Lecture 8
Backpropagation and Gradient Descent

Ivan Titov
(with slides from Edoardo Ponti)

School of informatics

Forward and Backward Pass

Last lectures: **forward pass** (inference): from a document $\mathbf{x}$ to the conditional probability over the classes $p_{\theta_s}(y \mid \mathbf{x})$, given current parameters θ_s.

$$p(y \mid \mathbf{x}) = f_{\text{forward}}(\mathbf{x}, \theta_s)$$

This lecture: **backward** pass (training): from a **loss function** (divergence between true and predicted probabilities) $\ell(\cdot)$ to **updated parameters** θ_{s+1}.

$$\theta_{s+1} = f_{\text{backward}}(\ell(\cdot), \theta_s)$$

Recap: Loss function

Minimizing the negative conditional log-likelihood (or maximizing conditional log-likelihood):

$$\arg\min_{\hat{\theta}} \sum_{\mathbf{x},y\in\mathcal{D}} -\log p_{\hat{\theta}}(\mathbf{y}|\mathbf{x})$$

In this lecture, we will treat $\mathbf{y}$ as a one-hot vector, i.e. a vector where the component corresponding to the ground-truth (i.e. annotated) class is set to 1, and the rest are set to 0.

Recap: Parameter update

- Generally, we cannot find the solution to the minimisation problem analytically.
- Instead, we resort to numerical methods to find an approximately optimal solution by iteratively updating the parameters:

$$\theta_{s+1} = \theta_s - \text{update\_size} \times \text{update\_direction}$$

A parameter update step

More formally:

$$\theta_{s+1} = \theta_s - \underbrace{\eta}_{\text{update\_size}} \times \underbrace{\nabla_\theta \left(- \sum_{\mathbf{x} \in \mathcal{D}} \log f_{\theta_s}(\mathbf{x}) \right)}_{\text{update\_direction}}$$

where η is the **learning rate** (the size of a training step).

∇_θ denotes the **gradient** of a function (here, the loss $\ell(\cdot)$) with respect to a variable θ.

Intuitively, the gradient points to the direction of **steepest ascent** in the function. By subtracting it, we reduce the loss function value (more on this later).

Recap: Gradient Descent Algorithm

An **optimisation algorithm** will start from a parameter initialisation θ_0 and perform update steps until a stopping criterion $C(\cdot)$ is met.

Require: data $\mathcal{D}$, loss function $\ell(\cdot)$
initial parameters θ_0
learning rate η
stopping criterion $C(\cdot) \rightarrow \{\text{True, False}\}$

$s \leftarrow 0$
while $s < S$ **do**
$\theta_{s+1} = \theta_s - \eta \times \nabla_\theta \ell(\theta, \mathcal{D})$
if $C(\cdot) = \text{True}$ **then**
return θ_s
$s \leftarrow s + 1$
return θ_S

Parameter Initialisation

The approximate solution found by gradient descent (and its quality) depends on the parameter initialisation θ_0 (especially for deep neural networks).

Commonly, θ_0 is **randomly sampled** from a Uniform or Normal distribution.

Stopping Criteria

Examples of stopping criteria include:

- **Convergence**: when the difference between parameters before and after the update is smaller than a threshold: $\theta_{s+1} - \theta_s < \tau$.

- **Early stopping**: when the negative log likelihood (NLL) on a development step stops decreasing (or classification performance stops increasing).

Learning Rate

The learning rate η, which determines the step size, is a hyper-parameter.

The choice of its value constitutes a trade-off between the rate of convergence and the risk of overshooting $\theta^\star$.

Different values can be **scheduled** for different optimisation steps (e.g. η can be linearly or exponentially decayed).

Some optimisers (e.g., Adam) contrary to SGD allow for parameter-specific learning rates (e.g., based on the second-order **momentum** of the gradient).

Recup: Mini-batch/Stochatc Gradient Descent

Vanilla Gradient Descent requires us to estimate the gradient on the entire training dataset $\mathcal{D}$, which is unfeasible for large $|\mathcal{D}|$.

Stochastic gradient descent (SGD) instead performs parameter updates based on mini-batches of data $\mathcal{B}$ (i.e., small subsets of examples sampled i.i.d. from $\mathcal{D}$).

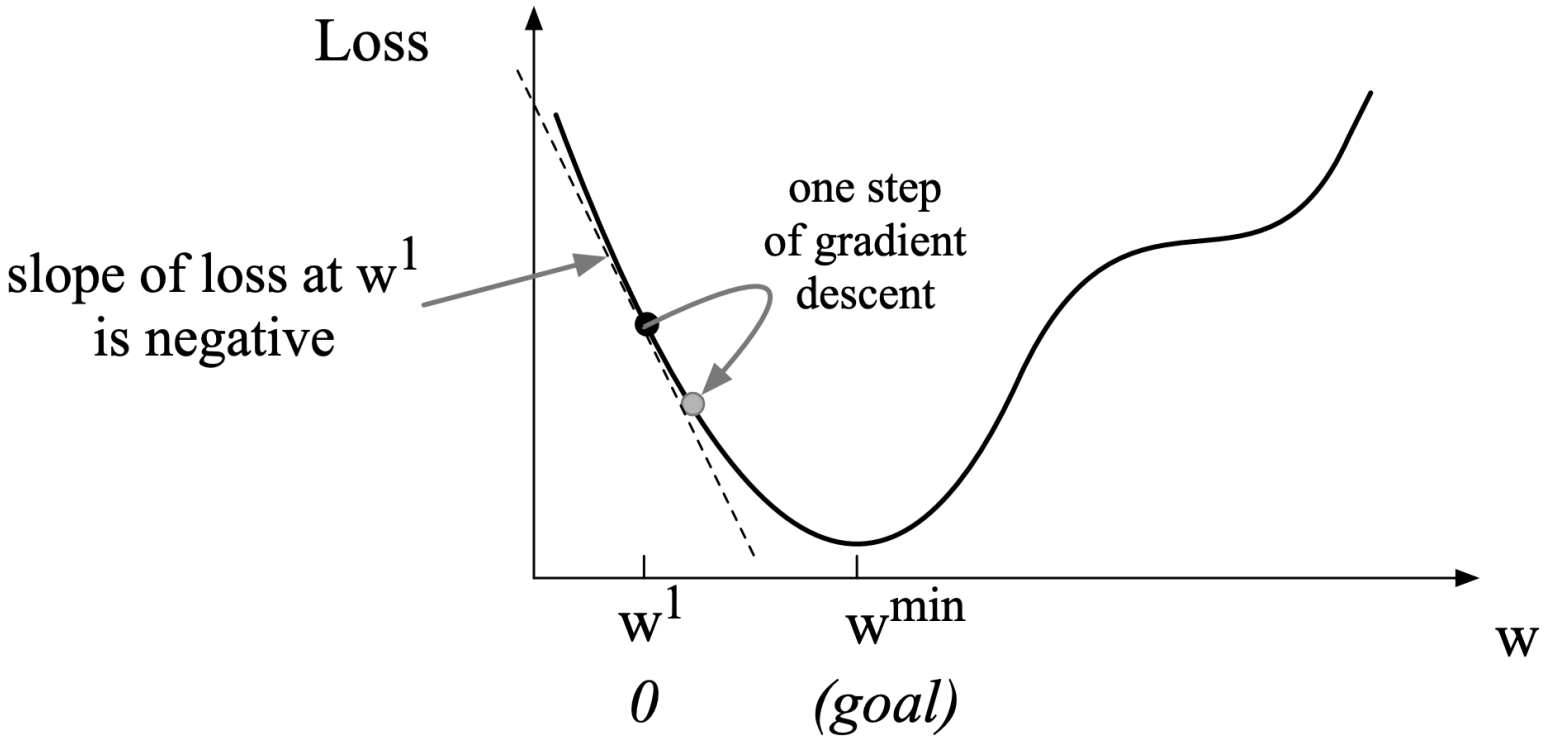
This relies on the fact that:

$$\mathbb{E}_{\mathcal{B}\sim\mathcal{D}}\nabla_\theta \ell(\theta, \mathcal{B}) = \nabla_\theta \ell(\theta, \mathcal{D})$$

This means that a gradient obtained from a mini-batch is an unbiased estimator of a gradient estimated on the full dataset!

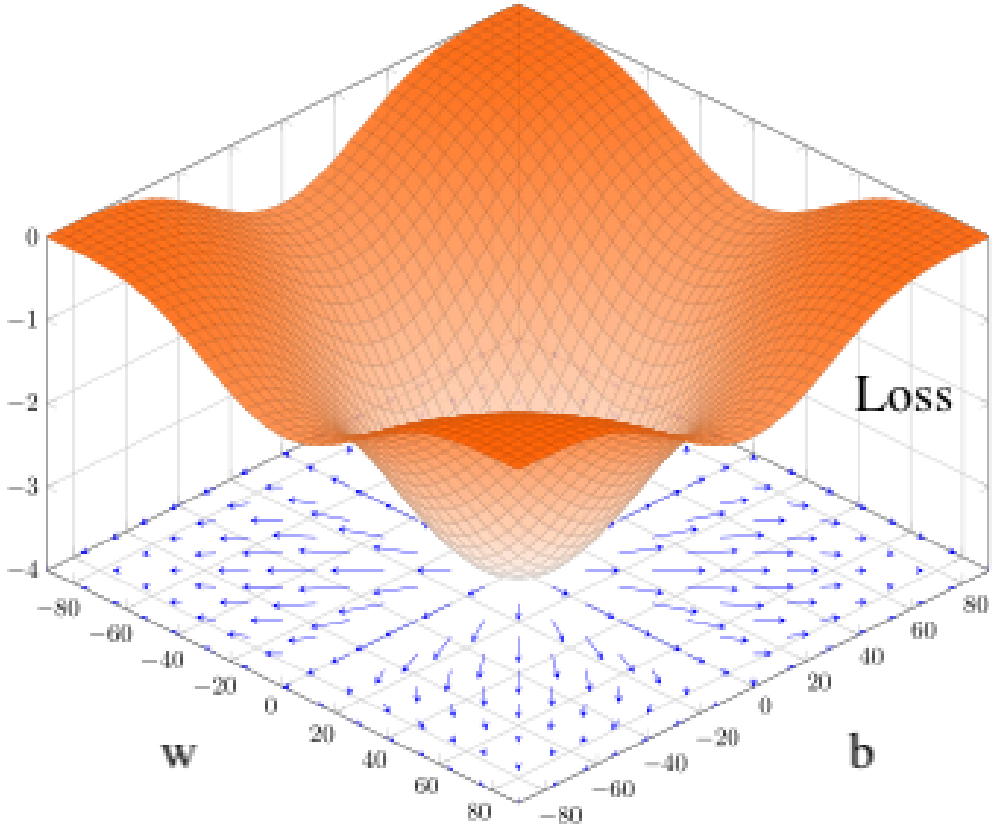
Reminder: Gradient

The gradient is the generalisation to multidimensional variables of the derivative. Consider this 1-dimensional example:



Generalising to 2 dimensions

2-dim example: a weight w and a bias b vs their loss (the gradients are the blue arrows):



Generalising to more dimensions

For a parameter space $\Theta \in \mathbb{R}^n$, the gradient becomes a vector with n dimensions pointing towards the direction of steepest ascent and with a magnitude proportional to the loss function steepness.

This vector consists of partial derivatives for the individual weights:

$$\nabla_\theta \ell(\theta, \mathcal{D}) = \begin{bmatrix} \frac{\delta}{\delta w_1} \ell(\theta, \mathcal{D}) \\ \frac{\delta}{\delta w_2} \ell(\theta, \mathcal{D}) \\ \dots \\ \frac{\delta}{\delta w_n} \ell(\theta, \mathcal{D}) \end{bmatrix}$$

Each partial derivative answers the question: how much does a small change in value for this weight affect the value of the loss function?

Reminder: Computing the gradient

Consider a 1-layer neural network:

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

The gradient for the weight W_{ij} given the class $\mathbf{y}$ can be computed as (as we have seen a couple of lectures ago):

$$\begin{aligned}\frac{\delta}{\delta W_{ij}}\ell(f(\mathbf{x})) &= -(\mathbf{y}_i - \hat{\mathbf{y}}_i)\mathbf{x}_j \\ &= -\left[\mathbf{y}_i - \frac{\exp(W_i^\top \mathbf{x} + \mathbf{b}_i)}{\sum_k \exp(W_k^\top \mathbf{x} + \mathbf{b}_k)}\right]\mathbf{x}_j\end{aligned}$$

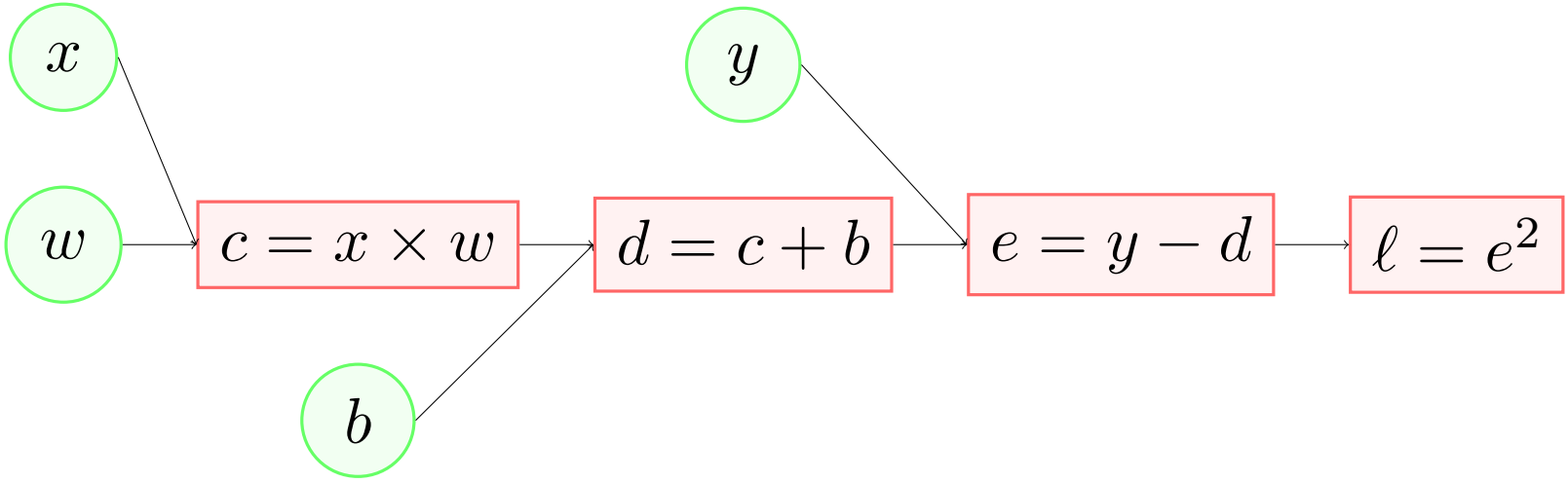
Back-propagation

- This works for this single layer network.
- Very complex to compute the partial derivative for weights of early layers in deep networks.
- **Back-propagation** (a.k.a. reverse-mode automatic differentiation), which relies on computation graphs, is an algorithm that can do it **efficiently**!

Computation Graphs

Representation of the process to compute the feed-forward pass, where **operations** (add, multiply, etc.) are nodes and **operands** are incoming edges.

Consider a 1-dimensional neuron with a Mean Squared Error loss. Its computation graph is:



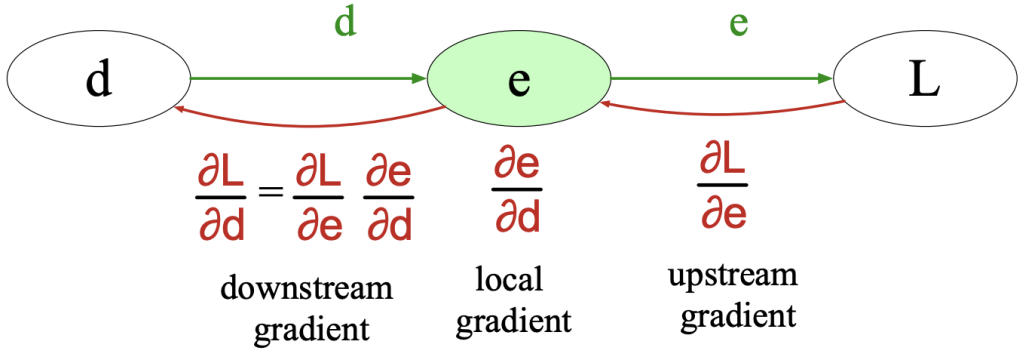
The backward pass calculates $\frac{\delta}{\delta w}\ell$ and $\frac{\delta}{\delta b}\ell$

Reminder: Chain Rule of Differentiation

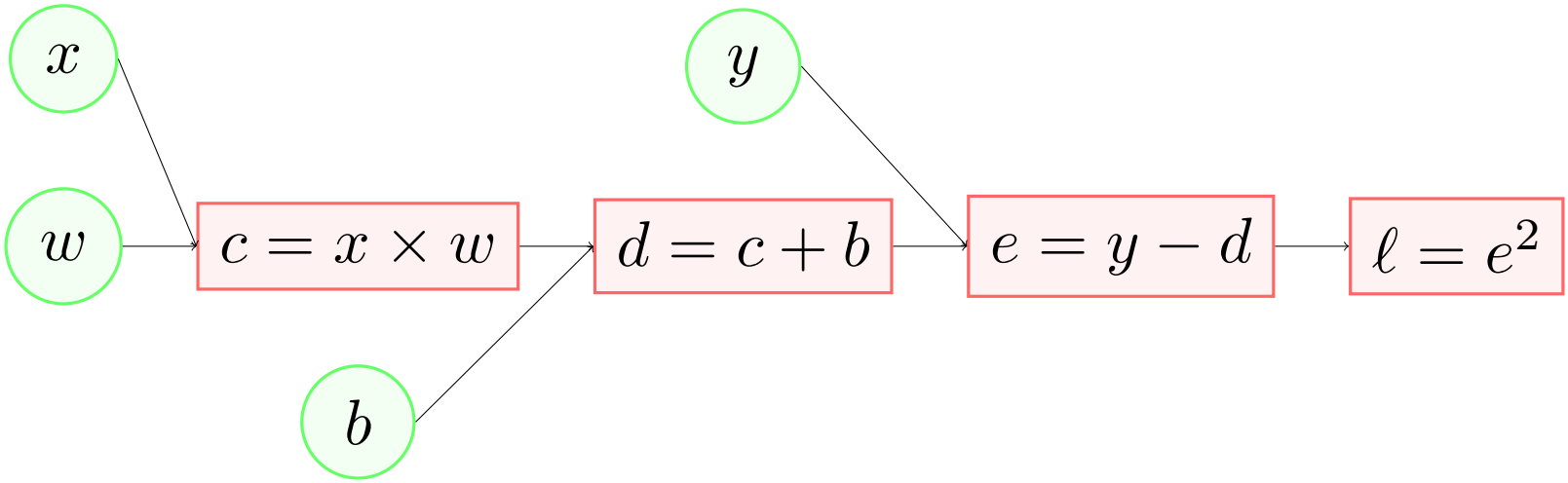
The **chain rule** of differentiation states that for a composite function $f(x) = u(v(x))$, its derivative can be decomposed as:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}u}{\mathrm{d}v} \cdot \frac{\mathrm{d}v}{\mathrm{d}x}$$

Backprop takes advantage of the chain rule by calculating the gradient of downstream nodes wrt the root ℓ as the product of the gradients of all intermediate nodes wrt their children.



Working through the example



$$\frac{\delta \ell}{\delta w} = \frac{\delta \ell}{\delta e}\frac{\delta e}{\delta d}\frac{\delta d}{\delta c}\frac{\delta c}{\delta w} = 2e \times -1 \times 1 \times x = -2(y - wx - b)x$$

$$\frac{\delta \ell}{\delta b} = \frac{\delta \ell}{\delta e}\frac{\delta e}{\delta d}\frac{\delta d}{\delta b} = 2e \times -1 \times 1 = -2(y - wx - b)$$

Useful derivatives in NNs

- $\frac{\mathrm{d}\,\mathrm{softmax}(\mathbf{x})_i}{\mathrm{d}\mathbf{x}_j} = \mathrm{softmax}(\mathbf{x})_i \cdot [\delta_i(j) - \mathrm{softmax}(\mathbf{x})_j]$
- $\frac{\mathrm{d}\,\mathrm{ReLU}(x)}{\mathrm{d}x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$

Summary so far

- Stochastic gradient descent is an optimiser that iteratively updates the parameter estimate with the (scaled) gradient from mini-batches of data.

- Back-propagation is an algorithm to calculate the gradient wrt parameters of deep networks efficiently.

A concrete example

- How do we calculate the forward and backward pass concretely?
- What is the experimental routine in practice?
- An example with multilayer percetrpon (MLP) with toy vocabulary $\mathcal{V} = \{\text{duck, goose}\}$, and binary classification

Experimental Routine

- Training
  1. Choose the hyperparameters (architecture and optimiser)
  2. Initialise the model
  3. Optimise the model with gradient descent
     (a) Sample an input–label pair from the data
     (b) Perform a forward pass to obtain a prediction
     (c) Calculate the loss between the prediction and the label
     (d) Back-propagate to get the gradient of the loss wrt parameters
     (e) Parameter update

- Evaluation
  1. Model selection on the development set
  2. Perform inference with the best model

Producing the Output: The Forward Pass

How to obtain $p(y \mid x_1, \ldots, x_n)$

1. $\mathbf{h}_0 = \mathrm{enc}(x_1, \ldots, x_n)$
2. For every layer $1 \leq l \leq L-1$:
   - $\mathbf{h}_l = a(W_l\, \mathbf{h}_{l-1} + \mathbf{b}_l)$
3. $\hat{y} = p(\cdot \mid x_1, \ldots, x_n) = \mathrm{softmax}(W_L\, \mathbf{h}_{L-1})$

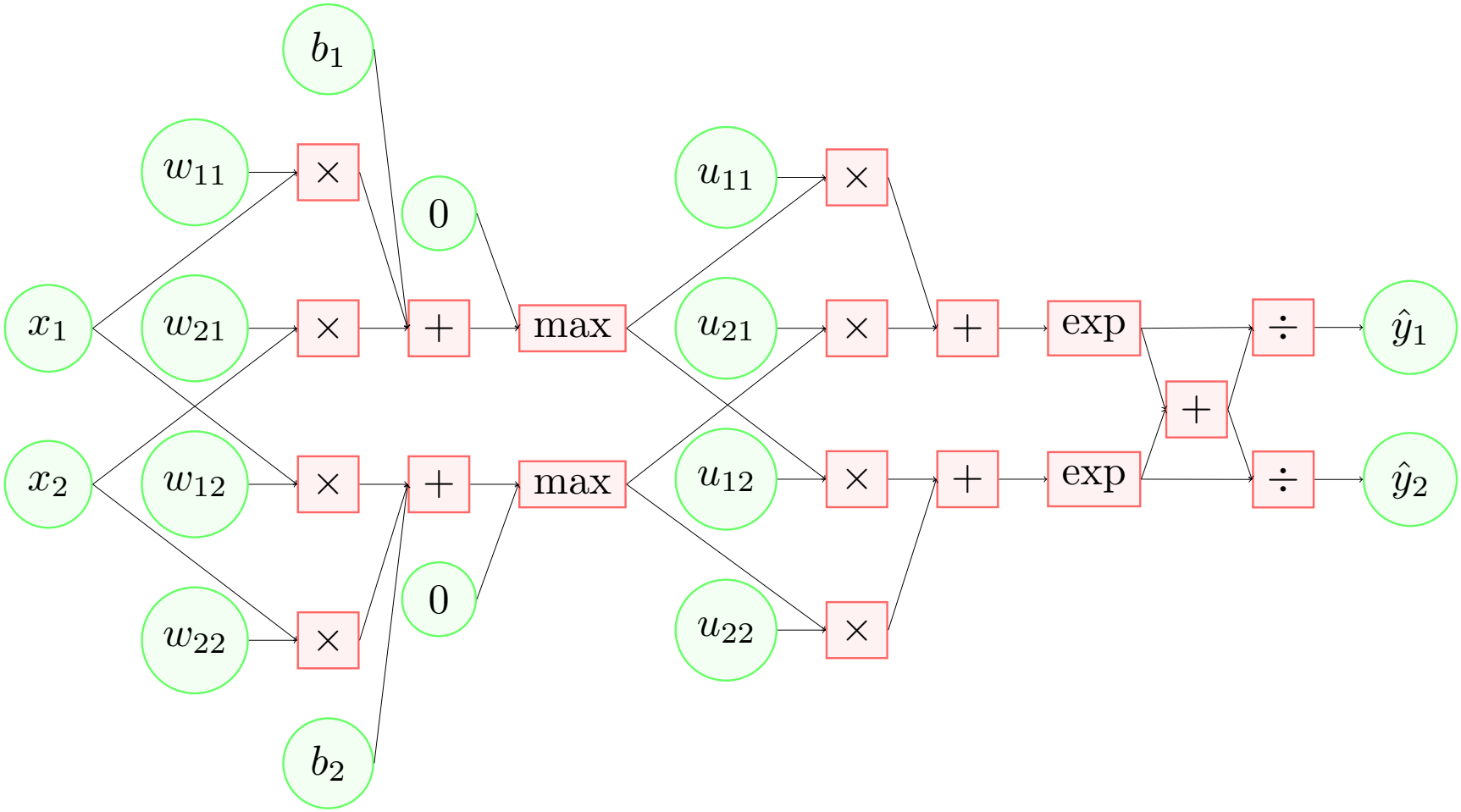
This is the forward pass for an MLP of arbitrary depth $L \in \mathbb{N}$.

Choosing the hyper-parameters

Assume that we choose the following hyper-parameters for the **architecture**: $L = 2$, $a = \mathrm{ReLU}$, embedding dimension $E \in \mathbb{R}^{2 \times |\mathcal{V}|}$ and hidden dimension $\mathbf{h} \in \mathbb{R}^2$.

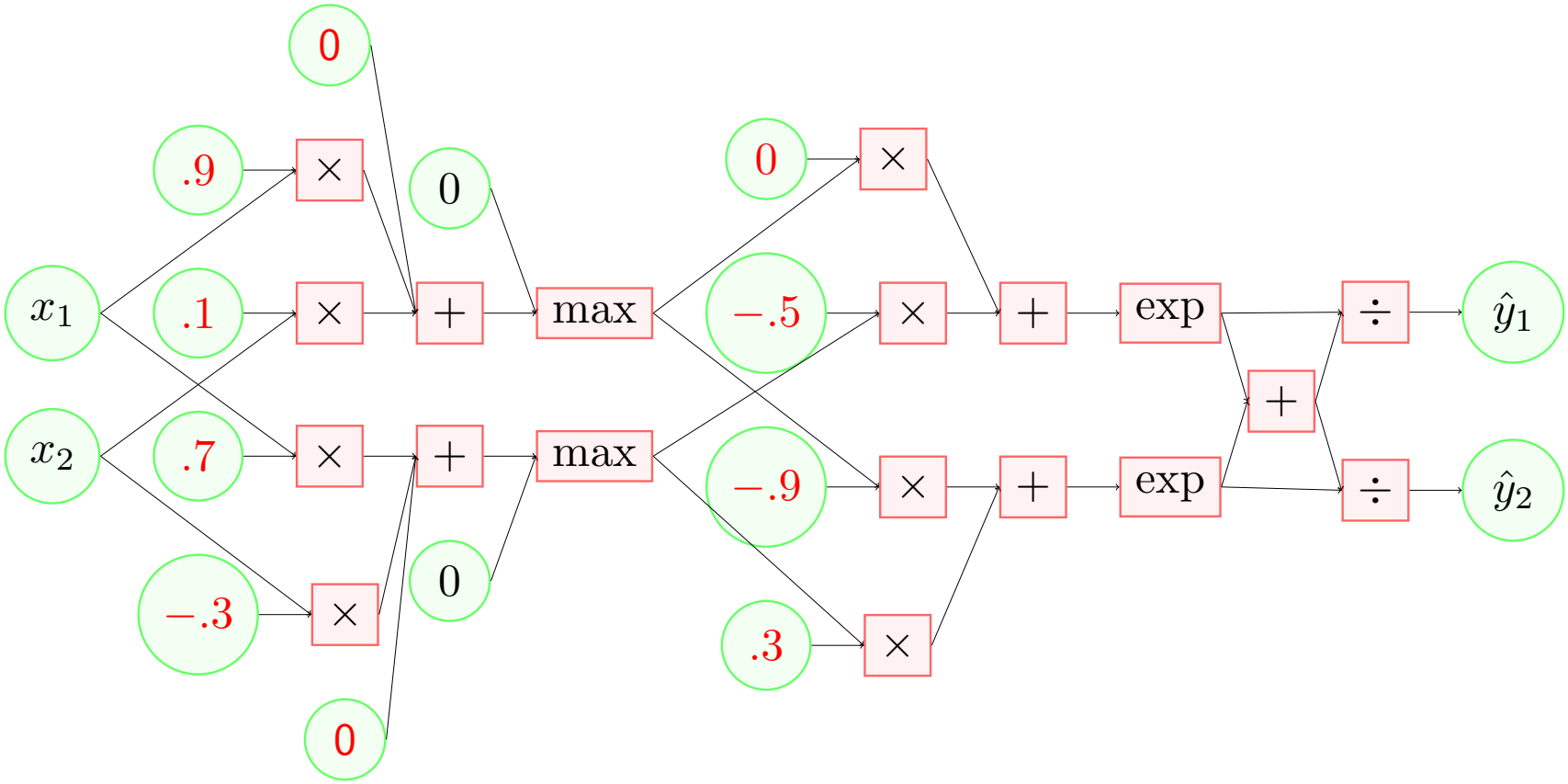
For the **optimiser**, we set the learning rate to $\eta = 10^{-2}$

Computation Graph



Initialising the model

Sample weights from $W_{ij} \sim \mathcal{U}(-1, 1)$ and set the biases to $\mathbf{b}_j = 0$



So at $t = 0$: $W = \begin{bmatrix} .9 & 1 \\ .7 & -.3 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $U = \begin{bmatrix} 0 & -.5 \\ -.9 & .3 \end{bmatrix}$, $E = \ldots$

Training iteration: encode the input

Sample an example from the train data, e.g. "duck duck goose"

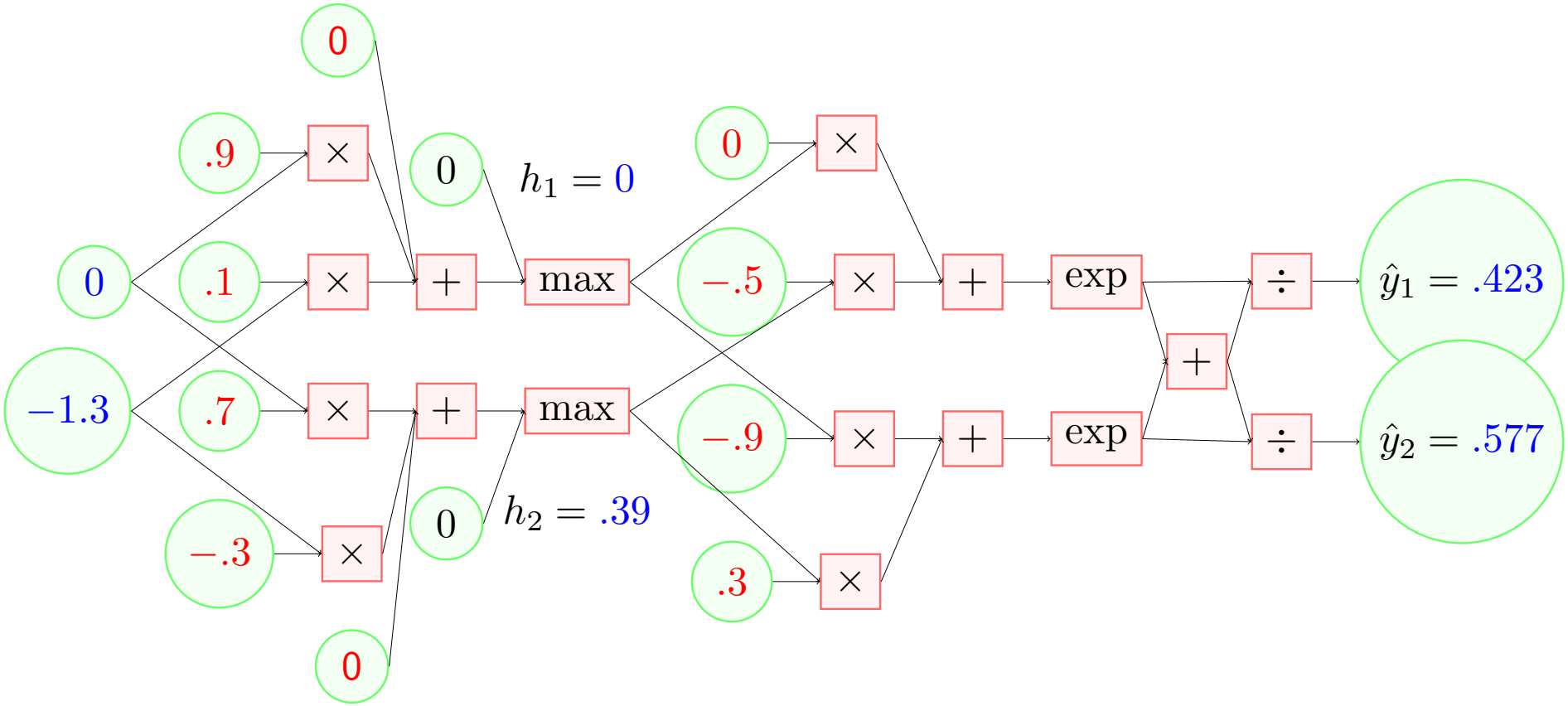
Estimate $p(\mathbf{y} \mid \text{duck duck goose})$. In this example, $\mathbf{y} = [1, 0]$ (e.g., 'text about animals')

$$E = \begin{array}{c} \begin{array}{cc} \text{duck} & \text{goose} \end{array} \\ \begin{bmatrix} 0.05 & -0.1 \\ -1.5 & -0.9 \end{bmatrix} \end{array}$$

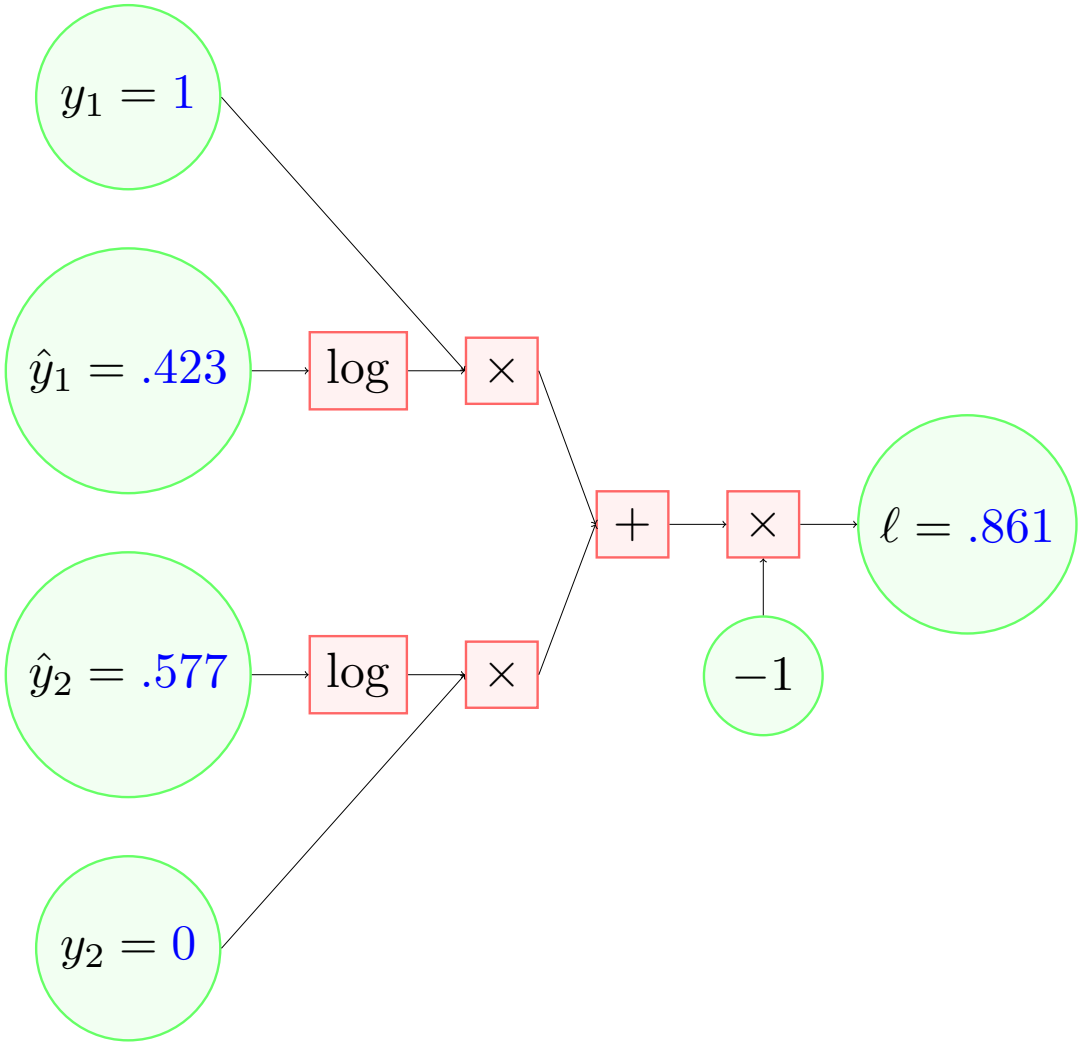
A Bag-of-words encoder (average token embedding):

$$\text{enc}(\textit{duck duck goose}) = \frac{1}{3}\left(\begin{bmatrix} 0.05 \\ -1.5 \end{bmatrix} + \begin{bmatrix} 0.05 \\ -1.5 \end{bmatrix} + \begin{bmatrix} -0.1 \\ -0.9 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ -1.3 \end{bmatrix}$$

Training iteration: forward pass



Training iteration: calculate the loss



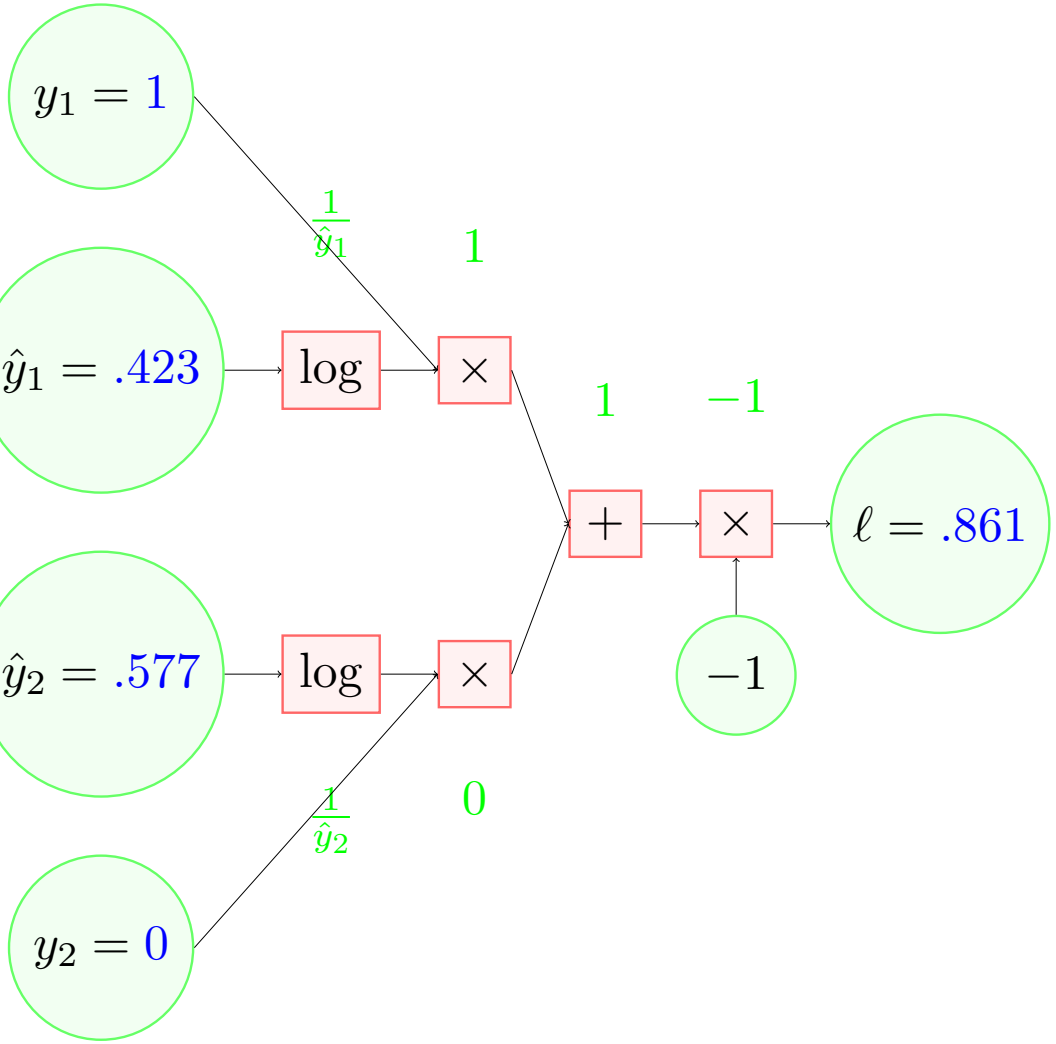
$$\ell = -\sum_{y \in C} p_{\theta^\star}(y) \log p_{\hat{\theta}}(y)$$

Training iteration: calculate the gradient

What is $\nabla_\theta \ell = \nabla_{[E,W,\mathbf{b},U]} \ell$? E.g. to find $\frac{\delta \ell}{\delta_U}$, first get $\frac{\delta \ell}{\delta_{\hat{y}_1}}$ and $\frac{\delta \ell}{\delta_{\hat{y}_2}}$

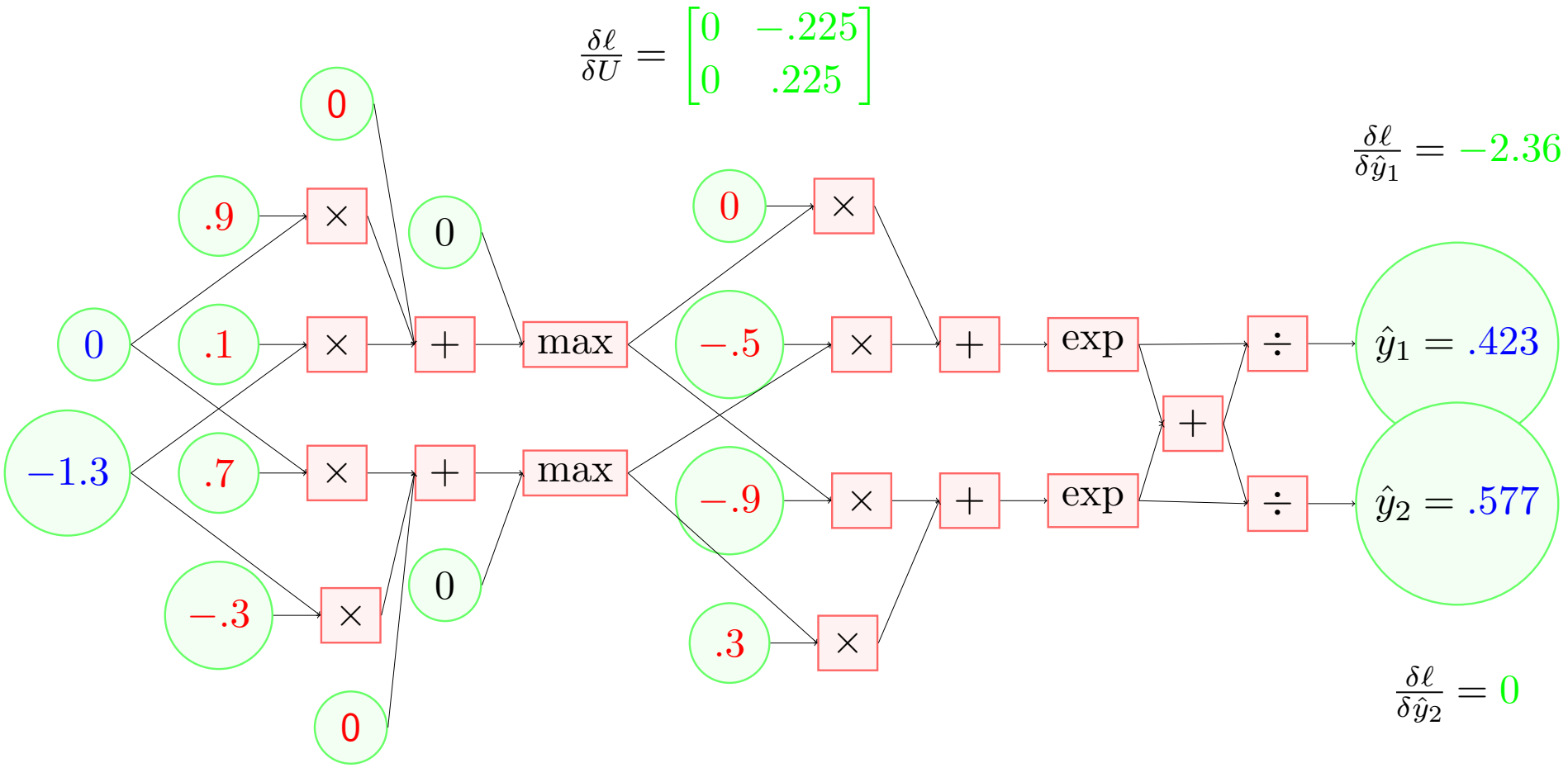
$\frac{\delta \ell}{\delta \hat{y}_1} = -1 \times 1 \times 1 \times \frac{1}{\hat{y}_1} = -2.36$

$\frac{\delta \ell}{\delta \hat{y}_2} = -1 \times 1 \times 0 \times \frac{1}{\hat{y}_2} = 0$



Training iteration: calculate the gradient

From $\hat{y}_1$ and $\hat{y}_2$, continue to back-propagate to find $\frac{\delta \ell}{\delta_U}$ (if a node has multiple parents, such as $\exp$, sum over all the incoming edges)



Training iteration: parameter update

For the parameter U, SGD performs the following update:

$$
\begin{aligned}
U_{t=1} &= U_{t=0} - \eta \frac{\delta \ell}{\delta U} \\
&= \begin{bmatrix} 0 & -.5 \\ -.9 & .3 \end{bmatrix} - 10^{-2} \times \begin{bmatrix} 0 & -.225 \\ 0 & .225 \end{bmatrix} \\
&= \begin{bmatrix} 0 & -.502 \\ -.9 & .297 \end{bmatrix}
\end{aligned}
$$

Similarly, we will update all the other model parameters $E, W, \mathbf{b}$.

We will repeatedly perform training iterations until the stopping criterion is met (e.g. after k iterations).

Model Selection on the Dev Set

Our choice of hyper-parameters for the model architecture and optimiser at the start were arbitrary.

Grid search defines ranges for each hyper-parameter, then trains multiple models, one per config (from their Cartesian product).

E.g., if $L = \{2, 3\}$ and $\eta = \{10^{-2}, 10^{-3}\}$, it will run $[L = 2, \eta = 10^{-2}]$, $[L = 3, \eta = 10^{-2}]$, $[L = 2, \eta = 10^{-3}]$, $[L = 3, \eta = 10^{-3}]$.

We then compare the performance of these models on the dev set, and select the best one with parameters $\hat{\theta}$.

Next Lectures

- Friday, next Tuesday: Distributional semantics - inducing 'meanings' of word from context
- next Wed: Language modeling (classic approach)
- Friday: back to neural networks, we will be moving from text classification into language modeling and text generation