DENSE NEURAL NETWORK CLASSIFIERS

INF5860 — Machine Learning for Image Analysis

Ole-Johan Skrede

07.02.2018

University of Oslo

OUTLINE

- Introduction and motivation
- Representation and architecture of a neural network
- Forward propagation
- Cross entropy loss
- Optimization
- Backward propagation
- Vectorisation

ABOUT TODAY

- We are going through a lot today
- A lot of concepts and notation to be familiar with
- Requires some knowledge of statistics, linear algebra, and calculus
- It is not expected that you understand everything after today
- But this lecture should be sufficiently self-contained: with some work, you should be able to
  - Know how a classification neural network is built up from scratch
  - Know why some things are as they are
  - Know how to efficiently implement a vanilla classifier
- Concepts from todays lecture is the basis for the rest of the course

INTRODUCTION AND MOTIVATION

Machine learning

- A set of methods and algorithms that solves a task by learning from experience on observed data.
- Traditionally, requires preprocessing to generate feature representations of data

Representation learning

- The method itself extracts useful features (data representations)
- Requires little or no preprocessing of input data

Deep learning

- Multiple iterations of representation learning
- Hierarchical structure:
  - Learn representation of input
  - Learn representation of representation
  - Learn representation of representation
  - ...
- From low level to high level features

- Given a training set with input x and desired output y

$$\Omega_{\text{train}} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$$

- Create a function f that "approximates" this mapping

$$f(x) \approx y, \quad \forall (x, y) \in \Omega_{\text{train}}$$

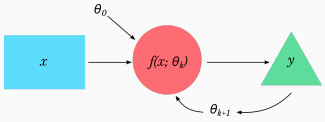
- Hope that this generalises well to unseen examples, such that

$$f(x) = \hat{y} \approx y, \quad \forall (x, y) \in \Omega_{\text{test}}$$

  where Ω_{test} is a set of relevant unseen examples.
- Hope that this is also true for all unseen relevant examples.

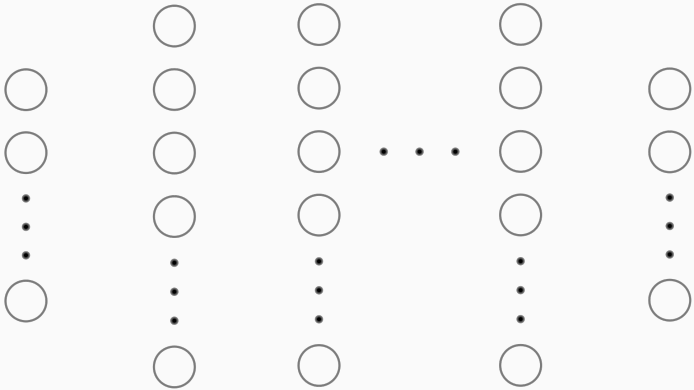
WHAT ARE WE GOING TO DO TODAY

1. Build a function f that maps input to output
   - Input: Array of numbers.
   - Output: Probability mass function conditional on observed input.
2. This function will have multiple layers, where each layer is a representation of the previous.
3. Measure how well the output of the function is approximating the true output
4. Use information from the error to update the function
5. Repeat step 3 and 4 with multiple training examples

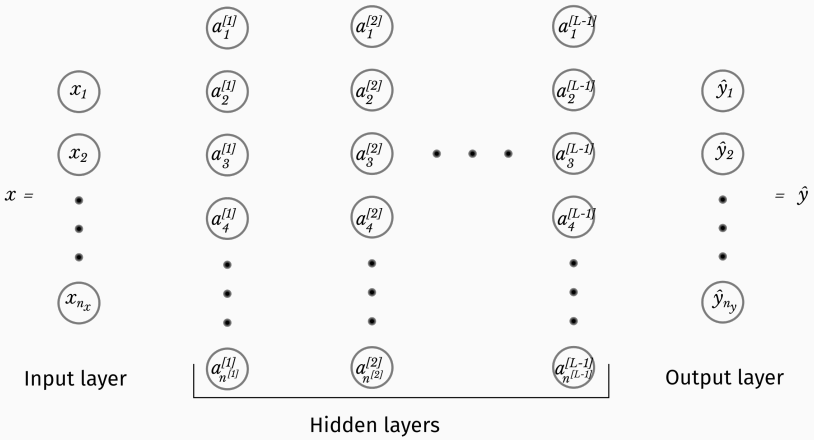


REPRESENTATION OF THE NETWORK

BASE ARCHITECTURE



NODES AND LAYERS



WHAT IS HAPPENING IN THE HIDDEN LAYER NODES

- $a_k^{[l]}$ is the *activation* of node k in layer l

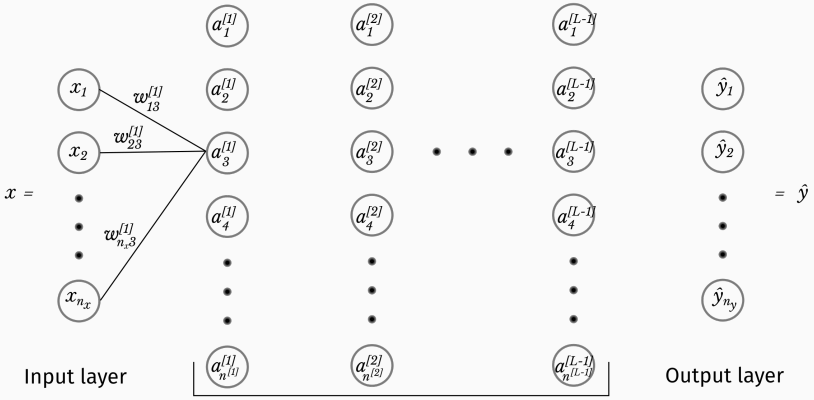
$$a_k^{[l]} = g\left(\sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}\right)$$

- $w_{jk}^{[l]}$ is the *weight* from node j in layer $l-1$ to node k in layer l
- $b_k^{[l]}$ is the *bias* of node k in layer l
- All above are scalars
- g is some non-linear function
- All w and b are "trainable", and will be adjusted according to some optimization routine
- By convention
  - $a_k^{[0]} = x_k$
  - $a_k^{[L]} = \hat{y}_k$
  - The network have L layers (we do not include the input layer in the count)

NOTATION — FOR REFERENCE

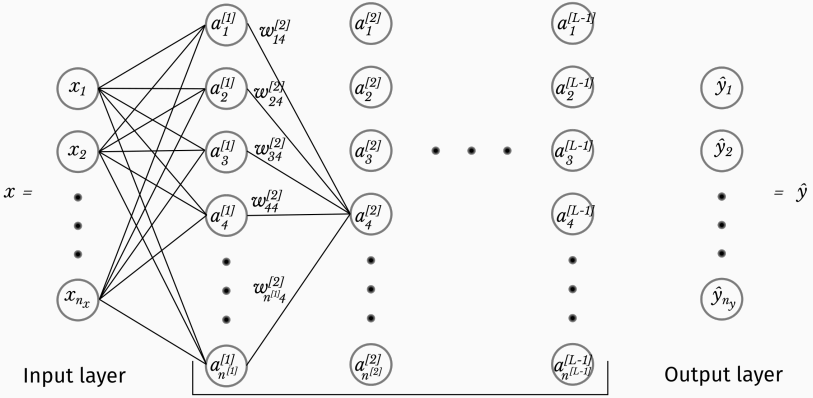
- Superscript with square brackets $[l]$: layer l
- L: Number of layers in the network.
- $n^{[l]}$: Number of nodes in layer l
- $n_x = n^{[0]}$: Input dimension
- $n_y = n^{[L]}$: Output dimension (number of classes)
- x, X, $\mathcal{X}$: Arrays representing input
- y, Y, $\mathcal{Y}$: Arrays representing *true* output
- $\tilde{y}$, $\tilde{Y}$, $\tilde{\mathcal{Y}}$: Arrays representing *one-hot encoded true* output.
- $\hat{y}$, $\hat{Y}$: Arrays representing *predicted* output
- w, W: Edge weights
- b, B: Node bias
- z, Z: Linear combination of activation in previous layer
- $a^{[l]}$, $A^{[l]}$: Node activation in layer l.
- $a^{[0]} = x$: Input vector
- $a^{[L]} = \hat{y}$: Output vector
- Subscript j or jk: Element in vector, or matrix
- Superscript with parenthesis (i): data example (i)
- Ω_{dataset}: A collection of examples $\{(x^{(i)}, y^{(i)})\}$ constituting a dataset.
- m: Number of examples

ACTIVATION IN NODE 3 OF LAYER 1



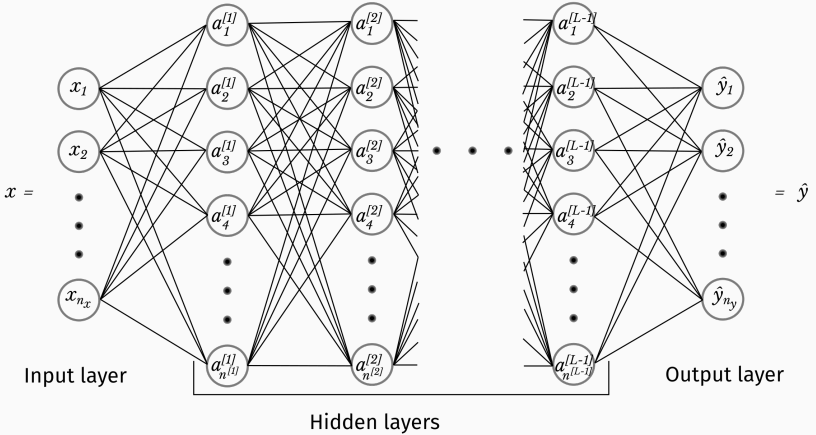
$$a_3^{[1]} = g\left(\sum_{j=1}^{n_x} w_{j3}^{[1]} x_j + b_3^{[1]}\right)$$

ACTIVATION IN NODE 4 OF LAYER 2



$$a_4^{[2]} = g\left(\sum_{j=1}^{n^{[1]}} w_{j4}^{[2]} a_j^{[1]} + b_4^{[2]}\right)$$

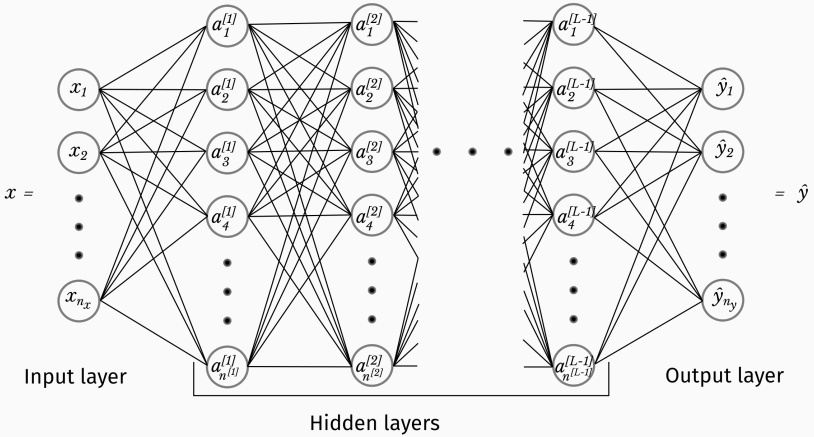
ALL CONNECTIONS



$$a_k^{[l]} = g\left(\sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}\right), \quad k \in \{1, 2, \ldots, n^{[l]}\}, \quad l \in \{1, 2, \ldots, L\}$$

FORWARD PROPAGATION

NETWORK ARCHITECTURE OVERVIEW



INPUT LAYER TO FIRST HIDDEN LAYER

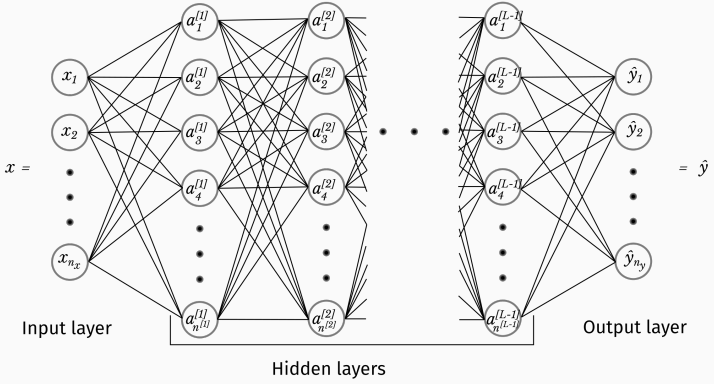
$$z_k^{[1]} = \sum_{j=1}^{n_x} w_{jk}^{[1]} x_j + b_k^{[1]}$$

$$= \sum_{j=1}^{n^{[0]}} w_{jk}^{[1]} a_j^{[0]} + b_k^{[1]}$$

$$a_k^{[1]} = g(z_k^{[1]})$$

for

$$k = 1, \ldots, n^{[1]}.$$



BETWEEN TWO HIDDEN LAYERS

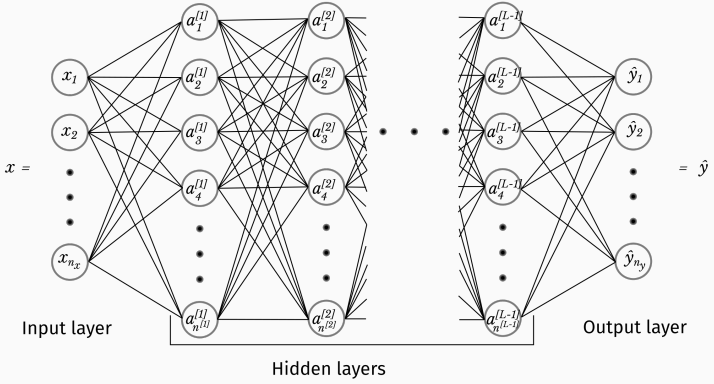
$$z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}$$

$$a_k^{[l]} = g(z_k^{[l]})$$

for

$$k = 1, \ldots, n^{[l]},$$

$$l = 1, \ldots, L-1$$



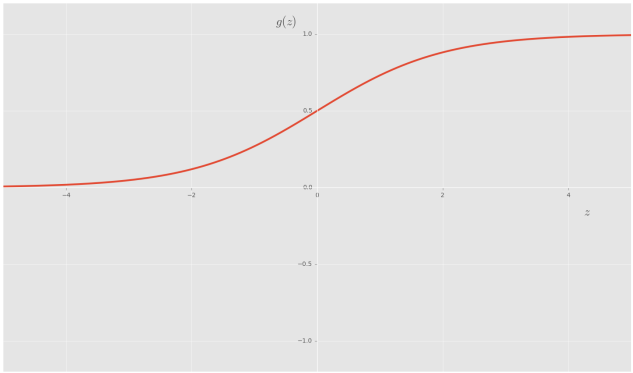
ACTIVATION FUNCTIONS

- Functions that introduce non-linearity to our network
- Without it, our network just becomes a linear mapping from input to output
- Enables DNN to become *universal function approximators*
- Can in theory be any function that is
  - Non-linear
  - Differentiable (if you are using a gradient-based optimization)

SIGMOID ACTIVATION

$$
\begin{aligned}
g(z) &= \sigma(z) \\
&= \frac{1}{1 + e^{-z}} \\
&= \frac{e^z}{e^z + 1}.
\end{aligned}
$$

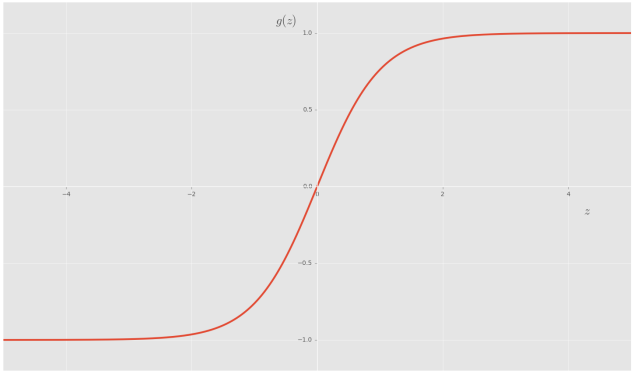
- Stems from logistic regression, and was used a lot historically
- Several problems
- Very flat slope except at z close to zero
- This can result in slow learning (*vanishing gradient problem*)
- Not centered around zero



HYPERBOLIC TANGENT ACTIVATION

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

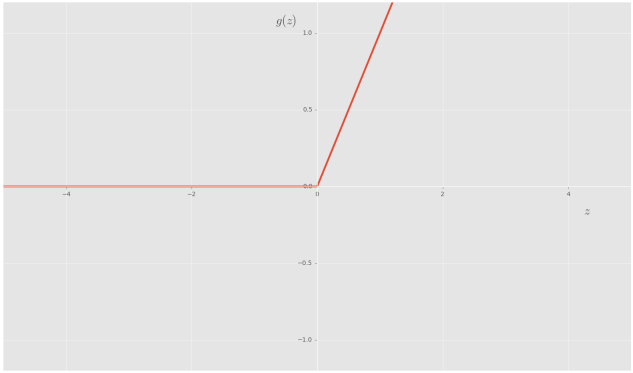
- Similar to the sigmoid function, but centered around zero
- Often performs a bit better
- Still suffers from vanishing gradients



RECTIFIED LINEAR UNIT ACTIVATION

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

- Currently the most popular choice
- Faster convergence
- Not without problems: e.g. “dead neurons”
- Extensions:
  - Leaky ReLU
  - ELU: Exponential linear unit
  - SELU: Scaled exponential linear unit



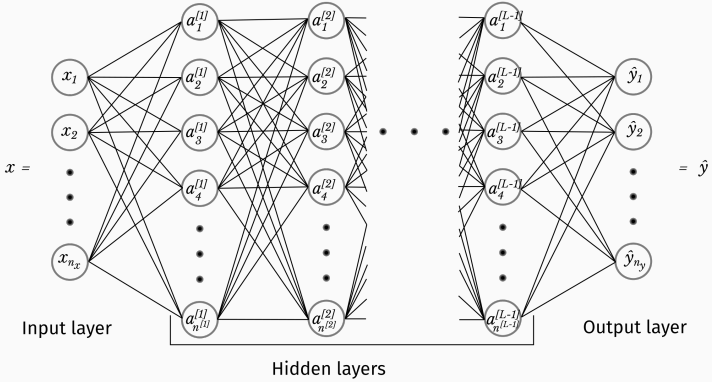
OUTPUT LAYER

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]}$$

$$a_k^{[L]} = s(z_k^{[L]})$$
$$= \hat{y}_k$$

for

$$k = 1, \ldots, n_y,$$
$$= 1, \ldots, n^{[L]}.$$



$$s(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{n} e^{z_i}}$$

- $\sum_k s(z)_k = 1$, and the softmax can be interpreted as a probability
- Using the softmax as our final activation, we can interpret the output of our network as

$$f(x;\Theta)_k = \Pr(\mathcal{Y} = k | \mathcal{X} = x; \Theta) \qquad (1)$$

- $\mathcal{X}$ is a random vector modeling our input
- $\mathcal{Y}$ is a categorical random variable modeling the true output
- Θ is the collection of parameters

$$\Theta = \{w_{jk}^{[l]}, b_k^{[l]}\}$$

for

$$\begin{cases} j & = 1, \ldots, n^{[l-1]} \\ k & = 1, \ldots, n^{[l]} \\ l & = 1, \ldots, L \end{cases}$$

IMPLEMENTING THE SOFTMAX FUNCTION

- Numerical instability can be a problem, because of the exponential function, and division.
- Two common “tricks” that can help this follows
- Shift exponential arguments to max zero

$$
\begin{aligned}
s(z)_k &= \frac{e^{z_k}}{\sum_{i=1}^{n} e^{z_i}} \\
&= \frac{e^{z_k - \max(z)}}{\sum_{i=1}^{n} e^{z_i - \max(z)}}
\end{aligned}
$$

- Take logarithm and exponentiate it to get rid of division

$$
\begin{aligned}
t(z)_k &= \log s(z)_k \\
&= z_k - \log \sum_{i=1}^{n} e^{z_i} \\
s(z)_k &= e^{t(z)_k}
\end{aligned}
$$

- The above can be combined

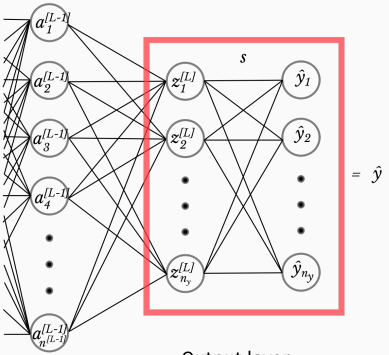
In the output layer we have

$$
\begin{aligned}
z_k^{[L]} &= \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]} \\
a_k^{[L]} &= s(z_k^{[L]}) \\
&= \hat{y}_k
\end{aligned}
$$

for

$$
\begin{aligned}
k &= 1, \ldots, n_y, \\
&= 1, \ldots, n^{[L]}.
\end{aligned}
$$

$z^{[L]}$ are called *logits*, and $\hat{y}$ will be predicted output probabilities.



CROSS ENTROPY COST FUNCTION

- We have defined the network architecture
- Now, we need to select values for the parameters

$$\Theta = \{w_{jk}^{[l]}, b_k^{[l]} : j \in \{1, \dots, n^{[l-1]}\}, k \in \{1, \dots, n^{[l]}\}, l \in \{1, \dots, L\}\}$$

- Several possible ways of doing this
- Vanilla classification with deep learning with dense neural networks:
- Minimizing the *cross entropy* cost function using a *stochastic gradient descent* optimizer
- Will first derive the cost function using a *maximum likelihood estimation*
- In the next part, we will discuss how to actually compute the estimator values

- Suppose we have a random variable X which distribution is described by $p_X(x;\theta)$
- We want to estimate the deterministic, but unknown parameter θ.
- The maximum likelihood estimator (MLE) $\hat{\theta}$ of θ is the parameter

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta; x) \tag{2}$$

- The *likelihood* $\ell(\theta; x)$ has the same form as $p_X(x;\theta)$, except x is a fixed realization of X, and θ is a variable.
- Interpreted as describing the probability of observing $X = x$ for various values of θ (and is therefore a function of θ)

CATEGORICAL DISTRIBUTION OF OUTPUT CONDITIONED ON INPUT

- Let $\mathcal{Y}$ be a categorical random variable modeling our true class.
- Let $\mathcal{X}$ be a random vector modeling the corresponding input.
- $\mathcal{Y}$ can only take one of K values, and conditioned on $\mathcal{X} = x$, is distributed according to a *categorical distribution*

$$p_{\mathcal{Y}}(y|\mathcal{X} = x; \Theta) = \prod_{k=1}^{K} \Pr(\mathcal{Y} = k|\mathcal{X} = x; \Theta)^{[y=k]} \tag{3}$$

- $[y = k]$ is the *Iverson bracket*

$$[y = k] = \begin{cases} 1, & \text{if} \quad y = k \\ 0, & \text{else} \end{cases}$$

- We defined the probability in eq. (1)

$$\Pr(\mathcal{Y} = k|\mathcal{X} = x; \Theta) = \hat{y}(x; \Theta)_k$$

- It is common to represent the true output y as a so-called *one-hot encoded* vector $\tilde{y}$ with elements

$$\tilde{y}_k = \begin{cases} 1, & \text{if} \quad y = k \\ 0, & \text{else} \end{cases} \tag{4}$$

- Inserting this, and eq. (1) into eq. (3) yields

$$p_{\mathcal{Y}}(y|\mathcal{X} = x; \Theta) = \prod_{k=1}^{n_y} \hat{y}(x; \Theta)_k^{\tilde{y}_k} \tag{5}$$

DISTRIBUTION OF MULTIPLE INPUTS

- We can extend eq. (5) to a case with m examples
- Let the random variables $\mathcal{Y}_i$ and $\mathcal{X}_i$ model our output and input for examples $i = 1, \ldots, m$
- The joint, conditional distribution for this collection is then

$$p_{\mathcal{Y}_1,\ldots,\mathcal{Y}_m}(y_1, \ldots, y_m | \mathcal{X}_1 = x_1, \ldots, \mathcal{X}_m = x_m; \Theta) \overset{\text{i.i.d.}}{=} \prod_{i=1}^{m} p_{\mathcal{Y}_i}(y_i | \mathcal{X}_i = x_i; \Theta) \qquad (6)$$

- Here i.i.d. stands for *independent and identically distributed*
- This means that we assume that $(\mathcal{Y}_i, \mathcal{X}_i)$ is independent of $(\mathcal{Y}_j, \mathcal{X}_j)$ when $i \neq j$, but follows the exact same distribution
- This can be shown using the so-called chain rule of conditional probability

$$\begin{aligned} p(y_1, \ldots, y_m | x_1, \ldots, x_m) &= \frac{p(y_1, \ldots, y_m, x_1, \ldots, x_m)}{p(x_1, \ldots, x_m)} \\ &\overset{\text{i.i.d.}}{=} \frac{p(y_1, x_1) p(y_2, x_2) \ldots p(y_m, x_m)}{p(x_1) p(x_2) \ldots p(x_m)} \\ &= p(y_1 | x_1) p(y_2 | x_2) \ldots p(y_m | x_m). \end{aligned}$$

- In our case, the likelihood function becomes

$$\begin{aligned}\ell(\Theta; \Omega^y_{\text{train}} | \Omega^x_{\text{train}}) &= \prod_{i=1}^{m} \ell(\Theta; y^{(i)} | x^{(i)}), \\ &= \prod_{i=1}^{m} \prod_{k=1}^{n_y} \left(\hat{y}_k^{(i)} \right)^{\tilde{y}_k^{(i)}}.\end{aligned} \tag{7}$$

- Where Ω^y_{train} is our training example outputs $\{y^{(1)}, \ldots, y^{(m)}\}$ and Ω^x_{train} is our training example inputs $\{x^{(1)}, \ldots, x^{(m)}\}$

- We are interested in estimating Θ using the MLE $\hat{\Theta}$

$$\hat{\Theta} = \arg\max_{\Theta} \{\ell(\Theta; \Omega^y_{\text{train}} | \Omega^x_{\text{train}})\} \tag{8a}$$

$$= \arg\max_{\Theta} \{\log \ell(\Theta; \Omega^y_{\text{train}} | \Omega^x_{\text{train}})\} \tag{8b}$$

$$= \arg\min_{\Theta} \{-\log \ell(\Theta; \Omega^y_{\text{train}} | \Omega^x_{\text{train}})\} \tag{8c}$$

$$= \arg\min_{\Theta} \left\{-\frac{1}{m} \ell(\Theta; \Omega^y_{\text{train}} | \Omega^x_{\text{train}})\right\} \tag{8d}$$

- Eq. (8a): Definition of the maximum likelihood estimator.
- Eq. (8b): Maximizing the log-likelihood is more numerically stable.
- Eq. (8c): Minimizing the negative likelihood is equivalent.
- Eq. (8d): Makes it "invariant" to the number of examples.
- Eq. (8c) and Eq. (8d): Necessary in order to connect it to information-theoretical interpretation.

- Using the operations in eqs. (8) on our likelihood, defined in eq. (7), we get

$$\hat{\Theta} = \arg\min_{\Theta} \left\{ -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)} \right\}.$$

- Needs to be found with numerical optimization
- The optimization objective function will therefore be the *cross entropy cost*

$$\mathcal{C}(\Theta, \Omega^y_{\text{train}}, \Omega^x_{\text{train}}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}. \tag{9}$$

- Also common to term this as a *loss function*.
- We will distinguish between *cost* function and *loss* function
- We will reserve loss function to the discrepancy between the predicted and true output for a single example
- Our *cross entropy loss* is then

$$\mathcal{L}(y^{(i)}, \hat{y}^{(i)}) = -\sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}. \tag{10}$$

CROSS ENTROPY FROM INFORMATION THEORY

- We have derived the cross entropy loss from a probabilistic maximum likelihood framework
- In information theory, the cross entropy metric is also known my the name *relative entropy*
- We can also arrive to the cross entropy loss using a distance metric called the *Kullbach-Leibler divergence*

DISTANCE METRIC: KULLBACK-LEIBLER DIVERGENCE

- The Kullback-Liebler divergence over a discrete random variable $\mathcal{X}$

$$D_{KL}(p_{\mathcal{X}}||q_{\mathcal{X}}) = \sum_x p_{\mathcal{X}}(x) \log \frac{p_{\mathcal{X}}(x)}{q_{\mathcal{X}}(x)} \tag{11}$$

- Measures the distance between two probability distributions $p_{\mathcal{X}}$ and $q_{\mathcal{X}}$ over the same set of events, modeled with the random variable $\mathcal{X}$.
- Expectation of logarithmic difference between p and q when expectation is taken w.r.t. p.
- Measures the amount of information that is lost when using q to approximate p.
- It is non-negative
- Zero for $p = q$
- Increasing for "increasing difference" between p and q.

CROSS ENTROPY FROM KL DIVERGENCE

- Let p_{model} be the *model distribution* defined in eq. (1)
- Let p_{data} be the *empirical data distribution* defined by our data (using the one-hot encoding from eq. (4)).
- The Kullback-Liebler divergence between the two is then

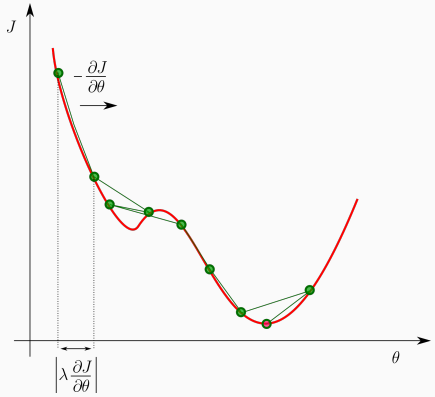
$$\begin{aligned} D_{KL}(p_{\text{model}}||p_{\text{data}}) &= \sum_k \tilde{y}_k \log \frac{\tilde{y}_k}{\hat{y}_k} \\ &= -\sum_k \tilde{y}_k \log \hat{y}_k \end{aligned}$$

  which we recognise as the cross entropy loss.
- In the last step, we used that
  - $\tilde{y}_k \log \tilde{y}_k \to 0$ when $\tilde{y}_k \to 0$ and therefore we set it to zero even though $\log 0$ is undefined
  - Also $\tilde{y}_k \log \tilde{y}_k = 0$ when $\tilde{y}_k = 1$.

OPTIMIZATION

$$\theta \leftarrow \theta - \lambda \frac{\partial J}{\partial \theta}(\theta) \tag{12}$$

- Where
  - J is some objective function that is to be optimized
  - θ is the parameter that is to be updated
  - λ is the step length (often called learning rate in machine learning environments)
- $\frac{\partial J}{\partial \theta}(\theta_k)$ gives the direction ($+$ or $-$) of steepest ascent at the point θ_k
- λ controls how long to move in that direction

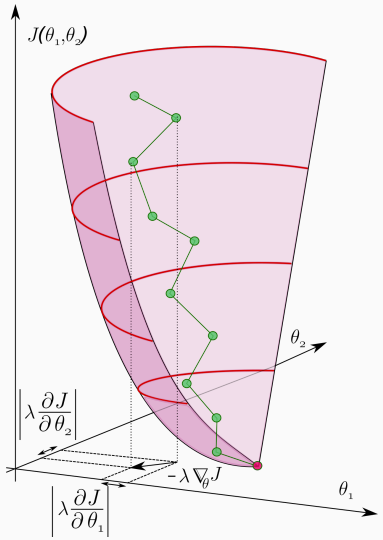


$$\theta \leftarrow \theta - \lambda \nabla_\theta J(\theta) \tag{13}$$

- The gradient of J w.r.t. a set of variables $\theta = [\theta_1, \ldots, \theta_m]$

$$\nabla_\theta J = \left[\frac{\partial J}{\partial \theta_1}, \ldots, \frac{\partial J}{\partial \theta_m} \right]$$

- $\nabla_\theta J(\theta_k)$ gives the direction ($+$ or $-$ of every element in θ) of steepest ascent at the point θ_k
- λ determines how long to move in that direction



Gradient descent (steepest descent) optimization

- This is the simplest, most naive, gradient-based optimization method.
- Turns out to fit very well with deep learning
  - Very fast per update
  - Traverses the parameter space fairly well
  - Not so dependent on initialisations
- Not completely understood why it works well (still trying to figure out the topology of the solution space: saddle points vs. local minima, etc.)
- Traditional problems
  - Oscillations around (local) minima
  - Slow convergence

We want to find values for our weights and biases

$$w_{jk}^{[l]} \leftarrow w_{jk}^{[l]} - \lambda \frac{\partial \mathcal{C}}{\partial w_{jk}^{[l]}} \tag{14}$$

$$b_{k}^{[l]} \leftarrow b_{k}^{[l]} - \lambda \frac{\partial \mathcal{C}}{\partial b_{k}^{[l]}} \tag{15}$$

for all

$$\begin{cases} j & = 1, \ldots, n^{[l-1]} \\ k & = 1, \ldots, n^{[l]} \\ l & = 1, \ldots, L \end{cases}$$

This is done with the so-called *backpropagation algorithm*.

STOCHASTIC LEARNING

- Expencive to consider the full training set at each update
- Instead: consider only a randomly sampled subset; a *mini-batch* of size m_b
- This is called *stochastic gradient descent*
- This approximates the actual step direction fairly well
- Size
  - Too small: Poor approximation. Inneficient because of bad linalg library utilization
  - Too large: Better approximation. Inneficient because of many samples.
  - Recommended: Order of one to a couple of hundred (problem dependent)
  - Usually a power of 2 (can be more efficient because of memory layout on computers)

$$\mathcal{C}_b = \frac{1}{m_b} \sum_{i=1}^{m_b} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}$$

$$\theta \leftarrow \theta - \lambda \nabla_\theta \mathcal{C}_b$$

BACKWARD PROPAGATION

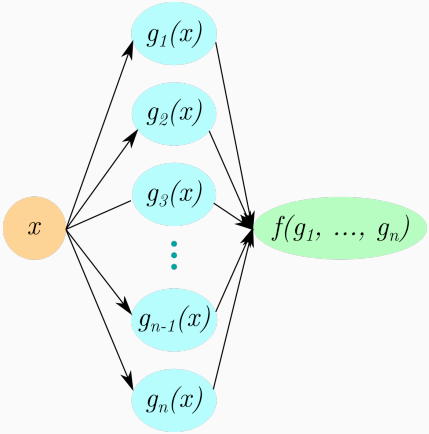
- We are going to update all $w_{jk}^{[l]}$ and $b_k^{[l]}$
- This is done by minimizing the cross entropy loss using a gradient descent optimizer
- Therefore, we need to compute all $\frac{\partial \mathcal{C}}{\partial w_{jk}^{[l]}}$ and $\frac{\partial \mathcal{C}}{\partial b_k^{[l]}}$.
- This can be done with repeated recursive use of the chain rule

For a function f dependent on g which is dependent on x

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}$$

For a function f dependent on multiple $g_1, \ldots, g_n$, all which are dependent on x

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$$

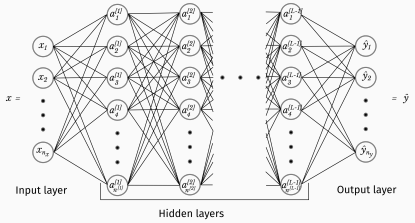


- The cost is just an average over all losses, so we are going to derive the partial derivatives from a single example.
- We start from the output layer and move backwards through the network
- First, we compute the derivative of the loss w.r.t. the linear combinations

$$\frac{\partial \mathcal{L}}{\partial z_k^{[l]}}, \quad k = 1, \ldots, n^{[l]}, \quad l = 1, \ldots, L$$

- Then, we use this to derive all

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} \text{ and } \frac{\partial \mathcal{L}}{\partial b_k^{[l]}}.$$



- We have the expression for the loss and the softmax

$$\mathcal{L}(y, a^{[L]}) = -\sum_{k=1}^{n_y} \tilde{y}_k \log a_k^{[L]}$$

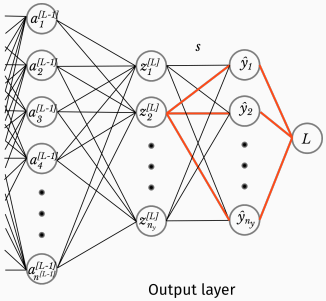
$$a_k^{[L]} = \frac{e^{z_k}}{\sum_{i=1}^{n} e^{z_i}} = \hat{y}_k$$

- As we see, every $z_k^{[L]}$ is involved in all $a_k^{[L]}$, therefore we get

$$\frac{\partial \mathcal{L}}{\partial z_k^{[L]}} = \sum_{j=1}^{n_y} \frac{\partial \mathcal{L}}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_k^{[L]}}. \tag{16}$$

- The first factor is pretty straight forward

$$\frac{\partial \mathcal{L}}{\partial a_j^{[L]}} = -\frac{\tilde{y}_j}{a_j^{[L]}}. \tag{17}$$



The second factor is a bit more involved. First we compute it when $j = k$

$$\begin{aligned}
\frac{\partial a_k^{[L]}}{\partial z_k^{[L]}} &= \frac{\frac{\partial}{\partial z_k^{[L]}}\left(e^{z_k^{[L]}}\right)\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right) - e^{z_k^{[L]}}\frac{\partial}{\partial z_k^{[L]}}\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right)}{\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right)^2} \\
&= \frac{e^{z_k^{[L]}}\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right) - e^{z_k^{[L]}} e^{z_k^{[L]}}}{\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right)^2} \\
&= a_k^{[L]}(1 - a_k^{[L]}).
\end{aligned}$$

When $j \neq k$, we get

$$\begin{aligned}\frac{\partial a_j^{[L]}}{\partial z_k^{[L]}} &= \frac{\frac{\partial}{\partial z_k^{[L]}}\left(e^{z_j^{[L]}}\right)\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right) - e^{z_j^{[L]}}\frac{\partial}{\partial z_k^{[L]}}\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right)}{\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right)^2} \\ &= \frac{-e^{z_j^{[L]}} e^{z_k^{[L]}}}{\left(\sum_{i=1}^{n_y} e^{z_i^{[L]}}\right)^2} \\ &= -a_j^{[L]} a_k^{[L]}.\end{aligned}$$

Combining the results from when $j = k$ and when $j \neq k$, we get, for all nodes $a_j^{[L]}$

$$\frac{\partial a_j^{[L]}}{\partial z_k^{[L]}} = a_j^{[L]}(\delta_{jk} - a_k^{[L]}). \tag{18}$$

where $\delta_{ab} = \begin{cases} 1, & \text{if} \quad a = b \\ 0, & \text{if} \quad a \neq b \end{cases}$ is the *Kronecker delta*.

Inserting eq. (17) and eq. (18) into eq. (16) yields

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z_k^{[L]}} &= \sum_{j=1}^{n_y} \frac{\partial \mathcal{L}}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_k^{[L]}} \\
&= \sum_{j=1}^{n_y} \frac{-\tilde{y}_j}{a_j^{[L]}} a_j^{[L]} (\delta_{jk} - a_k^{[L]}) \\
&= \sum_{j=1}^{n_y} -\tilde{y}_j (\delta_{jk} - a_k^{[L]}) \\
&= a_k^{[L]} \sum_{j=1}^{n_y} \delta_{jk} - \sum_{j=1}^{n_y} \tilde{y}_j \delta_{jk} \\
&= a_k^{[L]} - \tilde{y}_k \\
&= \hat{y}_k - \tilde{y}_k
\end{aligned}
\tag{19}
$$

The first use of the chain rule is straight forward. Since $a_k^{[l]}$ is only dependent on a single $z_k^{[l]}$

$$a_k^{[l]} = g(z_k^{[l]})$$

we get

$$\frac{\partial \mathcal{L}}{\partial z_k^{[l]}} = \frac{\partial \mathcal{L}}{\partial a_k^{[l]}} \frac{\partial a_k^{[l]}}{\partial z_k^{[l]}}.$$

The second factor is dependent on the activation function g, and we will just write

$$\frac{\partial a_k^{[l]}}{\partial z_k^{[l]}} = g'(z_k^{[l]}). \tag{20}$$

Sigmoid function derivative

$$\begin{aligned}\frac{\partial a_k^{[l]}}{\partial z_k^{[l]}} &= \frac{\partial}{\partial z_k^{[l]}}\left(\frac{e^{z_k^{[l]}}}{e^{z_k^{[l]}}+1}\right)\\ &= \frac{e^{z_k^{[l]}}(e^{z_k^{[l]}}+1) - e^{2z_k^{[l]}}}{\left(e^{z_k^{[l]}}+1\right)^2}\\ &= \frac{e^{z_k^{[l]}}}{e^{z_k^{[l]}}+1} - \left(\frac{e^{z_k^{[l]}}}{e^{z_k^{[l]}}+1}\right)^2\\ &= a_k^{[l]}(1-a_k^{[l]}).\end{aligned}$$

Rectified linear unit derivative

$$\begin{aligned}\frac{\partial a_k^{[l]}}{\partial z_k^{[l]}} &= \frac{\partial}{\partial z_k^{[l]}}\left(\max\{0, z_k^{[l]}\}\right)\\ &= \begin{cases}0, & z<0\\ 1, & z>0\\ \text{undefined}, & z=0\end{cases}.\end{aligned}$$

In practice, we just use the *Heaviside step function*

$$H(z) = \begin{cases}0, & z<0\\ 1, & z\geq 0\end{cases}.$$

since we are implementing with floats that are rarely exactly zero

DERIVATIVE W.R.T. Z — BETWEEN HIDDEN LAYERS

To derive the next factor, remember that a single activation in layer l, $a_k^{[l]}$ is connected to every node j in the next layer, by

$$z_j^{[l+1]} = \sum_{i=1}^{n^{[l]}} w_{ij}^{[l+1]} a_i^{[l]} + b_j^{[l+1]}.$$

With this, using the multidimensional chain rule, we get

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial a_k^{[l]}} &= \sum_{j=1}^{n^{[l+1]}} \frac{\partial \mathcal{L}}{\partial z_j^{[l+1]}} \frac{\partial z_j^{[l+1]}}{\partial a_k^{[l]}} \\ &= \sum_{j=1}^{n^{[l+1]}} \frac{\partial \mathcal{L}}{\partial z_j^{[l+1]}} w_{kj}^{[l+1]}.\end{aligned}$$

- Putting the last results together

$$\frac{\partial \mathcal{L}}{\partial z_k^{[l]}} = g'(z_k^{[l]}) \sum_{j=1}^{n^{[l+1]}} \frac{\partial \mathcal{L}}{\partial z_j^{[l+1]}} w_{kj}^{[l+1]}.$$

- Note that this is dependent on the derivative in the next layer, which then again is dependent on derivatives in the next layer.
- At the end, we reach the output layer, and then we can use eq. (19), which terminate the recursion.
- Because of this, we often say that gradients propagate backwards from the ouptut layer through the nodes in the network.

- Remember

$$z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}$$

- This is valid for all layers $l \in \{1, 2, \ldots, L\}$
- The weight derivatives becomes

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} &= \frac{\partial \mathcal{L}}{\partial z_k^{[l]}} \frac{\partial z_k^{[l]}}{\partial w_{jk}^{[l]}} \\ &= \frac{\partial \mathcal{L}}{\partial z_k^{[l]}} a_j^{[l-1]}\end{aligned}$$

- The bias derivatives becomes

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b_k^{[l]}} &= \frac{\partial \mathcal{L}}{\partial z_k^{[l]}} \frac{\partial z_k^{[l]}}{\partial b_k^{[l]}} \\ &= \frac{\partial \mathcal{L}}{\partial z_k^{[l]}}\end{aligned}$$

- Note that $a_j^{[0]} = x_j$, i.e. element j in the input vector
- We already have defined $\frac{\partial \mathcal{L}}{\partial z_k^{[l]}}$ for all layers $l \in \{1, 2, \ldots, L\}$
- We have everything we need.

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_k^{[l]}} a_j^{[l-1]}, \quad l = 1, \ldots, L. \tag{21a}$$

$$\frac{\partial \mathcal{L}}{\partial b_k^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_k^{[l]}}, \quad l = 1, \ldots, L. \tag{21b}$$

$$\frac{\partial \mathcal{L}}{\partial z_k^{[l]}} = g'(z_k^{[l]}) \sum_{j=1}^{n^{[l+1]}} \frac{\partial \mathcal{L}}{\partial z_j^{[l+1]}} w_{kj}^{[l+1]}, \quad l = 1, \ldots, L-1 \tag{21c}$$

$$\frac{\partial \mathcal{L}}{\partial z_k^{[L]}} = \hat{y}_k - \tilde{y}_k. \tag{21d}$$

Note that

- Eqs. (21a)— (21c) are generally applicable
- Eq. (21d) assumes that $\mathcal{L}$ is the cross-entropy loss, and that $a^{[L]} = s(z^{[L]})$ with s as the softmax function.

VECTORISATION

- We have all the equations we need, both forward propagation and backpropagation
- Implementing via for loops is slow (in python)
- Better to represent things as vectors and matrices, and utilise optimised linear algebra libraries (numpy for python)
- We can vectorise over layers
- Can also vectorise over multiple input examples

FORWARD PROPAGATION — VECTORISE OVER A LAYER

- For node k of layer l, we had

$$z_k^{[l]} = \sum_j w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}$$
$$a_k^{[l]} = g(z_k^{[l]}),$$

- Vectorising over the entire layer l yields

$$z^{[l]} = W^{[l]\intercal} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g(z^{[l]}),$$

  where the activation function g is applied element wise
- The dimensions are as follows

$$\begin{cases} z^{[l]} & : n^{[l]} \\ W^{[l]} & : n^{[l-1]} \times n^{[l]} \\ a^{[l]} & : n^{[l]} \\ b^{[l]} & : n^{[l]} \end{cases}.$$

FORWARD PROPAGATION — VECTORISE OVER MULTIPLE EXAMPLES

- Let $X = [x^{(1)}, \ldots, x^{(m)}]$ is a $n_x \times m$ matrix with m input column vectors.
- The forward propagation equations in a layer can be computed for all elements as

$$Z^{[l]} = W^{[l]\intercal} A^{[l-1]} + B^{[l]}$$
$$A^{[l]} = g(Z^{[l]}),$$

- The dimensions are as follows

$$\begin{cases} Z^{[l]} & : n^{[l]} \times m \\ W^{[l]} & : n^{[l-1]} \times n^{[l]} \\ A^{[l]} & : n^{[l]} \times m \\ B^{[l]} & : n^{[l]} \times m \end{cases}.$$

- A single element
  - $Z^{[l]}_{ji}$: Linear combination for node j at layer l for input example i
  - $A^{[l]}_{ji}$: Activation for node j at layer l for input example i
  - $B^{[l]}_{ji}$: Bias value for node j at layer l for input example i. Note that B consist of m equal columns.

COST FUNCTION

- Remember our cost function over m examples

$$\mathcal{C} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n_y} \tilde{y}_k^{(i)} \log \hat{y}_k^{(i)}.$$

- This can be vectorised as

$$\mathcal{C} = -\frac{1}{m} \mathbf{1}(n_y)^\intercal \left(\tilde{Y} \circ \log \hat{Y} \right) \mathbf{1}(m)$$

- Here
  - $\hat{Y}$ and $\tilde{Y}$ are $n_y \times m$ matrices with the predicted output, and one-hot encoded true output, respectively, over m examples.
  - $\mathbf{1}(n)$ is a n-dimensional column vector with ones: $[1, 1, \ldots, 1]$
  - $\circ$ denotes the *Hadamard product* between two arrays of equal dimension

$$(A \circ B)_{ij} = A_{ij} B_{ij}.$$

  - The Hadamard product is also called element wise multiplication

- We will derive the vectorised versions of eqs. (21a) — (21d), one at the time
- First, we introduce the *gradient* and *Jacobian*
- For a function f dependent on $x = [x_1, x_2, \ldots, x_n]$, the gradient of f w.r.t. x is a n-dimensional column vector with elements

$$(\nabla_x f)_i = \frac{\partial f}{\partial x_i}.$$

- For m functions $f = [f_1, f_2, \ldots, f_m]$, all dependent on $x = [x_1, x_2, \ldots, x_n]$, the Jacobian of f w.r.t. x is a $n \times m$ matrix with elements

$$(\mathcal{J}_x(f))_{ij} = \frac{\partial f_j}{\partial x_i}.$$

- Restating eq. (21a)

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = a_j^{[l-1]} \frac{\partial \mathcal{L}}{\partial z_k^{[l]}}$$

- Over one layer, this can be written as

$$\nabla_{W^{[l]}} \mathcal{L} = a^{[l-1]} \nabla_{z^{[l]}} \mathcal{L}^\intercal$$

- For multiple examples, the weight derivative will just be the average over all elements
- This can be seen from our cost function

$$\mathcal{C} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, \hat{y}^{(i)})$$

- For multiple examples, we end up with

$$\nabla_{W^{[l]}} \mathcal{C} = \frac{1}{m} A^{[l-1]} \mathcal{J}_{z^{[l]}}(\mathcal{C})^\intercal$$

- As before, this is valid for all layers $l = 1, \ldots, L$
- Note that we have taken some notational freedom, and defined

$$(\nabla_{W^{[l]}} \mathcal{P})_{ij} = \frac{\partial \mathcal{P}}{\partial w_{ij}^{[l]}}, \quad \text{for } \mathcal{P} \in \{\mathcal{L}, \mathcal{C}\}$$

- We have from eq. (21b)

$$\frac{\partial \mathcal{L}}{\partial b_k^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_k^{[l]}}$$

- Over one layer, with a single example

$$\nabla_{b^{[l]}} \mathcal{L} = \nabla_{z^{[l]}} \mathcal{L}$$

- Over one layer, and multiple examples

$$\nabla_{b^{[l]}} \mathcal{C} = \frac{1}{m} \mathcal{J}_{z^{[l]}}(\mathcal{C}) \mathbf{1}(m)$$

- This is valid for layers $l = 1, \ldots, L$

- We are going to vectorise eq. (21c)

$$\frac{\partial \mathcal{L}}{\partial z_k^{[l]}} = g'(z_k^{[l]}) \sum_{j=1}^{n^{[l+1]}} w_{kj}^{[l+1]} \frac{\partial \mathcal{L}}{\partial z_j^{[l+1]}}$$

- Over one layer l, we get the gradient of the loss w.r.t. z

$$\nabla_{z^{[l]}} \mathcal{L} = g'(z^{[l]}) \circ \left(W^{[l+1]} \nabla_{z^{[l+1]}} \mathcal{L} \right)$$

- Extending to multiple examples is quite straight forward

$$\mathcal{J}_{z^{[l]}}(\mathcal{C}) = g'(Z^{[l]}) \circ \left(W^{[l+1]} \mathcal{J}_{z^{[l+1]}}(\mathcal{C}) \right)$$

- This is valid for layers $l = 1, \ldots, L-1$

- The equation that we are to vectorise is

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial z_k^{[L]}} &= a_k^{[L]} - \tilde{y}_k \\ &= \hat{y}_k - \tilde{y}_k\end{aligned}$$

- For a single example, this becomes

$$\nabla_{z^{[L]}} \mathcal{L} = \hat{y} - \tilde{y}$$

- And for multiple examples, with a one-hot encoded collection of true outputs $\tilde{Y}$

$$\mathcal{J}_{z^{[L]}}(\mathcal{C}) = \hat{Y} - \tilde{Y}$$

BACKWARD PROPAGATION VECTORISED SUMMARY — SINGLE EXAMPLE

$$\nabla_{W^{[l]}}\mathcal{L} = a^{[l-1]}\nabla_{z^{[l]}}\mathcal{L}^{\intercal} \tag{22a}$$

$$\nabla_{b^{[l]}}\mathcal{L} = \nabla_{z^{[l]}}\mathcal{L} \tag{22b}$$

$$\nabla_{z^{[l]}}\mathcal{L} = g'(z^{[l]}) \circ \left(W^{[l+1]}\nabla_{z^{[l+1]}}\mathcal{L}\right) \tag{22c}$$

$$\nabla_{z^{[L]}}\mathcal{L} = \hat{y} - \tilde{y}. \tag{22d}$$

$$\nabla_{W^{[l]}}\mathcal{C} = \frac{1}{m} A^{[l-1]} \mathcal{J}_{z^{[l]}}(\mathcal{C})^{\intercal} \tag{23a}$$

$$\nabla_{b^{[l]}}\mathcal{C} = \frac{1}{m} \left(\mathcal{J}_{z^{[l]}}(\mathcal{C})\right) \mathbf{1}(m) \tag{23b}$$

$$\mathcal{J}_{z^{[l]}}(\mathcal{C}) = g'(Z^{[l]}) \circ \left(W^{[l+1]} \mathcal{J}_{z^{[l+1]}}(\mathcal{C})\right) \tag{23c}$$

$$\mathcal{J}_{z^{[L]}}(\mathcal{C}) = \hat{Y} - \tilde{Y}. \tag{23d}$$

Questions?