.01$, independent of D_{in}

Question 5

Why do we use data augmentation?

(A) To train for more gradient steps

(B) To achieve a stable initialization

(C) To reduce the training error

(D) To make the network robust to changes in the input

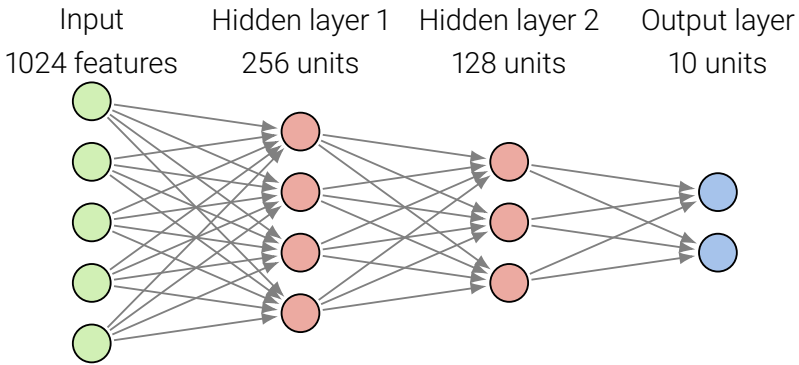
Warm-up Questions

True or False?

Statement	**True**	**False**
Logistic regression is used to solve regression problems.		
The backpropagation algorithm computes the output of all compute nodes in the forward pass and all model parameter updates in the backward pass.		
Maximizing a Bernoulli model distribution equals maximizing the BCE loss.		
The output of the softmax function always sums to 1. **Reminder:** $\mathrm{softmax}(\mathbf{s})_i = \exp(s_i) \,/\, \sum_{k=1}^{C} \exp(s_k),\ i = 1, \ldots, C$		
The universal approximation theorem states that neural networks with 1 layer which may be arbitrarily wide are able to approximate any continuous function in the n-dimensional unit cube arbitrarily well.		

MLPs: Counting parameters

How many learnable parameters are in this MLP (ignoring normalization)?

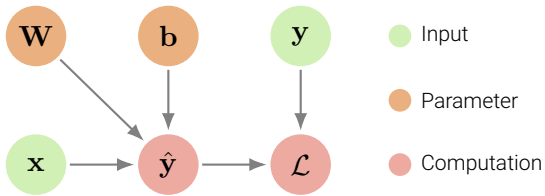


Recap: Backpropagation

Recap: Computation graphs

Consider linear regression with an L_2 loss:

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b}, \qquad \mathcal{L} = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$$



The gradients we are interested in belong to the **parameter nodes**: $\nabla_{\mathbf{b}}\mathcal{L}$, $\nabla_{\mathbf{W}}\mathcal{L}$

Recap: Gradient descent and the Jacobian

Gradient descent updates each entry using its derivative at the current values:

$$b_i \leftarrow b_i - \eta \, \partial \mathcal{L} / \partial b_i$$

Recap: Gradient descent and the Jacobian

Gradient descent updates each entry using its derivative at the current values:

$$b_i \leftarrow b_i - \eta \, \partial \mathcal{L} / \partial b_i$$

The **gradient** collects these partial derivatives to simplify notation:

$$\nabla_{\mathbf{b}} \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial b_1}, \ldots, \frac{\partial \mathcal{L}}{\partial b_n} \right)^\top \quad \Rightarrow \quad \mathbf{b} \leftarrow \mathbf{b} - \eta \, \nabla_{\mathbf{b}} \mathcal{L}$$

Recap: Gradient descent and the Jacobian

Gradient descent updates each entry using its derivative at the current values:

$$b_i \leftarrow b_i - \eta \, \partial \mathcal{L} / \partial b_i$$

The **gradient** collects these partial derivatives to simplify notation:

$$\nabla_{\mathbf{b}} \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial b_1}, \ldots, \frac{\partial \mathcal{L}}{\partial b_n} \right)^\top \quad \Rightarrow \quad \mathbf{b} \leftarrow \mathbf{b} - \eta \, \nabla_{\mathbf{b}} \mathcal{L}$$

The **Jacobian** of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is the matrix of all partial derivatives:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

Recap: Gradient descent and the Jacobian

Gradient descent updates each entry using its derivative at the current values:

$$b_i \leftarrow b_i - \eta \, \partial \mathcal{L} / \partial b_i$$

The **gradient** collects these partial derivatives to simplify notation:

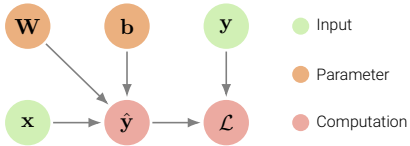
$$\nabla_{\mathbf{b}} \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial b_1}, \ldots, \frac{\partial \mathcal{L}}{\partial b_n} \right)^\top \quad \Rightarrow \quad \mathbf{b} \leftarrow \mathbf{b} - \eta \, \nabla_{\mathbf{b}} \mathcal{L}$$

The **Jacobian** of $\mathbf{f} \colon \mathbb{R}^n \to \mathbb{R}^m$ is the matrix of all partial derivatives:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

Since $\mathcal{L}$ is a scalar-valued function ($m = 1$), its Jacobian equals the row vector $\left(\frac{\partial \mathcal{L}}{\partial b_1}, \ldots, \frac{\partial \mathcal{L}}{\partial b_n} \right)$ and thus the transpose of the gradient: $\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \left(\nabla_{\mathbf{b}} \mathcal{L} \right)^\top$

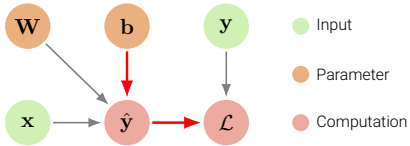
Recap: Backpropagation



Backpropagation provides a mathematical framework to efficiently compute the Jacobian $\partial\mathcal{L}/\partial\mathbf{b}$:

- Trace the path backwards from $\mathcal{L}$ to $\mathbf{b}$ and multiply all neighboring derivatives along the way.
- If there are multiple paths, sum over all of them.

Recap: Backpropagation

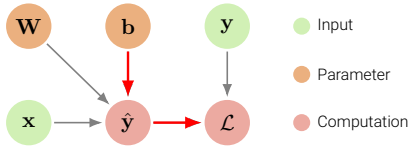


Backpropagation provides a mathematical framework to efficiently compute the Jacobian $\partial\mathcal{L}/\partial\mathbf{b}$:

- Trace the path backwards from $\mathcal{L}$ to $\mathbf{b}$ and multiply all neighboring derivatives along the way.
- If there are multiple paths, sum over all of them.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{b}}$$

Recap: Backpropagation



Backpropagation provides a mathematical framework to efficiently compute the Jacobian $\partial\mathcal{L}/\partial\mathbf{b}$:

- Trace the path backwards from $\mathcal{L}$ to $\mathbf{b}$ and multiply all neighboring derivatives along the way.
- If there are multiple paths, sum over all of them.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{b}}$$

Use this algorithm to find $\partial\mathcal{L}/\partial\mathbf{b}$ and take its transpose to get the gradient $\nabla_{\mathbf{b}}\mathcal{L}$.

Gradient descent for matrices

Analogously to vectors, we want to update each entry of a matrix using its derivative:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

Gradient descent for matrices

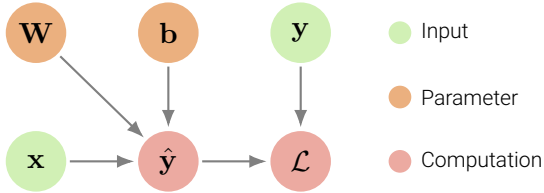
Analogously to vectors, we want to update each entry of a matrix using its derivative:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

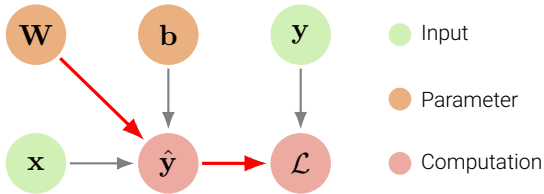
The **gradient** again collects these partial derivatives and preserves the original shape:

$$\nabla_{\mathbf{W}} \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{m1}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{mn}} \end{pmatrix} \quad \Rightarrow \quad \mathbf{W} \leftarrow \mathbf{W} - \eta \, \nabla_{\mathbf{W}} \mathcal{L}$$

Backpropagation for matrices

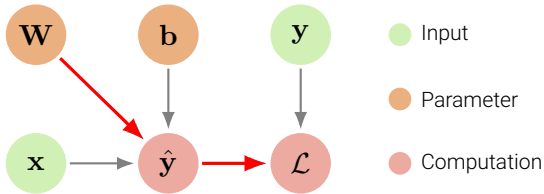


Backpropagation for matrices



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}}$$

Backpropagation for matrices



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}}$$

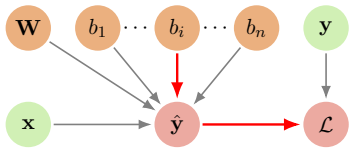
This is not defined for matrices:

$\partial \hat{\mathbf{y}} / \partial \mathbf{W}$ has three indices, so it does not fit into a matrix product.

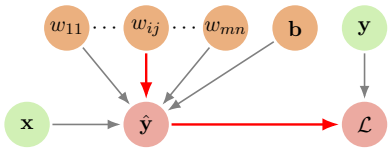
Solution: break down the parameters

As in the loops of lecture 3, take one entry at a time. The chain rule then works.

Bias $\mathbf{b}$



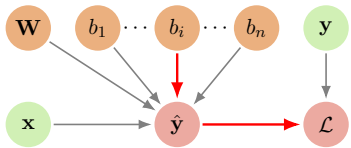
Matrix $\mathbf{W}$



Solution: break down the parameters

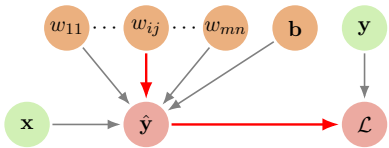
As in the loops of lecture 3, take one entry at a time. The chain rule then works.

Bias $\mathbf{b}$



$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial b_i}$$

Matrix $\mathbf{W}$

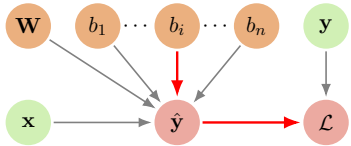


$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial w_{ij}}$$

Solution: break down the parameters

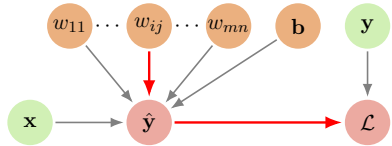
As in the loops of lecture 3, take one entry at a time. The chain rule then works.

Bias $\mathbf{b}$



$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial b_i}$$

Matrix $\mathbf{W}$



$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial w_{ij}}$$

Stack the entries:

$$\nabla_{\mathbf{b}} \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial b_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial b_n} \end{pmatrix}$$

$$\nabla_{\mathbf{W}} \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{m1}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{mn}} \end{pmatrix}$$

Backpropagation on MLPs

Old exam question

Consider a simple MLP with one hidden layer:

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \qquad y = \mathbf{W}_2\mathbf{h} + b_2$$

Compute the value of the output y and the gradient $\nabla_{\mathbf{W}_1} y$ for the following values:

$$\mathbf{W}_1 = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad \mathbf{W}_2 = \begin{pmatrix} 1 & 2 \end{pmatrix}, \quad b_2 = -1, \quad \mathbf{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Reminder: $\text{ReLU}(x) = \max(x, 0)$

Old exam question

Consider a simple MLP with one hidden layer:

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \qquad y = \mathbf{W}_2\mathbf{h} + b_2$$

Compute the value of the output y and the gradient $\nabla_{\mathbf{W}_1} y$ for the following values:

$$\mathbf{W}_1 = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad \mathbf{W}_2 = \begin{pmatrix} 1 & 2 \end{pmatrix}, \quad b_2 = -1, \quad \mathbf{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

1. Draw the computation graph
2. Compute y in the forward pass
3. Write out the equation for $\partial y / \partial [\mathbf{W}_1]_{ij}$ using the chain rule

Reminder: $\text{ReLU}(x) = \max(x, 0)$

Steps 4–5: your turn

$$\mathbf{W}_1 = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad \mathbf{W}_2 = \begin{pmatrix} 1 & 2 \end{pmatrix}, \quad b_2 = -1, \quad \mathbf{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

(2) $$\mathbf{u} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \quad \mathbf{h} = \mathrm{ReLU}(\mathbf{u}) = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \quad y = \mathbf{W}_2\mathbf{h} + b_2 = 1$$

(3) $$\frac{\partial y}{\partial [\mathbf{W}_1]_{ij}} = \frac{\partial y}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{u}} \cdot \frac{\partial \mathbf{u}}{\partial [\mathbf{W}_1]_{ij}}$$

4. Derive $\partial y / \partial [\mathbf{W}_1]_{ij}$
5. Plug in the values and arrange the gradient

Reminder: $\mathrm{ReLU}'(u) = \begin{cases} 1 & u > 0 \\ 0 & \text{else} \end{cases}$

Backpropagation with Tensors

Old exam question (2024)

Assume that we are given as input a matrix $\mathbf{X} \in \mathbb{R}^{B \times C}$ and we want to apply the softmax function across the second dimension. Assume that our output is called $\mathbf{Y}$.

a) What shape does $\mathbf{Y}$ have? Define the batchwise softmax function in terms of its elements y_{ij}.

c) Given a loss $\mathcal{L}(\mathbf{Y})$, specify $\partial \mathcal{L} / \partial x_{ij}$ as well as $\partial y_{ic} / \partial x_{ij}$. You can consider the partial derivative $\partial \mathcal{L} / \partial y_{ic}$ as given.

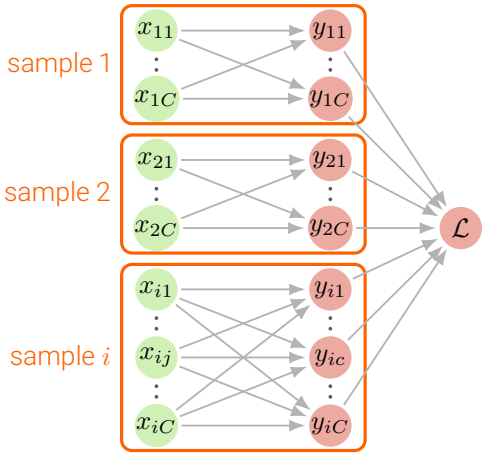
Task a)

What shape does $\mathbf{Y}$ have for $\mathbf{X} \in \mathbb{R}^{B \times C}$? Define the batchwise softmax via y_{ij}.

One sample: $\mathbf{x} \in \mathbb{R}^C \Rightarrow \mathbf{y} \in \mathbb{R}^C, \quad y_j = \frac{\exp(x_j)}{\sum_{c=1}^{C} \exp(x_c)}$

Task c): the chain rule

For $\mathcal{L}(\mathbf{Y})$, specify $\partial\mathcal{L}/\partial x_{ij}$ and $\partial y_{ic}/\partial x_{ij}$. $\partial\mathcal{L}/\partial y_{ic}$ is given.



Chain rule: sum over all paths from $\mathcal{L}$ to x_{ij}

Task c): the softmax derivative

For $\mathcal{L}(\mathbf{Y})$, specify $\partial\mathcal{L}/\partial x_{ij}$ and $\partial y_{ic}/\partial x_{ij}$. $\partial\mathcal{L}/\partial y_{ic}$ is given.

One sample: the softmax derivative from exercise sheet 2

$$\frac{\partial y_c}{\partial x_j} = \begin{cases} y_j(1 - y_j) & c = j \\ -y_c\, y_j & c \neq j \end{cases}$$

Old exam question (2024): implementation

Softmax across the second dimension of $\mathbf{X} \in \mathbb{R}^{B \times C}$, output $\mathbf{Y}$.

a) $\mathbf{Y} \in \mathbb{R}^{B \times C}, \quad y_{ij} = \dfrac{\exp(x_{ij})}{\sum_{c=1}^{C} \exp(x_{ic})}$

b) Implement the forward pass of the softmax function from task a) in EDF. You only need to write the body of the forward function.

c) $\dfrac{\partial \mathcal{L}}{\partial x_{ij}} = \sum_{c=1}^{C} \dfrac{\partial \mathcal{L}}{\partial y_{ic}} \dfrac{\partial y_{ic}}{\partial x_{ij}}, \quad \dfrac{\partial y_{ic}}{\partial x_{ij}} = \begin{cases} y_{ij}(1 - y_{ij}) & c = j \\ -y_{ic}\, y_{ij} & c \neq j \end{cases}$

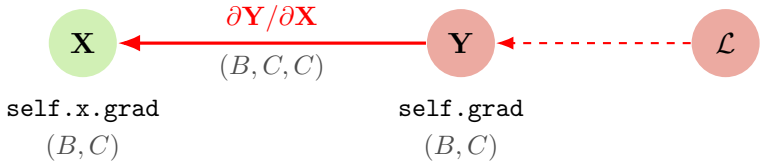
d) Implement the backward pass of the softmax function in EDF. You only need to write the body of the backward function. It is **not** required to use vectorized numpy code. An implementation with nested for-loops suffices.

Recap: EDF, the forward pass

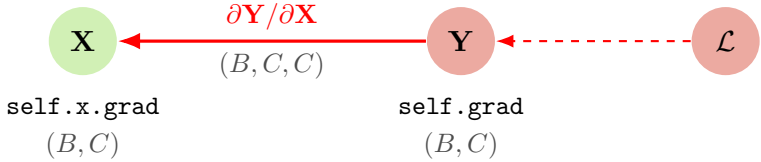
b) Implement the forward pass of the softmax function from task a) in EDF.



Recap: EDF, the backward pass



Recap: EDF, the backward pass



`self.grad[i, c]` $= \partial \mathcal{L}/\partial y_{ic}$ given, filled in by the nodes after $\mathbf{Y}$

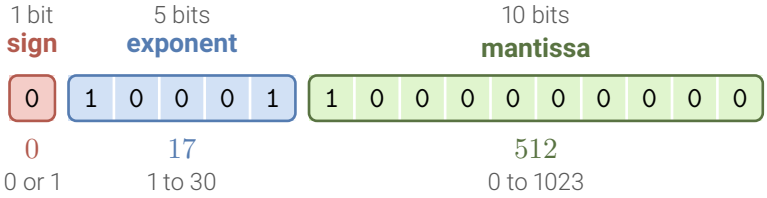
`self.x.grad[i, j]` $= \partial \mathcal{L}/\partial x_{ij}$ our job, via `self.x.addgrad(...)`

Task d): the backward pass

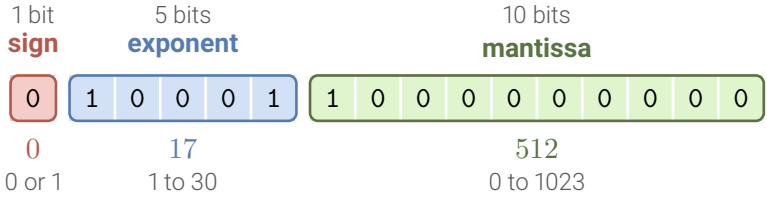
d) Implement the backward pass of the softmax in EDF (for-loops suffice).

Numerical Stability

How a float16 stores a number

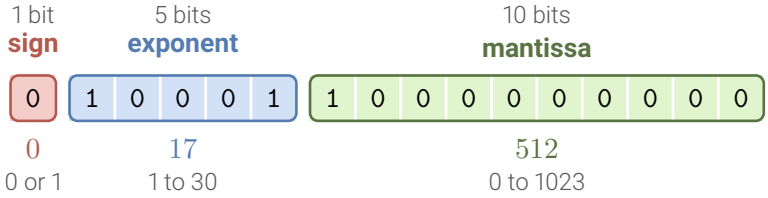


How a float16 stores a number



$$x = (-1)^{\text{sign}} \cdot 2^{\text{exponent} - 15} \cdot \left(1 + \frac{\text{mantissa}}{1024}\right)$$

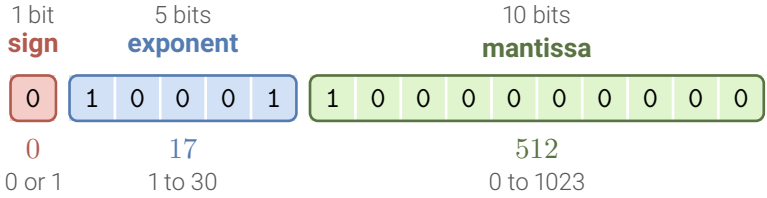
How a float16 stores a number



$$x = (-1)^{\text{sign}} \cdot 2^{\text{exponent} - 15} \cdot \left(1 + \frac{\text{mantissa}}{1024}\right)$$

$$6.0 = (-1)^{0} \cdot 2^{17-15} \cdot \left(1 + \frac{512}{1024}\right) = 1 \cdot 4 \cdot 1.5$$

How a float16 stores a number



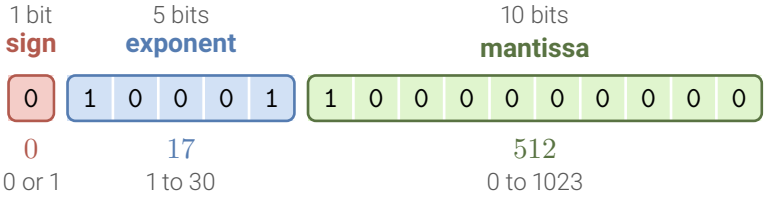
$$x = (-1)^{\text{sign}} \cdot 2^{\text{exponent} - 15} \cdot \left(1 + \frac{\text{mantissa}}{1024}\right)$$

$$6.0 = (-1)^{0} \cdot 2^{17-15} \cdot \left(1 + \frac{512}{1024}\right) = 1 \cdot 4 \cdot 1.5$$

Limited range: largest x

$$2^{30-15} \cdot \left(1 + \tfrac{1023}{1024}\right) = 65504$$

How a float16 stores a number



$$x = (-1)^{\text{sign}} \cdot 2^{\text{exponent} - 15} \cdot \left(1 + \frac{\text{mantissa}}{1024}\right)$$

$$6.0 = (-1)^{0} \cdot 2^{17-15} \cdot \left(1 + \frac{512}{1024}\right) \;=\; 1 \cdot 4 \cdot 1.5$$

Limited range: largest x

$$2^{30-15} \cdot \left(1 + \tfrac{1023}{1024}\right) = 65504$$

Limited accuracy: smallest $x > 0$

$$2^{1-15} \cdot \left(1 + \tfrac{0}{1024}\right) \approx 6.1 \cdot 10^{-5}$$

Two failure cases

Overflow: numbers outside the range are stored as $\pm\infty$ (float16: $70000 \to \infty$)

Two failure cases

Overflow: numbers outside the range are stored as $\pm\infty$ (float16: $70000 \to \infty$)

- The value is lost: $70000 \neq 80000$, but both become ∞
- Arithmetic with ∞ gives NaN: $\infty - \infty = \text{NaN}$, $0 \cdot \infty = \text{NaN}$

Two failure cases

Overflow: numbers outside the range are stored as $\pm\infty$ (float16: $70000 \to \infty$)

- The value is lost: $70000 \neq 80000$, but both become ∞
- Arithmetic with ∞ gives NaN: $\infty - \infty = \text{NaN}$, $0 \cdot \infty = \text{NaN}$

⚠ **Warning:** a single NaN in the loss makes all gradients and weights NaN.

Two failure cases

Overflow: numbers outside the range are stored as $\pm\infty$ (float16: $70000 \to \infty$)

- The value is lost: $70000 \neq 80000$, but both become ∞
- Arithmetic with ∞ gives NaN: $\infty - \infty = \text{NaN}$, $0 \cdot \infty = \text{NaN}$

> ⚠ **Warning:** a single NaN in the loss makes all gradients and weights NaN.

Underflow: non-zero numbers too close to 0 are stored as 0 (float16: $10^{-8} \to 0$)

Two failure cases

Overflow: numbers outside the range are stored as $\pm\infty$ (float16: $70000 \to \infty$)

- The value is lost: $70000 \neq 80000$, but both become ∞
- Arithmetic with ∞ gives NaN: $\infty - \infty = \text{NaN}$, $0 \cdot \infty = \text{NaN}$

> ⚠ **Warning:** a single NaN in the loss makes all gradients and weights NaN.

Underflow: non-zero numbers too close to 0 are stored as 0 (float16: $10^{-8} \to 0$)

- Harmless on its own: $10^{-8} \approx 0$
- A problem once a function needs $x \neq 0$: $\log(0) = -\infty$, $1/0 = \infty$

Recap: the sigmoid function

Standard definition: $\sigma(x) = \frac{1}{1 + e^{-x}}$

Recap: the sigmoid function

Standard definition: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

- If $x \gg 0$: $\sigma(x) = \dfrac{1}{1 + e^{-x}} = \dfrac{1}{1 + 0} = 1$

  $\Rightarrow$ e^{-x} underflows along the way, but that is no concern

Recap: the sigmoid function

Standard definition: $\sigma(x) = \dfrac{1}{1+e^{-x}}$

- If $x \gg 0$: $\sigma(x) = \dfrac{1}{1+e^{-x}} = \dfrac{1}{1+0} = 1$

  $\Rightarrow$ e^{-x} underflows along the way, but that is no concern

- If $x \ll 0$: $\sigma(x) = \dfrac{1}{1+e^{-x}} = \dfrac{1}{1+\infty} = \dfrac{1}{\infty} = 0$

  $\Rightarrow$ e^{-x} overflows along the way and the result underflows

Recap: the sigmoid function

Standard definition: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

- If $x \gg 0$: $\sigma(x) = \dfrac{1}{1 + e^{-x}} = \dfrac{1}{1 + 0} = 1$

  $\Rightarrow$ e^{-x} underflows along the way, but that is no concern

- If $x \ll 0$: $\sigma(x) = \dfrac{1}{1 + e^{-x}} = \dfrac{1}{1 + \infty} = \dfrac{1}{\infty} = 0$

  $\Rightarrow$ e^{-x} overflows along the way and the result underflows

Even though we arrive at an acceptable result by accident, let us **avoid the overflow**.

Your turn: a second form of the sigmoid

Show that $$\frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}.$$

1. What happens for $x \ll 0$? Did it help?

2. What happens for $x \gg 0$? Does it still work?

Your turn: the log-sigmoid

Losses use $\log \sigma(x)$, but even the stable sigmoid **underflows** for $x \ll 0$:

$$\sigma(x) \to 0 \quad \Rightarrow \quad \log \sigma(x) = \log(0) = -\infty$$

Show that:

i) $\log \sigma(x) = -\log\left(1 + e^{-x}\right)$

ii) $\log \sigma(x) = x - \log\left(1 + e^{x}\right)$

Which one would you choose for $x \geq 0$, and which for $x < 0$?