ECE521 Lecture 7/8

Logistic Regression

UNIVERSITY OF
TORONTO

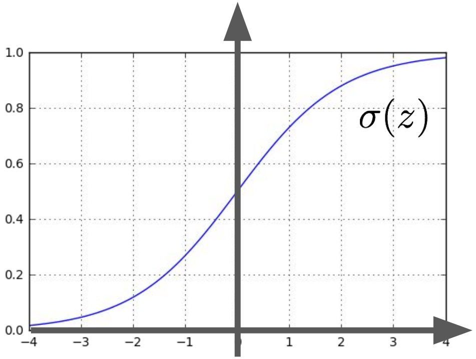
Outline

- **Logistic regression (Continue)**
- A single neuron
- Learning neural networks
- Multi-class classification

Logistic regression

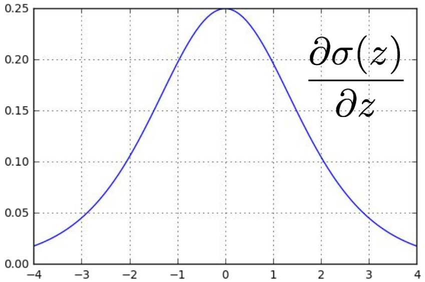
- The output of a logistic regression model is a sigmoid function of a weighted sum of its inputs:

$$\hat{y}^{(m)} = \sigma(z^{(m)}) = \sigma(W^T \mathbf{x}^{(m)} + b)$$



- Recall the sigmoid function and its nice derivatives:
  - The sigmoid output is bounded between 0 and 1

$$\sigma(z) = \frac{1}{1 + e^{-z}} \qquad \frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$$



Logistic regression

- We may choose to train logistic regression model using a squared L2 loss function:

$$\mathcal{L} = \frac{1}{2}\sum_m \|\hat{y}^{(m)} - t^{(m)}\|_2^2 \qquad \hat{y}^{(m)} = \sigma(z^{(m)}) = \sigma(W^T\mathbf{x}^{(m)} + b)$$

- The gradient of the loss function w.r.t. W or b can be obtained easily using chain-rule of calculus:

Gradient w.r.t. the weight vector

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m \frac{\partial \mathcal{L}}{\partial \hat{y}^{(m)}} \frac{\partial \hat{y}^{(m)}}{\partial z^{(m)}} \frac{\partial z^{(m)}}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)})\hat{y}^{(m)}(1 - \hat{y}^{(m)})\mathbf{x}^{(m)}$$

Gradient w.r.t. the bias?? (your own exercise)

Logistic regression

- Using a squared L2 loss function to train logistic regression models has a major flaw:
  - If the model parameters are poorly initialized and the model is making nearly binary predictions at the first iteration of a gradient descent algorithm, then learning will happen really slowly using squared L2 loss because of the vanishing gradient from the sigmoid function.
  - One way to see this is to have a model

$$\mathcal{L} = \frac{1}{2} \sum_m \|\hat{y}^{(m)} - t^{(m)}\|_2^2$$

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)}) \hat{y}^{(m)} (1 - \hat{y}^{(m)}) \mathbf{x}^{(m)}$$

Cross-entropy loss function

- Cross-entropy is a better loss function.

$$\mathcal{L} = \sum_m -t^{(m)} \log \sigma(z^{(m)}) - (1 - t^{(m)}) \log(1 - \sigma(z^{(m)}))$$

- Take the gradient w.r.t. W. The gradient under the cross-entropy loss is the same as the gradient for the linear regression model!

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)}) \mathbf{x}^{(m)}$$

where, $\hat{y}^{(m)} = \sigma(z^{(m)})$

$= \sigma(W^T \mathbf{x}^{(m)} + b)$

Learning logistic regression

- What is happening during learning?

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{m} (\hat{y}^{(m)} - t^{(m)}) \mathbf{x}^{(m)}$$

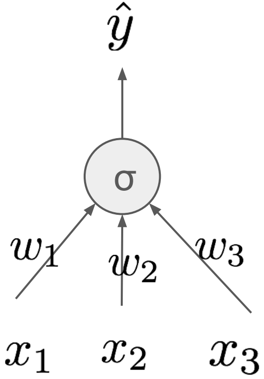
The gradient is the correlation between **the error** and **the inputs**

- In what circumstances is the gradient zero? i.e. the model stops learning any new information. Either:
  - All the individual gradients are zero (perfectly separable case), or
  - The gradients from different training examples cancel out (most likely scenario)

Learning logistic regression

- The gradient reflects the correlation between the mistakes and the input features.
  - After learning, the values of the individual weights indicate the importance of its input to the final prediction
  - If an input feature Xn is positively correlated with the target label, its weight will be a large positive value

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)})\mathbf{x}^{(m)}$$



Learning logistic regression for movie review

- Convert a text string into reasonable input-feature vectors:
    - The bag-of-words representation:
    - Count the frequency of a word appearing from a preset vocabulary

[good, fantastic, …, terrible, disappointed, awesome,...]

[2, 0, …, 0, 1, 5, ...]

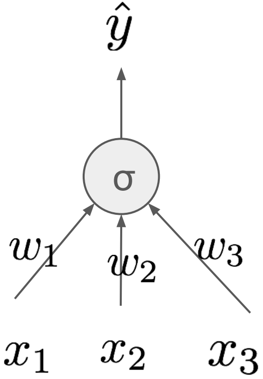
Learning logistic regression for movie review

- Consider a positive word that correlates to the positive review of a movie:
  - Assume the prediction starts at random 50% random guessing.
  - The weight of the positive word should increase during learning

[good, fantastic, …, terrible, disappointed, awesome,...]

[2, 0, …, 0, 1, 5, ...]

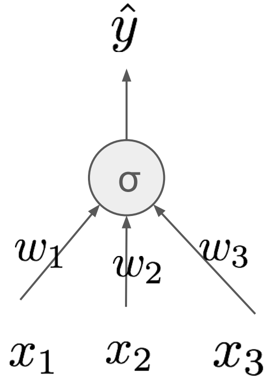
$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)})\mathbf{x}^{(m)}$$



Learning logistic regression for movie review

- Consider a random word that appears frequently and appears in both positive and negative review:
  - Its weight will likely to be around zero.
  - The gradient is not very informative and the error will cancel out between the positive class and negative class.

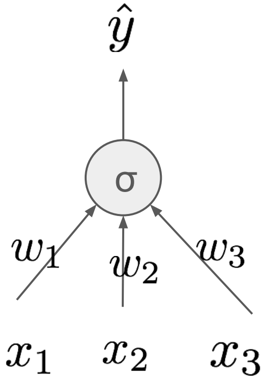
$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)})\mathbf{x}^{(m)}$$



Learning logistic regression for movie review

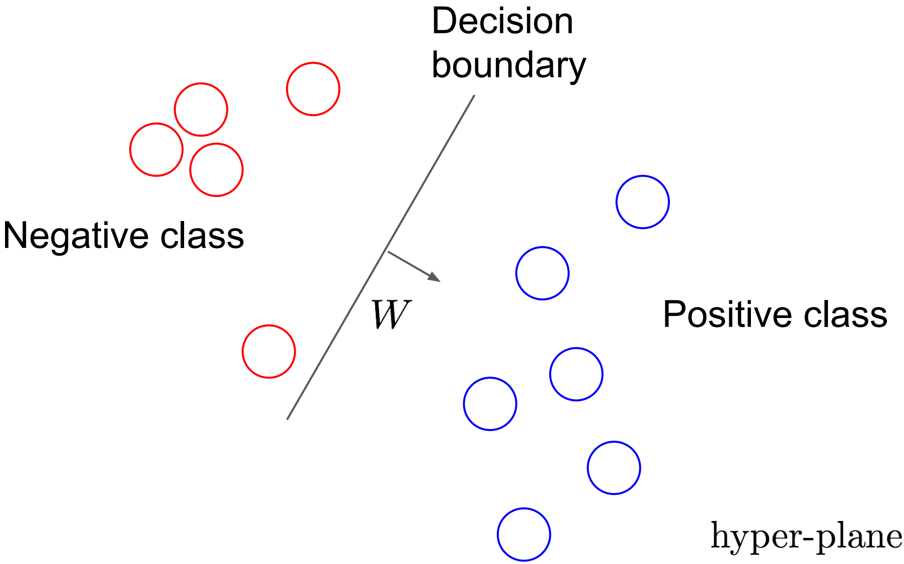
- Consider a random word that appears rarely and only appears in the positive review:
  - The model will likely not perform well because it is overly confident about a rare instance.
  - MAP should fix this problem

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_m (\hat{y}^{(m)} - t^{(m)})\mathbf{x}^{(m)}$$



Intuitive geometry of logistic regression

- Decision boundary of logistic regression
    - The weight vector is perpendicular to the decision boundary



$$P(t=1|\mathbf{x},W) = P(t=0|\mathbf{x},W)$$

$$\sigma(W^T\mathbf{x}+b) = 1-\sigma(W^T\mathbf{x}+b)$$

$$\text{hyper-plane of a decision boundary: } W^T\mathbf{x}=0$$

Concept in the course so far

- **Problem formulations:**
  - i.i.d., different distance functions, squared L2 loss, cross-entropy loss, MLE, MAP, weight-decay regularizer

- **Learning algorithms:**
  - gradient descent, stochastic gradient descent, momentum

- **Models:**
  - linear regression, logistic regression, k-NN (no learning required)

- **Some theoretical results**
  - Provide some additional intuition: how to pick the optimal regressor, optimal decision rules (how to set the threshold/decision boundary), expected loss

Outline

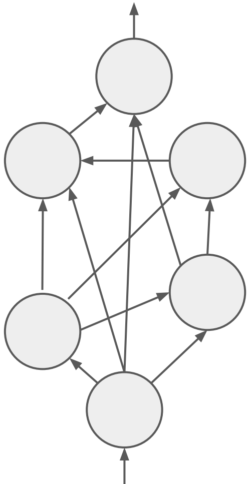
- **Logistic regression**
- Learning objectives:
    - Logistic/sigmoid function and its derivatives
    - Cross-entropy loss function and its derivatives
    - Probabilistic interpretation (assignment 2)

Outline

- Logistic regression (Continue)
- **A single neuron**
- Learning neural networks
- Multi-class classification

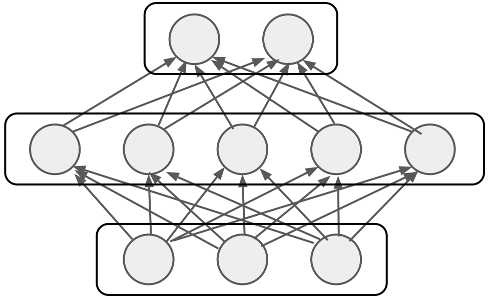
Neural networks

- Neural networks are flexible computation models that consist of many smaller computational modules called neurons or hidden units:
  - Neural networks are like continuous real-valued electrical circuits.
  - It is very modular and some special modules are designed for reusability and abstraction.
  - All continuous functions are neural networks.
  - All the learnt knowledge of a neural network is stored in its weight connections; it is also called “connectionism”
    (a name popular before the AI Winter)



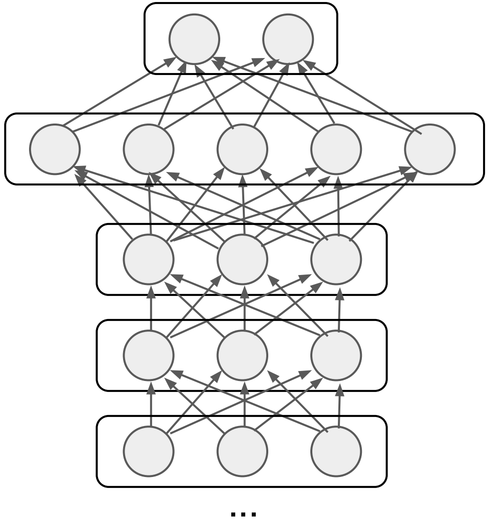
Neural networks

- One very useful abstraction is the concept of a “layer”:
    - A hidden layer is a group of hidden units that have connections one layer above and one layer below.
    - There is no inter-layer connection among the hidden units within a layer.
    - This abstraction is computationally efficient because all the hidden units within a layer can be computed in parallel.



Neural networks

- Deep learning typically refers to a neural network with more than three hidden layers.
  - Deep neural networks can mathematically represent any continuous function given enough layers, but they also require additional tricks to learn useful representations for any tasks.
  - They work really well in supervised learning given enough data.
  - A deep neural network is like a complex system in biology: we understand a lot about what the simple module does, but it quickly becomes really hard to understand what the system does, i.e. a “black box”.

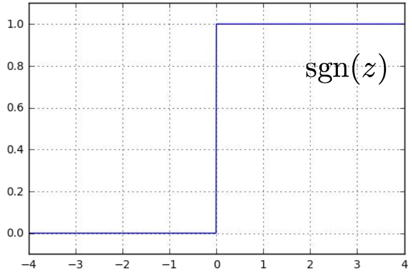


An artificial neuron

- An artificial neuron is a simple computation unit that receives inputs from other simple computation units:
    - The effect of each input on the final output of the neuron is controlled by a weight
    - The weights can be positive or negative values for encoding respectively positive or negative contributions from the inputs

An artificial neuron

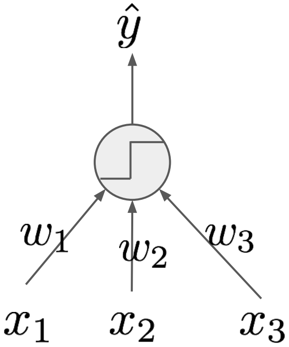
- An artificial neuron is a simple computation unit that receives inputs from other simple computation units:
  - The effect of each input on the final output of the neuron is controlled by a weight
  - The weights can be positive or negative values for encoding +ve or -ve contributions from the inputs
  - A weighted sum of the inputs was first proposed by McCulloch-Pitts (1943)



$$\mathrm{sgn}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

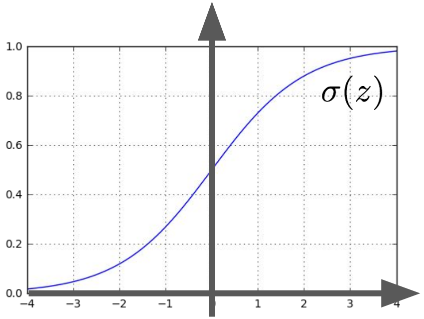
$$\hat{y} = \mathrm{sgn}(\sum_n w_n x_n + b)$$

$$\frac{\partial sgn}{\partial z} = 0, \forall z \neq 0$$



Some simple neurons: sigmoid neurons

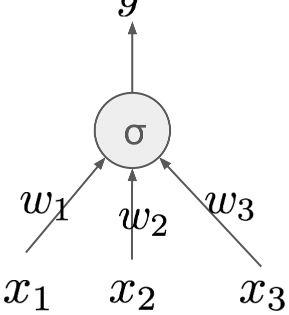
- Instead of using a hard step function, a soft, smooth and differentiable step function is desirable if we are going to use the gradient descent algorithm to learn our model:
  - Sigmoid neurons can be thought of as soft thresholding units.
  - Logistic regression models are simply neural networks with a single logistic neuron.



$$\sigma(z) = \frac{1}{1+e^{-z}}$$

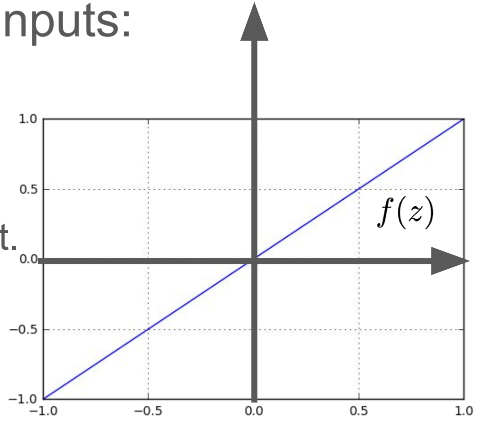
$$\frac{\partial \sigma}{\partial z} = \sigma(z)(1-\sigma(z))$$

$$\hat{y} = \sigma(\sum_n w_n x_n + b)$$



Some simple neurons: linear neurons

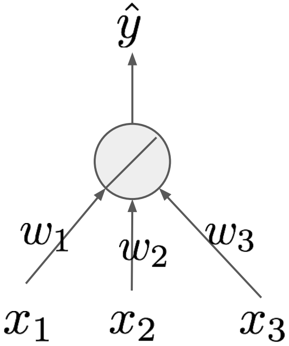
- A Linear neuron directly outputs the weighted sum of the inputs:
  - Linear regression is the simplest neural network with a single linear neuron.
  - It has a constant partial derivative which is great for gradient descent.
  - However, stacking layers of linear neurons does not increase the representational power of a model. Nonlinearity is important for building richer and more flexible models.



$$f(z) = z$$

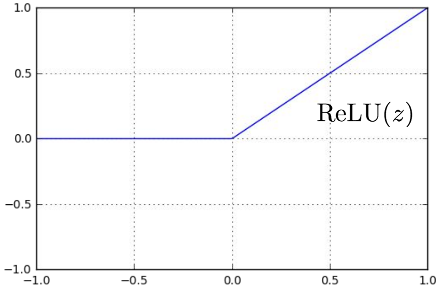
$$\frac{\partial f}{\partial z} = 1$$

$$\hat{y} = \sum_n w_n x_n + b$$



Some simple neurons: rectified linear units (ReLU)

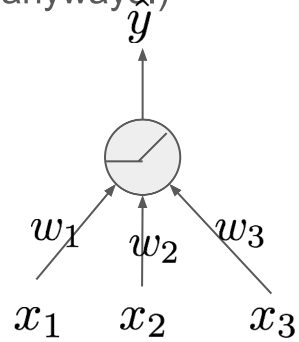
- Linear neurons can easily be modified to exhibit nonlinear behaviors:
  - The non-positive value are forced to be zero.
  - The ReLU neurons still have a very nice constant gradient if the weighted sum of the inputs is positive.
  - It is mathematically non-differentiable at zero, but we ignore that and use gradient descent anyways. It will work brilliantly well. (numerically, we will never get exactly zero summed inputs anyways.)



$$\mathrm{ReLU}(z) = \max(0, z)$$

$$\hat{y} = \mathrm{ReLU}\Big(\sum_n w_n x_n + b\Big)$$

$$\frac{\partial \mathrm{ReLU}}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

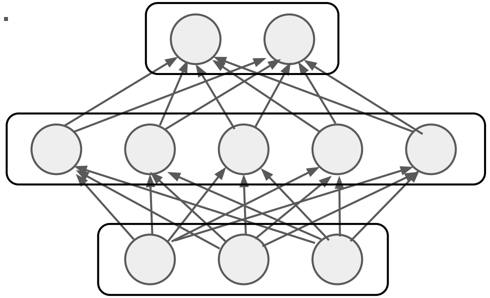


Outline

- Logistic regression (Continue)
- A single neuron
- **Learning neural networks**
- Multi-class classification

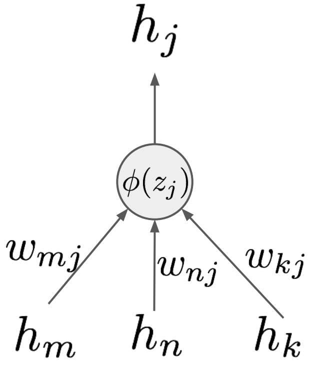
Neural networks

- There are two ways to solve a problem:
  1. Hire the most ingenious software engineers to hard-code a program.
  2. Gather a huge dataset and learn the program from the data.
  - Deep neural networks avoid time-consuming feature engineering by hand, and as the datasets grow larger they can discover better and better features with no human intervention.
  - Neural networks can also be understood as a form of *adaptive basis function model* where the model learns layers of basis functions. The activation function used for a neuron is similar to the nonlinear basis functions.



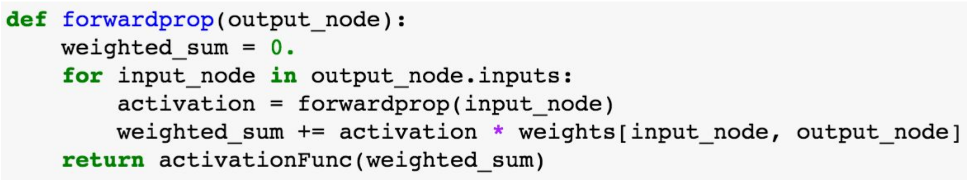
Notations for neural networks

- The model now consists of many artificial neurons wired together into a large network. For clarity, we will use the following notation for our algorithms:
    - The output of a neuron or the hidden activation is denoted as *h*
    - Scalar weight connections are indexed by the two neurons it connects
    - The input to the network is denoted *x*
    - The output of the network is denoted as $\hat{y}$
    - The element-wise hidden activation function or the activation function or nonlinearity, denoted as $\phi(\cdot)$, is the nonlinear transformation for the weighted sum of the inputs of a neuron, e.g. sigmoid, ReLU...
    - The weighted sum of a neuron's inputs is denoted as *z*



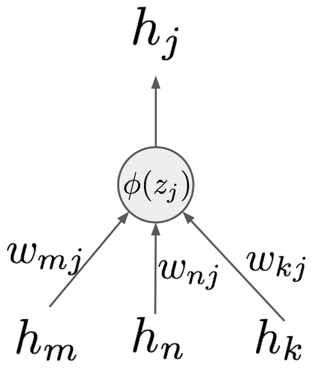
Forward propagation

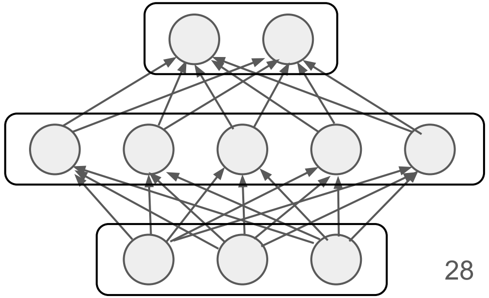
- Forward propagation computes all the hidden activations *h* and the output of the neural network $\hat{y}$
  - $h_j = \phi(z_j) = \phi(\sum_n w_{nj} h_n + b_j)$
  - This requires computing all the hidden activations that are the inputs to the current hidden units.
  - The forward propagation can be written as a recursive algorithm:



```
def forwardprop(output_node):
    weighted_sum = 0.
    for input_node in output_node.inputs:
        activation = forwardprop(input_node)
        weighted_sum += activation * weights[input_node, output_node]
    return activationFunc(weighted_sum)
```

  - The naive recursive algorithm is bad because there are a lot of redundant computations. We would like to cache the appropriate intermediate values and reuse them.

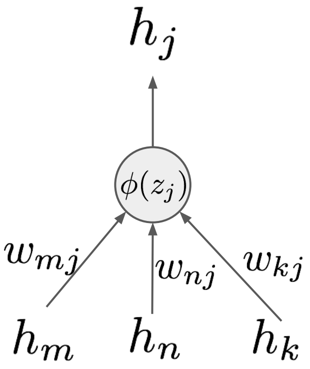


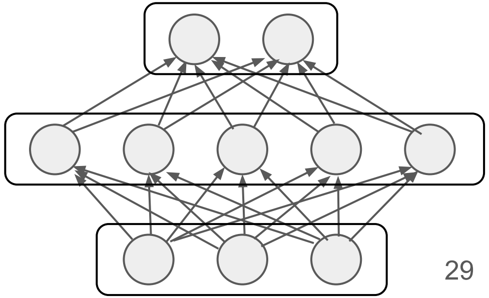


Back-propagation

- Back-propagation (Rumelhart, Hinton and Williams, 1986) is a dynamic programming method to reuse previous computations when computing the gradient of some variable using the chain rule from calculus.
  - $h_j = \phi(z_j) = \phi(\sum_n w_{nj} h_n + b_j)$
  - In its simplest form: $\frac{\partial \mathcal{L}}{\partial w_{nj}} = \frac{\partial \mathcal{L}}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial w_{nj}}$
  - $\frac{\partial \mathcal{L}}{\partial h_j}$ can be further expanded until the output of the neural network.
  - The key observation here is that the gradient of a connection is a product between the input and the partial derivative of the weighted sum of that neuron.

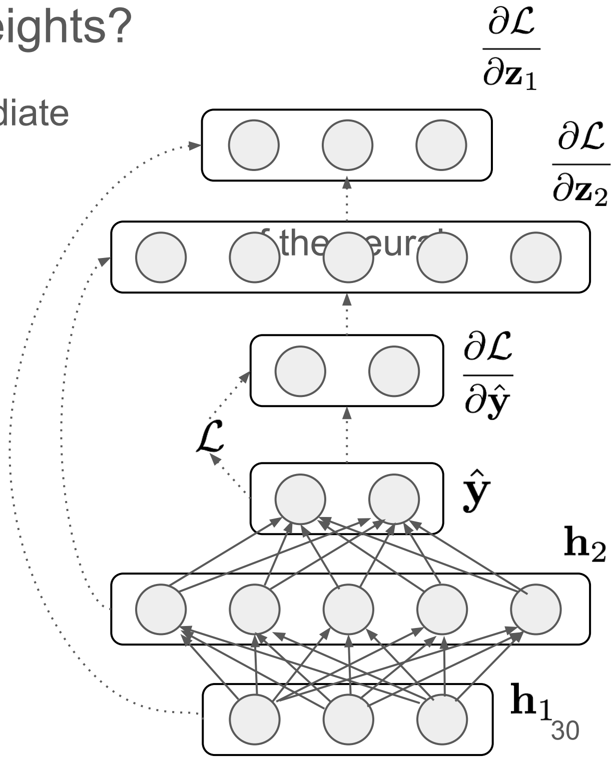
$$\frac{\partial \mathcal{L}}{\partial w_{nj}} = \frac{\partial \mathcal{L}}{\partial z_j} h_n$$





Back-propagation

- What do we need to compute the gradients of the weights?
  - First, we need to do a forward pass and cache all the intermediate hidden activations.
  - Differentiate the loss function w.r.t. the output network as the initial step for back-propagation
  - The intermediate hidden activations are needed for the partial derivative of the weighted sums


$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2}$
$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}}$
$\mathcal{L}$
$\hat{\mathbf{y}}$
$\mathbf{h}_2$
$\mathbf{h}_1$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{z}_1}$$

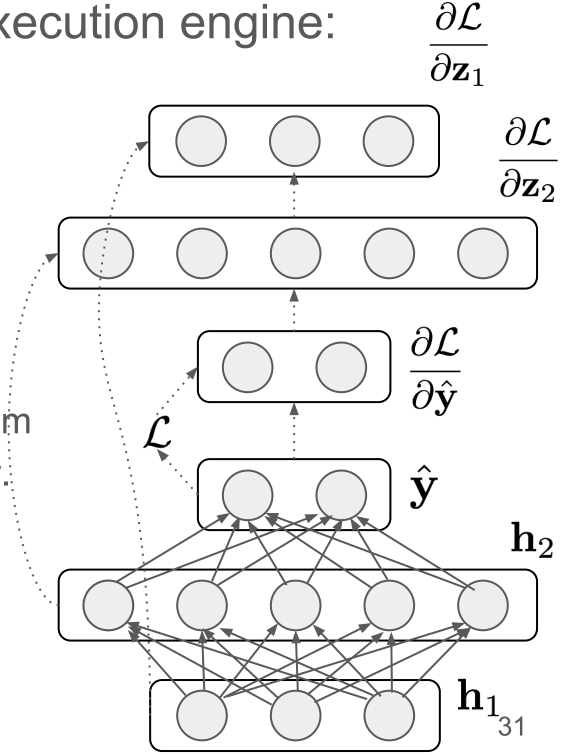
Back-propagation (left to right)

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_1} \mathbf{h}_1^T$$

Weight matrix gradient is an outer product

TensorFlow, back-propagation and auto-diff

- TensorFlow at its core is a forward/back-propagation execution engine:
  - The computation graphs are neural networks
  - The automatic differentiation executes back-propagation for the variables (weights) in the computation graph. It can be automated if the partial derivatives of each math operator are pre-defined.
  - session.run() or eval() runs the forward/back-propagation algorithm and caches the needed intermediate computation results for later.
  - The TensorFlow framework also computes the independent computations in parallel asynchronously.



$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2}$
$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}}$
$\mathcal{L}$
$\hat{\mathbf{y}}$
$\mathbf{h}_2$
$\mathbf{h}_1$

Outline

- Logistic regression (Continue)
- A single neuron
- **Learning neural networks**
- Multi-class classification